MIT Scheme Reference Manual

Edition 1.1
for Scheme Release 7.1.3
November 1991

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 of the Massachusetts Institute of Technology. Support for the laboratory's artificial intelligence research is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contracts N00014-85-K-0124 and N00014-86-K-0180.

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 `signal-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, the error message is shown, prefaced by '$\boxed{\text{error}}$ ':

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

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

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

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

1.1.3 Entry Format

Each description of an MIT 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 19), 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.

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*	a real number
q *n*	an integer
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 13). 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 signals an "Unassigned Variable" error. Unassigned variables are useful only in combination with side effects (see Section 2.5 [Assignments], page 28).

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 an "Unbound Variable" error.

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 27) 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 26) 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. REP loops that are created by the error handler use the environment in which the error occurred, if possible, or failing that they use the previous REP loop's environment.

```
1 ]=> z                                          error  Unbound variable

2 Error-> (define z 3)

2 Error-> z                                      ⇒  3

2 Error-> <Control-G> interrupt typed

1 ]=> z                                          ⇒  3
```

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 31).

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 soon 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 defined representations for them).

An external representation may be written in a program to obtain the corresponding object (see Section 2.6 [Quoting], page 28).

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 10, for an exception):

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

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.

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 17).
- 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 45), 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 45 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 19).

`( )` Parentheses are used for grouping and to notate lists (see Chapter 7 [Lists], page 87).

`"` The double quote delimits strings (see Chapter 6 [Strings], page 75).

`\` The backslash is used in the syntax for character constants (see Chapter 5 [Characters], page 65) and as an escape character within string constants (see Chapter 6 [Strings], page 75).

`;` The semicolon starts a comment.

`'` The single quote indicates literal data; it suppresses evaluation (see Section 2.6 [Quoting], page 28).

`` ` `` The backquote indicates almost-constant data (see Section 2.6 [Quoting], page 28).

`,` The comma is used in conjunction with the backquote (see Section 2.6 [Quoting], page 28).

`,@` A comma followed by an at-sign is used in conjunction with the backquote (see Section 2.6 [Quoting], page 28).

`#` 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 113).

`#\` This character sequence introduces a character constant (see Chapter 5 [Characters], page 65).

`#(` This character sequence introduces a vector constant (see Chapter 8 [Vectors], page 103). A close parenthesis, '`)`', terminates a vector constant.

`#e #i #b #o #d #x`
These character sequences are used in the notation for numbers (see Chapter 4 [Numbers], page 45).

`#|` 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 107). This notation 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 28); 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 13) 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
bkpt                    error                   or
case                    fluid-let               quasiquote
cond                    if                      quote
cons-stream             in-package              scode-quote
declare                 lambda                  sequence
default-object?         let                     set!
define                  let*                    the-environment
define-integrable       let-syntax              unassigned?
define-macro            letrec                  using-syntax
define-structure        local-declare
```

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 19, 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, a "Wrong Number of Arguments" error 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, a "Wrong Number of Arguments" error 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.

`(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 *expression*s as its region.

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

Note that the following are equivalent:

```
(let ((variable init) ...) 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 35, 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 *variables* are bound to fresh locations holding unassigned values, the *inits* 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 *inits* to be omitted, in which case the corresponding *variables* 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 *inits* are `lambda` or `delay` expressions and the restriction is satisfied automatically.

2.3 Fluid 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 values 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 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-fluid-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-fluid-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 is unassigned; an attempt to reference such a variable signals an "Unassigned Variable" 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; subsequent references to such *variable* are "Unassigned Variable" errors. 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 149). 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 10). 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 10).

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 *objects* 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 39) 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

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 *expression*s to a procedure whose formal arguments are the *variable*s and whose body is the *expression*s. Thus the execution of the *expression*s may be repeated by invoking the procedure named by *name*.

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

Note: the following expressions are equivalent:

```
(let name ((variable init) ...)
  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 *expressions* 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
```

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 45).
- *obj1* and *obj2* are both characters and are the same character according to the `char=?` procedure (see Chapter 5 [Characters], page 65).
- 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, 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 47.

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 45), but they must implement a coherent subset consistent with both the purposes of the implementation and the spirit of the Scheme language. For example, an implementation in which all numbers are real may still be quite useful.[1]

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

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

[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.

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.

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.

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

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.

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

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

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

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

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

zero? *z* procedure
positive? *x* procedure
negative? *x* procedure
odd? *x* procedure
even? *x* procedure

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

max *x1 x2 ...* procedure
min *x1 x2 ...* procedure

These procedures return the maximum or minimum of their arguments.

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

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

`+` *z1 ...* procedure
`*` *z1 ...* procedure

These procedures return the sum or product of their arguments.

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

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

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

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

1+ *z* procedure+
-1+ *z* procedure+

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

abs *x* procedure

`abs` returns the magnitude of its argument.

```
(abs -7)                ⇒  7
```

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

$$n_1 = n_2 n_3 + n_4$$

[4] MIT Scheme signals an error in this case.

$$0 \leq n_4 < n_2$$

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 *x1 x2* procedure
real-part *z* procedure
imag-part *z* procedure
magnitude *z* procedure
angle *z* procedure
conjugate *z* procedure+

Suppose *x1*, *x2*, *x3*, and *x4* are real numbers and *z* is a complex number such that

$$z = x_1 + x_2 i = x_3 \cdot e^{i x_4}$$

Then `make-rectangular` and `make-polar` return *z*, `real-part` returns *x1*, `imag-part` returns *x2*, `magnitude` returns *x3*, and `angle` returns *x4*. In the case of `angle`, whose value is not uniquely determined by the preceding rule, the value returned will be the one in the range minus *pi* (exclusive) to *pi* (inclusive).

`conjugate` returns the complex conjugate of *z*.

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 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; MIT Scheme signals an error in this case.

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 47.

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.

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
```

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 0))
  (let ((m (+ n 1)))
    (if (fix:fixnum? m)
        (loop m)
        n)))
```

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 #432 -10)                  ⇒  1
(fix:lsh -1 3)                      ⇒  -8
(fix:lsh -128 -4)                   ⇒  -8
```

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.

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,

[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.

```
#\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:

```
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 either `#\linefeed` or `#\return`, depending on the operating system that Scheme is running under. 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, 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
```

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 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]

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:

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

```
! " # $ % & ' ( ) * + , - . /
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*`)`.

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.

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 84). 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?`.

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)                                  ⇒  ""
```

For compatibility with old code, `char->string` is a synonym for this procedure.

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`.

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-length *string* procedure

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

string-null? *string* procedure

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

string-ref *string k* procedure

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

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?`.

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<? *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 67). 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-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.

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 `#t` if the first character in the string (substring) is an uppercase letter and none of the remaining characters are uppercase letters. If the first character is not an uppercase letter or if any of the remaining characters are uppercase letters, they return `#f`. If the string (substring) contains less than two letters, they return `#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`.

string-capitalize *string* procedure+
string-capitalize! *string* procedure+

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

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*.

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 (`""`).

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).

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-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*, these procedures are equivalent to `string-head`. `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-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:whitespace`.

6.6 Searching Strings

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

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-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-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-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? "def" "abcdef")         ⇒  #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.

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.

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.

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-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* 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 record 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:

---

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

```
(define x (list 'a 'b 'c))
(define y x)
y                                   ⇒  (a b c)
(list? y)                           ⇒  #t
(set-cdr! x 4)                      ⇒  unspecified
x                                   ⇒  (a . 4)
(eqv? x y)                          ⇒  #t
y                                   ⇒  (a . 4)
(list? y)                           ⇒  #f
(set-cdr! x x)                      ⇒  unspecified
(list? y)                           ⇒  #f
```

A chain of pairs that doesn't end in the empty list is called an *improper list*. Note that an improper list is not a list. The list and dotted notations can be combined to represent improper lists, as the following equivalent notations show:

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

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.

---

[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).

```
(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.

```
(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 10).

```
(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*, signalling 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)
```

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.

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?))
```

7.7 Mapping of Lists

[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`.

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 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)))
```

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*, that value 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 *list procedure* procedure+

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

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

`sort` returns a newly allocated list whose elements are the elements of *list*, except that the elements are rearranged so that they are sorted in the order defined by *procedure*. So, for example, if the elements of *list* are numbers, and *procedure* is `<`, then the resulting list is sorted in monotonically nondecreasing order. Likewise, if *procedure* is `>`, the resulting list is sorted in monotonically nonincreasing order. To be precise, if *x* and *y* are any two adjacent elements in the resulting list, where *x* precedes *y*, it is the case that

```
(procedure y x)
     ⇒  #f
```

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, which 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.

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.

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 *vector*'s length is too small.

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 be 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.

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.

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

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

Sets the *k*th bit in *bit-string* to 1 and returns an unspecified value. *K* must be a valid index of *bit-string*.

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

Sets the *k*th bit in *bit-string* to 0 and returns an unspecified value. *K* must be a valid index of *bit-string*.

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 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 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 10).

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 some standard declarations, 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 returns `#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]

[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 132). 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

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 `eqv?` 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 13). 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 `eqv?`).

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]

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

---

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 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.

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 13), otherwise the resulting symbol cannot be read as itself.

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 not to be `eqv?` to any other object in the Scheme system. The symbol's name consists of a string (initially `"G"`) followed by an integer that is incremented on every call (the integer is initially 0). The optional *object* can be an integer or a symbol. If *object* is a symbol, the string prefix of all subsequently generated symbol names will be that symbol's name. If *object* is an integer, the integer suffix of all subsequently generated symbol names will start counting from that value.

```
(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 this3]
(generate-uninterned-symbol 100)
     ⇒  #[uninterned-symbol 35 this100]
(generate-uninterned-symbol)
     ⇒  #[uninterned-symbol 36 this101]
```

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.

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 fluid binding of a variable, including the behavior when continuations are used (see Section 2.3 [Fluid Binding], page 24).

10.4 Records

MIT Scheme provides a *record* abstraction, which is a simple and flexible mechanism for building structures with named components. This abstraction will very likely be a part of the next edition of the Scheme standard. The procedures defined in this section that are expected to be in the standard are not marked with '`+`'.

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-updater *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 updater 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.[3] 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*.[4]

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*.

---

[3] In the current implementation of MIT Scheme, any object that satisfies this predicate also satisfies `vector?`. However, we plan to change the implementation to make records distinct from all other data types.

[4] However, in MIT Scheme, the record-type descriptor representing a given record type is unique.

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*.[5]

10.5 Promises

delay *expression* special form

The `delay` construct is used together with the procedure `force` to implement *lazy evaluation* or *call by need*. `(delay` *expression*`)` returns an object called a *promise* which at some point in the future may be asked (by the `force` procedure) to evaluate *expression* and deliver the resulting value.

force *promise* procedure

Forces the value of *promise*. If no value has been computed for the promise, then a value is computed and returned. The value of the promise is cached (or "memoized") so that if it is forced a second time, the previously computed value is returned without any recomputation.

```
(force (delay (+ 1 2)))                ⇒  3

(let ((p (delay (+ 1 2))))
  (list (force p) (force p)))          ⇒  (3 3)

(define a-stream
  (letrec ((next
            (lambda (n)
              (cons n (delay (next (+ n 1)))))))
    (next 0)))

(define head car)

(define tail
  (lambda (stream)
    (force (cdr stream))))

(head (tail (tail a-stream)))          ⇒  2
```

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

promise? *object* procedure+

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

promise-forced? *promise* procedure+

Returns `#t` if *promise* has been forced and its value cached; otherwise returns `#f`.

promise-value *promise* procedure+

If *promise* has been forced and its value cached, this procedure returns the cached value. Otherwise, an error is signalled.

`force` and `delay` are mainly intended for programs written in functional style. The following examples should not be considered to illustrate good programming style, but they illustrate the property that the value of a promise is computed at most once.

```
(define count 0)

(define p
  (delay
   (begin
     (set! count (+ count 1))
     (* x 3))))

(define x 5)

count                                   ⇒  0
p                                       ⇒  #[promise 54]
(force p)                               ⇒  15
p                                       ⇒  #[promise 54]
count                                   ⇒  1
(force p)                               ⇒  15
count                                   ⇒  1
```

Here is a possible implementation of `delay` and `force`. We define the expression

```
(delay expression)
```

to have the same meaning as the procedure call

```
(make-promise (lambda () expression))
```

where `make-promise` is defined as follows:

```
(define make-promise
  (lambda (proc)
    (let ((already-run? #f)
          (result #f))
      (lambda ()
        (cond ((not already-run?)
               (set! result (proc))
               (set! already-run? #t)))
        result))))
```

Promises are implemented here as procedures of no arguments, and `force` simply calls its argument.

```
(define force
  (lambda (promise)
    (promise)))
```

Various extensions to this semantics of `delay` and `force` are supported in some implementations (none of these are currently supported in MIT 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.[6]

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+

Returns the first element in *stream*. `stream-car` is equivalent to `car`.[7]

stream-cdr *stream* procedure+

Returns the first tail of *stream*. Equivalent to `(force (cdr` *stream*`))`.[8]

---

[6] The variable `the-empty-stream`, which is bound to the end-of-stream marker, is provided for compatibility with old code; use `(stream)` in new code.

[7] `head`, a synonym for `stream-car`, is provided for compatibility with old code; use `stream-car` in new code.

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.[9]

stream-length *stream* procedure+

Returns the number of elements in *stream*. If *stream* has an infinite number of elements this procedure will not terminate. Note that this procedure forces all of the promises that comprise *stream*.

stream-ref *stream k* procedure+

Returns the element of *stream* that is indexed by *k*; that is, the *k*th element. *K* must be an exact non-negative integer strictly less than the length of *stream*.

stream-tail *stream k* procedure+

Returns the tail of *stream* that is indexed by *k*; that is, the *k*th tail. This is equivalent to performing `stream-cdr` *k* times. *K* must be an exact non-negative integer strictly less than the length of *stream*.

stream-map *stream procedure* procedure+

Returns a newly allocated stream, each element being the result of invoking *procedure* with the corresponding element of *stream* as its argument. *Procedure* must be a procedure of one argument.

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?`.

---

[8] `tail`, a synonym for `stream-cdr`, is provided for compatibility with old code; use `stream-cdr` in new code.

[9] `empty-stream?`, a synonym for `stream-null?`, is provided for compatibility with old code; use `stream-null?` in new code.

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 is lookup required for both keys.
- *Hash tables* are a powerful mechanism with nearly constant-time access to large amounts of data. However, the overhead for hash tables is somewhat high, both in time and space, making them unsuitable for small tables. And hash tables are not as flexible as association lists.

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]

---

[1] This introduction is taken from *Common Lisp, The Language*, second edition, p. 431.

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 98).

For example, here is how `assv` could be implemented:

```
(define assv (association-procedure eqv? car))
```

Another example is a “reverse association” procedure:

---

[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 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. `del-assq` uses `eq?` to compare *object* with the keys, while `del-assv` uses `eqv?` and `del-assoc` uses `equal?`.

```
(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 can often be used as a higher-performance alternative to the two-dimensional association table (see Section 11.3 [The Association Table], page 133). 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 and `#t` is returned. Otherwise `#f` is returned.

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 growth, customizable growth parameters, and customizable hash functions.

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.

make-object-hash-table [*k*] procedure+

Returns a newly allocated hash table that accepts arbitrary objects as keys, and uses `eq?` to compare the keys. The keys are weakly held, i.e. the hash table does not protect them from being reclaimed by the garbage collector. *K* specifies the initial usable size of the hash table, and defaults to 10 if not specified. The entries of the hash table are weak pairs whose car field is the *key* and whose cdr field is the *datum*.

make-string-hash-table [*k*] procedure+

Returns a newly allocated hash table that accepts strings as keys, and compares them with `string=?`. The keys are strongly held, i.e. while in the hash table they will not be reclaimed by the garbage collector. *K* specifies the initial usable size of the hash table, and defaults to 10 if not specified. The entries of the hash table are ordinary pairs whose car field is the *key* and whose cdr field is the *datum*.

make-symbol-hash-table [*k*] procedure+

Returns a newly allocated hash table that accepts symbols as keys, and compares them with `eq?`. The keys are strongly held, i.e. while in the hash table they will not be reclaimed by the garbage collector. *K* specifies the initial usable size of the hash table, and defaults to 10 if not specified. The entries of the hash table are ordinary pairs whose car field is the *key* and whose cdr field is the *datum*.

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.

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

If *hash-table* has an association for *key*, removes it. Returns an unspecified result.

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

Removes all associations in *hash-table* and returns an unspecified result.

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.

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/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 entry being processed.

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

Returns a newly allocated list of the entries in *hash-table*. The elements of the list are usually either pairs or weak pairs, depending on the type of 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))
```

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

If *hash-table* is a type of hash table that holds its *key*s 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.

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

Returns a constructor procedure for a hash table. The returned procedure accepts one optional argument *k*, which specifies the initial usable size of the table, and returns a newly allocated hash table; `k` defaults to `10` if not supplied. The arguments to `hash-table/constructor` define the characteristics of the hash table as follows:

key-hash
: The hashing function. A procedure that accepts two arguments, a *key* and an exact positive integer (the hash modulus), and returns an exact non-negative integer that is less than the second argument. Examples: `string-hash-mod`, `symbol-hash-mod`.

key=?
: A equivalence predicate that accepts two *key* arguments and is true if they are the same key. Examples: `eq?`, `string=?`.

make-entry
: A procedure that accepts a *key* and a *datum* as arguments and returns an entry. Typically `cons` or `weak-cons`.

entry-valid?
: A procedure that accepts an entry and returns `#f` only if the entry's *key* has been reclaimed by the garbage collector. For example, if entries are weak pairs, this should be `weak-pair/car?`. 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*. Typically `car` or `weak-car`.

entry-value
: A procedure that accepts an entry as an argument and returns the entry's *datum*. Typically `cdr` or `weak-cdr`.

set-entry-value!
: 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. Typically `set-cdr!` or `weak-set-cdr!`.

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-value *hash-table* procedure+
hash-table/set-entry-value! *hash-table* procedure+

Each of these procedures corresponds to the respective argument of `hash-table/constructor`. When called, these procedures return the value of the argument that was used to construct *hash-table*.

Two parameters control the growth of a hash table, the *rehash threshold* and the *rehash size*.

The *rehash threshold* is a real number, between zero exclusive and one inclusive, that specifies how full the hash table can get before it must grow. In other words it is the ratio between a hash table's *usable size* and its *physical size*. If the number of entries in the table exceeds this fraction of the table's physical size, the table is grown to a larger size. The default rehash threshold of a newly constructed hash table is `1`, but this can be changed with `set-hash-table/rehash-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.

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. The default rehash size of a newly constructed hash table is `2.0`, but this can be changed with `set-hash-table/rehash-size!`.

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.

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

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

hash-table/count *hash-table* procedure+

Returns the number of entries in *hash-table* as an exact non-negative integer. This is always less than or equal to the usable size of *hash-table*.

11.5 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 some objects (see Section 14.7 [Custom Output], page 165).

object-hash *object* procedure+
object-unhash *k* procedure+

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

`object-unhash` takes an exact non-negative integer *k* and returns the object associated with that integer. If there is no object associated with *k*, `#f` is returned. In other words, if `object-hash` previously returned *k* for some object, that object is the value of the call to `object-unhash`.

An object that is passed to `object-hash` as an argument is not protected from being garbage-collected. If all other references to that object are eliminated, the object will be garbage-collected. Subsequently calling `object-unhash` with the hash number of the (garbage-collected) object will return `#f`.

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

Note: `hash` is a synonym for `object-hash` and `unhash` is a synonym for `object-unhash`. These synonyms are obsolete and should not be used.

12 Procedures

Procedures are created by evaluating `lambda` expressions (see Section 2.1 [Lambda Expressions], page 19); the `lambda` may either be explicit or may be implicit as in a "procedure `define`" (see Section 2.4 [Definitions], page 26). 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`.

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+

Continuation must be a continuation produced by `call-with-current-continuation`. *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. A future revision of the Scheme standard will support a facility similar to, but almost certainly different from, this one.

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 `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 `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. Signals an error if *symbol* is unassigned.

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 ("closed") 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 not be used lightly, as they will significantly degrade the performance of compiled code. In particular, they force the current environment to 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.

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 ports, the keyboard and the terminal screen, are 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 the keyboard, 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 the terminal screen, but Scheme provides procedures that let you change the current output port to be a file or string.

Most ports read or write only ASCII characters. However, it is possible to create ports that will read and write arbitrary characters. The limitation to ASCII characters is imposed entirely by the port, not by the I/O operations.

input-port? *object* procedure
output-port? *object* procedure

Returns `#t` if *object* is an input port or output port respectively, otherwise returns `#f`. Any object satisfying one of these predicates also satisfies `vector?`.

guarantee-input-port *object* procedure+
guarantee-output-port *object* procedure+

These procedures check the type of *object*, signalling an error if it is not an input port or output port, respectively. Otherwise they return *object*.

current-input-port procedure

Returns the current input port. Initially, `current-input-port` returns the value of `console-input-port`.

current-output-port procedure

Returns the current output port. Initially, `current-output-port` returns the value of `console-output-port`.

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

Thunk must be a procedure of no arguments. `with-input-from-port` binds the current input port to *input-port*, calls *thunk* with no arguments, restores the current input port to its original value, and returns the result that was returned by *thunk*. This temporary binding is performed the same way as fluid binding of a variable, including the behavior in the presence of continuations (see Section 2.3 [Fluid Binding], page 24).

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

Thunk must be a procedure of no arguments. `with-output-to-port` binds the current output port to *output-port*, calls *thunk* with no arguments, restores the current output port to its original value, and returns the result that was returned by *thunk*. This temporary binding is performed the same way as fluid binding of a variable, including the behavior in the presence of continuations (see Section 2.3 [Fluid Binding], page 24).

console-input-port variable+

`console-input-port` is an input port that reads from the terminal keyboard (in unix, standard input).

console-output-port variable+

`console-output-port` is an output port that writes to the terminal screen (in unix, standard output).

close-input-port *input-port* procedure

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

close-output-port *output-port* procedure

Closes *output-port* and returns an unspecified value. If *output-port* is a file port, the file 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-input-port` or `close-output-port`, depending on their type.

Before opening an input file, by whatever method, the *filename* argument is converted to canonical form by the procedure `canonicalize-input-filename`. Similarly, the *filename* argument used to open an output file is converted using `canonicalize-output-filename`. See those procedures' definitions for details on the canonicalization process. For naive purposes, the following guidelines can be used: *filename* is a character string that is the name of the file; relative filenames (on unix, those not beginning with '`/`') are interpreted relative to the directory in which MIT Scheme was started.

Implementation notes:

- On unix systems, opening an output file that already exists causes the existing file to be removed and a new one opened in its place. However, if the user does not have permission to write in the file's directory, the file is truncated to zero length and overwritten.
- File input ports always deliver ASCII characters, and file output ports only accept ASCII characters. This will change if someone ports MIT Scheme to a non-ASCII operating system.

Here are the specific operations for file ports:

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 is signalled.

Use the procedure `close-input-port` to close the port opened by this procedure. If an input port returned by this procedure is reclaimed by the garbage collector, it is automatically closed.

open-output-file *filename* 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 is signalled. If a file with the given name already exists, if possible it will be deleted and a new file opened in its place, or failing that the existing file will be truncated and overwritten.

Use the procedure `close-output-port` to close the port opened by this procedure. If an output port returned by this procedure is reclaimed by the garbage collector, it is automatically closed.

close-all-open-files procedure+

This procedure closes all file ports (input and output) that are open at the time that it is called.

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

Procedure must be a procedure of one argument. For `call-with-input-file` the file specified by *filename* must already exist; for `call-with-output-file`, if the file exists, if possible it will be deleted and a new file opened in its place, or failing that the existing file will be truncated and overwritten.

These procedures call *procedure* with one argument: the port obtained by opening the named file for input or output. If the file cannot be opened, an 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]

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

Thunk must be a procedure of no arguments. For `with-input-from-file` the file

[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`.

specified by *filename* must already exist; for `with-output-to-file`, if the file exists, if possible it will be deleted and a new file opened in its place, or failing that the existing file will be truncated and overwritten.

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 closes the port.

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, that can limit the length of the resulting string to a given value.

String input ports always deliver ASCII characters, and string output ports only accept ASCII characters. This will change if someone ports MIT Scheme to a non-ASCII operating system.

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

Creates a new input port that reads from *string*, makes that port the current input port, and calls *thunk*. When *thunk* returns, `with-input-from-string` restores the previous current input port and returns the result returned 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)
```

string->input-port *string* procedure+

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

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

Thunk must be a procedure of no arguments. 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"
```

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`, the output is truncated after *k* characters. Unlike `with-output-to-truncated-string`, if *k* is specified, this procedure always returns a string. There is no sure way to find out whether or not the returned string was truncated.

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 particular port. Remember that to read from a file, you must first open a port to the file.[2]

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.

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.[3]

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

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 guaranteed not

---

[2] Previous implementations of MIT Scheme treated interactive ports specially: when certain of these procedures were called, the input editor was temporarily enabled or disabled or the port was temporarily switched between blocking and non-blocking modes. In the current implementation, these procedures have no effect on the input editor or the blocking mode.

[3] 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.

to hang. If *input-port* is a file port at end of file then `char-ready?` returns `#t`.[4]

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 72) 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. However, if end of file was encountered before any characters were read, `read-string` returns an end-of-file object.

On many input ports, this operation is significantly faster than the following equivalent code using `peek-char` and `read-char`:

---

[4] `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))))))))))
```

14.5 Output Procedures

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*, and returns an unspecified value.

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

Writes a representation of *object* to *output-port*. Strings that appear in the written representation are not enclosed in doublequotes, and no characters are escaped within those strings. Character objects appear in the representation as if written by `write-char` instead of by `write`. `display` returns an unspecified value.[5]

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` returns an unspecified value.

---

[5] `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* and returns an unspecified value. Equivalent to `(write-char #\newline` *output-port*`)`.

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

Like `write`, except that it writes an end of line to *output-port* before writing *object*'s representation. Returns an unspecified value.

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

Writes *string* to *output-port* 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-string` *string*`)` is the same as `(display` *string*`)` except that it is faster. Use `write-string` when you know the argument is a string, and `display` when you don't.

beep [*output-port*] procedure+

Performs a "beep" operation on *output-port* and returns an unspecified value. On the console output port, this usually causes the terminal 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* 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 is equivalent to `(write-char #\page` *output-port*`)`.

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` returns an unspecified value.

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.[6]

`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.

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

---

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

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.

`~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 119), vectors that have special tags in their zero-th elements (see Chapter 8 [Vectors], page 103), and pairs that have special tags in their car fields (see Chapter 7 [Lists], page 87). 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: first, an unparser state, and second, 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 `unparser/standard-method`. The second is to define your own method using the procedures `unparse-char`, `unparse-string`, and `unparse-object`. 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.

unparser/standard-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*, if supplied, must be `#f` or a procedure of two arguments.

If *procedure* is not supplied, or 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 `display`, 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 `object-hash` on the object). Subsequently, the expression

`#@`*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 includes the space between *hash* and *output*.
2. *Procedure* is invoked on two arguments: the unparser state and the object.
3. The closing bracket is written to the output port.

The following three procedures are useful when writing unparser methods.

unparse-char *unparser-state char* procedure+

Writes *char* to the output-port component of *unparser-state*, and returns an unspecified value. Similar to `write-char`.

unparse-string *unparser-state string* procedure+

Writes *string* to the output-port component of *unparser-state*, and returns an unspecified value. Similar to `write-string`.

unparse-object *unparser-state object* procedure+

Writes *object* to the output-port component of *unparser-state*, and returns an unspecified value. *Object* is generated either as if by `display` or as if by `write`, depending on other components of *unparser-state*.

14.8 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. Typically the names are symbols, but any object that can be discriminated by `eq?` may be used.

The 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 appropriate set of operations is implemented for every port.

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.

14.8.1 Input Port Primitives

make-input-port *operations object* procedure+

Operations must be a list; each element is a list of two elements, the name of the operation and the procedure that implements it. A new input port is returned with the given operations and a state component of *object*. *Operations* need not contain definitions for all of the standard operations. `make-input-port` will provide defaults for any standard operations that are not defined. At a minimum, the operations `read-char`, `peek-char`, and `char-ready?` must be defined.

input-port/copy *input-port object* procedure+

Returns a new copy of *input-port*, identical to the original except that its state component is *object*. *Input-port* is not modified.

`input-port/copy` is normally used to speed up creation of input ports. This is done by creating a template using `make-input-port`; a dummy state component is supplied for the template. Then `input-port/copy` is used to make a copy of the template, supplying the copy with the correct state. This is useful because `make-input-port` is somewhat slow, as it must parse the *operations* list, provide defaulting for missing operations, etc.

input-port/state *input-port* procedure+

Returns the state component of *input-port*.

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

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

input-port/operation *input-port name* procedure+

Returns the procedure that implements the operation called *name*, or `#f` if *input-port* has no such operation.

input-port/custom-operation *input-port name* procedure+

Returns the procedure that implements the custom operation called *name*. If *name* names a standard operation, or if *input-port* has no such custom operation, `#f` is returned. This is faster than `input-port/operation` if *name* is known to be the name of a custom operation.

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

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

The following are the standard operations on input ports.

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.

input-port/operation/read-char *input-port* procedure+
input-port/operation/peek-char *input-port* procedure+
input-port/operation/discard-char *input-port* procedure+
input-port/operation/char-ready? *input-port* procedure+
input-port/operation/read-string *input-port* procedure+
input-port/operation/discard-chars *input-port* procedure+

Each of these procedures returns the procedure that implements the respective operation for *input-port*. Each is equivalent to, but faster than, `input-port/operation` on the respective operation name:

```
(input-port/operation/read-char input-port)
(input-port/operation input-port 'read-char)
```

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+

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

```
(input-port/read-string input-port char-set)
((input-port/operation/read-string input-port) input-port char-set)
```

14.8.2 Output Port Primitives

make-output-port *operations object* procedure+

Operations must be a list; each element is a list of two elements, the name of the operation and the procedure that implements it. A new output port is returned with the given operations and a state component of *object*. *Operations* need not contain definitions for all of the standard operations. `make-output-port` will provide defaults for any standard operations that are not defined. At a minimum, the operation `write-char` must be defined.

output-port/copy *output-port object* procedure+

Returns a new copy of *output-port*, identical to the original except that its state component is *object*. *Output-port* is not modified.

`output-port/copy` is normally used to speed up creation of output ports. This is done by creating a template using `make-output-port`; a dummy state component is supplied for the template. Then `output-port/copy` is used to make a copy of the template, supplying the copy with the correct state. This is useful because `make-output-port` is somewhat slow, as it must parse the *operations* list, provide defaulting for missing operations, etc.

output-port/state *output-port* procedure+

Returns the state component of *output-port*.

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

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

output-port/operation *output-port name* procedure+

Returns the procedure that implements the operation called *name*, or `#f` if *output-port* has no such operation.

output-port/custom-operation *output-port name* procedure+

Returns the procedure that implements the custom operation called *name*. If *name* names a standard operation, or if *output-port* has no such custom operation, `#f` is returned. This is faster than `output-port/operation` if *name* is known to be the name of a custom operation.

The following are the standard operations on output ports.

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

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

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

Writes *string* to *output-port* and returns an unspecified value. Equivalent to writing the characters of *string*, one by one, to *output-port*, but is often implemented more efficiently.

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.

output-port/operation/write-char *output-port* procedure+
output-port/operation/write-string *output-port* procedure+
output-port/operation/flush-output *output-port* procedure+

Each of these procedures returns the procedure that implements the respective operation for *output-port*. Each is equivalent to, but faster than, `output-port/operation` on the respective operation name:

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

output-port/write-char *output-port char* procedure+
output-port/write-string *output-port string* procedure+
output-port/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/write-char output-port) output-port char)
```

The custom operation `x-size` is so useful that we provide a procedure to call it:

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 79).

`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.

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.

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
```

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*, 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. In addition, `->pathname` will convert other objects, such as symbols, into pathnames.

->pathname *object* procedure+

Returns a pathname that is the equivalent of *object*. *Object* must be a pathname, a string, or a symbol. If *object* is a pathname, it is returned. If *object* is a string, this procedure acts like `string->pathname`. If *object* is a symbol, it is converted to a string, passed to `string->pathname`, and the result returned.

```
(->pathname "foo")              ⇒  #[pathname 65 "foo"]
(->pathname '/usr/morris)       ⇒  #[pathname 66 "/usr/morris"]
```

string->pathname *string* procedure+

Returns the pathname that corresponds to the filename *string*. This operation is the inverse of `pathname->string`.

pathname->string *pathname* procedure+

Returns a newly allocated string that is the filename corresponding to *pathname*. This operation is the inverse of `string->pathname`.

---

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

```
(pathname->string (string->pathname "/usr/morris/foo"))
    ⇒  "/usr/morris/foo"
```

15.1.2 Components of Pathnames

A pathname object always has six components, described below. These components are the common interface that allows programs to work the same way with different file systems; the mapping of the pathname components into the concepts peculiar to each file system is taken care of by the Scheme implementation.

host The name of the file system on which the file resides. In the current implementation of MIT Scheme, this component is not used and should always be `#f`.

device Corresponds to the "device" or "file structure" concept in many host file systems: the name of a (logical or physical) device containing files. In the current implementation of MIT Scheme, this component is not used and should always be `#f`.

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. In the current implementation of MIT Scheme, this component is not used and should always be `#f`.

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`.[2]

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

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 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:

- 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. If the first element in the list is the symbol `root`, 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. Otherwise, the directory is *relative*, meaning that 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 the special case of `root`, which may only appear as the first element of the list (if it appears at all), each element in the list is either a string or the symbol `wild` (each with the same meaning as described above), or one of these symbols: `self`, which means the next directory in the sequence is the same as the previous one, or `up`, which means the next directory is the "parent" of the previous one. `self` and `up` correspond to the files '`.`' and '`..`' in unix file systems.

  In file systems that do not have "heirarchical" structure, a specified directory component will always be a list whose first element is `root`. 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 subdirectories, then the list will contain a second element, whiich is either a string or `wild`. In other words, a non-heirarchical file system is treated as if it were heirarchical, but the heirarchical 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 heirarchy. 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 this component is not used and should be `#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 '(root "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)       ⇒  #f
(pathname-device x)     ⇒  #f
(pathname-directory x)  ⇒  (root "usr" "morris")
(pathname-name x)       ⇒  "foo"
(pathname-type x)       ⇒  "scm"
(pathname-version x)    ⇒  #f
```

pathname-components *pathname receiver* procedure+

Calls *receiver* with the six components of *pathname*, and returns the result yielded by *receiver*.

```
(pathname-components (->pathname "/usr/morris/foo.scm") list)
     ⇒  (#f #f (root "usr" "morris") "foo" "scm" #f)
```

pathname-new-host *pathname host* procedure+
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.

```
(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 '("test" "morris"))
     ⇒  #[pathname 73 "test/morris/rel15"]
p
     ⇒  #[pathname 71 "/usr/blisp/rel15"]
```

pathname-name-path *pathname* procedure+

Extracts the name, type, and version components of *pathname* and returns a pathname with just these components. For example,

```
(pathname-name-path (->pathname "/usr/blisp/rel5"))
     ⇒  #[pathname 69 "rel5"]
```

pathname-directory-path *pathname* procedure+

Extracts the host, device, and directory components of *pathname* and returns a pathname with just these components.

```
(pathname-directory-path (->pathname "/usr/blisp/rel5"))
     ⇒  #[pathname 70 "/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. Otherwise, these file components are converted into a string, and the string is added to the end of the list of directory components. Note the difference between this procedure and `pathname-directory-path`.

```
(pathname-as-directory (->pathname "/usr/blisp/rel5"))
    ⇒  #[pathname "/usr/blisp/rel5/"]
```

pathname-name-string *pathname* procedure+

Extracts the name, type, and version components of *pathname* and returns a newly allocated filename (string) with just these components. For example,

```
(pathname-name-string (->pathname "/usr/blisp/rel5"))
    ⇒  "rel5"
```

`pathname-name-string` could have been defined as follows:

```
(define (pathname-name-string pathname)
  (pathname->string (pathname-name-path pathname)))
```

pathname-directory-string *pathname* procedure+

Extracts the host, device, and directory components of *pathname* and returns a newly allocated filename (string) with just these components.

```
(pathname-directory-string (->pathname "/usr/blisp/rel5"))
    ⇒  "/usr/blisp/"
```

`pathname-directory-string` could have been defined as follows:

```
(define (pathname-directory-string pathname)
  (pathname->string (pathname-directory-path pathname)))
```

15.1.3 Operations on Pathnames

pathname? *object* procedure+

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

merge-pathnames *pathname1 pathname2* procedure+

Returns a pathname whose components are obtained by combining those of *pathname1* and *pathname2*. The pathnames are combined by components: if *pathname1* has a non-missing component, that is the resulting component, otherwise the component from *pathname2* is used. The directory component is handled specially: if both pathnames have directory components that are lists, and the directory component from *pathname1* is relative (i.e. does not start with `root`), the the resulting directory component is formed by appending *pathname1*'s component to *pathname2*'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"]
```

pathname-default-host *pathname host* procedure+
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*.

pathname-default *pathname host 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
                          host device directory
                          name type version)
  (make-pathname (or (pathname-host pathname) host)
                 (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)))
```

15.2 Working Directory

A pathname may be absolute or relative. An *absolute pathname* is a complete path from the top level of the file system to the destination resource (under unix, a filename corresponding to an absolute pathname begins with the slash character, '`/`'). A *relative pathname* is a partial path that's interpreted relative to the current 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.

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 using `->pathname`, `pathname->absolute-pathname`, and `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.

```
(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 actual directory in the file system.

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 canonicalized in exactly as does `set-working-directory-pathname!`. The binding is performed in exactly the same way as fluid binding of a variable (see Section 2.3 [Fluid Binding], page 24).

pathname->absolute-pathname *pathname* procedure+

Converts *pathname* into an absolute pathname. If *pathname* is relative, it is converted to absolute form starting at the current working directory. If *pathname* is already an absolute pathname, it is returned.

pathname-absolute? *pathname* procedure+

Returns `#t` if *pathname* is an absolute rather than relative pathname object; otherwise returns `#f`. All pathnames are either absolute or relative, so if this procedure returns `#f`, the argument is a relative pathname.

15.3 File Manipulation

file-exists? *filename* procedure+

Returns `#t` if *filename* is an existing file or directory; otherwise returns `#f`. 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*.

canonicalize-input-filename *filename* procedure+

If *filename* refers to an existing file or directory, returns as a string the absolute pathname of the file (or directory). If *filename* doesn't refer to an existing file or directory, an error is signalled.

canonicalize-output-filename *filename* procedure+

Returns as a string the absolute pathname of *filename*, whether or not the file or directory that *filename* refers to exists.

```
(pwd)
     ⇒  #[pathname "/usr/morris/"]
(canonicalize-output-filename "foo")
     ⇒  "/usr/morris/foo"
```

pathname->input-truename *pathname* procedure+

Converts *pathname* into the corresponding absolute pathname. If pathname doesn't exist, returns `#f`.

pathname->output-truename *pathname* procedure+

Converts *pathname* into the corresponding absolute pathname.

file-modification-time *filename* procedure+

Returns the modification time of *filename* as an exact integer. The result may be compared to other modification times using ordinary integer arithmetic. If *filename* names a file that does not exist, returns `#f`. If *filename* names a symbolic link, this returns the modification time of the file linked to, not the link itself.

file-directory? *filename* procedure+

Returns `#t` if the file named *filename* exists and is a directory. Otherwise returns `#f`. If *filename* names a symbolic link, this examines the file linked to, not the link itself.

file-symbolic-link? *filename* procedure+

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, this procedure returns `#f`.

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`. The information returned is a vector of 10 items:

1. 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.
2. The number of links to the file.
3. The user id of the file's owner, an exact non-negative integer.
4. The group id of the file's group, an exact non-negative integer.
5. The last access time of the file, an exact non-negative integer.
6. The last modification time of the file, an exact non-negative integer.
7. The last change time of the file, an exact non-negative integer.
8. The size of the file in bytes.
9. The mode string of the file. This is a newly allocated string showing the file's mode bits.
10. The inode number of the file, an exact non-negative integer.

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".

16 Error System

The Scheme error system provides a uniform mechanism for the signalling of errors and other exceptional conditions. For most purposes, the only part of the error system that is needed is the special form `error`, which is used to signal simple errors, specifying a message and some irritant objects (see Section 16.1 [Simple Errors], page 188). In addition, an option to `error` permits users to do simple formatting of their error messages (see Section 16.3 [Error Messages], page 190).

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 the specification of 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 kind of behavior that is useful 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 imposes a partial order on the types, because each type is allowed to have multiple generalizations. This is allowed because some condition types can be generalized in several different ways. An example of this is "floating-point divide by zero", which can be generalized to either a "floating-point overflow" or a "divide by zero". The latter generalization, "divide by zero", cannot be a specialization of the former, because it can also be signalled by arithmetic on numbers other than floating-point.

To summarize, the error system provides facilities for the following tasks. The sections that follow will describe these facilities in more detail.

Signalling conditions
: Conditions 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 conditions
: 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.

Classification of conditions
: Each condition has a type, which is 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.

Packaging condition state
: Each condition is represented by an explicit object. Condition objects contain information about the nature of the condition as well as information that describes the state of the computation from which the condition arose.

16.1 Simple Errors

The simplest error-signalling mechanism is the special form `error`. It allows the programmer to signal errors by specifying an error message and providing a list of some objects (*irritants*) that are relevant to the error.

error *message irritant ...* special form+

Signals an error. This special form expands into the following code:

```
(error-procedure message (list irritant ...) (the-environment))
```

The use of `the-environment` would normally force the environment in which it appears to be an interpreter environment. However, the compiler treats `error` expressions specially so that errors signalled from compiled code do not supply the environment argument to `error-procedure`.

error-procedure *message irritants environment* procedure+

Signals an error. *Message* is normally a short string that summarizes the error, and *irritants* is a list of objects that contain interesting information about the error. *Environment* is the environment in which the error occurred.

Normally, the message and irritants are used to build a condition whose type is `error-type:vanilla`, *environment* is attached to that condition, and then the condition is signalled. However, if *message* is a `condition-type` object, then that object is used (instead of `error-type:vanilla`) to build the condition.

The argument *environment* is used by the standard error handler. Currently there is no mechanism to retrieve this information.

error-type:vanilla condition type+

This is the error type used by `error-procedure` when it signals anonymous errors. Do not use this type for signalling new errors, as some of the system code expects to find the message in a special place for errors of this type. This object is made public solely so that condition handlers can be bound to it.

warn *message irritant ...* procedure+

This procedure takes arguments just like `error`, formats and prints a message in the usual way, but does not signal an exception. The output goes to the output-port of the nearest `cmdl` object. In the future this may be made customizable in some ways, such as allowing conditional error signalling based on a flag.

16.2 Error Handler

In the absence of more specific handling, errors invoke the *standard error handler*. This handler normally prints an error message and enters a new REP loop.

standard-error-handler *condition* procedure+

This procedure is used to handle error conditions that are not otherwise taken care of. It can be useful when building custom error handlers.

While the standard error handler is executing, the following procedures provide useful information.

error-condition procedure+

This procedure returns the condition that was passed to the standard error handler as its argument. If the standard error handler is not executing, this procedure returns `#f`.

error-message procedure+
error-irritants procedure+
error-continuation procedure+

These procedures extract standard components from `(error-condition)`. `error-continuation` returns `#f` if `error-condition` does.

16.3 Error Messages

The error system provides a simple formatting language that allows the programmer to have some control over the printing of error messages. The basic idea is as follows.

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, which is 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 #\newline)
         (error-irritant/noise "within procedure")
         procedure))
```

This would format as follows:

```
(error-within-procedure "bad widget" 'widget-32 'invert-widget) error

bad widget widget-32
within procedure invert-widget
```

Note the use of a separate noise object for the newline. In general, for characters such as newline or formfeed (i.e. non-graphic characters), this is desirable since it makes it easier for the formatter to notice formatting characters with special meanings and handle them specially should it be necessary.

Here are the operations supporting error messages:

format-error-message *message irritants port* procedure+

Message must be a string, *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*.

error-irritant/noise? *object* procedure+

Returns `#t` if *object* was created by `error-irritant/noise`; otherwise returns `#f`.

error-irritant/noise-value *noise* procedure+

Returns the value of the noise object *noise*. This is the object that was passed to `error-irritant/noise` as an argument when *noise* was created.

error-irritants/sans-noise procedure+

This returns the value of `error-irritants` with the noise objects removed. It could have been written:

```
(define (error-irritants/sans-noise)
  (list-transform-negative (error-irritants)
    error-irritant/noise?))
```

16.4 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 inheritance relationships between types.

Condition types are defined by the following operations. Any argument called *condition-type* is assumed to be a condition type object.

make-condition-type *generalizations reporter* procedure+

This procedure creates and returns a new condition type. *Generalizations* must be a list of condition types; the resulting condition type will be a specialization of each of these types. *Reporter* is a procedure of two arguments, a condition instance and an output port, which will write a concise description of the condition on the output port. As a special option, *reporter* may be a string, in which case a standard error message will be printed using that string.

condition-type? *object* procedure+

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

guarantee-condition-type *object* procedure+

Signals an error if *object* is not a condition type. Returns *object* as its value otherwise.

condition-type/generalizations *condition-type* procedure+

Returns the generalizations of *condition-type*. This is the reflexive and transitive closure of the *generalizations* argument to `make-condition-type`, i.e. all of the members of *generalizations* plus all of their generalizations as well. This list always contains *condition-type*.

condition-type/reporter *condition-type* procedure+

Returns the report procedure for *condition-type*.

condition-type/properties *condition-type* procedure+

Each condition type contains a 1D table for associating arbitrary properties with the condition type. This operation returns the table associated with *condition-type*.

condition-type/error? *condition-type* procedure+

This predicate is true only of condition types that are considered to be errors. Condition types satisfying this predicate are treated specially under certain cirumstances, and are sometimes referred to as *error types*.

The following two expressions are equivalent (as predicates):

```
(condition-type/error? x)
(memq condition-type:error (condition-type/generalizations x))
```

condition-type:error condition type+

The value of this variable a condition type with no generalizations that is used to mark condition types as being errors.

make-error-type *generalizations reporter* procedure+

This operation is like `make-condition-type`, except that if none of *generalizations* satisfies `condition-type/error?`, it adds `condition-type:error` to the set of generalizations.

error-type? *object* procedure+

Returns `#t` if *object* is a condition type that satisfies `condition-type/error?`; otherwise returns `#f`.

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 *condition* objects, also called *condition instances*.

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 when condition handlers want to restart the computation in some way, or sometimes for analyzing the computation to learn more about the context in which the condition occurred.

The following operations define the condition datatype. Any argument called *condition* is assumed to be a condition object.

make-condition *condition-type irritants continuation* procedure+

Makes a new condition instance, with a type of *condition-type*. *Irritants* is a list of arbitrary objects, and *continuation* must be a continuation, normally the continuation of the computation which caused the condition.

condition? *object* procedure+

Returns true iff *object* is a condition instance.

error? *object* procedure+

Returns `#t` if *object* is a condition instance that satisfies `condition/error?`; otherwise returns `#f`.

guarantee-condition *object* procedure+

Signals an error if *object* is not a condition instance. Returns *object* otherwise.

condition/type *condition* procedure+

Returns the condition type associated with *condition*.

condition/irritants *condition* procedure+

Returns the irritants associated with *condition*.

condition/continuation *condition* procedure+

Returns the continuation associated with *condition*.

condition/write-report *condition* [*output-port*] procedure+

Writes a concise description of *condition* to *output-port*, which defaults to the current output port.

condition/report-string *condition* procedure+

Returns a newly allocated string that is a concise description of *condition*.

condition/properties *condition* procedure+

Each condition instance contains a 1D table for associating arbitrary properties with the condition. This procedure returns the table associated with *condition*.

condition/internal? *condition* procedure+

This predicate is true of certain conditions that normally should not be handled by user code. Condition handlers that handle "all" conditions should ignore conditions satisfying this predicate, unless the handler's writer has a good understanding of internal conditions.

condition/generalizations *condition* procedure+
condition/error? *condition* procedure+
condition/reporter *condition* procedure+

These operations access the corresponding components in the type of *condition*. For example, the following expressions are equivalent:

```
(condition/generalizations condition)
(condition-type/generalizations (condition/type condition))
```

Note that condition instances satisfying the predicate `condition/error?` are sometimes referred to as *error conditions*, or even *errors*.

16.6 Condition Signalling

Once a condition instance has been created using `make-condition`, 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.

There is a single procedure responsible for signalling conditions. All other signalling mechanisms go through this procedure:

signal-condition *condition* [*default-handler*] procedure+

Signals *condition*. *Default-handler*, if given, must be a procedure of one argument. The signalling procedure attempts to find a handler for the condition, using the following rules:

- The set of condition handlers is searched, until it finds one that will handle any of the condition types specified by the generalizations of *condition*. Each handler specifies, at the time it is bound, a set of condition types it will handle; if the intersection of that set and *condition*'s generalizations is not empty, the handler is selected.

  If a handler is found, it is invoked on *condition*. If it returns `#f`, that means the handler has declined to handle the condition and the search continues. Otherwise the handler's value is returned as the value of `signal-condition`.
- If no condition handler is found, and the argument *default-handler* is supplied, it is invoked on *condition*; its value is returned as the value of `signal-condition`.
- Otherwise, `#f` is returned, indicating that no handler could be found.

signal-error *condition* procedure+

Signals *condition*, which must be a condition instance. If this condition is not otherwise handled, the standard error handler is invoked. Equivalent to `(signal-condition` *condition* `standard-error-handler)`.

16.7 Condition Handling

Condition handling refers to the act of defining the behavior of a signalled condition. This is controlled by the binding of *condition handlers*.

A *condition handler* is a procedure of one argument. When a handler is invoked by `signal-condition`, a condition is supplied to the handler as its argument. The handler may return a value: if the value is `#f`, this indicates that the handler has decided not to handle this particular condition. In that case, `signal-condition` continues its search for another handler. Otherwise the value returned by the handler is returned as the value of `signal-condition`.

Often, though, the handler will not return a value. Usually a condition handler will exit by invoking some continuation. Sometimes it will signal a different condition instead, or signal the same condition again. This latter action, *resignalling* the same condition, is equivalent to returning `#f` from the handler. This is because when the handler is invoked, the set of condition-handler bindings is unwound to the point at which the handler was bound. In other words, when a condition handler is bound, the set of condition handlers that is in effect at the time of binding is the set that will be in effect when the handler is invoked.

A condition handler is *bound* by specifying that it will handle a condition whose generalizations intersect a given set of condition types. Condition handlers are bound using a mechanism very similar to fluid binding of variables. A condition handler binding has a dynamic extent rather than a lexical scope, that is, it is effective for a specified time segment rather than for a specified code segment.

The order in which the bindings are searched (by `signal-condition`) is the opposite from the order in which they are bound. Thus, more recently bound handlers appear earlier in the search order.

bind-condition-handler *condition-types handler thunk* procedure+

Executes *thunk*, handling any condition whose generalizations intersect *condition-types* by passing it to *handler*. *Condition-types* must be a list of condition-type objects. If *condition-types* is the empty list, it means *handler* should handle *all* conditions, regardless of their type; such handlers should ignore conditions satisfying the predicate `condition/internal?`.

16.8 Predefined Errors

error-type:wrong-type-argument condition type+

Signalled when the argument to a procedure is determined to have an incorrect type. For example, this error is signalled by `car` if its argument is not a pair. This is a specialization of `error-type:illegal-argument`.

error:illegal-datum *object* [*procedure-name*] procedure+

Signals `error-type:wrong-type-argument`. *Object* is the argument in question, and `procedure-name`, if supplied, is the name of the procedure that determined the argument's incorrectness.

error-type:bad-range-argument condition type+

Signalled when the argument to a procedure is determined to have the correct type but is not in the acceptable range. For example, this error is signalled by `vector-ref` if its second argument is an exact negative integer. This is a specialization of `error-type:illegal-argument`.

error:datum-out-of-range *object* [*procedure-name*] procedure+

Signals `error-type:bad-range-argument`. *Object* is the argument in question, and `procedure-name`, if supplied, is the name of the procedure that determined the argument's incorrectness.

error-type:illegal-argument condition type+

This type is not signalled – one of its specializations is signalled when the argument to a procedure is determined to be incorrect in some fashion.

error-type:open-file condition type+

Signalled if an error occurs while trying to open a file for input or output.

error-type:file condition type+

This type is not signalled – one of its specializations is signalled when any I/O error occurs. The name of this type is a misnomer; it can be signalled by any I/O operation, whether or not it involves files.

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 two types of graphics device are implemented.

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.

17.1 Opening and Closing of Graphics Devices

graphics-type-available? *graphics-device-type* procedure+

This predicate returns `#t` if *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*. This is useful because a given graphics device type may exist as an object in the Scheme runtime environment even though the primitive procedures required to support that type do not. This predicate determines when full support is available.

make-graphics-device *graphics-device-type object ...* procedure+

This operation creates a graphics device object. The first argument is 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). This procedure opens and initializes the device, which remains valid until explicitly closed by the procedure `graphics-close`. In the current implementation of MIT Scheme, if this object is garbage-collected, the graphics device remains open, and any resources it is using are not released. In the future the garbage collector may be changed to automatically close such devices.

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. Thus, 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*. It is 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*. It 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*. It 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 X Graphics

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.

17.8.1 X Graphics Type

x-graphics-device-type variable+

This is the graphics device type for X windows. X windows are opened as follows:

```
(make-graphics-device x-graphics-device-type
                      display
                      geometry
                      #!optional 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. 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 and not `#f`, prevents the window from being mapped after it is created.

The window is initialized using the resource name `"scheme-graphics"`, and is sensitive to the following resource properties:

Property	Class	Default
`geometry`	`Geometry`	[none]
`font`	`Font`	`9x15`
`borderWidth`	`BorderWidth`	`2`
`internalBorder`	`BorderWidth`	[border width]
`background`	`Background`	`white`
`foreground`	`Foreground`	`black`
`borderColor`	`BorderColor`	[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"`.

17.8.2 Utilities for X Graphics

x-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 the usual X display name; however, `#f` is also allowed, meaning to use the value of the unix environment variable `DISPLAY`.

x-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 those 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.

17.8.3 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.

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. 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
unmap-window operation+ on x-graphics-device

These operations control the mapping of windows. They correspond directly to the Xlib procedures `XMapWindow` and `XUnmapWindow`.

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 the Xlib procedure `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 the Xlib procedure `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 the Xlib procedure `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.

starbase-filename operation+ on x-graphics-device

On Hewlett-Packard computers that support Starbase graphics, this operation returns a character string that can be used to open the device's window as a Starbase graphics device using the "sox11" driver. Note that the default distribution of Scheme for HP computers does not include support for Starbase — you must rebuild the microcode to get this support.

17.9 Starbase Graphics

On Hewlett-Packard computers under the HP-UX operating system, Scheme supports graphics through the Starbase graphics library. Note that the default distribution of Scheme for HP computers does not include support for Starbase — you must rebuild the microcode to get this support.

starbase-graphics-device-type variable+

This is the device type for Starbase graphics devices. A Starbase device is opened as follows:

```
(make-graphics-device starbase-graphics-device-type
                      device-name
                      driver-name)
```

where *device-name* and *driver-name* are strings that are used as the device and driver arguments to the Starbase `gopen` call. The device is opened with kind `OUTDEV` and mode `0`. The device is initialized to have a mapping mode of `DISTORT`, and a line color index of `1`.

write-image-file *filename invert?* operation+ on starbase-graphics-device

This operation writes an image of the Starbase device's display in the file specified by *filename*. The image is formatted to print on an HP Laserjet printer. Normally pixels with a color index of 0 are not drawn by the printer, and all other pixels are; this results in the background being white and the foreground being black in the printed image. If *invert?* is not `#f`, this is reversed: the background is printed as black and the foreground is not printed.

color-map-size operation+ on starbase-graphics-device

Returns, as an exact non-negative integer, the number of entries in the color map for the device.

define-color *color-index red green blue* operation+ on starbase-graphics-device

Defines the color associated with the color-map index *color-index*. *Color-index* must be an exact non-negative integer strictly less than the number of entries in the color map. *Red*, *green*, and *blue* must be real numbers in the range 0 to 1 inclusive, which define the color to be put in the map.

set-line-color *color-index* operation+ on starbase-graphics-device

Changes the foreground color used in graphics operations for this device. *Color-index* must be an exact non-negative integer strictly less than the number of entries in the color map. Graphics drawn after this operation is invoked will appear in this new color.

The text drawn by a Starbase device is controlled by the following characteristics:

Aspect
: The *aspect* of a character is its height-to-width ratio, a real number. By default, this has the value `1`.

Height
: The *height* of a character in virtual device coordinates, a real number. This is measured along the "up vector", which is defined by the slant of the character. By default, the height is `.1`.

Rotation
: The *rotation* of a character defines the direction in which the characters are drawn. It is specified as a real number in degrees, but only 4 values have any meaning: `0`, `90`, `180`, and `270`. `0` draws left-to-right with upright characters; `90` draws top-to-bottom with characters on their right side; `180` draws right-to-left with upside-down characters; `270` draws bottom-to-top with characters on their left side. The default rotation is `0`.

Slant
: The *slant* of a character defines the "up vector"; it is a real number which is the tangent of the angle between the character's "vertical" (defined by the rotation), and the "up vector", measured clockwise. The default slant is `0`.

text-aspect operation+ on starbase-graphics-device
text-height operation+ on starbase-graphics-device
text-rotation operation+ on starbase-graphics-device
text-slant operation+ on starbase-graphics-device

These operations return the current values of the text characteristics.

set-text-aspect *aspect* operation+ on starbase-graphics-device
set-text-height *height* operation+ on starbase-graphics-device
set-text-rotation *rotation* operation+ on starbase-graphics-device
set-text-slant *slant* operation+ on starbase-graphics-device

These operations alter the current values of the text characteristics. They have no effect on text drawn prior to their invocation.

Index of Procedures, Special Forms, and Variables

<

1

2

A

B

C

D

E

G

H

I

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z

Index of Concepts

B

C

D

E

F

G

H

I

K

L

M

N

O

P

T

U

V

W

X

Z

Short Contents

Table of Contents