IEEE Std 1178™-1990 (R2008)

IEEE Standard for the Scheme Programming Language

Sponsor

Microprocessor and Microcomputer Standards Subcommittee of the IEEE Computer Society

Approved December 10, 1990

Reaffirmed 27 March 2008

IEEE Standards Board

Abstract: The form and meaning of programs written in the Scheme programming language in particular, their syntax, the semantic rules for interpreting them, and the representation of data to be input or output by them, are specified. The fundamental ideas of the language and the notational conventions used for describing and writing programs in the language are presented. The syntax and semantics of expressions, programs, and definitions are specified. Scheme's built-in procedures, which include all of the language's data manipulation and input/output primitives, are described, and a formal syntax for Scheme written in extended Backus-Naur form is provided. A formal denotational semantics for Schemes and some issues in the implementation of Scheme's arithmetic are covered in the appendixes.

Keywords: Lisp, Scheme, Scheme programming language

ISBN 1-55937-125-0

Foreword

(This Foreword is not a part of IEEE Std 1178-1990, IEEE Standard for the Scheme Programming Language.)

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. The Scheme programming language demonstrates that a very small number of rules for forming expressions, with few 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 places few restrictions on the use of procedural abstractions: procedures are full first-class objects. Although Scheme is a block-structured language, and it permits side effects, it differs from most imperative block-structured languages by encouraging a functional style of programming that uses procedures to encapsulate state.

In a similar spirit, Scheme implementations impose no storage penalty for tail-recursive procedure calls, and continuations (which are present, although behind the scenes, in all programming languages) are first-lass Scheme objects that act like procedures. This permits nearly all known sequential control structures to be expressed in terms of procedure calls.

Purpose of This Standard

Throughout its thirty-year life, the Lisp family of languages has continually evolved to encompass changing ideas about programming-language design. Scheme has participated in the evolution of Lisp. Scheme was one of the first programming languages to incorporate first-lass 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. Scheme was the first widely used programming language to rely entirely on procedure calls to express iteration and to embrace first-class escape procedures.

Specifying a standard for Scheme is intended to encourage the continued evolution of Lisp dialects by identifying a coherent set of constructs that is large enough to support the implementation of substantial programs, but also small enough to admit significant extensions and alternate approaches to language design. For example, this standard does not mandate the inclusion in Scheme of large run-time libraries, particular user interfaces, or complex interactions with external operating systems, although practical Scheme implementations ordinarily provide such features.

In particular, there are important linguistic design issues that are not discussed in this standard *precisely because* Scheme has sparked fruitful new approaches in these areas, and this working group does not wish to discourage the further development of good ideas by taking a position on issues under active investigation. Some of these issues are macros, packaging, and object-oriented programming.

The working group hopes that future revisions of this standard will be sensitive to the fact that good ideas need time to mature, and that exploration can often be stifled by the premature adoption of standards.

Background

The first description of Scheme was written in 1975 [B20].[1] A revised report [B17] appeared in 1978, which described the evolution of the language as its MIT implementation was upgraded to support an innovative compiler 4. Three distinct projects began in 1981 and 1982 to use variants of Scheme for courses at MIT, Yale, and Indiana University [B14], [B12], [B6]. An introductory computer science textbook using Scheme was published in 1984 [B2].

[1]See footnote 1, page 7.

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 [B4] was published at MIT and Indiana University in the summer of 1985. Subsequent rounds of revision took place in the spring of 1986 [B15], and at a meeting that preceded the 1988 ACM Conference on Lisp and Functional Programming [B5]. The working group for this standard first met at that same conference; the standard draws heavily on the earlier reports.

Readers interested in the evolution of the Scheme language are referred to the above documents, and in particular to the “Notes” following chapter 7 in [B15] and [B5].

Members of the Scheme Working Group of the Microprocessor and Microcomputer Standards Subcommittee and those who participated by correspondence were as follows:

Christopher T. Haynes, *Chair*
David H. Bartley, Chris Hanson, James S. Miller *(Editors)*

Harold Abelson
Norman Adams
Cyril N. Alberga
Joel F. Bartlett
Scott Burson
Clyde Camp
Bill Campbell
Jerome Chailloux
Stewart Clamen
William Clinger
Pavel Curtis
Jeffrey Dalton
Olivier Danvy
Klaus Dassler
Kenneth Dickey
Bruce Duba

R. Kent Dybvig
Marc Feeley
Daniel P. Friedman
Mark Friedman
Richard P. Gabriel
John Gateley
Arthur Gleckler
Patrick Greussay
Jed Harris
Robert Hieb
Takayasu Ito
Roger Kirchner
Paul Kristoff
Tim McNerney
William Maddox
Sidney Marshall
Robert Mathis

Richard Mlynarik
Andy Norman
Eric Ost
John D. Ramsdell
Jonathan Rees
Guillermo J. Rozas
Benjamin Schreiber
George Springer
Guy L. Steele Jr.
Gerald Jay Sussman
Eric Tiedemann
Queyen Tran
R. L. Tritchard
Mitchell Wand
Jon L. White
Taiichi Yuasa

The following persons were on the balloting committee that approved this document for submission to the IEEE Standards Board:

Harold Abelson
William B. Adams
Mohammad Al-Malki
Duane L. Anderson
Joel F. Bartlett
David H. Bartley
Winsor A. Brown
Lyle Burnett
Todd Busniewski
Carl Cagan
Michael J. Caldwell
George Carson
Donald Chi
David Cohen
Paul D. Cook
J. R. Davis
Kenneth Alan Dickey
Su Dongzhuang
Kent R. Dybvig
Jeff S. Ebert

Neal M. Gafter
John Gateley
Tayeb Giuma
Willard Graves
Gregory Guthrie
Christopher T. Haynes
Herbert Hecht
Robert H. Hyerle
John Jensen
Richard Karcich
Evan Kirshenbaum
Peter M. Kogge
Ezzat Korany
Paul Kristoff
Takahiko Kuki
Tuvia Lamdan
James S. Miller
John E. Montague
Keith Morgan

Cuong Nguyen
Peter Olsen
James R. Otto
Don Oxley
Chandresh J. Patel
Crispin Perdue
Robert Pettengill, Jr.
John D. Ramsdell
Hans Roosli
Norman Schneidewind
David Seraphin
Michael Smolin
Serge Szukalow
Mitchell Wand
Carl Warren
W. L. Whipple
Fritz Whittington
Andrew Wilson
Qiufeng Wu
Oren Yuen

The final conditions for approval of this standard were met on December 10, 1990. This standard was conditionally approved by the IEEE Standards Board on December 6, 1990, with the following membership:

This page is intentionally blank.

IEEE Standard for the Scheme Programming Language

Introduction

Objectives

This standard specifies the form and meaning of programs written in the Scheme programming language.

Scheme was initially conceived as a vehicle for programming language research founded on a few key concepts that together critically distinguish it from others in the related Algol and Lisp families of languages. Through several independent implementations, Scheme has evolved to include a commonly accepted base set of data types and procedures as well as other features that are considered experimental or controversial.

The principal objective of this standard is to specify a language containing exactly those features for which there has been extensive experience and thus a conservative consensus for standardization. This provides users with a basis for constructing programs that are portable across implementations without jeopardizing research and future evolution of the language.

Scope

This standard specifies the representation of Scheme programs, their syntax, the semantic rules for interpreting them, and the representation of data to be input or output by them.

This standard does not specify the mechanisms by which Scheme programs are transferred to and from system memory and placed into execution, or the size or complexity of a program and its data that will exceed the capacity of a particular implementation.

Future Directions

Future revisions to this standard may be anticipated by a series of informal language specifications collectively known as the “Revised Reports on the Algorithmic Language Scheme” (see [B4],[1] [B15], [B5]). These reports extend the standard language to include features that are considered mature enough to merit widespread implementation and experimentation. Addition of such a feature to the report suggests that Scheme implementors should examine the addition and carefully consider it as a candidate for formal standardization.

Compliance

A *conforming program* shall use only those features of the language that are specified in this standard. It shall not contain dependencies on any unspecified or undefined aspect of this specification.

A *conforming implementation* shall accept any conforming program and execute it as specified in this standard. Conforming implementations may have extensions, provided they do not alter the behavior of any conforming program.

Organization of the Document

This document is divided into the following major sections:

1) this introduction, which includes definitions and references;
2) three chapters that present the fundamental ideas of the language and describe the notational conventions used for describing the language and writing programs in the language;
3) two chapters that specify the syntax and semantics of expressions, programs, and definitions;
4) a chapter describing Scheme’s built-in procedures, which include all of the language’s data manipulation and input/output primitives;
5) a chapter providing a formal syntax for Scheme written in extended Backus-Naur form;
6) an appendix providing a formal denotational semantics for Scheme;
7) two appendixes outlining some issues in the implementation of Scheme’s arithmetic;
8) an index.

Definitions of Terms

In this standard, *shall* is to be interpreted as a requirement on an implementation or a program; conversely, *shall not* is to be interpreted as a prohibition.

The word *should* is to be interpreted as a strong recommendation in keeping with the intent of the standard; conversely, *should not* is to be interpreted as a strong discouragement.

The following terms are used in this document. Other terms are defined at their first appearance, indicated by *italic* type.

[1] The numbers in brackets correspond to those of the references on page 4. When preceded by B, they correspond to those of the bibliography on page 57.

applicative order. A property of a programming language or procedure: the arguments to a procedure call are evaluated before the procedure is invoked, and the result of each evaluation is passed to the procedure in place of its argument expression.

bignum. A multiple-precision computer representation for very large integers (*see also*: **fixnum**).

extent. A period of time, usually referring to the lifetime of an object. Once created, an object with *unlimited extent* exists forever.

first-class object. An object that can be the value of a variable or can be stored in a data structure. In Scheme, first-class objects have *unlimited extent.*

fixnum. A limited-precision computer representation for integers, where the limitation is imposed by machine-architecture constraints (*see also*: **bignum**).

flonum. A floating-point number.

procedure. A parameterized program fragment, called a *subroutine* or *function* in some programming languages.

recursive. Self-referential. In common usage, a *recursive procedure* is one that calls itself; similarly a *recursion* is a call by a procedure to itself. A set of procedures is *mutually recursive* if they refer to one another.

scope. The region of a program's source text that is associated with a linguistic construct. Normally used with “variable” to describe the region over which a variable is bound: “the *scope* of a variable.”

side effect. Loosely speaking, an expression has a side effect if it performs some observable action in addition to returning a value. For example, a variable assignment is a side effect.

tail recursive. A property of the implementation of a programming language. In a *tail-recursive* implementation, iterative processes can be expressed by means of procedure calls. (The process described by a program is iterative if and only if the order of its space growth is constant, aside from that used for the values of the program's variables.)

top-level environment. The environment used to resolve free variable references in a Scheme program.

Examples

This standard describes in detail the form and meaning of the features of the Scheme language. To assist the reader, the document provides examples to clarify the intent or operation of certain features. Because many aspects of the language are purposely left undefined, these examples often merely typify the results to be expected and should not be construed as implying further constraints on the language specification.

Base Document

This document is derived from “Revised^4 Report on the Algorithmic Language Scheme” [B5], edited by William Clinger and Jonathan Rees.

References

[1] IEEE Std 754-1985, IEEE Standard for Binary Floating-Point Arithmetic (ANSI).[2]

[2] Paul Hudak and Philip Wadler, editors. Report on the Functional Programming Language Haskell. Yale University Research Report YALEU/DCS/RR-666, Dec. 1988.

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

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

[5] Guy Lewis Steele Jr. *Common Lisp: The Language.* 2d ed. Bedford, MA: Digital Press, 1990.

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 Sections 3 through 6. For reference purposes, Appendix A 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; for this reason static scoping is also referred to as *lexical scoping*.

Scheme has latent as opposed to manifest types. Types are associated with values 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.

The values of Scheme expressions are called *objects*. This use of the term "object" predates, and should not be confused with, its use in object-oriented programming languages. 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 [4]. 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 tail-recursive implementation, iteration can be expressed using the ordinary procedure-call mechanics, so that special iteration constructs are useful only as abbreviations.

[2]IEEE publications are available from the Institute of Electrical and Electronics Engineers, Service Center, 445 Hoes Lane, P.O. Box 1331, Piscataway, NJ 08855-1331, USA.

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 6.9).

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 [2], 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.

The **`read`** procedure performs syntactic as well as lexical decomposition of the data it reads. The **`read`** procedure parses its input as data (see 7.2), not as program.

The formal syntax of Scheme is described in Section 7.

1.3 Notation and Terminology

1.3.1 Error Situations and Unspecified Behavior

When speaking of an error situation, this standard uses the phrase “an error is signalled” to indicate that implementations shall 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 standard. Implementations may extend a procedure's domain of definition to include such arguments.

This standard 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 should 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 shall evaluate to some object without signalling an error, but the value depends on the implementation; this standard explicitly does not say what value should be returned.

1.3.2 Entry Format

Sections 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
```

If *category* is "syntax," then the entry describes an expression type, and the header line 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

⟨thing$_1$⟩ …

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

⟨thing$_1$⟩ ⟨thing$_2$⟩ …

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).

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 3.4, 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 6.3)
z, *z*$_1$, *z*$_j$, …	complex number
x, *x*$_1$, … *x*$_j$, …	real number
y, *y*$_1$, … *y*$_j$, …	real number
q, *q*$_1$, … *q*$_j$, …	rational number
n, *n*$_1$, … *n*$_j$, …	integer
k, *k*$_1$, … *k*$_j$, …	exact non-negative integer

1.3.3 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 (see Section 6.), to an object that may be represented externally by the sequence of characters "**`{40`**". See 3.3 for a discussion of external representations of objects.

1.3.4 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 3.5) 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..

Upper-case 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. Implementations that extend the syntax of identifiers shall not introduce ambiguities with other parts of the grammar; in particular nothing that can be generated by ⟨number⟩ (see 7.1) shall be interpreted as an identifier.

A conforming implementation shall not place limits on the lengths of 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 7.1 for a formal syntax of identifiers.

Identifiers have several uses within Scheme programs:

- Certain identifiers are reserved for use as syntactic keywords (see below).
- Any identifier that is not a syntactic keyword may be used as a variable (see 3.1).
- When an identifier appears as a literal or within a literal (see 4.1.2), it is being used to denote a *symbol* (see 6.4). An identifier may be used to denote a symbol regardless of whether it is used as a syntactic keyword or a variable.

The following identifiers are syntactic keywords, and shall not be used as variables:

=>	**else**	**or**
and	**if**	**quasiquote**
begin	**lambda**	**quote**
case	**let**	**set !**
cond	**let***	**unquote**
define	**letrec**	**unquote-splicing**
do		

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 6.5.

`+ - .`
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 (see 6.3), and to indicate a rest-parameter in a formal parameter list (see 4.1.4). A delimited sequence of three successive periods is also an identifier.

`( )`
Parentheses are used for grouping and to notate lists (see 6.3).

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

`` ` ``
The backquote character is used to indicate almost-constant data (see 4.2.5).

`, ,@`
The character comma and the sequence comma at-sign are used in conjunction with backquote (see 4.2.5).

`"`
The double quote character is used to delimit strings (see 6.7).

`\`
Backslash is used in the syntax for character constants (see 6.6) and as an escape character within string constants (see 6.7).

`[ ] { }`
Left and right square brackets and curly braces 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 (see 6.1).

`#\`
This introduces a character constant (see 6.6).

`#(`
This introduces a vector constant (see 6.8). Vector constants are terminated by **`)`** .

`#e #i #b #o #d #x`
These are used in the notation for numbers (see 6.5.4).

3. Basic Concepts

3.1 Variables and Regions

Any identifier that is not a syntactic keyword (see 2.1) may be used as a variable. A variable may name a location where a value can be stored. A variable that does so is said to be *bound* to the location. The 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 locations and to bind variables to those locations. The most fundamental of these *binding constructs* is the lambda expression, because all other binding constructs can be explained in terms of lambda expressions. The other binding constructs are **`let`**, **`let*`**, **`letrec`**, and **`do`** expressions (see 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. To each place where a variable is bound in a program there corresponds a *region* of the program text within which the binding is effective. 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. Two distinct regions are either disjoint, or one is nested inside the other. These regions are sometimes called *blocks*, and this nesting property is called *block structure*.

Every reference to or assignment of a variable refers to the binding of the variable that established the innermost of the regions containing the use. If there is no binding of the variable whose region contains the use, then the use refers to the binding for the variable in the top-level environment, if any (see Section 6.); if there is no binding for the identifier, it is said to be *unbound*.

3.2 True and False

Any Scheme value can be used as a boolean value for the purpose of a conditional test. As explained in 6.1, all values count as true in such a test except for **`#f`**. This standard 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 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 6.3).

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

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

External representations can also be used for input and output. The procedure **`read`** (see 6.10.2) parses external representations, and the procedure **`write`** (see 6.10.3) 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 subsections of Section 6.

3.4 Disjointness of Types

Every object satisfies at most one of the following predicates:

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

These predicates define the types *boolean*, *pair*, *empty list*, *symbol*, *number*, *char* (or *character*), *string*, *vector*, and *procedure*.

3.5 Storage Model

This section describes a model that can be used to understand Scheme's use of storage. A conforming implementation may be built using other storage models. However, the behavior of a conforming program in any conforming implementation shall be indistinguishable from the behavior of that program in an implementation built using the model described here.

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?`** (see 6.2) 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 standard speaks of storage being allocated for a variable or object, what is meant is that an appropriate number of locations are chosen from the set of locations that are not in use, and the chosen locations are marked to indicate that they are now in use before the variable or object is made to denote them.

In many systems it is desirable for constants (i.e., the values of literal expressions) to reside in read-only memory. To express this, it is convenient to imagine that every object that denotes locations is associated with a flag telling whether that object is mutable or immutable. The constants and the strings returned by **`symbol->string`** are then the immutable objects, while all objects created by the other procedures listed in this standard are mutable. It is an error to attempt to store a new value into a location that is denoted by an immutable object.

4. Expressions

A Scheme expression is a construct that returns a value, such as a variable reference, literal, procedure call, or conditional.

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 explained in terms of the primitive constructs as in 7.6. They are redundant in the strict sense of the word, but they capture common patterns of usage, and are therefore provided as convenient abbreviations.

4.1 Primitive Expression Types

4.1.1 Variable References

```
⟨variable⟩                                        syntax
```

An expression consisting of a variable (see 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 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. The external syntax generated by **write** for two-element lists whose car is the symbol **quote** may vary between implementations.

```
   '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 3.5, 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⟩ ⟨operand₁⟩ …)                      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 4.1.4).

Procedure calls are also called *combinations*.

NOTES:

1 — 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.

2 — 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.

3 — 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 Lambda Expressions

(lambda ⟨formals⟩ ⟨body⟩) **syntax**

Syntax: ⟨Formals⟩ shall be a formal arguments list as described below, and ⟨body⟩ shall 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 of the last expression in the body will be returned as the result 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⟩ shall 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$_{n-1}$⟩ **.** ⟨variable$_n$⟩**)**: If a space-delimited period precedes the last variable, then 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.

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

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

Each procedure created as the result of evaluating a lambda expression is tagged with a storage location. Ordinarily, this storage location is newly allocated for each evaluation of a lambda expression. However, a storage location can be reused provided the implementation can show that all of the procedures tagged with the location behave identically (return the same value and have the same side effects) for any given arguments. See 6.2.

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 6.1), then ⟨consequent⟩ is evaluated and its value is returned. Otherwise ⟨alternate⟩ is evaluated and its value is 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)
    (/ 1 0))               ? 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⟩ shall be bound either in some region enclosing the **set!** expression or in the top-level environment. 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

For reference purposes, rewrite rules that convert constructs described in this section into the primitive constructs described in the previous section are provided; see 7.6.

4.2.1 Conditionals

(cond ⟨clause$_1$)⟩ ⟨clause$_2$⟩ …) **syntax**

Syntax: Each ⟨clause⟩ shall be of the form

(⟨test⟩ ⟨expression⟩ …)

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

(⟨test⟩ => ⟨expression⟩)

where again, ⟨test⟩ is any expression. The last ⟨clause⟩ may be an “else clause,” which has the form

(else ⟨expression$_1$⟩ ⟨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 6.1). When a ⟨test⟩ evaluates to a true value, then the remaining ⟨expression⟩s in its ⟨clause⟩ are evaluated in order, and the result of the last ⟨expression⟩ in the ⟨clause⟩ is returned as the result 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 ⟨clause⟩ uses the => alternate form, then ⟨expression⟩ shall evaluate to a procedure, which is invoked with the value of ⟨test⟩ as its single argument; the value returned by this procedure is returned as the value of the entire 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 of the last one is returned.

```
(cond ((> 3 2) 'win)
      ((/ 1 0) 'lose))      ⇒  win
(cond ((> 3 3) 'greater)
      ((< 3 3) 'less)
      (else 'equal))        ⇒  equal
(cond ((aasv 'b '((a 1) (b 2))) => cadr)
      (else #f))            ⇒  2
```

(case ⟨key⟩ ⟨clause$_1$⟩⟨clause$_2$⟩ …) **syntax**

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

((⟨datum$_1$⟩ …) ⟨expression$_1$⟩ ⟨expression$_2$⟩ …),

where each ⟨datum⟩ is an external representation of some object. The last ⟨clause⟩ may be an “else clause,” which has the form

(else ⟨expression$_1$⟩ ⟨expression$_2$⟩ …).

Semantics: A **case** expression is evaluated as follows. ⟨Key⟩ is evaluated and then each successive ⟨clause⟩ is processed in order. If the result of evaluating ⟨key⟩ is equivalent (in the sense of **eqv?**; see 6.2) to any ⟨datum⟩ in the ⟨clause⟩, then the expressions in that ⟨clause⟩ are evaluated from left to right and the result of the last expression in the ⟨clause⟩ is returned as the result of the **case** expression. If no ⟨datum⟩ in any ⟨clause⟩ is equivalent to the result of evaluating ⟨key⟩, then if there is an else clause its expressions are evaluated and the result of the last is the result 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 ⟨test$_1$⟩ …) **syntax**

The ⟨test⟩ expressions are evaluated from left to right, and the value of the first expression that evaluates to a false value (see 6.1) 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 ⟨test$_1$⟩ …) **syntax**

The ⟨test⟩ expressions are evaluated from left to right, and the value of the first expression that evaluates to a true value (see 6.1) is returned. Any remaining expressions are not evaluated. If all the 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 similar, 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⟩) **syntax**

Syntax: ⟨Bindings⟩ shall have the form

((⟨variable$_1$⟩ ⟨init$_1$⟩) …),

where each ⟨init⟩ is an expression, and ⟨body⟩ shall 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 expressions in ⟨body⟩ are evaluated sequentially in the extended environment, and the value of the last expression of ⟨body⟩ is 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 4.2.4, named **let**.

(let* ⟨bindings⟩ ⟨body⟩) syntax

Syntax: ⟨Bindings⟩ shall have the form

((⟨variable$_1$⟩ ⟨init$_1$⟩) …),

and ⟨body⟩ shall 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⟩) syntax

Syntax: ⟨Bindings⟩ shall have the form

((⟨variable$_1$⟩ ⟨init$_1$⟩) …),

and ⟨body⟩ shall 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 expressions in ⟨body⟩ are evaluated sequentially in the extended environment, and the value of the last expression in ⟨body⟩ is 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 shall 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$_2$⟩ …) syntax

The ⟨expression⟩s are evaluated sequentially from left to right, and the value of the last ⟨expression⟩ is returned. This expression type is used to sequence side effects such as input and output. (See also 5.2.)

```
     (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 ((⟨variable1⟩ ⟨init1⟩ ⟨step1⟩)                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 with a specified result value.

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

Each iteration begins by evaluating ⟨test⟩; if the result is false (see 6.1), 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 of the last ⟨expression⟩ is returned as the value of the do expression. If no ⟨expression⟩s are present, then the value of the **do** expression is unspecified.

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⟩) **syntax**

This is a variant on the syntax of **let** called “named **let**,” which provides a more general looping construct than do, and may also be used to express recursions.

Named **let** has the same syntax and semantics as ordinary **let** except that ⟨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))
            ((negative? (car numbers))
             (loop (cdr numbers)
                   nonneg
                   (cons (car numbers) neg)))
            (else
             (loop (cdr numbers)
                   (cons (car numbers) nonneg)
                   neg))))
              ⇒  ((6 1 3) (-5 -2))
```

4.2.5 Quasiquotation

(quasiquote ⟨template⟩) **syntax**
\`⟨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 ⟨template⟩, the result of evaluating \` ⟨template⟩ is equivalent (in the sense of **equal?**; see 6.2) to the result of evaluating ’ ⟨template⟩. If a comma appears within the ⟨template⟩, however, the expression following the comma is evaluated (“unquoted”) and its result is inserted into the structure instead of the comma and the expression. If a comma appears followed immediately by an at-sign (@), then the following expression shall evaluate to a list; the opening and closing parentheses of the list are then “stripped away” and the elements of the list are inserted in place of the comma at-sign expression sequence.

```
     `(list ,(+ 1 2) 4)      ⇒  (list 3 4)
     (let ((name 'a)) `(list ,name ',name))
               ⇒  (list a (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)
`,(+ 2 3)                ⇒  5
```

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

```
`(a `(b ,(+ 1 2) ,(foo ,(+ 1 3) d) e) f)
         ⇒  (a `(b ,(+ 1 2) ,(foo 4 d) e) f)
(let ((name1 'x)
      (name2 'y))
  `(a `(b ,,name1 ,',name2 d) e))
         ⇒  (a `(b ,x ,'y d) e)
```

The notations `` ` ``⟨template⟩ and **`(quasiquote`** ⟨template⟩**`)`** are identical in all respects. **`,`**⟨expression⟩ is identical to **`(unquote`** ⟨expression⟩**`)`**, and **`,@`**⟨expression⟩ is identical to **(unquote-splicing** ⟨expression⟩**`)`**. The external syntax generated by **`write`** for two-element lists whose car is one of these symbols (**`quasiquote`**, **`unquote`**, or **`unquote-splicing`**) 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 ⟨template⟩ otherwise than as described above.

5. Program Structure

5.1 Programs

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

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 standard. (Indeed, Scheme would still be useful as a notation for expressing computational methods even in the absence of a mechanical implementation.)

Definitions occurring at the top level of a program can be interpreted declaratively. They cause bindings to be created in the top-level environment. 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.

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 shall have one of the following forms:

- **(define** ⟨variable⟩ ⟨expression⟩**)**
- **(define (**⟨variable⟩ ⟨formals⟩**)** ⟨body⟩**)**
  ⟨Formals⟩ shall 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⟩ shall be a single variable. This form is equivalent to
  ```
  (define ⟨variable⟩
   (lambda ⟨φορμαλ⟩ ⟨body⟩)).
  ```
- **(begin** ⟨definition$_1$⟩ …**)**

This form is equivalent to the set of definitions that form the body of the **begin**. See also 4.2.3.

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 (but see 6.). If ⟨variable⟩ is not bound, 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
```

It is permissible for an implementation to use an initial environment in which all possible variables are bound to locations, most of which contain undefined values.

5.2.2 Internal Definitions

Definitions are also permitted to occur at the beginning of a ⟨body⟩ (that is, the body of a **lambda**, **let**, **let***, **letrec**, or **define** expression). 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 shall 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.

6. Standard Procedures

The *top-level environment* starts out with a number of variables bound to locations containing useful values, including all of the standard procedures described in this section. 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. This standard refers to the initial contents of the top-level environment as the *initial environment*.

A conforming 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 shall not modify the behavior of the procedures specified in this standard. Altering any other top-level binding has an unspecified effect on the behavior of the procedures specified in this standard; conforming programs shall not alter such bindings.

6.1 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. Of all the standard Scheme values, only **`#f`** counts as false in conditional expressions. All other 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 **`#f`** from the empty list, and both **`#f`** and the empty list from the symbol nil.

Boolean constants evaluate to themselves, so they don't need to be quoted in programs.

```
#t                      ⇒ #t
#f                      ⇒ #f
'#f                     ⇒ #f
```

`(not `*`obj`*`)` **`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*) **procedure**

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

```
(boolean? #f)           ⇒ #t
(boolean? 0)            ⇒ #f
(boolean? '())          ⇒ #f
```

6.2 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 as defined in this standard and
```
(string=? (symbol->string obj1)
          (symbol->string obj2))
                                ⇒  #t
```
- *obj*$_1$ and *obj*$_2$ are both numbers, are numerically equal (see 6.5, =), 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 (see 6.6).
- 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 (see 3.5).
- *obj*$_1$ and *obj*$_2$ are procedures whose location tags are equal (see 4.1.4).The **eqv?** procedure returns **#f** if
- *obj*$_1$ and *obj*$_2$ are of different types (see 3.4).
- 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 a different value 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?** shall be a boolean.

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

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

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

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

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

```
     (eqv? f g))
                                ⇒  #f
```

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

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.

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

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

(eq? ***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, 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***$_2$**)** **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
```

6.3 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 (see 3.4); 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.

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
```

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; for example,

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

is equivalent to

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

Within literal expressions and representations of objects read by the **`read`** procedure, the forms **`'`**⟨datum⟩, **`` ` ``**⟨datum⟩, **`,`**⟨datum⟩, and **`,@`**⟨datum⟩ denote two-element lists whose first elements are the symbols **`quote`**, **`quasiquote`**, **`unquote`**, and **`unquote-splicing`**, respectively. The second element in each case is ⟨datum⟩ This convention is supported so that arbitrary Scheme programs may be represented as lists. That is, according to Scheme's grammar, every ⟨expression⟩ is also a ⟨datum⟩ (see 7.2). Among other things, this permits the use of the read procedure to parse Scheme programs. See 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 ; constant list
```

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

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

```
(caar pair)                                  procedure
(cadr pair)                                  procedure
     .                                           .
     .                                           .
     .                                           .
(cdddar pair)                                procedure
(cddddr pair)                                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*) **procedure**

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

(list? *obj*) **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. This procedure must return an answer even for circular structures.

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

(list *obj* ...) **procedure**

Returns a newly allocated list of its arguments.

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

(list *obj* ...) **procedure**

Returns the length of *list*.

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

(append *list* ...) **procedure**

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

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

The resulting list is always newly allocated, except that it shares structure with the last *list* argument. The last argument may actually be any object; an improper list results if the last argument is not a proper list.

```
(append '(a b) '(c . d))   ⇒ (a b c . d)
(append '() 'a)            ⇒ a
```

(reverse *list*) **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-ref *list* *k*) **procedure**

Returns the *k*th element of *list*, using zero-origin indexing. The *valid indexes* of a list are the exact non-negative integers less than the length of the list. The first element of a list has index 0, the second has index 1, and so on.

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

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

These procedures return the first pair of list whose car is *obj*; the returned pair is always one from which *list* is composed. If *obj* does not occur in *list*, **#f** (N.B., 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*) procedure
(assv *obj alist*) procedure
(assoc obj alist) 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; the returned pair is always an *element* of *alist*, *not* one of the pairs from which *alist* is composed. If no pair in *alist* has *obj* as its car, **#f** (N.B., 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 frequently 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**.

6.4 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 2.1 and 7.1.

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.

`(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 (see 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 nonstandard 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`**.

If *string* is modified by side effect after being passed to **`symbol->string`**, the resulting symbol shall not be affected. For example:

```
(define x (string #\a #\b))
(define y (string->symbol x))
(string-set! x 0 #\c)
x                           ⇒ “cb”
(symbol->string y)          ⇒ “ab”
(eq? y (string->symbol “ab”))
         ⇒ #t
```

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
```

6.5 Numbers

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 standard 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*.

6.5.1 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.

6.5.2 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 6.5.3.

With the exception of **`inexact->exact`**, the operations described in this section shall 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.

6.5.3 Implementation Restrictions

Implementations of Scheme are not required to implement the whole tower of subtypes given in 6.5.1, but they shall 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 an effectively 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 shall 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 shall 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 shall return an exact integer result provided all their arguments are exact integers and the mathematically expected result is representable as an exact integer within the implementation:

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

Implementations should 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 standard 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 1.

In particular, implementations that use flonum representations shall follow these rules: A flonum result shall be represented with at least as much precision as is used to express any of the inexact arguments to that operation. Potentially inexact operations such as `sqrt`, when applied to exact arguments, should produce exact answers whenever possible (for example, the square root of an exact 4 should 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 shall be used; but if the result is represented in some other way, then the representation shall have at least as much precision as the most precise flonum format available.

A particular implementation need not implement all of the procedures defined in this section. For example, an implementation in which all numbers are real need not provide `make-rectangular`. However, all implementations shall provide the following procedures:

`*`	`exact?`	`number?`
`+`	`floor`	`odd?`
`-`	`gcd`	`positive?`
`<`	`inexact?`	`quotient`
`<=`	`integer?`	`rational?`
`=`	`lcm`	`real?`
`>`	`max`	`remainder`
`>=`	`min`	`round`
`abs`	`modulo`	`string->number`
`ceiling`	`negative?`	`truncate`
`complex?`	`number->string`	`zero?`
`even?`		

Although Scheme allows a variety of written notations for numbers, any particular implementation may support only some of them (see the NOTE associated with `string->number`). 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.

6.5.4 Syntax of numerical constants

The syntax of the written representations for numbers is described formally in 7.1.

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
```

6.5.5 Numerical Operations

The reader is referred to 1.3.2 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 predicate 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 in exact 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
```

NOTES:

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

2 — 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 noncomplex 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.

NOTES:

1 — The traditional implementations of these predicates in Lisp-like languages are not transitive.

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

(zero? ***z*****)** **procedure**
(positive? ***x*****)** **procedure**
(negative? ***x*****)** **procedure**
(odd? ***n*****)** **procedure**
(even? ***n*****)** **procedure**

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

(max x_1 x_2 **…)** **procedure**
(max x_1 x_2 **…)** **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 presented 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. With no arguments, + returns 0 and * returns 1.

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

(- *z*$_1$ *z*$_2$) **procedure**
(- *z*) **procedure**
(/ *z*$_1$ *z*$_2$) **procedure**
(/ *z*) **procedure**

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

```
(- 3 4)          ⇒  -1
(- 3)            ⇒  -3
(/ 3 4)          ⇒  3/4
(/ 3)            ⇒  1/3
```

(abs *x*) **procedure**

Abs returns the magnitude 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: For positive integers n_1 and n_2, if n_3 and n_4 are integers such that $n_1 = n_2 n_3 + n_4$ and $0 \leq n_4 < n_2$, then

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

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.

The value returned by **quotient** always has the sign of the product of its arguments. **Remainder** and **modulo** differ on negative arguments—the **remainder** is either zero or has the sign of the dividend, the **modulo** always has the sign of the divisor:

```
(modulo 13 4)          ⇒ 1
```

```
(remainder 13 4)          ⇒ 1

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

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

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

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

(gcd *n*$_1$ …) **procedure**
(lcm *n*$_1$ …) **procedure**

These procedures return the greatest common divisor or least common multiple of their arguments. The result is always non-negative. With no arguments, **gcd** returns **0** and **lcm** returns **1**.

```
(gcd 32 -36)              ⇒ 4
(gcd)                     ⇒ 0
(lcm 32 -36)              ⇒ 288
(1cm 32.0 -36)            ⇒ 288.0 ; inexact
(lcm)                     ⇒ 1
```

(numerator *q*) **procedure**
(denominator *q*) **procedure**

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

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

(floor *x*) **procedure**
(ceiling *x*) **procedure**
(truncate *x*) **procedure**
(round *x*) **procedure**

These procedures return integers. **Floor** returns the largest integer not larger than *x*. **Ceiling** returns the smallest integer not smaller than *x*. **Truncate** returns the integer closest to *x* whose absolute value is not larger than the absolute value of *x*. **Round** returns the closest integer to *x*, rounding to even when *x* is halfway between two integers.

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

NOTE — If the argument to one of these procedures is inexact, then the result will also be inexact. If an exact value is needed, the result should be passed to the **inexact->exact** procedure.

```
(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*) **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 $r_2 = p_2/q_2$ (both in lowest terms) and $|p_1| \le |p_2|$ and $|q_1| \le |q_2|$. 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. For nonzero real x, the value of log z is defined to be the one whose imaginary part lies in the range –π (exclusive) to π (inclusive). log 0 is undefined. The value of log z when z is complex is defined according to the formula

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

With log defined this way, the values of $\sin^{-1} z$, $\cos^{-1} z$, 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 5, which in turn cites 3; 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 z_1 z_2) **procedure**

Returns z_1 raised to the power z_2:

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

0^0 is defined to be equal to 1.

(make-rectangular x_1 x_2)	**procedure**
(make-polar x_3 x_4)	**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 and make-polar return *z*, real-part returns x_1, imag-part returns x_2, magnitude returns x_3, and angle returns x_4. In the case of angle, whose value is not uniquely determined by the preceding rule, the value returned will be the one in the range —π (exclusive) to π (inclusive).

Rationale: Magnitude is the same as abs for a real argument, but abs shall 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 6.5.3.

6.5.6 Numerical Input and Output

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

Radix must be an exact integer, either 2, 8, 10, or 16. If omitted, *radix* defaults to 10. The procedure **`number->string`** takes a number and a radix and returns as a string an external representation of the given number in the given radix such that

```
(let ((number number)
      (radix radix))
  (eqv? number
        (string->number (number->string number
                                        radix)
                        radix)))
```

is true. It is an error if no possible result makes this expression true.

If *number* is inexact, the radix is 10, and the above expression can be satisfied by a result that contains a decimal point, then the result contains a decimal point and is expressed using the minimum number of digits (exclusive of exponent and trailing zeroes) needed to make the above expression true; otherwise the format of the result is unspecified.

The result returned by **`number->string`** never contains an explicit radix prefix.

NOTE — The error case can occur only when *number* is not a complex number or is a complex number with a nonrational real or imaginary part.

Rationale: If *number* is an inexact number represented using flonums, and the radix is 10, then the above expression is normally satisfied by a result containing a decimal point. The unspecified case allows for infinities, NaNs, and non-flonum representations.

```
(string->number string)                      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 — An implementation may restrict the domain of **`string->number`** 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.

6.6 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
#\u          ; 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 ignorethe 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) ? #t`
- The lower-case characters are in order. For example,
  `(char·? #\a #\b) ? #t`
- The digits are in order. For example,
  `(char·? #\0 #\9) ? #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.

(char-ci=? *char1 char2*) procedure
(char-ci<? *char*$_1$ *char*$_2$) **procedure**
(char-ci>? *char*$_1$ *char*$_2$) **procedure**
(char-ci≤? *char*$_1$ *char*$_2$) **procedure**
(char-ci≥? *char*$_1$ *char*$_2$) **procedure**

These procedures are similar to **char=?** etc., but they treat upper-case and lower-case letters as the same. For example,

```
    (char-ci=? #\A #\a) ? #t
```

(char-alphabet icy *char*) **procedure**
(char-numeric? *char*) **procedure**
(char-whitespace? *char*) **procedure**
(char-upper-case? *char*) **procedure**
(char-lower-case? *char*) **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**, **char–>integer** returns that character. These procedures implement injective order isomorphisms between the set of characters under the **char<=?** ordering and some subset of the integers under the **<=** ordering. That is, if

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

and *x* and *y* are in the domain of **integer->char**, then

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

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

(char-upcase *char*) **procedure**
(char-downcase *char*) **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.

6.7 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. This standard does not specify the effect of a backslash within a string that is not followed by a doublequote or backslash.

A string may continue from one line to the next, but in such a situation the exact contents of the string are unspecified.

The *length* of a string is the number of characters that it contains. This number is a non-negative integer that is fixed when the string is created. The *valid indexes* of a string are the exact non-negative integers less than the length of the string. The first character of a string has index 0, the second has index 1, and so on.

Some of the procedures that operate on strings ignore the difference between upper case 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* …) **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 ; constant string
   (string-set! (symbol->string 'immutable)
                0
                #\?)          ⇒  error
```

(string=? *string1 string2*) **procedure**
(string-ci=? *string1 string2*) **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 case and lower case as distinct characters.

(string<? *string*$_1$ *string*$_2$) **procedure**
(string>? *string*$_1$ *string*$_2$) **procedure**
(string<=? *string*$_1$ *string*$_2$) **procedure**
(string>=? *string*$_1$ *string*$_2$) **procedure**
(string-ci<? *string*$_1$ *string*$_2$) **procedure**
(string-ci>? *string*$_1$ *string*$_2$) **procedure**
(string-ci<=? *string*$_1$ *string*$_2$) **procedure**
(string-ci>=? *string*$_1$ *string*$_2$) **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.

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

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

0 ≤ *start* ≤ *end* ≤ (string-length *string*).

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

(string-append *string* …) **procedure**

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

6.8 Vectors

Vectors are heterogeneous structures whose elements are indexed by exact non-negative integers. A vector typically occupies less space than a list of the same length, and the average time required to access a randomly chosen element is typically less for the vector than for the list.

The *length* of a vector is the number of elements that it contains. This number is an exact non-negative integer that is fixed when the vector is created. The *valid indexes* of a vector are the exact non-negative integers less than the length of the vector. The first element in a vector is indexed by zero, and the last element is indexed by one less than the length of the vector.

Vectors are written using the notation **#(***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”)
             ⇒ ex=1c〉 #(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 are unspecified.

(vector *obj* …) **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*.

(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)
                (inexact->exact
                  (round (* 2 (acos -1)))))
               ⇒ 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
```

6.9 Control Features

(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* *obj*$_1$ … *args*) **procedure**

Proc must be a procedure and *args* must be a list. Calls *proc* with the elements of the list

(append (list *obj*$_1$ …) *args*)

as the actual arguments.

```
(apply + (list 3 4))        ⇒  7

(apply + 10 (list 3 4))     ⇒  17

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

((compose sqrt *) 12 75)    ⇒  30
```

(map *proc* *list*$_1$ *list*$_2$ …) **procedure**

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

(for-each *proc* *list*$_1$ *list*$_2$ …) **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 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)
```

`(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 of one argument that, if it is later passed a value, will ignore whatever continuation is in effect at that later time and will give the value instead to the continuation that was in effect when the escape procedure was created.

The escape procedure 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 uses of **`call-with-current-continuation`**. If all real programs were as simple as these examples, there would be no need for a procedure with the power of **`call-with-current-continuation`**.

```
    (call-with-current-continuation
      (lambda (exit)
        (for-each (lambda (x)
                    (if (negative? x)
                        (exit x)))
                  '(54 0 37 -3 245 19))
        #t))                   ⇒  -3
    (define (list-length obj)
      (call-with-current-continuation
        (lambda (return)
          (let r ((obj obj))
            (cond ((null? obj) 0)
                  ((pair? obj) (+ 1 (r (cdr obj))))
                  (else (return #f)))))))

     (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, nonlocal 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 the top level of a Scheme interpreter, for example, then the continuation will take the result, print it on the screen, prompt for the next input, evaluate it, and so on forever. Most of the time the continuation includes actions specified by user code, as in a continuation that will take the result, multiply it by the value stored in a local variable, add seven, and give the answer to the top-level continuation to be printed. Normally these ubiquitous continuations are hidden behind the scenes and programmers don't think much about them. On 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.

6.10 Input and Output

6.10.1 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.

NOTE — Ports are not required to be distinct from other data types. For example, a port may be implemented as a vector, in which case it will satisfy the **`vector?`** predicate.

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

Proc must be a procedure of one argument, and *string* must be a string naming a file. For **`call-with-input-file`**, the file must 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 the procedure returns, then the port is closed automatically and the value yielded by the procedure is returned. If the procedure 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.

`(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.

6.10.2 Input

`(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`.

NOTE — End-of-file objects are not required to be distinct from other data types.

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

`Read` converts external representations of Scheme objects into the objects themselves. That is, it is a parser for the nonterminal (datum) (see 7.2 and 6.3). However, `read` need not implement the full grammar for the nonterminal (number): it may report the violation of an implementation restriction instead. See the NOTE associated with `string->number` on page 30 for permissible restrictions of numeric syntax.

`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-cha`r on that *port* will return the value returned by the preceding call to `peak-char`. In particular, a call to `peek-cha` on an interactive port will hang waiting for input whenever a call to `read-char` would have hung.

6.10.3 Output

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

Writes a written representation of *obj* to the given *port*. If *obj* has a standard external representation, then the written representation generated by `write` shall be parsable by `read` into an equivalent object; but see 6.4 for an exception.

Thus strings that appear in the written representation are enclosed in doublequotes, and within those strings backslash and doublequote characters are escaped by backslashes.

`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`*****`)`** **procedure**
`(display` ***`obj port`*****`)`** **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.

`(newline)` **procedure**
`(newline` ***`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`**.

7. Formal Syntax

This section provides a formal syntax for Scheme written in an extended BNF. The syntax for the entire language is given here.

All spaces in the grammar are for legibility. Case is insignificant (except in the rules for <any character> and ⟨string element⟩); 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 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 that require implicit termination (identifiers, numbers, characters, and dot) may be terminated by any ⟨delimiter⟩, but not necessarily by anything else.

```
(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 | 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 shall 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
```

7.2 External Representations

⟨Datum⟩ generates the standard external representations for Scheme datatypes. The read procedure parses ⟨datum⟩, but see 6.10.2 for restrictions. 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⟩*)
```

7.3 Expressions

```
⟨expression⟩ → ⟨variable⟩
    | ⟨literal⟩
    | ⟨procedure call⟩
    | ⟨lambda expression⟩
    | ⟨conditional⟩
    | ⟨assignment⟩
    | ⟨derived expression⟩

⟨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⟩ ⟨sequence⟩)
        ⟨command⟩*)
    | ⟨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⟩
```

7.4 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, …. *D* keeps track of the nesting depth.

```
⟨quasiquotation⟩ →⟨quasiquotation 1⟩
⟨template 0⟩ → ⟨expression⟩
⟨quasiquotation D⟩ → '⟨template D⟩
    | (quasiquote ⟨template D⟩)
⟨template D⟩ → ⟨simple datum⟩
```

```
    | ⟨list template D⟩
    | ⟨vector template D⟩
    | ⟨unquotation D⟩
⟨list template D⟩ → (⟨template or splice D⟩*)
    | (⟨template or splice D⟩+ . ⟨template D⟩)
    | '⟨template D⟩
    | ⟨quasiquotation D + 1⟩
⟨vector template D⟩ → #(⟨template or splice D⟩*)
⟨unquotation D⟩ → ,⟨template D - 1⟩
    | (unquote ⟨template D - 1⟩)
⟨template or splice D⟩ → ⟨template D⟩
    | ⟨splicing unquotation D⟩
⟨splicing unquotation D) → ,@⟨template D - 1⟩
    | (unquote-splicing ⟨template D - 1⟩)
```

In ⟨quasiquotation⟩s, a ⟨list 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.

7.5 Programs and Definitions

```
⟨program⟩ → ⟨command or definition⟩*
⟨command or definition⟩ → ⟨command⟩ | ⟨definition⟩
⟨definition⟩ → (define ⟨variable⟩ ⟨expression⟩)
     | (define (⟨variable⟩ ⟨def formals⟩) ⟨body⟩)
     | (begin ⟨definition⟩*)
⟨def formals⟩ → ⟨variable⟩*
     | ⟨variable⟩+ . ⟨variable⟩
```

7.6 Derived Expression Types

This section gives rewrite rules for the derived expression types. By the application of these rules, any expression can be reduced to a semantically equivalent expression in which only the primitive expression types (literal, variable, call, **lambda**, **if**, **set!**) occur.

```
(cond (⟨test⟩ ⟨sequence⟩)
      ⟨clause2⟩ …)
  ≡   (if ⟨test⟩
         (begin ⟨sequence⟩)
         (cond ⟨clause2⟩ …))

(cond (⟨test⟩)
      ⟨clause2⟩ …)
  ≡ (or ⟨test⟩ (cond ⟨clause2⟩ …))

(cond (⟨test⟩ => ⟨recipient⟩)
      ⟨clause2⟩ …)
  ≡ (let ((test-result ⟨test⟩)
           (thunk2 (lambda () ⟨recipient⟩))
           (thunk3 (lambda () (cond ⟨clause2⟩ …))))
       (if test-result
           ((thunk2) test-result)
           (thunk3)))

(cond (else ⟨sequence⟩))
  ≡   (begin ⟨sequence⟩)
```

```
(cond)
  ≡   ⟨some expression returning an unspecified value⟩

(case ⟨key⟩
  ((d1 …) ⟨sequence⟩)
  …)
  ≡  (let ((key ⟨key⟩)
           (thunk1 (lambda () ⟨sequence⟩))
           …)
       (cond ((⟨memv⟩ key '(d1 …)) (thunk1))
             …))

(case ⟨key⟩
  ((d1 …) ⟨sequence⟩)
  …
  (else f1 f2 …))
  ≡  (let ((key ⟨key⟩)
           (thunk1 (lambda () ⟨sequence⟩))
           …
           (elsethunk (lambda () f1 f2 …)))
       (cond ((⟨memv⟩ key '(d1 …)) (thunk1))
             …
             (else (elsethunk))))
```

where ⟨memv⟩ is an expression evaluating to the **memv** procedure.

```
(and)              ≡  #t
(and ⟨test⟩)       ≡  ⟨test⟩
(and ⟨test1⟩ ⟨test2⟩ …)
  ≡  (let ((x ⟨test1⟩)
           (thunk (lambda () (and ⟨tests2⟩ …))))
       (if x (thunk) x))

(or)                  ≡  #f
(or ⟨test⟩)           ≡  ⟨test⟩
(or ⟨test1⟩ ⟨test2⟩ …)
  ≡  (let ((x ⟨test1⟩)
       (thunk (lambda () (or ⟨test2⟩ …))))
       if x (thunk) x))

(let ((⟨variable1⟩ ⟨init1⟩) …)
  ⟨body⟩)
  ≡ ((lambda (⟨variable1⟩ …) ⟨body⟩) ⟨init1⟩ …)

(let* () ⟨body⟩)
  ≡  ((lambda () ⟨body⟩))
(let * ((⟨variable1⟩ ⟨init1⟩)
        (⟨variable2⟩ ⟨init2⟩)
        …)
  ⟨body⟩)
  ≡  (let ((⟨variable1⟩ ⟨init1⟩))
       (let* ((⟨variable2⟩ ⟨init2⟩)
              …)
```

```
         ⟨body⟩))

(letrec ((⟨variable1⟩ ⟨init1⟩)
        …)
  ⟨body⟩)
  ≡  (let ((⟨variable1⟩ ⟨undefined⟩)
           …)
        (let ((⟨temp1⟩ ⟨init1⟩)
                …)
         (set! ⟨variable1⟩ ⟨temp1⟩)
         …)
        ⟨body⟩)
```

where (temp$_1$⟩, ⟨temp$_2$⟩, … are variables, distinct from ⟨variable$_1$⟩,…, that do not free occur in the original ⟨init⟩ expressions, and ⟨undefined⟩ is 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, but one is assumed to exist for the purposes of this rewrite rule.) The second **let** expression in the expansion is not strictly necessary, but it serves to preserve the property that the ⟨init⟩ expressions are evaluated in an arbitrary order.

```
(begin ⟨sequence⟩)
  ≡  ((lambda () ⟨sequence⟩))
```

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 ⟨sequence⟩ contains no definitions.

```
(begin ⟨expression⟩)   ≡  (⟨expression⟩
(begin ⟨command⟩ ⟨sequence⟩)
  ≡  ((lambda (ignore thunk) (thunk))
       ⟨command⟩
       (lambda () (begin ⟨sequence⟩)))
```

The following expansion for do is simplified by the assumption that no ⟨step⟩ is omitted. Any do expression in which a ⟨step⟩ is omitted can be replaced by an equivalent **do** expression in which the corresponding ⟨variable⟩ appears as the ⟨step⟩.

```
(do ((⟨variable1⟩ ⟨init1⟩ ⟨step1⟩)
     …)
    (⟨test⟩ ⟨sequence⟩)
  ⟨command1⟩ …)
  ≡  (letrec ((⟨loop⟩
               (lambda (⟨variable1⟩ …)
                 (if ⟨test⟩
                     (begin ⟨sequence⟩)
                     (begin ⟨command1⟩
                            …
                            (⟨loop⟩ ⟨step1⟩ …))))))
       (⟨loop⟩ ⟨init1⟩ …))
```

where ⟨loop⟩ is any variable that is distinct from ⟨variable$_1$⟩, …, and that does not occur free in the **do** expression.

```
(let ⟨variable0⟩ ((⟨variable1⟩ ⟨init1⟩) …)
  ⟨body⟩)
  ≡  ((letrec ((⟨variable0⟩ (lambda (⟨variable1⟩ …)
                              ⟨body⟩)))
        ⟨variable0⟩)
     ⟨init1⟩ …)
```

Bibliography

[B1] IEEE Std 754-1985, IEEE Standard for Binary Floating-Point Arithmetic (ANSI).

[B2] Harold Abelson and Gerald Jay Sussman with Julie Sussman. *Structure and Interpretation of Computer Programs.* Cambridge, MA: MIT Press, 1985.

[B3] William Clinger. How to read floating point numbers accurately. In *ACM SIGPLAN Notices* 25(6), ACM, June 1990.

[B4] 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.

[B5] William Clinger and Jonathan Rees, editors. The revised^4 report on the algorithmic language Scheme.University of Oregon Technical Report CIS-TR-90-02.

[B6] 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 [B7] .

[B7] D. Friedman, C. Haynes, E. Kohlbecker, and M. Wand. Scheme 84 interim reference manual. Indiana University Computer Science Technical Report 153, January 1985.

[B8] G. H. Hardy and E. M. Wright. *An Introduction to the Theory of Numbers.* 5th ed. Oxford University Press, New York NY, 1979.

[B9] Donald E. Knuth. The Art of Computer Programming, vol. 2: Seminumerical Algorithms. Reading, MA: Addison-Wesley, 1969.

[B10] David W. Matula. In-and-Out Conversions. *Communications of the ACM* 11(1), Jan. 1968, pp. 47-50.

[B11] David W. Matula. A Formalization of Floating-Point Numeric Base Conversion. *IEEE Transactions on Computers* C-19, 8, Aug. 1970, pp. 681-692.

[B12] MIT Department of Electrical Engineering and Computer Science. Scheme manual, seventh edition. Sept. 1984.

[B13] Kent M. Pitman. The revised MacLisp manual (Saturday evening edition). MIT Laboratory for Computer Science Technical Report 295, May 1983.

[B14] 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*, pp. 114-122.

[B15] Jonathan Rees and William Clinger, editors. The revised(3) report on the algorithmic language Scheme. In *ACM SIGPLAN Notices* 21(12), ACM, Dec. 1986.

[B16] Guy Lewis Steele Jr. *Common Lisp: The Language.* Burlington, MA: Digital Press, 1984.

[B17] Guy Lewis Steele Jr. and Gerald Jay Sussman. The revised report on Scheme, a dialect of Lisp. MIT Artificial Intelligence Memo 452, Jan. 1978.

[B18] Guy Lewis Steele Jr. and Jon L White. How to Print Floating-Point Numbers Accurately. In *ACM SIGPLAN Notices* 25(6), ACM, June 1990.

[B19] Joseph E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. Cambridge, MA: MIT Press, 1977.

[B20] Gerald Jay Sussman and Guy Lewis Steele Jr. Scheme: an interpreter for extended lambda calculus. MIT Artificial Intelligence Memo 349, Dec. 1975.

[B21] Jean Vuillemin. Exact real computer arithmetic with continued fractions. In *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming*, pp. 14–27.

Annex A Formal Semantics

(Informative)

(This appendix is not a part of IEEE Std 1178–1990, IEEE Standard for the Scheme Programming Language, but is included for information only.)

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 [B19]; the notation is summarized below:

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

The reason that expression continuations take sequences of values instead of single values is to simplify the formal treatment of procedure calls and to make it easy to add 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 *new* $\sigma \in$ L, then σ (*new* $\sigma \mid$ L) $\downarrow 2 =$ *false*.

The definition of is omitted because an accurate definition of 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 is

ε[((lambda (I*) P′) ⟨undefined⟩ …)]

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 ε is the semantic function that assigns meaning to expressions.

A.1 Abstract syntax

K ∈ **Con**	**constants, including quotations**
I ∈ **Ide**	**identifiers (variables)**
E ∈ **Exp**	**expressions**
Γ ∈ **Com = Exp**	**commands**

$$
\begin{aligned}
\mathbf{Exp} \rightarrow\ & \mathbf{K} \mid \mathbf{I} \mid (\mathbf{E_0}\ \mathbf{E}^*) \\
& \mid (\mathtt{lambda}\ (\mathbf{I}^*)\ \Gamma^*\ \mathbf{E_0}) \\
& \mid (\mathtt{lambda}\ (\mathbf{I}^*\ .\ \mathbf{I})\ \Gamma^*\ \mathbf{E_0}) \\
& \mid (\mathtt{lambda}\ \mathbf{I}\ \Gamma^*\ \mathbf{E_0}) \\
& \mid (\mathtt{if}\ \mathbf{E_0}\ \mathbf{E_1}\ \mathbf{E_2}) \mid (\mathtt{if}\ \mathbf{E_0}\ \mathbf{E_1}) \\
& \mid (\mathtt{set!}\ \mathbf{I}\ \mathbf{E})
\end{aligned}
$$

A.2 Domain Equations

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

A.3 Semantic Functions

$\mathrm{K} : \mathrm{Con} \rightarrow \mathrm{E}$

$\varepsilon :\ \mathrm{Exp} \rightarrow \mathrm{U} \rightarrow\ \rightarrow \mathrm{C}$

$\varepsilon^* :\mathrm{Exp}^* \rightarrow \mathrm{U} \rightarrow\ \rightarrow \mathrm{C}$

$C : \mathrm{Com}^* \rightarrow\ \mathrm{U} \rightarrow \mathrm{C} \rightarrow \mathrm{C}$

Definition of K deliberately omitted.

$$\varepsilon\ [\![\mathbf{K}]\!] = \lambda\rho\kappa\ .\ \mathit{send}([\![\mathbf{K}]\!])\ \kappa$$

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

$$
\begin{aligned}
& \varepsilon[\![(\mathbf{E_0}\ \mathbf{E}^*]\!] = \\
& \quad \lambda\rho\kappa.\ \varepsilon^*(\mathit{permute}(\langle \mathbf{E_0}\rangle\ \S\ \mathbf{E}^*)) \\
& \qquad\quad \rho \\
& \qquad\quad (\lambda\in^*\ .\ ((\lambda\in^*\ .\ \mathit{applicable}\ (\in^*\ \downarrow\ 1)\ (\in^*\ \dagger\ 1)\ \kappa)
\end{aligned}
$$

```
                  (unpermute ∈*)))

ε⟦(lambda (I*) Γ* E₀)⟧ =
  λρκ . λσ .
    new σ ∈ L →
      send(⟨new σ | L,
            λ∈*κ′ . #∈* = #I* →
                    tievals(λα* . (λρ. C⟦Γ*⟧ρ′(ε⟦E₀⟧ρ′κ′) )
                                  (extends ρ I* α*))
                          ∈*,
                    wrong  “wrong number of arguments”⟩
            in E)
          κ
          (update (new σ | L) unspecified σ),
      wrong   “out of memory” σ

ε⟦(lambda (I* . I) Γ* E₀)⟧ =
  λρκ . λσ .
    new σ ∈ L →
      send(⟨new σ | L,
            λ∈*κ′ . #∈* ≥ #I* →
                       tievalsrest
                          (λα* . (λρ. C⟦Γ*⟧ρ′(ε⟦E₀⟧ρ′κ′))
                                 (extends ρ (I* § ⟩I⟨) α*))
                       ∈*,
                       (#I*),
                     wrong  “too few arguments”⟩ in E)
              κ
              (update (new σ | L) unspecified σ),
         wrong   “out of memory” σ

ε⟦(lambda I Γ* E₀)⟧ = ε⟦(lambda (. I) Γ* E₀)⟧

ε⟦(if E₀ E₁ E₂)⟧ =
  λρκ . ε⟦E0⟧ ρ (single (λ∈ . truish ∈ → ε⟦E₁⟧ρκ,
                                          ε⟦E₂⟧ρκ))

ε⟦(if E₀ E₁)⟧ =
  λρκ . ε⟦E₀⟧ ρ (single(λ∈ . truish ∈ → ε⟦E₁⟧ρκ,
                                      send unspecified κ))
```

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

```
ε⟦(set&! I E)⟧ =
  λρκ . ε⟦E⟧ ρ (single(λ∈ . assign (lookup ρ I)
                                       ∈
                                       (send unspecified κ)))

ε*⟦⟧ = λρκ . κ⟨⟩

ε*⟦E₀ E*⟧ =
  λρκ . ε⟦E₀⟧ ρ (single(λ∈₀ . ε*⟦E*⟧ ρ (λ∈* . κ (⟨⟨∈₀⟩ § ∈*))) )

C⟦⟧ = λρθ . θ
```

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

A.4 Auxiliary Functions

```
$\mathit{lookup} : \mathbf{U} \to \mathbf{Ide} \to \mathbf{L}$
$\mathit{lookup} = \lambda\rho\mathbf{I}.\ \rho\mathbf{I}$

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

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

$\mathit{send} : \mathbf{E} \to \mathbf{K} \to \mathbf{C}$
$\mathit{send} = \lambda\in\kappa\ .\kappa\langle\in\rangle$

$\mathit{single} : (\mathbf{E} \to \mathbf{C}) \to \mathbf{K}$
$\mathit{single} =$
   $\lambda\psi\in^*\ .\ \#\in^* = 1 \to \psi(\in^* \downarrow 1),$
             $\mathit{wrong}$ “wrong number of return values”

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

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

$\mathit{assign} : \mathbf{L} \to \mathbf{E} \to \mathbf{C} \to \mathbf{C}$
$\mathit{assign} = \lambda\alpha\in\sigma\ .\ (\mathit{update}\ \alpha\in\sigma)$

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

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

$\mathit{tievalsrest} : (\mathbf{L}^* \to \mathbf{C}) \to \mathbf{E}^* \to \mathbf{N} \to \mathbf{C}$
$\mathit{tievalsrest} =$
   $\lambda\psi\in^*\nu\ .\ \mathit{list}(\mathit{dropfirst}\ \in^*\nu)$
                $(\mathit{single}(\lambda\in\ .\ \mathit{tievals}\ \psi\ ((\mathit{takefirst}\ \in^*\nu)\ \S\ \langle\in\rangle)))$

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

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

$\mathit{truish} : \mathbf{E} \to \mathbf{T}$
$\mathit{truish} = \lambda\in\ .\ \in = \mathit{false} \to \mathit{false},\ \mathit{true}$
```

```
permute : Exp* → Exp*   [implementation-dependent]

unpermute : E* → E*   [inverse of permute]

applicate : E → E* → K → C
applicate =
  λ∈∈*κ . ∈ ∈ F → (∈ | ↓ 2)∈*κ, wrong  “bad procedure”

onearg : (E → K → C) → (E* → K → C)
onearg =
  λζ∈*κ . #∈* = 1 → ζ(∈* ↓ 1)κ,
           wrong  “wrong number of arguments”

twoarg : (E → E → K → C) → (E* → K → C)
twoarg =
  λζ∈*κ . #∈* = 2 → ζ(∈* ↓ 1)(∈* ↓ 2)κ,
           wrong  “wrong number of arguments”

list : E* → K → C
list =
    λ∈*κ . #∈* = 0 → send null κ,
             list(∈* † 1)(single(λ∈ . cons⟨∈* ↓ 1, ∈⟩κ))

cons : E* → K → C
cons =
  twoarg(λ∈_1∈_2κσ . new σ ∈ L →
                   (λσ′ . new σ′ ∈ L →
                            send(⟨new σ | L, new σ′ | L, true⟩
                                 in E)
                                κ
                                (update(new σ′ | L)∈_2σ′),
                  wrong  “out of memory” σ′)
                   (update(new σ | L)∈_1σ),
                              wrong  “out of memory” σ)

less : E* → K → C
less =
       twoarg(λ∈_1∈_2κ . (∈_1 ∈ R Λ ∈_2 ∈ R) →
                             send(∈_1 | R < ∈_2 | R → true, false)κ,
                               wrong  “non-numeric argument to <”)

add : E* → K → C
add =
  twoarg(λ∈_1∈_2κ . (∈_1 ∈ R ⌊ ∈_2 ∈ R) →
                        send((∈_1 | R + ∈_2 | R) in E)κ,
                         wrong  “non-numeric argument to +”)

car : E* → K → C
car =
  onearg(λ∈κ . ∈ ∈ E_p → hold(∈ | E_p ↓ 1)κ,
                wrong  “non-pair argument to car”)

cdr : E* → K → C   [similar to car]
```

```
setcar : E* → K → C
setcar =
   twoarg(λ$\in_1$ $\in_2$ . $\in_1$ ∈  $E_p$ →
            ($\in_1$ | $E_p$3) → assign($\in_1$ | $E_p$ ↓ 1)
                                    $\in_2$
                                    (send unspecified κ),
            wrong  “immutable argument to set-car!”,
            wrong  “non-pair argument to set-caruo;)

eqv : E* →  → C
eqv =
  twoarg(λ$\in_1$ $\in_2$ . ($\in_1$ ε M Λ $\in_2$ ε M) →
                     send($\in_1$ | M = $\in_2$ | M true, false)κ,
                ($\in_1$ ∈ Q Λ  $\in_2$ ∈ Q) →
                     send($\in_1$ | Q = $\in_2$ | Q true, false)κ,
                 ($\in_1$ ∈ H Λ  $\in_2$ ∈ H) →
                     send($\in_1$ | H = $\in_2$ | H → true, false)κ,
                ($\in_1$ ∈ R Λ $\in_2$  ∈ R) →
                     send($\in_1$ | R = $\in_2$ | R → true, false)κ,
                ($\in_1$ ∈ $E_p$  Λ $\in_2$   ∈ $E_p$) →
                     send((λ$p_1p_2$ . (($p_1$ ↓ 1) = ($p_2$ ↓ 1)Λ
                                   ($p_1$ ↓ 2) = ($p_2$ ↓ 2)) → true,
                                        false)
                        ($\in_1$ | $E_p$)
                        ($\in_2$ | $E_p$))
                      κ,
                ($\in_1$ ∈ $E_v$ Λ $\in_2$  ∈   $E_v$) → …,
                ($\in_1$ ∈ $E_s$ Λ $\in_2$ ∈ $E_s$) → …,
                ($\in_1$ ∈ F Λ $\in_2$ ∈ F) →
                    send(($\in_1$ | F ↓ 1) = ($\in_2$ | F ↓ 1) → true, false)
                      κ,
                    send false κ)
apply : E* → K → C
apply =
  twoarg(λ$\in_1\in_2$κ . $\in_1$ ∈ F → valueslist ⟨$\in_2$⟩(λ∈* . applicate $\in_1$ ∈*κ),
                      wrong  “bad procedure argument to apply”)

valueslist : E* → K → C
valueslist =
   onearg(λ∈κ . ∈ ∈ $E_p$ →
                    cdr⟨∈⟩
                       (λ∈* . valueslist
                              ∈*
                              (λ∈* . car⟨∈⟩(single(λ∈ . κ(⟨∈⟩ § ∈*))))),
               ∈ = null → κ⟨ ⟩,
               wrong  “non-list argument to values-list”)

cwcc : E* → K → C [call-with-current-continuation]
cwcc =
  onearg(λ∈κ . ∈ ∈ F →
                    (λσ . new σ ∈ L →
                          applicate ∈
                                  ⟨⟨new σ | L, λ∈*κ′ . κ∈*⟩ in E⟩
                    κ
                    (update(new σ | L)
                           unspecified
                           σ),
          wrong  “out of memory” σ),
  wrong  “bad procedure argument”)
```

Annex B Number System Subsets

(Informative)

(This appendix is not a part of IEEE Std 1178-1990, IEEE Standard for the Scheme Programming Language, but is included for information only.)

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 [B13] little effort was made to execute numerical code efficiently. This standard recognizes the excellent work of the Common Lisp committee and accepts many of their recommendations. In some ways this standard simplifies and generalizes their proposals in a manner consistent with the purposes of Scheme.

The Scheme number system (see 6.5) is typically implemented using several internal representations. A complete implementation of Scheme numbers is both large and complex. For this reason, it may be undesirable to implement the entire functionality and the standard specifically allows implementations to provide subsets of the full functionality. This appendix is a guide to Scheme implementors suggesting reasonable and consistent subsets.

The appendix is organized into three parts. The first part describes a minimal subset, known as the fixnum subset. Following this are two sections discussing two distinct classes of extensions: exact and inexact arithmetic. Either or both may be compatibly added to the fixnum base. The addition of any other numeric representation transforms simple fixnum arithmetic into a more complicated "generic" system. This is usually accompanied by an additional performance cost associated with the determination (usually at run-time) of the representation used for the arguments.

B.1 Minimal Subset

The minimum practical Scheme implementation should include signed exact integers of a fixed limited precision. (Even more restrictive subsets of the integers, e.g., the omission of negative integers, are consistent with this standard. Such subsets are significantly less useful and their implementation is discouraged.) In such an implementation, when the mathematically expected result of a computation is not representable as an integer within the appropriate range, the computation is considered to have violated an implementation restriction. In general when such a situation occurs, the implementation may choose either to coerce the result to an inexact number or to report the violation; in this case, since there are no inexact numbers, the implementation must report the violation.

This representation choice is traditionally known as *fixnum*, and has two primary advantages. First, the precision is usually chosen to correspond to the natural word size of the machine, thus allowing the implementation to use the built-in integer arithmetic hardware. (This advantage is not fully realizable on many machines because of the need to test for overflow.) Second, the size of the fixnum representation is typically chosen so that fixnums need not be allocated in the heap.

B.2 Exact Arithmetic

A natural first extension is to eliminate the precision restriction on integers. The traditional way is to implement *bignum* arithmetic, also known as extended (or infinite) precision arithmetic (see [B9]). Implementation of these operations, along with the coercion between fixnum and bignum representations, is a time-consuming and exacting task; but most implementations have found the benefits outweigh the costs. The algorithms are rather slow, and space for these numbers is allocated in the heap.

The next logical extension is the implementation of exact rational arithmetic, sometimes referred to as *ratnum* arithmetic; the advantages of exact rational arithmetic are significant, and given bignum arithmetic, the implementation cost is small. While it is possible to provide an implementation of exact rational numbers based on a fixnum (rather than bignum) representation, experience has shown that the limitations of such implementations are unintuitive and consequently hard to use.

If the implementation provides rational numbers, it should supply the following procedures in addition to those required in 6.5:

```
/
denominator
numerator
rationalize
```

The final extension to exact arithmetic is to implement exact real arithmetic. This is usually not done despite the existence of at least one example implementation [B21], because of the inefficiency of the known algorithms.

B.3 Inexact Arithmetic

The usual representation employed to implement inexact real numbers is a well-established one: floating point. We strongly urge implementors using this representation to adhere to the IEEE 32-bit and 64-bit floating-point standards, IEEE Std 754-1985 [1]. A simple implementation might support only one format (preferably the 64-bit format) while a more elaborate one might support multiple formats, carefully observing the restrictions of this standard (see 6.5).

An interesting alternative to floating point is rational numbers whose denominator never exceeds a predetermined maximum value but whose numerator is unbounded. This conveniently allows the representation of arbitrarily large numbers but has a fixed smallest increment (and hence a fixed smallest representable positive value). The algorithms for this representation are virtually identical to those used for exact rational arithmetic, and they have mathematically acceptable behavior for many applications.

We do not recommend implementation of more restrictive forms of inexact arithmetic, such as inexact integer arithmetic, because inexact arithmetic is primarily useful for approximations to the real number system, and the more restrictive forms are not very useful. Also, a good implementation of inexact real arithmetic will to a large extent subsume the other representations.

If inexact real numbers are implemented, the following operations should be supported in addition to those required in 6.5:

```
/             exact->inexact      numerator
acos          exp                 rationalize
asin          expt                sin
atan          inexact->exact      sqrt
cos           log                 tan
denominator
```

The final step in the number tree is to complex arithmetic. The usual implementation of these numbers is as pairs of real numbers in a simple rectangular coordinate representation ($x + iy$). We suggest implementors provide complex arithmetic only if real arithmetic is also provided. Without a reasonable approximation to the real numbers the usefulness of complex numbers is suspect, and it is preferable to omit them entirely.

The operations that should be provided to support complex numbers (in addition to those of inexact real arithmetic) are as follows:

```
angle
imag-part
magnitude
make-polar
make-rectanqullar
real-part
```

Annex C Implementation of Numeric Datatypes

(Informative)

(This appendix is not a part of IEEE Std 1178-1990, IEEE Standard for the Scheme Programming Language, but is included for information only.)

This appendix discusses several aspects of Scheme's numeric datatypes that are of interest to implementors. We assume that the reader is familiar with the implementation of generic arithmetic in Lisp (or other languages like Lisp in this respect), and focus on those features that are both new and Unusual (a good discussion of the basics of generic arithmetic can be found in 2.4 of [B2]). This sheds some light on common problems that Scheme implementors may encounter.

C.1 Exactness

Exactness is the one fundamentally new idea that is introduced by Scheme's arithmetic, and consequently it is the most important topic for implementors to consider.

Exactness serves two basic purposes. First, it can be used as an “alarm” to inform the programmer when the Scheme system has caused precision to be lost; naively, if the result of a computation is exact, the programmer can trust that it is correct. Conversely, if the result is inexact, the programmer is warned that the answer is only approximate and that numerical analysis is required to determine the error involved.

The second purpose of exactness is abstraction, isolating programmers from the methods used to implement inexact arithmetic. While it seems likely that floating-point arithmetic will be the most common representation, this abstraction permits implementors to experiment with other numerical methods and to take advantage of unusual hardware. Provided that the implementor does a good job, many programmers won't need to know what methods are being used (the exception being programmers who need to guarantee tight bounds on the error of their computations).

The property of exactness that serves these two purposes is a simple one: an output of a computation is inexact whenever any of its inputs is inexact. This general rule holds in nearly all cases (see 6.5.2), but some of its consequences may be surprising; we discuss three such consequences.

First, the “rounding operations” (**`floor`**, **`ceiling`**, **`truncate`**, and **`round`**) return inexact results when given inexact arguments. Many programmers familiar with other Lisp implementations will expect these operations to return exact results, and may be confused when their programs don't work correctly. Implementors should be careful not to make the same mistake. As a concrete example, Common Lisp supports two datatypes, floating-point numbers and integers, that respectively correspond to Scheme's inexact reals and exact integers. Unlike Scheme, Common Lisp provides two sets of procedures for rounding (see [B16], pp. 215–218). The procedures **`floor`**, **`ceiling`**, **`truncate`**, and **`round`** convert floating-point numbers to integers, while the procedures **`ffloor`**, **`fceiling`**, **`ftruncate`**, and **`fround`** convert floating point to floating point. Despite their names, it is the second set of procedures that corresponds to Scheme’s rounding operations.

The second surprising consequence is that **`max`** and **`min`**, when given mixed exact and inexact arguments, return an inexact result. This is true even in the following case, which demonstrates the surprising aspect:

```
(max 3.9 4)              ⇒  4.0
```

Implementors must be careful to notice this, as the “obvious” way to implement these procedures is to compare the arguments and return the largest (smallest) one. In Scheme, this “obvious” implementation is incorrect; the procedures must notice when one or more of the arguments is inexact and in that case guarantee that the result is inexact.

NOTE — A problem arises in this situation when an exact argument is to be coerced to an inexact result. If there is no inexact number equal to the argument, the implementation must choose one that is close to the argument. The implementation can choose the inexact number that is closest to the argument, or it can choose the closest inexact number that exceeds the argument in the appropriate direction.

Finally, and perhaps most surprising for implementors, is that operations defined on integers (**`quotient`**, **`remainder`**, **`modulo`**, **`gcd`**, **`lcm`**) and rationals (**`numerator`**, **`denominator`**) must accept inexact arguments (producing, of course, inexact results). This may not matter much to programmers, but is notable for implementors. As an example, if floating point is used to represent all inexact real numbers, this means that any floating-point number satisfying the predicate **`integer?`** can be used as an argument to **`gcd`**; furthermore, **`numerator`** and **`denominator`** must accept all floating-point numbers as arguments. Observe also, complex numbers with inexact zero imaginary part must satisfy the **`real?`** predicate, and thus might also satisfy **`integer?`** or **`rational?`**.

Extending the integer and rational operations to cover these cases is straightforward, especially if exact equivalents exist for every inexact number; in that case merely map the inexact arguments to exact, perform the operation, and apply **`exact->inexact`** to the result (this is one of several reasons why it is desirable to implement unlimited-precision exact integers and rationals). Otherwise, the implementation is somewhat more involved; small implementations may find it desirable to refuse such arguments as violations of an implementation restriction.

C.2 Transitivity of Order Predicates

Unlike most other dialects of Lisp, Scheme requires the numerical order predicates (**`=`**, **`<`**, **`<=`**, **`>`**, and **`>=`**) to be transitive. In most cases this is easily accomplished by standard techniques; however, in the case of mixed exact and inexact arithmetic, the standard techniques may fail to satisfy this requirement. This problem arises specifically when floating point is used to represent inexact numbers, although it may occur with other representations as well.

A particular technique that does not satisfy Scheme's requirements is the *floating-point contagion* rule of Common Lisp (see [B16], p. 194; interestingly, [5], pp. 289-291 explains that the rule was recently modified to be compatible with Scheme's requirements). This rule specifies that when operating on a floating-point number and a rational number, the rational number is first converted to floating-point and the result of the conversion is operated on in place of the original rational. In a Scheme system, this rule can be interpreted as applying to inexact reals (implemented as floating-point numbers) and exact rationals. In the case of number-valued operators, such as +, this contagion rule always generates a correct answer; not so for the order predicates.

The reason that this rule fails has to do with precision. Floating-point numbers have a limited amount of precision, while exact rational numbers have effectively unlimited precision. Thus, the conversion of an exact rational number to a floating-point number can result in a loss of precision. In mathematical terms, the mapping from rational numbers to floating-point numbers is "many-to-one," which means that coercion of two distinct rational numbers may result in the same floating-point number.

The simplest way to avoid this problem is analogous to the floating-point contagion rule: when comparing a floating-point number to an exact rational, convert the floatingpoint number to an exact rational and then compare the two rational numbers. This works because the mapping from floating-point to rational numbers, if properly implemented, preserves all of the precision in the argument. However, this can only work if unlimited-precision rational numbers are available (another argument for implementing them).

When unlimited-precision rationals are not available (i.e., when all exact real numbers are integers), a more complex method is required. A detailed description of this method is beyond the scope of this appendix, but we can sketch an outline. Call the floating-point argument x_1 and the exact integer argument n_1. Convert n_1 to a floating-point number and call the result x_2. Then convert x_2 back to an integer, and call the result n_2. Finally compare x_1 to x_2, modifying the result of the comparison to account for the fact that x_2 differs from n_1 by $n_2 - n_1$. (Note that this description ignores some limit cases, such as when n_1 is too large to be converted to floating point or x_2 is too large to be converted to an integer, but these cases are easily handled.)

C.3 External Representations

The difficulties inherent in converting a binary floating-point number to or from a decimal external representation have been known for some time (for example, see , [B11], and [B18]), but are still not widely appreciated. This is of interest to the Scheme implementor when floating-point representations are used, and may also apply to other inexact representations.

The definition of **`number->string`** places strong constraints on that procedure when its argument is a floating-point number; it simultaneously places strong constraints on **`string->number`**. Two requirements from this definition deserve attention. The first requirement is that the result of **`number->string`**, if passed to **`string->number`**, returns a number which is **eqv**? to the original argument of **`number->string`** (in [B18] this is referred to as the "internal identity requirement"). The second requirement is that the number of digits produced for decimal-point notation is the minimum required to satisfy the first requirement. The combination of these requirements is sufficiently severe that most standard algorithms cannot be used (for example, the algorithms described in [B9], 4.4).

Fortunately, when IEEE floating-point arithmetic is used, it is possible to implement these procedures in a straight-forward manner: IEEE floating-point arithmetic specifies the implementation of binary-to-decimal and decimal-to-binary conversions that satisfy the internal identity requirement within certain ranges; it is desirable to take advantage of these conversions as they are typically implemented in hardware. The binary-to-decimal conversion is not required to minimize the number of output digits, but a suitable conversion can be implemented by trimming its output and testing the trimmed output for equivalence.

If IEEE floating-point arithmetic is unavailable these procedures will have to be implemented in software. To our knowledge, the only correct algorithm for **`number->string`** is that described in [B18]; we know of two correct algorithms for **`string->number`**, which are described in [B3].

C.4 Rationalize

The procedure **`rationalize`** is interesting because most programming languages do not provide anything analogous to it. For simplicity, we present an algorithm that z computes the correct result for exact arguments (provided the implementation supports exact rational numbers of unlimited precision), and produces a reasonable answer for inexact arguments when inexact arithmetic is implemented using floating point. Those interested in the theory of this algorithm should refer to [B8](e thank Alan Bawden for contributing this algorithm).

```
(define (rationalize × e)
    (simplest-rational (– × e) (+ × e)))
(define (simplest-rational × y)
    (define (simplest-rational-internal × y)
       ;; aumes 0 < X < Y
       (let ((fx (floor x))
             (fy (floor y)))
         (cond ((not (< fx x))
                 fx)
                ((= fx fy)
                 (+ fx
                    (/ (simplest-rational-internal
                        (/ (- y fy))
                        (/ (- x fx))))))
                (else
                  (+ 1 fx)))))
    ;; do some juggling to satisfy preconditions
    ;; of simplest-rational-internal.
    (cond ((< y x)
             (simplest-rational y x))
            ((not (< x y))
             (if (rational? x) × (error)))
            ((positive? x)
             (simplest-rational-internal × y))
              ((negative? y)
             (- (simplest-rational-internal (- y)
                                            (- x))))
            (else
             (if (and (exact? x) (exact? y))
                 0
                 0.0))))
```