[Go to first, previous, next page; contents; index]

Chapter 6

Standard procedures

This chapter describes Scheme's built-in procedures. The initial (or ``top level'') Scheme environment starts out with a number of variables bound to locations containing useful values, most of which are primitive procedures that manipulate data. For example, the variable `abs` is bound to (a location initially containing) a procedure of one argument that computes the absolute value of a number, and the variable `+` is bound to a procedure that computes sums. Built-in procedures that can easily be written in terms of other built-in procedures are identified as ``library procedures''.

A program may use a top-level definition to bind any variable. It may subsequently alter any such binding by an assignment (see 4.1.6). These operations do not modify the behavior of Scheme's built-in procedures. Altering any top-level binding that has not been introduced by a definition has an unspecified effect on the behavior of the built-in procedures.

6.1 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?`.

procedure: `(eqv?` obj_1 obj_2`)`

The `eqv?` procedure defines a useful equivalence relation on objects. Briefly, it returns `#t` if obj_1 and obj_2 should normally be regarded as the same object. This relation is left slightly open to interpretation, but the following partial specification of `eqv?` holds for all implementations of Scheme.

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

- obj_1 and obj_2 are both `#t` or both `#f`.
- obj_1 and obj_2 are both symbols and

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

Note: This assumes that neither obj_1 nor obj_2 is an ``uninterned

symbol" as alluded to in section 6.3.3. This report does not presume to specify the behavior of `eqv?` on implementation-dependent extensions.

- *obj*$_1$ and *obj*$_2$ are both numbers, are numerically equal (see `=`, section 6.2), and are either both exact or both inexact.
- *obj*$_1$ and *obj*$_2$ are both characters and are the same character according to the `char=?` procedure (section 6.3.4).
- both *obj*$_1$ and *obj*$_2$ are the empty list.
- *obj*$_1$ and *obj*$_2$ are pairs, vectors, or strings that denote the same locations in the store (section 3.4).
- *obj*$_1$ and *obj*$_2$ are procedures whose location tags are equal (section 4.1.4).

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

- *obj*$_1$ and *obj*$_2$ are of different types (section 3.2).
- one of *obj*$_1$ and *obj*$_2$ is `#t` but the other is `#f`.
- *obj*$_1$ and *obj*$_2$ are symbols but

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

- one of *obj*$_1$ and *obj*$_2$ is an exact number but the other is an inexact number.
- *obj*$_1$ and *obj*$_2$ are numbers for which the `=` procedure returns `#f`.
- *obj*$_1$ and *obj*$_2$ are characters for which the `char=?` procedure returns `#f`.
- one of *obj*$_1$ and *obj*$_2$ is the empty list but the other is not.
- *obj*$_1$ and *obj*$_2$ are pairs, vectors, or strings that denote distinct locations.
- *obj*$_1$ and *obj*$_2$ are procedures that would behave differently (return different value(s) or have different side effects) for some arguments.

```
(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
```

Since it is an error to modify constant objects (those returned by literal expressions), implementations are permitted, though not required, to share structure between constants where appropriate. Thus the value of `eqv?` on constants is sometimes implementation-dependent.

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

> *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.

procedure: `(eq? `obj_1 obj_2`)`

`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, procedures, 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?`.

library procedure: `(equal? `obj_1 obj_2`)`

`Equal?` recursively compares the contents of pairs, vectors, and strings, applying `eqv?` on other objects such as numbers and symbols. 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
```

6.2 Numbers

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 [20] 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*.

6.2.1 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.

6.2.2 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 6.2.3.

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.

6.2.3 Implementation restrictions

Implementations of Scheme are not required to implement the whole tower of subtypes given in section 6.2.1, 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.

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:

```
+             -             *
quotient      remainder     modulo
max           min           abs
numerator     denominator   gcd
lcm           floor         ceiling
truncate      round         rationalize
expt
```

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 [12].

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. 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.

6.2.4 Syntax of numerical constants

The syntax of the written representations for numbers is described formally in section 7.1.1. Note that case is not significant in numerical constants.

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

A numerical constant may be specified to be either exact or inexact by a prefix. The prefixes are `#e` for exact, and `#i` for inexact. An exactness prefix may appear before or after any radix prefix that is used. If the written representation of a number has no exactness prefix, the constant may be either inexact or exact. It is inexact if it contains a decimal point, an exponent, or a ``#'' character in the place of a digit, otherwise it is exact. In systems with inexact numbers of varying precisions it may be useful to specify the precision of a constant. For this purpose, numerical constants may be written with an exponent marker that indicates the desired precision of the inexact representation. The letters `s`, `f`, `d`, and `l` specify the use of *short*, *single*, *double*, and *long* precision, respectively. (When fewer than four internal inexact representations exist, the four size specifications are mapped onto those available. For example, an implementation with two internal representations may map short and single together and long and double together.) In addition, the exponent marker `e` specifies the default precision for the implementation. The default precision has at least as much precision as *double*, but implementations may wish to allow this default to be set by the user.

```
3.14159265358979F0
        Round to single --- 3.141593
0.6L0
        Extend to long --- .600000000000000
```

6.2.5 Numerical operations

The reader is referred to section 1.3.3 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.

procedure: `(number? obj)`
procedure: `(complex? obj)`
procedure: `(real? obj)`
procedure: `(rational? obj)`
procedure: `(integer? obj)`

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. 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.

Note: In many implementations the `rational?` procedure will be the same as `real?`, and the `complex?` procedure will be the same as `number?`, but unusual implementations may be able to represent some irrational numbers exactly or may extend the number system to support some kind of non-complex numbers.

procedure: `(exact? z)`
procedure: `(inexact? z)`

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

procedure: `(= z1 z2 z3 ...)`
procedure: `(< x1 x2 x3 ...)`
procedure: `(> x1 x2 x3 ...)`
procedure: `(<= x1 x2 x3 ...)`
procedure: `(>= x1 x2 x3 ...)`

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

These predicates are required to be transitive.

Note: 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.

library procedure: `(zero? z)`
library procedure: `(positive? x)`
library procedure: `(negative? x)`
library procedure: `(odd? n)`
library procedure: `(even? n)`

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

library procedure: `(max x1 x2 ...)`
library procedure: `(min x1 x2 ...)`

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.

procedure: `(+ `z_1 `...)`
procedure: `(* `z_1 `...)`

These procedures return the sum or product of their arguments.

```
(+ 3 4)                      ===>  7
(+ 3)                        ===>  3
(+)                          ===>  0
(* 4)                        ===>  4
(*)                          ===>  1
```

procedure: `(- `z_1 z_2`)`
procedure: `(- `z`)`
optional procedure: `(- `z_1 z_2 `...)`
procedure: `(/ `z_1 z_2`)`
procedure: `(/ `z`)`
optional procedure: `(/ `z_1 z_2 `...)`

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
```

library procedure: `(abs `x`)`

`Abs` returns the absolute value of its argument.

```
(abs -7)                     ===>  7
```

procedure: `(quotient `n_1 n_2`)`
procedure: `(remainder `n_1 n_2`)`
procedure: `(modulo `n_1 n_2`)`

These procedures implement number-theoretic (integer) division. n_2 should be non-zero. All three procedures return integers. If n_1/n_2 is an integer:

```
(quotient n1 n2)             ===> n1/n2
(remainder n1 n2)            ===> 0
```

```
(modulo n1 n2)              ===> 0
```

If n_1/n_2 is not an integer:

```
(quotient n1 n2)            ===> nq
(remainder n1 n2)           ===> nr
(modulo n1 n2)              ===> nm
```

where n_q is n_1/n_2 rounded towards zero, $0 < |n_r| < |n_2|$, $0 < |n_m| < |n_2|$, n_r and n_m differ from n_1 by a multiple of n_2, n_r has the same sign as n_1, and n_m has the same sign as n_2.

From this we can conclude that for integers n_1 and n_2 with n_2 not equal to 0,

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

provided all numbers involved in that computation are exact.

```
(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
```

library procedure: `(gcd n1 ...)`
library procedure: `(lcm n1 ...)`

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
```

procedure: `(numerator q)`
procedure: `(denominator q)`

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
```

procedure: `(floor x)`
procedure: `(ceiling x)`
procedure: `(truncate x)`
procedure: `(round x)`

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 default rounding mode specified 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.

```
(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
```

library procedure: `(rationalize x y)`

`Rationalize` returns the *simplest* rational number differing from *x* by no more than *y*. A rational number r_1 is *simpler* than another rational number r_2 if $r_1 = p_1/q_1$ and $r_2 = p_2/q_2$ (in lowest terms) and $|p_1| \leq |p_2|$ and $|q_1| \leq |q_2|$. 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
```

procedure: `(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)`

These procedures are part of every implementation that supports general real numbers; they compute the usual transcendental functions. `Log` computes the natural logarithm of z (not the base ten logarithm). `Asin`, `acos`, and `atan` compute arcsine ($\sin^{-1}$), arccosine ($\cos^{-1}$), and arctangent ($\tan^{-1}$), respectively. The two-argument variant of `atan` computes `(angle (make-rectangular x y))` (see below), even in implementations that don't support general complex numbers.

In general, the mathematical functions log, arcsine, arccosine, and arctangent are multiply defined. The value of `log` z is defined to be the one whose imaginary part lies in the range from $-\pi$ (exclusive) to π (inclusive). `log` 0 is undefined. With `log` defined this way, the values of $\sin^{-1} z$, $\cos^{-1} z$, and $\tan^{-1} z$ are according to the following formulæ:

$$\sin^{-1} z = -i \log (i z + (1 - z^2)^{1/2})$$

$$\cos^{-1} z = \pi / 2 - \sin^{-1} z$$

$$\tan^{-1} z = (\log (1 + i z) - \log (1 - i z)) / (2 i)$$

The above specification follows [27], which in turn cites [19]; 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.

procedure: `(sqrt z)`

Returns the principal square root of z. The result will have either positive real part, or zero real part and non-negative imaginary part.

procedure: `(expt z1 z2)`

Returns z_1 raised to the power z_2. For $z_1 \neq 0$

$$z_1^{z_2} = e^{z_2 \log z_1}$$

0^z is 1 if $z = 0$ and 0 otherwise.

procedure: `(make-rectangular x1 x2)`
procedure: `(make-polar x3 x4)`
procedure: `(real-part z)`
procedure: `(imag-part z)`
procedure: `(magnitude z)`
procedure: `(angle z)`

These procedures are part of every implementation that supports general complex numbers. Suppose x_1, x_2, x_3, and x_4 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 x1 x2)        ===> z
(make-polar x3 x4)              ===> z
(real-part z)                          ===> x1
(imag-part z)                          ===> x2
(magnitude z)                          ===> |x3|
(angle z)                              ===> xangle
```

where $-\pi < x_{angle} \leq \pi$ with $x_{angle} = x_4 + 2\pi n$ for some integer n.

> *Rationale:* `Magnitude` is the same as `abs` for a real argument, but `abs` must be present in all implementations, whereas `magnitude` need only be present in implementations that support general complex numbers.

procedure: `(exact->inexact z)`
procedure: `(inexact->exact z)`

`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.

`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.

These procedures implement the natural one-to-one correspondence between exact and inexact integers throughout an implementation-dependent range. See section 6.2.3.

6.2.6 Numerical input and output

procedure: `(number->string z)`
procedure: `(number->string z radix)`

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 z 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 [3, 5]; 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 *z* is not a complex number or is a complex number with a non-rational real or imaginary part.
>
> *Rationale:* If *z* 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.

procedure: `(string->number` *`string`*`)`
procedure: `(string->number` *`string radix`*`)`

Returns a number of the maximally precise representation expressed by the given *string*. *Radix* must be an exact integer, either 2, 8, 10, or 16. If supplied, *radix* is a default radix that may be overridden by an explicit radix prefix in *string* (e.g. `"#o177"`). If *radix* is not supplied, then the default radix is 10. If *string* is not a syntactically valid notation for a number, then `string->number` returns `#f`.

```
(string->number "100")                  ===>  100
(string->number "100" 16)               ===>  256
(string->number "1e2")                  ===>  100.0
(string->number "15##")                 ===>  1500.0
```

> *Note:* The domain of `string->number` may be restricted by implementations in the following ways. `String->number` is permitted to return `#f` whenever *string* contains an explicit radix prefix. If all numbers supported by an implementation are real, then `string->number` is permitted to return `#f` whenever *string* uses the polar or rectangular notations for complex numbers. If all numbers are integers, then `string->number` may return `#f` whenever the fractional notation is used. If all numbers are exact, then `string->number` may return `#f` whenever an exponent marker or explicit exactness prefix is used, or if a `#` appears in place of a digit. If all inexact numbers are integers, then `string->number` may return `#f` whenever a decimal point is used.

6.3 Other data types

This section describes operations on some of Scheme's non-numeric data types: booleans, pairs, lists, symbols, characters, strings and vectors.

6.3.1 Booleans

The standard boolean objects for true and false are written as `#t` and `#f`. What really matters, though, are the objects that the Scheme conditional expressions (`if`, `cond`, `and`, `or`, `do`) treat as true or false. The phrase ``a true value'' (or sometimes just ``true'') means any object treated as true by the conditional expressions, and the phrase ``a false value'' (or ``false'') means any object treated as false by the conditional expressions.

Of all the standard Scheme values, only `#f` counts as false in conditional expressions. Except for `#f`, all standard Scheme values, including `#t`, pairs, the empty list, symbols, numbers, strings, vectors, and procedures, count as true.

> *Note:* Programmers accustomed to other dialects of Lisp should be aware that Scheme distinguishes both `#f` and the empty list from the symbol `nil`.

Boolean constants evaluate to themselves, so they do not need to be quoted in programs.

```
#t                 ===>  #t
#f                 ===>  #f
'#f                ===>  #f
```

library procedure: `(not` *obj*`)`

`Not` returns `#t` if *obj* is false, and returns `#f` otherwise.

```
(not #t)           ===>  #f
(not 3)                  ===>  #f
(not (list 3))           ===>  #f
(not #f)           ===>  #t
(not '())                ===>  #f
(not (list))             ===>  #f
(not 'nil)               ===>  #f
```

library procedure: `(boolean?` *obj*`)`

`Boolean?` returns `#t` if *obj* is either `#t` or `#f` and returns `#f` otherwise.

```
(boolean? #f)          ===>  #t
(boolean? 0)                 ===>  #f
(boolean? '())               ===>  #f
```

6.3.2 Pairs and 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.

Note: The above definitions imply that all lists have finite length and are terminated by the empty list.

The most general notation (external representation) for Scheme pairs is the ``dotted'' notation `(`c_1 `.` c_2`)` where c_1 is the value of the car field and c_2 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,

```
(a b c d e)
```

and

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

are equivalent notations for a list of symbols.

A chain of pairs not ending 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:

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

is equivalent to

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

Whether a given pair is a list depends upon what is stored in the cdr field. When the `set-cdr!` procedure is used, an object can be a list one moment and not the next:

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

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. That is, according to Scheme's grammar, every <expression> is also a <datum> (see section 7.1.2). Among other things, this permits the use of the `read` procedure to parse Scheme programs. See section 3.3.

procedure: `(pair? obj)`

`Pair?` returns `#t` if *obj* is a pair, and otherwise returns `#f`.

```
(pair? '(a . b))                ===>  #t
(pair? '(a b c))                ===>  #t
(pair? '())                     ===>  #f
(pair? '#(a b))                 ===>  #f
```

procedure: `(cons` obj_1 obj_2`)`

Returns a newly allocated pair whose car is obj_1 and whose cdr is obj_2. The pair is guaranteed to be different (in the sense of `eqv?`) from every 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)
```

procedure: `(car pair)`

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
```

procedure: `(cdr pair)`

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
```

procedure: `(set-car! pair obj)`

Stores *obj* 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
```

procedure: `(set-cdr! pair obj)`

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

library procedure: `(caar pair)`
library procedure: `(cadr pair)`

library procedure: `(cdddar` *`pair`*`)`
library procedure: `(cddddr` *`pair`*`)`

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

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

Arbitrary compositions, up to four deep, are provided. There are twenty-eight of these procedures in all.

library procedure: `(null?` *`obj`*`)`

Returns `#t` if *obj* is the empty list, otherwise returns `#f`.

library procedure: `(list?` *`obj`*`)`

Returns `#t` if *obj* is a list, otherwise returns `#f`. By definition, all lists have finite length and are terminated by the empty list.

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

library procedure: `(list` *`obj ...`*`)`

Returns a newly allocated list of its arguments.

```
(list 'a (+ 3 4) 'c)                    ===>  (a 7 c)
(list)                                  ===>  ()
```

library procedure: `(length` *`list`*`)`

Returns the length of *list*.

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

library procedure: `(append` *`list ...`*`)`

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

```
(append '(x) '(y))                      ===>  (x y)
(append '(a) '(b c d))                  ===>  (a b c d)
(append '(a (b)) '((c)))                ===>  (a (b) (c))
```

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
```

library procedure: `(reverse` *`list`*`)`

Returns a newly allocated list consisting of the 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)
```

library procedure: `(list-tail` *`list k`*`)`

Returns the sublist of *list* obtained by omitting the first *k* elements. It is an error if *list* has fewer than *k* elements. `List-tail` could be defined by

```
(define list-tail
  (lambda (x k)
    (if (zero? k)
        x
        (list-tail (cdr x) (- k 1)))))
```

library procedure: `(list-ref` *`list k`*`)`

Returns the *k*th element of *list*. (This is the same as the car of `(list-tail` *`list k`*`)`.) It is an error if *list* has fewer than *k* elements.

```
(list-ref '(a b c d) 2)                         ===>  c
(list-ref '(a b c d)
          (inexact->exact (round 1.8)))
               ===>  c
```

library procedure: `(memq` *`obj list`*`)`
library procedure: `(memv` *`obj list`*`)`
library procedure: `(member` *`obj list`*`)`

These procedures return the first sublist of *list* whose car is *obj*, where the sublists of *list* are the non-empty lists returned by `(list-tail` *`list k`*`)` for *k* less than the length of *list*. If *obj* does not occur in *list*, then `#f` (not the empty list) is returned. `Memq` uses `eq?` to compare *obj* with the elements of *list*, while `memv` uses `eqv?` and `member` uses `equal?`.

```
(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)
```

library procedure: `(assq` *`obj alist`*`)`
library procedure: `(assv` *`obj alist`*`)`
library procedure: `(assoc` *`obj alist`*`)`

Alist (for ``association list'') must be a list of pairs. These procedures find the first pair in *alist* whose car field is *obj*, and returns that pair. If no pair in *alist* has *obj* as its car, then

`#f` (not the empty list) is returned. `Assq` uses `eq?` to compare *obj* with the car fields of the pairs in *alist*, while `assv` uses `eqv?` and `assoc` uses `equal?`.

```
(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)
```

Rationale: Although they are ordinarily used as predicates, `memq`, `memv`, `member`, `assq`, `assv`, and `assoc` do not have question marks in their names because they return useful values rather than just `#t` or `#f`.

6.3.3 Symbols

Symbols are objects whose usefulness rests on the fact that two symbols are identical (in the sense of `eqv?`) if and only if their names are spelled the same way. This is exactly the property needed to represent identifiers in programs, and so most implementations of Scheme use them internally for that purpose. Symbols are useful for many other applications; for instance, they may be used the way enumerated values are used in Pascal.

The rules for writing a symbol are exactly the same as the rules for writing an identifier; see sections 2.1 and 7.1.1.

It is guaranteed that any 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?`). The `string->symbol` procedure, however, can create symbols for which this write/read invariance may not hold because their names contain special characters or letters in the non-standard case.

Note: Some implementations of Scheme have a feature known as ``slashification'' in order to guarantee write/read invariance for all symbols, but historically the most important use of this feature has been to compensate for the lack of a string data type.

Some implementations also have ``uninterned symbols'', which defeat write/read invariance even in implementations with slashification, and also generate exceptions to the rule that two symbols are the same if and only if their names are spelled the same.

procedure: `(symbol? obj)`

Returns `#t` if *obj* is a symbol, otherwise returns `#f`.

```
(symbol? 'foo)                  ===>  #t
(symbol? (car '(a b)))          ===>  #t
(symbol? "bar")                 ===>  #f
```

```
(symbol? 'nil)                  ===>  #t
(symbol? '())                   ===>  #f
(symbol? #f)              ===>  #f
```

procedure: `(symbol->string` *`symbol`*`)`

Returns the name of *symbol* as a string. If the symbol was part of an object returned as the value of a literal expression (section 4.1.2) or by a call to the `read` procedure, and its name contains alphabetic characters, then the string returned will contain characters in the implementation's preferred standard case -- some implementations will prefer upper case, others lower case. If the symbol was returned by `string->symbol`, the case of characters in the string returned will be the same as the case in the string that was passed to `string->symbol`. It is an error to apply mutation procedures like `string-set!` to strings returned by this procedure.

The following examples assume that the implementation's standard case is lower case:

```
(symbol->string 'flying-fish)
                                        ===>  "flying-fish"
(symbol->string 'Martin)                ===>  "martin"
(symbol->string
   (string->symbol "Malvina"))
                                        ===>  "Malvina"
```

procedure: `(string->symbol` *`string`*`)`

Returns the symbol whose name is *string*. This procedure can create symbols with names containing special characters or letters in the non-standard case, but it is usually a bad idea to create such symbols because in some implementations of Scheme they cannot be read as themselves. See `symbol->string`.

The following examples assume that the implementation's standard case is lower case:

```
(eq? 'mISSISSIppi 'mississippi)
                ===>  #t
(string->symbol "mISSISSIppi")
                ===>  the symbol with 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
```

6.3.4 Characters

Characters are objects that represent printed characters such as letters and digits. Characters are written using the notation #\<character> or #\<character name>. For example:

```
#\a         ; lower case letter
```

```
#\A        ; upper case letter
#\(        ; left parenthesis
#\         ; the space character
#\space    ; the preferred way to write a space
#\newline  ; the newline character
```

Case is significant in #\<character>, but not in #\<character name>. If <character> in #\<character> is alphabetic, then the character following <character> must be a delimiter character such as a space or parenthesis. This rule resolves the ambiguous case where, for example, the sequence of characters ``#\space'' could be taken to be either a representation of the space character or a representation of the character ``#\s'' followed by a representation of the symbol ``pace.''

Characters written in the #\ notation are self-evaluating. That is, they do not have to be quoted in programs. Some of the procedures that operate on characters ignore the difference between upper case and lower case. The procedures that ignore case have ``-ci'' (for ``case insensitive'') embedded in their names.

procedure: `(char? obj)`

Returns `#t` if *obj* is a character, otherwise returns `#f`.

procedure: `(char=? char1 char2)`
procedure: `(char<? char1 char2)`
procedure: `(char>? char1 char2)`
procedure: `(char<=? char1 char2)`
procedure: `(char>=? char1 char2)`

These procedures impose a total ordering on the set of characters. It is guaranteed that under this ordering:

- The upper case characters are in order. For example, `(char<? #\A #\B)` returns `#t`.
- The lower case characters are in order. For example, `(char<? #\a #\b)` returns `#t`.
- The digits are in order. For example, `(char<? #\0 #\9)` returns `#t`.
- Either all the digits precede all the upper case letters, or vice versa.
- Either all the digits precede all the lower case letters, or vice versa.

Some implementations may generalize these procedures to take more than two arguments, as with the corresponding numerical predicates.

library procedure: `(char-ci=? char1 char2)`
library procedure: `(char-ci<? char1 char2)`
library procedure: `(char-ci>? char1 char2)`
library procedure: `(char-ci<=? char1 char2)`
library procedure: `(char-ci>=? char1 char2)`

These procedures are similar to `char=?` et cetera, but they treat upper case and lower case letters as the same. For example, `(char-ci=? #\A #\a)` returns `#t`. Some implementations may generalize these procedures to take more than two arguments, as

with the corresponding numerical predicates.

library procedure: `(char-alphabetic? char)`
library procedure: `(char-numeric? char)`
library procedure: `(char-whitespace? char)`
library procedure: `(char-upper-case? letter)`
library procedure: `(char-lower-case? letter)`

These procedures return `#t` if their arguments are alphabetic, numeric, whitespace, upper case, or lower case characters, respectively, otherwise they return `#f`. The following remarks, which are specific to the ASCII character set, are intended only as a guide: The alphabetic characters are the 52 upper and lower case letters. The numeric characters are the ten decimal digits. The whitespace characters are space, tab, line feed, form feed, and carriage return.

procedure: `(char->integer char)`
procedure: `(integer->char n)`

Given a character, `char->integer` returns an exact integer representation of the character. Given an exact integer that is the image of a character under `char->integer`, `integer->char` returns that character. These procedures implement order-preserving 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 domain of `integer->char`, then

```
(<= (char->integer a)
    (char->integer b))                ===>  #t

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

library procedure: `(char-upcase char)`
library procedure: `(char-downcase char)`

These procedures return a character $char_2$ such that `(char-ci=? char` $char_2$`)`. In addition, if *char* is alphabetic, then the result of `char-upcase` is upper case and the result of `char-downcase` is lower case.

6.3.5 Strings

Strings are sequences of characters. Strings are written as sequences of characters enclosed within doublequotes ("). A doublequote can be written inside a string only by escaping it with a backslash (\), as in

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

A backslash can be written inside a string only by escaping it with another backslash. Scheme does not specify the effect of a backslash within a string that is not followed by a doublequote or backslash.

A string constant may continue from one line to the next, but the exact contents of such a string are unspecified. The *length* of a string is the number of characters that it contains. This number is an exact, non-negative integer that is fixed when the string is created. The *valid indexes* of a string are the exact non-negative integers less than the length of the string. The first character of a string has index 0, the second has index 1, and so on.

In phrases such as ``the characters of *string* beginning with index *start* and ending with index *end*,'' it is understood that the index *start* is inclusive and the index *end* is exclusive. Thus if *start* and *end* are the same index, a null substring is referred to, and if *start* is zero and *end* is the length of *string*, then the entire string is referred to.

Some of the procedures that operate on strings ignore the difference between upper and lower case. The versions that ignore case have ``-ci'' (for ``case insensitive'') embedded in their names.

procedure: `(string? obj)`

Returns `#t` if *obj* is a string, otherwise returns `#f`.

procedure: `(make-string k)`
procedure: `(make-string k char)`

`Make-string` returns a newly allocated string of length *k*. If *char* is given, then all elements of the string are initialized to *char*, otherwise the contents of the *string* are unspecified.

library procedure: `(string char ...)`

Returns a newly allocated string composed of the arguments.

procedure: `(string-length string)`

Returns the number of characters in the given *string*.

procedure: `(string-ref string k)`

k must be a valid index of *string*. `String-ref` returns character *k* of *string* using zero-origin indexing.

procedure: `(string-set! string k char)`

k must be a valid index of *string*. `String-set!` stores *char* in element *k* of *string* and returns an unspecified value.

```
(define (f) (make-string 3 #\*))
(define (g) "***")
(string-set! (f) 0 #\?)          ===>  unspecified
(string-set! (g) 0 #\?)          ===>  error
(string-set! (symbol->string 'immutable)
             0
             #\?)          ===>  error
```

library procedure: `(string=? string1 string2)`

library procedure: `(string-ci=? `*`string`*`1 `*`string`*`2)`

Returns `#t` if the two strings are the same length and contain the same characters in the same positions, otherwise returns `#f`. `String-ci=?` treats upper and lower case letters as though they were the same character, but `string=?` treats upper and lower case as distinct characters.

library procedure: `(string<? `*`string`*`1 `*`string`*`2)`
library procedure: `(string>? `*`string`*`1 `*`string`*`2)`
library procedure: `(string<=? `*`string`*`1 `*`string`*`2)`
library procedure: `(string>=? `*`string`*`1 `*`string`*`2)`
library procedure: `(string-ci<? `*`string`*`1 `*`string`*`2)`
library procedure: `(string-ci>? `*`string`*`1 `*`string`*`2)`
library procedure: `(string-ci<=? `*`string`*`1 `*`string`*`2)`
library procedure: `(string-ci>=? `*`string`*`1 `*`string`*`2)`

These procedures are the lexicographic extensions to strings of the corresponding orderings on characters. For example, `string<?` is the lexicographic ordering on strings induced by the ordering `char<?` on characters. 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 lexicographically less than the longer string.

Implementations may generalize these and the `string=?` and `string-ci=?` procedures to take more than two arguments, as with the corresponding numerical predicates.

library procedure: `(substring `*`string start end`*`)`

String must be a string, and *start* and *end* must be exact integers satisfying

$0 \le start \le end \le$ `(string-length `*`string`*`)`.

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

library procedure: `(string-append `*`string ...`*`)`

Returns a newly allocated string whose characters form the concatenation of the given strings.

library procedure: `(string->list `*`string`*`)`
library procedure: `(list->string `*`list`*`)`

`String->list` returns a newly allocated list of the characters that make up the given string. `List->string` returns a newly allocated string formed from the characters in the list *list*, which must be a list of characters. `String->list` and `list->string` are inverses so far as `equal?` is concerned.

library procedure: `(string-copy `*`string`*`)`

Returns a newly allocated copy of the given *string*.

library procedure: `(string-fill! *string char*)`

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

6.3.6 Vectors

Vectors are heterogenous structures whose elements are indexed by 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 a 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 `#(`*obj* `...)`. 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 following:

```
#(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")
```

procedure: `(vector? *obj*)`

Returns `#t` if *obj* is a vector, otherwise returns `#f`.

procedure: `(make-vector *k*)`
procedure: `(make-vector *k fill*)`

Returns a newly allocated vector of *k* elements. If a second argument is given, then each element is initialized to *fill*. Otherwise the initial contents of each element is unspecified.

library procedure: `(vector *obj ...*)`

Returns a newly allocated vector whose elements contain the given arguments. Analogous to `list`.

```
(vector 'a 'b 'c)                       ===>  #(a b c)
```

procedure: `(vector-length *vector*)`

Returns the number of elements in *vector* as an exact integer.

procedure: `(vector-ref *vector k*)`

k must be a valid index of *vector*. `Vector-ref` returns the contents of element *k* of *vector*.

```
(vector-ref '#(1 1 2 3 5 8 13 21)
            5)
                ===>  8
(vector-ref '#(1 1 2 3 5 8 13 21)
            (let ((i (round (* 2 (acos -1)))))
              (if (inexact? i)
                  (inexact->exact i)
                  i)))
                ===> 13
```

procedure: `(vector-set! vector k obj)`

k must be a valid index of *vector*. `Vector-set!` stores *obj* in element *k* of *vector*. The value returned by `vector-set!` is unspecified.

```
(let ((vec (vector 0 '(2 2 2 2) "Anna")))
  (vector-set! vec 1 '("Sue" "Sue"))
  vec)
                ===>  #(0 ("Sue" "Sue") "Anna")

(vector-set! '#(0 1 2) 1 "doe")
                ===>  error  ; constant vector
```

library procedure: `(vector->list vector)`
library procedure: `(list->vector list)`

`Vector->list` returns a newly allocated list of the objects contained in the elements of *vector*. `List->vector` returns a newly created vector initialized to the elements of the list *list*.

```
(vector->list '#(dah dah didah))
                ===>  (dah dah didah)
(list->vector '(dididit dah))
                ===>  #(dididit dah)
```

library procedure: `(vector-fill! vector fill)`

Stores *fill* in every element of *vector*. The value returned by `vector-fill!` is unspecified.

6.4 Control features

This chapter describes various primitive procedures which control the flow of program execution in special ways. The `procedure?` predicate is also described here.

procedure: `(procedure? obj)`

Returns `#t` if *obj* is a procedure, otherwise returns `#f`.

```
(procedure? car)                    ===>  #t
(procedure? 'car)                   ===>  #f
(procedure? (lambda (x) (* x x)))
                                    ===>  #t
(procedure? '(lambda (x) (* x x)))
                                    ===>  #f
(call-with-current-continuation procedure?)
                                    ===>  #t
```

procedure: `(apply` *proc* arg_1 `...` *args*`)`

Proc must be a procedure and *args* must be a list. Calls *proc* with the elements of the list `(append (list` arg_1 `...) args)` as the actual arguments.

```
(apply + (list 3 4))                        ===>  7

(define compose
  (lambda (f g)
    (lambda args
      (f (apply g args)))))

((compose sqrt *) 12 75)                        ===>  30
```

library procedure: `(map` *proc* $list_1$ $list_2$ `...)`

The *list*s must be lists, and *proc* must be a procedure taking as many arguments as there are *list*s and returning a single value. If more than one *list* is given, then they must all be the same length. `Map` applies *proc* element-wise to the elements of the *list*s and returns a list of the results, in order. The dynamic order in which *proc* is applied to the elements of the *list*s is unspecified.

```
(map cadr '((a b) (d e) (g h)))
               ===>  (b e h)

(map (lambda (n) (expt n n))
     '(1 2 3 4 5))
               ===>  (1 4 27 256 3125)

(map + '(1 2 3) '(4 5 6))                   ===>  (5 7 9)

(let ((count 0))
  (map (lambda (ignored)
         (set! count (+ count 1))
         count)
       '(a b)))                           ===>  (1 2) or (2 1)
```

library procedure: `(for-each` *proc* $list_1$ $list_2$ `...)`

The arguments to `for-each` are like the arguments to `map`, but `for-each` calls *proc* for its side effects rather than for its values. Unlike `map`, `for-each` is guaranteed to call *proc* on the elements of the *list*s in order from the first element(s) to the last, and the value returned by `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)
```

library procedure: `(force` *promise*`)`

Forces the value of *promise* (see `delay`, section 4.2.5). 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.

```
(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
```

`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 only one value is computed for a promise, no matter how many times it is forced.

```
(define count 0)
(define p
  (delay (begin (set! count (+ count 1))
                (if (> count x)
                    count
                    (force p)))))
(define x 5)
p                               ===>  a promise
(force p)                       ===>  6
p                               ===>  a promise, still
(begin (set! x 10)
       (force p))               ===>  6
```

Here is a possible implementation of `delay` and `force`. Promises are implemented here as procedures of no arguments, and `force` simply calls its argument:

```
(define force
  (lambda (object)
    (object)))
```

We define the expression

```
(delay <expression>)
```

to have the same meaning as the procedure call

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

as follows

```
(define-syntax delay
  (syntax-rules ()
    ((delay expression)
     (make-promise (lambda () expression)))))
```

where `make-promise` is defined as follows:

```
(define make-promise
  (lambda (proc)
    (let ((result-ready? #f)
          (result #f))
      (lambda ()
        (if result-ready?
            result
            (let ((x (proc)))
              (if result-ready?
                  result
                  (begin (set! result-ready? #t)
                         (set! result x)
                         result)))))))))
```

Rationale: A promise may refer to its own value, as in the last example above. Forcing such a promise may cause the promise to be forced a second time before the value of the first force has been computed. This complicates the definition of `make-promise`.

Various extensions to this semantics of `delay` and `force` are supported in some implementations:

- 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 to `#f`, depending on the implementation:

```
(eqv? (delay 1) 1)                  ===>  unspecified
(pair? (delay (cons 1 2)))          ===>  unspecified
```

- Some implementations may implement ``implicit forcing,'' where the value of a promise is forced by primitive procedures like `cdr and +:`

```
(+ (delay (* 3 7)) 13)          ===>  34
```

procedure: `(call-with-current-continuation` *proc*`)`

Proc must be a procedure of one argument. The procedure `call-with-current-continuation` packages up the current continuation (see the rationale below) as an ``escape procedure'' and passes it as an argument to *proc*. The escape procedure is a Scheme procedure that, if it is later called, will abandon whatever continuation is in effect at that later time and will instead use the continuation that was in effect when the escape procedure was created. Calling the escape procedure may cause the invocation of *before* and *after* thunks installed using `dynamic-wind`.

The escape procedure accepts the same number of arguments as the continuation to the original call to `call-with-current-continuation`. Except for continuations created by the `call-with-values` procedure, all continuations take exactly one value. The effect of passing no value or more than one value to continuations that were not created by `call-with-values` is unspecified.

The escape procedure that is passed to *proc* 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 ways in which `call-with-current-continuation` is used. If all real uses 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
```

Rationale:

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 extremely useful for implementing a wide variety of advanced control structures.

Whenever a Scheme expression is evaluated there is a *continuation* wanting 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, then the continuation might 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 do not think much about them. On rare occasions, however, a programmer may need to deal with continuations explicitly. `Call-with-current-continuation` allows Scheme programmers to do that by creating a procedure that acts just like the current continuation.

Most programming languages incorporate one or more special-purpose escape constructs with names like `exit`, `return`, or even `goto`. In 1965, however, Peter Landin [16] invented a general purpose escape operator called the J-operator. John Reynolds [24] described a simpler but equally powerful construct in 1972. The `catch` special form described by Sussman and Steele

in the 1975 report on Scheme is exactly the same as Reynolds's construct, though its name came from a less general construct in MacLisp. Several Scheme implementors noticed that the full power of the `catch` construct could be provided by a procedure instead of by a special syntactic construct, and the name `call-with-current-continuation` was coined in 1982. This name is descriptive, but opinions differ on the merits of such a long name, and some people use the name `call/cc` instead.

procedure: `(values` *`obj ...`*`)`

Delivers all of its arguments to its continuation. Except for continuations created by the `call-with-values` procedure, all continuations take exactly one value. `Values` might be defined as follows:

```
(define (values . things)
  (call-with-current-continuation
    (lambda (cont) (apply cont things))))
```

procedure: `(call-with-values` *`producer consumer`*`)`

Calls its *producer* argument with no values and a continuation that, when passed some values, calls the *consumer* procedure with those values as arguments. The continuation for the call to *consumer* is the continuation of the call to `call-with-values`.

```
(call-with-values (lambda () (values 4 5))
                  (lambda (a b) b))
                                                    ===>  5

(call-with-values * -)                              ===>  -1
```

procedure: `(dynamic-wind` *`before thunk after`*`)`

Calls *thunk* without arguments, returning the result(s) of this call. *Before* and *after* are called, also without arguments, as required by the following rules (note that in the absence of calls to continuations captured using `call-with-current-continuation` the three arguments are called once each, in order). *Before* is called whenever execution enters the dynamic extent of the call to *thunk* and *after* is called whenever it exits that dynamic extent. The dynamic extent of a procedure call is the period between when the call is initiated and when it returns. In Scheme, because of `call-with-current-continuation`, the dynamic extent of a call may not be a single, connected time period. It is defined as follows:

- The dynamic extent is entered when execution of the body of the called procedure begins.
- The dynamic extent is also entered when execution is not within the dynamic extent and a continuation is invoked that was captured (using `call-with-current-continuation`) during the dynamic extent.
- It is exited when the called procedure returns.
- It is also exited when execution is within the dynamic extent and a continuation is invoked that was captured while not within the dynamic extent.

If a second call to `dynamic-wind` occurs within the dynamic extent of the call to *thunk* and then a continuation is invoked in such a way that the *after*s from these two invocations of `dynamic-wind` are both to be called, then the *after* associated with the second (inner) call to `dynamic-wind` is called first.

If a second call to `dynamic-wind` occurs within the dynamic extent of the call to *thunk* and then a continuation is invoked in such a way that the *before*s from these two invocations of `dynamic-wind` are both to be called, then the *before* associated with the first (outer) call to `dynamic-wind` is called first.

If invoking a continuation requires calling the *before* from one call to `dynamic-wind` and the *after* from another, then the *after* is called first.

The effect of using a captured continuation to enter or exit the dynamic extent of a call to *before* or *after* is undefined.

```
(let ((path '())
      (c #f))
  (let ((add (lambda (s)
               (set! path (cons s path)))))
    (dynamic-wind
      (lambda () (add 'connect))
      (lambda ()
        (add (call-with-current-continuation
               (lambda (c0)
                 (set! c c0)
                 'talk1))))
      (lambda () (add 'disconnect)))
    (if (< (length path) 4)
        (c 'talk2)
        (reverse path))))

              ===> (connect talk1 disconnect
              connect talk2 disconnect)
```

6.5 Eval

procedure: `(eval` *expression environment-specifier*`)`

Evaluates *expression* in the specified environment and returns its value. *Expression* must be a valid Scheme expression represented as data, and *environment-specifier* must be a value returned by one of the three procedures described below. Implementations may extend `eval` to allow non-expression programs (definitions) as the first argument and to allow other values as environments, with the restriction that `eval` is not allowed to create new bindings in the environments associated with `null-environment` or `scheme-report-environment`.

```
(eval '(* 7 3) (scheme-report-environment 5))
                                                ===>  21

(let ((f (eval '(lambda (f x) (f x x))
               (null-environment 5))))
  (f + 10))
                                                ===>  20
```

procedure: `(scheme-report-environment` *`version`*`)`
procedure: `(null-environment` *`version`*`)`

Version must be the exact integer `5`, corresponding to this revision of the Scheme report (the Revised^5 Report on Scheme). `Scheme-report-environment` returns a specifier for an environment that is empty except for all bindings defined in this report that are either required or both optional and supported by the implementation. `Null-environment` returns a specifier for an environment that is empty except for the (syntactic) bindings for all syntactic keywords defined in this report that are either required or both optional and supported by the implementation.

Other values of *version* can be used to specify environments matching past revisions of this report, but their support is not required. An implementation will signal an error if *version* is neither `5` nor another value supported by the implementation.

The effect of assigning (through the use of `eval`) a variable bound in a `scheme-report-environment` (for example `car`) is unspecified. Thus the environments specified by `scheme-report-environment` may be immutable.

optional procedure: `(interaction-environment)`

This procedure returns a specifier for the environment that contains implementation-defined bindings, typically a superset of those listed in the report. The intent is that this procedure will return the environment in which the implementation would evaluate expressions dynamically typed by the user.

6.6 Input and output

6.6.1 Ports

Ports represent input and output devices. To Scheme, an input port is a Scheme object that can deliver characters upon command, while an output port is a Scheme object that can accept characters.

library procedure: `(call-with-input-file` *`string proc`*`)`
library procedure: `(call-with-output-file` *`string proc`*`)`

String should be a string naming a file, and *proc* should be a procedure that accepts one argument. For `call-with-input-file`, the file should already exist; for `call-with-output-file`, the effect is unspecified if the file already exists. These procedures call *proc* 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 *proc* returns, then the port is closed automatically and the value(s) yielded by the *proc* is(are) returned. If *proc* does not return, then the port will not be closed automatically unless it is possible to prove that the port will never again be used for a read or write operation.

> *Rationale:* Because Scheme's escape procedures have unlimited extent, it is possible to escape from the current continuation but later to 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`.

procedure: `(input-port? *obj*)`
procedure: `(output-port? *obj*)`

Returns `#t` if *obj* is an input port or output port respectively, otherwise returns `#f`.

procedure: `(current-input-port)`
procedure: `(current-output-port)`

Returns the current default input or output port.

optional procedure: `(with-input-from-file *string thunk*)`
optional procedure: `(with-output-to-file *string thunk*)`

String should be a string naming a file, and *proc* should be a procedure of no arguments. For `with-input-from-file`, the file should already exist; for `with-output-to-file`, the effect is unspecified if the file already exists. 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 is used by `(read)`, `(write *obj*)`, and so forth), 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(s) the value(s) yielded by *thunk*. If an escape procedure is used to escape from the continuation of these procedures, their behavior is implementation dependent.

procedure: `(open-input-file *filename*)`

Takes a string naming 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.

procedure: `(open-output-file *filename*)`

Takes a string naming 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, the effect is unspecified.

procedure: `(close-input-port *port*)`
procedure: `(close-output-port *port*)`

Closes the file associated with *port*, rendering the *port* incapable of delivering or accepting characters. These routines have no effect if the file has already been closed. The value returned is unspecified.

6.6.2 Input

library procedure: `(read)`
library procedure: `(read *port*)`

`Read` converts external representations of Scheme objects into the objects themselves. That is, it is a parser for the nonterminal <datum> (see sections 7.1.2 and 6.3.2). `Read` returns the next object parsable from the given input *port*, updating *port* to point to the first

character past the end of the external representation of the object.

If an end of file is encountered in the input before any characters are found that can begin an object, then an end of file object is returned. The port remains open, and further attempts to read will also return an end of file object. If an end of file is encountered after the beginning of an object's external representation, but the external representation is incomplete and therefore not parsable, an error is signalled.

The *port* argument may be omitted, in which case it defaults to the value returned by `current-input-port`. It is an error to read from a closed port.

procedure: `(read-char)`
procedure: `(read-char` *`port`*`)`

Returns the next character available from the input *port*, updating the *port* to point to the following character. If no more characters are available, an end of file object is returned. *Port* may be omitted, in which case it defaults to the value returned by `current-input-port`.

procedure: `(peek-char)`
procedure: `(peek-char` *`port`*`)`

Returns the next character available from the input *port*, *without* updating the *port* to point to the following character. If no more characters are available, an end of file object is returned. *Port* may be omitted, in which case it defaults to the value returned by `current-input-port`.

> *Note:* The value returned by a call to `peek-char` is the same as the value that would have been returned by a call to `read-char` with the same *port*. The only difference is that the very next call to `read-char` or `peek-char` on that *port* will return the value returned by the preceding call to `peek-char`. In particular, a call to `peek-char` on an interactive port will hang waiting for input whenever a call to `read-char` would have hung.

procedure: `(eof-object?` *`obj`*`)`

Returns `#t` if *obj* is an end of file object, otherwise returns `#f`. The precise set of end of file objects will vary among implementations, but in any case no end of file object will ever be an object that can be read in using `read`.

procedure: `(char-ready?)`
procedure: `(char-ready?` *`port`*`)`

Returns `#t` if a character is ready on the input *port* and returns `#f` otherwise. If `char-ready` returns `#t` then the next `read-char` operation on the given *port* is guaranteed not to hang. If the *port* is at end of file then `char-ready?` returns `#t`. *Port* may be omitted, in which case it defaults to the value returned by `current-input-port`.

> *Rationale:* `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 ensure 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.

6.6.3 Output

library procedure: `(write` *`obj`*`)`
library procedure: `(write` *`obj port`*`)`

Writes a written representation of *obj* to the given *port*. Strings that appear in the written representation are enclosed in doublequotes, and within those strings backslash and doublequote characters are escaped by backslashes. Character objects are written using the `#\` notation. `Write` returns an unspecified value. The *port* argument may be omitted, in which case it defaults to the value returned by `current-output-port`.

library procedure: `(display` *`obj`*`)`
library procedure: `(display` *`obj port`*`)`

Writes a representation of *obj* to the given *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. The *port* argument may be omitted, in which case it defaults to the value returned by `current-output-port`.

Rationale: `Write` is intended for producing machine-readable output and `display` is for producing human-readable output. Implementations that allow ``slashification'' within symbols will probably want `write` but not `display` to slashify funny characters in symbols.

library procedure: `(newline)`
library procedure: `(newline` *`port`*`)`

Writes an end of line to *port*. Exactly how this is done differs from one operating system to another. Returns an unspecified value. The *port* argument may be omitted, in which case it defaults to the value returned by `current-output-port`.

procedure: `(write-char` *`char`*`)`
procedure: `(write-char` *`char port`*`)`

Writes the character *char* (not an external representation of the character) to the given *port* and returns an unspecified value. The *port* argument may be omitted, in which case it defaults to the value returned by `current-output-port`.

6.6.4 System interface

Questions of system interface generally fall outside of the domain of this report. However, the following operations are important enough to deserve description here.

optional procedure: `(load` *`filename`*`)`

Filename should be a string naming an existing file containing Scheme source code. The `load` procedure reads expressions and definitions from the file and evaluates them

sequentially. It is unspecified whether the results of the expressions are printed. The `load` procedure does not affect the values returned by `current-input-port` and `current-output-port`. `Load` returns an unspecified value.

> *Rationale:* For portability, `load` must operate on source files. Its operation on other kinds of files necessarily varies among implementations.

optional procedure: `(transcript-on` *`filename`*`)`
optional procedure: `(transcript-off)`

Filename must be a string naming an output file to be created. The effect of `transcript-on` is to open the named file for output, and to cause a transcript of subsequent interaction between the user and the Scheme system to be written to the file. The transcript is ended by a call to `transcript-off`, which closes the transcript file. Only one transcript may be in progress at any time, though some implementations may relax this restriction. The values returned by these procedures are unspecified.

[Go to first, previous, next page; contents; index]