MIT Scheme Reference Manual

Edition 1.65
for Scheme Release 7.5
26 February 1999

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

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 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 Scheme User's Manual*.

This chapter summarizes the semantics of Scheme, briefly describes the MIT 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^4 Report on the Algorithmic Language Scheme*, by William Clinger, Jonathan Rees, et al., or by IEEE Std. 1178-1990, *IEEE Standard for the Scheme Programming Language* (in fact, several parts of this document are copied from the *Revised Report*). MIT 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 '$\boxed{\text{error}}$':

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

An example that shows printed output marks it with '$\dashv$':

```
(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 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"), and how the item conforms to standard Scheme, as follows:

category *Category*, with no extra marking, indicates that the item is described in the *Revised^4 Report on the Algorithmic Language Scheme*.

category+ A plus sign after *category* indicates that the item is an MIT Scheme extension.

The form of *template* is interpreted depending on *category*.

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

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 22).

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 21) 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 21) may add a binding to an existing environment.

1.2.3 Initial and Current Environments

MIT 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 24).

Implementation note: In MIT Scheme, `#f` and the empty list are the same object, and the printed representation of `#f` is always '`()`'. As this contradicts the Scheme standard, MIT Scheme will be changed to make `#f` and the empty list different objects.

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 22).

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 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:

- Certain identifiers are reserved for use as syntactic keywords; they should not be used as variables (for a list of the initial syntactic keywords, see Section 1.4.3 [Special Form Syntax], page 13).
- Any identifier that is not a syntactic keyword can be used as a variable.
- 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 Scheme allows all of the identifiers that standard Scheme does, plus many more.

MIT 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 41) 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 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 41, 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 81).

`"` The double quote delimits strings (see Chapter 6 [Strings], page 69).

`\` The backslash is used in the syntax for character constants (see Chapter 5 [Characters], page 61) and as an escape character within string constants (see Chapter 6 [Strings], page 69).

`;` The semicolon starts a comment.

`'` The single quote indicates literal data; it suppresses evaluation (see Section 2.6 [Quoting], page 22).

`` ` `` The backquote indicates almost-constant data (see Section 2.6 [Quoting], page 22).

`,` The comma is used in conjunction with the backquote (see Section 2.6 [Quoting], page 22).

`,@` A comma followed by an at-sign is used in conjunction with the backquote (see Section 2.6 [Quoting], page 22).

`#` 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 103).

`#\` This character sequence introduces a character constant (see Chapter 5 [Characters], page 61).

`#(` This character sequence introduces a vector constant (see Chapter 8 [Vectors], page 95). 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 41).

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

`#*` This character sequence introduces a bit string (see Chapter 9 [Bit Strings], page 99). This notation is an MIT Scheme extension.

`#[` This character sequence is used to denote objects that do not have a readable external representation (see Section 14.7 [Custom Output], page 164). A close bracket, '`]`', terminates the object's notation. This notation is an MIT Scheme extension.

`#@` This character sequence is a convenient shorthand used to refer to objects by their hash number (see Section 14.7 [Custom Output], page 164). This notation is an MIT 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 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 22); 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. The following list contains all of the syntactic keywords that are defined when MIT Scheme is initialized:

```
access               define-syntax        macro
and                  delay                make-environment
begin                do                   named-lambda
case                 fluid-let            or
cond                 if                   quasiquote
cons-stream          in-package           quote
declare              lambda               scode-quote
default-object?      let                  sequence
define               let*                 set!
define-integrable    let-syntax           the-environment
define-macro         letrec               unassigned?
define-structure     local-declare        using-syntax
```

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. Thus you should not use syntactic keywords as procedure names. If you were to bind one of these keywords to a procedure, you would have to use `apply` to call the procedure. MIT Scheme signals an error when such a binding is attempted.

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 expression expression . . .* 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 *expressions* 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 *expressions* in the body. The result of evaluating the last *expression* 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 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 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 three binding constructs `let`, `let*`, and `letrec`, give Scheme block structure. The syntax of the three 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 evaluations and bindings are sequentially interleaved. And in a `letrec` expression, all the bindings are in effect while the initial values are being computed (thus allowing mutually recursive definitions).

let ((*variable init*) ...) *expression expression* ... special form

The *inits* are evaluated in the current environment (in some unspecified order), the *variables* are bound to fresh locations holding the results, the *expressions* are evaluated sequentially in the extended environment, and the value of the last *expression* is returned. Each binding of a *variable* has the *expressions* as its region.

MIT Scheme allows any of the *inits* to be omitted, in which case the corresponding *variables* are unassigned.

Note that the following are equivalent:

```
(let ((variable init) ...) expression expression ...)
((lambda (variable ...) expression expression ...) 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 27, for information on "named `let`".

let* ((*variable init*) ...) *expression expression* ... 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))
   expression
   expression ...)
```

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

An example:

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

letrec ((*variable init*) ...) *expression expression* ... 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 *expressions* are evaluated sequentially in the extended environment, and the value of the last *expression* is returned. Each binding of a *variable* has the entire `letrec` expression as its region, making it possible to define mutually recursive procedures.

MIT 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.

2.3 Dynamic Binding

fluid-let ((*variable init*) . . .) *expression expression* . . . special form+

The *inits* are evaluated in the current environment (in some unspecified order), the current values of the *variables* are saved, the results are assigned to the *variables*, the *expressions* are evaluated sequentially in the current environment, the *variables* 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 Scheme allows any of the *inits* to be omitted, in which case the corresponding *variables* are temporarily unassigned.

An error of type `condition-type:unbound-variable` is signalled if any of the *variables* 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.

Here is an example showing the difference between `fluid-let` and `let`. First see how `let` affects the binding of a variable:

```
(define variable #t)
(define (access-variable) variable)
variable                                ⇒  #t
(let ((variable #f))
  (access-variable))                    ⇒  #t
variable                                ⇒  #t
```

`access-variable` returns `#t` in this case because it is defined in an environment with `variable` bound to `#t`. `fluid-let`, on the other hand, temporarily reuses an existing variable:

```
variable                                ⇒  #t
(fluid-let ((variable #f))              ;reuses old binding
  (access-variable))                    ⇒  #f
variable                                ⇒  #t
```

The *extent* of a dynamic binding is defined to be the time period during which the variable contains the new value. Normally this time period begins when the body is entered and ends when it is exited; on a sequential machine it is normally a contiguous time period. However, because Scheme has first-class continuations, it is possible to leave the body and then reenter it, as many times as desired. In this situation, the extent becomes non-contiguous.

When the body is exited by invoking a continuation, the new value is saved, and the variable is set to the old value. Then, if the body is reentered by invoking a continuation, the old value is saved, and the variable is set to the new value. In addition, side effects to the variable that occur both inside and outside of body are preserved, even if continuations are used to jump in and out of body repeatedly.

Here is a complicated example that shows the interaction between dynamic binding and continuations:

```
(define (complicated-dynamic-binding)
  (let ((variable 1)
        (inside-continuation))
    (write-line variable)
    (call-with-current-continuation
     (lambda (outside-continuation)
       (fluid-let ((variable 2))
         (write-line variable)
         (set! 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
          (set! 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 after the `fluid-let`'s body is entered. Immediately after they are written, `variable` is set to '`3`', and then `outside-continuation` is invoked, causing us to exit the body. At this point, '`1`' is written, demonstrating that the original value of `variable` has been restored, because we have left 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 side effect that previously occurred within the body has been preserved. Finally, we exit body normally, and write '`4`', demonstrating that the side effect that occurred outside of the body was also preserved.

2.4 Definitions

define *variable* [*expression*] special form
define *formals expression expression* . . . 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`, `fluid-let`, 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`, `fluid-let`, 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*] 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 147). 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* 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)                          ⇒  'a
''a                                 ⇒  '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* 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*] 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 ...* 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* ... 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 35) to an *object*, `case` evaluates the *expressions* 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 *expressions* 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* ... special form

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

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

The *expressions* 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
fluid-let
lambda
let
let*
letrec
named-lambda
```

The obsolete special form `sequence` is identical to `begin`. It should not be used in new code.

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*) ...) *expression expression ...* special form

MIT 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 *expressions* to a procedure whose formal arguments are the *variables* and whose body is the *expressions*. Thus the execution of the *expressions* may be repeated by invoking the procedure named by *name*.

MIT Scheme allows any of the *inits* to be omitted, in which case the corresponding *variables* are unassigned.

Note: the following expressions are equivalent:

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

((letrec ((name
           (named-lambda (name variable ...)
             expression
             expression ...)))
   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* ... 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 Scheme, the value of *test* is returned.

The region of the binding of a *variable* consists of the entire `do` expression except for the *inits*. 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 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 108) 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 `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 the same as the name of the structure, e.g. '`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 an *unparser method* (see Section 14.7 [Custom Output], page 164). If the structure instances are records, this option has the same effect as calling `set-record-type-unparser-method!`.

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 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-rtd)) a b)
bar             error  Unbound variable: bar
bar-rtd         ⇒ #[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 unparser method for the structure instances (which can be overridden by the `print-procedure` option). If the default unparser 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 unparser 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
```

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 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 unparser state and the structure instance) 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 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.

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 41).
- *obj1* and *obj2* are both characters and are the same character according to the `char=?` procedure (see Chapter 5 [Characters], page 61).
- 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 42.

With the exception of `inexact->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 41), 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

[1] MIT 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.

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 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
-                 inexact->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 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]

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.

---

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

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

`abs` returns the magnitude of its argument.

```
(abs -7)            ⇒  7
```

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

quotient *n1 n2* procedure
remainder *n1 n2* procedure
modulo *n1 n2* procedure

These procedures implement number-theoretic (integer) division: for positive integers *n1* and *n2*, if *n3* and *n4* are integers such that

$$n_1 = n_2 n_3 + n_4$$
$$0 \leq n_4 < n_2$$

then

```
(quotient n1 n2)           ⇒  n3
(remainder n1 n2)          ⇒  n4
(modulo n1 n2)             ⇒  n4
```

For integers *n1* and *n2* with *n2* not equal to 0,

```
(= n1
   (+ (* n2 (quotient n1 n2))
      (remainder n1 n2)))
                                          ⇒  #t
```

provided all numbers involved in that computation are exact.

The value returned by `quotient` always has the sign of the product of its arguments. `remainder` and `modulo` differ on negative arguments — the `remainder` always has the sign of the dividend, the `modulo` always has the sign of the divisor:

```
(modulo 13 4)              ⇒  1
(remainder 13 4)           ⇒  1

(modulo -13 4)             ⇒  3
(remainder -13 4)          ⇒  -1

(modulo 13 -4)             ⇒  -3
(remainder 13 -4)          ⇒  1

(modulo -13 -4)            ⇒  -1
(remainder -13 -4)         ⇒  -1

(remainder -13 -4.0)       ⇒  -1.0   ; inexact
```

Note that `quotient` is the same as `integer-truncate`.

integer-floor *n1 n2* procedure+
integer-ceiling *n1 n2* procedure+
integer-truncate *n1 n2* procedure+
integer-round *n1 n2* procedure+

These procedures combine integer division with rounding. For example, the following are equivalent:

```
(integer-floor n1 n2)
(floor (/ n1 n2))
```

However, the former is faster and does not produce an intermediate result.

Note that `integer-truncate` is the same as `quotient`.

integer-divide *n1 n2* procedure+
integer-divide-quotient *qr* procedure+
integer-divide-remainder *qr* procedure+

`integer-divide` is equivalent to performing both `quotient` and `remainder` at once. The result of `integer-divide` is an object with two components; the procedures `integer-divide-quotient` and `integer-divide-remainder` select those components. These procedures are useful when both the quotient and remainder are needed; often computing both of these numbers simultaneously is much faster than computing them separately.

For example, the following are equivalent:

```
(lambda (n d)
  (cons (quotient n d)
        (remainder n d)))

(lambda (n d)
  (let ((qr (integer-divide n d)))
    (cons (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
```

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 (exact->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 `inexact->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)
(inexact->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 (inexact->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

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

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.

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.

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.

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

$$z = x_1 + x_2 i = x_3 \cdot e^{ix_4}$$

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*.

exact->inexact *z* procedure
inexact->exact *z* procedure

`exact->inexact` returns an inexact representation of *z*. The value returned is the inexact number that is numerically closest to the argument. If an exact argument has no reasonably close inexact equivalent, then a violation of an implementation restriction may be reported; MIT Scheme signals an error of type `condition-type:bad-range-argument` in this case.

`inexact->exact` returns an exact representation of *z*. The value returned is the exact number that is numerically closest to the argument. If an inexact argument has no reasonably close exact equivalent, then a violation of an implementation restriction may be reported; in MIT Scheme this case does not occur because all inexact numbers are representable as exact numbers.

These procedures implement the natural one-to-one correspondence between exact and inexact integers throughout an implementation-dependent range. See Section 4.3 [Implementation restrictions], page 42.

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-unparser-cutoff variable+

This variable controls the action of `number->string` when *number* is a flonum (and consequently controls all printing of flonums). The value of this variable 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.xxxeyyy`. In other words, the number is printed as a mantissa 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 mantissa 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 double-precision 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 `flonum-unparser-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 `flonum-unparser-cutoff`:

```
(number->string (* 4 (atan 1 1)))
                                  ⇒ "3.141592653589793"
(fluid-let ((flonum-unparser-cutoff '(relative 5)))
  (number->string (* 4 (atan 1 1))))
                                  ⇒ "3.1416"
(fluid-let ((flonum-unparser-cutoff '(relative 5)))
  (number->string (* 4000 (atan 1 1))))
                                  ⇒ "3141.6"
(fluid-let ((flonum-unparser-cutoff '(relative 5 scientific)))
  (number->string (* 4000 (atan 1 1))))
                                  ⇒ "3.1416e3"
(fluid-let ((flonum-unparser-cutoff '(relative 5 scientific)))
  (number->string (* 40000 (atan 1 1))))
                                  ⇒ "3.1416e4"
(fluid-let ((flonum-unparser-cutoff '(relative 5 engineering)))
  (number->string (* 40000 (atan 1 1))))
                                  ⇒ "31.416e3"
(fluid-let ((flonum-unparser-cutoff '(absolute 5)))
  (number->string (* 4 (atan 1 1))))
                                  ⇒ "3.14159"
(fluid-let ((flonum-unparser-cutoff '(absolute 5)))
  (number->string (* 4000 (atan 1 1))))
                                  ⇒ "3141.59265"
(fluid-let ((flonum-unparser-cutoff '(absolute -4)))
  (number->string (* 4e10 (atan 1 1))))
                                  ⇒ "31415930000."
(fluid-let ((flonum-unparser-cutoff '(absolute -4 scientific)))
  (number->string (* 4e10 (atan 1 1))))
                                  ⇒ "3.141593e10"
(fluid-let ((flonum-unparser-cutoff '(absolute -4 engineering)))
```

```
  (number->string (* 4e10 (atan 1 1))))
                                  ⇒ "31.41593e9"
(fluid-let ((flonum-unparser-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 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.

4.7.1 Fixnum Operations

A *fixnum* is an exact integer that is small enough to fit in a machine word. In MIT 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 logically shifting *fixnum1* by *fixnum2* bits. If *fixnum2* is positive, *fixnum1* is shifted left; if negative, it is shifted right. 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)                    ⇒  #x3FFFF8
```

4.7.2 Flonum Operations

A *flonum* is an inexact real number that is implemented as a floating-point number. In MIT Scheme, all inexact real numbers are flonums. For this reason, constants such as `0.` and `2.3` are guaranteed to be flonums.

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+

These procedures are the standard order and equality predicates on flonums. When compiled, they do not check the types of their arguments.

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.

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:negate *flonum* procedure+

This procedure returns the negation of its argument. When compiled, it does not check the type of its argument. Equivalent to `(flo:- 0. flonum)`.

flo:abs *flonum* procedure+
flo:exp *flonum* procedure+
flo:log *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:sqrt *flonum* procedure+
flo:expt *flonum1 flonum2* 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+

This is the flonum version of `atan` with two arguments. When compiled, it does not check the types of its arguments.

4.8 Random Numbers

MIT Scheme provides a facility for generating pseudo-random numbers. The current implementation is a "subtract-with-carry" random-number generator, based on the algorithm from *A New Class of Random Number Generators*, George Marsaglia and Arif Zaman, *The Annals of Applied Probability*, Vol. 1, No. 3, 1991. At the time it was implemented, this was a good algorithm for general purposes, but the state of the art in random-number generation is constantly changing. If necessary, the implementation will be updated to use a new algorithm while retaining the same interface.

The interface described here is very similar to that of Common Lisp.

random *modulus* [*state*] procedure+

Modulus must be a positive real number. `random` returns a pseudo-random number between zero (inclusive) and *modulus* (exclusive). The exactness of the returned number is the same as the exactness of *modulus*. Additionally, if *modulus* is an exact integer, the returned number will be also. Usually, *modulus* is either an exact integer or an inexact real; the current implementation has been tuned to make these two cases fast.

If *state* is given and not `#f`, it must be a random-state object; otherwise, it defaults to the value of the variable `*random-state*`. This object is used to maintain the state of the pseudo-random-number generator and is altered as a side effect of the `random` procedure.

```
(random 1.0)    ⇒ .32744744667719056
(random 1.0)    ⇒ .01668326768172354
(random 10)     ⇒ 3
(random 10)     ⇒ 8
(random 100)    ⇒ 38
(random 100)    ⇒ 63
(random 100/3)  ⇒ 130501475769920525/6755399441055744
(random 100/3)  ⇒ 170571694016427575/13510798882111488
```

flo:random-unit *state* procedure+

State must be a random-state object. `flo:random-unit` returns a pseudo-random number between zero inclusive and one exclusive; the returned number is always a flonum and therefore an inexact real number. `flo:random-unit` is equivalent to `random` with a *modulus* of `1.0`, except that it is faster.

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 pseudo-random sequence.

random-state variable+

This variable holds a data structure, a random-state object, that encodes the internal state of the random-number generator that `random` uses by default. A call to `random` will perform a side effect on this data structure. This variable may be changed, using `set!` or `fluid-let`, to hold a new random-state object.

make-random-state [*state*] procedure+

This procedure returns a new random-state object, suitable for use as the value of the variable `*random-state*`, or as the *state* argument to `random`. If *state* is not given or `#f`, `make-random-state` returns a *copy* of the current random-number state object (the value of the variable `*random-state*`). 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`.

5 Characters

Characters are objects that represent printed characters, such as letters and digits.[1]

5.1 External Representation of Characters

Characters are written using the notation `#\`*character* or `#\`*character-name*. For example:

```
#\a                    ; lowercase letter
#\A                    ; uppercase letter
#\(                    ; left parenthesis
#\space                ; the space character
#\newline              ; the newline character
```

Case is significant in `#\`*character*, but not in `#\`*character-name*. If *character* in `#\`*character* is a letter, *character* must be followed by a delimiter character such as a space or parenthesis. Characters written in the `#\` notation are self-evaluating; you don't need to quote them.

A character name may include one or more *bucky bit* prefixes to indicate that the character includes one or more of the keyboard shift keys Control, Meta, Super, Hyper, or Top (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
Top             T- or Top-                 16
```

For example,

```
#\c-a                  ; Control-a
#\meta-b               ; Meta-b
#\c-s-m-h-a            ; Control-Meta-Super-Hyper-A
```

The following *character-names* are supported, shown here with their ASCII equivalents:

---

[1] Some of the details in this section depend on the fact that the underlying operating system uses the ASCII character set. This may change when someone ports MIT Scheme to a non-ASCII operating system.

```
Character Name         ASCII Name
--------------         ----------

altmode                ESC
backnext               US
backspace              BS
call                   SUB
linefeed               LF
page                   FF
return                 CR
rubout                 DEL
space
tab                    HT
```

In addition, `#\newline` is the same as `#\linefeed` (but this may change in the future, so you should not depend on it). All of the standard ASCII names for non-printing characters are supported:

```
NUL     SOH     STX     ETX     EOT     ENQ     ACK     BEL
BS      HT      LF      VT      FF      CR      SO      SI
DLE     DC1     DC2     DC3     DC4     NAK     SYN     ETB
CAN     EM      SUB     ESC     FS      GS      RS      US
DEL
```

char->name *char* [*slashify?*] procedure+

Returns a string corresponding to the printed representation of *char*. This is the *character* or *character-name* 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"
```

Slashify?, if specified and true, says to insert the necessary backslash characters in the result so that `read` will parse it correctly. In other words, the following generates the external representation of *char*:

```
(string-append "#\\" (char->name char #t))
```

If *slashify?* is not specified, it defaults to `#f`.

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 "c-a")                  ⇒  #\C-a
(name->char "control-a")            ⇒  #\C-a
```

5.2 Comparison of Characters

char=? *char1 char2* procedure
char<? *char1 char2* procedure
char>? *char1 char2* procedure
char<=? *char1 char2* procedure
char>=? *char1 char2* procedure
char-ci=? *char1 char2* procedure
char-ci<? *char1 char2* procedure
char-ci>? *char1 char2* procedure
char-ci<=? *char1 char2* procedure
char-ci>=? *char1 char2* procedure

Returns `#t` if the specified characters are have the appropriate order relationship to one another; otherwise returns `#f`. The `-ci` procedures don't distinguish uppercase and lowercase letters.

Character ordering follows these rules:

- The digits are in order; for example, `(char<? #\0 #\9)` returns `#t`.
- The uppercase characters are in order; for example, `(char<? #\A #\B)` returns `#t`.
- The lowercase characters are in order; for example, `(char<? #\a #\b)` returns `#t`.

In addition, MIT Scheme orders those characters that satisfy `char-standard?` the same way that ASCII does. Specifically, all the digits precede all the uppercase letters, and all the upper-case letters precede all the lowercase letters.

Characters are ordered by first comparing their bucky bits part and then their code part. In particular, characters without bucky bits come before characters with bucky bits.

5.3 Miscellaneous Character Operations

char? *object* procedure

Returns `#t` if *object* is a character; otherwise returns `#f`.

char-upcase *char* procedure
char-downcase *char* procedure

Returns the uppercase or lowercase equivalent of *char* if *char* is a letter; otherwise returns *char*. These procedures return a character *char2* such that `(char-ci=?` *char char2*`)`.

char->digit *char* [*radix*] procedure+

If *char* is a character representing a digit in the given *radix*, returns the corresponding integer value. If you specify *radix* (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*. *Radix* must be an exact integer between 2 and 36 (inclusive), and defaults to 10. *Digit*, which must be an exact non-negative integer, should be less than *radix*; if *digit* is greater than or equal to *radix*, `digit->char` returns `#f`.

```
(digit->char 8)                    ⇒  #\8
(digit->char 14 16)                ⇒  #\E
```

5.4 Internal Representation of Characters

An MIT Scheme character consists of a *code* part and a *bucky bits* part. The MIT Scheme set of characters can represent more characters than ASCII can; it includes characters with Super, Hyper, and Top bucky bits, as well as Control and Meta. Every ASCII character corresponds to some MIT Scheme character, but not vice versa.[2]

MIT Scheme uses a 7-bit ASCII character code with 5 bucky bits. The least significant bucky bit, Meta, is stored adjacent to the MSB of the character code, allowing the least significant 8 bits of a character object to be interpreted as ordinary ASCII with a meta bit. This is compatible with standard practice for 8-bit characters when meta bits are employed.

make-char *code bucky-bits* procedure+

Builds a character from *code* and *bucky-bits*. Both *code* and *bucky-bits* must be exact non-negative integers in the appropriate range. Use `char-code` and `char-bits` to extract the code and bucky bits from the character. If `0` is specified for *bucky-bits*, `make-char` produces an ordinary character; otherwise, the appropriate bits are turned on as follows:

```
1               Meta
2               Control
4               Super
8               Hyper
16              Top
```

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-bits *char* procedure+

Returns the exact integer representation of *char*'s bucky bits. For example,

[2] Note that the Control bucky bit is different from the ASCII control key. This means that `#\SOH` (ASCII ctrl-A) is different from `#\C-A`. In fact, the Control bucky bit is completely orthogonal to the ASCII control key, making possible such characters as `#\C-SOH`.

```
(char-bits #\a)                    ⇒  0
(char-bits #\m-a)                  ⇒  1
(char-bits #\c-a)                  ⇒  2
(char-bits #\c-m-a)                ⇒  3
```

char-code *char* procedure+

Returns the character code of *char*, an exact integer. For example,

```
(char-code #\a)                    ⇒  97
(char-code #\c-a)                  ⇒  97
```

char-code-limit variable+
char-bits-limit variable+

These variables define the (exclusive) upper limits for the character code and bucky bits (respectively). The character code and bucky bits are always exact non-negative integers, and are strictly less than the value of their respective limit variable.

char->integer *char* procedure
integer->char *k* procedure

`char->integer` returns the character code representation for *char*. `integer->char` returns the character whose character code representation is *k*.

In MIT Scheme, if `(char-ascii? char)` is true, then

```
(eqv? (char->ascii char) (char->integer char))
```

However, this behavior is not required by the Scheme standard, and code that depends on it is not portable to other implementations.

These procedures implement order isomorphisms between the set of characters under the `char<=?` ordering and some subset of the integers under the `<=` ordering. That is, if

```
(char<=? a b)  ⇒  #t    and    (<= x y)  ⇒  #t
```

and `x` and `y` are in the range of `char->integer`, then

```
(<= (char->integer a)
    (char->integer b))             ⇒  #t
(char<=? (integer->char x)
         (integer->char y))        ⇒  #t
```

Note: if *char* is a character constant for which `char->integer` returns an integer strictly less than 256, then the compiler will constant-fold the call, replacing it with the corresponding integer. Likewise, if *k* is an integer constant strictly less than 256, the compiler will constant-fold a call to `integer->char`, replacing it with the corresponding character. This is a very useful way to denote unusual character constants or ASCII codes.

char-integer-limit variable+

The range of `char->integer` is defined to be the exact non-negative integers that are less than the value of this variable (exclusive).

5.5 ASCII Characters

MIT Scheme internally uses ASCII codes for I/O, and stores character objects in a fashion that makes it convenient to convert between ASCII codes and characters. Also, character strings are implemented as byte vectors whose elements are ASCII codes; these codes are converted to character objects when accessed. For these reasons it is sometimes desirable to be able to convert between ASCII codes and characters.

Not all characters can be represented as ASCII codes. A character that has an equivalent ASCII representation is called an *ASCII character*.

char-ascii? *char* procedure+

Returns the ASCII code for *char* if *char* has an ASCII representation; otherwise returns `#f`.

In the current implementation, the characters that satisfy this predicate are those in which the Control, Super, Hyper, and Top bucky bits are turned off. All characters for which the `char-bits` procedure returns `0` or `1` (i.e. no bucky bits, or just Meta) count as legal ASCII characters.

char->ascii *char* procedure+

Returns the ASCII code for *char*. An error `condition-type:bad-range-argument` is signalled if *char* doesn't have an ASCII representation.

ascii->char *code* procedure+

Code must be the exact integer representation of an ASCII code. This procedure returns the character corresponding to *code*.

5.6 Character Sets

MIT Scheme's character-set abstraction is used to represent groups of characters, such as the letters or digits. Character sets may contain only ASCII characters; in the future this may be changed to allow the full range of characters.

There is no meaningful external representation for character sets; use `char-set-members` to examine their contents. There is (at present) no specific equivalence predicate for character sets; use `equal?` for this purpose.

char-set? *object* procedure+

Returns `#t` if *object* is a character set; otherwise returns `#f`.[3]

[3] Because character sets are implemented as strings, `string?` returns `#t` for character set objects. However, string operations aren't meaningful with character sets.

char-set:upper-case variable+
char-set:lower-case variable+
char-set:alphabetic variable+
char-set:numeric variable+
char-set:alphanumeric variable+
char-set:whitespace variable+
char-set:not-whitespace variable+
char-set:graphic variable+
char-set:not-graphic variable+
char-set:standard variable+

These variables contain predefined character sets. To see the contents of one of these sets, use `char-set-members`.

Alphabetic characters are the 52 upper and lower case letters. *Numeric* characters are the 10 decimal digits. *Alphanumeric* characters are those in the union of these two sets. *Whitespace* characters are `#\space`, `#\tab`, `#\page`, `#\linefeed`, and `#\return`. *Graphic* characters are the printing characters and `#\space`. *Standard* characters are the printing characters, `#\space`, and `#\newline`. These are the printing characters:

```
! " # $ % & ' ( ) * + , - . /
0 1 2 3 4 5 6 7 8 9
: ; < = > ? @
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
[ \ ] ^ _ `
a b c d e f g h i j k l m n o p q r s t u v w x y z
{ | } ~
```

char-upper-case? *char* procedure
char-lower-case? *char* procedure
char-alphabetic? *char* procedure
char-numeric? *char* procedure
char-alphanumeric? *char* procedure+
char-whitespace? *char* procedure
char-graphic? *char* procedure+
char-standard? *object* procedure+

These predicates are defined in terms of the respective character sets defined above.

char-set-members *char-set* procedure+

Returns a newly allocated list of the characters in *char-set*.

char-set-member? *char-set char* procedure+

Returns `#t` if the *char* is in *char-set*; otherwise returns `#f`.

char-set *char ...* procedure+

Returns a character set consisting of the specified ASCII characters. With no arguments, `char-set` returns an empty character set.

chars->char-set *chars* procedure+

Returns a character set consisting of *chars*, which must be a list of ASCII characters. This is equivalent to `(apply char-set` *chars*`)`.

string->char-set *string* procedure+

Returns a character set consisting of all the characters that occur in *string*.

ascii-range->char-set *lower upper* procedure+

Lower and *upper* must be exact non-negative integers representing ASCII character codes, and *lower* must be less than or equal to *upper*. This procedure creates and returns a new character set consisting of the characters whose ASCII codes are between *lower* (inclusive) and *upper* (exclusive).

predicate->char-set *predicate* procedure+

Predicate must be a procedure of one argument. `predicate->char-set` creates and returns a character set consisting of the ASCII characters for which *predicate* is true.

char-set-difference *char-set1 char-set2* procedure+

Returns a character set consisting of the characters that are in *char-set1* but aren't in *char-set2*.

char-set-intersection *char-set1 char-set2* procedure+

Returns a character set consisting of the characters that are in both *char-set1* and *char-set2*.

char-set-union *char-set1 char-set2* procedure+

Returns a character set consisting of the characters that are in one or both of *char-set1* and *char-set2*.

char-set-invert *char-set* procedure+

Returns a character set consisting of the ASCII characters that are not in *char-set*.

6 Strings

A *string* is a mutable sequence of characters. In the current implementation of MIT Scheme, the elements of a string must all satisfy the predicate `char-ascii?`; if someone ports MIT Scheme to a non-ASCII operating system this requirement will change.

A string is written as a sequence of characters enclosed within double quotes `" "`. To include a double quote inside a string, precede the double quote with a backslash `\` (escape it), as in

```
"The word \"recursion\" has many meanings."
```

The printed representation of this string is

```
The word "recursion" has many meanings.
```

To include a backslash inside a string, precede it with another backslash; for example,

```
"Use #\\Control-q to quit."
```

The printed representation of this string is

```
Use #\Control-q to quit.
```

The effect of a backslash that doesn't precede a double quote or backslash is unspecified in standard Scheme, but MIT Scheme specifies the effect for three other characters: `\t`, `\n`, and `\f`. These escape sequences are respectively translated into the following characters: `#\tab`, `#\newline`, and `#\page`. Finally, a backslash followed by exactly three octal digits is translated into the character whose ASCII code is those digits.

If a string literal is continued from one line to another, the string will contain the newline character (`#\newline`) at the line break. Standard Scheme does not specify what appears in a string literal at a line break.

The *length* of a string is the number of characters that it contains. This number is an exact non-negative integer that is established when the string is created (but see Section 6.9 [Variable-Length Strings], page 78). Each character in a string has an *index*, which is a number that indicates the character's position in the string. The index of the first (leftmost) character in a string is 0, and the index of the last character is one less than the length of the string. The *valid indexes* of a string are the exact non-negative integers less than the length of the string.

A number of the string procedures operate on substrings. A *substring* is a segment of a *string*, which is specified by two integers *start* and *end* satisfying these relationships:

```
0 <= start <= end <= (string-length string)
```

Start is the index of the first character in the substring, and *end* is one greater than the index of the last character in the substring. Thus if *start* and *end* are equal, they refer to an empty substring, and if *start* is zero and *end* is the length of *string*, they refer to all of *string*.

Some of the procedures that operate on strings ignore the difference between uppercase and lowercase. The versions that ignore case include '`-ci`' (for "case insensitive") in their names.

6.1 Construction of Strings

make-string *k* [*char*] procedure

Returns a newly allocated string of length *k*. If you specify *char*, all elements of the string are initialized to *char*, otherwise the contents of the string are unspecified. *Char* must satisfy the predicate `char-ascii?`.

```
(make-string 10 #\x)              ⇒  "xxxxxxxxxx"
```

string *char* ... procedure+

Returns a newly allocated string consisting of the specified characters. The arguments must all satisfy `char-ascii?`.

```
(string #\a)                                ⇒  "a"
(string #\a #\b #\c)                        ⇒  "abc"
(string #\a #\space #\b #\space #\c)        ⇒  "a b c"
(string)                                    ⇒  ""
```

list->string *char-list* procedure

Char-list must be a list of ASCII characters. `list->string` returns a newly allocated string formed from the elements of *char-list*. This is equivalent to `(apply string` *char-list*`)`. The inverse of this operation is `string->list`.

```
(list->string '(#\a #\b))          ⇒  "ab"
(string->list "Hello")             ⇒  (#\H #\e #\l #\l #\o)
```

string-copy *string* procedure

Returns a newly allocated copy of *string*.

Note regarding variable-length strings: the maximum length of the result depends only on the length of *string*, not its maximum length. If you wish to copy a string and preserve its maximum length, do the following:

```
(define (string-copy-preserving-max-length string)
  (let ((length))
    (dynamic-wind
     (lambda ()
       (set! length (string-length string))
       (set-string-length! string
                           (string-maximum-length string)))
     (lambda ()
       (string-copy string))
     (lambda ()
       (set-string-length! string length)))))
```

6.2 Selecting String Components

string? *object* procedure

Returns `#t` if *object* is a string; otherwise returns `#f`.

```
(string? "Hi")                    ⇒  #t
(string? 'Hi)                     ⇒  #f
```

string-length *string* procedure

Returns the length of *string* as an exact non-negative integer.

```
(string-length "")                ⇒  0
(string-length "The length")      ⇒  10
```

string-null? *string* procedure

Returns `#t` if *string* has zero length; otherwise returns `#f`.

```
(string-null? "")                 ⇒  #t
(string-null? "Hi")               ⇒  #f
```

string-ref *string k* procedure

Returns character *k* of *string*. *K* must be a valid index of *string*.

```
(string-ref "Hello" 1)            ⇒  #\e
(string-ref "Hello" 5)            error  5 not in correct range
```

string-set! *string k char* procedure

Stores *char* in element *k* of *string* and returns an unspecified value. *K* must be a valid index of *string*, and *char* must satisfy the predicate `char-ascii?`.

```
(define str "Dog")                ⇒  unspecified
(string-set! str 0 #\L)           ⇒  unspecified
str                               ⇒  "Log"
(string-set! str 3 #\t)           error  3 not in correct range
```

6.3 Comparison of Strings

string=? *string1 string2* procedure
substring=? *string1 start end string2 start end* procedure+
string-ci=? *string1 string2* procedure
substring-ci=? *string1 start end string2 start end* procedure+

Returns `#t` if the two strings (substrings) are the same length and contain the same characters in the same (relative) positions; otherwise returns `#f`. `string-ci=?` and `substring-ci=?` don't distinguish uppercase and lowercase letters, but `string=?` and `substring=?` do.

```
(string=? "PIE" "PIE")                    ⇒  #t
(string=? "PIE" "pie")                    ⇒  #f
(string-ci=? "PIE" "pie")                 ⇒  #t
(substring=? "Alamo" 1 3 "cola" 2 4)      ⇒  #t ; compares "la"
```

string<? *string1 string2* procedure
substring<? *string1 start1 end1 string2 start2 end2* procedure+
string>? *string1 string2* procedure
string<=? *string1 string2* procedure
string>=? *string1 string2* procedure
string-ci<? *string1 string2* procedure
substring-ci<? *string1 start1 end1 string2 start2 end2* procedure+
string-ci>? *string1 string2* procedure
string-ci<=? *string1 string2* procedure
string-ci>=? *string1 string2* procedure

These procedures compare strings (substrings) according to the order of the characters they contain (also see Section 5.2 [Comparison of Characters], page 62). The arguments are compared using a lexicographic (or dictionary) order. If two strings differ in length but are the same up to the length of the shorter string, the shorter string is considered to be less than the longer string.

```
(string<? "cat" "dog")          ⇒  #t
(string<? "cat" "DOG")          ⇒  #f
(string-ci<? "cat" "DOG")       ⇒  #t
(string>? "catkin" "cat")       ⇒  #t ; shorter is lesser
```

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-hash *string* procedure+
string-hash-mod *string k* procedure+

`string-hash` returns 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=?`) return equal (`=`) hash codes, and non-equal but similar strings are usually mapped to distinct hash codes.

`string-hash-mod` is like `string-hash`, except that it limits the result to a particular range based on the exact non-negative integer *k*. The following are equivalent:

```
(string-hash-mod string k)
(modulo (string-hash string) k)
```

6.4 Alphabetic Case in Strings

string-capitalized? *string* procedure+
substring-capitalized? *string start end* procedure+

These procedures return `#t` if the first word in the string (substring) is capitalized, and any subsequent words are either lower case or capitalized. Otherwise, they return `#f`. A word is defined as a non-null contiguous sequence of alphabetic characters, delimited by non-alphabetic characters or the limits of the string (substring). A word is capitalized if its first letter is upper case and all its remaining letters are lower case.

```
(map string-capitalized? '(""    "A"    "art"  "Art"  "ART"))
                       ⇒ (#f    #t     #f     #t     #f)
```

string-upper-case? *string* procedure+
substring-upper-case? *string start end* procedure+
string-lower-case? *string* procedure+
substring-lower-case? *string start end* procedure+

These procedures return `#t` if all the letters in the string (substring) are of the correct case, otherwise they return `#f`. The string (substring) must contain at least one letter or the procedures return `#f`.

```
(map string-upper-case?  '(""    "A"    "art"  "Art"  "ART"))
                       ⇒ (#f    #t     #f     #f     #t)
```

string-capitalize *string* procedure+
string-capitalize! *string* procedure+
substring-capitalize! *string start end* procedure+

`string-capitalize` returns a newly allocated copy of *string* in which the first alphabetic character is uppercase and the remaining alphabetic characters are lowercase. For example, `"abcDEF"` becomes `"Abcdef"`. `string-capitalize!` is the destructive version of `string-capitalize`: it alters *string* and returns an unspecified value. `substring-capitalize!` destructively capitalizes the specified part of *string*.

string-downcase *string* procedure+
string-downcase! *string* procedure+
substring-downcase! *string start end* procedure+

`string-downcase` returns a newly allocated copy of *string* in which all uppercase letters are changed to lowercase. `string-downcase!` is the destructive version of `string-downcase`: it alters *string* and returns an unspecified value. `substring-downcase!` destructively changes the case of the specified part of *string*.

```
(define str "ABCDEFG")          ⇒  unspecified
(substring-downcase! str 3 5)   ⇒  unspecified
str                             ⇒  "ABCdeFG"
```

string-upcase *string* procedure+
string-upcase! *string* procedure+
substring-upcase! *string start end* procedure+

`string-upcase` returns a newly allocated copy of *string* in which all lowercase letters are changed to uppercase. `string-upcase!` is the destructive version of `string-upcase`: it alters *string* and returns an unspecified value. `substring-upcase!` destructively changes the case of the specified part of *string*.

6.5 Cutting and Pasting Strings

string-append *string . . .* procedure

Returns a newly allocated string made from the concatenation of the given strings. With no arguments, `string-append` returns the empty string (`""`).

```
(string-append)                      ⇒  ""
(string-append "*" "ace" "*")        ⇒  "*ace*"
(string-append "" "" "")             ⇒  ""
(eq? str (string-append str))        ⇒  #f ; newly allocated
```

substring *string start end* procedure

Returns a newly allocated string formed from the characters of *string* beginning with index *start* (inclusive) and ending with *end* (exclusive).

```
(substring "" 0 0)             ⇒ ""
(substring "arduous" 2 5)      ⇒ "duo"
(substring "arduous" 2 8)      error  8 not in correct range

(define (string-copy s)
  (substring s 0 (string-length s)))
```

string-head *string end* procedure+

Returns a newly allocated copy of the initial substring of *string*, up to but excluding *end*. It could have been defined by:

```
(define (string-head string end)
  (substring string 0 end))
```

string-tail *string start* procedure+

Returns a newly allocated copy of the final substring of *string*, starting at index *start* and going to the end of *string*. It could have been defined by:

```
(define (string-tail string start)
  (substring string start (string-length string)))

(string-tail "uncommon" 2)       ⇒   "common"
```

string-pad-left *string k [char]* procedure+
string-pad-right *string k [char]* procedure+

These procedures return a newly allocated 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-trim *string* [*char-set*] procedure+
string-trim-left *string* [*char-set*] procedure+
string-trim-right *string* [*char-set*] procedure+

Returns a newly allocated 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)"
```

6.6 Searching Strings

substring? *pattern string* procedure+

Searches *string* to see if it contains the substring *pattern*. Returns the index of the first substring of *string* that is equal to *pattern*; or `#f` if *string* does not contain *pattern*.

```
(substring? "rat" "pirate")           ⇒  2
(substring? "rat" "outrage")          ⇒  #f
(substring? "" any-string)            ⇒  0
(if (substring? "moon" text)
    (process-lunar text)
    'no-moon)
```

string-find-next-char *string char* procedure+
substring-find-next-char *string start end char* procedure+
string-find-next-char-ci *string char* procedure+
substring-find-next-char-ci *string start end char* procedure+

Returns the index of the first occurrence of *char* in the string (substring); returns `#f` if *char* does not appear in the string. For the substring procedures, the index returned is relative to the entire string, not just the substring. The `-ci` procedures don't distinguish uppercase and lowercase letters.

```
(string-find-next-char "Adam" #\A)              ⇒  0
(substring-find-next-char "Adam" 1 4 #\A)       ⇒  #f
(substring-find-next-char-ci "Adam" 1 4 #\A)    ⇒  2
```

string-find-next-char-in-set *string char-set* procedure+
substring-find-next-char-in-set *string start end char-set* procedure+

Returns the index of the first character in the string (or substring) that is also in *char-set*, or returns `#f` if none of the characters in *char-set* occur in *string*. For the substring procedure, only the substring is searched, but the index returned is relative to the entire string, not just the substring.

```
(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* procedure+
substring-find-previous-char *string start end char* procedure+
string-find-previous-char-ci *string char* procedure+
substring-find-previous-char-ci *string start end char* procedure+

Returns the index of the last occurrence of *char* in the string (substring); returns `#f` if *char* doesn't appear in the string. For the substring procedures, the index returned is relative to the entire string, not just the substring. The `-ci` procedures don't distinguish uppercase and lowercase letters.

string-find-previous-char-in-set *string char-set* procedure+
substring-find-previous-char-in-set *string start end char-set* procedure+

Returns the index of the last character in the string (substring) that is also in *char-set*. For the substring procedure, the index returned is relative to the entire string, not just the substring.

6.7 Matching Strings

string-match-forward *string1 string2* procedure+
substring-match-forward *string1 start end string2 start end* procedure+
string-match-forward-ci *string1 string2* procedure+
substring-match-forward-ci *string1 start end string2 start end* procedure+

Compares the two strings (substrings), starting from the beginning, and returns the number of characters that are the same. If the two strings (substrings) start differently, returns 0. The `-ci` procedures don't distinguish uppercase and lowercase letters.

```
(string-match-forward "mirror" "micro") ⇒  2  ; matches "mi"
(string-match-forward "a" "b")          ⇒  0  ; no match
```

string-match-backward *string1 string2* procedure+
substring-match-backward *string1 start end string2 start end* procedure+
string-match-backward-ci *string1 string2* procedure+
substring-match-backward-ci *string1 start end string2 start end* procedure+

Compares the two strings (substrings), starting from the end and matching toward the front, returning the number of characters that are the same. If the two strings (substrings) end differently, returns 0. The `-ci` procedures don't distinguish uppercase and lowercase letters.

```
(string-match-backward-ci "BULBOUS" "fractious")
                                      ⇒  3   ; matches "ous"
```

string-prefix? *string1 string2* procedure+
substring-prefix? *string1 start1 end1 string2 start2 end2* procedure+
string-prefix-ci? *string1 string2* procedure+
substring-prefix-ci? *string1 start1 end1 string2 start2 end2* procedure+

These procedures return `#t` if the first string (substring) 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+
substring-suffix? *string1 start1 end1 string2 start2 end2* procedure+
string-suffix-ci? *string1 string2* procedure+
substring-suffix-ci? *string1 start1 end1 string2 start2 end2* procedure+

These procedures return `#t` if the first string (substring) 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
```

6.8 Modification of Strings

string-replace *string char1 char2* procedure+
substring-replace *string start end char1 char2* procedure+
string-replace! *string char1 char2* procedure+
substring-replace! *string start end char1 char2* procedure+

These procedures replace all occurrences of *char1* with *char2* in the original string (substring). `string-replace` and `substring-replace` return a newly allocated string containing the result. `string-replace!` and `substring-replace!` destructively modify *string* and return an unspecified value.

```
(define str "a few words")                  ⇒  unspecified
(string-replace str #\space #\-)            ⇒  "a-few-words"
(substring-replace str 2 9 #\space #\-)     ⇒  "a few-words"
str                                         ⇒  "a few words"
(string-replace! str #\space #\-)           ⇒  unspecified
str                                         ⇒  "a-few-words"
```

string-fill! *string char* procedure

Stores *char* in every element of *string* and returns an unspecified value.

substring-fill! *string start end char* procedure+

Stores *char* in elements *start* (inclusive) to *end* (exclusive) of *string* and returns an unspecified value.

```
(define s (make-string 10 #\space))    ⇒  unspecified
(substring-fill! s 2 8 #\*)            ⇒  unspecified
s                                      ⇒  "  ******  "
```

substring-move-left! *string1 start1 end1 string2 start2* procedure+
substring-move-right! *string1 start1 end1 string2 start2* procedure+

Copies the characters from *start1* to *end1* of *string1* into *string2* at the *start2*-th position. The characters are copied as follows (note that this is only important when *string1* and *string2* are `eqv?`):

`substring-move-left!`
: The copy starts at the left end and moves toward the right (from smaller indices to larger). Thus if *string1* and *string2* are the same, this procedure moves the characters toward the left inside the string.

`substring-move-right!`
: The copy starts at the right end and moves toward the left (from larger indices to smaller). Thus if *string1* and *string2* are the same, this procedure moves the characters toward the right inside the string.

The following example shows how these procedures can be used to build up a string (it would have been easier to use `string-append`):

```
(define answer (make-string 9 #\*))          ⇒  unspecified
answer                                       ⇒  "*********"
(substring-move-left! "start" 0 5 answer 0)  ⇒  unspecified
answer                                       ⇒  "start****"
(substring-move-left! "-end" 0 4 answer 5)   ⇒  unspecified
answer                                       ⇒  "start-end"
```

6.9 Variable-Length Strings

MIT Scheme allows the length of a string to be dynamically adjusted in a limited way. This feature works as follows. When a new string is allocated, by whatever method, it has a specific length. At the time of allocation, it is also given a *maximum length*, which is guaranteed to be at least as large as the string's length. (Sometimes the maximum length will be slightly larger than the length, but it is a bad idea to count on this. Programs should assume that the maximum length is the same as the length at the time of the string's allocation.) After the string is allocated, the operation `set-string-length!` can be used to alter the string's length to any value between 0 and the string's maximum length, inclusive.

string-maximum-length *string* procedure+

Returns the maximum length of *string*. The following is guaranteed:

```
(<= (string-length string)
    (string-maximum-length string))     ⇒  #t
```

The maximum length of a string *never* changes.

set-string-length! *string k* procedure+

Alters the length of *string* to be *k*, and returns an unspecified value. *K* must be less than or equal to the maximum length of *string*. `set-string-length!` does not change the maximum length of *string*.

6.10 Byte Vectors

MIT Scheme implements strings as packed vectors of 8-bit ASCII bytes. Most of the string operations, such as `string-ref`, coerce these 8-bit codes into character objects. However, some lower-level operations are made available for use.

vector-8b-ref *string k* procedure+

Returns character *k* of *string* as an ASCII code. *K* must be a valid index of *string*.

```
(vector-8b-ref "abcde" 2)            ⇒  99 ; ascii for 'c'
```

vector-8b-set! *string k ascii* procedure+

Stores *ascii* in element *k* of *string* and returns an unspecified value. *K* must be a valid index of *string*, and *ascii* must be a valid ASCII code.

vector-8b-fill! *string start end ascii* procedure+

Stores *ascii* in elements *start* (inclusive) to *end* (exclusive) of *string* and returns an unspecified value. *Ascii* must be a valid ASCII code.

vector-8b-find-next-char *string start end ascii* procedure+
vector-8b-find-next-char-ci *string start end ascii* procedure+

Returns the index of the first occurrence of *ascii* in the given substring; returns `#f` if *ascii* does not appear. The index returned is relative to the entire string, not just the substring. *Ascii* must be a valid ASCII code.

`vector-8b-find-next-char-ci` doesn't distinguish uppercase and lowercase letters.

vector-8b-find-previous-char *string start end ascii* procedure+
vector-8b-find-previous-char-ci *string start end ascii* procedure+

Returns the index of the last occurrence of *ascii* in the given substring; returns `#f` if *ascii* does not appear. The index returned is relative to the entire string, not just the substring. *Ascii* must be a valid ASCII code.

`vector-8b-find-previous-char-ci` doesn't distinguish uppercase and lowercase letters.

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:

[1] The above definitions imply that all lists have finite length and are terminated by the empty list.

```
(a b c . d)
(a . (b . (c . d)))
```

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* 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* 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)
```

car *pair* 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* 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
```

set-car! *pair object* 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* procedure

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

caar *pair* procedure
cadr *pair* procedure
cdar *pair* procedure
cddr *pair* procedure
caaar *pair* procedure
caadr *pair* procedure
cadar *pair* procedure
caddr *pair* procedure
cdaar *pair* procedure
cdadr *pair* procedure
cddar *pair* procedure
cdddr *pair* procedure
caaaar *pair* procedure
caaadr *pair* procedure
caadar *pair* procedure
caaddr *pair* procedure
cadaar *pair* procedure
cadadr *pair* procedure
caddar *pair* procedure
cadddr *pair* procedure
cdaaar *pair* procedure
cdaadr *pair* procedure
cdadar *pair* procedure
cdaddr *pair* procedure
cddaar *pair* procedure
cddadr *pair* procedure
cdddar *pair* procedure
cddddr *pair* 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* 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 ...* 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 '()) ...))
```

make-list *k* [*element*] procedure+

This procedure returns a newly allocated list of length *k*, whose elements are all *element*. If *element* is not supplied, it defaults to the empty list.

cons* *object object ...* 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.

[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).

```
(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-copy *list* 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)))))
```

vector->list *vector* procedure
subvector->list *vector start end* procedure+

`vector->list` returns a newly allocated list of the elements of *vector*. `subvector->list` returns a newly allocated list of the elements of the given subvector. The inverse of `vector->list` is `list->vector`.

```
(vector->list '#(dah dah didah))    ⇒  (dah dah didah)
```

string->list *string* procedure
substring->list *string start end* procedure

`string->list` returns a newly allocated list of the character elements of *string*. `substring->list` returns a newly allocated list of the character elements of the given substring. The inverse of `string->list` is `list->string`.

```
(string->list "abcd")               ⇒  (#\a #\b #\c #\d)
(substring->list "abcdef" 1 3)      ⇒  (#\b #\c)
```

7.3 Selecting List Components

list? *object* procedure+

Returns `#t` if *object* is a list, otherwise returns `#f`. By definition, all lists have finite length and are terminated by the empty list. This procedure returns an answer even for circular structures.

Any *object* satisfying this predicate will also satisfy exactly one of `pair?` or `null?`.

```
(list? '(a b c))                    ⇒  #t
(list? '())                         ⇒  #t
(list? '(a . b))                    ⇒  #f
(let ((x (list 'a)))
  (set-cdr! x x)
  (list? x))                        ⇒  #f
```

length *list* procedure

Returns the length of *list*.

```
(length '(a b c))                    ⇒  3
(length '(a (b) (c d e)))            ⇒  3
(length '())                         ⇒  0
```

null? *object* procedure

Returns `#t` if *object* is the empty list; otherwise returns `#f` (but see Section 1.2.5 [True and False], page 8).

```
(null? '(a . b))                     ⇒  #f
(null? '(a b c))                     ⇒  #f
(null? '())                          ⇒  #t
```

list-ref *list k* 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)
          (inexact->exact (round 1.8)))
     ⇒  c
```

`(list-ref` *list k*`)` is equivalent to `(car (list-tail` *list k*`))`.

first *list* procedure+
second *list* procedure+
third *list* procedure+
fourth *list* procedure+
fifth *list* procedure+
sixth *list* procedure+
seventh *list* procedure+
eighth *list* procedure+
ninth *list* procedure+
tenth *list* 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

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).

list-head *list k* procedure+

Returns a newly allocated list consisting of the first *k* elements of *list*. *K* must not be greater than the length of *list*.

We could have defined `list-head` this way:

```
(define (list-head list k)
  (sublist list 0 k))
```

list-tail *list k* procedure

Returns the sublist of *list* obtained by omitting the first *k* elements. The result, if it is not the empty list, shares structure with *list*. *K* must not be greater than the length of *list*.

append *list ...* procedure

Returns a list consisting of the elements of the first *list* followed by the elements of the other *lists*.

```
(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 ...* procedure+

Returns a list that is the argument *lists* concatenated together. The arguments are changed rather than copied. (Compare this with `append`, which copies arguments rather than destroying them.) For example:

```
(define x '(a b c))
(define y '(d e f))
(define z '(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)
```

7.5 Filtering Lists

list-transform-positive *list predicate* procedure+
list-transform-negative *list predicate* procedure+

These procedures return a newly allocated copy of *list* containing only the elements for which *predicate* is (respectively) true or false. *Predicate* must be a procedure of one argument.

```
(list-transform-positive '(1 2 3 4 5) odd?) ⇒ (1 3 5)
(list-transform-negative '(1 2 3 4 5) odd?) ⇒ (2 4)
```

delq *element list* procedure+
delv *element list* procedure+
delete *element list* procedure+

Returns a newly allocated copy of *list* with all entries equal to *element* removed. `delq` uses `eq?` to compare *element* with the entries in *list*, `delv` uses `eqv?`, and `delete` uses `equal?`.

delq! *element list* procedure+
delv! *element list* procedure+
delete! *element list* procedure+

Returns a list consisting of the top-level elements of *list* with all entries equal to *element* removed. These procedures are like `delq`, `delv`, and `delete` except that they destructively modify *list*. `delq!` uses `eq?` to compare element with the entries in *list*, `delv!` uses `eqv?`, and `delete!` uses `equal?`. Because the result may not be `eq?` to *list*, it is desirable to do something like `(set! x (delete! x))`.

```
(define x '(a b c b))
(delete 'b x)                          ⇒  (a c)
x                                      ⇒  (a b c b)

(define x '(a b c b))
(delete! 'b x)                         ⇒  (a c)
x                                      ⇒  (a c)
;; Returns correct result:
(delete! 'a x)                         ⇒  (c)

;; Didn't modify what x points to:
x                                      ⇒  (a c)
```

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

list-search-positive *list predicate* procedure+
list-search-negative *list predicate* procedure+

Returns the first element in *list* for which *predicate* is (respectively) true or false; returns `#f` if it doesn't find such an element. (This means that if *predicate* is true (false) for `#f`, it may be impossible to distinguish a successful result from an unsuccessful one.) *Predicate* must be a procedure of one argument.

memq *object list* procedure
memv *object list* procedure
member *object list* 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 `equal?`.[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 ...* procedure

Procedure must be a procedure taking as many arguments as there are *lists*. If more than one *list* is given, then they must all be the same length. `map` applies *procedure* element-wise to the elements of the *lists* and returns a list of the results, in order from left to right. The dynamic order in which *procedure* is applied to the elements of the *lists* is unspecified; use `for-each` to sequence side effects.

```
(map cadr '((a b) (d e) (g h)))           ⇒  (b e h)
(map (lambda (n) (expt n n)) '(1 2 3 4))  ⇒  (1 4 27 256)
(map + '(1 2 3) '(4 5 6))                 ⇒  (5 7 9)
(let ((count 0))
  (map (lambda (ignored)
         (set! count (+ count 1))
         count)
       '(a b c)))                          ⇒  unspecified
```

map* *initial-value procedure list1 list2 ...* procedure+

Similar to `map`, except that the resulting list is terminated by *initial-value* rather than the empty list. The following are equivalent:

```
(map procedure list list ...)
(map* '() procedure list list ...)
```

append-map *procedure list list ...* procedure+

append-map* *initial-value procedure list list ...* procedure+

Similar to `map` and `map*`, respectively, except that the results of applying *procedure* to the elements of *lists* are concatenated together by `append` rather than by `cons`. The following are equivalent, except that the former is more efficient:

```
(append-map procedure list list ...)
(apply append (map procedure list list ...))
```

append-map! *procedure list list ...* procedure+

append-map*! *initial-value procedure list list ...* procedure+

Similar to `map` and `map*`, respectively, except that the results of applying *procedure* to the elements of *lists* are concatenated together by `append!` rather than by `cons`. The following are equivalent, except that the former is more efficient:

```
(append-map! procedure list list ...)
(apply append! (map procedure list list ...))
```

for-each *procedure list list ...* procedure

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 *lists* in order from the first element to the last, and the value returned by `for-each` is unspecified.

```
(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)
```

7.8 Reduction of Lists

reduce *procedure initial list* procedure+

Combines all the elements of *list* using the binary operation *procedure*. For example, using `+` one can add up all the elements:

```
(reduce + 0 list-of-numbers)
```

The argument *initial* is used only if *list* is empty; in this case *initial* is the result of the call to `reduce`. If *list* has a single argument, it is returned. Otherwise, the arguments are reduced in a left-associative fashion. For example:

```
(reduce + 0 '(1 2 3 4))                ⇒  10
(reduce + 0 '(1 2))                    ⇒  3
(reduce + 0 '(1))                      ⇒  1
(reduce + 0 '())                       ⇒  0
(reduce + 0 '(foo))                    ⇒  foo
(reduce list '() '(1 2 3 4))           ⇒  (((1 2) 3) 4)
```

reduce-right *procedure initial list* procedure+

Like `reduce` except that it is right-associative.

```
(reduce-right list '() '(1 2 3 4))     ⇒  (1 (2 (3 4)))
```

fold-right *procedure initial list* procedure+

Combines all of the elements of *list* using the binary operation *procedure*. Unlike `reduce` and `reduce-right`, *initial* is always used:

```
(fold-right + 0 '(1 2 3 4))            ⇒  10
(fold-right + 0 '(foo))                error  Illegal datum
(fold-right list '() '(1 2 3 4))       ⇒  (1 (2 (3 (4 ()))))
```

`Fold-right` has interesting properties because it establishes a homomorphism between (`cons`, `()`) and (*procedure*, *initial*). It can be thought of as replacing the pairs in the spine of the list with *procedure* and replacing the `()` at the end with *initial*. Many of the classical list-processing procedures can be expressed in terms of `fold-right`, at least for the simple versions that take a fixed number of arguments:

```
(define (copy-list list)
  (fold-right cons '() list))

(define (append list1 list2)
  (fold-right cons list2 list1))

(define (map p list)
  (fold-right (lambda (x r) (cons (p x) r)) '() list))

(define (reverse items)
  (fold-right (lambda (x r) (append r (list x))) '() items))
```

fold-left *procedure initial list* procedure+

Combines all the elements of *list* using the binary operation *procedure*. Elements are combined starting with *initial* and then the elements of *list* from left to right. Whereas `fold-right` is recursive in nature, capturing the essence of `cdr`-ing down a list and then computing a result, *fold-left* is iterative in nature, combining the elements as the list is traversed.

```
(fold-left list '() '(1 2 3 4))          ⇒  ((((() 1) 2) 3) 4)

(define (length list)
  (fold-left (lambda (sum element) (+ sum 1)) 0 list))

(define (reverse items)
  (fold-left (lambda (x y) (cons y x)) () items))
```

there-exists? *list predicate* procedure+

Predicate must be a procedure of one argument. Applies *predicate* to each element of *list*, in order from left to right. If *predicate* is true for any element of *list*, the value yielded by *predicate* is immediately returned as the value of `there-exists?`; *predicate* will not be applied to the remaining elements of *list*. If *predicate* returns `#f` for all of the elements of *list*, then `#f` is returned.

for-all? *list predicate* procedure+

Predicate must be a procedure of one argument. Applies *predicate* to each element of *list*, in order from left to right. If *predicate* returns `#f` for any element of *list*, `#f` is immediately returned as the value of `for-all?`; *predicate* will not be applied to the remaining elements of *list*. If *predicate* is true for all of the elements of *list*, then `#t` is returned.

7.9 Miscellaneous List Operations

circular-list *object ...* procedure+
make-circular-list *k* [*element*] procedure+

These procedures are like `list` and `make-list`, respectively, except that the returned lists are circular. `circular-list` could have been defined like this:

```
(define (circular-list . objects)
  (append! objects objects))
```

reverse *list* 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* 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))`.

last-pair *list* procedure+

Returns the last pair in *list*, which may be an improper list. `last-pair` could have been defined this way:

```
(define last-pair
  (lambda (x)
    (if (pair? (cdr x))
        (last-pair (cdr x))
        x)))
```

except-last-pair *list* procedure+
except-last-pair! *list* procedure+

These procedures remove the last pair from *list*. *List* may be an improper list, except that it must consist of at least one pair. `except-last-pair` returns a newly allocated copy of *list* that omits the last pair. `except-last-pair!` destructively removes the last pair from *list* and returns *list*. If the cdr of *list* is not a pair, the empty list is returned by either procedure.

sort *sequence procedure* procedure+

Sequence must be either a list or a vector. *Procedure* must be a procedure of two arguments that defines a *total ordering* on the elements of *sequence*. In other words, if *x* and *y* are two distinct elements of *sequence*, then it must be the case that

```
(and (procedure x y)
     (procedure y 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*. 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 y x)
     ⇒  #f
```

See also the definition of `sort!`.

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)
```

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 *vector procedure* procedure+

Procedure must be a procedure of one argument. `vector-map` applies *procedure* element-wise to the elements of *vector* and returns a newly allocated vector of the results, in order from left to right. The dynamic order in which *procedure* is applied to the elements of *vector* is unspecified.

```
(vector-map '#((a b) (d e) (g h)) cadr)     ⇒  #(b e h)
(vector-map '#(1 2 3 4) (lambda (n) (expt n n)))
                                            ⇒  #(1 4 27 256)
(vector-map '#(5 7 9) +)                    ⇒  #(5 7 9)
```

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* procedure+

Procedure must be a procedure of two arguments that defines a *total ordering* on the elements of *vector*. The elements of *vector* are rearranged so that they are sorted in the order defined by *procedure*. 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.

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 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 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 kth bit is 1; otherwise returns `#f`. K must be a valid index of *bit-string*.

bit-string-set! *bit-string k* procedure+

Sets the kth 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 kth 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* procedure

Returns `#t` if *object* is either `#t` or `#f`; otherwise returns `#f`.

```
(boolean? #f)                       ⇒  #t
(boolean? 0)                        ⇒  #f
```

not *object* 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
```

boolean=? *obj1 obj2* procedure+

This predicate is true iff *obj1* and *obj2* are either both true or both false.

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 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 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 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 117). For these reasons, uninterned symbols are not very useful in MIT Scheme. In fact, their primary purpose is to simplify the generation of unique variable names in programs that generate Scheme code.

[2] MIT 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

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.

---

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.

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-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 Cells

Cells are data structures similar to pairs except that they have only one element. They are useful for managing state.

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 19).

10.4 Records

MIT 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 29).

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-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)
```

[3] In MIT Scheme, the returned list is always newly allocated.

```
(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
```

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 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 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 kth 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 kth 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 *streams* as its arguments.

10.7 Weak Pairs

Weak pairs are a mechanism for building data structures that point at objects without protecting them from garbage collection. The car of a weak pair holds its pointer weakly, while the cdr holds its pointer in the normal way. If the object in the car of a weak pair is not held normally by any other data structure, it will be garbage-collected.

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.

weak-pair/car? *weak-pair* 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.

weak-car *weak-pair* procedure+

Returns the car component of *weak-pair*. If the car component has been garbage-collected, this operation returns `#f`, but it can also return `#f` if that is the value that was stored in the car.

Normally, `weak-pair/car?` is used to determine if `weak-car` would return a valid value. An obvious way of doing this would be:

```
(if (weak-pair/car? x)
    (weak-car x)
    ...)
```

However, since a garbage collection could occur between the call to `weak-pair/car?` and `weak-car`, this would not always work correctly. Instead, the following should be used, which always works:

```
(or (weak-car x)
    (and (not (weak-pair/car? x))
         ...))
```

The reason that the latter expression works is that `weak-car` returns `#f` in just two instances: when the car component is `#f`, and when the car component has been garbage-collected. In the former case, if a garbage collection happens between the two calls, it won't matter, because `#f` will never be garbage-collected. And in the latter case, it also won't matter, because the car component no longer exists and cannot be affected by the garbage collector.

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-cdr*.

weak-set-cdr! *weak-pair object* procedure+

Sets the cdr component of *weak-pair* to *object* and returns an unspecified result.

11 Associations

MIT 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 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 89).

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 118). 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 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 Scheme's hash tables feature automatic resizing, customizable growth parameters, and customizable hash procedures.

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 122).

The hash-table implementation is a run-time-loadable option. To use hash tables, execute

```
(load-option 'hash-table)
```

once before calling any of the procedures defined here.

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 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 122).

Hash tables are normally characterized by two things: the equivalence predicate that is used to compare keys, and whether or not the table allows its keys to be reclaimed by the garbage collector. If a table prevents its keys from being reclaimed by the garbage collector, it is said to hold its keys *strongly*; otherwise it holds its keys *weakly* (see Section 10.7 [Weak Pairs], page 113).

make-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. These are the fastest of the standard hash tables.

make-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, and numbers are held strongly. These hash tables are a little slower than those made by `make-eq-hash-table`.

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 are held strongly. These hash tables are quite a bit slower than those made by `make-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 are held strongly.

The next two procedures are used to create new hash-table constructors. All of the above hash table constructors, with the exception of `make-eqv-hash-table`, could have been created by calls to these "constructor-constructors"; see the examples below.

strong-hash-table/constructor *key-hash key=?* [*rehash-after-gc?*] procedure+

weak-hash-table/constructor *key-hash key=?* [*rehash-after-gc?*] procedure+

Each of these procedures accepts two arguments and returns a hash-table constructor. The *key=?* argument is an equivalence predicate for the keys of the hash table. The *key-hash* argument is a procedure that computes a hash number. Specifically, *key-hash* 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 optional argument *rehash-after-gc?*, if true, says that the values returned by *key-hash* 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 125, for information about hash procedures that have this property.) Otherwise, it is assumed that *key-hash* always returns the same value for the same arguments. The default value of this argument is `#f`.

The constructors returned by `strong-hash-table/constructor` make hash tables that hold their keys strongly. The constructors returned by `weak-hash-table/constructor` make hash tables that hold their keys weakly.

Some examples showing how some standard hash-table constructors could have been defined:

```
(define make-eq-hash-table
  (weak-hash-table/constructor eq-hash-mod eq? #t))

(define make-equal-hash-table
  (strong-hash-table/constructor equal-hash-mod equal? #t))

(define make-string-hash-table
  (strong-hash-table/constructor string-hash-mod string=? #f))
```

The following procedure is sometimes useful in conjunction with weak hash tables. Normally it is not needed, because such hash tables clean themselves automatically as they are used.

hash-table/clean! *hash-table* procedure+

If *hash-table* is a type of hash table that holds its *keys* weakly, 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.

11.4.2 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/put! *hash-table key datum* 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/get *hash-table key default* procedure+

Returns the datum associated with *key* in *hash-table*. If there is no association for *key*, *default* is returned. The average time required by this operation is bounded by a constant.

hash-table/remove! *hash-table key* 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+

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/count *hash-table* procedure+

Returns the number of associations in *hash-table* as an exact non-negative integer. If *hash-table* holds its keys weakly, this is a conservative upper bound that may count some associations whose keys 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/key-list *hash-table* 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/datum-list *hash-table* 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/for-each *hash-table procedure* 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/remove!` to remove the association being processed.

The following procedure is an alternate form of `hash-table/get` that is useful in some situations. Usually, `hash-table/get` is preferable because it is faster.

hash-table/lookup *hash-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 *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.

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/size *hash-table* procedure+

Returns the usable size of *hash-table* as an exact positive integer. This is the number of associations that *hash-table* can hold before it will grow.

hash-table/rehash-size *hash-table* procedure+

Returns the rehash size of *hash-table*.

set-hash-table/rehash-size! *hash-table x* 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+

Returns the rehash threshold of *hash-table*.

set-hash-table/rehash-threshold! *hash-table x* 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 "constructor-constructor" procedure) 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.

The following procedures are the key-hashing procedures used by the standard address-hash-based hash tables.

eq-hash-mod *object modulus* procedure+

This procedure is the key-hashing procedure used by `make-eq-hash-table`.

eqv-hash-mod *object modulus* procedure+

This procedure is the key-hashing procedure used by `make-eqv-hash-table`.

equal-hash-mod *object modulus* procedure+

This procedure is the key-hashing procedure used by `make-equal-hash-table`.

11.4.5 Low-Level Hash Table Operations

The procedures in this section allow the programmer to control some of the internal structure of a hash table. Normally, hash tables maintain associations between keys and datums using pairs or weak pairs. These procedures allow the programmer to specify the use of some other data structure to maintain the association. In this section, the data structure that represents an association in a hash table is called an *entry*.

hash-table/constructor *key-hash key=? make-entry entry-valid? entry-key entry-datum set-entry-datum!* [*rehash-after-gc?*] procedure+

Creates and returns a hash-table constructor procedure (see Section 11.4.1 [Construction of Hash Tables], page 119). The arguments define the characteristics of the hash table as follows:

key-hash The hashing procedure. A procedure that 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.

key=? A equivalence predicate that accepts two keys and is true iff they are the same key. If this predicate is true of two keys, then *key-hash* must return the same value for each of these keys (given the same modulus in both cases).

make-entry
A procedure that accepts a key and a datum as arguments and returns a newly allocated entry.

entry-valid?
A procedure that accepts an entry and returns `#f` iff the entry's key has been reclaimed by the garbage collector. Instead of a procedure, this may be `#t`, which is equivalent to `(lambda (entry) #t)`.

entry-key A procedure that accepts an entry as an argument and returns the entry's key.

entry-datum
A procedure that accepts an entry as an argument and returns the entry's datum.

set-entry-datum!
A procedure that accepts an entry and an object as arguments, modifies the entry's datum to be the object, and returns an unspecified result.

rehash-after-gc?
An optional argument that, if true, says the values returned by *key-hash* 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 125, for information about hash procedures that have this property.) Otherwise, it is assumed that *key-hash* always returns the same value for the same arguments. The default value of this argument is `#f`.

For example, here is how the constructors for ordinary hash tables could be defined:

```
(define (strong-hash-table/constructor key-hash key=?
                                       #!optional rehash-after-gc?)
  (hash-table/constructor key-hash key=?
                          cons #t car cdr set-cdr!
                          (if (default-object? rehash-after-gc?)
                              #f
                              rehash-after-gc?)))
```

```
(define (weak-hash-table/constructor key-hash key=?
                                     #!optional rehash-after-gc?)
  (hash-table/constructor key-hash key=? weak-cons weak-pair/car?
                          weak-car weak-cdr weak-set-cdr!
                          (if (default-object? rehash-after-gc?)
                              #f
                              rehash-after-gc?)))
```

hash-table/key-hash *hash-table* procedure+
hash-table/key=? *hash-table* procedure+
hash-table/make-entry *hash-table* procedure+
hash-table/entry-valid? *hash-table* procedure+
hash-table/entry-key *hash-table* procedure+
hash-table/entry-datum *hash-table* procedure+
hash-table/set-entry-datum! *hash-table* procedure+

Each procedure returns the value of the corresponding argument that was used to construct *hash-table*.

The following procedures return the contents of a hash table as a collection of entries. While the data structure holding the entries is newly allocated, the entries themselves are not copied. Since hash table operations can modify these entries, the entries should be copied if it is desired to keep them while continuing to modify the table.

hash-table/entries-list *hash-table* procedure+

Returns a newly allocated list of the entries in *hash-table*.

hash-table/entries-vector *hash-table* procedure+

Returns a newly allocated vector of the entries in *hash-table*. Equivalent to

```
(list->vector (hash-table/entries-list hash-table))
```

11.5 Object Hashing

The MIT 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 useful in conjunction with hash tables, but it may be used for other things as well. In particular, it is used in the generation of the written representation for many objects (see Section 14.7 [Custom Output], page 164).

All of these procedures accept an optional argument called *table*; this table contains the object-integer associations. If given, this argument must be an object-hash table as constructed by `hash-table/make` (see below). If not given, a default table is used.

hash *object* [*table*] procedure+

`hash` associates an exact non-negative integer with *object* and returns that integer. If `hash` was previously called with *object* as its argument, the integer returned is the same as was returned by the previous call. `hash` guarantees that distinct objects (in the sense of `eq?`) are associated with distinct integers.

unhash *k* [*table*] procedure+

unhash 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` previously returned *k* for some object, and that object has not been reclaimed, it is the value of the call to `unhash`.

An object that is passed to `hash` 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` with the hash number of the (now reclaimed) object will signal an error.

```
(define x (cons 0 0))           ⇒  unspecified
(hash x)                        ⇒  77
(eqv? (hash x) (hash x))        ⇒  #t
(define x 0)                    ⇒  unspecified
(gc-flip)                       ;force a garbage collection
(unhash 77)                     error
```

object-hashed? *object* [*table*] procedure+

This predicate is true if *object* has an associated hash number. Otherwise it is false.

valid-hash-number? *k* [*table*] procedure+

This predicate is true if *k* is the hash number associated with some object. Otherwise it is false.

The following two procedures provide a lower-level interface to the object-hashing mechanism.

object-hash *object* [*table* [*insert?*]] procedure+

`object-hash` is like `hash`, except that it accepts an additional optional argument, *insert?*. If *insert?* is supplied and is `#f`, `object-hash` will return an integer for *object* only if there is already an association in the table; otherwise, it will return `#f`. If *insert?* is not supplied, or is not `#f`, `object-hash` always returns an integer, creating an association in the table if necessary.

`object-hash` additionally treats `#f` differently than does `hash`. Calling `object-hash` with `#f` as its argument will return an integer that, when passed to `unhash`, will signal an error rather than returning `#f`. Likewise, `valid-hash-number?` will return `#f` for this integer.

object-unhash *k* [*table*] procedure+

`object-unhash` is like `unhash`, except that when *k* is not associated with any object or was previously associated with an object that has been reclaimed, `object-unhash` returns `#f`. This means that there is an ambiguity in the value returned by `object-unhash`: if `#f` is returned, there is no way to tell if *k* is associated with `#f` or is not associated with any object at all.

Finally, this procedure makes new object-hash tables:

hash-table/make procedure+

This procedure creates and returns a new, empty object-hash table that is suitable for use as the optional *table* argument to the above procedures. The returned table 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 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.

The red-black tree implementation is a run-time-loadable option. To use red-black trees, execute

```
(load-option 'rb-tree)
```

once before calling any of the procedures defined here.

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/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->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 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?`.

The weight-balanced tree implementation is a run-time-loadable option. To use weight-balanced trees, execute

```
(load-option 'wt-tree)
```

once before calling any of the procedures defined here.

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))
```

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 20). 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 Scheme also provides *application hooks*, which support the construction of data structures that act like procedures.

In MIT 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`.

The following two procedures test the *arity* of a procedure, that is, the number of arguments that the procedure accepts. 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. This is because these procedures examine the apparent arity of a procedure. 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-valid? *procedure k* procedure+

Returns `#t` if *procedure* accepts *k* arguments; otherwise returns `#f`.

procedure-arity *procedure* procedure+

Returns a description of the number of arguments that *procedure* accepts. The result is a newly allocated pair whose car field is the minimum number of arguments, and whose cdr field is the maximum number of arguments. The minimum is an exact non-negative integer. The maximum is either an exact non-negative integer, or `#f` meaning that the procedure has no maximum number of arguments.

```
(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-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 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.

12.3 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 context of *continuation*. In other words, the "current" continuation is abandoned before *thunk* is invoked.

dynamic-wind *before-thunk action-thunk after-thunk* procedure+

This facility is a generalization of Common Lisp `unwind-protect`, designed to take into account the fact that continuations produced by `call-with-current-continuation` may be reentered. The arguments *before-thunk*, *action-thunk*, and *after-thunk* must all be procedures of no arguments (thunks).

`dynamic-wind` behaves as follows. First *before-thunk* is called. Then *action-thunk* is called. Finally, *after-thunk* is called. The value returned by *action-thunk* is returned as the result of `dynamic-wind`. *After-thunk* is also called if *action-thunk* escapes from its continuation. If *action-thunk* captures its continuation as an escape procedure, escapes from it, then escapes back to it, *after-thunk* is invoked when escaping away, and *before-thunk* is invoked when escaping back.

`dynamic-wind` is useful, for example, for ensuring the proper maintenance of locks: locking would occur in the *before-thunk*, protected code would appear in the *action-thunk*, and unlocking would occur in the *after-thunk*.

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 *objects*.

12.4 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 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.

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*.

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: `(`*name*`)` indicates that *name* is bound but unassigned, while `(`*name object*`)` indicates that *name* is bound, and its value is *object*.

environment-bound? *environment symbol* procedure+

Returns `#t` if *symbol* is bound in *environment* or one of its ancestor environments; otherwise returns `#f`.

environment-lookup *environment symbol* procedure+

Symbol must be bound in *environment* or one of its ancestor environments. Returns the value to which it is bound.

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.

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 stores definitions. It is a child of the `system-global-environment`, which is where all of the Scheme system definitions are stored. 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 environment that's the parent of the `user-initial-environment`. Primitives and system procedures 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 Scheme system procedures, which are defined in the `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 Interpreter Environments

The operations in this section return environments that are constructed by the interpreter. These operations should only be used at the top level of a file; they are not supported in any other place. In particular, they force the current environment to be represented in a form suitable for use by the interpreter. This prevents the compiler from performing many useful optimizations on such environments, and forces the use of the interpreter for variable references in them. However, because all top-level environments (such as `user-initial-environment`) are already interpreter environments, it does no harm to use such operations on them.

Warning: A future release of MIT Scheme will support hygenic macros, which are incompatible with non-top-level instances of `make-environment` and `the-environment`. At that time, other uses of these constructs will be disallowed.

make-environment *expression* ... special form+

Produces a new environment that is a child of the environment in which it is executed, evaluates the *expressions* sequentially in the new environment, and returns the new environment. Note that

```
(make-environment expression ...)
```

is equivalent to:

```
(let ()
  expression ...
  (the-environment))
```

the-environment special form+

Returns the current environment.

interpreter-environment? *object* procedure+

Returns `#t` if *object* is an interpreter environment; otherwise returns `#f`.

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

Scheme uses ports for I/O. A *port*, which can be treated like any other Scheme object, serves as a source or sink for data. A port must be open before it can be read from or written to. The standard I/O port, `console-i/o-port`, is opened automatically when you start Scheme. When you use a file for input or output, you need to explicitly open and close a port to the file (with procedures described in this chapter). Additional procedures let you open ports to strings.

Many input procedures, such as `read-char` and `read`, read data from the current input port by default, or from a port that you specify. The current input port is initially `console-i/o-port`, but Scheme provides procedures that let you change the current input port to be a file or string.

Similarly, many output procedures, such as `write-char` and `display`, write data to the current output port by default, or to a port that you specify. The current output port is initially `console-i/o-port`, but Scheme provides procedures that let you change the current output port to be a file or string.

All ports read or write only ASCII characters.

Every port is either an input port, an output port, or both. The following predicates distinguish all of the possible cases.

port? *object* procedure+

Returns `#t` if *object* is a port, otherwise returns `#f`.

input-port? *object* procedure

Returns `#t` if *object* is an input port, otherwise returns `#f`. Any object satisfying this predicate also satisfies `port?`.

output-port? *object* procedure

Returns `#t` if *object* is an output port, otherwise returns `#f`. Any object satisfying this predicate also satisfies `port?`.

i/o-port? *object* procedure+

Returns `#t` if *object* is both an input port and an output port, otherwise returns `#f`. Any object satisfying this predicate also satisfies `port?`, `input-port?`, and `output-port?`.

guarantee-port *object* procedure+
guarantee-input-port *object* procedure+
guarantee-output-port *object* procedure+
guarantee-i/o-port *object* procedure+

These procedures check the type of *object*, signalling an error of type `condition-type:wrong-type-argument` if it is not a port, input port, output port, or I/O port, respectively. Otherwise they return *object*.

The next five procedures return the runtime system's *standard ports*. All of the standard ports are dynamically bound by the REP loop; this means that when a new REP loop is started, for example by an error, each of these ports is dynamically bound to the I/O port of the REP loop. When the REP loop exits, the ports revert to their original values.

current-input-port procedure

Returns the current input port. This is the default port used by many input procedures. Initially, `current-input-port` returns the value of `console-i/o-port`.

current-output-port procedure

Returns the current output port. This is the default port used by many output procedures. Initially, `current-output-port` returns the value of `console-i/o-port`.

notification-output-port procedure+

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. Initially, `notification-output-port` returns the value of `console-i/o-port`.

trace-output-port procedure+

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. Initially, `trace-output-port` returns the value of `console-i/o-port`.

interaction-i/o-port procedure+

Returns an I/O port suitable for querying or prompting the user. The standard prompting procedures use this port by default (see Section 14.8 [Prompting], page 166). Initially, `interaction-i/o-port` returns the value of `console-i/o-port`.

with-input-from-port *input-port thunk* procedure+
with-output-to-port *output-port thunk* procedure+
with-notification-output-port *output-port thunk* procedure+
with-trace-output-port *output-port thunk* procedure+
with-interaction-i/o-port *i/o-port thunk* procedure+

Thunk must be a procedure of no arguments. Each of these procedures binds one of the standard ports to its first argument, calls *thunk* with no arguments, restores the port to its original value, and returns the result that was yielded by *thunk*. This temporary binding is performed the same way as dynamic binding of a variable, including the behavior in the presence of continuations (see Section 2.3 [Dynamic Binding], page 19).

`with-input-from-port` binds the current input port, `with-output-to-port` binds the current output port, `with-notification-output-port` binds the "notification" output port, `with-trace-output-port` binds the "trace" output port, and `with-interaction-i/o-port` binds the "interaction" I/O port.

set-current-input-port! *input-port* procedure+
set-current-output-port! *output-port* procedure+
set-notification-output-port! *output-port* procedure+
set-trace-output-port! *output-port* procedure+
set-interaction-i/o-port! *i/o-port* procedure+

Each of these procedures alters the binding of one of the standard ports and returns an unspecified value. The binding that is modified corresponds to the name of the procedure.

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 and OS/2, 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.

close-port *port* procedure+

Closes *port* and returns an unspecified value. If *port* is a file port, the file is closed.

close-input-port *port* procedure

Closes *port* and returns an unspecified value. *Port* must be an input port or an I/O port; if it is an I/O port, then only the input side of the port is closed.

close-output-port *port* procedure

Closes *port* and returns an unspecified value. *Port* must be an output port or an I/O port; if it is an I/O port, then only the output side of the port is closed.

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.

Any file can be opened in one of two modes, *normal* or *binary*. Normal mode is for accessing text files, and binary mode is for accessing other files. Unix does not distinguish these modes, but Windows and OS/2 do: in normal mode, their file ports perform *newline translation*, mapping between the carriage-return/linefeed sequence that terminates text lines in files, and the `#\newline` that terminates lines in Scheme. In binary mode, such ports do not perform newline translation. Unless otherwise mentioned, the procedures in this section open files in normal mode.

open-input-file *filename* procedure

Takes a filename referring to an existing file and returns an input port capable of delivering characters from the file. If the file cannot be opened, an error of type `condition-type:file-operation-error` is signalled.

open-output-file *filename* [*append?*] procedure

Takes a filename referring to an output file to be created and returns an output port capable of writing characters to a new file by that name. If the file cannot be opened, an error of type `condition-type:file-operation-error` is signalled.

The optional argument *append?* is an MIT 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+

Takes a filename referring to an existing file and returns an I/O port capable of both reading and writing the file. If the file cannot be opened, an error of type `condition-type:file-operation-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.

open-binary-input-file *filename* procedure+
open-binary-output-file *filename* [*append?*] procedure+
open-binary-i/o-file *filename* procedure+

These procedures open files in binary mode. In all other respects they are identical to `open-input-file`, `open-output-file`, and `open-i/o-file`, respectively.

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.

call-with-input-file *filename procedure* procedure
call-with-output-file *filename procedure* procedure

These procedures call *procedure* with one argument: the port obtained by opening the named file for input or output, respectively. If the file cannot be opened, an error of type `condition-type:file-operation-error` is signalled. If *procedure* returns, then the port is closed automatically and the value yielded by *procedure* is returned. If *procedure* does not return, then the port will not be closed automatically unless it is reclaimed by the garbage collector.[1]

call-with-binary-input-file *filename procedure* procedure+
call-with-binary-output-file *filename procedure* procedure+

These procedures open files in binary mode. In all other respects they are identical to `call-with-input-file` and `call-with-output-file`, respectively.

with-input-from-file *filename thunk* procedure
with-output-to-file *filename thunk* procedure

Thunk must be a procedure of no arguments. The file is opened for input or output, an input or output port connected to it is made the default value returned by `current-input-port` or `current-output-port`, and the *thunk* is called with no arguments. When the *thunk* returns, the port is closed and the previous default is restored. `with-input-from-file` and `with-output-to-file` return the value yielded by *thunk*. If an escape procedure is used to escape from the continuation of these procedures, their behavior is implementation-dependent; in that situation MIT Scheme leaves the files open.

with-input-from-binary-file *filename thunk* procedure+
with-output-to-binary-file *filename thunk* procedure+

These procedures open files in binary mode. In all other respects they are identical to `with-input-from-file` and `with-output-to-file`, respectively.

14.3 String Ports

This section describes the simplest kinds of ports: input ports that read their input from given strings, and output ports that accumulate their output and return it as a string. It also describes "truncating" output ports, which can limit the length of the resulting string to a given value.

string->input-port *string* [*start* [*end*]] procedure+

Returns a new string port that delivers characters from *string*. 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*`)`.

with-input-from-string *string thunk* procedure+

Thunk must be a procedure of no arguments. `with-input-from-string` creates a new input port that reads from *string*, makes that port the current input

[1] Because Scheme's escape procedures have unlimited extent, it is possible to escape from the current continuation but later to escape back in. 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-input-file` or `call-with-output-file`.

port, and calls *thunk*. When *thunk* returns, `with-input-from-string` restores the previous current input port and returns the result yielded by *thunk*.

```
(with-input-from-string "(a b c) (d e f)" read)  ⇒  (a b c)
```

Note: this procedure is equivalent to:

```
(with-input-from-port (string->input-port string) thunk)
```

with-string-output-port *procedure* procedure+

Procedure is called with one argument, an output port. The value yielded by *procedure* is ignored. When *procedure* returns, `with-string-output-port` returns the port's accumulated output as a newly allocated string.

with-output-to-string *thunk* procedure+

Thunk must be a procedure of no arguments. `with-output-to-string` creates a new output port that accumulates output, makes that port the default value returned by `current-output-port`, and calls *thunk* with no arguments. When *thunk* returns, `with-output-to-string` restores the previous default and returns the accumulated output as a newly allocated string.

```
(with-output-to-string
  (lambda ()
    (write 'abc)))                ⇒  "abc"
```

Note: this procedure is equivalent to:

```
(with-string-output-port
 (lambda (port)
   (with-output-to-port port thunk)))
```

with-output-to-truncated-string *k thunk* procedure+

Similar to `with-output-to-string`, except that the output is limited to *k* characters. If *thunk* attempts to write more than *k* characters, it will be aborted by invoking an escape procedure that returns from `with-output-to-truncated-string`.

The value of this procedure is a pair; the car of the pair is `#t` if *thunk* attempted to write more than *k* characters, and `#f` otherwise. The cdr of the pair is a newly allocated string containing the accumulated output.

This procedure is helpful for displaying circular lists, as shown in this example:

```
(define inf (list 'inf))
(with-output-to-truncated-string 40
  (lambda ()
    (write inf)))                 ⇒  (#f . "(inf)")
(set-cdr! inf inf)
(with-output-to-truncated-string 40
  (lambda ()
    (write inf)))
        ⇒  (#t . "(inf inf inf inf inf inf inf inf inf inf")
```

write-to-string *object* [*k*] procedure+

Writes *object* to a string output port, and returns the resulting newly allocated string. If *k* is supplied and not `#f`, this procedure is equivalent to

```
(with-output-to-truncated-string k
  (lambda ()
    (write object)))
```

otherwise it is equivalent to

```
(with-output-to-string
 (lambda ()
   (write object)))
```

14.4 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.

All optional arguments called *input-port*, if not supplied, default to the current input port.

read-char [*input-port*] procedure

Returns the next character available from *input-port*, updating *input-port* to point to the following character. If no more characters are available, an end-of-file object is returned.

In MIT Scheme, if *input-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.

peek-char [*input-port*] procedure

Returns the next character available from *input-port*, *without* updating *input-port* to point to the following character. If no more characters are available, an end-of-file object is returned.[2]

In MIT Scheme, if *input-port* is an interactive input port and no characters are immediately available, `peek-char` will hang waiting for input, even if the port is in non-blocking mode.

char-ready? [*input-port*] procedure

Returns `#t` if a character is ready on *input-port* and returns `#f` otherwise. If `char-ready?` returns `#t` then the next `read-char` operation on *input-port* is

[2] 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 *input-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.

guaranteed not to hang. If *input-port* is a file port at end of file then `char-ready?` returns `#t`.[3]

read [*input-port*] procedure

Converts external representations of Scheme objects into the objects themselves. `read` returns the next object parsable from *input-port*, updating *input-port* to point to the first character past the end of the written representation of the object. If an end of file is encountered in the input before any characters are found that can begin an object, `read` returns an end-of-file object. The *input-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.

eof-object? *object* procedure

Returns `#t` if *object* is an end-of-file object; otherwise returns `#f`.

read-char-no-hang [*input-port*] procedure+

If *input-port* can deliver a character without blocking, this procedure acts exactly like `read-char`, immediately returning that character. Otherwise, `#f` is returned, unless *input-port* is a file port at end of file, in which case an end-of-file object is returned. In no case will this procedure block waiting for input.

read-string *char-set* [*input-port*] procedure+

Reads characters from *input-port* until it finds a terminating character that is a member of *char-set* (see Section 5.6 [Character Sets], page 66) 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-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 eof 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`:

---

[3] `char-ready?` exists to make it possible for a program to accept characters from interactive ports without getting stuck waiting for input. Any input editors associated with such ports must make sure that characters whose existence has been asserted by `char-ready?` cannot be rubbed out. If `char-ready?` were to return `#f` at end of file, a port at end of file would be indistinguishable from an interactive port that has no ready characters.

```
(define (read-string char-set input-port)
  (let ((char (peek-char input-port)))
    (if (eof-object? char)
        char
        (list->string
         (let loop ((char char))
           (if (or (eof-object? char)
                   (char-set-member? char-set char))
               '()
               (begin
                 (read-char input-port)
                 (cons char
                       (loop (peek-char input-port))))))))))
```

read-line [*input-port*] procedure+

`read-line` reads a single line of text from *input-port*, and returns that line as a newly allocated string. The `#\newline` terminating the line, if any, is discarded and does not appear in the returned string.

This procedure ignores the blocking mode of the port, blocking unconditionally until it has read an entire line. If end of file is encountered before any characters are read, an end-of-file object is returned.

read-string! *string* [*start* [*end* [*input-port*]]] procedure+

`read-string!` fills the specified region of *string* with characters read from *input-port* until the region is full or else there are no more characters available from the port. The region is specified by *start* and *end*, which default to `0` and *string*'s length, respectively.

The returned value is the number of characters written to *string*. However, there are several interesting cases to consider:

- If `read-string!` is called when *input-port* is at "end-of-file", then the returned value is `0`. Note that "end-of-file" can mean a file port that is at the file's end, a string port that is at the string's end, or any other port that will never produce more characters.
- If *input-port* is an interactive port (e.g. a terminal), and one or more characters are immediately available, the region is filled using the available characters. The procedure then returns immediately, even if the number of characters is less than the size of the region. The returned value is the number of characters actually written.
- If *input-port* is an interactive port and no characters are immediately available, the result of the operation depends on the blocking mode of the port. If the port is in non-blocking mode, `read-string!` immediately returns the value `#f`. Otherwise, the operation blocks until a character is available. As soon as at least one character is available, the region is filled with the available characters, and the number of characters written is immediately returned (even if the number available is insufficient to fill the region).

The importance of `read-string!` is that it is both flexible and extremely fast.

14.5 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`.

All optional arguments called *output-port*, if not supplied, default to the current output port.

write-char *char* [*output-port*] procedure

Writes *char* (the character itself, not a written representation of the character) to *output-port*, performs discretionary output flushing, and returns an unspecified value.

write-string *string* [*output-port*] procedure+

Writes *string* to *output-port*, performs discretionary output flushing, and returns an unspecified value. This is equivalent to writing the contents of *string*, one character at a time using `write-char`, except that it is usually much faster.

write *object* [*output-port*] procedure

Writes a written representation of *object* to *output-port*, and returns an unspecified value. If *object* has a standard external representation, then the written representation generated by `write` shall be parsable by `read` into an equivalent object. Thus strings that appear in the written representation are enclosed in doublequotes, and within those strings backslash and doublequote are escaped by backslashes. `write` performs discretionary output flushing and returns an unspecified value.

display *object* [*output-port*] procedure

Writes a representation of *object* to *output-port*. Strings appear in the written representation as if written by `write-string` instead of by `write`. Character objects appear in the representation as if written by `write-char` instead of by `write`. `display` performs discretionary output flushing and returns an unspecified value.[4]

[4] `write` is intended for producing machine-readable output and `display` is for producing human-readable output.

newline [*output-port*] procedure

Writes an end-of-line to *output-port*, performs discretionary output flushing, and returns an unspecified value. Equivalent to `(write-char #\newline` *output-port*`)`.

fresh-line [*output-port*] procedure+

Most output ports are able to tell whether or not they are at the beginning of a line of output. If *output-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 *output-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* [*output-port*] procedure+

Like `write`, except that it writes an end-of-line to *output-port* before writing *object*'s representation. This procedure performs discretionary output flushing and returns an unspecified value.

flush-output [*output-port*] procedure+

If *output-port* is buffered, this causes the contents of its buffer to be written to the output device. Otherwise it has no effect. Returns an unspecified value.

beep [*output-port*] procedure+

Performs a "beep" operation on *output-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 [*output-port*] procedure+

"Clears the screen" of *output-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* [*output-port* [*as-code?*]] procedure+

`pp` prints *object* in a visually appealing and structurally revealing manner on *output-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 variables may be dynamically bound to change the behavior of the `write` and `display` procedures.

unparser-radix variable+

This variable 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`. If `*unparser-radix*` is not `10`, numbers are prefixed to indicate their radix.

unparser-list-breadth-limit variable+

This variable 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 variable must be an exact non-negative integer, or `#f` meaning no limit; the default is `#f`.

```
(fluid-let ((*unparser-list-breadth-limit* 4))
  (write-to-string '(a b c d)))
                              ⇒ "(a b c d)"
(fluid-let ((*unparser-list-breadth-limit* 4))
  (write-to-string '(a b c d e)))
                              ⇒ "(a b c d ...)"
```

unparser-list-depth-limit variable+

This variable 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 variable must be an exact non-negative integer, or `#f` meaning no limit; the default is `#f`.

```
(fluid-let ((*unparser-list-depth-limit* 4))
  (write-to-string '((((a))) b c d)))
                              ⇒ "((((a))) b c d)"
(fluid-let ((*unparser-list-depth-limit* 4))
  (write-to-string '(((((a)))) b c d)))
                              ⇒ "((((...))) b c d)"
```

unparser-string-length-limit variable+

This variable 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 variable must be an exact non-negative integer, or `#f` meaning no limit; the default is `#f`.

```
(fluid-let ((*unparser-string-length-limit* 4))
  (write-to-string "abcd"))
                              ⇒ "\"abcd\""
(fluid-let ((*unparser-string-length-limit* 4))
  (write-to-string "abcde"))
                              ⇒ "\"abcd...\""
```

unparse-with-maximum-readability? variable+

This variable, 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.

14.6 Format

The procedure `format` is very useful for producing nicely formatted text, producing good-looking messages, and so on. MIT Scheme's implementation of `format` is similar to that of Common Lisp, except that Common Lisp defines many more directives.[5]

`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 *arguments* 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 14.5 [Output Procedures], page 160).

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.

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.

[5] This description of `format` is adapted from *Common Lisp, The Language*, second edition, section 22.3.3.

`~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 make 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.

14.7 Custom Output

MIT Scheme provides hooks for specifying that certain kinds of objects have special written representations. There are no restrictions on the written representations, but only a few kinds of objects may have custom representation specified for them, specifically: records (see Section 10.4 [Records], page 108), vectors that have special tags in their zeroth elements (see Chapter 8 [Vectors], page 95), and pairs that have special tags in their car fields (see Chapter 7 [Lists], page 81). There is a different procedure for specifying the written representation of each of these types.

set-record-type-unparser-method! *record-type unparser-method* procedure+

Changes the unparser method of the type represented by *record-type* to be *unparser-method*, and returns an unspecified value. Subsequently, when the unparser encounters a record of this type, it will invoke *unparser-method* to generate the written representation.

unparser/set-tagged-vector-method! *tag unparser-method* procedure+

Changes the unparser method of the vector type represented by *tag* to be *unparser-method*, and returns an unspecified value. Subsequently, when the unparser encounters a vector with *tag* as its zero-th element, it will invoke *unparser-method* to generate the written representation.

unparser/set-tagged-pair-method! *tag unparser-method* procedure+

Changes the unparser method of the pair type represented by *tag* to be *unparser-method*, and returns an unspecified value. Subsequently, when the unparser encounters a pair with *tag* in its car field, it will invoke *unparser-method* to generate the written representation.

An *unparser method* is a procedure that is invoked with two arguments: an unparser state and an object. An unparser method generates a written representation for the object, writing it to the output port specified by the unparser state. The value yielded by an unparser method is ignored. Note that an unparser state is not an output port, rather it is an object that contains an output port as one of its components. Application programs generally do not construct or examine unparser state objects, but just pass them along.

There are two ways to create an unparser method (which is then registered by one of the above procedures). The first, and easiest, is to use `standard-unparser-method`. The second is to define your own method using the procedure `with-current-unparser-state`. We encourage the use of the first method, as it results in a more uniform appearance for objects. Many predefined datatypes, for example procedures and environments, already have this appearance.

standard-unparser-method *name procedure* procedure+

Returns a standard unparser method. *Name* may be any object, and is used as the name of the type with which the unparser method is associated; *name* is usually a symbol. *Procedure* must be `#f` or a procedure of two arguments.

If *procedure* is `#f`, the returned method generates an external representation of this form:

```
#[name hash]
```

Here *name* is the external representation of the argument *name*, as generated by `write`,[6] and *hash* is the external representation of an exact non-negative integer unique to the object being printed (specifically, it is the result of calling `hash` on the object). Subsequently, the expression

---

[6] Except that if the argument *name* is a string, its external representation is generated by `write-string`.

`#@`*hash*

is notation for the object.

If *procedure* is supplied, the returned method generates a slightly different external representation:

`#[`*name hash output*`]`

Here *name* and *hash* are as above, and *output* is the output generated by *procedure*. The representation is constructed in three stages:

1. The first part of the format (up to *output*) is written to the output port specified by the unparser state. This is `"#["`, *name*, `" "`, and *hash*.
2. *Procedure* is invoked on two arguments: the object and an output port.
3. The closing bracket is written to the output port.

The following procedure is useful for writing more general kinds of unparser methods.

with-current-unparser-state *unparser-state procedure* procedure+

This procedure calls *procedure* with one argument, the output port from *unparser-state*. Additionally, it arranges for the remaining components of *unparser-state* to be given to the printer when they are needed. The *procedure* generates some output by writing to the output port using the usual output operations, and the value yielded by *procedure* is returned from `with-current-unparser-state`.

The port passed to *procedure* should only be used within the dynamic extent of *procedure*.

14.8 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 must be an I/O port if given. 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 `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.

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.

14.9 Port Primitives

This section describes the low-level operations that can be used to build and manipulate I/O ports.

The purpose of these operations is twofold: to allow programmers to construct new kinds of I/O ports, and to provide faster I/O operations than those supplied by the standard high level procedures. The latter is useful because the standard I/O operations provide defaulting and error checking, and sometimes other features, which are often unnecessary. This interface provides the means to bypass such features, thus improving performance.

The abstract model of an 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 *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.

14.9.1 Port Types

The procedures in this section provide means for constructing port types with standard and custom operations, and accessing their operations.

make-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.

port-type? *object* procedure+
input-port-type? *object* procedure+
output-port-type? *object* procedure+
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`.

port-type/operations *port-type* procedure+

Returns a newly allocated list containing all of the operations implemented by *port-type*. Each element of the list is a list of two elements — the name and its associated operation.

port-type/operation-names *port-type* procedure+

Returns a newly allocated list whose elements are the names of the operations implemented by *port-type*.

port-type/operation *port-type symbol* procedure+

Returns the operation named *symbol* in *port-type*. If *port-type* has no such operation, returns `#f`.

14.9.2 Constructors and Accessors for 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-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*.

port/type *port* procedure+

Returns the port type of *port*.

port/state *port* procedure+

Returns the state component of *port*.

set-port/state! *port object* procedure+

Changes the state component of *port* to be *object*. Returns an unspecified value.

port/operation *port symbol* procedure+

Equivalent to

```
(port-type/operation (port/type port) symbol)
```

port/operation-names *port* procedure+

Equivalent to

```
(port-type/operation-names (port/type port))
```

make-eof-object *input-port* procedure+

Returns an object that satisfies the predicate `eof-object?`. This is sometimes useful when building input ports.

14.9.3 Input Port Operations

This section describes the standard operations on input ports. Following that, some useful custom operations are described.

read-char *input-port* operation+ on input port

Removes the next character available from *input-port* and returns it. If *input-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 *input-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 *input-port* operation+ on input port

Reads the next character available from *input-port* and returns it. The character is *not* removed from *input-port*, and a subsequent attempt to read from the port will get that character again. In other respects this operation behaves like `read-char`.

discard-char *input-port* operation+ on input port

Discards the next character available from *input-port* and returns an unspecified value. In other respects this operation behaves like `read-char`.

char-ready? *input-port k* operation+ on input port

`char-ready?` returns `#t` if at least one character is available to be read from *input-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 *input-port char-set* operation+ on input port
discard-chars *input-port char-set* operation+ on input port

These operations are like `read-char` and `discard-char`, except that they read or discard multiple characters at once. This can have a marked performance improvement on buffered input ports. All characters up to, but excluding, the first character in *char-set* (or end of file) are read from *input-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 *input-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 *input-port string start end* operation+ on input port

Reads characters from *input-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 *input-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 *input-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 *input-port* procedure+
input-port/peek-char *input-port* procedure+
input-port/discard-char *input-port* procedure+
input-port/char-ready? *input-port k* procedure+
input-port/read-string *input-port char-set* procedure+
input-port/discard-chars *input-port char-set* procedure+
input-port/read-substring *input-port string start end* procedure+

Each of these procedures invokes the respective operation on *input-port*. For example, the following are equivalent:

```
(input-port/read-char input-port)
((input-port/operation input-port 'read-char) 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? *input-port* operation+ on input port

Returns `#t` if *input-port* is known to be at end of file, otherwise it returns `#f`.

chars-remaining *input-port* operation+ on input port

Returns an estimate of the number of characters remaining to be read from *input-port*. This is useful only when *input-port* is a file port in binary mode; in other cases, it returns `#f`.

buffered-input-chars *input-port* operation+ on input port

Returns the number of unread characters that are stored in *input-port*'s buffer. This will always be less than or equal to the buffer's size.

input-buffer-size *input-port* operation+ on input port

Returns the maximum number of characters that *input-port*'s buffer can hold.

set-input-buffer-size *input-port size* operation+ on input port

Resizes *input-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.

14.9.4 Output Port Operations

This section describes the standard operations on output ports. Following that, some useful custom operations are described.

write-char *output-port char* operation+ on output port

Writes *char* to *output-port* and returns an unspecified value.

write-substring *output-port string start end* operation+ on output port

Writes the substring specified by *string*, *start*, and *end* to *output-port* and returns an unspecified value. Equivalent to writing the characters of the substring, one by one, to *output-port*, but is implemented very efficiently.

fresh-line *output-port* operation+ on output port

Most output ports are able to tell whether or not they are at the beginning of a line of output. If *output-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 *output-port* is not such a port, and end-of-line is unconditionally written to the port. Returns an unspecified value.

flush-output *output-port* operation+ on output port

If *output-port* is buffered, this causes its buffer to be written out. Otherwise it has no effect. Returns an unspecified value.

discretionary-flush-output *output-port* operation+ on 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 *output-port char* procedure+
output-port/write-substring *output-port string start end* procedure+
output-port/fresh-line *output-port* procedure+
output-port/flush-output *output-port* procedure+
output-port/discretionary-flush-output *output-port* procedure+

Each of these procedures invokes the respective operation on *output-port*. For example, the following are equivalent:

```
(output-port/write-char output-port char)
((output-port/operation output-port 'write-char)
 output-port char)
```

output-port/write-string *output-port string* procedure+

Writes *string* to *output-port*. Equivalent to

```
(output-port/write-substring output-port
                             string
                             0
                             (string-length string))
```

The following custom operations are generally useful.

buffered-output-chars *output-port* — operation+ on output port

Returns the number of unwritten characters that are stored in *output-port*'s buffer. This will always be less than or equal to the buffer's size.

output-buffer-size *output-port* — operation+ on output port

Returns the maximum number of characters that *output-port*'s buffer can hold.

set-output-buffer-size *output-port size* — operation+ on output port

Resizes *output-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 *output-port* — operation+ on output port

Returns an exact positive integer that is the width of *output-port* in characters. If *output-port* has no natural width, e.g. if it is a file port, `#f` is returned.

y-size *output-port* — operation+ on output port

Returns an exact positive integer that is the height of *output-port* in characters. If *output-port* has no natural height, e.g. if it is a file port, `#f` is returned.

output-port/x-size *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 *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.

14.9.5 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 independent 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.

port/input-blocking-mode *port* procedure+

Returns the input blocking mode of *port*.

port/set-input-blocking-mode *port mode* procedure+

Changes the input blocking mode of *port* to be *mode*. Returns an unspecified value.

port/with-input-blocking-mode *port mode thunk* procedure+

Thunk must be a procedure of no arguments. `port/with-input-blocking-mode` binds the input blocking mode of *port* to be *mode*, executes *thunk*, restores the input blocking mode of *port* to what it was when `port/with-input-blocking-mode` was called, and returns the value that was yielded by *thunk*. This binding is performed by `dynamic-wind`, which guarantees that the input blocking mode is restored if *thunk* escapes from its continuation.

port/output-blocking-mode *port* procedure+

Returns the output blocking mode of *port*.

port/set-output-blocking-mode *port mode* procedure+

Changes the output blocking mode of *port* to be *mode*. Returns an unspecified value.

port/with-output-blocking-mode *port mode thunk* procedure+

Thunk must be a procedure of no arguments. `port/with-output-blocking-mode` binds the output blocking mode of *port* to be *mode*, executes *thunk*, restores the output blocking mode of *port* to what it was when `port/with-output-blocking-mode` was called, and returns the value that was yielded by *thunk*. This binding is performed by `dynamic-wind`, which guarantees that the output blocking mode is restored if *thunk* escapes from its continuation.

14.9.6 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 independent 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 rubout processing 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`. Additionally, the value `#f` represents "no mode"; it is the terminal mode of a port that is not a terminal. An argument called *mode* must be one of these three values. 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.

port/input-terminal-mode *port* procedure+

Returns the input terminal mode of *port*.

port/set-input-terminal-mode *port mode* procedure+

Changes the input terminal mode of *port* to be *mode*. Returns an unspecified value.

port/with-input-terminal-mode *port mode thunk* procedure+

Thunk must be a procedure of no arguments. `port/with-input-terminal-mode` binds the input terminal mode of *port* to be *mode*, executes *thunk*, restores the input terminal mode of *port* to what it was when `port/with-input-terminal-mode` was called, and returns the value that was yielded by *thunk*. This binding is performed by `dynamic-wind`, which guarantees that the input terminal mode is restored if *thunk* escapes from its continuation.

port/output-terminal-mode *port* procedure+

Returns the output terminal mode of *port*.

port/set-output-terminal-mode *port mode* procedure+

Changes the output terminal mode of *port* to be *mode*. Returns an unspecified value.

port/with-output-terminal-mode *port mode thunk* procedure+

Thunk must be a procedure of no arguments. `port/with-output-terminal-mode` binds the output terminal mode of *port* to be *mode*, executes *thunk*, restores the output terminal mode of *port* to what it was when `port/with-output-terminal-mode` was called, and returns the value that was yielded by *thunk*. This binding is performed by `dynamic-wind`, which guarantees that the output terminal mode is restored if *thunk* escapes from its continuation.

15 File-System Interface

The Scheme standard provides a simple mechanism for reading and writing files: file ports. MIT Scheme provides additional tools for dealing with other aspects of the file 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.
- A facility for reading the contents of a directory.

15.1 Pathnames

MIT 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 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 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 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 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 `*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 be the same as '`/usr/`'. In cases of uncertainty the behavior is conservative, returning the original or a partly simplified pathname.

---

[1] This introduction is adapted from *Common Lisp, The Language*, second edition, section 23.1.

```
(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.

device Corresponds to the "device" or "file structure" concept in many host file systems: the name of a (logical or physical) device containing files.

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).

name The name of a group of files that can be thought of as conceptually the "same" file.

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.

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.

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, Windows, and OS/2 file systems usually do not support version numbers, so the version component for such a host might be `unspecific`.[2]

Each component in a pathname is typically one of the following (with some exceptions that will be described below):

a string
: This is a *literal component*. It is considered to be fully specified.

`#f`
: This is a *missing component*. It is considered to be unspecified.

`wild`
: This is a *wildcard component*. It is useful only when the pathname is being used with the directory reader, where it means that the pathname component matches anything.

`unspecific`
: This is an *unspecifiable component*. It is treated the same as a missing component except that it is not considered to be missing for purposes of merging or defaulting components.

The host, directory, and version pathname components are exceptions to these rules in that they may never be strings, although the values `#f`, `wild`, and `unspecific` are allowed with their usual meanings. Here are the other values allowed for these components:

- The host component is of an implementation defined type which may be tested for using the `host?` predicate.
- A directory, if it is not one of the above values, must be 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 or the symbol `wild` (each with the same meaning as described above), or `up`, which means the next directory is the "parent" of the previous one. `up` corresponds to the file '`..`' in unix file systems.

  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 the system supports "flat" directories, i.e. a global set of directories with no

[2] This description is adapted from *Common Lisp, The Language*, second edition, section 23.1.1.

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. Fortunately few such file systems are in common use today.

- A version component may take the following values: an exact positive integer, which is a literal component; the symbol `newest`, which means to choose the largest available version number for that file; or the symbol `oldest`, which means to choose the smallest version number. In the future some other possible values may be added, e.g. `installed`. Note that in the current implementation there are no file systems that 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.

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 `*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.

default-pathname-defaults variable+

This is the default pathname-defaults pathname; if any pathname primitive that needs a set of defaults is not given one, it uses this one. `set-working-directory-pathname!` sets this variable to a new value, computed by merging the new working directory with the variable's old value.

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 `*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 `*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 and OS/2, 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 `geteuid` system call. The resulting user name is passed to the `getpwnam` system call to obtain the home directory.

Under OS/2, several heuristics are tried to find the user's home directory. First, if the environment variable `HOME` is defined, that is the home directory. If `HOME` is undefined, but the `USERDIR` and `USER` environment variables are defined and the directory '`%USERDIR%\%USER%`' exists, then it is used. Failing that, if the directory '`%USER%`' exists on the OS/2 system drive, then it is used. As a last resort, the OS/2 system drive is the home directory.

Like OS/2, the Windows implementation uses heuristics based on environment variables. 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 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 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 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.

Additionally, `set-working-directory-pathname!` modifies the value of `*default-pathname-defaults*` by merging the new working directory into it.

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 temporarily rebinds the current working directory to *filename*, invokes *thunk* (a procedure of no arguments), then restores the previous working directory and returns the value yielded by *thunk*. *Filename* is coerced to a pathname using `pathname-as-directory`. In addition to binding the working directory, `with-working-directory-pathname` also binds the variable `*default-pathname-defaults*`, merging the old value of that variable with the new working directory pathname. Both bindings are performed in exactly the same way as dynamic binding of a variable (see Section 2.3 [Dynamic Binding], page 19).

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+

Returns `#t` if *filename* is an existing file or directory; otherwise returns `#f`. In operating systems that support symbolic links, if the file is a symbolic link, this procedure tests the existence of the file linked to, not the link itself.

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*. If *target-filename* exists prior to this procedure's invocation, it is deleted before the new output file is created.

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.

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 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.

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 writable 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 OS/2 or Windows, the following directories on the system drive: '`\temp`', '`\tmp`', or '`\`'.
- Under OS/2 or Windows, the current directory, as specified by `*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.

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-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-writable? *filename* procedure+

Returns `#t` if *filename* names a file that can be opened for output; i.e. a *writable* 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 OS/2, an executable file has one of the file extensions '`.exe`', '`.com`', '`.cmd`', or '`.bat`'. 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 "writable", 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 OS/2 and 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, but typically it contains bits that indicate what users and processes are allowed to read, write, or execute the file. 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*.

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*.

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 OS/2 and 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 and OS/2, 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 OS/2 and 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.

The following additional accessor is defined under OS/2:

file-attributes/allocated-length *attributes* procedure+

The allocated length of the file, which can be larger than the length of the file due to fixed-length allocation units.

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 OS/2 and Windows implementations support "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*.

16 Error System

The MIT 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 194). 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 194). 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 197). 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 209) 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 *arguments* are ignored. If *reason* is a condition type, then a new instance of this type is generated and signalled; the *arguments* 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 *arguments*.

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 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 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 variable+

This variable controls the behavior of the procedure `standard-error-handler`, and hence `error`. It is intended to be bound with `fluid-let` 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 variable+

This variable controls the behavior of the procedure `standard-warning-handler`, and hence `warn`. It is intended to be bound with `fluid-let` 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 199.

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 199). 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 *arguments* 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 108). 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 199. 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 199. 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 209).

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 196.

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 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-error *filename 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.

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 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-device-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 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-opereration 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.

17.10 Win32 Graphics

MIT Scheme supports graphics on Microsoft Windows 95, Windows 98, and Windows NT. In addition to the usual operations, there are operations to control the size, position and colors of a graphics window. Win32 devices support images, which are implemented as device independent bitmaps (DIBs).

The Win32 graphics device type is implemented as a top level window. `graphics-enable-buffering` is implemented and gives a 2x to 4x speedup on many graphics operations. As a convenience, when buffering is enabled clicking on the graphics window's title bar effects a `graphics-flush` operation. The user has the benefit of the increased performance and the ability to view the progress in drawing at the click of a mouse button.

17.10.1 Win32 Graphics Type

Win32 graphics devices are created by specifying the symbol `win32` as the *graphics-device-type* argument to `make-graphics-device`. The Win32 graphics device type is implemented as a top-level window and supports color drawing in addition to the standard Scheme graphics operations.

Graphics devices are opened as follows:

```
(make-graphics-device 'win32 #!optional width height palette)
```

where *width* and *height* specify the size, in pixels, of the drawing area in the graphics window (i.e. excluding the frame). *Palette* determines the colors available for drawing in the window.

When a color is specified for drawing, the nearest color available in the palette is used. Permitted values for *palette* are

`'grayscale`
: The window allocates colors from a grayscale palette of approximately 236 shades of gray.

`'grayscale-128`
: The window allocates colors from a grayscale palette of 128 shades of gray.

`'standard`
: The standard palette has good selection of colors and grays.

`#f` or `'system`
: The colors available are those in the system palette. There are usually 16 to 20 colors in the system palette and these are usually sufficent for simple applications like line drawings and x-vs-y graphs of mathematical functions. Drawing with the system palette can be more efficient.

If *palette* is not specified then the `standard` palette is used.

17.10.2 Custom Operations for Win32 Graphics

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 win32-graphics-device
set-foreground-color *color-name* — operation+ on win32-graphics-device

These operations change the colors associated with a window. *Color-name* must be of one of the valid color specification forms listed below. `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.

The foreground color affects the drawing of text, points, lines, ellipses and filled polygons.

Colors are specified in one of three ways:

An integer
: This is the Win32 internal RGB value.

By name
: A limited number of names are understood by the system. Names are strings, e.g. `"red"`, `"blue"`, `"black"`. More names can be registered with the `define-color` operation.

RGB (Red-Green-Blue) triples
: A triple is either a vector or list of three integers in the range 0–255 inclusive which specify the intensity of the red, green and blue components of the color. Thus `#(0 0 0)` is black, `(0 0 128)` is dark blue and `#(255 255 255)` is white.

If the color is not available in the graphics device then the nearest available color is used instead.

define-color *name spec* operation+ on win32-graphics-device

Define the string *name* to be the color specified by *spec*. *Spec* may be any acceptable color specification. Note that the color names defined this way are available to any Win32 graphics device, and the names do *not* have to be defined for each device.

Color names defined by this interface may also be used when setting the colors of the Scheme console window, or the colors of Edwin editor windows.

find-color *name* operation+ on win32-graphics-device

Looks up a color previously defined by `define-color`. This returns the color in its most efficient form for operations `set-foreground-color` or `set-background-color`.

draw-ellipse *left top right bottom* operation+ on win32-graphics-device

Draw an ellipse. *Left*, *top*, *right* and *bottom* indicate the coordinates of the bounding rectangle of the ellipse. Circles are merely ellipses with equal width and height. Note that the bounding rectangle has horizontal and vertical sides. Ellipses with rotated axes cannot be drawn. The rectangle applies to the center of the line used to draw the ellipse; if the line width has been set to greater than 1 then the ellipse will spill outside the bounding rectange by half of the line width.

fill-polygon *points* operation+ on win32-graphics-device

Draws a filled polygon using the current foreground color. *Points* is a vector of real numbers. The numbers are in the order x1 y1 x2 y2 ... xn yn. For example,

```
(graphics-operation device 'fill-polygon #(0 0 0 1 1 0))
```

draws a solid triangular region between the points (0, 0), (0, 1) and (1, 0).

load-bitmap *pathname* operation+ on win32-graphics-device

The graphics device contents and size are initialized from the windows bitmap file specified by *pathname*. If no file type is supplied then a `".BMP"` extension is added. If a clip rectangle is in effect when this procedure is called, it is necessary to redefine the clip rectangle afterwards.

save-bitmap *pathname* operation+ on win32-graphics-device

The graphics device contents are saved as a bitmap to the file specified by *pathname*. If no file type is supplied then a `".BMP"` extension is added. The saved bitmap may be incorporated into documents or printed.

move-window *x y* operation+ on win32-graphics-device

The graphics device window is moved to the screen position specified by *x* and *y*.

resize-window *width height* operation+ on win32-graphics-device

The graphics device window is resized to the specified *width* and *height* in device coordinates (pixels). If a clip rectangle is in effect when this procedure is called, it is necessary to redefine the clip rectangle afterwards.

set-line-width *width* operation+ on win32-graphics-device

This operation sets the line width for future drawing of lines, points and ellipses. It does not affect existing lines and has no effect on filled polygons. The line width is specified in device units. The default and initial value of this parameter is 1 pixel.

set-window-name *name* operation+ on win32-graphics-device

This sets the window title to the string *name*. The window is given the name `"Scheme Graphics"` at creation.

set-font *handle* operation+ on win32-graphics-device

Sets the font for drawing text. Currently not well supported. If you can get a Win32 font handle it can be used here.

copy-area *source-x-left source-y-top width height destination-x-left destination-y-top* operation+ on win32-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*.

17.11 OS/2 Graphics

MIT Scheme supports graphics under the OS/2 Presentation Manager in OS/2 version 2.1 and later. The OS/2 graphics device type is implemented as a top level window. In addition to the usual operations, there are operations to control the size, position, and colors of a graphics window. OS/2 graphics devices support images, which are implemented as memory presentation spaces.

The custom graphics operations defined in this section are invoked using the procedure `graphics-operation`. For example,

```
(graphics-operation device 'set-foreground-color "blue")
```

17.11.1 OS/2 Graphics Type

OS/2 graphics devices are created by specifying the symbol `os/2` as the *graphics-device-type* argument to `make-graphics-device`. The OS/2 graphics device type is implemented as a top-level window and supports color drawing in addition to the standard Scheme graphics operations.

Graphics devices are opened as follows:

```
(make-graphics-device 'os/2 #!optional width height)
```

where *width* and *height* specify the size, in pixels, of the drawing area in the graphics window (i.e. excluding the frame).

17.11.2 Color Operations for OS/2 Graphics

These operations control the colors used when drawing on an OS/2 graphics device.

color? — operation+ on os2-graphics-device

This operation returns `#t` if the display supports color.

set-background-color *color-name* — operation+ on os2-graphics-device
set-foreground-color *color-name* — operation+ on os2-graphics-device

These operations change the colors associated with a window. *Color-name* must be one of the valid color specification forms listed below. `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.

The foreground color affects the drawing of text, points, and lines. Colors are specified in one of these ways:

An integer between `0` and `#xffffff` inclusive
: This is the OS/2 internal RGB value.

By name
: A limited number of names are understood by the system. Names are strings, e.g. `"red"`, `"blue"`, `"black"`. More names can be registered with the `define-color` operation.

RGB (Red-Green-Blue) triples
: A triple is a list of three integers between `0` and `#xff` inclusive which specify the intensity of the red, green and blue components of the color. Thus `(0 0 0)` is black, `(0 0 128)` is dark blue and `(255 255 255)` is white.

If the color is not available in the graphics device then the nearest available color is used instead.

define-color *name spec* — operation+ on os2-graphics-device

Define the string *name* to be the color specified by *spec*. *Spec* may be any acceptable color specification. Note that the color names defined this way are available to any OS/2 graphics device, and the names do *not* have to be defined for each device.

Color names defined by this interface may also be used when setting the colors of the Scheme console window, or the colors of Edwin editor windows.

find-color *name* — operation+ on os2-graphics-device

Looks up a color previously defined by `define-color`. This returns the color in its most efficient form for operations `set-foreground-color` or `set-background-color`.

17.11.3 Window Operations for OS/2 Graphics

These operations control the window that contains the OS/2 graphics device. They provide facilities to change the window's size and position; to raise and lower the window relative to other windows on the desktop; to hide or minimize the window, and to restore it from the hidden or minimized state; to activate or deactivate the window (that is, control the keyboard focus); and to control the text that appears in the window's title bar.

window-position operation+ on os2-graphics-device

This operation returns the position of the graphics-device window on the desktop. The position is returned as two values (see Section 12.3 [Continuations], page 143), which are the x and y coordinates of the position. These coordinates are in units of pels (pixels), and measure the distance between the lower left hand corner of the desktop and the lower left hand corner of the graphics device window's frame.

set-window-position *x y* operation+ on os2-graphics-device

The graphics-device window is moved to the screen position specified by *x* and *y*. The coordinates *x* and *y* are in units of pels (pixels), and measure the distance between the lower left hand corner of the desktop and the lower left hand corner of the graphics device window's frame.

window-size operation+ on os2-graphics-device

This operation returns the size of the client area of the graphics-device window. The client area is the part of the window that you draw on; it does not include the window frame, title bar, etc. The size is returned as two values (see Section 12.3 [Continuations], page 143), which are the width and height of the client area in units of pels (pixels).

set-window-size *width height* operation+ on os2-graphics-device

This operation sets the size of the client area of the graphics-device window to the specified *width* and *height*, which are in units of pels (pixels). The client area is the part of the window that you draw on; it does not include the window frame, title bar, etc.

window-frame-size operation+ on os2-graphics-device

This operation returns the size of the graphics-device window's frame. This includes the client area, as well as the border, title bar, etc. The size is returned as two values (see Section 12.3 [Continuations], page 143), which are the width and height of the frame in units of pels (pixels).

The frame size is useful in conjunction with the window position and the desktop size to determine relative placement of the window or to guarantee that the entire window is visible on the desktop.

desktop-size operation+ on os2-graphics-device

This operation returns the size of the OS/2 desktop. The size is returned as two values (see Section 12.3 [Continuations], page 143), which are the width and height of the frame in units of pels (pixels).

raise-window operation+ on os2-graphics-device

This operation raises the graphics-device window so that it is on top of any other windows on the desktop.

lower-window operation+ on os2-graphics-device

This operation lowers the graphics-device window so that it is below all other windows on the desktop.

hide-window operation+ on os2-graphics-device

This operation hides the graphics-device window. The window disappears from the desktop, but still appears in the window list.

minimize-window operation+ on os2-graphics-device

This operation minimizes the graphics-device window. The window disappears from the desktop, but still appears in the window list. Depending on how you have configured your desktop, the window may appear as an icon, either on the desktop or in the minimized window viewer.

maximize-window operation+ on os2-graphics-device

This operation maximizes the graphics-device window. This causes the window to fill the entire desktop.

restore-window operation+ on os2-graphics-device

This operation restores the graphics-device window to its normal state. If the window is hidden or minimized, it is shown again, at its former position on the desktop. If the window is maximized, it is returned to its normal size.

activate-window operation+ on os2-graphics-device

This operation makes the graphics-device window be the active window. This causes the window to be put in front of all other windows on the desktop, highlights its frame, and gives it the keyboard focus.

deactivate-window operation+ on os2-graphics-device

This operation deactivates the graphics-device window if it was active (otherwise it has no effect). This causes some other window to be chosen to be active in its place.

set-window-title *title* operation+ on os2-graphics-device

This operation changes the text that appears in the graphics device window's title bar. The new text is given by *title*, which must be a string.

17.11.4 Event Operations for OS/2 Graphics

These operations allow you to read some of the events that are generated by the Presentation Manager and put in the message queue of a graphics-device window.

read-button operation+ on os2-graphics-device

This operation waits for the user to push a mouse button inside the client area of the graphics-device window. It then returns four values (see Section 12.3

[Continuations], page 143) which are: the button number; the x and y coordinates of the mouse pointer at the time the button was pressed, in pels (pixels) relative to the lower left hand corner of the client area; and the graphics device that the mouse pointer was over at the time the button was pressed.

Note that this operation only works when button events are selected (which is the default).

select-user-events *mask* operation+ on os2-graphics-device

This operation sets the event-selection mask for the graphics device to *mask*. The event-selection mask is an exact non-negative integer that specifies which types of incoming events are to be saved in the user-event queue for later retrieval by the `read-user-event` operation. The mask is specified by setting the bits corresponding to the event types that you are interested in, as follows:

```
Number  Mask    Description
------  -----   -----------
0       #x001   Button press/release
1       #x002   Close (close the window) [WM_CLOSE]
2       #x004   Focus change [WM_SETFOCUS]
3       #x008   Key press/release [WM_CHAR]
4       #x010   Paint [WM_PAINT]
5       #x020   Size change [WM_SIZE]
6       #x040   Visibility change [WM_SHOW]
7       #x080   Command [WM_COMMAND]
8       #x100   Help [WM_HELP]
9       #x200   Mouse-move [WM_MOUSEMOVE]
```

Note that this operation does not affect any events that are already in the user-event queue. Changing the mask only affects what events will be added to the queue in the future.

read-user-event operation+ on os2-graphics-device

This operation returns the next user event available from the user-event queue. If there are no events in the queue, the operation waits for an event to arrive before returning.

An event is a vector whose first element is the event-type number, whose second element is the graphics device that the event refers to, and whose remaining elements provide information about the event. Here is a table of the possible event types and their vector layout:

`#(0` *device number type x y flags*`)`

A button event. *Number* is the button number, for example button number `0` is usually the left mouse button, `1` is usually the right button, etc. *Type* specifies what occurred: `0` means the button was pressed, `1` means the button was released, `2` means the button was clicked, and `3` means the button was double clicked. *X* and *y* are the position of the mouse pointer at the time of the event, in units of pels (pixels) measured from the lower left corner of the client area of the associated window. Finally, *flags* specifies what shift keys were pressed at the time of the button event; it is a mask word created by combining zero or more of the following flags: `#x08` means the shift key was pressed, `#x10` means the control key was pressed, and `#x20` means the alt key was pressed.

`#(1` *device*`)`
: A close event. The user has selected the close button from the system menu, or typed ⟨Alt-f4⟩.

`#(2` *device gained?*`)`
: A focus event. If *gained?* is `#t`, the keyboard focus is being gained, and if *gained?* is `#f`, it is being lost.

`#(3` *device code flags repeat*`)`
: A keyboard event. This is much too complicated to describe here. See the OS/2 toolkit documentation for details.

`#(4` *device xl xh yl yh*`)`
: A paint event. Part of the graphics-device window that was obscured has been revealed and the Presentation Manager is informing the window that it must repaint that area. Scheme will take care of the painting for you, so this event isn't very useful.

`#(5` *device width height*`)`
: A size-change event. The size of the graphics-device window has changed, and *width* and *height* specify the new size in pels (pixels).

`#(6` *device shown?*`)`
: A visibility event. Indicates that the graphics-device window has been hidden or revealed. If *shown?* is `#f`, the window is hidden, and if it is `#t`, the window is shown.

`#(7` *device source mouse?*`)`
`#(8` *device source mouse?*`)`
: A menu command. *Source* specifies which menu item was selected to cause this event, and *mouse?* is a boolean indicating whether the item was selected with the mouse or the keyboard. The event-type number `7` indicates a command from a '`WM_COMMAND`' message, while `8` is a command from a '`WM_HELP`' message.

`#(9` *device x y hit-test flags*`)`
: The mouse was moved. *X* and *y* specify the position of the mouse, *hit-test* contains the hit-test information, and *flags* specifies the modifier keys that were pressed at the time.

discard-events operation+ on os2-graphics-device

This operation discards any events that are in the user-event queue. This is sometimes useful when you want to prompt the user for some input and don't want to consider any previous input.

17.11.5 Miscellaneous Operations for OS/2 Graphics

These operations allow you to: change the font used for drawing text in a graphics-device window; take a snapshot of a graphics-device window and return it as an image object; and draw multiple lines efficiently.

set-font *font-name* operation+ on os2-graphics-device

This operation sets the font used for drawing text in the graphics-device window. *Font-name* is a string describing the font; this string is in the form "<point-size>.<family-name>", for example, "10.Courier". You may specify any fixed-pitch font family, in any point size that is supported for that font family. This includes both image fonts and outline fonts.

capture-image *x-left y-bottom x-right y-top* operation+ on os2-graphics-device

This operation creates and returns an image that contains part of the client area of the graphics-device window. The portion of the client area that is selected is specified by the four coordinate arguments, which are given in the current virtual coordinates for the device. See Section 17.8 [Images], page 225, for more information about manipulating images.

draw-lines *xv yv* operation+ on os2-graphics-device

This operation draws multiple disjoint lines; it is like multiple calls to `graphics-draw-line` but much faster. The arguments *xv* and *yv* are vectors of coordinates; these vectors must be the same length, and the length must be a multiple of two. The contents of the vectors are alternating start/end pairs. For example, the following are equivalent:

```
(graphics-draw-line device xs ys xe ye)
(graphics-operation device 'draw-lines
                    (vector xs xe)
                    (vector ys ye))
```

18 Win32 Package Reference

18.1 Overview

The Win32 implementation is still in a state of development. It is expected that changes will be necessary when MIT Scheme is ported to Windows NT on the DEC Alpha architecture. In particular, the current system is not arranged in a way that adequately distinguishes between issues that are a consequence of the NT operating system and those which are a consequence of the Intel x86 architecture.

Thus this documentation is not definitive, it merely outlines how the current system works. Parts of the system will change and any project implemented using the win32 system must plan for a re-implementation stage.

The Win32 implementation has several components:

- Special microcode primitives.
- A foreign function interface (FFI) for calling procedures in dynamically linked libraries (DLLs).
- An interface for Edwin.
- The Win32 package provides support for using the features of the Windows 3.1 and Windows NT 3.1 environments.
- Device Independent Bitmap utilities. These are used by the win32 Scheme Graphics implementation. (The Scheme Graphics implementation is described in the Reference Manual).

Note that all the names in the Win32 support are part of the `win32` package. The names are bound in the `(win32)` environment, and do not appear as bindings in the user or root environments. An effect of this is that it is far easier to develop Win32 software in the `(win32)` package environment or a child environment.

18.2 Foreign Function Interface

The Win32 foreign function interface (FFI) is a primitive and fairly simple system for calling procedures written in C in a dynamically linked library (DLL). Both user's procedures from a custom DLL and system procedures (e.g. MessageBox) are called using the same mechanism.

Warning: The FFI as it stands has several flaws which make it difficult to use reliably. It is expected that both the interface to and the mechanisms used by the FFI will be changed in the future. We provide it, and this documentation, only to give people an early start in accessing some of the features of Win32 from Scheme. Should you use it in an experiment we welcome any feedback.

The FFI is designed for calling C procedures that use C data types rather than Scheme data objects. Thus it is not possible to write and call a C procedure that returns, for example, a Scheme list. The object returned will always be an integer (which may represent the address of a C data structure).

Warning: It is extremely dangerous to try to pass Scheme callback procedures to C procedures. It is only possible by passing integer 'handles' rather than the actual procedures, and even so, if a garbage collection occurs during the execution of the callback procedure objects in Scheme's heap will have moved. Thus in a foreign procedure that has a callback and a string, after calling the callback the string value may no longer be valid. Playing this game requires a profound knowledge of the implementation.

The interface to the FFI has two main components: a language for declaring the types of values passed to and returned from the foreign procedures and a form for declaring foreign procedures.

18.2.1 Windows Types

Foreign types are designed to represent a correspondence between a Scheme data type that is used to represent an object within the Scheme world and a C data type that represents the data object in the C world. Thus we cannot manipulate true C objects in Scheme, nor can we manipulate Scheme objects in C.

Each foreign type has four aspects that together ensure that the correspondence between the Scheme and C objects is maintained. These aspects are all encoded as procedures that either check for validity or convert between representations. Thus a foreign type is not a declarative type so much as a procedural description of how to pass the type. The underlying foreign procedure call mechanism can pass integers and vector-like Scheme objects, and returns integer values. All other objects must be translated into integers or some other basic type, and must be recovered from integers.

The aspects are:

check
: A predicate that returns `#t` if the argument is of an acceptable Scheme type, otherwise returns `#f`. The *check* procedure is used for type-checking.

convert
: A procedure of one argument which returns a Scheme object of one of the basic types. It is used to convert an object into a 'simpler' object that will eventually be converted into a C object. The legal simpler objects are integers and strings.

return-convert
: A procedure of one argument that, given an integer, returns a Scheme object of a type satisfying *check*. Its purpose is to convert the result returned by the foreign procedure into a Scheme value.

revert
: Some C procedures modify one or more of their arguments. These arguments are passed by reference, i.e. as a pointer to their address. Since a Scheme object might have a different memory layout and storage conventions, it must be passed by copy-in and copy-out rather than by reference. *Revert* is a procedure of two parameters, the original object passed and the result of *convert* on that object. *Revert* may then inspect the converted object and copy back the changes to the original.

define-windows-type *name check convert return revert* special form+
define-similar-windows-type *name model* [*check* [*convert* [*return* [*revert*]]]] special form+

Both forms define a windows type. The first form defines a type in terms of its aspects as described above. The second defines the type as being like another type, except for certain aspects, which are redefined. *Name* is the name of the type. *Model* is the name of a type. *Check*, *convert*, *return* and *revert* are procedures or the value `#f`. A `#f` means use the default value, which in the second form means use the definition provided for *model*. The defaults are

check `(lambda (x) #t)`, i.e. unchecked.

convert `(lambda (x) x)`, i.e. no translation performed.

return `(lambda (x) x)`, i.e. no translation performed.

revert `(lambda (x y) unspecific)`, i.e. no update performed

The `unchecked` windows type (see below) is defined as:

```
(define-windows-type unchecked #f #f #f #f)
```

Windows types are *not* first class values, so they cannot be stored in variables or defined using `define`:

```
(define my-type unchecked)             error  Unbound variable
(define-similar-windows-type my-type unchecked)
                                        ;; the correct way
```

Scheme characters must be converted to integers. This is accomplished as follows:

```
(define-windows-type char
   char?            ; check
   char->integer    ; convert
   integer->char    ; convert return value
   #f               ; cannot be passed by reference
)
```

unchecked windows type

The type which is not checked and undergoes only the basic conversion from a Scheme integer to a C integer or from a Scheme string to a C pointer to the first byte of the string. Returned `unchecked` values are returned as integers.

bool windows type

Scheme booleans are analogous to C integers `0` and `1`. Windows type `bool` have been defined as:

```
(define-windows-type bool
   boolean?
   (lambda (x) (if x 1 0))
   (lambda (x) (if (eq? x 0) #f #t))
   #f)
```

char windows type

Scheme characters are converted into C objects of type `char`, which are indistinguishable from small integers.

int windows type
uint windows type
long windows type
ulong windows type
short windows type
ushort windows type
word windows type
byte windows type

Various integer types that are passed without conversion.

string windows type

A string that is passed as a C pointer of type `char*` to the first character in the string.

char* windows type

A string or `#f`. The string is passed as a pointer to characters. The string is correctly null-terminated. `#f` is passed as the null pointer. This is an example where there is a more complex mapping between C objects and Scheme objects. C's `char*` type is represented as one of two Scheme types depending on its value. This allows us us to distinguish between the C string (pointer) that points to the empty sequence of characters and the null pointer (which doesnt point anywhere).

handle windows type
hbitmap windows type
hbrush windows type
hcursor windows type
hdc windows type
hicon windows type
hinstance windows type
hmenu windows type
hpalette windows type
hpen windows type
hrgn windows type
hwnd windows type

Various kinds of Win32 handle. These names correspond to the same, but all uppercase, names in the Windows C language header files. Win32 API calls are the source of values of this type and the values are meaningless except as arguments to other Win32 API calls. Currently these values are represented as integers but we expect that Win32 handles will in future be represented by allocated Scheme objects (e.g. records) that will allow predicates (e.g. `hmenu?`) and sensible interlocking with the garbage collector to free the programmer of the current tedious allocation and deallocation of handles.

resource-id windows type

A Windows resource identifier is either a small integer or a string. In C, this distinction is possible because pointers look like larger integers, so a machine word representing a small integer can be distinguished from a machine word that is a pointer to the text of the name of the resource.

18.2.2 Windows Foreign Procedures

Foreign procedures are declared as callable entry-points in a module, usually a dynamically linked library (DLL).

find-module *name* procedure+

Returns a module suitable for use in creating procedures with `windows-procedure`. *Name* is a string which is the name of a DLL file. Internally, `find-module` uses the `LoadLibrary` Win32 API, so *name* should conform to the specifications for this call. *Name* should be either a full path name of a DLL, or the name of a DLL that resides in the same directory as the Scheme binary '`SCHEME.EXE`' or in the system directory.

The module returned is a description for the DLL, and the DLL need not necessarily be linked at or immediately after this call. DLL modules are linked on need and unlinked before Scheme exits and when there are no remaining references to entry points after a garbage-collection. This behavior ensures that the Scheme system can run when a DLL is absent, provided the DLL is not actually used (i.e. no attempt is made to call a procedure in the DLL).

gdi32.dll variable+

This variable is bound to the module describing the '`GDI32.DLL`' library, which contains the Win32 API graphics calls, e.g. `LineTo`.

kernel32.dll variable+

This variable is bound to the module describing the '`KERNEL32.DLL`' library.

user32.dll variable+

This variable is bound to the module describing the '`USER32.DLL`' library. This module contains many useful Win32 API procedures, like `MessageBox` and `SetWindowText`.

windows-procedure (*name* (*parameter type*) ...) *return-type module entry-name* [*options*] special form+

This form creates a procedure, and could be thought of as "foreign-named-lambda". The form creates a Scheme procedure that calls the C procedure identified by the exported entry point *entry-name* in the module identified by the value of *module*. Both *entry-name* and *module* are evaluated at procedure creation time, so either may be expression. *Entry-name* must evaluate to a string and *module* must evaluate to a module as returned by `find-module`. These are the only parts of the form that are evaluated at procedure creation time.

Name is the name of the procedure and is for documentation purposes only. This form *does not* define a procedure called *name*. It is more like `lambda`. The name might be used for debugging and pretty-printing.

A windows procedure has a fixed number of parameters (i.e. no 'rest' parameters or 'varargs'), each of which is named and associated with a windows type *type*. Both the name *parameter* and the windows type *type* must be symbols and are not evaluated. The procedure returns a value of the windows type *return-type*.

The following example creates a procedure that takes a window handle (`hwnd`) and a string and returns a boolean (`bool`) result. The procedure does this by calling the `SetWindowText` entry in the module that is the value of the variable `user32.dll`. The variable `set-window-title` is defined to have this procedure as it's value.

```
(define set-window-title
  (windows-procedure
   (set-window-text (window hwnd) (text string))
   bool user32.dll "SetWindowText"))

(set-window-title my-win "Hi")
                          ⇒  #t
                          ;; Changes window's title/text

set-window-title          ⇒  #[compiled-procedure  ...]
set-window-text           error   Unbound variable
```

When there are no *options* the created procedure will (a) check its arguments against the types, (b) convert the arguments, (c) call the C procedure and (d) convert the returned value. No reversion is performed, even if one of the *types* has a reversion defined. (Reverted types are rare [I have never used one], so paying a cost for this unless it is used seems silly).

The following options are allowed:

`with-reversions`
: The reversions are included in the type conversions.

`expand`
: A synonym for `with-reversions`.

Scheme code
: The *Scheme code* is placed between steps (a) and (b) in the default process. The Scheme code can enforce constraints on the arguments, including constraints between arguments such as checking that an index refers to a valid position in a string.

If both options (i.e. `with-reversions` and Scheme code) are used, `with-reversions` must appear first. There can be arbitrarily many Scheme expression.

18.2.3 Win32 API names and procedures

This section is a moving target.

The `#define` values from '`wingdi.h`' and '`winuser.h`' are available as bindings in the `(win32)` package environment. The `#define` symbols are all uppercase; these have been translated to all lowercase Scheme identifiers, thus `WM_LBUTTONUP` is the scheme variable `wm_lbuttonup`. As Scheme is case insensitive, the upper-case version may be used and probably should to make the code look more like conventional Windows code. The Scheme bindings have been produced automagically. Most of the `#define`-symbols contain an underscore so there are not many name clashes. There is one very notable name clash, however: `ERROR` is `#define`d to 0, which shadows the scheme procedure `error` in the root package environment. To signal an error, use `access` to get `error` from the system global environment:

```
(declare (usual-integrations))
...
((access error system-global-environment) "Complain" ...)
```

The set of procedures is incomplete because procedures have been added on a by-need basis for the implementation of other parts of the system, e.g. Scheme Graphics. Look in the implementation for further details.

Win32 API procedure names have been uniformly converted into Scheme identifiers as follows:

- A leading uppercase letter is translated into a lowercase letter.
- Subsequent sequences of uppercase letters are translated into lowercase letters preceeded by a hyphen (minus symbol), i.e. hyphens are inserted at a lowercase to uppercase transition.
- Predicates beginning with `Is` finally have a question-mark appended.

Example: applying these rules to `IsWindow` yields `is-window?`, and `GetDC` is translated into `get-dc`.

18.3 Device Independent Bitmap Utilities

The Device Independent Bitmap (DIB) utilities library '`DIBUTILS.DLL`' and the associated procedures in '`dib.scm`' in the Win32 system source is an example of how to use the foreign function interface to access and manipulate non-Scheme objects.

dib windows type

In the C world a DIB is a *handle* to a piece of memory containing the bits that represent information about the image and the pixels of the image. The handle is a machine-word sized piece of data which may be thought of as a 32 bit integer. The handle may be null (i.e. zero), indicating that there is no block of memory describing the DIB. The null value is usually returned by C functions that are supposed to create a DIB but failed, for some reason like the memory could not be allocated or a file could not be opened.

In the Scheme world a DIB is a structure containing information about the bitmap (specifically the integer that represents the handle). We also include `#f` in the `dib` windows type to mirror the null handle error value.

```
(define dib-result
  (lambda (handle)
    (if (= handle 0)
        #f
        (make-dib handle))))

(define dib-arg
  (lambda (dib)
    (if dib
        (cell-contents (dib-handle dib))
        0)))

(define-windows-type dib
  (lambda (thing) (or (dib? thing) (eq? thing #f)))
  dib-arg
  dib-result)
```

18.3.1 DIB procedures

The following procedures have typed parameters, using the same convention as `windows-procedure`.

open-dib (*filename string*) procedure+

Return type: *dib*. Calls the `OpenDIB` entry of '`DIBUTILS.DLL`'. If the return value is not `#f` then the file *filename* was found, successfully opened, and the contents were suitable for loading into memory as a device independent bitmap.

write-dib (*filename string*) (*dib dib*) procedure+

Return type: *bool*. Calls the `WriteDIB` entry of '`DIBUTILS.DLL`'. Returns `#t` if the file *filename* could be opened and written to. After this operation the file contains the bitmap data in a standard format that is understood by `open-dib` and various system utilities like the bitmap editor. Any problems resulting in failure are signalled by a `#f` return value.

bitmap-from-dib (*dib dib*) (*palette hpalette*) procedure+

Return type: *hbitmap*. Calls the `BitmapFromDib` entry of '`DIBUTILS.DLL`'. The returned value is a device dependent bitmap. The colours from the DIB are matched against colors in *palette*.

dib-from-bitmap (*bitmap hbitmap*) (*style dword*) (*bits word*) (*palette hpalette*) procedure+

Return type: *dib*. Returns a DIB containing the same image as the device dependent bitmap *bitmap*. *Style* determines the kind of DIB, e.g. compression style. Calls the `DibFromBitmap` entry of '`DIBUTILS.DLL`'.

dib-blt (*dest hdc*) (*x int*) (*y int*) (*w int*) (*h int*) (*src dib*) (*src-x int*) (*src-y int*) (*raster-op long*) procedure+

Return type: *bool*. Calls the `DibBlt` entry of '`DIBUTILS.DLL`'. Similar to the Win32 API `BitBlt` call, but draws a DIB rather than a piece of another device context. Draws the *dib* on device context *hdc* at position (*x*,*y*). A rectangle of width *w* and height *h* is copied from position (*src-x*,*src-y*) of *dib*. *Raster-op* is supposed to allow the source and destination to be combined but I don't think I got this right so stick to `SRCCOPY`.

%delete-dib (*dib-handle handle*) procedure+

Return type: *bool*. Calls the `DeleteDIB` entry of '`DIBUTILS.DLL`'. Note that the parameter is a *handle*, and not a *dib*. This allows us to destroy a DIB and reclaim its memory by knowing only the handle value, and not needing the `dib` record. The importance of this is that if the `dib` record is GC-ed then a GC hook can reclaim the storage knowing only the handle.

delete-dib (*dib dib*) procedure+

Return type: *bool*. This procedure calls `%delete-dib` to reclaim the storage occupied by a DIB. After being deleted, the DIB should not be used. This procedure allows the programmer to reclaim external heap storage rather than risking it running out before the next garbage collection.

dib-height (*dib dib*) procedure+

Return type: *int*. Calls the `DibHeight` expand entry of '`DIBUTILS.DLL`', which returns the height of the bitmap in pixels.

dib-width (*dib dib*) procedure+

Return type: *int*. Calls the `DibWidth` entry of '`DIBUTILS.DLL`', which returns the width of the bitmap in pixels.

copy-bitmap (*bm hbitmap*) procedure+

Return type: *hbitmap*. Calls the `CopyBitmap` of '`DIBUTILS.DLL`', which creates a new bitmap with the same size and contents as the original.

create-dib (*width int*) (*height int*) (*style int*) (*depth int*) (*palette hpalette*) procedure+

Return type: *dib*. Calls the `CreateDIB` entry of '`DIBUTILS.DLL`'. Creates a DIB of *width* by *height* pixels and *depth* bits of colour information. The *style* parameter determines how the bitmap is stored. I have only ever used `BI_RGB`. If *depth*<=8 then the *palette* determines the DIB's colour table.

crop-bitmap (*bm hbitmap*) (*left int*) (*top int*) (*right int*) (*bottom int*) procedure+

Return type: *hbitmap*. Calls the `CropBitmap` entry of '`DIBUTILS.DLL`'. Returns a new bitmap containing the image from a region of the original.

dib-set-pixels-unaligned *dib* (*pixels string*) procedure+

Return type: *bool*. Calls the `DIBSetPixelsUnaligned` entry of '`DIBUTILS.DLL`'. Stuffs bytes from *pixels* into the bitmap. There are no alignment constraints on *pixels* (the usual way of doing this is to use the `SetDIBits` function which requires that every scan line of the bitmap is 32-bit word aligned, even if the scan lines are not a multiple of 4 bytes long). doing this

18.3.2 Other parts of the DIB Utilities implementation

The 'DIBUTILS.DLL' library is an ordinary DLL. See the standard Microsoft Windows documentation on how to create DLLs. Look at the code in the 'WIN32/DIBUTILS' directory of the Scheme source.

Please note:

- For the foreign function interface to find the procedures they must be declared as exports in the '.DEF' definition file.
- To load the '.DLL' file use the `find-module` Scheme function. Look at 'WIN32/DIB.SCM' to see how this is done.
- The current system works with C procedures with the `__stdcall` and `__cdecl` calling conventions but *not* the `__fastcall` calling convention.

Index

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z

Short Contents

Table of Contents