# Revised$^5$ Report on the Algorithmic Language Scheme

RICHARD KELSEY, WILLIAM CLINGER, AND JONATHAN REES (*Editors*)

| | | | |
|---|---|---|---|
| H. ABELSON | R. K. DYBVIG | C. T. HAYNES | G. J. ROZAS |
| N. I. ADAMS IV | D. P. FRIEDMAN | E. KOHLBECKER | G. L. STEELE JR. |
| D. H. BARTLEY | R. HALSTEAD | D. OXLEY | G. J. SUSSMAN |
| G. BROOKS | C. HANSON | K. M. PITMAN | M. WAND |


*Dedicated to the Memory of Robert Hieb*

**20 February 1998**

## SUMMARY

The report gives a defining description of the programming language Scheme. Scheme is a statically scoped and properly tail-recursive dialect of the Lisp programming language invented by Guy Lewis Steele Jr. and Gerald Jay Sussman. It was designed to have an exceptionally clear and simple semantics and few different ways to form expressions. A wide variety of programming paradigms, including imperative, functional, and message passing styles, find convenient expression in Scheme.

The introduction offers a brief history of the language and of the report.

The first three chapters present the fundamental ideas of the language and describe the notational conventions used for describing the language and for writing programs in the language.

Chapters 4 and 5 describe the syntax and semantics of expressions, programs, and definitions.

Chapter 6 describes Scheme's built-in procedures, which include all of the language's data manipulation and input/output primitives.

Chapter 7 provides a formal syntax for Scheme written in extended BNF, along with a formal denotational semantics. An example of the use of the language follows the formal syntax and semantics.

The report concludes with a list of references and an alphabetic index.

## CONTENTS

# INTRODUCTION

Programming languages should be designed not by piling feature on top of feature, but by removing the weaknesses and restrictions that make additional features appear necessary. Scheme demonstrates that a very small number of rules for forming expressions, with no restrictions on how they are composed, suffice to form a practical and efficient programming language that is flexible enough to support most of the major programming paradigms in use today.

Scheme was one of the first programming languages to incorporate first class procedures as in the lambda calculus, thereby proving the usefulness of static scope rules and block structure in a dynamically typed language. Scheme was the first major dialect of Lisp to distinguish procedures from lambda expressions and symbols, to use a single lexical environment for all variables, and to evaluate the operator position of a procedure call in the same way as an operand position. By relying entirely on procedure calls to express iteration, Scheme emphasized the fact that tail-recursive procedure calls are essentially goto's that pass arguments. Scheme was the first widely used programming language to embrace first class escape procedures, from which all previously known sequential control structures can be synthesized. A subsequent version of Scheme introduced the concept of exact and inexact numbers, an extension of Common Lisp's generic arithmetic. More recently, Scheme became the first programming language to support hygienic macros, which permit the syntax of a block-structured language to be extended in a consistent and reliable manner.

### Background

The first description of Scheme was written in 1975 [28]. A revised report [25] appeared in 1978, which described the evolution of the language as its MIT implementation was upgraded to support an innovative compiler [26]. Three distinct projects began in 1981 and 1982 to use variants of Scheme for courses at MIT, Yale, and Indiana University [21, 17, 10]. An introductory computer science textbook using Scheme was published in 1984 [1].

As Scheme became more widespread, local dialects began to diverge until students and researchers occasionally found it difficult to understand code written at other sites. Fifteen representatives of the major implementations of Scheme therefore met in October 1984 to work toward a better and more widely accepted standard for Scheme. Their report [4] was published at MIT and Indiana University in the summer of 1985. Further revision took place in the spring of 1986 [23], and in the spring of 1988 [6]. The present report reflects further revisions agreed upon in a meeting at Xerox PARC in June 1992.



### Acknowledgements

We would like to thank the following people for their help: Alan Bawden, Michael Blair, George Carrette, Andy Cromarty, Pavel Curtis, Jeff Dalton, Olivier Danvy, Ken Dickey, Bruce Duba, Marc Feeley, Andy Freeman, Richard Gabriel, Yekta Gürsel, Ken Haase, Robert Hieb, Paul Hudak, Morry Katz, Chris Lindblad, Mark Meyer, Jim Miller, Jim Philbin, John Ramsdell, Mike Shaff, Jonathan Shapiro, Julie Sussman, Perry Wagle, Daniel Weise, Henry Wu, and Ozan Yigit. We thank Carol Fessenden, Daniel Friedman, and Christopher Haynes for permission to use text from the Scheme 311 version 4 reference manual. We thank Texas Instruments, Inc. for permission to use text from the *TI Scheme Language Reference Manual*[30]. We gladly acknowledge the influence of manuals for MIT Scheme[17], T[22], Scheme 84[11],Common Lisp[27], and Algol 60[18].

We also thank Betty Dexter for the extreme effort she put into setting this report in TEX, and Donald Knuth for designing the program that caused her troubles.

The Artificial Intelligence Laboratory of the Massachusetts Institute of Technology, the Computer Science Department of Indiana University, the Computer and Information Sciences Department of the University of Oregon, and the NEC Research Institute supported the preparation of this report. Support for the MIT work was provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-80-C-0505. Support for the Indiana University work was provided by NSF grants NCS 83-04567 and NCS 83-03325.

# DESCRIPTION OF THE LANGUAGE

## 1. Overview of Scheme

### 1.1. Semantics

This section gives an overview of Scheme's semantics. A detailed informal semantics is the subject of chapters 3 through 6. For reference purposes, section 7.2 provides a formal semantics of Scheme.

Following Algol, Scheme is a statically scoped programming language. Each use of a variable is associated with a lexically apparent binding of that variable.

Scheme has latent as opposed to manifest types. Types are associated with values (also called objects) rather than with variables. (Some authors refer to languages with latent types as weakly typed or dynamically typed languages.) Other languages with latent types are 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.

All objects created in the course of a Scheme computation, including procedures and continuations, have unlimited extent. No Scheme object is ever destroyed. The reason that implementations of Scheme do not (usually!) run out of storage is that they are permitted to reclaim the storage occupied by an object if they can prove that the object cannot possibly matter to any future computation. Other languages in which most objects have unlimited extent include APL and other Lisp dialects.

Implementations of Scheme are required to be properly tail-recursive. This allows the execution of an iterative computation in constant space, even if the iterative computation is described by a syntactically recursive procedure. Thus with a properly tail-recursive implementation, iteration can be expressed using the ordinary procedure-call mechanics, so that special iteration constructs are useful only as syntactic sugar. See section 3.5.

Scheme procedures are objects in their own right. Procedures can be created dynamically, stored in data structures, returned as results of procedures, and so on. Other languages with these properties include Common Lisp and ML.

One distinguishing feature of Scheme is that continuations, which in most other languages only operate behind the scenes, also have "first-class" status. Continuations are useful for implementing a wide variety of advanced control constructs, including non-local exits, backtracking, and coroutines. See section 6.4.

Arguments to Scheme procedures are always passed by value, which means that the actual argument expressions are evaluated before the procedure gains control, whether the procedure needs the result of the evaluation or not. ML, C, and APL are three other languages that always pass arguments by value. This is distinct from the lazy-evaluation semantics of Haskell, or the call-by-name semantics of Algol 60, where an argument expression is not evaluated unless its value is needed by the procedure.

Scheme's model of arithmetic is designed to remain as independent as possible of the particular ways in which numbers are represented within a computer. In Scheme, every integer is a rational number, every rational is a real, and every real is a complex number. Thus the distinction between integer and real arithmetic, so important to many programming languages, does not appear in Scheme. In its place is a distinction between exact arithmetic, which corresponds to the mathematical ideal, and inexact arithmetic on approximations. As in Common Lisp, exact arithmetic is not limited to integers.

### 1.2. Syntax

Scheme, like most dialects of Lisp, employs a fully parenthesized prefix notation for programs and (other) data; the grammar of Scheme generates a sublanguage of the language used for data. An important consequence of this simple, uniform representation is the susceptibility of Scheme programs and data to uniform treatment by other Scheme programs. For example, the `eval` procedure evaluates a Scheme program expressed as data.

The `read` procedure performs syntactic as well as lexical decomposition of the data it reads. The `read` procedure parses its input as data (section 7.1.2), not as program.

The formal syntax of Scheme is described in section 7.1.

### 1.3. Notation and terminology

#### 1.3.1. Primitive, library, and optional features

It is required that every implementation of Scheme support all features that are not marked as being *optional*. Implementations are free to omit optional features of Scheme or to add extensions, provided the extensions are not in conflict with the language reported here. In particular, implementations must support portable code by providing a syntactic mode that preempts no lexical conventions of this report.

To aid in understanding and implementing Scheme, some features are marked as *library*. These can be easily implemented in terms of the other, primitive, features. They are redundant in the strict sense of the word, but they capture common patterns of usage, and are therefore provided as convenient abbreviations.

#### 1.3.2. Error situations and unspecified behavior

When speaking of an error situation, this report uses the phrase "an error is signalled" to indicate that implementations must detect and report the error. If such wording does not appear in the discussion of an error, then implementations are not required to detect or report the error, though they are encouraged to do so. An error situation that implementations are not required to detect is usually referred to simply as "an error."

For example, it is an error for a procedure to be passed an argument that the procedure is not explicitly specified to handle, even though such domain errors are seldom mentioned in this report. Implementations may extend a procedure's domain of definition to include such arguments.

This report uses the phrase "may report a violation of an implementation restriction" to indicate circumstances under which an implementation is permitted to report that it is unable to continue execution of a correct program because of some restriction imposed by the implementation. Implementation restrictions are of course discouraged, but implementations are encouraged to report violations of implementation restrictions.

For example, an implementation may report a violation of an implementation restriction if it does not have enough storage to run a program.

If the value of an expression is said to be "unspecified," then the expression must evaluate to some object without signalling an error, but the value depends on the implementation; this report explicitly does not say what value should be returned.

#### 1.3.3. Entry format

Chapters 4 and 6 are organized into entries. Each entry describes one language feature or a group of related features, where a feature is either a syntactic construct or a built-in procedure. An entry begins with one or more header lines of the form

*template* *category*

for required, primitive features, or

*template* *qualifier category*

where *qualifier* is either "library" or "optional" as defined in section 1.3.1.

If *category* is "syntax", the entry describes an expression type, and the template gives the syntax of the expression type. Components of expressions are designated by syntactic variables, which are written using angle brackets, for example, ⟨expression⟩, ⟨variable⟩. Syntactic variables should be understood to denote segments of program text; for example, ⟨expression⟩ stands for any string of characters which is a syntactically valid expression. The notation

$\langle\text{thing}_1\rangle \ldots$

indicates zero or more occurrences of a ⟨thing⟩, and

$\langle\text{thing}_1\rangle\ \langle\text{thing}_2\rangle \ldots$

indicates one or more occurrences of a ⟨thing⟩.

If *category* is "procedure", then the entry describes a procedure, and the header line gives a template for a call to the procedure. Argument names in the template are *italicized*. Thus the header line

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

indicates that the built-in procedure `vector-ref` takes two arguments, a vector *vector* and an exact non-negative integer *k* (see below). The header lines

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

indicate that the `make-vector` procedure must be defined to take either one or two arguments.

It is an error for an operation to be presented with an argument that it is not specified to handle. For succinctness, we follow the convention that if an argument name is also the name of a type listed in section 3.2, then that argument must be of the named type. For example, the header line for `vector-ref` given above dictates that the first argument to `vector-ref` must be a vector. The following naming conventions also imply type restrictions:

| | |
|---|---|
| *obj* | any object |
| *list*, $list_1$, … $list_j$, … | list (see section 6.3.2) |
| $z, z_1, \ldots z_j, \ldots$ | complex number |
| $x, x_1, \ldots x_j, \ldots$ | real number |
| $y, y_1, \ldots y_j, \ldots$ | real number |
| $q, q_1, \ldots q_j, \ldots$ | rational number |
| $n, n_1, \ldots n_j, \ldots$ | integer |
| $k, k_1, \ldots k_j, \ldots$ | exact non-negative integer |

#### 1.3.4. Evaluation examples

The symbol "⟹" used in program examples should be read "evaluates to." For example,

```
(* 5 8)                 ⟹  40
```

means that the expression `(* 5 8)` evaluates to the object `40`. Or, more precisely: the expression given by the sequence of characters "`(* 5 8)`" evaluates, in the initial environment, to an object that may be represented externally by the sequence of characters "`40`". See section 3.3 for a discussion of external representations of objects.

#### 1.3.5. Naming conventions

By convention, the names of procedures that always return a boolean value usually end in "?". Such procedures are called predicates.

By convention, the names of procedures that store values into previously allocated locations (see section 3.4) usually end in "`!`". Such procedures are called mutation procedures. By convention, the value returned by a mutation procedure is unspecified.

By convention, "`->`" appears within the names of procedures that take an object of one type and return an analogous object of another type. For example, `list->vector` takes a list and returns a vector whose elements are the same as those of the list.

## 2. Lexical conventions

This section gives an informal account of some of the lexical conventions used in writing Scheme programs. For a formal syntax of Scheme, see section 7.1.

Upper and lower case forms of a letter are never distinguished except within character and string constants. For example, `Foo` is the same identifier as `FOO`, and `#x1AB` is the same number as `#X1ab`.

### 2.1. Identifiers

Most identifiers allowed by other programming languages are also acceptable to Scheme. The precise rules for forming identifiers vary among implementations of Scheme, but in all implementations a sequence of letters, digits, and "extended alphabetic characters" that begins with a character that cannot begin a number is an identifier. In addition, `+`, `-`, and `...` are identifiers. Here are some examples of identifiers:

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

Extended alphabetic characters may be used within identifiers as if they were letters. The following are extended alphabetic characters:

```
! $ % & * + - . / : < = > ? @ ^ _ ~
```

See section 7.1.1 for a formal syntax of identifiers.

Identifiers have two uses within Scheme programs:

- Any identifier may be used as a variable or as a syntactic keyword (see sections 3.1 and 4.3).
- When an identifier appears as a literal or within a literal (see section 4.1.2), it is being used to denote a *symbol* (see section 6.3.3).

### 2.2. Whitespace and comments

*Whitespace* characters are spaces and newlines. (Implementations typically provide additional whitespace characters such as tab or page break.) Whitespace is used for improved readability and as necessary to separate tokens from each other, a token being an indivisible lexical unit such as an identifier or number, but 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.

A semicolon (`;`) indicates the start of a comment. The comment continues to the end of the line on which the semicolon appears. Comments are invisible to Scheme, but the end of the line is visible as whitespace. This prevents a comment from appearing in the middle of an identifier or number.

```
;;; The FACT procedure computes the factorial
;;; of a non-negative integer.
(define fact
  (lambda (n)
    (if (= n 0)
        1        ;Base case: return 1
        (* n (fact (- n 1))))))
```

### 2.3. Other notations

For a description of the notations used for numbers, see section 6.2.

`. + -` These are used in numbers, and may also occur anywhere in an identifier except as the first character. A delimited plus or minus sign by itself is also an identifier. A delimited period (not occurring within a number or identifier) is used in the notation for pairs (section 6.3.2), and to indicate a rest-parameter in a formal parameter list (section 4.1.4). A delimited sequence of three successive periods is also an identifier.

`( )` Parentheses are used for grouping and to notate lists (section 6.3.2).

`'` The single quote character is used to indicate literal data (section 4.1.2).

`` ` `` The backquote character is used to indicate almost-constant data (section 4.2.6).

`, ,@` The character comma and the sequence comma at-sign are used in conjunction with backquote (section 4.2.6).

`"` The double quote character is used to delimit strings (section 6.3.5).

\ Backslash is used in the syntax for character constants (section 6.3.4) and as an escape character within string constants (section 6.3.5).

[ ] { } | Left and right square brackets and curly braces and vertical bar are reserved for possible future extensions to the language.

`#` Sharp sign is used for a variety of purposes depending on the character that immediately follows it:

`#t` `#f` These are the boolean constants (section 6.3.1).

`#\` This introduces a character constant (section 6.3.4).

`#(` This introduces a vector constant (section 6.3.6). Vector constants are terminated by `)` .

`#e #i #b #o #d #x` These are used in the notation for numbers (section 6.2.4).

# 3. Basic concepts

## 3.1. Variables, syntactic keywords, and regions

An identifier may name a type of syntax, or it may name a location where a value can be stored. An identifier that names a type of syntax is called a *syntactic keyword* and is said to be *bound* to that syntax. An identifier that names a location is called a *variable* and is said to be *bound* to that location. The set of all visible bindings in effect at some point in a program is known as the *environment* in effect at that point. The value stored in the location to which a variable is bound is called the variable's value. By abuse of terminology, the variable is sometimes said to name the value or to be bound to the value. This is not quite accurate, but confusion rarely results from this practice.

Certain expression types are used to create new kinds of syntax and bind syntactic keywords to those new syntaxes, while other expression types create new locations and bind variables to those locations. These expression types are called *binding constructs*. Those that bind syntactic keywords are listed in section 4.3. The most fundamental of the variable binding constructs is the `lambda` expression, because all other variable binding constructs can be explained in terms of `lambda` expressions. The other variable binding constructs are `let`, `let*`, `letrec`, and `do` expressions (see sections 4.1.4, 4.2.2, and 4.2.4).

Like Algol and Pascal, and unlike most other dialects of Lisp except for Common Lisp, Scheme is a statically scoped language with block structure. To each place where an identifier is bound in a program there corresponds a *region* of the program text within which the binding is visible. The region is determined by the particular binding construct that establishes the binding; if the binding is established by a `lambda` expression, for example, then its region is the entire `lambda` expression. Every mention of an identifier refers to the binding of the identifier that established the innermost of the regions containing the use. If there is no binding of the identifier whose region contains the use, then the use refers to the binding for the variable in the top level environment, if any (chapters 4 and 6); if there is no binding for the identifier, it is said to be *unbound*.

## 3.2. Disjointness of types

No object satisfies more than one of the following predicates:

```
boolean?        pair?
symbol?         number?
char?           string?
vector?         port?
procedure?
```

These predicates define the types *boolean*, *pair*, *symbol*, *number*, *char* (or *character*), *string*, *vector*, *port*, and *procedure*. The empty list is a special object of its own type; it satisfies none of the above predicates.

Although there is a separate boolean type, any Scheme value can be used as a boolean value for the purpose of a conditional test. As explained in section 6.3.1, all values count as true in such a test except for `#f`. This report uses the word "true" to refer to any Scheme value except `#f`, and the word "false" to refer to `#f`.

## 3.3. External representations

An important concept in Scheme (and Lisp) 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 . ()))`" (see section 6.3.2).

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

An external representation may be written in a program to obtain the corresponding object (see `quote`, section 4.1.2).

External representations can also be used for input and output. The procedure `read` (section 6.6.2) parses external representations, and the procedure `write` (section 6.6.3) 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 evaluating 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 vice versa).

The syntax of external representations of various kinds of objects accompanies the description of the primitives for manipulating the objects in the appropriate sections of chapter 6.

## 3.4. Storage model

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?` (section 6.1) 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 report 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. In such systems literal constants and the strings returned by `symbol->string` are immutable objects, while all objects created by the other procedures listed in this report are mutable. It is an error to attempt to store a new value into a location that is denoted by an immutable object.

## 3.5. Proper tail recursion

Implementations of Scheme are required to be *properly tail-recursive*. Procedure calls that occur in certain syntactic contexts defined below are 'tail calls'. A Scheme implementation is properly tail-recursive if it supports an unbounded number of active tail calls. A call is *active* if the called procedure may still return. Note that this includes calls that may be returned from either by the current continuation or by continuations captured earlier by `call-with-current-continuation` that are later invoked. In the absence of captured continuations, calls could return at most once and the active calls would be those that had not yet returned. A formal definition of proper tail recursion can be found in [8].

*Rationale:*

Intuitively, no space is needed for an active tail call because the continuation that is used in the tail call has the same semantics as the continuation passed to the procedure containing the call. Although an improper implementation might use a new continuation in the call, a return to this new continuation would be followed immediately by a return to the continuation passed to the procedure. A properly tail-recursive implementation returns to that continuation directly.

Proper tail recursion was one of the central ideas in Steele and Sussman's original version of Scheme. Their first Scheme interpreter implemented both functions and actors. Control flow was expressed using actors, which differed from functions in that they passed their results on to another actor instead of returning to a caller. In the terminology of this section, each actor finished with a tail call to another actor.

Steele and Sussman later observed that in their interpreter the code for dealing with actors was identical to that for functions and thus there was no need to include both in the language.

A *tail call* is a procedure call that occurs in a *tail context*. Tail contexts are defined inductively. Note that a tail context is always determined with respect to a particular lambda expression.

- The last expression within the body of a lambda expression, shown as ⟨tail expression⟩ below, occurs in a tail context.

  ```
  (lambda ⟨formals⟩
    ⟨definition⟩* ⟨expression⟩* ⟨tail expression⟩)
  ```

- If one of the following expressions is in a tail context, then the subexpressions shown as ⟨tail expression⟩ are in a tail context. These were derived from rules in the grammar given in chapter 7 by replacing some occurrences of ⟨expression⟩ with ⟨tail expression⟩. Only those rules that contain tail contexts are shown here.

```
(if ⟨expression⟩ ⟨tail expression⟩ ⟨tail expression⟩)
(if ⟨expression⟩ ⟨tail expression⟩)

(cond ⟨cond clause⟩+)
(cond ⟨cond clause⟩* (else ⟨tail sequence⟩))

(case ⟨expression⟩
  ⟨case clause⟩+)
(case ⟨expression⟩
  ⟨case clause⟩*
  (else ⟨tail sequence⟩))

(and ⟨expression⟩* ⟨tail expression⟩)
(or ⟨expression⟩* ⟨tail expression⟩)

(let (⟨binding spec⟩*) ⟨tail body⟩)
(let ⟨variable⟩ (⟨binding spec⟩*) ⟨tail body⟩)
(let* (⟨binding spec⟩*) ⟨tail body⟩)
(letrec (⟨binding spec⟩*) ⟨tail body⟩)

(let-syntax (⟨syntax spec⟩*) ⟨tail body⟩)
(letrec-syntax (⟨syntax spec⟩*) ⟨tail body⟩)

(begin ⟨tail sequence⟩)

(do (⟨iteration spec⟩*)
    (⟨test⟩ ⟨tail sequence⟩)
  ⟨expression⟩*)
```

where

```
⟨cond clause⟩ ⟶ (⟨test⟩ ⟨tail sequence⟩)
⟨case clause⟩ ⟶ ((⟨datum⟩*) ⟨tail sequence⟩)

⟨tail body⟩ ⟶ ⟨definition⟩* ⟨tail sequence⟩
⟨tail sequence⟩ ⟶ ⟨expression⟩* ⟨tail expression⟩
```

- If a `cond` expression is in a tail context, and has a clause of the form (⟨expression$_1$⟩ => ⟨expression$_2$⟩) then the (implied) call to the procedure that results from the evaluation of ⟨expression$_2$⟩ is in a tail context. ⟨expression$_2$⟩ itself is not in a tail context.

Certain built-in procedures are also required to perform tail calls. The first argument passed to `apply` and to `call-with-current-continuation`, and the second argument passed to `call-with-values`, must be called via a tail call. Similarly, `eval` must evaluate its argument as if it were in tail position within the `eval` procedure.

In the following example the only tail call is the call to `f`. None of the calls to `g` or `h` are tail calls. The reference to `x` is in a tail context, but it is not a call and thus is not a tail call.

```
(lambda ()
  (if (g)
      (let ((x (h)))
        x)
      (and (g) (f))))
```

*Note:* Implementations are allowed, but not required, to recognize that some non-tail calls, such as the call to `h` above, can be evaluated as though they were tail calls. In the example above, the `let` expression could be compiled as a tail call to `h`. (The possibility of `h` returning an unexpected number of values can be ignored, because in that case the effect of the `let` is explicitly unspecified and implementation-dependent.)

# 4. Expressions

Expression types are categorized as *primitive* or *derived*. Primitive expression types include variables and procedure calls. Derived expression types are not semantically primitive, but can instead be defined as macros. With the exception of `quasiquote`, whose macro definition is complex, the derived expressions are classified as library features. Suitable definitions are given in section 7.3.

## 4.1. Primitive expression types

### 4.1.1. Variable references

⟨variable⟩ syntax

An expression consisting of a variable (section 3.1) is a variable reference. The value of the variable reference is the value stored in the location to which the variable is bound. It is an error to reference an unbound variable.

```
(define x 28)
x                          ⟹  28
```

### 4.1.2. Literal expressions

(quote ⟨datum⟩) syntax
'⟨datum⟩ syntax
⟨constant⟩ syntax

`(quote ⟨datum⟩)` evaluates to ⟨datum⟩. ⟨Datum⟩ may be any external representation of a Scheme object (see section 3.3). This notation is used to include literal constants in Scheme code.

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

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

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

Numerical constants, string constants, character constants, and boolean constants evaluate "to themselves"; they need not be quoted.

```
'"abc"                   ⟹  "abc"
"abc"                    ⟹  "abc"
'145932                  ⟹  145932
145932                   ⟹  145932
'#t                      ⟹  #t
#t                       ⟹  #t
```

As noted in section 3.4, it is an error to alter a constant (i.e. the value of a literal expression) using a mutation procedure like `set-car!` or `string-set!`.

### 4.1.3. Procedure calls

`(`⟨operator⟩ ⟨operand$_1$⟩ ...`)` syntax

A procedure call is written by simply enclosing in parentheses expressions for the procedure to be called and the arguments to be passed to it. The operator and operand expressions are evaluated (in an unspecified order) and the resulting procedure is passed the resulting arguments.

```
(+ 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 (see section 4.1.4).

Procedure calls may return any number of values (see `values` in section 6.4). With the exception of `values` the procedures available in the initial environment return one value or, for procedures such as `apply`, pass on the values returned by a call to one of their arguments.

Procedure calls are also called *combinations*.

*Note:* In contrast to other dialects of Lisp, the order of evaluation is unspecified, and the operator expression and the operand expressions are always evaluated with the same evaluation rules.

*Note:* Although the order of evaluation is otherwise unspecified, the effect of any concurrent evaluation of the operator and operand expressions is constrained to be consistent with some sequential order of evaluation. The order of evaluation may be chosen differently for each procedure call.

*Note:* In many dialects of Lisp, the empty combination, `()`, is a legitimate expression. In Scheme, combinations must have at least one subexpression, so `()` is not a syntactically valid expression.

### 4.1.4. Procedures

`(lambda` ⟨formals⟩ ⟨body⟩`)` syntax

*Syntax:* ⟨Formals⟩ should be a formal arguments list as described below, and ⟨body⟩ should be a sequence of one or more expressions.

*Semantics:* A `lambda` expression evaluates to a procedure. The environment in effect when the `lambda` expression was evaluated is remembered as part of the procedure. When the procedure is later called with some actual arguments, the environment in which the `lambda` expression was evaluated will be extended by binding the variables in the formal argument list to fresh locations, the corresponding actual argument values will be stored in those locations, and the expressions in the body of the `lambda` expression will be evaluated sequentially in the extended environment. The result(s) of the last expression in the body will be returned as the result(s) of the procedure call.

```
(lambda (x) (+ x x))          ⟹  a procedure
((lambda (x) (+ x x)) 4)      ⟹  8

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

(define add4
  (let ((x 4))
    (lambda (y) (+ x y))))
(add4 6)                      ⟹  10
```

⟨Formals⟩ should have one of the following forms:

- `(`⟨variable$_1$⟩ ...`)`: The procedure takes a fixed number of arguments; when the procedure is called, the arguments will be stored in the bindings of the corresponding variables.
- ⟨variable⟩: The procedure takes any number of arguments; when the procedure is called, the sequence of actual arguments is converted into a newly allocated list, and the list is stored in the binding of the ⟨variable⟩.
- `(`⟨variable$_1$⟩ ... ⟨variable$_n$⟩ `.` ⟨variable$_{n+1}$⟩`)`: If a space-delimited period precedes the last variable, then the procedure takes $n$ or more arguments, where $n$ is the number of formal arguments before the period (there must be at least one). The value stored in the binding of the last variable will be a newly allocated list of the actual arguments left over after all the other actual arguments have been matched up against the other formal arguments.

It is an error for a ⟨variable⟩ to appear more than once in ⟨formals⟩.

```
((lambda x x) 3 4 5 6)         ⟹  (3 4 5 6)
((lambda (x y . z) z)
 3 4 5 6)                      ⟹  (5 6)
```

Each procedure created as the result of evaluating a `lambda` expression is (conceptually) tagged with a storage location, in order to make `eqv?` and `eq?` work on procedures (see section 6.1).

#### 4.1.5. Conditionals

`(if` ⟨test⟩ ⟨consequent⟩ ⟨alternate⟩`)` syntax
`(if` ⟨test⟩ ⟨consequent⟩`)` syntax

*Syntax:* ⟨Test⟩, ⟨consequent⟩, and ⟨alternate⟩ may be arbitrary expressions.

*Semantics:* An `if` expression is evaluated as follows: first, ⟨test⟩ is evaluated. If it yields a true value (see section 6.3.1), then ⟨consequent⟩ is evaluated and its value(s) is(are) returned. Otherwise ⟨alternate⟩ is evaluated and its value(s) is(are) returned. If ⟨test⟩ yields a false value and no ⟨alternate⟩ is specified, then the result of the expression is unspecified.

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

#### 4.1.6. Assignments

`(set!` ⟨variable⟩ ⟨expression⟩`)` syntax

⟨Expression⟩ is evaluated, and the resulting value is stored in the location to which ⟨variable⟩ is bound. ⟨Variable⟩ must be bound either in some region enclosing the `set!` expression or at top level. The result of the `set!` expression is unspecified.

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

### 4.2. Derived expression types

The constructs in this section are hygienic, as discussed in section 4.3. For reference purposes, section 7.3 gives macro definitions that will convert most of the constructs described in this section into the primitive constructs described in the previous section.

#### 4.2.1. Conditionals

`(cond` ⟨$\text{clause}_1$⟩ ⟨$\text{clause}_2$⟩ ...`)` library syntax

*Syntax:* Each ⟨clause⟩ should be of the form

`(`⟨test⟩ ⟨$\text{expression}_1$⟩ ...`)`

where ⟨test⟩ is any expression. Alternatively, a ⟨clause⟩ may be of the form

`(`⟨test⟩ `=>` ⟨expression⟩`)`

The last ⟨clause⟩ may be an "else clause," which has the form

`(else` ⟨$\text{expression}_1$⟩ ⟨$\text{expression}_2$⟩ ...`)`.

*Semantics:* A `cond` expression is evaluated by evaluating the ⟨test⟩ expressions of successive ⟨clause⟩s in order until one of them evaluates to a true value (see section 6.3.1). When a ⟨test⟩ evaluates to a true value, then the remaining ⟨expression⟩s in its ⟨clause⟩ are evaluated in order, and the result(s) of the last ⟨expression⟩ in the ⟨clause⟩ is(are) returned as the result(s) of the entire `cond` expression. If the selected ⟨clause⟩ contains only the ⟨test⟩ and no ⟨expression⟩s, then the value of the ⟨test⟩ is returned as the result. If the selected ⟨clause⟩ uses the `=>` alternate form, then the ⟨expression⟩ is evaluated. Its value must be a procedure that accepts one argument; this procedure is then called on the value of the ⟨test⟩ and the value(s) returned by this procedure is(are) returned by the `cond` expression. If all ⟨test⟩s evaluate to false values, and there is no else clause, then the result of the conditional expression is unspecified; if there is an else clause, then its ⟨expression⟩s are evaluated, and the value(s) of the last one is(are) returned.

```
(cond ((> 3 2) 'greater)
      ((< 3 2) 'less))         ⟹  greater
(cond ((> 3 3) 'greater)
      ((< 3 3) 'less)
      (else 'equal))           ⟹  equal
(cond ((assv 'b '((a 1) (b 2))) => cadr)
      (else #f))               ⟹  2
```

`(case` ⟨key⟩ ⟨$\text{clause}_1$⟩ ⟨$\text{clause}_2$⟩ ...`)` library syntax

*Syntax:* ⟨Key⟩ may be any expression. Each ⟨clause⟩ should have the form

`((`⟨$\text{datum}_1$⟩ ...`)` ⟨$\text{expression}_1$⟩ ⟨$\text{expression}_2$⟩ ...`)`,

where each ⟨datum⟩ is an external representation of some object. All the ⟨datum⟩s must be distinct. The last ⟨clause⟩ may be an "else clause," which has the form

`(else` ⟨$\text{expression}_1$⟩ ⟨$\text{expression}_2$⟩ ...`)`.

*Semantics:* A `case` expression is evaluated as follows. ⟨Key⟩ is evaluated and its result is compared against each ⟨datum⟩. If the result of evaluating ⟨key⟩ is equivalent (in the sense of `eqv?`; see section 6.1) to a ⟨datum⟩, then the expressions in the corresponding ⟨clause⟩ are evaluated from left to right and the result(s) of the last expression in the ⟨clause⟩ is(are) returned as the result(s) of the `case` expression. If the result of evaluating ⟨key⟩ is different from every ⟨datum⟩, then if there is an else clause its expressions are evaluated and the result(s) of the last is(are) the result(s) of the `case` expression; otherwise the result of the `case` expression is unspecified.

```
(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` $\langle\text{test}_1\rangle$ `...)` library syntax

The ⟨test⟩ expressions are evaluated from left to right, and the value of the first expression that evaluates to a false value (see section 6.3.1) 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` $\langle\text{test}_1\rangle$ `...)` library syntax

The ⟨test⟩ expressions are evaluated from left to right, and the value of the first expression that evaluates to a true value (see section 6.3.1) 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)
```

### 4.2.2. Binding constructs

The three binding constructs `let`, `let*`, and `letrec` give Scheme a block structure, like Algol 60. 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 bindings and evaluations are performed sequentially; while in a `letrec` expression, all the bindings are in effect while their initial values are being computed, thus allowing mutually recursive definitions.

`(let` ⟨bindings⟩ ⟨body⟩`)` library syntax

*Syntax:* ⟨Bindings⟩ should have the form

`((`$\langle\text{variable}_1\rangle$ $\langle\text{init}_1\rangle$`) ...),`

where each ⟨init⟩ is an expression, and ⟨body⟩ should be a sequence of one or more expressions. It is an error for a ⟨variable⟩ to appear more than once in the list of variables being bound.

*Semantics:* The ⟨init⟩s are evaluated in the current environment (in some unspecified order), the ⟨variable⟩s are bound to fresh locations holding the results, the ⟨body⟩ is evaluated in the extended environment, and the value(s) of the last expression of ⟨body⟩ is(are) returned. Each binding of a ⟨variable⟩ has ⟨body⟩ as its region.

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

(let ((x 2) (y 3))
  (let ((x 7)
        (z (+ x y)))
    (* z x)))           ⟹  35
```

See also named `let`, section 4.2.4.

`(let*` ⟨bindings⟩ ⟨body⟩`)` library syntax

*Syntax:* ⟨Bindings⟩ should have the form

`((`$\langle\text{variable}_1\rangle$ $\langle\text{init}_1\rangle$`) ...),`

and ⟨body⟩ should be a sequence of one or more expressions.

*Semantics:* `Let*` is similar to `let`, but the bindings are performed sequentially from left to right, and the region of a binding indicated by `(`⟨variable⟩ ⟨init⟩`)` 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.

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

`(letrec` ⟨bindings⟩ ⟨body⟩`)` library syntax

*Syntax:* ⟨Bindings⟩ should have the form

`((`$\langle\text{variable}_1\rangle$ $\langle\text{init}_1\rangle$`) ...),`

and ⟨body⟩ should be a sequence of one or more expressions. It is an error for a ⟨variable⟩ to appear more than once in the list of variables being bound.

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

```
(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 must be possible to evaluate each ⟨init⟩ without assigning or referring to the value of any ⟨variable⟩. If this restriction is violated, then it is an error. The restriction is necessary because Scheme passes arguments by value rather than by name. In the most common uses of `letrec`, all the ⟨init⟩s are `lambda` expressions and the restriction is satisfied automatically.

### 4.2.3. Sequencing

(`begin` ⟨expression$_1$⟩ ⟨expression$_2$⟩ ...) library syntax

The ⟨expression⟩s are evaluated sequentially from left to right, and the value(s) of the last ⟨expression⟩ is(are) 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)))    ⟹  unspecified
           and prints  4 plus 1 equals 5
```

### 4.2.4. Iteration

(`do` ((⟨variable$_1$⟩ ⟨init$_1$⟩ ⟨step$_1$⟩) library syntax
  ...)
 (⟨test⟩ ⟨expression⟩ ...)
 ⟨command⟩ ...)

`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 after evaluating the ⟨expression⟩s.

`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 (see section 6.3.1), 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(s) of the last ⟨expression⟩ is(are) returned. If no ⟨expression⟩s are present, then the value of the `do` expression is unspecified.

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

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

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

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

(`let` ⟨variable⟩ ⟨bindings⟩ ⟨body⟩) library syntax

"Named `let`" is a variant on the syntax of `let` which provides a more general looping construct than `do` and may also be used to express recursions. It has the same syntax and semantics as ordinary `let` except that ⟨variable⟩ is bound within ⟨body⟩ to a procedure whose formal arguments are the bound variables and whose body is ⟨body⟩. Thus the execution of ⟨body⟩ may be repeated by invoking the procedure named by ⟨variable⟩.

```
(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))
        ((< (car numbers) 0)
```

```
(loop (cdr numbers)
      nonneg
      (cons (car numbers) neg)))))
⟹ ((6 1 3) (-5 -2))
```

### 4.2.5. Delayed evaluation

(delay ⟨expression⟩) library syntax

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. The effect of ⟨expression⟩ returning multiple values is unspecified.

See the description of `force` (section 6.4) for a more complete description of `delay`.

### 4.2.6. Quasiquotation

(quasiquote ⟨qq template⟩) syntax
\`⟨qq template⟩ syntax

"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 ⟨qq template⟩, the result of evaluating \`⟨qq template⟩ is equivalent to the result of evaluating '⟨qq template⟩. If a comma appears within the ⟨qq 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 must 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. A comma at-sign should only appear within a list or vector ⟨qq template⟩.

```
`(list ,(+ 1 2) 4)          ⟹ (list 3 4)
(let ((name 'a)) `(list ,name ',name))
          ⟹ (list a (quote 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)
```

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 two notations \`⟨qq template⟩ and (quasiquote ⟨qq template⟩) are identical in all respects. ,⟨expression⟩ is identical to (unquote ⟨expression⟩), and ,@⟨expression⟩ is identical to (unquote-splicing ⟨expression⟩). The external syntax generated by `write` for two-element lists whose car is one of these symbols may vary between implementations.

```
(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 positions within a ⟨qq template⟩ otherwise than as described above.

## 4.3. Macros

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

(⟨keyword⟩ ⟨datum⟩ ...)

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

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

The macro definition facility consists of two parts:

- A set of expressions used to establish that certain identifiers are macro keywords, associate them with macro transformers, and control the scope within which a macro is defined, and
- a pattern language for specifying macro transformers.

The syntactic keyword of a macro may shadow variable bindings, and local variable bindings may shadow keyword bindings. All macros defined using the pattern language are "hygienic" and "referentially transparent" and thus preserve Scheme's lexical scoping [14, 15, 2, 7, 9]:

- If a macro transformer inserts a binding for an identifier (variable or keyword), the identifier will in effect be renamed throughout its scope to avoid conflicts with other identifiers. Note that a `define` at top level may or may not introduce a binding; see section 5.2.
- If a macro transformer inserts a free reference to an identifier, the reference refers to the binding that was visible where the transformer was specified, regardless of any local bindings that may surround the use of the macro.

### 4.3.1. Binding constructs for syntactic keywords

`Let-syntax` and `letrec-syntax` are analogous to `let` and `letrec`, but they bind syntactic keywords to macro transformers instead of binding variables to locations that contain values. Syntactic keywords may also be bound at top level; see section 5.3.

`(let-syntax` ⟨bindings⟩ ⟨body⟩`)` syntax

*Syntax:* ⟨Bindings⟩ should have the form

`((`⟨keyword⟩ ⟨transformer spec⟩`) ...)`

Each ⟨keyword⟩ is an identifier, each ⟨transformer spec⟩ is an instance of `syntax-rules`, and ⟨body⟩ should be a sequence of one or more expressions. It is an error for a ⟨keyword⟩ to appear more than once in the list of keywords being bound.

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

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

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

`(letrec-syntax` ⟨bindings⟩ ⟨body⟩`)` syntax

*Syntax:* Same as for `let-syntax`.

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

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

### 4.3.2. Pattern language

A ⟨transformer spec⟩ has the following form:

`(syntax-rules` ⟨literals⟩ ⟨syntax rule⟩ ...`)`

*Syntax:* ⟨Literals⟩ is a list of identifiers and each ⟨syntax rule⟩ should be of the form

`(`⟨pattern⟩ ⟨template⟩`)`

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

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

`(`⟨pattern⟩ `...)`
`(`⟨pattern⟩ ⟨pattern⟩ `... .` ⟨pattern⟩`)`
`(`⟨pattern⟩ `...` ⟨pattern⟩ ⟨ellipsis⟩`)`
`#(`⟨pattern⟩ `...)`
`#(`⟨pattern⟩ `...` ⟨pattern⟩ ⟨ellipsis⟩`)`

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

`(`⟨element⟩ `...)`
`(`⟨element⟩ ⟨element⟩ `... .` ⟨template⟩`)`
`#(`⟨element⟩ `...)`

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

*Semantics:* An instance of `syntax-rules` produces a new macro transformer by specifying a sequence of hygienic rewrite rules. A use of a macro whose keyword is associated with a transformer specified by `syntax-rules` is matched against the patterns contained in the ⟨syntax rule⟩s, beginning with the leftmost ⟨syntax rule⟩. When a match is found, the macro use is transcribed hygienically according to the template.

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

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

*Rationale:* The scope of the keyword is determined by the expression or syntax definition that binds it to the associated macro transformer. If the keyword were a pattern variable or literal identifier, then the template that follows the pattern would be within its scope regardless of whether the keyword were bound by `let-syntax` or by `letrec-syntax`.

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

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

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

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

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

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

Identifiers that appear in the template but are not pattern variables or the identifier `...` are inserted into the output as literal identifiers. If a literal identifier is inserted as a free identifier then it refers to the binding of that identifier within whose scope the instance of `syntax-rules` appears. If a literal identifier is inserted as a bound identifier then it is in effect renamed to prevent inadvertent captures of free identifiers.

As an example, if `let` and `cond` are defined as in section 7.3 then they are hygienic (as required) and the following is not an error.

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

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

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

instead of

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

which would result in an invalid procedure call.

# 5. Program structure

## 5.1. Programs

A Scheme program consists of a sequence of expressions, definitions, and syntax definitions. Expressions are described in chapter 4; definitions and syntax definitions are the subject of the rest of the present chapter.

Programs are typically stored in files or entered interactively to a running Scheme system, although other paradigms are possible; questions of user interface lie outside the scope of this report. (Indeed, Scheme would still be useful as a notation for expressing computational methods even in the absence of a mechanical implementation.)

Definitions and syntax definitions occurring at the top level of a program can be interpreted declaratively. They cause bindings to be created in the top level environment or modify the value of existing top-level bindings. Expressions occurring at the top level of a program are interpreted imperatively; they are executed in order when the program is invoked or loaded, and typically perform some kind of initialization.

At the top level of a program `(begin` ⟨form$_1$⟩ `...)` is equivalent to the sequence of expressions, definitions, and syntax definitions that form the body of the `begin`.

## 5.2. Definitions

Definitions are valid in some, but not all, contexts where expressions are allowed. They are valid only at the top level of a ⟨program⟩ and at the beginning of a ⟨body⟩.

A definition should have one of the following forms:

- `(define` ⟨variable⟩ ⟨expression⟩`)`
- `(define (`⟨variable⟩ ⟨formals⟩`)` ⟨body⟩`)`

  ⟨Formals⟩ should be either a sequence of zero or more variables, or a sequence of one or more variables followed by a space-delimited period and another variable (as in a lambda expression). This form is equivalent to

  ```
  (define ⟨variable⟩
    (lambda (⟨formals⟩) ⟨body⟩)).
  ```

- `(define (`⟨variable⟩ `.` ⟨formal⟩`)` ⟨body⟩`)`

  ⟨Formal⟩ should be a single variable. This form is equivalent to

  ```
  (define ⟨variable⟩
    (lambda ⟨formal⟩ ⟨body⟩)).
  ```

### 5.2.1. Top level definitions

At the top level of a program, a definition

```
(define ⟨variable⟩ ⟨expression⟩)
```

has essentially the same effect as the assignment expression

```
(set! ⟨variable⟩ ⟨expression⟩)
```

if ⟨variable⟩ is bound. If ⟨variable⟩ is not bound, however, then the definition will bind ⟨variable⟩ to a new location before performing the assignment, whereas it would be an error to perform a `set!` on an unbound variable.

```
(define add3
  (lambda (x) (+ x 3)))
(add3 3)                ⟹  6
(define first car)
(first '(1 2))          ⟹  1
```

Some implementations of Scheme use an initial environment in which all possible variables are bound to locations, most of which contain undefined values. Top level definitions in such an implementation are truly equivalent to assignments.

### 5.2.2. Internal definitions

Definitions may occur at the beginning of a ⟨body⟩ (that is, the body of a `lambda`, `let`, `let*`, `letrec`, `let-syntax`, or `letrec-syntax` expression or that of a definition of an appropriate form). Such definitions are known as *internal definitions* as opposed to the top level definitions described above. 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))))
```

Just as for the equivalent `letrec` expression, it must be possible to evaluate each ⟨expression⟩ of every internal definition in a ⟨body⟩ without assigning or referring to the value of any ⟨variable⟩ being defined.

Wherever an internal definition may occur `(begin` ⟨definition$_1$⟩ `...)` is equivalent to the sequence of definitions that form the body of the `begin`.

## 5.3. Syntax definitions

Syntax definitions are valid only at the top level of a ⟨program⟩. They have the following form:

`(define-syntax ⟨keyword⟩ ⟨transformer spec⟩)`

⟨Keyword⟩ is an identifier, and the ⟨transformer spec⟩ should be an instance of `syntax-rules`. The top-level syntactic environment is extended by binding the ⟨keyword⟩ to the specified transformer.

There is no `define-syntax` analogue of internal definitions.

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

```
(define define 3)

(begin (define begin list))

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

## 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?`.

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

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.

(eq? $obj_1$ $obj_2$)                                                  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, 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?`.

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

`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.

(`number?` *obj*) procedure
(`complex?` *obj*) procedure
(`real?` *obj*) procedure
(`rational?` *obj*) procedure
(`integer?` *obj*) 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.

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.

(`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.

(`=` $z_1$ $z_2$ $z_3$ ...) procedure
(`<` $x_1$ $x_2$ $x_3$ ...) procedure
(`>` $x_1$ $x_2$ $x_3$ ...) procedure
(`<=` $x_1$ $x_2$ $x_3$ ...) procedure
(`>=` $x_1$ $x_2$ $x_3$ ...) procedure

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.

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

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

`(max ` $x_1$ $x_2$ `...)` library procedure
`(min ` $x_1$ $x_2$ `...)` library 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.

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

These procedures return the sum or product of their arguments.

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

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

`(abs ` $x$`)` library procedure

`Abs` returns the absolute value of its argument.

```
(abs -7)           ⟹  7
```

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

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
```

(gcd $n_1$ ...) library procedure
(lcm $n_1$ ...) library 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 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
```

(rationalize $x$ $y$) library procedure

`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
```

(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 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 $\pi$ (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:

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

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

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

The above specification follows [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.

`(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 $z_1$ 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.

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

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.

`(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.

`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

`(number->string z)` procedure
`(number->string z 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 $z$ 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.

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

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

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

*Note:* 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
```

`(not obj)` library procedure

`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
```

`(boolean? obj)` library procedure

`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.

`(pair?` *obj*`)` procedure

`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
```

`(cons` $obj_1$ $obj_2$`)` procedure

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)
```

`(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
```

`(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
```

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

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
```

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

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

`(caar` *pair*`)` library procedure
`(cadr` *pair*`)` library procedure
⋮ ⋮
`(cdddar` *pair*`)` library procedure
`(cddddr` *pair*`)` library procedure

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.

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

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

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

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
```

(list *obj* ...) library procedure

Returns a newly allocated list of its arguments.

```
(list 'a (+ 3 4) 'c)       ⟹  (a 7 c)
(list)                     ⟹  ()
```

(length *list*) library procedure

Returns the length of *list*.

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

(append *list* ...) library procedure

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

```
(append '(x) '(y))         ⟹  (x y)
(append '(a) '(b c d))     ⟹  (a b c d)
(append '(a (b)) '((c)))   ⟹  (a (b) (c))
```

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
```

(reverse *list*) library procedure

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)
```

(list-tail *list k*) library procedure

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)))))
```

(list-ref *list k*) library procedure

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
```

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

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)
```

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

*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.

`(symbol? `*obj*`)` procedure

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
```

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

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"
```

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

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.

`(char? `*obj*`)` procedure

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

`(char=? `$char_1$ $char_2$`)` procedure
`(char<? `$char_1$ $char_2$`)` procedure
`(char>? `$char_1$ $char_2$`)` procedure
`(char<=? `$char_1$ $char_2$`)` procedure
`(char>=? `$char_1$ $char_2$`)` procedure

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.

`(char-ci=? `$char_1$ $char_2$`)` library procedure
`(char-ci<? `$char_1$ $char_2$`)` library procedure
`(char-ci>? `$char_1$ $char_2$`)` library procedure
`(char-ci<=? `$char_1$ $char_2$`)` library procedure
`(char-ci>=? `$char_1$ $char_2$`)` library procedure

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

`(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*`)` library procedure

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.

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

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
```

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

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.

`(string? ` *obj*`)` procedure

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

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

`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.

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

Returns a newly allocated string composed of the arguments.

`(string-length ` *string*`)` procedure

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

`(string-ref ` *string k*`)` procedure

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

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

*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
```

`(string=? ` *string*$_1$ *string*$_2$`)` library procedure
`(string-ci=? ` *string*$_1$ *string*$_2$`)` library procedure

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.

`(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*$_2$`)` library procedure

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.

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

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

$$0 \leq start \leq end \leq \texttt{(string-length } string\texttt{)}.$$

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

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

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

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

`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.

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

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

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

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")
```

`(vector? obj)` procedure

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

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

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.

`(vector obj ...)` library procedure

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

```
(vector 'a 'b 'c)          ⟹  #(a b c)
```

`(vector-length vector)` procedure

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

`(vector-ref vector k)` procedure

*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
```

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

*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
```

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

`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)
```

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

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? obj)` procedure

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
```

`(apply proc arg1 ... args)` procedure

*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
```

`(map proc list1 list2 ...)` library procedure

The *lists* must be lists, and *proc* must be a procedure taking as many arguments as there are *lists* 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 *lists* and returns a list of the results, in order. The dynamic order in which *proc* is applied to the elements of the *lists* 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)
```

`(for-each proc list1 list2 ...)` library procedure

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 *lists* 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)
```

`(force promise)` library procedure

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
  ```

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

*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.

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

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))))
```

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

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

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

(call-with-values * -)    ⟹  -1
```

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

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

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

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

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

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

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

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

          ⟹ (connect talk1 disconnect
               connect talk2 disconnect)
```

## 6.5. `Eval`

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

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
```

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

*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.

(`interaction-environment`) optional procedure

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.

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

*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`.

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

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

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

Returns the current default input or output port.

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

*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.

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

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.

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

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.

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

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

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

`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.

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

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`.

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

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.

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

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`.

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

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

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

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`.

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

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.

`(newline)` library procedure
`(newline port)` library procedure

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`.

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

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.

`(load filename)` optional procedure

*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.

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

*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.

# 7. Formal syntax and semantics

This chapter provides formal descriptions of what has already been described informally in previous chapters of this report.

## 7.1. Formal syntax

This section provides a formal syntax for Scheme written in an extended BNF.

All spaces in the grammar are for legibility. Case is insignificant; for example, `#x1A` and `#X1a` are equivalent. ⟨empty⟩ stands for the empty string.

The following extensions to BNF are used to make the description more concise: ⟨thing⟩* means zero or more occurrences of ⟨thing⟩; and ⟨thing⟩$^+$ means at least one ⟨thing⟩.

### 7.1.1. Lexical structure

This section describes how individual tokens (identifiers, numbers, etc.) are formed from sequences of characters. The following sections describe how expressions and programs are formed from sequences of tokens.

⟨Intertoken space⟩ may occur on either side of any token, but not within a token.

Tokens which require implicit termination (identifiers, numbers, characters, and dot) may be terminated by any ⟨delimiter⟩, but not necessarily by anything else.

The following five characters are reserved for future extensions to the language: `[ ] { } |`

⟨token⟩ ⟶ ⟨identifier⟩ | ⟨boolean⟩ | ⟨number⟩
    | ⟨character⟩ | ⟨string⟩
    | `(` | `)` | `#(` | `'` | `` ` `` | `,` | `,@` | `.`
⟨delimiter⟩ ⟶ ⟨whitespace⟩ | `(` | `)` | `"` | `;`
⟨whitespace⟩ ⟶ ⟨space or newline⟩
⟨comment⟩ ⟶ `;` ⟨all subsequent characters up to a line break⟩
⟨atmosphere⟩ ⟶ ⟨whitespace⟩ | ⟨comment⟩
⟨intertoken space⟩ ⟶ ⟨atmosphere⟩*

⟨identifier⟩ ⟶ ⟨initial⟩ ⟨subsequent⟩*
    | ⟨peculiar identifier⟩
⟨initial⟩ ⟶ ⟨letter⟩ | ⟨special initial⟩
⟨letter⟩ ⟶ `a` | `b` | `c` | ... | `z`

⟨special initial⟩ ⟶ `!` | `$` | `%` | `&` | `*` | `/` | `:` | `<` | `=`
    | `>` | `?` | `^` | `_` | `~`
⟨subsequent⟩ ⟶ ⟨initial⟩ | ⟨digit⟩
    | ⟨special subsequent⟩
⟨digit⟩ ⟶ `0` | `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9`
⟨special subsequent⟩ ⟶ `+` | `-` | `.` | `@`
⟨peculiar identifier⟩ ⟶ `+` | `-` | `...`

⟨syntactic keyword⟩ ⟶ ⟨expression keyword⟩
    | `else` | `=>` | `define`
    | `unquote` | `unquote-splicing`
⟨expression keyword⟩ ⟶ `quote` | `lambda` | `if`
    | `set!` | `begin` | `cond` | `and` | `or` | `case`
    | `let` | `let*` | `letrec` | `do` | `delay`
    | `quasiquote`

⟨variable⟩ ⟶ ⟨any ⟨identifier⟩ that isn't also a ⟨syntactic keyword⟩⟩

⟨boolean⟩ ⟶ `#t` | `#f`
⟨character⟩ ⟶ `#\` ⟨any character⟩
    | `#\` ⟨character name⟩
⟨character name⟩ ⟶ `space` | `newline`

⟨string⟩ ⟶ `"` ⟨string element⟩* `"`
⟨string element⟩ ⟶ ⟨any character other than `"` or `\`⟩
    | `\"` | `\\`

⟨number⟩ ⟶ ⟨num 2⟩| ⟨num 8⟩
    | ⟨num 10⟩| ⟨num 16⟩

The following rules for ⟨num $R$⟩, ⟨complex $R$⟩, ⟨real $R$⟩, ⟨ureal $R$⟩, ⟨uinteger $R$⟩, and ⟨prefix $R$⟩ should be replicated for $R = 2, 8, 10,$ and 16. There are no rules for ⟨decimal 2⟩, ⟨decimal 8⟩, and ⟨decimal 16⟩, which means that numbers containing decimal points or exponents must be in decimal radix.

⟨num $R$⟩ ⟶ ⟨prefix $R$⟩ ⟨complex $R$⟩
⟨complex $R$⟩ ⟶ ⟨real $R$⟩ | ⟨real $R$⟩ `@` ⟨real $R$⟩
    | ⟨real $R$⟩ `+` ⟨ureal $R$⟩ `i` | ⟨real $R$⟩ `-` ⟨ureal $R$⟩ `i`
    | ⟨real $R$⟩ `+ i` | ⟨real $R$⟩ `- i`
    | `+` ⟨ureal $R$⟩ `i` | `-` ⟨ureal $R$⟩ `i` | `+ i` | `- i`
⟨real $R$⟩ ⟶ ⟨sign⟩ ⟨ureal $R$⟩
⟨ureal $R$⟩ ⟶ ⟨uinteger $R$⟩
    | ⟨uinteger $R$⟩ `/` ⟨uinteger $R$⟩
    | ⟨decimal $R$⟩
⟨decimal 10⟩ ⟶ ⟨uinteger 10⟩ ⟨suffix⟩
    | `.` ⟨digit 10⟩$^+$ `#*` ⟨suffix⟩
    | ⟨digit 10⟩$^+$ `.` ⟨digit 10⟩* `#*` ⟨suffix⟩
    | ⟨digit 10⟩$^+$ `#`$^+$ `.` `#*` ⟨suffix⟩
⟨uinteger $R$⟩ ⟶ ⟨digit $R$⟩$^+$ `#*`
⟨prefix $R$⟩ ⟶ ⟨radix $R$⟩ ⟨exactness⟩
    | ⟨exactness⟩ ⟨radix $R$⟩

⟨suffix⟩ ⟶ ⟨empty⟩
    | ⟨exponent marker⟩ ⟨sign⟩ ⟨digit 10⟩$^+$
⟨exponent marker⟩ ⟶ `e` | `s` | `f` | `d` | `l`
⟨sign⟩ ⟶ ⟨empty⟩ | `+` | `-`
⟨exactness⟩ ⟶ ⟨empty⟩ | `#i` | `#e`
⟨radix 2⟩ ⟶ `#b`
⟨radix 8⟩ ⟶ `#o`
⟨radix 10⟩ ⟶ ⟨empty⟩ | `#d`

⟨radix 16⟩ ⟶ #x
⟨digit 2⟩ ⟶ 0 | 1
⟨digit 8⟩ ⟶ 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7
⟨digit 10⟩ ⟶ ⟨digit⟩
⟨digit 16⟩ ⟶ ⟨digit 10⟩ | a | b | c | d | e | f

### 7.1.2. External representations

⟨Datum⟩ is what the `read` procedure (section 6.6.2) successfully parses. Note that any string that parses as an ⟨expression⟩ will also parse as a ⟨datum⟩.

⟨datum⟩ ⟶ ⟨simple datum⟩ | ⟨compound datum⟩
⟨simple datum⟩ ⟶ ⟨boolean⟩ | ⟨number⟩
    | ⟨character⟩ | ⟨string⟩ | ⟨symbol⟩
⟨symbol⟩ ⟶ ⟨identifier⟩
⟨compound datum⟩ ⟶ ⟨list⟩ | ⟨vector⟩
⟨list⟩ ⟶ (⟨datum⟩*) | (⟨datum⟩$^{+}$ . ⟨datum⟩)
    | ⟨abbreviation⟩
⟨abbreviation⟩ ⟶ ⟨abbrev prefix⟩ ⟨datum⟩
⟨abbrev prefix⟩ ⟶ ' | ` | , | ,@
⟨vector⟩ ⟶ #(⟨datum⟩*)

### 7.1.3. Expressions

⟨expression⟩ ⟶ ⟨variable⟩
    | ⟨literal⟩
    | ⟨procedure call⟩
    | ⟨lambda expression⟩
    | ⟨conditional⟩
    | ⟨assignment⟩
    | ⟨derived expression⟩
    | ⟨macro use⟩
    | ⟨macro block⟩

⟨literal⟩ ⟶ ⟨quotation⟩ | ⟨self-evaluating⟩
⟨self-evaluating⟩ ⟶ ⟨boolean⟩ | ⟨number⟩
    | ⟨character⟩ | ⟨string⟩
⟨quotation⟩ ⟶ '⟨datum⟩ | (quote ⟨datum⟩)
⟨procedure call⟩ ⟶ (⟨operator⟩ ⟨operand⟩*)
⟨operator⟩ ⟶ ⟨expression⟩
⟨operand⟩ ⟶ ⟨expression⟩

⟨lambda expression⟩ ⟶ (lambda ⟨formals⟩ ⟨body⟩)
⟨formals⟩ ⟶ (⟨variable⟩*) | ⟨variable⟩
    | (⟨variable⟩$^{+}$ . ⟨variable⟩)
⟨body⟩ ⟶ ⟨definition⟩* ⟨sequence⟩
⟨sequence⟩ ⟶ ⟨command⟩* ⟨expression⟩
⟨command⟩ ⟶ ⟨expression⟩

⟨conditional⟩ ⟶ (if ⟨test⟩ ⟨consequent⟩ ⟨alternate⟩)
⟨test⟩ ⟶ ⟨expression⟩
⟨consequent⟩ ⟶ ⟨expression⟩
⟨alternate⟩ ⟶ ⟨expression⟩ | ⟨empty⟩

⟨assignment⟩ ⟶ (set! ⟨variable⟩ ⟨expression⟩)

⟨derived expression⟩ ⟶
    (cond ⟨cond clause⟩$^{+}$)
    | (cond ⟨cond clause⟩* (else ⟨sequence⟩))
    | (case ⟨expression⟩
        ⟨case clause⟩$^{+}$)
    | (case ⟨expression⟩
        ⟨case clause⟩*
        (else ⟨sequence⟩))
    | (and ⟨test⟩*)
    | (or ⟨test⟩*)
    | (let (⟨binding spec⟩*) ⟨body⟩)
    | (let ⟨variable⟩ (⟨binding spec⟩*) ⟨body⟩)
    | (let* (⟨binding spec⟩*) ⟨body⟩)
    | (letrec (⟨binding spec⟩*) ⟨body⟩)
    | (begin ⟨sequence⟩)
    | (do (⟨iteration spec⟩*)
            (⟨test⟩ ⟨do result⟩)
        ⟨command⟩*)
    | (delay ⟨expression⟩)
    | ⟨quasiquotation⟩

⟨cond clause⟩ ⟶ (⟨test⟩ ⟨sequence⟩)
    | (⟨test⟩)
    | (⟨test⟩ => ⟨recipient⟩)
⟨recipient⟩ ⟶ ⟨expression⟩
⟨case clause⟩ ⟶ ((⟨datum⟩*) ⟨sequence⟩)
⟨binding spec⟩ ⟶ (⟨variable⟩ ⟨expression⟩)
⟨iteration spec⟩ ⟶ (⟨variable⟩ ⟨init⟩ ⟨step⟩)
    | (⟨variable⟩ ⟨init⟩)
⟨init⟩ ⟶ ⟨expression⟩
⟨step⟩ ⟶ ⟨expression⟩
⟨do result⟩ ⟶ ⟨sequence⟩ | ⟨empty⟩

⟨macro use⟩ ⟶ (⟨keyword⟩ ⟨datum⟩*)
⟨keyword⟩ ⟶ ⟨identifier⟩

⟨macro block⟩ ⟶
    (let-syntax (⟨syntax spec⟩*) ⟨body⟩)
    | (letrec-syntax (⟨syntax spec⟩*) ⟨body⟩)
⟨syntax spec⟩ ⟶ (⟨keyword⟩ ⟨transformer spec⟩)

### 7.1.4. Quasiquotations

The following grammar for quasiquote expressions is not context-free. It is presented as a recipe for generating an infinite number of production rules. Imagine a copy of the following rules for $D = 1, 2, 3, \ldots$. $D$ keeps track of the nesting depth.

⟨quasiquotation⟩ ⟶ ⟨quasiquotation 1⟩
⟨qq template 0⟩ ⟶ ⟨expression⟩

⟨quasiquotation $D$⟩ ⟶ `` ` ``⟨qq template $D$⟩
   | `(quasiquote` ⟨qq template $D$⟩`)`
⟨qq template $D$⟩ ⟶ ⟨simple datum⟩
   | ⟨list qq template $D$⟩
   | ⟨vector qq template $D$⟩
   | ⟨unquotation $D$⟩
⟨list qq template $D$⟩ ⟶ `(`⟨qq template or splice $D$⟩*`)`
   | `(`⟨qq template or splice $D$⟩$^+$ `.` ⟨qq template $D$⟩`)`
   | `'`⟨qq template $D$⟩
   | ⟨quasiquotation $D+1$⟩
⟨vector qq template $D$⟩ ⟶ `#(`⟨qq template or splice $D$⟩*`)`
⟨unquotation $D$⟩ ⟶ `,`⟨qq template $D-1$⟩
   | `(unquote` ⟨qq template $D-1$⟩`)`
⟨qq template or splice $D$⟩ ⟶ ⟨qq template $D$⟩
   | ⟨splicing unquotation $D$⟩
⟨splicing unquotation $D$⟩ ⟶ `,@`⟨qq template $D-1$⟩
   | `(unquote-splicing` ⟨qq template $D-1$⟩`)`

In ⟨quasiquotation⟩s, a ⟨list qq template $D$⟩ can sometimes be confused with either an ⟨unquotation $D$⟩ or a ⟨splicing unquotation $D$⟩. The interpretation as an ⟨unquotation⟩ or ⟨splicing unquotation $D$⟩ takes precedence.

#### 7.1.5. Transformers

⟨transformer spec⟩ ⟶
   `(syntax-rules (`⟨identifier⟩*`)` ⟨syntax rule⟩*`)`
⟨syntax rule⟩ ⟶ `(`⟨pattern⟩ ⟨template⟩`)`
⟨pattern⟩ ⟶ ⟨pattern identifier⟩
   | `(`⟨pattern⟩*`)`
   | `(`⟨pattern⟩$^+$ `.` ⟨pattern⟩`)`
   | `(`⟨pattern⟩* ⟨pattern⟩ ⟨ellipsis⟩`)`
   | `#(`⟨pattern⟩*`)`
   | `#(`⟨pattern⟩* ⟨pattern⟩ ⟨ellipsis⟩`)`
   | ⟨pattern datum⟩
⟨pattern datum⟩ ⟶ ⟨string⟩
   | ⟨character⟩
   | ⟨boolean⟩
   | ⟨number⟩
⟨template⟩ ⟶ ⟨pattern identifier⟩
   | `(`⟨template element⟩*`)`
   | `(`⟨template element⟩$^+$ `.` ⟨template⟩`)`
   | `#(`⟨template element⟩*`)`
   | ⟨template datum⟩
⟨template element⟩ ⟶ ⟨template⟩
   | ⟨template⟩ ⟨ellipsis⟩
⟨template datum⟩ ⟶ ⟨pattern datum⟩
⟨pattern identifier⟩ ⟶ ⟨any identifier except `...`⟩
⟨ellipsis⟩ ⟶ ⟨the identifier `...`⟩

#### 7.1.6. Programs and definitions

⟨program⟩ ⟶ ⟨command or definition⟩*
⟨command or definition⟩ ⟶ ⟨command⟩
   | ⟨definition⟩
   | ⟨syntax definition⟩
   | `(begin` ⟨command or definition⟩$^+$`)`
⟨definition⟩ ⟶ `(define` ⟨variable⟩ ⟨expression⟩`)`
   | `(define (`⟨variable⟩ ⟨def formals⟩`)` ⟨body⟩`)`
   | `(begin` ⟨definition⟩*`)`
⟨def formals⟩ ⟶ ⟨variable⟩*
   | ⟨variable⟩* `.` ⟨variable⟩
⟨syntax definition⟩ ⟶
   `(define-syntax` ⟨keyword⟩ ⟨transformer spec⟩`)`

### 7.2. Formal semantics

This section provides a formal denotational semantics for the primitive expressions of Scheme and selected built-in procedures. The concepts and notation used here are described in [29]; the notation is summarized below:

| | |
|---|---|
| $\langle \dots \rangle$ | sequence formation |
| $s \downarrow k$ | $k$th member of the sequence $s$ (1-based) |
| $\#s$ | length of sequence $s$ |
| $s \mathbin{\S} t$ | concatenation of sequences $s$ and $t$ |
| $s \dagger k$ | drop the first $k$ members of sequence $s$ |
| $t \rightarrow a, b$ | McCarthy conditional "if $t$ then $a$ else $b$" |
| $\rho[x/i]$ | substitution "$\rho$ with $x$ for $i$" |
| $x$ in $\mathtt{D}$ | injection of $x$ into domain $\mathtt{D}$ |
| $x \mid \mathtt{D}$ | projection of $x$ to domain $\mathtt{D}$ |

The reason that expression continuations take sequences of values instead of single values is to simplify the formal treatment of procedure calls and multiple return values.

The boolean flag associated with pairs, vectors, and strings will be true for mutable objects and false for immutable objects.

The order of evaluation within a call is unspecified. We mimic that here by applying arbitrary permutations *permute* and *unpermute*, which must be inverses, to the arguments in a call before and after they are evaluated. This is not quite right since it suggests, incorrectly, that the order of evaluation is constant throughout a program (for any given number of arguments), but it is a closer approximation to the intended semantics than a left-to-right evaluation would be.

The storage allocator *new* is implementation-dependent, but it must obey the following axiom: if $\mathit{new}\,\sigma \in \mathtt{L}$, then $\sigma\,(\mathit{new}\,\sigma \mid \mathtt{L}) \downarrow 2 = \mathit{false}$.

The definition of $\mathcal{K}$ is omitted because an accurate definition of $\mathcal{K}$ would complicate the semantics without being very interesting.

If P is a program in which all variables are defined before being referenced or assigned, then the meaning of P is

$$\mathcal{E}[\![\texttt{((lambda (I*) P')} \ \langle\text{undefined}\rangle \ \dots\texttt{)}]\!]$$

where I* is the sequence of variables defined in P, P′ is the sequence of expressions obtained by replacing every definition in P by an assignment, ⟨undefined⟩ is an expression that evaluates to *undefined*, and $\mathcal{E}$ is the semantic function that assigns meaning to expressions.

### 7.2.1. Abstract syntax

| | |
|---|---|
| $\mathrm{K} \in \mathrm{Con}$ | constants, including quotations |
| $\mathrm{I} \in \mathrm{Ide}$ | identifiers (variables) |
| $\mathrm{E} \in \mathrm{Exp}$ | expressions |
| $\Gamma \in \mathrm{Com} = \mathrm{Exp}$ | commands |

$$
\begin{aligned}
\mathrm{Exp} \longrightarrow\; & \mathrm{K} \mid \mathrm{I} \mid \texttt{(}\mathrm{E}_0\ \mathrm{E}^*\texttt{)}\\
& \mid \texttt{(lambda (}\mathrm{I}^*\texttt{)}\ \Gamma^*\ \mathrm{E}_0\texttt{)}\\
& \mid \texttt{(lambda (}\mathrm{I}^*\ \texttt{.}\ \mathrm{I}\texttt{)}\ \Gamma^*\ \mathrm{E}_0\texttt{)}\\
& \mid \texttt{(lambda}\ \mathrm{I}\ \Gamma^*\ \mathrm{E}_0\texttt{)}\\
& \mid \texttt{(if}\ \mathrm{E}_0\ \mathrm{E}_1\ \mathrm{E}_2\texttt{)} \mid \texttt{(if}\ \mathrm{E}_0\ \mathrm{E}_1\texttt{)}\\
& \mid \texttt{(set!}\ \mathrm{I}\ \mathrm{E}\texttt{)}
\end{aligned}
$$

### 7.2.2. Domain equations

| | | |
|---|---|---|
| $\alpha \in \mathtt{L}$ | | locations |
| $\nu \in \mathtt{N}$ | | natural numbers |
| $\mathtt{T}$ | $= \{\mathit{false}, \mathit{true}\}$ | booleans |
| $\mathtt{Q}$ | | symbols |
| $\mathtt{H}$ | | characters |
| $\mathtt{R}$ | | numbers |
| $\mathtt{E_p}$ | $= \mathtt{L} \times \mathtt{L} \times \mathtt{T}$ | pairs |
| $\mathtt{E_v}$ | $= \mathtt{L}^* \times \mathtt{T}$ | vectors |
| $\mathtt{E_s}$ | $= \mathtt{L}^* \times \mathtt{T}$ | strings |
| $\mathtt{M}$ | $= \{\mathit{false}, \mathit{true}, \mathit{null}, \mathit{undefined}, \mathit{unspecified}\}$ | miscellaneous |
| $\phi \in \mathtt{F}$ | $= \mathtt{L} \times (\mathtt{E}^* \to \mathtt{K} \to \mathtt{C})$ | procedure values |
| $\epsilon \in \mathtt{E}$ | $= \mathtt{Q} + \mathtt{H} + \mathtt{R} + \mathtt{E_p} + \mathtt{E_v} + \mathtt{E_s} + \mathtt{M} + \mathtt{F}$ | expressed values |
| $\sigma \in \mathtt{S}$ | $= \mathtt{L} \to (\mathtt{E} \times \mathtt{T})$ | stores |
| $\rho \in \mathtt{U}$ | $= \mathrm{Ide} \to \mathtt{L}$ | environments |
| $\theta \in \mathtt{C}$ | $= \mathtt{S} \to \mathtt{A}$ | command continuations |
| $\kappa \in \mathtt{K}$ | $= \mathtt{E}^* \to \mathtt{C}$ | expression continuations |
| $\mathtt{A}$ | | answers |
| $\mathtt{X}$ | | errors |

### 7.2.3. Semantic functions

$$
\begin{aligned}
\mathcal{K} &: \mathrm{Con} \to \mathtt{E}\\
\mathcal{E} &: \mathrm{Exp} \to \mathtt{U} \to \mathtt{K} \to \mathtt{C}\\
\mathcal{E}^* &: \mathrm{Exp}^* \to \mathtt{U} \to \mathtt{K} \to \mathtt{C}\\
\mathcal{C} &: \mathrm{Com}^* \to \mathtt{U} \to \mathtt{C} \to \mathtt{C}
\end{aligned}
$$

Definition of $\mathcal{K}$ deliberately omitted.

$$\mathcal{E}[\![\mathrm{K}]\!] = \lambda\rho\kappa\,.\,\mathit{send}\,(\mathcal{K}[\![\mathrm{K}]\!])\,\kappa$$

$$
\begin{aligned}
\mathcal{E}[\![\mathrm{I}]\!] = \lambda\rho\kappa\,.\,&\mathit{hold}\,(\mathit{lookup}\,\rho\,\mathrm{I})\\
&(\mathit{single}(\lambda\epsilon\,.\,\epsilon = \mathit{undefined} \to\\
&\qquad \mathit{wrong}\ \text{“undefined variable”},\\
&\qquad \mathit{send}\,\epsilon\,\kappa))
\end{aligned}
$$

$$
\begin{aligned}
&\mathcal{E}[\![\texttt{(}\mathrm{E}_0\ \mathrm{E}^*\texttt{)}]\!] =\\
&\quad \lambda\rho\kappa\,.\,\mathcal{E}^*(\mathit{permute}(\langle\mathrm{E}_0\rangle \mathbin{\S} \mathrm{E}^*))\\
&\qquad\quad \rho\\
&\qquad\quad (\lambda\epsilon^*\,.\,((\lambda\epsilon^*\,.\,\mathit{applicate}\,(\epsilon^* \downarrow 1)\,(\epsilon^* \dagger 1)\,\kappa)\\
&\qquad\qquad (\mathit{unpermute}\,\epsilon^*)))
\end{aligned}
$$

$$
\begin{aligned}
&\mathcal{E}[\![\texttt{(lambda (}\mathrm{I}^*\texttt{)}\ \Gamma^*\ \mathrm{E}_0\texttt{)}]\!] =\\
&\quad \lambda\rho\kappa\,.\,\lambda\sigma\,.\\
&\qquad \mathit{new}\,\sigma \in \mathtt{L} \to\\
&\qquad\quad \mathit{send}\,(\langle \mathit{new}\,\sigma \mid \mathtt{L},\\
&\qquad\qquad\quad \lambda\epsilon^*\kappa'\,.\,\#\epsilon^* = \#\mathrm{I}^* \to\\
&\qquad\qquad\qquad \mathit{tievals}(\lambda\alpha^*\,.\,(\lambda\rho'\,.\,\mathcal{C}[\![\Gamma^*]\!]\rho'(\mathcal{E}[\![\mathrm{E}_0]\!]\rho'\kappa'))\\
&\qquad\qquad\qquad\qquad (\mathit{extends}\,\rho\,\mathrm{I}^*\,\alpha^*))\\
&\qquad\qquad\qquad\quad \epsilon^*,\\
&\qquad\qquad\qquad \mathit{wrong}\ \text{“wrong number of arguments”}\rangle\\
&\qquad\qquad\quad \text{in }\mathtt{E})\\
&\qquad\qquad \kappa\\
&\qquad\qquad (\mathit{update}\,(\mathit{new}\,\sigma \mid \mathtt{L})\,\mathit{unspecified}\,\sigma),\\
&\qquad\quad \mathit{wrong}\ \text{“out of memory”}\ \sigma
\end{aligned}
$$

$$
\begin{aligned}
&\mathcal{E}[\![\texttt{(lambda (}\mathrm{I}^*\ \texttt{.}\ \mathrm{I}\texttt{)}\ \Gamma^*\ \mathrm{E}_0\texttt{)}]\!] =\\
&\quad \lambda\rho\kappa\,.\,\lambda\sigma\,.\\
&\qquad \mathit{new}\,\sigma \in \mathtt{L} \to\\
&\qquad\quad \mathit{send}\,(\langle \mathit{new}\,\sigma \mid \mathtt{L},\\
&\qquad\qquad\quad \lambda\epsilon^*\kappa'\,.\,\#\epsilon^* \geq \#\mathrm{I}^* \to\\
&\qquad\qquad\qquad \mathit{tievalsrest}\\
&\qquad\qquad\qquad\quad (\lambda\alpha^*\,.\,(\lambda\rho'\,.\,\mathcal{C}[\![\Gamma^*]\!]\rho'(\mathcal{E}[\![\mathrm{E}_0]\!]\rho'\kappa'))\\
&\qquad\qquad\qquad\qquad (\mathit{extends}\,\rho\,(\mathrm{I}^* \mathbin{\S} \langle\mathrm{I}\rangle)\,\alpha^*))\\
&\qquad\qquad\qquad\quad \epsilon^*\\
&\qquad\qquad\qquad\quad (\#\mathrm{I}^*),\\
&\qquad\qquad\qquad \mathit{wrong}\ \text{“too few arguments”}\rangle\ \text{in }\mathtt{E})\\
&\qquad\qquad \kappa\\
&\qquad\qquad (\mathit{update}\,(\mathit{new}\,\sigma \mid \mathtt{L})\,\mathit{unspecified}\,\sigma),\\
&\qquad\quad \mathit{wrong}\ \text{“out of memory”}\ \sigma
\end{aligned}
$$

$$\mathcal{E}[\![\texttt{(lambda}\ \mathrm{I}\ \Gamma^*\ \mathrm{E}_0\texttt{)}]\!] = \mathcal{E}[\![\texttt{(lambda (. }\mathrm{I}\texttt{)}\ \Gamma^*\ \mathrm{E}_0\texttt{)}]\!]$$

$$
\begin{aligned}
&\mathcal{E}[\![\texttt{(if}\ \mathrm{E}_0\ \mathrm{E}_1\ \mathrm{E}_2\texttt{)}]\!] =\\
&\quad \lambda\rho\kappa\,.\,\mathcal{E}[\![\mathrm{E}_0]\!]\,\rho\,(\mathit{single}\,(\lambda\epsilon\,.\,\mathit{truish}\,\epsilon \to \mathcal{E}[\![\mathrm{E}_1]\!]\rho\kappa,\\
&\qquad\qquad\qquad\qquad \mathcal{E}[\![\mathrm{E}_2]\!]\rho\kappa))
\end{aligned}
$$

$$
\begin{aligned}
&\mathcal{E}[\![\texttt{(if}\ \mathrm{E}_0\ \mathrm{E}_1\texttt{)}]\!] =\\
&\quad \lambda\rho\kappa\,.\,\mathcal{E}[\![\mathrm{E}_0]\!]\,\rho\,(\mathit{single}\,(\lambda\epsilon\,.\,\mathit{truish}\,\epsilon \to \mathcal{E}[\![\mathrm{E}_1]\!]\rho\kappa,\\
&\qquad\qquad\qquad\qquad \mathit{send}\ \mathit{unspecified}\,\kappa))
\end{aligned}
$$

Here and elsewhere, any expressed value other than *undefined* may be used in place of *unspecified*.

$$
\begin{aligned}
&\mathcal{E}[\![\texttt{(set!}\ \mathrm{I}\ \mathrm{E}\texttt{)}]\!] =\\
&\quad \lambda\rho\kappa\,.\,\mathcal{E}[\![\mathrm{E}]\!]\,\rho\,(\mathit{single}(\lambda\epsilon\,.\,\mathit{assign}\,(\mathit{lookup}\,\rho\,\mathrm{I})\\
&\qquad\qquad\qquad\qquad \epsilon\\
&\qquad\qquad\qquad\qquad (\mathit{send}\ \mathit{unspecified}\,\kappa)))
\end{aligned}
$$

$$\mathcal{E}^*[\![\ ]\!] = \lambda\rho\kappa\,.\,\kappa\langle\,\rangle$$

$$
\begin{aligned}
&\mathcal{E}^*[\![\mathrm{E}_0\ \mathrm{E}^*]\!] =\\
&\quad \lambda\rho\kappa\,.\,\mathcal{E}[\![\mathrm{E}_0]\!]\,\rho\,(\mathit{single}(\lambda\epsilon_0\,.\,\mathcal{E}^*[\![\mathrm{E}^*]\!]\,\rho\,(\lambda\epsilon^*\,.\,\kappa\,(\langle\epsilon_0\rangle \mathbin{\S} \epsilon^*))))
\end{aligned}
$$

$$\mathcal{C}[\![\ ]\!] = \lambda\rho\theta\,.\,\theta$$

$$\mathcal{C}[\![\Gamma_0\ \Gamma^*]\!] = \lambda\rho\theta\,.\,\mathcal{E}[\![\Gamma_0]\!]\,\rho\,(\lambda\epsilon^*\,.\,\mathcal{C}[\![\Gamma^*]\!]\rho\theta)$$

### 7.2.4. Auxiliary functions

$lookup : \mathtt{U} \to \mathrm{Ide} \to \mathtt{L}$
$lookup = \lambda\rho\mathrm{I}\,.\,\rho\mathrm{I}$

$extends : \mathtt{U} \to \mathrm{Ide}^* \to \mathtt{L}^* \to \mathtt{U}$
$extends =$
$\quad \lambda\rho\mathrm{I}^*\alpha^*\,.\,\#\mathrm{I}^* = 0 \to \rho,$
$\qquad extends\,(\rho[(\alpha^* \downarrow 1)/(\mathrm{I}^* \downarrow 1)])\,(\mathrm{I}^* \dagger 1)\,(\alpha^* \dagger 1)$

$wrong : \mathtt{X} \to \mathtt{C}$ [implementation-dependent]

$send : \mathtt{E} \to \mathtt{K} \to \mathtt{C}$
$send = \lambda\epsilon\kappa\,.\,\kappa\langle\epsilon\rangle$

$single : (\mathtt{E} \to \mathtt{C}) \to \mathtt{K}$
$single =$
$\quad \lambda\psi\epsilon^*\,.\,\#\epsilon^* = 1 \to \psi(\epsilon^* \downarrow 1),$
$\qquad wrong$ "wrong number of return values"

$new : \mathtt{S} \to (\mathtt{L} + \{error\})$ [implementation-dependent]

$hold : \mathtt{L} \to \mathtt{K} \to \mathtt{C}$
$hold = \lambda\alpha\kappa\sigma\,.\,send\,(\sigma\alpha \downarrow 1)\kappa\sigma$

$assign : \mathtt{L} \to \mathtt{E} \to \mathtt{C} \to \mathtt{C}$
$assign = \lambda\alpha\epsilon\theta\sigma\,.\,\theta(update\,\alpha\epsilon\sigma)$

$update : \mathtt{L} \to \mathtt{E} \to \mathtt{S} \to \mathtt{S}$
$update = \lambda\alpha\epsilon\sigma\,.\,\sigma[\langle\epsilon, true\rangle/\alpha]$

$tievals : (\mathtt{L}^* \to \mathtt{C}) \to \mathtt{E}^* \to \mathtt{C}$
$tievals =$
$\quad \lambda\psi\epsilon^*\sigma\,.\,\#\epsilon^* = 0 \to \psi\langle\,\rangle\sigma,$
$\qquad new\,\sigma \in \mathtt{L} \to tievals\,(\lambda\alpha^*\,.\,\psi(\langle new\,\sigma \mid \mathtt{L}\rangle \S\, \alpha^*))$
$\qquad\qquad (\epsilon^* \dagger 1)$
$\qquad\qquad (update(new\,\sigma \mid \mathtt{L})(\epsilon^* \downarrow 1)\sigma),$
$\qquad wrong$ "out of memory"$\sigma$

$tievalsrest : (\mathtt{L}^* \to \mathtt{C}) \to \mathtt{E}^* \to \mathtt{N} \to \mathtt{C}$
$tievalsrest =$
$\quad \lambda\psi\epsilon^*\nu\,.\,list\,(dropfirst\,\epsilon^*\nu)$
$\qquad (single(\lambda\epsilon\,.\,tievals\,\psi\,((takefirst\,\epsilon^*\nu)\,\S\,\langle\epsilon\rangle)))$

$dropfirst = \lambda ln\,.\,n = 0 \to l, dropfirst\,(l \dagger 1)(n-1)$

$takefirst = \lambda ln\,.\,n = 0 \to \langle\,\rangle, \langle l \downarrow 1\rangle\,\S\,(takefirst\,(l \dagger 1)(n-1))$

$truish : \mathtt{E} \to \mathtt{T}$
$truish = \lambda\epsilon\,.\,\epsilon = false \to false, true$

$permute : \mathrm{Exp}^* \to \mathrm{Exp}^*$ [implementation-dependent]

$unpermute : \mathtt{E}^* \to \mathtt{E}^*$ [inverse of $permute$]

$applicate : \mathtt{E} \to \mathtt{E}^* \to \mathtt{K} \to \mathtt{C}$
$applicate =$
$\quad \lambda\epsilon\epsilon^*\kappa\,.\,\epsilon \in \mathtt{F} \to (\epsilon \mid \mathtt{F} \downarrow 2)\epsilon^*\kappa, wrong$ "bad procedure"

$onearg : (\mathtt{E} \to \mathtt{K} \to \mathtt{C}) \to (\mathtt{E}^* \to \mathtt{K} \to \mathtt{C})$
$onearg =$
$\quad \lambda\zeta\epsilon^*\kappa\,.\,\#\epsilon^* = 1 \to \zeta(\epsilon^* \downarrow 1)\kappa,$
$\qquad wrong$ "wrong number of arguments"

$twoarg : (\mathtt{E} \to \mathtt{E} \to \mathtt{K} \to \mathtt{C}) \to (\mathtt{E}^* \to \mathtt{K} \to \mathtt{C})$
$twoarg =$
$\quad \lambda\zeta\epsilon^*\kappa\,.\,\#\epsilon^* = 2 \to \zeta(\epsilon^* \downarrow 1)(\epsilon^* \downarrow 2)\kappa,$
$\qquad wrong$ "wrong number of arguments"

$list : \mathtt{E}^* \to \mathtt{K} \to \mathtt{C}$
$list =$
$\quad \lambda\epsilon^*\kappa\,.\,\#\epsilon^* = 0 \to send\,null\,\kappa,$
$\qquad list\,(\epsilon^* \dagger 1)(single(\lambda\epsilon\,.\,cons\langle\epsilon^* \downarrow 1, \epsilon\rangle\kappa))$

$cons : \mathtt{E}^* \to \mathtt{K} \to \mathtt{C}$
$cons =$
$\quad twoarg\,(\lambda\epsilon_1\epsilon_2\kappa\sigma\,.\,new\,\sigma \in \mathtt{L} \to$
$\qquad (\lambda\sigma'\,.\,new\,\sigma' \in \mathtt{L} \to$
$\qquad\quad send\,(\langle new\,\sigma \mid \mathtt{L}, new\,\sigma' \mid \mathtt{L}, true\rangle$
$\qquad\qquad \text{in } \mathtt{E})$
$\qquad\quad \kappa$
$\qquad\quad (update(new\,\sigma' \mid \mathtt{L})\epsilon_2\sigma'),$
$\qquad\quad wrong$ "out of memory"$\sigma'$)
$\qquad (update(new\,\sigma \mid \mathtt{L})\epsilon_1\sigma),$
$\qquad wrong$ "out of memory"$\sigma$)

$less : \mathtt{E}^* \to \mathtt{K} \to \mathtt{C}$
$less =$
$\quad twoarg\,(\lambda\epsilon_1\epsilon_2\kappa\,.\,(\epsilon_1 \in \mathtt{R} \wedge \epsilon_2 \in \mathtt{R}) \to$
$\qquad send\,(\epsilon_1 \mid \mathtt{R} < \epsilon_2 \mid \mathtt{R} \to true, false)\kappa,$
$\qquad wrong$ "non-numeric argument to `<`")

$add : \mathtt{E}^* \to \mathtt{K} \to \mathtt{C}$
$add =$
$\quad twoarg\,(\lambda\epsilon_1\epsilon_2\kappa\,.\,(\epsilon_1 \in \mathtt{R} \wedge \epsilon_2 \in \mathtt{R}) \to$
$\qquad send\,((\epsilon_1 \mid \mathtt{R} + \epsilon_2 \mid \mathtt{R}) \text{ in } \mathtt{E})\kappa,$
$\qquad wrong$ "non-numeric argument to `+`")

$car : \mathtt{E}^* \to \mathtt{K} \to \mathtt{C}$
$car =$
$\quad onearg\,(\lambda\epsilon\kappa\,.\,\epsilon \in \mathtt{E_p} \to hold\,(\epsilon \mid \mathtt{E_p} \downarrow 1)\kappa,$
$\qquad wrong$ "non-pair argument to `car`")

$cdr : \mathtt{E}^* \to \mathtt{K} \to \mathtt{C}$ [similar to $car$]

$setcar : \mathtt{E}^* \to \mathtt{K} \to \mathtt{C}$
$setcar =$
$\quad twoarg\,(\lambda\epsilon_1\epsilon_2\kappa\,.\,\epsilon_1 \in \mathtt{E_p} \to$
$\qquad (\epsilon_1 \mid \mathtt{E_p} \downarrow 3) \to assign\,(\epsilon_1 \mid \mathtt{E_p} \downarrow 1)$
$\qquad\qquad \epsilon_2$
$\qquad\qquad (send\,unspecified\,\kappa),$
$\qquad wrong$ "immutable argument to `set-car!`",
$\qquad wrong$ "non-pair argument to `set-car!`")

$eqv : \mathtt{E}^* \to \mathtt{K} \to \mathtt{C}$
$eqv =$
$\quad twoarg\,(\lambda\epsilon_1\epsilon_2\kappa\,.\,(\epsilon_1 \in \mathtt{M} \wedge \epsilon_2 \in \mathtt{M}) \to$
$\qquad send\,(\epsilon_1 \mid \mathtt{M} = \epsilon_2 \mid \mathtt{M} \to true, false)\kappa,$
$\qquad (\epsilon_1 \in \mathtt{Q} \wedge \epsilon_2 \in \mathtt{Q}) \to$
$\qquad send\,(\epsilon_1 \mid \mathtt{Q} = \epsilon_2 \mid \mathtt{Q} \to true, false)\kappa,$
$\qquad (\epsilon_1 \in \mathtt{H} \wedge \epsilon_2 \in \mathtt{H}) \to$
$\qquad send\,(\epsilon_1 \mid \mathtt{H} = \epsilon_2 \mid \mathtt{H} \to true, false)\kappa,$
$\qquad (\epsilon_1 \in \mathtt{R} \wedge \epsilon_2 \in \mathtt{R}) \to$
$\qquad send\,(\epsilon_1 \mid \mathtt{R} = \epsilon_2 \mid \mathtt{R} \to true, false)\kappa,$
$\qquad (\epsilon_1 \in \mathtt{E_p} \wedge \epsilon_2 \in \mathtt{E_p}) \to$
$\qquad send\,((\lambda p_1 p_2\,.\,((p_1 \downarrow 1) = (p_2 \downarrow 1) \wedge$
$\qquad\qquad (p_1 \downarrow 2) = (p_2 \downarrow 2)) \to true,$
$\qquad\qquad false)$
$\qquad\quad (\epsilon_1 \mid \mathtt{E_p})$
$\qquad\quad (\epsilon_2 \mid \mathtt{E_p}))$
$\qquad \kappa,$

$$
\begin{array}{l}
\qquad\qquad (\epsilon_1 \in \mathtt{E_v} \wedge \epsilon_2 \in \mathtt{E_v}) \to \ldots, \\
\qquad\qquad (\epsilon_1 \in \mathtt{E_s} \wedge \epsilon_2 \in \mathtt{E_s}) \to \ldots, \\
\qquad\qquad (\epsilon_1 \in \mathtt{F} \wedge \epsilon_2 \in \mathtt{F}) \to \\
\qquad\qquad\quad \mathit{send}\,((\epsilon_1 \mid \mathtt{F} \downarrow 1) = (\epsilon_2 \mid \mathtt{F} \downarrow 1) \to \mathit{true}, \mathit{false}) \\
\qquad\qquad\qquad \kappa, \\
\qquad\qquad\quad \mathit{send}\ \mathit{false}\,\kappa)
\end{array}
$$

$$
\begin{array}{l}
\mathit{apply} : \mathtt{E}^* \to \mathtt{K} \to \mathtt{C} \\
\mathit{apply} = \\
\quad \mathit{twoarg}\,(\lambda\epsilon_1\epsilon_2\kappa\,.\,\epsilon_1 \in \mathtt{F} \to \mathit{valueslist}\,\langle\epsilon_2\rangle(\lambda\epsilon^*\,.\,\mathit{applicate}\,\epsilon_1\epsilon^*\kappa), \\
\qquad\qquad\qquad \mathit{wrong}\ \text{“bad procedure argument to }\mathtt{apply}\text{”})
\end{array}
$$

$$
\begin{array}{l}
\mathit{valueslist} : \mathtt{E}^* \to \mathtt{K} \to \mathtt{C} \\
\mathit{valueslist} = \\
\quad \mathit{onearg}\,(\lambda\epsilon\kappa\,.\,\epsilon \in \mathtt{E_p} \to \\
\qquad\qquad\quad \mathit{cdr}\langle\epsilon\rangle \\
\qquad\qquad\qquad (\lambda\epsilon^*\,.\,\mathit{valueslist} \\
\qquad\qquad\qquad\qquad \epsilon^* \\
\qquad\qquad\qquad\qquad (\lambda\epsilon^*\,.\,\mathit{car}\langle\epsilon\rangle(\mathit{single}(\lambda\epsilon\,.\,\kappa(\langle\epsilon\rangle \S \epsilon^*))))), \\
\qquad\qquad \epsilon = \mathit{null} \to \kappa\langle\,\rangle, \\
\qquad\qquad\quad \mathit{wrong}\ \text{“non-list argument to }\mathtt{values\text{-}list}\text{”})
\end{array}
$$

$$
\begin{array}{l}
\mathit{cwcc} : \mathtt{E}^* \to \mathtt{K} \to \mathtt{C} \qquad [\mathtt{call\text{-}with\text{-}current\text{-}continuation}] \\
\mathit{cwcc} = \\
\quad \mathit{onearg}\,(\lambda\epsilon\kappa\,.\,\epsilon \in \mathtt{F} \to \\
\qquad\qquad (\lambda\sigma\,.\,\mathit{new}\,\sigma \in \mathtt{L} \to \\
\qquad\qquad\qquad \mathit{applicate}\,\epsilon \\
\qquad\qquad\qquad\qquad \langle\langle \mathit{new}\,\sigma \mid \mathtt{L}, \lambda\epsilon^*\kappa'\,.\,\kappa\epsilon^*\rangle \text{ in } \mathtt{E}\rangle \\
\qquad\qquad\qquad\qquad \kappa \\
\qquad\qquad\qquad\qquad (\mathit{update}\,(\mathit{new}\,\sigma \mid \mathtt{L}) \\
\qquad\qquad\qquad\qquad\qquad \mathit{unspecified} \\
\qquad\qquad\qquad\qquad\qquad \sigma), \\
\qquad\qquad\qquad \mathit{wrong}\ \text{“out of memory”}\ \sigma), \\
\qquad\qquad \mathit{wrong}\ \text{“bad procedure argument”})
\end{array}
$$

$$
\begin{array}{l}
\mathit{values} : \mathtt{E}^* \to \mathtt{K} \to \mathtt{C} \\
\mathit{values} = \lambda\epsilon^*\kappa\,.\,\kappa\epsilon^*
\end{array}
$$

$$
\begin{array}{l}
\mathit{cwv} : \mathtt{E}^* \to \mathtt{K} \to \mathtt{C} \qquad [\mathtt{call\text{-}with\text{-}values}] \\
\mathit{cwv} = \\
\quad \mathit{twoarg}\,(\lambda\epsilon_1\epsilon_2\kappa\,.\,\mathit{applicate}\,\epsilon_1\langle\,\rangle(\lambda\epsilon^*\,.\,\mathit{applicate}\,\epsilon_2\,\epsilon^*))
\end{array}
$$

## 7.3. Derived expression types

This section gives macro definitions for the derived expression types in terms of the primitive expression types (literal, variable, call, `lambda`, `if`, `set!`). See section 6.4 for a possible definition of `delay`.

```
(define-syntax cond
  (syntax-rules (else =>)
    ((cond (else result1 result2 ...))
     (begin result1 result2 ...))
    ((cond (test => result))
     (let ((temp test))
       (if temp (result temp))))
    ((cond (test => result) clause1 clause2 ...)
     (let ((temp test))
       (if temp
           (result temp)
           (cond clause1 clause2 ...))))
    ((cond (test)) test)
    ((cond (test) clause1 clause2 ...)
     (let ((temp test))
       (if temp
           temp
           (cond clause1 clause2 ...))))
    ((cond (test result1 result2 ...))
     (if test (begin result1 result2 ...)))
    ((cond (test result1 result2 ...)
           clause1 clause2 ...)
     (if test
         (begin result1 result2 ...)
         (cond clause1 clause2 ...)))))
```

```
(define-syntax case
  (syntax-rules (else)
    ((case (key ...)
       clauses ...)
     (let ((atom-key (key ...)))
       (case atom-key clauses ...)))
    ((case key
       (else result1 result2 ...))
     (begin result1 result2 ...))
    ((case key
       ((atoms ...) result1 result2 ...))
     (if (memv key '(atoms ...))
         (begin result1 result2 ...)))
    ((case key
       ((atoms ...) result1 result2 ...)
       clause clauses ...)
     (if (memv key '(atoms ...))
         (begin result1 result2 ...)
         (case key clause clauses ...)))))
```

```
(define-syntax and
  (syntax-rules ()
    ((and) #t)
    ((and test) test)
    ((and test1 test2 ...)
     (if test1 (and test2 ...) #f))))
```

```
(define-syntax or
  (syntax-rules ()
    ((or) #f)
    ((or test) test)
    ((or test1 test2 ...)
     (let ((x test1))
       (if x x (or test2 ...))))))
```

```
(define-syntax let
  (syntax-rules ()
    ((let ((name val) ...) body1 body2 ...)
     ((lambda (name ...) body1 body2 ...)
      val ...))
    ((let tag ((name val) ...) body1 body2 ...)
     ((letrec ((tag (lambda (name ...)
                      body1 body2 ...)))
        tag)
```

```
         val ...))))

(define-syntax let*
  (syntax-rules ()
    ((let* () body1 body2 ...)
     (let () body1 body2 ...))
    ((let* ((name1 val1) (name2 val2) ...)
       body1 body2 ...)
     (let ((name1 val1))
       (let* ((name2 val2) ...)
         body1 body2 ...)))))
```

The following `letrec` macro uses the symbol `<undefined>` in place of an expression which returns something that when stored in a location makes it an error to try to obtain the value stored in the location (no such expression is defined in Scheme). A trick is used to generate the temporary names needed to avoid specifying the order in which the values are evaluated. This could also be accomplished by using an auxiliary macro.

```
(define-syntax letrec
  (syntax-rules ()
    ((letrec ((var1 init1) ...) body ...)
     (letrec "generate_temp_names"
       (var1 ...)
       ()
       ((var1 init1) ...)
       body ...))
    ((letrec "generate_temp_names"
       ()
       (temp1 ...)
       ((var1 init1) ...)
       body ...)
     (let ((var1 <undefined>) ...)
       (let ((temp1 init1) ...)
         (set! var1 temp1)
         ...
         body ...)))
    ((letrec "generate_temp_names"
       (x y ...)
       (temp ...)
       ((var1 init1) ...)
       body ...)
     (letrec "generate_temp_names"
       (y ...)
       (newtemp temp ...)
       ((var1 init1) ...)
       body ...))))

(define-syntax begin
  (syntax-rules ()
    ((begin exp ...)
     ((lambda () exp ...)))))
```

The following alternative expansion for `begin` does not make use of the ability to write more than one expression in the body of a lambda expression. In any case, note that these rules apply only if the body of the `begin` contains no definitions.

```
(define-syntax begin
  (syntax-rules ()
    ((begin exp)
     exp)
    ((begin exp1 exp2 ...)
     (let ((x exp1))
       (begin exp2 ...)))))
```

The following definition of `do` uses a trick to expand the variable clauses. As with `letrec` above, an auxiliary macro would also work. The expression `(if #f #f)` is used to obtain an unspecific value.

```
(define-syntax do
  (syntax-rules ()
    ((do ((var init step ...) ...)
         (test expr ...)
         command ...)
     (letrec
       ((loop
         (lambda (var ...)
           (if test
               (begin
                 (if #f #f)
                 expr ...)
               (begin
                 command
                 ...
                 (loop (do "step" var step ...)
                       ...))))))
       (loop init ...)))
    ((do "step" x)
     x)
    ((do "step" x y)
     y)))
```

# NOTES

### Language changes

This section enumerates the changes that have been made to Scheme since the "Revised[4] report" [6] was published.

- The report is now a superset of the IEEE standard for Scheme [13]: implementations that conform to the report will also conform to the standard. This required the following changes:
  - The empty list is now required to count as true.
  - The classification of features as essential or inessential has been removed. There are now three classes of built-in procedures: primitive, library, and optional. The optional procedures are `load`, `with-input-from-file`, `with-output-to-file`, `transcript-on`, `transcript-off`, and `interaction-environment`, and `-` and `/` with more than two arguments. None of these are in the IEEE standard.
  - Programs are allowed to redefine built-in procedures. Doing so will not change the behavior of other built-in procedures.
- *Port* has been added to the list of disjoint types.
- The macro appendix has been removed. High-level macros are now part of the main body of the report. The rewrite rules for derived expressions have been replaced with macro definitions. There are no reserved identifiers.
- `Syntax-rules` now allows vector patterns.
- Multiple-value returns, `eval`, and `dynamic-wind` have been added.
- The calls that are required to be implemented in a properly tail-recursive fashion are defined explicitly.
- '`@`' can be used within identifiers. '`|`' is reserved for possible future extensions.

# ADDITIONAL MATERIAL

The Internet Scheme Repository at

`http://www.cs.indiana.edu/scheme-repository/`

contains an extensive Scheme bibliography, as well as papers, programs, implementations, and other material related to Scheme.

# EXAMPLE

`Integrate-system` integrates the system

$$y'_k = f_k(y_1, y_2, \ldots, y_n),\ k = 1, \ldots, n$$

of differential equations with the method of Runge-Kutta.

The parameter `system-derivative` is a function that takes a system state (a vector of values for the state variables $y_1, \ldots, y_n$) and produces a system derivative (the values $y'_1, \ldots, y'_n$). The parameter `initial-state` provides an initial system state, and `h` is an initial guess for the length of the integration step.

The value returned by `integrate-system` is an infinite stream of system states.

```
(define integrate-system
  (lambda (system-derivative initial-state h)
    (let ((next (runge-kutta-4 system-derivative h)))
      (letrec ((states
                (cons initial-state
                      (delay (map-streams next
                                          states)))))
        states))))
```

`Runge-Kutta-4` takes a function, `f`, that produces a system derivative from a system state. `Runge-Kutta-4` produces a function that takes a system state and produces a new system state.

```
(define runge-kutta-4
  (lambda (f h)
    (let ((*h (scale-vector h))
          (*2 (scale-vector 2))
          (*1/2 (scale-vector (/ 1 2)))
          (*1/6 (scale-vector (/ 1 6))))
      (lambda (y)
        ;; y is a system state
        (let* ((k0 (*h (f y)))
               (k1 (*h (f (add-vectors y (*1/2 k0)))))
               (k2 (*h (f (add-vectors y (*1/2 k1)))))
               (k3 (*h (f (add-vectors y k2)))))
          (add-vectors y
            (*1/6 (add-vectors k0
                               (*2 k1)
                               (*2 k2)
                               k3))))))))

(define elementwise
  (lambda (f)
    (lambda vectors
      (generate-vector
        (vector-length (car vectors))
        (lambda (i)
          (apply f
                 (map (lambda (v) (vector-ref  v i))
                      vectors)))))))

(define generate-vector
  (lambda (size proc)
```

```
    (let ((ans (make-vector size)))
      (letrec ((loop
                (lambda (i)
                  (cond ((= i size) ans)
                        (else
                         (vector-set! ans i (proc i))
                         (loop (+ i 1)))))))
        (loop 0)))))

(define add-vectors (elementwise +))

(define scale-vector
  (lambda (s)
    (elementwise (lambda (x) (* x s)))))
```

`Map-streams` is analogous to `map`: it applies its first argument (a procedure) to all the elements of its second argument (a stream).

```
(define map-streams
  (lambda (f s)
    (cons (f (head s))
          (delay (map-streams f (tail s))))))
```

Infinite streams are implemented as pairs whose car holds the first element of the stream and whose cdr holds a promise to deliver the rest of the stream.

```
(define head car)
(define tail
  (lambda (stream) (force (cdr stream))))
```

The following illustrates the use of `integrate-system` in integrating the system

$$C\frac{dv_C}{dt} = -i_L - \frac{v_C}{R}$$

$$L\frac{di_L}{dt} = v_C$$

which models a damped oscillator.

```
(define damped-oscillator
  (lambda (R L C)
    (lambda (state)
      (let ((Vc (vector-ref state 0))
            (Il (vector-ref state 1)))
        (vector (- 0 (+ (/ Vc (* R C)) (/ Il C)))
                (/ Vc L))))))

(define the-states
  (integrate-system
     (damped-oscillator 10000 1000 .001)
     '#(1 0)
     .01))
```

# REFERENCES


[1] Harold Abelson and Gerald Jay Sussman with Julie Sussman. *Structure and Interpretation of Computer Programs, second edition.* MIT Press, Cambridge, 1996.

[2] Alan Bawden and Jonathan Rees. Syntactic closures. In *Proceedings of the 1988 ACM Symposium on Lisp and Functional Programming*, pages 86–95.

[3] Robert G. Burger and R. Kent Dybvig. Printing floating-point numbers quickly and accurately. In *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*, pages 108–116.

[4] William Clinger, editor. The revised revised report on Scheme, or an uncommon Lisp. MIT Artificial Intelligence Memo 848, August 1985. Also published as Computer Science Department Technical Report 174, Indiana University, June 1985.

[5] William Clinger. How to read floating point numbers accurately. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 92–101. Proceedings published as *SIGPLAN Notices* 25(6), June 1990.

[6] William Clinger and Jonathan Rees, editors. The revised[4] report on the algorithmic language Scheme. In *ACM Lisp Pointers* 4(3), pages 1–55, 1991.

[7] William Clinger and Jonathan Rees. Macros that work. In *Proceedings of the 1991 ACM Conference on Principles of Programming Languages*, pages 155–162.

[8] William Clinger. Proper Tail Recursion and Space Efficiency. To appear in *Proceedings of the 1998 ACM Conference on Programming Language Design and Implementation*, June 1998.

[9] R. Kent Dybvig, Robert Hieb, and Carl Bruggeman. Syntactic abstraction in Scheme. *Lisp and Symbolic Computation* 5(4):295–326, 1993.

[10] Carol Fessenden, William Clinger, Daniel P. Friedman, and Christopher Haynes. Scheme 311 version 4 reference manual. Indiana University Computer Science Technical Report 137, February 1983. Superseded by [11].

[11] D. Friedman, C. Haynes, E. Kohlbecker, and M. Wand. Scheme 84 interim reference manual. Indiana University Computer Science Technical Report 153, January 1985.

[12] *IEEE Standard 754-1985. IEEE Standard for Binary Floating-Point Arithmetic.* IEEE, New York, 1985.

[13] *IEEE Standard 1178-1990. IEEE Standard for the Scheme Programming Language.* IEEE, New York, 1991.

[14] Eugene E. Kohlbecker Jr. *Syntactic Extensions in the Programming Language Lisp.* PhD thesis, Indiana University, August 1986.

[15] Eugene E. Kohlbecker Jr., Daniel P. Friedman, Matthias Felleisen, and Bruce Duba. Hygienic macro expansion. In *Proceedings of the 1986 ACM Conference on Lisp and Functional Programming*, pages 151–161.

[16] Peter Landin. A correspondence between Algol 60 and Church's lambda notation: Part I. *Communications of the ACM* 8(2):89–101, February 1965.

[17] MIT Department of Electrical Engineering and Computer Science. Scheme manual, seventh edition. September 1984.

[18] Peter Naur et al. Revised report on the algorithmic language Algol 60. *Communications of the ACM* 6(1):1–17, January 1963.

[19] Paul Penfield, Jr. Principal values and branch cuts in complex APL. In *APL '81 Conference Proceedings,* pages 248–256. ACM SIGAPL, San Francisco, September 1981. Proceedings published as *APL Quote Quad* 12(1), ACM, September 1981.

[20] Kent M. Pitman. The revised MacLisp manual (Saturday evening edition). MIT Laboratory for Computer Science Technical Report 295, May 1983.

[21] Jonathan A. Rees and Norman I. Adams IV. T: A dialect of Lisp or, lambda: The ultimate software tool. In *Conference Record of the 1982 ACM Symposium on Lisp and Functional Programming*, pages 114–122.

[22] Jonathan A. Rees, Norman I. Adams IV, and James R. Meehan. The T manual, fourth edition. Yale University Computer Science Department, January 1984.

[23] Jonathan Rees and William Clinger, editors. The revised$^3$ report on the algorithmic language Scheme. In *ACM SIGPLAN Notices* 21(12), pages 37–79, December 1986.

[24] John Reynolds. Definitional interpreters for higher order programming languages. In *ACM Conference Proceedings*, pages 717–740. ACM, 1972.

[25] Guy Lewis Steele Jr. and Gerald Jay Sussman. The revised report on Scheme, a dialect of Lisp. MIT Artificial Intelligence Memo 452, January 1978.

[26] Guy Lewis Steele Jr. Rabbit: a compiler for Scheme. MIT Artificial Intelligence Laboratory Technical Report 474, May 1978.

[27] Guy Lewis Steele Jr. *Common Lisp: The Language, second edition.* Digital Press, Burlington MA, 1990.

[28] Gerald Jay Sussman and Guy Lewis Steele Jr. Scheme: an interpreter for extended lambda calculus. MIT Artificial Intelligence Memo 349, December 1975.

[29] Joseph E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* MIT Press, Cambridge, 1977.

[30] Texas Instruments, Inc. TI Scheme Language Reference Manual. Preliminary version 1.0, November 1985.

# ALPHABETIC INDEX OF DEFINITIONS OF CONCEPTS, KEYWORDS, AND PROCEDURES

The principal entry for each term, procedure, or keyword is listed first, separated from the other entries by a semicolon.