

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

The Institute of Electrical and Electronics Engineers, Inc.

345 East 47th Street, New York, NY 10017-2394, USA

© 1991 by the Institute of Electrical and Electronics Engineers, Inc.

All rights reserved. Published 1991. Printed in the United States of America

Library of Congress Cataloging in Publication Data will be found in the following page.

No part of this publication may be reproduced in any form, in an electronic retrieval system or otherwise, without prior written permission of the publisher.

IEEE Standards documents are developed within the Technical Committees of the IEEE Societies and the Standards Coordinating Committees of the IEEE Standards Board. Members of the committees serve voluntarily and without compensation. They are not necessarily members of the Institute. The standards developed within IEEE represent a consensus of the broad expertise on the subject within the Institute as well as those activities outside of IEEE which have expressed an interest in participating in the development of the standard.

Use of an IEEE Standard is wholly voluntary. The existence of an IEEE Standard does not imply that there are no other ways to produce, test, measure, purchase, market, or provide other goods and services related to the scope of the IEEE Standard. Furthermore, the viewpoint expressed at the time a standard is approved and issued is subject to change brought about through developments in the state of the art and comments received from users of the standard. Every IEEE Standard is subjected to review at least every five years for revision or reaffirmation. When a document is more than five years old, and has not been reaffirmed, it is reasonable to conclude that its contents, although still of some value, do not wholly reflect the present state of the art. Users are cautioned to check to determine that they have the latest edition of any IEEE Standard.

Comments for revision of IEEE Standards are welcome from any interested party, regardless of membership affiliation with IEEE. Suggestions for changes in documents should be in the form of a proposed change of text, together with appropriate supporting comments.

Interpretations: Occasionally questions may arise regarding the meaning of portions of standards as they relate to specific applications. When the need for interpretations is brought to the attention of IEEE, the Institute will initiate action to prepare appropriate responses. Since IEEE Standards represent a consensus of all concerned interests, it is important to ensure that any interpretation has also received the concurrence of a balance of interests. For this reason IEEE and the members of its technical committees are not able to provide an instant response to interpretation requests except in those cases where the matter has previously received formal consideration.

Comments on standards and requests for interpretations should be addressed to:

Secretary, IEEE Standards Board
445 Hoes Lane
P.O. Box 1331
Piscataway, NJ 08555-1331
USA

IEEE Standards documents are adopted by the Institute of Electrical and Electronics Engineers without regard to whether their adoption may involve patents on articles, materials, or processes. Such adoption does not assume any liability to any patent owner, nor does it assume any obligation whatever to parties adopting the standards documents.

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-class 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-class procedures as in the lambda calculus, thereby proving the usefulness of static scope rules and block structure in a dynamically typed language. Scheme was the first major dialect of Lisp to distinguish procedures from lambda expressions and symbols, to use a single lexical environment for all variables, and to evaluate the operator position of a procedure call in the same way as an operand position. 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].

¹See 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	R. Kent Dybvig	Richard Mlynarik
Norman Adams	Marc Feeley	Andy Norman
Cyril N. Alberga	Daniel P. Friedman	Eric Ost
Joel F. Bartlett	Mark Friedman	John D. Ramsdell
Scott Burson	Richard P. Gabriel	Jonathan Rees
Clyde Camp	John Gateley	Guillermo J. Rozas
Bill Campbell	Arthur Gleckler	Benjamin Schreiber
Jerome Chailloux	Patrick Greussay	George Springer
Stewart Clamen	Jed Harris	Guy L. Steele Jr.
William Clinger	Robert Hieb	Gerald Jay Sussman
Pavel Curtis	Takayasu Ito	Eric Tiedemann
Jeffrey Dalton	Roger Kirchner	Queyen Tran
Olivier Danvy	Paul Kristoff	R. L. Tritchard
Klaus Dassler	Tim McNerney	Mitchell Wand
Kenneth Dickey	William Maddox	Jon L. White
Bruce Duba	Sidney Marshall	Taiichi Yuasa
	Robert Mathis	

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

Harold Abelson	Neal M. Gafter	Cuong Nguyen
William B. Adams	John Gateley	Peter Olsen
Mohammad Al-Malki	Tayeb Giuma	James R. Otto
Duane L. Anderson	Willard Graves	Don Oxley
Joel F. Bartlett	Gregory Guthrie	Chandresh J. Patel
David H. Bartley	Christopher T. Haynes	Crispin Perdue
Winsor A. Brown	Herbert Hecht	Robert Pettengill, Jr.
Lyle Burnett	Robert H. Hyerle	John D. Ramsdell
Todd Busniewski	John Jensen	Hans Roosli
Carl Cagan	Richard Karcich	Norman Schneidewind
Michael J. Caldwell	Evan Kirshenbaum	David Seraphin
George Carson	Peter M. Kogge	Michael Smolin
Donald Chi	Ezzat Korany	Serge Szukalow
David Cohen	Paul Kristoff	Mitchell Wand
Paul D. Cook	Takahiko Kuki	Carl Warren
J. R. Davis	Tuvia Lamdan	W. L. Whipple
Kenneth Alan Dickey	James S. Miller	Fritz Whittington
Su Dongzhuang	John E. Montague	Andrew Wilson
Kent R. Dybvig	Keith Morgan	Qiufeng Wu
Jeff S. Ebert		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:

Marco W. Migliaro, *Chair*
James M. Daly, *Vice Chair*
Andrew G. Salem, *Secretary*

Dennis Bodson
Paul L. Borrill
Fletcher J. Buckley
Allen L. Clapp
Stephen R. Dillon
Donald C. Fleckenstein
Jay Forster*
Thomas L. Hannan

Kenneth D. Hendrix
John W. Horch
Joseph L. Koepfinger*
Irving Kolodny
Michael A. Lawler
Donald J. Loughry
John E. May, Jr.

Lawrence V. McCall
L. Bruce McClung
Donald T. Michael*
Stig Nilsson
Roy T. Oishi
Gary S. Robinson
Terrance R. Whittemore
Donald W. Zipse

*Member Emeritus

CLAUSE	PAGE
Introduction	1
Objectives.....	1
Scope	1
Future Directions.....	2
Compliance	2
Organization of the Document	2
Definitions of Terms	2
Examples.....	3
Base Document	3
References.....	4
Description of the Language	4
1. Overview of Scheme	4
1.1 Semantics	4
1.2 Syntax.....	5
1.3 Notation and Terminology	5
2. Lexical Conventions	7
2.1 Identifiers	7
2.2 Whitespace and Comments.....	8
2.3 Other Notations	9
3. Basic Concepts	9
3.1 Variables and Regions	9
3.2 True and False	10
3.3 External Representations.....	10
3.4 Disjointness of Types.....	10
3.5 Storage Model	11
4. Expressions	11
4.1 Primitive Expression Types	12
4.2 Derived Expression Types	14
5. Program Structure	20
5.1 Programs	20
5.2 Definitions.....	20
6. Standard Procedures.....	22

CLAUSE	PAGE
6.1 Booleans.....	22
6.2 Equivalence Predicates	23
6.3 Pairs and Lists	26
6.4 Symbols.....	30
6.5 Numbers	32
6.6 Characters.....	42
6.7 Strings	43
6.8 Vectors	45
6.9 Control Features.....	46
6.10 Input and Output	49
7. Formal Syntax	51
7.1 Lexical Structure	51
7.2 External Representations.....	53
7.3 Expressions	53
7.4 Quasiquotations.....	54
7.5 Programs and Definitions.....	55
7.6 Derived Expression Types	55
7.6 Bibliography.....	58
Annex A (informative) Formal Semantics	60
Annex B (informative) Number System Subsets	67
Annex C (informative) Implementation of Numeric Datatypes	70

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 “Revised⁴ 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.

²IEEE 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, $\langle \text{expression} \rangle$, $\langle \text{variable} \rangle$. Syntactic variables should be understood to denote segments of program text; for example, $\langle \text{expression} \rangle$ stands for any string of characters which is a syntactically valid expression. The notation

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

indicates zero or more occurrences of a $\langle \text{thing} \rangle$, and

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

indicates one or more occurrences of a $\langle \text{thing} \rangle$.

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:

<i>obj</i>	any object
<i>list</i> , <i>list</i> ₁ , ... <i>list</i> _j , ...	list (see 6.3)
<i>z</i> , <i>z</i> ₁ , <i>z</i> _j , ...	complex number
<i>x</i> , <i>x</i> ₁ , ... <i>x</i> _j , ...	real number
<i>y</i> , <i>y</i> ₁ , ... <i>y</i> _j , ...	real number
<i>q</i> , <i>q</i> ₁ , ... <i>q</i> _j , ...	rational number
<i>n</i> , <i>n</i> ₁ , ... <i>n</i> _j , ...	integer
<i>k</i> , <i>k</i> ₁ , ... <i>k</i> _j , ...	exact non-negative integer

1.3.3 Evaluation Examples

The symbol “ $\Rightarrow$ ” used in program examples should be read “evaluates to.” For example,

(* 5 8) $\Rightarrow$ **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:

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

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:

- **(<variable₁> ...):** 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.

- $\langle \text{variable} \rangle$: 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 $\langle \text{variable} \rangle$.
- $(\langle \text{variable}_1 \rangle \dots \langle \text{variable}_{n-1} \rangle . \langle \text{variable}_n \rangle)$: 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.

<code>((lambda x x) 3 4 5 6)</code>	$\Rightarrow$	<code>(3 4 5 6)</code>
<code>((lambda (x y . z) z) 3 4 5 6)</code>	$\Rightarrow$	<code>(5 6)</code>

It is an error for a $\langle \text{variable} \rangle$ to appear more than once in $\langle \text{formals} \rangle$.

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

<code>(if <test> <consequent> <alternate>)</code>	syntax
<code>(if <test> <consequent>)</code>	syntax

Syntax: $\langle \text{Test} \rangle$, $\langle \text{consequent} \rangle$, and $\langle \text{alternate} \rangle$ may be arbitrary expressions.

Semantics: An **if** expression is evaluated as follows: first, $\langle \text{test} \rangle$ is evaluated. If it yields a true value (see 6.1), then $\langle \text{consequent} \rangle$ is evaluated and its value is returned. Otherwise $\langle \text{alternate} \rangle$ is evaluated and its value is returned. If $\langle \text{test} \rangle$ yields a false value and no $\langle \text{alternate} \rangle$ is specified, then the result of the expression is unspecified.

<code>(if (> 3 2) 'yes 'no)</code>	$\Rightarrow$	<code>? yes</code>
<code>(if (> 2 3) 'yes 'no)</code>	$\Rightarrow$	<code>? no</code>
<code>(if (> 3 2) (- 3 2) (/ 1 0))</code>	$\Rightarrow$	<code>? 1</code>

4.1.6 Assignments

<code>(set! <variable> <expression>)</code>	syntax
---	---------------

$\langle \text{Expression} \rangle$ is evaluated, and the resulting value is stored in the location to which $\langle \text{variable} \rangle$ is bound. $\langle \text{Variable} \rangle$ 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.

<code>(define x 2)</code>	
<code>(+ x 1)</code>	$\Rightarrow$ <code>3</code>
<code>(set! x 4)</code>	$\Rightarrow$ <i>unspecified</i>
<code>(+ x 1)</code>	$\Rightarrow$ <code>5</code>

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 <clause₁> <clause₂> ...) **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 <expression₁> <expression₂> ...).

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> <clause₁> <clause₂> ...) **syntax**

Syntax: <Key> may be any expression. Each <clause> shall have the form

((datum₁ ...) <expression₁> <expression₂> ...),

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

(else <expression₁> <expression₂> ...).

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 **equiv?**; 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 <test₁> ...)**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 <test₁> ...)**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

```
((<variable1> <init1>) ...),
```

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 $\langle \text{init} \rangle$ s are evaluated in the current environment (in some unspecified order), the $\langle \text{variable} \rangle$ s are bound to fresh locations holding the results, the expressions in $\langle \text{body} \rangle$ are evaluated sequentially in the extended environment, and the value of the last expression of $\langle \text{body} \rangle$ is returned. Each binding of a $\langle \text{variable} \rangle$ has $\langle \text{body} \rangle$ 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* $\langle \text{bindings} \rangle$ $\langle \text{body} \rangle$) **syntax**

Syntax: $\langle \text{Bindings} \rangle$ shall have the form

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

and $\langle \text{body} \rangle$ 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 $(\langle \text{variable} \rangle \langle \text{init} \rangle)$ 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 $\langle \text{bindings} \rangle$ $\langle \text{body} \rangle$) **syntax**

Syntax: $\langle \text{Bindings} \rangle$ shall have the form

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

and $\langle \text{body} \rangle$ shall be a sequence of one or more expressions. It is an error for a $\langle \text{variable} \rangle$ to appear more than once in the list of variables being bound.

Semantics: The $\langle \text{variable} \rangle$ s are bound to fresh locations holding undefined values, the $\langle \text{init} \rangle$ s are evaluated in the resulting environment (in some unspecified order), each $\langle \text{variable} \rangle$ is assigned to the result of the corresponding $\langle \text{init} \rangle$, the expressions in $\langle \text{body} \rangle$ are evaluated sequentially in the extended environment, and the value of the last expression in $\langle \text{body} \rangle$ is returned. Each binding of a $\langle \text{variable} \rangle$ 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)))))
```