Revised\textsuperscript{7} Report on the Algorithmic Language Scheme

ALEX SHINN, JOHN COWAN, AND ARTHUR A. GLECKLER (Editors)

STEVEN GANZ  
ALEXEY RADUL  
OLIN SHIVERS  
AARON W. HSU  
JEFFREY T. READ  
ALARIC SNELL-PYM  
BRADLEY LUCIER  
DAVID RUSH  
GERALD J. SUSSMAN  
EMMANUEL MEDERNACH  
BENJAMIN L. RUSSEL

RICHARD KELSEY, WILLIAM CLINGER, AND JONATHAN REES  
(Editors, Revised\textsuperscript{5} Report on the Algorithmic Language Scheme)

MICHAEL SPERBER, R. KENT DYBVIG, MATTHEW FLATT, AND ANTON VAN STRAATEN  
(Editors, Revised\textsuperscript{6} Report on the Algorithmic Language Scheme)

Dedicated to the memory of John McCarthy

8 August 2011
SUMMARY

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

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

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

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

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

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

Appendix A provides a list of the standard libraries and the identifiers that they export.

Appendix B provides a list of optional but standardized implementation feature names.

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

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

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

Background

The first description of Scheme was written in 1975 [34]. A revised report [31] appeared in 1978, which described the evolution of the language as its MIT implementation was upgraded to support an innovative compiler [32]. Three distinct projects began in 1981 and 1982 to use variants of Scheme for courses at MIT, Yale, and Indiana University [27, 23, 15]. An introductory computer science textbook using Scheme was published in 1984 [3].

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, the RRRS [7], was published at MIT and Indiana University in the summer of 1985. Further revision took place in the spring of 1986, resulting in the R^3RS [29]. Work in the spring of 1988 resulted in R^4RS [9], which became the basis for the IEEE Standard for the Scheme Programming Language in 1991 [18]. In 1998, several additions to the IEEE standard, including high-level hygienic macros, multiple return values and eval, were finalized as the R^5RS [2].

In the fall of 2006, work began on a more ambitious standard, including many new improvements and stricter requirements made in the interest of improved portability. The resulting standard, the R^6RS, was completed in August 2007 [1], and was organized as a core language and set of mandatory standard libraries. The size and goals of the R^6RS, however, were controversial, and adoption of the new standard was not as widespread as had been hoped.

In consequence, the Scheme Steering Committee decided in August 2009 to divide the standard into two separate but compatible languages — a “small” language, suitable for educators, researchers and embedded languages, focused on R^RScompatibility, and a “large” language focused on the practical needs of mainstream software development which would evolve to become a replacement for R^6RS.

The present report describes the “small” language of that effort.

We intend this report to belong to the entire Scheme community, and so we grant permission to copy it in whole or in part without fee. In particular, we encourage implementors of Scheme to use this report as a starting point for manuals and other documentation, modifying it as necessary.

Acknowledgements

We would like to thank the members of the Steering Committee, William Clinger, Marc Feeley, Chris Hanson, Jonathan Rees, and Olin Shivers, for their support and guidance. We’d like to thank the following people for their help: Per Bothner, Taylor Campbell, Ray Dillinger, Brian Harvey, Shiro Kawai, Jonathan Kraut, Thomas Lord, Vincent Manis, Jeronimo Pellegrini, Jussi Piitulainen, Alex Queiroz, Jim Rees, Jay Reynolds Freeman, Malcolm Tredinnick, Denis Washington, Andy Wingo, and Andre van Tonder.

In addition we would like to thank all the past editors, and the people who helped them in turn: Hal Abelson, Norman Adams, David Bartley, Alan Bawden, Michael Blair, Gary Brooks, George Carrette, Andy Cromarty, Pavel Curtis, Jeff Dalton, Olivier Danvy, Ken Dickey, Bruce Duba, Robert Findler, Andy Freeman, Richard Gabriel, Yekta G{"u}rsel, Ken Haase, Robert Halstead, Robert Hieb, Paul Hudak, Morry Katz, Eugene Kohlbecker, Chris Lindblad, Jacob Matthews, Mark Meyer, Jim Miller, Don Oxley, Jim Philbin, Kent Pitman, John Ramsdell, Guillermo Rozas, Mike Shaff, Jonathan Shapiro, Guy Steele, Julie Sussman, Perry Wagle, Mitchel Wand, Daniel Weise, Henry Wu, and
Ozan Yigit. We thank Carol Fessenden, Daniel Friedman, and Christopher Haynes for permission to use text from the Scheme 311 version 4 reference manual. We thank Texas Instruments, Inc. for permission to use text from the *TI Scheme Language Reference Manual*. We gladly acknowledge the influence of manuals for MIT Scheme, T, Scheme 84, Common Lisp, and Algol 60.
DESCRIPTION OF THE LANGUAGE

1. Overview of Scheme

1.1. Semantics

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

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

Scheme has latent as opposed to manifest types as a dynamically typed language. Types are associated with values (also called objects) rather than with variables. (Some authors refer to languages with latent types as weakly typed or dynamically typed languages.) Other languages with latent types are APL, Snobol, and other dialects of Lisp. Languages with manifest types (sometimes referred to as strongly typed or statically typed languages) include Algol 60, Pascal, and C. Statically typed languages, by contrast, associate types with variables and expressions as well as with values.

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

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

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

One distinguishing feature of Scheme is that continuations, which in most other languages only operate behind the scenes, also have “first-class” status. Continuations are useful for implementing a wide variety of advanced control constructs, including non-local exits, backtracking, and coroutines. See section [6.10].

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

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

1.2. Syntax

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

The read procedure performs syntactic as well as lexical decomposition of the data it reads. The read procedure parses its input as data (section [7.1.2]), not as program. The formal syntax of Scheme is described in section [7.1].

1.3. Notation and terminology

1.3.1. Primitive, library, Base and optional features

It is required that every implementation of Scheme support all features that Every identifier defined in this report appears in one of several libraries. Identifiers defined in the base library are not marked as being implementations specially in the body of the report. A summary of all the standard libraries and the features they provide is given in Appendix A.
Implementations must provide the base library and all the identifiers exported from it. Implementations are free to omit optional features of Scheme or to add extensions of the libraries given in this report, but each library must either be provided in its entirety, exporting no additional identifiers, or else omitted altogether.

Implementations may provide other libraries not described in this report. Implementations may also extend the function of any identifier in this report, provided the extensions are not in conflict with the language reported here. In particular, implementations must support portable code by providing a syntactic mode that preempts no lexical conventions of this report.

To aid in understanding and implementing Scheme, some features are marked as... These can be easily implemented in terms of the other, primitive features. They are redundant in the strict sense of the word, but they capture common patterns of usage, and are therefore provided as convenient abbreviations of operation in which the lexical syntax does not conflict with the lexical syntax described in this report.

### 1.3.2. Error situations and unspecified behavior

When speaking of an error situation, this report uses the phrase “an error is signalled” to indicate that implementations must detect and report the error. An error is signalled by raising a non-continuable exception, as by the procedure raise as described in section [6.11]. The object raised is implementation-dependent and need not be a newly allocated object every time. In addition to errors signalled by situations described in this report, programmers may signal their own errors and handle signalled errors.

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... Such a situation is sometimes but not always, referred to with the phrase “an error.”

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

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

For example, an implementation may report a violation of an implementation restriction if it does not have enough storage to run a program or an arithmetic operation would produce an exact number that is too large for the implementation to represent.

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

Finally, the words and phrases “must,” “must not,” “shall,” “should,” “should not,” “may,” “required,” “recommended,” and “optional,” although not capitalized in this report, are to be interpreted as described in RFC 2119 [12]. In particular, “must” and “must not” are used only when absolute restrictions are placed on implementations.

### 1.3.3. Entry format

Chapters[6] and [9] 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

```plaintext
template category
```

for

```plaintext
required, primitive features, identifiers in the base library, or
```

```plaintext
library category
```

where is either “library” or “optional” library is the short name of a library as defined in section [3.3.1][Appendix A].

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

```plaintext
⟨thing1⟩ ...
```

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

```plaintext
⟨thing1⟩ ⟨thing2⟩ ...
```

indicates one or more occurrences of a ⟨thing⟩.

If category is “procedure” auxiliary syntax,” then the entry describes a syntax binding that occurs only as
part of specific surrounding expressions. Any use as an independent syntactic construct or identifier is an error.

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

\[(\text{vector-ref vector } k)\] procedure

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

\[(\text{make-vector } k)\] procedure
\[(\text{make-vector } k \text{ fill})\] procedure

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

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

- **obj** any object
- **list, list\(_1\), \ldots, list\(_j\), \ldots** list (see section 6.4)
- \(z, z_1, \ldots, z_j, \ldots\) complex number
- \(x, x_1, \ldots, x_j, \ldots\) real number
- \(y, y_1, \ldots, y_j, \ldots\) rational number
- \(q, q_1, \ldots, q_j, \ldots\) integer
- \(n, n_1, \ldots, n_j, \ldots\) exact non-negative integer
- \(k, k_1, \ldots, k_j, \ldots\) string
- **pair**
- **alist** association list (list of pairs)
- **symbol** character
- **letter** alphabetic character
- **byte** exact non-negative integer < 256
- **bytevector** procedure
- **thunk** zero-argument procedure
- **port**

### 1.3.4. Evaluation examples

The symbol “\(\rightarrow\)” used in program examples should be is read “evaluates to.” For example,

\((\ast 5 8) \rightarrow 40\)

means that the expression \((\ast 5 8)\) evaluates to the object 40. Or, more precisely: the expression given by the sequence of characters “\((\ast 5 8)\)” evaluates, in the initial environment, to an object that \text{may can} be represented externally by the sequence of characters “\(40\)” See section 3.3 for a discussion of external representations of objects.

### 1.3.5. Naming conventions

By convention, the ? is the final character of the names of procedures that always return a boolean value usually and in . Such procedures are called predicates.

By convention, the ? is the final character of the names of procedures that store values into previously allocated locations (see section 3.4). Such procedures are called mutation procedures. By convention, the The value returned by a mutation procedure is unspecified.

By convention, “\(\Rightarrow\)” appears within the names of procedures that take an object of one type and return an analogous object of another type. For example, \text{list->vector} takes a list and returns a vector whose elements are the same as those of the list.

### 2. Lexical conventions

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

Upper and lower case forms of a letter are never distinguished except within character and string constants.

### 2.1. Identifiers

An identifier is any sequence of letters, digits, and “extended identifier characters” provided that it does not have a prefix which is a valid number. However, the . token (a single period) used in the list syntax is not an identifier.

All implementations of Scheme must support the following extended identifier characters:

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

In addition, any character supported by an implementation can be used within an identifier when specified using an (inline hex escape). For example, \text{the identifier} \$\text{FOO} is the same identifier as \text{FOO}, and \text{x1AB} as the identifier \text{Hello}, and in an implementation that supports the appropriate Unicode character the identifier \text{\x3BB} is the same number as \text{x1AB} as the identifier \x.
2.2. 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. As a convenience, identifiers may also be written as a sequence of letters, digits, and “extended alphabetic characters” that begins with a character that cannot begin a number — zero or more characters enclosed within vertical bars (|), analogous to string literals. Any character, including whitespace characters, but excluding the backslash and vertical bar characters, may appear verbatim in such an identifier. In addition, all and are identifiers. It is also possible to include the backslash and vertical bar characters, as well as any other character, in the identifier with an (inline hex escape). Thus the identifier |foo bar| is the same as the identifier foo\x20;bar. Note that |l| is a valid identifier that is not equal to any other identifier.

Here are some examples of identifiers:

```
lambda
list->vector
<=>
\*
string
```

See section 7.1.1 for the formal syntax of identifiers.

Identifiers have two uses within Scheme programs:

- Any identifier may be used as a variable or as a syntactic keyword (see sections 3.1 and 4.3).
- When an identifier appears as a literal or within a literal (see section 4.1.2), it is being used to denote a symbol (see section 6.5).

In contrast with earlier revisions of the report, the syntax distinguishes between upper and lower case in identifiers and in characters specified via their names, but not in numbers, nor in (inline hex escapes) used in the syntax of identifiers, characters, or strings. None of the identifiers defined in this report contain upper-case characters, even when they may appear to do so as a result of the English-language convention of capitalizing the word at the beginning of a sentence.

The following directives give explicit control over case folding.

```
#fold-case
#no-fold-case
```

These directives may appear anywhere comments are permitted (see section 2.2) and are treated as comments, except that they affect the reading of subsequent data. The #!fold-case directive causes the read procedure to case-fold (as if by string-foldcase; see section 6.7) each identifier and character name subsequently read from the same port. The #!no-fold-case directive causes the read procedure to return to the default, non-folding behavior.

2.2. White space and comments

Whitespace characters are spaces and newline include the space and newline characters. (Implementations typically may 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 occurring inside a string, where it or inside a symbol delimited by vertical bars is significant.

```
The lexical syntax includes several comment forms. Comments are treated exactly like whitespace.
A semicolon (;) indicates the start of a line 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.
```

```
#: Another way to indicate a comment is to prefix a (datum) (cf. section 7.1.2) with #: as well as possible
(whitespace) before the (datum). The comment consists of the comment prefix #: the space, and the (datum)
together. This notation is useful for "commenting out" sections of code.
```

Block comments are indicated with properly nested #| and |
# pairs.

```
#| The FACT procedure computes the factorial
|
# (define fact
(lambda (n)
(if (= n 0)
(#| n 1 |) 1) ;Base case: return 1
(* n (fact (- n 1)))))
```

2.3. Other notations

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

. + - These are used in numbers, and may also occur anywhere in an identifier except as the first character. A
delimited plus or minus sign by itself is also an identifier. A delimited period (not occurring within a number or identifier) is used in the notation for pairs (section 6.4), and to indicate a rest-parameter in a formal parameter list (section 4.1.4). A delimited sequence of three successive periods is also. Note that a sequence of two or more periods is an identifier.

( ) Parentheses are used for grouping and to note lists (section 6.4).

' The single quote character is used to indicate literal data (section 1.1.2).

` The backquote character is used to indicate almost constant partly-constant data (section 4.2.8).

, The character comma and the sequence comma at-sign are used in conjunction with backquote (section 4.2.8).

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

\ Backslash is used in the syntax for character constants (section 6.6) and as an escape character within string constants (section 6.7) and identifiers (section 7.1.1).

[] 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 (section 6.3.1), along with the alternatives #true and #false.

#$ This introduces a character constant (section 6.6).

#( This introduces a vector constant (section 6.8). Vector constants are terminated by ).

#\ This introduces a bytevector constant (section 6.9). Bytevector constants are terminated by ) .

#e #i #b #o #d #x These are used in the notation for numbers (section 6.2.5).

#(n)= #(n)# These are used for labeling and referencing other literal data (section 2.4).

2.4. Datum labels

#(n)=(datum) lexical syntax

#(n)# lexical syntax

The lexical syntax #(n)=(datum) reads the same as (datum), but also results in (datum) being labelled by (n). It is an error if (n) is not a sequence of digits.

The lexical syntax #(n)# serves as a reference to some object labelled by #(n)#: the result is the same object as the #(n)# as compared with eqv? (see section 6.1).

Together, these syntaxes permit the notation of structures with shared or circular substructure.

(let ((x (list 'a 'b 'c))) (set!Odd b Add #b#) x)

The scope of a datum label is the portion of the datum in which it appears that is to the right of the label. Consequently, a reference #(n)# may occur only after a label #(n)#; it is an error to attempt a forward reference. In addition, it is an error if the reference appears as the labelled object itself (as in in #(n)#, the object labelled by #(n)# is not well defined in this case.

It is an error for a (program) or (library) to include circular references. In particular, it is an error for quasiquote (section 4.2.8) to contain them.

#1=(begin (display #\x) #1#)⇒ error

3. Basic concepts

3.1. Variables, syntactic keywords, and regions

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

Certain expression types are used to create new kinds of syntax and to bind syntactic keywords to those new syntaxes, while other expression types create new locations and bind variables to those locations. These expression types are called binding constructs. Those that bind syntactic keywords are listed in section 4.3. The most fundamental of the variable binding constructs is the lambda expression, because all other variable binding constructs can be explained in terms of lambda expressions. The other variable binding constructs are let, let*, letrec, letrec*, let-values, let-values, and do expressions (see sections 4.1.4, 1.2.2, and 1.2.4).

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

3.2. Disjointness of types

No object satisfies more than one of the following predicates:

- boolean?
- symbol?
- char?
- vector?
- bytevector?
- pair?
- string?
- number?
- port?
- procedure?
- null?

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

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

3.3. External representations

An important concept in Scheme (and Lisp) is that of the external representation of an object as a sequence of characters. For example, an external representation of the integer 28 is the sequence of characters “28” and an external representation of a list consisting of the integers 8 and 13 is the sequence of characters “(8 13)”.

The external representation of an object is not necessarily unique. The integer 28 also has representations “#e28.000” and “#x1c” and the list in the previous paragraph also has the representations “(08 13)” and “(8 . (13 . ()))” (see section 6.4).

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

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

External representations can also be used for input and output. The procedure read (section 6.13.2) parses external representations, and the procedure write (section 6.13.3) 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 is not always obvious out of context whether a given sequence of characters is intended to denote data or program, but it is also a source of power, since it facilitates writing programs such as interpreters and compilers that treat programs as data (or vice versa).

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

3.4. Storage model

Variables and objects such as pairs, vectors, and strings implicitly denote locations or sequences of locations. A string, for example, denotes as many locations as there are characters in the string. (Those locations need not correspond to a full machine word.) A new value may can be stored into one of these locations using the string-set! procedure, but the string continues to denote the same locations as before.

An object fetched from a location, by a variable reference or by a procedure such as car, vector-ref, or string-ref, is equivalent in the sense of eqv? (section 6.1) to the object last stored in the location before the fetch.

Every location is marked to show whether it is in use. No variable or object ever refers to a location that is not in use. Whenever this report speaks of storage being allocated for a variable or object, what is meant is that an appropriate number of locations are chosen from the set of locations that are not in use, and the chosen locations are marked to indicate that they are now in use before the variable or object is made to denote them.

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

These locations should be understood as conceptual, not physical. Hence, they do not necessarily correspond to memory addresses, and even if they do, the memory address may not be constant.

Rationale: In many systems it is desirable for constants (i.e., the values of literal expressions) to reside in read-only memory. Making it an error to alter constants permits this implementation strategy, while not requiring other systems to distinguish between mutable and immutable objects.

### 3.5. Proper tail recursion

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

Rationale:

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

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

Steele and Sussman later observed that in their interpreter the code for dealing with actors was identical to that for functions and thus there was no need to include both in the language.
With exception library given in section 7.3.

(\text{definition}) \rightarrow (\text{expression})^* (\text{tail expression})

\begin{itemize}
  \item If a \text{cond} or \text{case} expression is in a tail context, and has a clause of the form \((\text{expression}_1) \Rightarrow (\text{expression}_2)\) then the (implied) call to the procedure that results from the evaluation of \((\text{expression}_2)\) is in a tail context. \((\text{expression}_2)\) itself is not in a tail context.
  \item \textbf{Note that} \((\text{cond clause})\)’s \texttt{appear in guard expressions as well as cond expressions.}
\end{itemize}

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

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

\begin{verbatim}
(lambda ()
  (if (g)
    (let ((x (h)))
      x)
    (and (g) (f))))
\end{verbatim}

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

4. Expressions

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

The procedures \texttt{force}, \texttt{eager}, and \texttt{make-parameter} are also described in this chapter, because they are intimately associated with the \texttt{delay}, \texttt{lazy}, and \texttt{parameterize} expression types.

4.1. Primitive expression types

4.1.1. Variable references

\begin{verbatim}
<quote (datum)>
\end{verbatim}

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

\begin{verbatim}
(define x 28)
x \quad \Rightarrow \quad 28
\end{verbatim}

4.1.2. Literal expressions

\begin{verbatim}
<quote (quote a) (quote #((a b c)) (quote (+ 1 2)))
\end{verbatim}

\begin{verbatim}
\Rightarrow a \quad \Rightarrow #((a b c)) \quad \Rightarrow (+ 1 2)
\end{verbatim}

\begin{verbatim}
<quote (quote (quote a)) (quote "a")
\end{verbatim}

\begin{verbatim}
\Rightarrow (quote a) \quad \Rightarrow "a"
\end{verbatim}

\textit{Note:} \texttt{quote} is a macro to itself, \texttt{\#(quote a)} is a macro to \texttt{quote}, and boolean constants evaluate \text{true} to \text{false} themselves; they need not be quoted.

\begin{verbatim}
'\text{a} \quad \Rightarrow \text{a} \quad \Rightarrow \text{quote a}
'\text{#((a b c))} \quad \Rightarrow \text{#((a b c))} \quad \Rightarrow \text{quote #((a b c))}
'\text{(+ 1 2)} \quad \Rightarrow \text{(+ 1 2)}
'\text{(quote a)} \quad \Rightarrow \text{(quote a)}
'\text{"a} \quad \Rightarrow \text{"a} \quad \Rightarrow \text{quote "a}
\end{verbatim}

\texttt{quote} may be abbreviated as \texttt{\#(quote datum)}. The two notations are equivalent in all respects.

\begin{verbatim}
'\text{"abc} \quad \Rightarrow \text{"abc} \quad \Rightarrow \text{quote "abc}
"\text{abc} \quad \Rightarrow \text{"abc} \quad \Rightarrow \text{quote "abc}
'\text{145932} \quad \Rightarrow \text{145932} \quad \Rightarrow \text{quote 145932}
145932 \quad \Rightarrow \text{145932} \quad \Rightarrow \text{quote 145932}
'\text{#t} \quad \Rightarrow \text{#t} \quad \Rightarrow \text{quote #t}
#t \quad \Rightarrow \text{#t} \quad \Rightarrow \text{quote #t}
\end{verbatim}

As noted in section 3.3 it is an error to alter a constant (i.e. the value of a literal expression) using a mutation procedure like \texttt{set-car!} or \texttt{string-set!}.  

\begin{verbatim}
\end{verbatim}
4.1.3. Procedure calls

\[ ((\text{operator}) \ (\text{operand}_1) \ldots) \]

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 unordered sense) and the resulting procedure is passed the resulting arguments.

\begin{align*}
    (+ 3 4) & \implies 7 \\
    ((\text{if} \ #f + *) 3 4) & \implies 12
\end{align*}

A number of procedures are available as the values of variables in the initial environment; for example, the addition and multiplication procedures in the above examples are the values of the variables + and *. New procedures are created by evaluating lambda expressions (see section 4.1.4).

Procedure calls may return any number of values (see values in section 6.10). With the exception of the procedures defined in this report return one value or, for procedures such as apply, pass on the values returned by a call to one of their arguments. Exceptions are noted in the individual descriptions.

Procedure calls are also called combinations.

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

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

Note: In many dialects of Lisp, the empty combination expression, (), is a legitimate expression evaluating to itself. In Scheme, combinations must have at least one subexpression, so () is not a syntactically valid expression—it is an error.

4.1.4. Procedures

\[ (\text{lambda} \ (\text{formals}) \ (\text{body})) \]

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

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

\[ (\text{lambda} \ (x) \ (+ x x)) \implies \text{a procedure} \]
\[ ((\text{lambda} \ (x) \ (+ x x)) \ 4) \implies 8 \]
\[ (\text{define} \ \text{reverse-subtract} \]
\[ (\lambda (x y) \ (- y x))) \]
\[ \langle\text{reverse-subtract} \ 7 \ 10\rangle \implies 3 \]
\[ (\text{define} \ \text{add4} \]
\[ (\text{let} \ ((x 4)) \]
\[ (\lambda (y) \ (+ x y)))) \]
\[ \langle\text{add4} \ 6\rangle \implies 10 \]

Formals should have one of the following forms:

- \((\text{variable}_1) \ldots\) : The procedure takes a fixed number of arguments; when the procedure is called, the arguments will be stored in the bindings of the newly allocated locations that are bound to the corresponding variables.

- \((\text{variable})\) : 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 a newly allocated location that is bound to (variable).

- \((\text{variable}_1) \ldots \text{variable}_n \ldots \text{variable}_{n+1})\) : If a space-delimited period precedes the last variable, then the procedure takes \(n\) or more arguments, where \(n\) is the number of formal arguments before the period (there must be \text{it is an error if there is not} at least one). The value stored in the binding of the last variable will be a newly allocated list of the actual arguments left over after all the other actual arguments have been matched up against the other formal arguments.

It is an error for a (variable) to appear more than once in (formals).

\[ ((\text{lambda} \ x \ x) \ 3 \ 4 \ 5 \ 6) \implies (3 \ 4 \ 5 \ 6) \]
\[ ((\text{lambda} \ (x \ y \ z) \ z) \]
\[ 3 \ 4 \ 5 \ 6) \implies (5 \ 6) \]

Each procedure created as the result of evaluating a is (conceptually) tagged with a storage location, in order to make work on procedures (see section 5.1).

4.1.5. Conditionals

\[ (\text{if} \ (\text{test}) \ (\text{consequent}) \ (\text{alternate})) \]
\[ (\text{if} \ (\text{test}) \ (\text{consequent})) \]

Syntax: (Test), (consequent), and (alternate) may be arbitrary expressions.
Semantics: An if expression is evaluated as follows: first, (test) is evaluated. If it yields a true value (see section 6.3.1), then (consequent) is evaluated and its value(s) are values are returned. Otherwise (alternate) is evaluated and its value(s) are values are returned. If (test) yields a false value and no (alternate) is specified, then the result of the expression is unspecified.

\[
\begin{align*}
\text{if} &\quad (>\ 3\ 2)\ 'yes\ 'no) &\quad \Rightarrow &\quad yes \\
\text{if} &\quad (>\ 2\ 3)\ 'yes\ 'no) &\quad \Rightarrow &\quad no \\
\text{if} &\quad (>\ 3\ 2) \\
&\quad (-\ 3\ 2) \\
&\quad (+\ 3\ 2)) &\quad \Rightarrow &\quad 1
\end{align*}
\]

4.1.6. Assignments

(set! (variable) (expression)) syntax

The (expression) is evaluated, and the resulting value is stored in the location to which (variable) is bound. must be It is an error if (variable) is not bound either in some region enclosing the set! expression or at top level. The result of the set! expression is unspecified.

\[
\begin{align*}
\text{define } &\quad x\ 2 \\
&\quad (+\ x\ 1) &\quad \Rightarrow &\quad 3 \\
\text{set! } &\quad x\ 4 \\
&\quad (+\ x\ 1) &\quad \Rightarrow &\quad unspecified
\end{align*}
\]

4.2. Derived expression types

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

4.2.1. Conditionals

(cond (clause1) (clause2) ...) syntax
else auxiliary syntax
=> auxiliary syntax

Syntax: Each should be of the form (Clauses) take one of two forms, either:

\[
\begin{align*}
\text{(test) } &\quad (\text{expression1} \ldots) \\
\text{(test} &\quad (\text{expression1} \ldots)
\end{align*}
\]

where (test) is any expression. Alternatively, a may be of the form or

\[
\begin{align*}
\text{(test} &\quad (\text{expression1})
\end{align*}
\]

The last (clause) may be an “else clause,” which has the form

\[
\begin{align*}
\text{else} &\quad (\text{expression1} \ldots)
\end{align*}
\]

Semantics: A cond expression is evaluated by evaluating the (test) expressions of successive (clause)s in order until one of them evaluates to a true value (see section 6.3.1). When a (test) evaluates to a true value, then the remaining (expression)s in its (clause) are evaluated in order, and the result(s) of the last (expression) in the (clause) are values are returned as the result(s) of the entire cond expression.

If the selected (clause) contains only the (test) and no (expression)s, then the value of the (test) is returned as the result. If the selected (clause) uses the => alternate form, then the (expression) is evaluated. Its value must be It is an error if its value is not a procedure that accepts one argument; this. This procedure is then called on the value of the (test) and the value(s) of the last one is(are) are returned.

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 in order, and the value(s) of the last one is(are) are returned.

\[
\begin{align*}
\text{(cond} &\quad (>\ 3\ 2)\ 'greater) \\
&\quad (((<\ 3\ 2)\ 'less)) \quad \Rightarrow \quad greater \\
\text{(cond} &\quad (>\ 3\ 3)\ 'greater) \\
&\quad (((<\ 3\ 3)\ 'less) \\
&\quad (else\ 'equal)) \quad \Rightarrow \quad equal \\
\text{(cond} &\quad ((\text{assv}\ b\ '((a\ 1)\ (b\ 2))\ \Rightarrow\ \text{cadr)} \\
&\quad (else\ #f)) \quad \Rightarrow \quad 2
\end{align*}
\]

case (key) (clause1) (clause2) ...} syntax

Syntax: (Key) may be any expression. Each (clause) should have the form

\[
\begin{align*}
\text{((\text{datum}) \ldots) } &\quad (\text{expression1} \ldots),
\end{align*}
\]

where each (datum) is an external representation of some object. All It is an error if any of the (datum)s must be distinct—are the same anywhere in the expression. Alternatively, a (clause) may be of the form

\[
\begin{align*}
\text{((\text{datum}) \ldots) => (expression))
\end{align*}
\]

The last (clause) may be an “else clause,” which has the form of the forms

\[
\begin{align*}
\text{(else} &\quad (\text{expression1}) \ldots)
\end{align*}
\]

or

\[
\begin{align*}
\text{(else } &\quad (\text{expression1}))
\end{align*}
\]

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

If the selected (clause) or else clause uses the => alternate form, then the (expression) is evaluated. It is an error if its value is not a procedure accepting one argument. This procedure is then called on the value of the (key) and the values returned by this procedure are returned by the case expression.

(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)
  ((v y) 'semivowel)
  (else 'consonant)) => composite

(and (test_1) ...)

The (test) expressions are evaluated from left to right, and the value of the first expression that evaluates to a false value if any expression evaluates to #f (see section 6.3.1). #f 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) ...)

The (test) expressions are evaluated from left to right, and the value of the first expression that evaluates to a true value (see section 6.3.1) is returned. Any remaining expressions are not evaluated. If all expressions evaluate to false values, the value of the last expression is returned. If #f or 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)

(let (bindings) (body))

Syntax: (Bindings) should have the form

(((variable_1) (init_1)) ...),

where each (init) is an expression, and (body) should be a sequence of zero or more definitions followed by a sequence of one or more expressions as described in section 4.1.4. It is an error for a (variable) to appear more than once in the list of variables being bound.

Semantics: The (init)s are evaluated in the current environment (in some unspecified order), the (variable)s are bound to fresh locations holding the results, the (body) is evaluated in the extended environment, and the value(s) of the last expression of (body) is(are) returned.

Each binding of a (variable) has (body) as its region.

(let ((x 2) (y 3))
  (* x y)) => 6
(let ((x 2) (y 3))
  (let ((x 7)
        (z (+ x y)))
      (* z x))) => 35

4.2.2. Binding constructs

The three binding constructs let, let*, and letrec, letrec let-values, and let-values give Scheme a block structure, like Algol 60. The syntax of the three first four constructs is identical, but they differ in the regions they establish for their variable bindings. In a let expression, the initial values are computed before any of the variables become bound; in a let* expression, the bindings and evaluations are performed sequentially; while in a letrec expression and letrec* expressions, all the bindings are in effect while their initial values are being computed, thus allowing mutually recursive definitions. let-values and let*-values are analogous to let and let* respectively, but are designed to handle multiple-valued expressions, binding different identifiers to each returned value.
See also “named let,” section 4.2.4.

(let* (bindings) (body)) syntax

Syntax: (Bindings) should have the form

(((variable) (init1)) ...),

and (body) should be a sequence of zero or more definitions followed by one or more expressions as described in section 4.1.4.

Semantics: The let* binding construct 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. The (variable)s need not be distinct.

(let ((x 2) (y 3))
  (let* ((x 7)
         (y (+ x y)))
         (* z x)))  =>  70

(letrec (bindings) (body)) syntax

Syntax: (Bindings) should have the form

(((variable) (init1)) ...),

and (body) should be a sequence of zero or more definitions followed by one or more expressions as described in section 4.1.4. 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, each (variable) is assigned in left-to-right order to the result of evaluating the corresponding (init), the (body) is evaluated in the resulting environment, and the values of the last expression in (body) are returned. Despite the left-to-right evaluation and assignment order, each binding of a (variable) has the entire letrec expression as its region, making it possible to define mutually recursive procedures.

If it is not possible to evaluate each (init) without assigning or referring to the value of the corresponding (variable) or the (variable) of any of the bindings that follow it in (bindings), it is an error.

(letrec* ((p (lambda (x) (+ 1 (q (- x 1)))))
          (q (lambda (x) (+ 1 (r (- x 1)))))
          (r (lambda (x) (+ 1 (s (- x 1)))))
          (s (lambda (x) (+ 1 (t (- x 1))))))

(let-values (mvbindings) (body)) syntax

Syntax: (Mvbindings) should have the form

(((formals) (init1)) ...)

where each (init) should be an expression, and (body) should be zero or more definitions followed by a sequence of one or more expressions as described in section 4.1.4.

It is an error for a variable to appear more than once in the set of (formals).

Semantics: The (init)s are evaluated in the current environment (in some unspecified order) as if by invoking call-with-values, the variables occurring in the (formals) are bound to fresh locations holding the values returned by the (init)s, where the (formals) are matched to the return values in the same way that the (formals) in a lambda expression are matched to the arguments in a
procedure call. Then, the \{(body)\} is evaluated in the extended environment, and the values of the last expression of \{(body)\} are returned. Each binding of a \{(variable)\} has \{(body)\} as its region.

It is an error if the \{(formals)\} do not match the number of values returned by the corresponding \{(init)\}.

\[
\text{let-values ((}(\text{root rem}) (\text{exact-ss neg sup sqrt 32)))
\]

\[
\text{let\text{*}-values (mvbindings) \{(body)\}}
\]

Syntax: \{(Mvbindings)\} should have the form

\[
((\text{formals}) \{(\text{init})\} \ldots),
\]

and \{(body)\} should be a sequence of zero or more definitions followed by one or more expressions described in section 1.1.1. In each \{(formals)\}, it is an error if any variable appears more than once.

Semantics: \text{let-values*} is similar to \text{let-values}, but the \{(init)\}s are evaluated and bindings created sequentially from left to right, with the region of the bindings of each \{(formals)\} including the \{(init)\}s to its right as well as \{(body)\}. Thus the second \{(init)\} is evaluated in an environment in which the first set of bindings is visible and initialized, and so on.

\[
\text{let ((a 'a) (b 'b) (x 'x) (y 'y) x y) key-values ((a b) values \varepsilon 0) (\text{list a b}) (\text{list x y})}
\]

### 4.2.3. Sequencing

Both of Scheme’s sequencing constructs are named begin, but the two have slightly different forms and uses:

\[
\text{begin \{(expression or definition) \ldots\}}
\]

Syntax

This form of \text{begin} may appear as part of a \{(body)\}, or at the (top-level), or directly nested in a \text{begin} that is itself of this form. It causes the contained expressions and definitions to be evaluated exactly as if the enclosing begin construct were not present.

Rationale: This form is commonly used in the output of macros (see section 1.3), which need to generate multiple definitions and splice them into the context in which they are expanded.

\[
\text{begin \{(expression\_1\} \{(expression\_2\} \ldots\}}
\]

Syntax

This form of \text{begin} can be used as an ordinary expression. The \{(expression)\}s are evaluated sequentially from left to right, and the value(s) values of the last \{(expression)\} before a \text{let-values} are returned. This expression type is used to sequence side effects such as assignments or input and output.

\[
\text{define x 0}
\]

\[
\text{begin (set! x 5)}
\]

\[
\text{begin (display "4 plus 1 equals ")}
\]

\[
\text{display (+ 4 1))}
\]

4.2.4. Iteration

\[
\text{do (}(\text{\{(variable\_1\} \{(init\_1\} \{(step\_1\)\} ...})
\]

\[
\text{\{(test\} \{(expression\) \ldots\})
\]

\[
\text{\{(command\} \ldots\)}
\]

A \text{do expression} is an iteration construct. It specifies a set of variables to be bound, how they are to be initialized at the start, and how they are to be updated on each iteration. When a termination condition is met, the loop exits after evaluating the \{(expression)\}s.

A \text{do expression} is evaluated as follows: The \{(init)\} expressions are evaluated (in some unspecified order), the \{(variable)\}s are bound to fresh locations, the results of the \{(init)\} expressions are stored in the bindings of the \{(variable)\}s, and then the iteration phase begins.

Each iteration begins by evaluating \{(test)\}; if the result is false (see section 6.3.1), then the \{(command)\} expressions are evaluated in order for effect, the \{(step)\} expressions are evaluated in some unspecified order, the \{(variable)\}s are bound to fresh locations, the results of the \{(step)\} expressions are stored in the bindings of the \{(variable)\}s, and the next iteration begins.

If \{(test)\} evaluates to a true value, then the \{(expression)\} expressions are evaluated from left to right and the value(s) values of the last \{(expression)\} \text{are} returned. If no \{(expression)\}s are present, then the value of the \text{do expression} is unspecified.

The region of the binding of a \{(variable)\} consists of the entire \text{do expression} except for the \{(init)\}s. It is an error for a \{(variable)\} to appear more than once in the list of \text{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)\)\}.

\[
\text{do (}(\text{\{(vec make-vector 5)\} \{(i 0 (i i))\} \{(= i 5) vec\)\} \{(vec set! vec i i)\} \Rightarrow \text{#(0 1 2 3 4)}}
\]

\[
\text{let ((x '(1 3 5 7 9)))}
\]

\[
\text{do (}(\text{x x (cdr x))\} \{(sum 0 (+ sum (car x)))\} \{(null? x) sum\})\} \Rightarrow \text{25}}
\]

\[
\text{let \{(variable \{(bindings) \{(body)\)\} \}}
\]

Syntax

“Named let” is a variant on the syntax of \text{let} which provides a more general looping construct than \text{do} and \text{let-values} can also be used to express recursions. It has the same syntax and semantics as ordinary \text{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 \(\text{defun} (\text{body})\) can be repeated by
invoking the procedure named by \(\text{defun} (\text{variable})\).

\[
\begin{align*}
\text{(let loop } ((\text{numbers} '(3 -2 1 6 -5)) \\
\quad (\text{nonneg} '()) \\
\quad (\text{neg} '())) \\
\text{(cond } ((\text{null? numbers}) (\text{list nonneg neg})) \\
\quad ((>= (\text{car numbers}) 0) \\
\quad (\text{loop } (\text{cdr numbers}) \\
\quad (\text{cons } (\text{car numbers}) \text{ nonneg} \\
\quad \text{neg})))) \\
\quad ((< (\text{car numbers}) 0) \\
\quad (\text{loop } (\text{cdr numbers}) \\
\quad (\text{cons } (\text{car numbers}) \text{ neg})))) \\
\Rightarrow ((6 1 3) (-5 -2))
\end{align*}
\]

4.2.5. Delayed evaluation

\[\text{(delay } \langle \text{expression} \rangle)\] lazy library syntax

The \text{delay} construct is used together with the procedure \text{force} to implement lazy evaluation or call by need. \(\text{delay} \langle \text{expression} \rangle\) returns an object called a promise which at some point in the future may be asked (by the \text{force} procedure) to evaluate \(\langle \text{expression} \rangle\), and deliver the resulting value. The effect of \(\langle \text{expression} \rangle\) returning multiple values is unspecified.

\[\text{See the description of}\]

\[\text{(lazy } \langle \text{expression} \rangle)\] lazy library syntax

The \text{lazy} construct is similar to \text{delay}, but it is an error for its argument not to evaluate to a promise. The returned promise, when forced, will evaluate to whatever the original promise would have evaluated to if it had been forced.

\[\text{(force } \langle \text{promise} \rangle)\] lazy library procedure

The \text{force} procedure forces the value of a more complete description \langle \text{promise} \rangle created by \text{delay} or \text{lazy}. If no value has been computed for the promise, then a value is computed and returned. The value of the promise is cached (or “memoized”) so that if it is forced a second time, the previously computed value is returned.

\[
\begin{align*}
\text{(force } (\text{delay } (+ 1 2))) & \Rightarrow 3 \\
\text{(let } ((p (\text{delay } (+ 1 2)))) \\
\quad (\text{list } (\text{force } p)) \text{ (force } p)) & \Rightarrow (3 3)
\end{align*}
\]

\[
\begin{align*}
\text{(define integers } (\text{letrec } ((\text{next} \\
\quad (\text{lambda } (n) (\text{list } (\text{cons } n n)))) \\
\quad (\text{head } (\text{tail } (\text{tail integers})))))) & \Rightarrow 2
\end{align*}
\]

The following example is a mechanical transformation of a lazy stream-filtering algorithm into Scheme. Each call to a constructor is wrapped in \text{delay}, and each argument passed to a deconstructor is wrapped in \text{force}. The use of \(\text{lazy} \ldots\) instead of \(\text{delay} \text{ (force} \ldots\)) around the body of the procedure ensures that an ever-growing sequence of pending promises does not exhaust the heap.

\[
\begin{align*}
\text{(define (stream-filter p? s) } (\text{lazy } \text{if } (\text{null? } (\text{force } s)))) \\
\text{(head } (\text{tail } (\text{tail } (\text{stream-filter5odd? integers}))))
\end{align*}
\]

The following examples are not intended to illustrate good programming style, as \text{delay}, \text{lazy}, and \text{force} are mainly intended for programs written in the functional style. However, they do illustrate the property that only one value is computed for a promise, no matter how many times it is forced.

\[
\begin{align*}
\text{(define count 0)(define p } (\text{delay } (\text{begin } (\text{set! count } (+ \text{count 1})) \\
\quad \text{p})) \Rightarrow 6 \\
\text{p } \Rightarrow \text{a promise, still} \\
\text{(begin } (\text{set! x 10}) \Rightarrow (\text{force } p))
\end{align*}
\]

Various extensions to this semantics of \text{delay}, \text{force} and \text{lazy} are supported in some implementations:

- Calling \text{force} on an object that is not a promise may simply return the object.
- It may be the case that there is no means by which a promise can be operationally distinguished from its forced value. That is, expressions like the following may evaluate to either \#t or \#f, depending on the implementation:

\[
\begin{align*}
\text{(eqv? } (\text{delay } 1) \Rightarrow \text{unspecified} \\
\text{(pair? } (\text{delay } (\text{cons } 1 2))) & \Rightarrow \text{unspecified}
\end{align*}
\]

- Some implementations may implement “implicit forcing,” where the value of a promise is forced by primitive procedures like \text{cdr} and \text{+}:

\[
\begin{align*}
\text{(+ } (\text{delay } (* 3 7)) 13) & \Rightarrow 34 \\
\text{(eager } \text{obj}) & \text{lazy library procedure}
\end{align*}
\]

The \text{eager} procedure \text{force} on a promise which has been forced will return \text{obj}. It is similar to \text{delay}, but does not delay its argument: it is a procedure rather than syntax.
4.2.6. Dynamic bindings

(make-parameter init) procedure
(make-parameter init converter) procedure

Returns a newly allocated parameter object, which is a procedure that accepts zero arguments and returns the value associated with the parameter object. Initially, this value is the value of (converter init), or of init if the conversion procedure converter is not specified. The associated value can be temporarily changed using parameterize, which is described below.

The effect of passing arguments to a parameter object is implementation-dependent.

(parameterize (⟨(param1) ⟨value1⟩) ...) syntax
⟨body⟩)

A parameterize expression is used to change the values returned by specified parameter objects during the evaluation of the body. It is an error if the value of any ⟨param⟩ expression is not a parameter object. The ⟨param⟩ and ⟨value⟩ expressions are evaluated in an unspecified order. The ⟨body⟩ is evaluated in a dynamic environment in which calls to the parameters return the results of passing the corresponding values to the conversion procedure specified when the parameters were created. Then the previous values of the parameters are restored without passing them to the conversion procedure. The results of the last expression in the ⟨body⟩ are returned as the results of the entire parameterize expression.

Note: If the conversion procedure is not idempotent, the results of (parameterize ((x (x))) ...), which appears to bind the parameter x to its current value, may not be what the user expects.

If an implementation supports multiple threads of execution, then parameterize must not change the associated values of any parameters in any thread other than the current thread or threads created inside ⟨body⟩.

Parameter objects can be used to specify configurable settings for a computation without the need to pass the value to every procedure in the call chain explicitly.

(define radix (make-parameter 10 (lambda (x)
  (+ x 2))))

(define (f n) (number->string n (radix)) (if (and (integer? x) (< 2 x 16)) (+ x 2) (error "invalid \}))

(f 12) ⇒ "12"
(parameterize ((f (radix 2)))
  (f 12)) ⇒ "1100"
  (f 12) ⇒ "12"
  (radix 16) ⇒ unspecified

(parameterize ((f (radix 0)) (f 12))

4.2.7. Exception Handling

(guard ⟨(variable)⟩ syntax
  ⟨(cond clause1) ⟨(cond clause2) ...)⟩
  ⟨(body)⟩)

Syntax: Each ⟨(cond clause)⟩ is as in the specification of cond.

Semantics: The ⟨(body)⟩ is evaluated with an exception handler that binds the raised object to ⟨(variable)⟩ and, within the scope of that binding, evaluates the clauses as if they were the clauses of a cond expression. That implicit cond expression is evaluated with the continuation and dynamic environment of the guard expression. If every ⟨(cond clause)⟩'s (test) evaluates to #f and there is no else clause, then raise-continuable is re-invoked on the raised object within the dynamic environment of the original call to raise except that the current exception handler is that of the guard expression.

See section 6.11 for a more complete discussion of exceptions.

4.2.8. Quasiquotation

(quasiquote ⟨qq template⟩) syntax
'⟨qq template⟩ syntax
unquote auxiliary syntax
unquote-splicing auxiliary syntax

"Backquote" or "quasiquote" expressions are useful for constructing a list or vector structure when most some but not all of the desired structure is known in advance. If no commas appear within the ⟨qq template⟩, the result of evaluating '⟨qq template⟩ is equivalent to the result of evaluating "⟨qq template⟩. If a comma appears within the ⟨qq template⟩, however, the expression following the comma is evaluated ("unquoted") and its result is inserted into the structure instead of the comma and the expression. If a comma appears followed immediately by an at-sign (@), then it is an error if the following expression does not evaluate to a list; the opening and closing parentheses of the list are then "stripped away" and the elements of the list are inserted in place of the comma at-sign expression sequence. A comma at-sign should only appear within a list or vector ⟨qq template⟩.

'(list ,(+ 1 2) 4) ⇒ (list 3 4)
(list ,(+ 1 2) ,@map abs (+ -5 6) b)
⇒ (a 3 4 5 6 b)
'(list ,((foo ,(+ 10 5)) ,@map sqrt '4 8))
⇒ #10 5 2 4 3 8
Quasiquotes form expressions 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)
```

A quasiquoted expression may return either fresh, mutable objects or literal structure for any structure that is constructed at run time during the evaluation of the expression. Portions that do not need to be rebuilt are always literal. Thus,

```
(let ((a 3)) `(1 2) ,a ,4 ,five 6))
```

may be equivalent to either of the following expressions:

```
`((1 2) 3 4 five 6)
(let ((a 3)) (cons '1 (2) (cons a (cons 4 (cons five '(6))))))
```

However, it is not equivalent to this expression:

```
(let ((a 3)) (list (list 1 2) a 4 'five 6))
```

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

```
(quasiquote (list (unquote (+ 1 2)) 4))
⇒ (list 3 4)
'(quasiquote (list (unquote (+ 1 2)) 4))
⇒ '(list ,(+ 1 2) 4)
i.e., `(quasiquote (list (unquote (+ 1 2)) 4))
```

Unpredictable behavior can result. It is an error if any of the symbols `⟩ or identifiers quasiquote, unquote, or unquote-splicing appear in positions within a ⟨qq template⟩ otherwise than as described above.

### 4.2.9. Case-lambda

```
(case-lambda (clause₁) (clause₂) ...)
```

case-lambda library syntax

**Syntax:** Each clause should be of the form ⟨(formals) (body)⟩, where ⟨formals⟩ and ⟨body⟩ have the same syntax as in a lambda expression.

**Semantics:** A case-lambda expression evaluates to a procedure that accepts a variable number of arguments and is lexically scoped in the same manner as a procedure resulting from a lambda expression. When the procedure is called, the first ⟨clause⟩ for which the arguments agree with ⟨formals⟩ is selected, where agreement is specified as for the ⟨formals⟩ of a lambda expression. The variables of ⟨formals⟩ are bound to fresh locations, the values of the arguments are stored in those locations, the ⟨body⟩ is evaluated in the extended environment, and the results of ⟨body⟩ are returned as the results of the procedure call.

It is an error for the arguments not to agree with the ⟨formals⟩ of any ⟨clause⟩.

```
(define range (case-lambda ((e) (range 0 e)) ((b e) (do (range b) e))))
```

```
(range 3) ⇒ (0 1 2)
(range 3 5) ⇒ (3 4)
```

### 4.3. Macros

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

```
⟨(keyword) ⟨datum⟩ ...⟩
```

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

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

The macro definition facility consists of two parts:

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

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

- If a macro transformer inserts a binding for an identifier (variable or keyword), the identifier will in effect be renamed throughout its scope to avoid conflicts with other identifiers. Note that a define at top level may or may not introduce a binding; see section 5.3.
• If a macro transformer inserts a free reference to an identifier, the reference refers to the binding that was visible where the transformer was specified, regardless of any local bindings that may surround the use of the macro.

In consequence, all macros defined using the pattern language are “hygienic” and “referentially transparent” and thus preserve Scheme’s lexical scoping.

4.3.1. Binding constructs for syntactic keywords

The `let-syntax` and `letrec-syntax` binding constructs are analogous to `let` and `letrec`, but they bind syntactic keywords to macro transformers instead of binding variables to locations that contain values. Syntactic keywords may also be bound at top level or elsewhere with `define-syntax`; see section 4.3.

```scheme
(let-syntax (bindings) (body))
```

Syntax: `(bindings)` should have the form

```scheme
((keyword) (transformer spec) ...)
```

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

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

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

4.3.2. Pattern language

A `(transformer spec)` has the following forms—one of the following forms:

```scheme
(syntax-rules ((literal) ...) syntax)
(syntax-rules (ellipsis) ((literal) ...) syntax)
```

- auxiliary syntax
- auxiliary syntax

Syntax: is a list of identifiers and each `it is an error if any of the `(literal)`s, or the `(ellipsis)` in the second form, is not an identifier. It is also an error if `(syntax rule)` should be is not of the form

```scheme
((pattern) (template))
```

The `(pattern)` in a `(syntax rule)` is a list `(pattern)` that begins with the keyword for the macro whose first element is an identifier.

A `(pattern)` is either an identifier, a constant, or one of the following

```scheme
((pattern) ...)
((pattern) (pattern) ... . (pattern))
((pattern) . (pattern) (ellipsis) . (pattern) ... . (pattern))
((pattern) ... (pattern) (pattern) ... . (pattern) ... . (pattern) ... . (pattern))
```

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

```scheme
((element) ...)
((element) (element) ... . (template))
((ellipsis) (template)) #((element) ...)
```
where an (element) is a (template) optionally followed by an (ellipsis) and an An (ellipsis) is the identifier specified in the second form of syntax-rules, or the default identifier ... (which cannot be used as an identifier in either a template or a pattern) (three consecutive periods) otherwise.

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

An identifier that appears in the pattern of a is a appearing within a (pattern) may be an underscore (SUBSCRIPTNB), a literal identifier listed in the list of (literal)s, or the (ellipsis). All other identifiers appearing in a (pattern) are pattern variables unless it is the keyword that begins the pattern, is listed in, or is the identifier.

The keyword at the beginning of the pattern in a (syntax rule) is not involved in the matching and is considered neither a pattern variable nor a literal identifier.

Pattern variables match arbitrary input elements and are used to refer to elements of the input in the template. It is an error for the same pattern variable to appear more than once in a (pattern).

The keyword at the beginning of the pattern in a is not involved in the matching and is not considered a pattern variable or literal identifier. Underscores also match arbitrary input elements but are not pattern variables and so cannot be used to refer to those elements. If an underscore appears in the (literal)s list, then that takes precedence and underscores in the (pattern) match as literals. Multiple underscores may appear in a (pattern).

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

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

A subpattern followed by (ellipsis) can match zero or more elements of the input. It is an error for to appear in ... Within a pattern the identifier ... must follow the last element of a nonempty sequence of subpatterns, unless (ellipsis) appears in the (literal)s in which case it is matched as a literal.

More formally, an input form \( F \) matches a pattern \( P \) if and only if:

- \( P \) is an underscore (SUBSCRIPTNB);
- \( P \) is a non-literal identifier; or
- \( P \) is a literal identifier and \( F \) is an identifier with the same binding; or
- \( P \) is a list \( (P_1 \ldots P_n) \) and \( F \) is a list of \( n \) forms elements that match \( P_1 \) through \( P_n \), respectively; or
- \( P \) is an improper list \( (P_1 P_2 \ldots P_n P_{n+1}) \) and \( F \) is a list or improper list of \( n \) or more forms elements that match \( P_1 \) through \( P_n \), respectively, and whose \( n \)th cdr tail matches \( P_{n+1} \); or
- \( P \) is of the form \( (P_1 \ldots P_{e-1} P_e \langle \text{ellipsis} \rangle P_{m+1} \ldots P_n) \) where \( F \) matches \( P_{e-1} \) is an list or improper list of \( e \) elements, the first \( e - 1 \) of which match \( P_1 \) through \( P_{e-2} \), whose next \( m - k \) elements match each \( P_k \), whose remaining \( n - m \) elements match \( P_{m+1} \) through \( P_n \); or
- \( P \) is a vector of the form \( #(P_1 \ldots P_n) \) and \( F \) is a vector of \( n \) forms elements that match \( P_1 \) through \( P_n \); or
- \( P \) is of the form \( #(P_1 \ldots P_{e-1} P_e \langle \text{ellipsis} \rangle P_{m+1} \ldots P_n) \) where \( F \) matches \( P_{e-1} \) is an list or improper list of \( e \) elements, the first \( e - 1 \) of which match \( P_1 \) through \( P_{e-2} \), whose next \( m - k \) elements match each \( P_k \), whose remaining \( n - m \) elements match \( P_{m+1} \) through \( P_n \); or
- \( P \) is a datum constant and \( F \) is equal to \( P \) in the sense of the equal? procedure.

It is an error to use a macro keyword, within the scope of its binding, in an expression that does not match any of the patterns.
When a macro use is transcribed according to the template of the matching (syntax rule), pattern variables that occur in the template are replaced by the subforms they match in the input. Pattern variables that occur in subpatterns followed by one or more instances of the identifier \(\ldots\) (ellipsis) are allowed only in subtemplates that are followed by as many instances of \(\ldots\) (ellipsis). They are replaced in the output by all of the subforms they match in the input, distributed as indicated. It is an error if the output cannot be built up as specified.

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

A template of the form \(\ldots\) (template) is identical to (template), except that ellipses within the template have no special meaning. That is, any ellipses contained within (template) are treated as ordinary identifiers. In particular, the template \(\ldots\) (ellipsis) (ellipsis) produces a single (ellipsis). This allows syntactic abstractions to expand into code containing ellipses.

\[
\textbf{(define-syntact rules () ((be-like-begin name) (define-syntact name (syntax-rules ()) (be-like-begin sequence) (sequence 1 2 3 4))}
\]

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

\[
\text{let ((=} #f)}
\text{ (cond (#t to 'ok)))} \implies \text{ok}
\]

The macro transformer for \(\text{cond}\) recognizes \(\to\) as a local variable, and hence an expression, and not as the top-level identifier \(\to\), which the macro transformer treats as a syntactic keyword. Thus the example expands into

\[
\text{let ((=} #f)}
\text{ (if #t (begin to 'ok)))}
\]

instead of

\[
\text{let ((=} #f)}
\text{ (let ((temp #f))}
\text{ (if temp ('ok temp)))}
\]

which would result in an invalid procedure call.

### 4.3.3. Signalling errors in macro transformers

\[
\textbf{(syntax-error (message) (args) \ldots) syntax}
\]
5. Program structure

5.1. Programs

A Scheme program consists of a sequence of program parts: expressions, definitions, and syntax definitions. A collection of program parts may be encapsulated in a library to be reused by multiple programs. Expressions are described in Chapter 4; definitions and syntax definitions, the other program parts, as well as libraries, are the subject of the rest of the present chapter.

Programs and libraries are typically stored in files; although programs can be entered interactively to a running Scheme system, although and other paradigms are possible (questions of user interface lie outside the scope of this report). (Indeed, Scheme would still be useful as a notation for expressing computational methods even in the absence of a mechanical implementation.) Implementations which store libraries in files should document the mapping from the name of a library to its location in the file system.

Definitions and syntax definitions occurring Program parts other than expressions that are present at the top level of a program can be interpreted declaratively. They cause bindings to be created in the top level environment or modify the value of existing top-level bindings. The initial (or "top level") Scheme environment is empty except for import, so further bindings can only be introduced with import.

Expressions occurring at the top level of a program are interpreted imperatively; they are executed in order when the program is invoked or loaded, and typically perform some kind of initialization.

At the top level of a program (begin ⟨form⟩ ...) is equivalent to the sequence of expressions, definitions, and syntax definitions that form the body of the in the begin. Macros can expand into such begins.

Implementations may provide an interactive session called a REPL (Read-Eval-Print Loop), where Scheme program parts can be entered and evaluated one at a time. For convenience and ease of use, the "top-level" Scheme environment in a REPL must not be empty, but must start with a number of variables bound to locations containing at least the bindings provided by the base library. This library includes the core syntax of Scheme and generally useful procedures that manipulate data. For example, the variable abs is bound to a procedure of one argument that computes the absolute value of a number, and the variable + is bound to a procedure that computes sums. The full list of (scheme base) bindings can be found in Appendix A.

5.2. Definitions

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

A definition should have In a (body) (begin (definition) ...) is equivalent to the sequence of definitions ⟨definition⟩ .... Macros can expand into such begins.

A definition takes one of the following forms:

- (define (variable) (expression))
- (define ((variable) (formals)) (body))
- (define ((variable) . (formal)) (body))

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

(define (variable)
    (lambda ((formals)) (body))).

5.2.1. Top level definitions

At the top level of a program, a definition

(define (variable) (expression))

has essentially the same effect as the assignment expression

(set! (variable) (expression))

if (variable) is bound to a non-syntactic value. However, if (variable) is not bound, however, or is bound to a syntax definition (see below), then the definition will bind (variable) to a new location before performing the assignment, whereas it would be an error to perform a set! on an unbound variable.

(define add3
    (lambda (x) (+ x 3)))

(add3 3) ⇒ 6
(define first car)
(first '(1 2)) ⇒ 1

Some implementations of Scheme may be permitted to provide an initial environment in which all possible variables are bound to locations, most of which contain undefined-unspecified values. Top level definitions in such an implementation are truly equivalent to assignments.
5.2.2. Internal definitions

Definitions may occur at the beginning of a ⟨body⟩ (that is, the body of a lambda, let, let*, letrec, letrec*, _let-values, _let-values*, _let-syntax, or letrec-syntax, or parameterize, guard, or case-lambda expression or that of a definition of an appropriate form). Such definitions are known as internal definitions as opposed to the top level definitions described above. The variable defined by an internal definition is local to the ⟨body⟩. That is, (variable) is bound rather than assigned, and the region of the binding is the entire ⟨body⟩. For example,

(let ((x 5))
  (define foo (lambda (y) (bar x y)))
  (define bar (lambda (a b) (+ (* a b) a)))
  (foo (+ x 3)))
⇒ 45

An expanded ⟨body⟩ containing internal definitions can always be converted into a completely equivalent letrecletrec* 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 letrecletrec* expression, it must be an error if it is not possible to evaluate each (expression) of every internal definition in a ⟨body⟩ without assigning or referring to the value of any the corresponding (variable) being defined for the (variable) of any of the definitions that follow it in the ⟨body⟩.

It is an error to define the same identifier more than once in the same ⟨body⟩.

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

5.2.3. Multiple-value definitions

The construct define-values introduces new definitions like define, but can create multiple definitions from a single expression returning multiple values. It is allowed wherever define is allowed.

(define-values ⟨formals⟩ ⟨expression⟩) syntax

It is an error if a variable appears more than once in the set of ⟨formals⟩.

Semantics: ⟨expression⟩ is evaluated, and the ⟨formals⟩ are bound to the return values in the same way that the ⟨formals⟩ in a lambda expression are matched to the arguments in a procedure call.

(let () (define-values (x y) (values 1 2)) (+ x y))

5.3. Syntax definitions

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

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

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

There is no, but existing references to any top-level binding for ⟨keyword⟩ remain unchanged. Otherwise, it is an error if syntaxinternal syntax definition analogue of internal definitions, and is local to the ⟨body⟩ in which it is defined.

(let ((x 1) (y 2)) (define-syntax dipp! (syntax-rules ()
  (dipp! (x y))))

Although macros may

Macros can expand into definitions and syntax definitions in any context that permits them. However, it is an error for a definition or syntax definition to shadow a syntactic keyword whose meaning is needed to determine whether some form in to define an identifier whose binding has to be known in order to determine the meaning of the group of forms that contains the shadowing definition. In fact, a definition, or, for internal definitions, is needed to definition itself, or of any preceding definition that belongs to the same group of internal definitions. Similarly, it is an error for an internal definition to define an identifier whose binding has to be known in order to determine the boundary between the group internal definitions and the expressions that follow the group of the body it belongs to. For example, the following are errors:

(define define 3)

(begin (define begin list))

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

5.4. Record type definitions

Record type definitions are used to introduce new data types, called record types. The values of a record type are called records and are aggregations of zero or more fields, each of which holds a single location. A predicate,
a constructor, and field accessors and mutators are defined for each record type. Record type definitions are valid wherever definitions are.

```
(define-record-type <pare> (kons x y) pare? (x kar set-kar!)
```

defines kons to be a constructor, kar and kdr to be accessors, set-kar! to be a modifier, and pare? to be a predicate for instances of <pare>.

```
(pare? (kons 1 2)) ;⇒ #t
(pare? (cons 1 2)) ;⇒ #f
(kar (kons 1 2)) ;⇒ 1
(kdr (kons 1 2)) ;⇒ 2
(let ((k (kons 1 2)))
  (set-kar! k 3)
  (kar k)) ;⇒ 3
```

5.5. Libraries

Libraries provide a way to organize Scheme programs into reusable parts with explicitly defined interfaces to the rest of the program. This section defines the notation and semantics for libraries.

5.5.1. Library Syntax

A library definition takes the following form:

```
(define-library (library name)
  (library declaration) ...)
```

(library name) is a list whose members are identifiers or unsigned exact integers that is used to identify the library uniquely when importing from other programs or libraries. Libraries whose first identifier is scheme are reserved for use by this report and future versions of this report. Libraries whose first identifier is srfi are reserved for libraries implementing Scheme Requests for Implementation.

A (library declaration) may be any of:

- (export (export spec) ...)
- (import (import set) ...)
- (begin (command or definition) ...)
- (include (filename1) (filename2) ...)
- (include-ci (filename1) (filename2) ...)
- (cond-expand (cond-expand clause) ...)

An export declaration specifies a list of identifiers which can be made visible to other libraries or programs. An (export spec) takes one of the following forms:

- (identifier)
- (rename (identifier1) (identifier2))
In an (export spec), an (identifier) names a single binding defined within or imported into the library, where the external name for the export is the same as the name of the binding within the library. A rename spec exports the binding defined within or imported into the library and named by (identifier₁) in each ((identifier₁) (identifier₂)) pairing, using (identifier₂) as the external name.

An import declaration provides a way to import the identifiers exported by a library. Each (import set) names a set of bindings from another library and possibly specifies local names for the imported bindings. It takes one of the following forms:

- (library name)
- (only (import set) (identifier) ...)
- (except (import set) (identifier) ...)
- (prefix (import set) (identifier))
- (rename (import set₁) ((identifier₂) (identifier)) ...)

In the first form, all of the identifiers in the named library's export clauses are imported with the same names (or the exported names if exported with rename). The additional (import set) forms modify this set as follows:

- only produces a subset of the given (import set), including only the listed identifiers (after any renaming). It is an error if any of the listed identifiers are not found in the original set.
- except produces a subset of the given (import set), excluding the listed identifiers (after any renaming). It is an error if any of the listed identifiers are not found in the original set.
- rename modifies the given (import set), replacing each instance of (identifier₁) with (identifier₂). It is an error if any of the listed identifiers are not found in the original set.
- prefix automatically renames all identifiers in the given (import set), prefixing each with the specified (identifier).

The top level of a program may also include import declarations. In a library declaration, it is an error to import the same identifier more than once with different bindings, or to redefine or mutate an imported binding with define, define-syntax or set!. However, a REPL should permit these actions.

The begin, include, and include-ci declarations are used to specify the commands and definitions that make up the body of the library. The begin declaration takes a list of expressions and definitions to be spliced literally, analogous to the top-level begin. Both the include and include-ci declarations take one or more filenames expressed as string literals, apply an implementation-specific algorithm to find corresponding files, read the whole contents of each file, and include the results into the library body or program as though wrapped in a top-level begin. The difference between the two is that include-ci reads each file as if it began with the #!fold-case directive, while include does not. All three may appear at the top level of a program.

Note: Implementations are encouraged to search for files in the directory which contains the including file, and to provide a way for users to specify other directories to search.

Note: For portability, include and include-ci must operate on source files. Their operation on other kinds of files necessarily varies among implementations.

The cond-expand library declaration provides a way to statically expand different library declarations depending on the implementation under which the library is being loaded. A (cond-expand clause) takes the following form:

(⟨feature requirement⟩ (library declaration) ...)

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

(else (library declaration) ...)

A (feature requirement) takes one of the following forms:

- (feature identifier)
- (library (library name))
- (and (feature requirement) ...)
- (or (feature requirement) ...)
- (not (feature requirement))

Each implementation maintains a list of feature identifiers which are present, as well as a list of libraries which can be imported. The value of a (feature requirement) is determined by replacing each (feature identifier) and (library (library name)) on the implementation's lists with #f, and all other feature identifiers and library names with #f, then evaluating the resulting expression as a Scheme boolean expression under the normal interpretation of and, or, and not.

A cond-expand is then expanded by evaluating the (feature requirement) of successive (cond-expand clause)s in order, until one of them returns #t. When a true clause is found, the corresponding (library declaration) s are spliced into the current library definition and the remaining clauses are ignored. If none of the (feature requirement) s
evaluate_to #t, then if there is an else clause, its
(library declaration)'s are included. Otherwise, the
cond-expand has no effect.

The exact features provided are implementation-defined, but for portability a core set of features is given in
appendix B.

After all cond-expand library declarations are expanded, a new environment is constructed for the library consisting
of all imported bindings. The expressions and declarations from all begin, include and include-ci declarations are expanded in that environment in the order in which they occur in the library declaration.

The top-level expressions in a library are executed in the order in which they occur when the library is loaded. A
library is loaded zero or more times when it is imported by a program or by another library which is about to be
loaded, but must be loaded at least once per program in which it is so imported.

5.5.2. Library example

The following example shows how a program may be
divided into libraries plus a relatively small main program.
If the main program is entered into a REPL, it is not
necessary to import the base module.

(define-library (example grid) (export make rows cols ref each)
                 (import (except (scheme base) set!)))

(define-library (example life) (export life) (import (except (scheme base) set!))

;; Main program. (import (scheme base) (only (example life) life)
;; (rename (prefix (example grid) grid-))

;; Initialize a grid with a glider. (define grid (make-grid 24 24))
;; (grid-set! grid 1 1 #true) (grid-set! grid 2 2 #true)

;; Run for 80 iterations. (life grid 80)

6. Standard procedures

This chapter describes Scheme’s built-in procedures. The
initial (or “top level”) Scheme environment starts out
with a number of variables bound to locations containing
useful values, most of which are primitive procedures that
manipulate data. For example, the variable

The procedures abefore is bound to (a location initially
containing) a procedure of one argument that computes the
absolute value of a number, and the variable and eager
are intimately associated with the expression type parameterize, and is described
with it in section 1.2.6.

A program may use a top-level definition to bind any variable.
It may subsequently alter any such binding by an assignment (see section 4.1.6). These operations do not modify the behavior of Scheme’s built-in procedures, or any procedure defined in a library (see section 5.5). Altering any top-level binding that has not been introduced by a definition has an unspecified effect on the behavior of the built-in procedures.

6.1. Equivalence predicates

A predicate is a procedure that always returns a boolean value (#t or #f). An equivalence predicate is the computational analogue of a mathematical equivalence relation (it is symmetric, reflexive, and transitive). Of the equivalence predicates described in this section, eq? is the finest and most discriminating, and equal? is the coarsest. and eqv? is slightly less discriminating than eq?.

(eqv? obj1 obj2) procedure

The eqv? procedure defines a useful equivalence relation on objects. Briefly, it returns #t if obj1 and obj2 should normally be regarded as the same object. 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:

- obj1 and obj2 are both #t or both #f.
- obj1 and obj2 are both symbols and

  (string=? (symbol->string obj1)
           (symbol->string obj2))

  (eqv? obj1 obj2)

Note: This assumes that neither obj1 nor obj2 is an “un-interned” symbol as alluded to in section 6.3. This report does not presume to specify the behavior of eqv? on implementation-dependent extensions.

- obj1 and obj2 are both numbers, are numerically equal (see =, section 6.2), and are either both exact or both inexact.
- obj1 and obj2 are both characters and are the same character according to the char=? procedure (section 6.6).
- both obj1 and obj2 are both the empty list.
• $obj_1$ and $obj_2$ are pairs, vectors, bytevectors, records, or strings that denote the same location in the store (section 3.4), and are procedures whose location tags are equal (section 4.1).

The $eqv?$ procedure returns #f if:

• $obj_1$ and $obj_2$ are of different types (section 3.2).
• one of $obj_1$ and $obj_2$ is # but the other is #f.
• $obj_1$ and $obj_2$ are symbols but
  $$
  (string=? (symbol->string obj_1) (symbol->string obj_2)) \quad \Rightarrow \quad #f
  $$
• 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, and $nan$? returns #f for both.
• $obj_1$ 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, bytevectors, records, or strings that denote distinct locations.
• $obj_1$ and $obj_2$ are procedures that would behave differently (return different values\textit{\ }values or have different side effects) for some arguments.

$$
  (eqv? 'a 'a) \quad \Rightarrow \quad #t
$$
$$
  (eqv? 'a 'b) \quad \Rightarrow \quad #f
$$
$$
  (eqv? 2 2) \quad \Rightarrow \quad #t
$$
$$
  (eqv? (cons 1 2) (cons 1 2)) \quad \Rightarrow \quad #f
$$
$$
  (eqv? (lambda (x) x) (lambda (x) x)) \quad \Rightarrow \quad #f
$$
$$
  (eqv? 'nil 'nil) \quad \Rightarrow \quad #f
$$
$$
  (let ((p (lambda (x) x))) (eqv? p p))
$$

The following examples illustrate cases in which the above rules do not fully specify the behavior of $eqv?$. All that can be said about such cases is that the value returned by $eqv?$ must be a boolean.

$$
  (eqv? "" "") \quad \Rightarrow \quad unspecified
$$
$$
  (eqv? '#() '#()) \quad \Rightarrow \quad unspecified
$$
$$
  (eqv? (lambda (x) x) (lambda (x) x)) \quad \Rightarrow \quad unspecified
$$
$$
  (eqv? (let ((p (lambda (x) x))) (eqv? p p)) (eqv? (lambda (x) x))) \quad \Rightarrow \quad unspecified
$$

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

```
(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) 'f 'both)))
         (g (lambda () (if (eqv? f g) 'both 'g))))
  (eqv? f g) => unspecified

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

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

$$
  (eqv? '(a) '(a)) \quad \Rightarrow \quad unspecified
$$
$$
  (eqv? "a" "a") \quad \Rightarrow \quad unspecified
$$
$$
  (eqv? '(b) (cdr '(a b))) \quad \Rightarrow \quad unspecified
$$
$$
  (let ((x '(a))
        (y (x)))
    (eqv? x y)) \quad \Rightarrow \quad #t
$$

\textit{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.

The $eqv?$ procedure is similar to $eqv?$ except that in some cases it is capable of discerning distinctions finer than those detectable by $eqv?$.
On symbols, booleans, the empty list, pairs, procedures, and non-empty strings, vectors, byteworkers, and records, \texttt{eq?} and \texttt{eqv?} are guaranteed to have the same behavior on symbols, booleans, the empty list, pairs, procedures, and non-empty strings and vectors. On numbers and characters, \texttt{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 \texttt{eqv?} would also return true. On empty vectors and empty strings, \texttt{eq?} may also behave differently from \texttt{eqv?}; on empty vectors and empty strings.

\[
\begin{array}{ll}
\text{(eq? 'a 'a)} & \Rightarrow \texttt{#t} \\
\text{(eq? '(a) '(a))} & \Rightarrow \texttt{unspecified} \\
\text{(eq? (list 'a) (list 'a))} & \Rightarrow \texttt{#f} \\
\text{(eq? "a" "a")} & \Rightarrow \texttt{unspecified} \\
\text{(eq? "a" "a")} & \Rightarrow \texttt{unspecified} \\
\text{(eq? '() '())} & \Rightarrow \texttt{#t} \\
\text{(eq? 2 2)} & \Rightarrow \texttt{unspecified} \\
\text{(eq? #\A #\A)} & \Rightarrow \texttt{unspecified} \\
\text{(eq? car car)} & \Rightarrow \texttt{unspecified} \\
\text{(let ((n (+ 2 3))) (eq? n n))} & \Rightarrow \texttt{unspecified} \\
\text{(let ((x '(a))) (eq? x x))} & \Rightarrow \texttt{#t} \\
\text{(let ((x '(#())) (eq? x x))} & \Rightarrow \texttt{#t} \\
\text{(let ((p (lambda (x) x))) (eq? p p))} & \Rightarrow \texttt{#t} \\
\end{array}
\]

Rationale: It will usually be possible to implement \texttt{eq?} much more efficiently than \texttt{eqv?}, for example, as a simple pointer comparison instead of as some more complicated operation. One reason is that it may not be is not always possible to compute \texttt{eqv?} of two numbers in constant time, whereas \texttt{eq?} implemented as pointer comparison will always finish in constant time. In applications using procedures to implement objects with state, \texttt{eqv?} may be used like instead of \texttt{eqv?} in applications using procedures to implement objects with state since it obeys the same constraints as \texttt{eqv?}.

\textbf{The \texttt{Equal} procedure} recursively compares the contents of pairs, vectors, and strings, strings, byteworkers, and records, applying \texttt{eqv?} on other objects such as numbers and symbols. A rule of thumb is that objects are generally If two objects are \texttt{equal?} if they print the same, they must be \texttt{Equal} may fail to terminate as well. Even if its arguments are circular data structures, \texttt{equal?} must always terminate.

\[
\begin{array}{ll}
\text{(equal? 'a 'a)} & \Rightarrow \texttt{#t} \\
\text{(equal? '(a) '(a))} & \Rightarrow \texttt{#t} \\
\text{(equal? '(a (b) c) 'a (b) c))} & \Rightarrow \texttt{#t} \\
\text{(equal? "abc" "abc")} & \Rightarrow \texttt{#t} \\
\text{(equal? 2 2)} & \Rightarrow \texttt{#t} \\
\text{(equal? (make-vector 5 'a))} & \Rightarrow \texttt{#t} \\
\text{(equal? (lambda (x) x))} & \Rightarrow \texttt{unspecified} \\
\end{array}
\]

Note: A rule of thumb is that objects are generally \texttt{equal?} if they print the same.

6.2. Numbers

Numerical computation has traditionally been neglected by the Lisp community. Until Common Lisp there was no carefully thought out strategy for organizing numerical computation, and with the exception of the MacLisp system little effort was made to execute numerical code efficiently. This report recognizes the excellent work of the Common Lisp committee and accepts many of their recommendations. In some ways this report simplifies and generalizes their proposals in a manner consistent with the purposes of Scheme.

It is important to distinguish between the mathematical numbers, the Scheme numbers that attempt to model them, the machine representations used to implement the Scheme numbers, and notations used to write numbers. This report uses the types \texttt{number}, \texttt{complex}, \texttt{real}, \texttt{rational}, and \texttt{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 and.

6.2.1. Numerical types

Mathematically, numbers may be are arranged into a tower of subtypes in which each level is a subset of the level above it:

\begin{align*}
\text{number} & \\
\text{complex} & \\
\text{real} & \\
\text{rational} & \\
\text{integer} & \\
\end{align*}

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 \texttt{number?}, \texttt{complex?}, \texttt{real?}, \texttt{rational?}, and \texttt{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 fixnums.
flonum, and perhaps other representations for multiple internal representations of 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 \textbf{may} not be. For example, indexes into data structures must be known exactly, as must some polynomial coefficients in a symbolic algebra system. On the other hand, the results of measurements are inherently inexact, and irrational numbers may be approximated by rational and therefore inexact approximations. In order to catch uses of inexact numbers where exact numbers are required, Scheme explicitly distinguishes exact from inexact numbers. This distinction is orthogonal to the dimension of type.

6.2.2. Exactness

Scheme numbers are either \textit{exact} or \textit{inexact}. A number is exact if it was written as an exact constant or was derived from exact numbers using only exact operations. A number is inexact if it was written as an inexact constant, if it was derived using inexact ingredients, or if it was derived using inexact operations. Thus inexactness is a contagious property of a number.

If two implementations produce exact results for a computation that did not involve inexact intermediate results, the two ultimate results will be mathematically equivalent. This is generally not true of computations involving inexact numbers since approximate methods such as floating-point arithmetic may be used, but it is the duty of each implementation to make the result as close as practical to the mathematically ideal result.

Rational operations such as $+$ should always produce exact results when given exact arguments. If the operation is unable to produce an exact result, then it may either report the violation of an implementation restriction or it may silently coerce its result to an inexact value. See section 6.2.3.

With the exception of \texttt{Except for inexact->exact}, the operations described in this section must generally return inexact results when given any inexact arguments. An operation may, however, return an exact result if it can prove that the value of the result is unaffected by the inexactness of its arguments. For example, multiplication of any number by an exact zero may produce an exact zero result, even if the other argument is inexact.

6.2.3. Implementation restrictions

Implementations of Scheme are not required to implement the whole tower of subtypes given in section 6.2.1 but they must implement a coherent subset consistent with both the purposes of the implementation and the spirit of the Scheme language. For example, an implementation implementations in which all numbers are real may still be, or in which non-real numbers are always inexact, or in which exact numbers are always integer, are still 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 IEEE double-precision floating-point numbers to represent all its inexact real numbers may also support a practically unbounded range of exact integers and rationals while limiting the range of inexact reals (and therefore the range of inexact integers and rationals) to the dynamic range of the IEEE double format. Furthermore, the gaps between the representable inexact integers and rationals are likely to be very large in such an implementation as the limits of this range are approached.

An implementation of Scheme must support exact integers throughout the range of numbers \textbf{that may be used for permitted as indexes} of lists, vectors, bytevectors, and strings or that \textbf{may result from computing the length of} a list, vector, or string \textbf{of one of these}. The \texttt{length}, \texttt{vector-length}, \texttt{bytevector-length}, and \texttt{string-length} procedures must return an exact integer, and it is an error to use anything but an exact integer as an index. Furthermore, any integer constant within the index range, if expressed by an exact integer syntax, will indeed be read as an exact integer, regardless of any implementation restrictions that \textbf{may apply outside this range}. Finally, the procedures listed below will always return an exact integer result; \texttt{exact integer results} provided all their arguments are exact integers and the mathematically expected result is representable as an exact integer results are representable as exact integers within the implementation:

\begin{verbatim}
+ - * quotient remainder modulo max min abs numerator denominator gcd lcm floor ceiling truncate round rationalize

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

\end{verbatim}

Implementations are encouraged, but not required, to support exact integers and exact rationals of practically unlimited size and precision, and to implement the above procedures and the \texttt{/} 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. Nevertheless, implementations that do not provide exact
rational numbers should return inexact rational numbers rather than reporting an implementation restriction.

An implementation may use floating-point and other approximate representation strategies for inexact numbers. This report recommends, but does not require, that the IEEE 32-bit and 64-bit floating point standards 754 standard be followed by implementations that use flonum floating-point representations, and that implementations using other representations should match or exceed the precision achievable using these floating-point floating-point standards [17].

In particular, implementations that use flonum representations must follow these rules: A the description of transcendental functions in IEEE 754-2008 should be followed by such implementations, particularly with respect to infinities and NaNs.

Although Scheme allows a variety of written notations for numbers, any particular implementation may support only some of them. For example, an implementation in which all numbers are real need not support the rectangular and polar notations for complex numbers. If an implementation encounters an exact numerical constant that it cannot represent as an exact number, then it may either report a violation of an implementation restriction or it may silently represent the constant by an inexact number.

6.2.4. Implementation extensions

Implementations may provide more than one representation of floating-point numbers with differing precisions. In an implementation which does so, an inexact result must be represented with at least as much precision as is used to express any of the inexact arguments to that operation. It is desirable (but not required) Although it is desirable for potentially inexact operations such as sqrt, when applied to arguments to to produce exact answers whenever possible (for example the square root of an when applied to exact 4 ought to be an 2). If, however, arguments, if an exact number is operated upon so as to produce an inexact result (as by sqrt), and if the result is represented as a, then the most precise format representation available must be used, but if the result is represented in some other way then the representation must have at least as much precision as the most precise format available. For example, the value of (sqrt 4) should be 2, but in an implementation that provides both single and double precision floating point numbers it may be the latter but must not be the former.

Although Scheme allows a variety of written notations for numbers, any particular implementation may support only some of them. For example, an implementation in which all implementations may distinguish special numbers called positive infinity, negative infinity, NaN, and negative zero.

Positive infinity is regarded as an inexact real (but not rational) number that represents an indeterminate value greater than the numbers represented by all rational numbers. Negative infinity is regarded as an inexact real (but not rational) number that represents an indeterminate value less than the numbers are need not support the rectangular and polar notations for complex numbers represented by all rational numbers.

A NaN is regarded as an inexact real (but not rational) number so indeterminate that it might represent any real value, including positive or negative infinity, and might even be greater than positive infinity or less than negative infinity. It might even represent no number at all, as in the case of (asin 2.0). If an implementation encounters an numerical constant that it cannot represent as an number, then it may either report a violation of an implementation restriction or it may silently represent the constant by an number.

6.2.5. Syntax of numerical constants

Note that the real and the imaginary parts of a complex number can be infinities or NaNs.

Negative zero is an inexact real value written -0.0 which is distinct (in the sense of eqv?) from 0.0. A Scheme implementation is not required to distinguish negative zero. If it does, however, the behavior of the transcendental functions is sensitive to the distinction in accordance with IEEE 754.

Furthermore, the negation of negative zero is ordinary zero and vice versa. This implies that the sum of two negative zeros is negative, and the result of subtracting (positive) zero from a negative zero is likewise negative. However, numerical comparisons treat negative zero as equal to zero.

6.2.5. Syntax of numerical constants

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

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

A numerical constant may numerical constant can 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 or. It is inexact if it
contains a decimal point, an exponent, or a "e" character in the place of a digit, otherwise or an exponent. 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 implementations may accept numerical constants 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 meaning short, single, double, and long precision, respectively. (When fewer than four internal 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 respectively are acceptable in place of 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

The numbers positive infinity, negative infinity and NaN are written +inf.0, -inf.0 and +nan.0 respectively. Implementations are not required to support them, but if they do, they must be in conformance with IEEE 754. However, implementations are not required to support signaling NaNs, or provide a way to distinguish between different NaNs.

6.2.6. Numerical operations

The reader is referred to section 1.3.3 for a summary of the naming conventions used to specify restrictions on the types of arguments to numerical routines. The examples used in this section assume that any numerical constant written using an exact notation is indeed represented as an exact number. Some examples also assume that certain numerical constants written using an inexact notation can be represented without loss of accuracy; the inexact constants were chosen so that this is likely to be true in implementations that use IEEE doubles to represent inexact numbers.

(object? object) procedure
(real? object) procedure
(complex? object) procedure
(rational? object) procedure
(integer? object) procedure

These numerical type predicates can be applied to any kind of argument, including non-numbers. They return #t if the object is of the named type, and otherwise they return #f. In general, if a type predicate is true of a number then all higher type predicates are also true of that number. Consequently, if a type predicate is false of a number, then all lower type predicates are also false of that number.

If z is an inexact complex number, then (real? z) is true if and only if (zero? (imag-part z)) and (exact? (imag-part z)) are both true. If z is an inexact real number, then (integer? z) is true if and only if (= x (round x)).

The numbers +inf.0, -inf.0, and +nan.0 are real but not rational.

(complex? 3+4i) ⇒ #t
(real? 3) ⇒ #t
(real? -2.5+ 0i) ⇒ #t
(real? -2.5+ 0.0i) ⇒ #f
(real? #e1e10) ⇒ #t
(rational? real? +inf.0) ⇒ #t
(rational? -inf.0) ⇒ #f
(rational? 6/10) ⇒ #t
(rational? 6/3) ⇒ #t
(integer? 3+0i) ⇒ #t
(integer? 3.0) ⇒ #t
(integer? 8/4) ⇒ #t

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

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

(exact? z) procedure
(inexact? z) procedure

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

(exact? 3.0) ⇒ #f
(exact? #3.0) ⇒ #t
(inexact? 3.) ⇒ #t

(exact-integer? z) procedure

Returns #t if z is both exact and an integer; otherwise returns #f.

(exact-integer? 32) ⇒ #t
(exact-integer? 32.0) ⇒ #f
(exact-integer? 32/5) ⇒ #f
The \texttt{finite?} procedure returns \#t on all real numbers except \(+\infty.0\), \(-\infty.0\), and \(\pm\texttt{nan}.0\), and on complex numbers if their real and imaginary parts are both finite. Otherwise it returns \#f.

\[
\begin{array}{ll}
(\texttt{finite? } 3) & \Rightarrow \#t \\
(\texttt{finite? } +\texttt{inf}.0) & \Rightarrow \#f \\
(\texttt{finite? } 3.0+\texttt{inf}.0i) & \Rightarrow \#f
\end{array}
\]

The \texttt{nan?} procedure returns \#t on \(+\texttt{nan}.0\), and on any complex number if its real part or its imaginary part or both are \(+\texttt{nan}.0\). Otherwise it returns \#f.

\[
\begin{array}{ll}
(\texttt{nan? } +\texttt{nan}.0) & \Rightarrow \#t \\
(\texttt{nan? } 32) & \Rightarrow \#f \\
(\texttt{nan? } +\texttt{nan}.0+5.0i) & \Rightarrow \#t \\
(\texttt{nan? } 1+2i) & \Rightarrow \#f
\end{array}
\]

The \texttt{abs} procedure returns the absolute value of its argument.

\[
\begin{array}{ll}
(\texttt{abs } -3 4) & \Rightarrow 4 ; \text{ exact} \\
(\texttt{abs } 3.9 4) & \Rightarrow 4.0 ; \text{ inexact}
\end{array}
\]

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 \texttt{min} or \texttt{max} is used to compare numbers of mixed exactness, and the numerical value of the result cannot be represented as an inexact number without loss of accuracy, then the procedure may report a violation of an implementation restriction.

These procedures return the sum or product of their arguments.

\[
\begin{array}{ll}
(+ 3 4) & \Rightarrow 7 \\
(+ 3) & \Rightarrow 3 \\
(*) & \Rightarrow 0 \\
(* 4) & \Rightarrow 4 \\
(*) & \Rightarrow 1
\end{array}
\]

These procedures return \#t if their arguments are (respectively): equal, monotonically increasing, monotonically decreasing, monotonically nondecreasing, or monotonically nonincreasing, and \#f otherwise. If any of the arguments are \(+\texttt{nan}.0\), all the predicates return \#f.

This predicate is required to be transitive.

Note: The traditional implementations of these predicates in Lisp-like languages are not transitive.

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

The \texttt{floor/} \texttt{floor-quotient恰好} \texttt{ceiling/} \texttt{ceiling-quotient} \texttt{truncate/} procedures return the maximum or minimum of their arguments.

\[
\begin{array}{ll}
(\texttt{floor/ } n_1 n_2) & \Rightarrow n_2 \\
(\texttt{floor-quotient } n_1 n_2) & \Rightarrow 1 \\
(\texttt{ceiling/ } n_1 n_2) & \Rightarrow n_1 \\
(\texttt{ceiling-quotient } n_1 n_2) & \Rightarrow 1 \\
(\texttt{truncate/ } n_1 n_2) & \Rightarrow n_1
\end{array}
\]
(truncate-quotient $n_1$ $n_2$) procedure
(truncate-remainder $n_1$ $n_2$) procedure
(round/ $n_1$ $n_2$) procedure
(round-quotient $n_1$ $n_2$) procedure
(round-remainder $n_1$ $n_2$) procedure
(euclidean/ $n_1$ $n_2$) procedure
(euclidean-quotient $n_1$ $n_2$) procedure
(euclidean-remainder $n_1$ $n_2$) procedure
(centered/ $n_1$ $n_2$) procedure
(centered-quotient $n_1$ $n_2$) procedure
(centered-remainder $n_1$ $n_2$) procedure

These procedures, all in the division library, implement number-theoretic (integer) division. It is an error if $n_2$ should be non-zero. All three procedures return integers.

If is-integer $n_2$ The procedures ending in / are integer: return two integers; the other procedures return an integer. All the procedures compute a quotient $n_q$ and remainder $n_r$ such that $n_1 = n_2n_q + n_r$. For each of the six division operators, there are three procedures defined as follows:

\[
\begin{align*}
\text{(quotient)} & \quad \text{(operator)} \leftarrow \text{(remainder), } n_q \\
\text{(operator)-quotient} & \quad n_1 \quad \Rightarrow \quad \text{module} \quad n_q \\
\text{(operator)-remainder} & \quad n_1 \quad \Rightarrow \quad n_r
\end{align*}
\]

where is $n_1/n_2$ rounded towards zero, $0 < |n_r| < |n_1|$, $0 < |n_r| < |n_2|$.

The remainder $n_r$ and differ from by a multiple of, has the same sign as, and has the same sign as, is determined by the choice of integer $n_q$: $n_r = n_1 - n_2n_q$. Each set of operators uses a different choice of $n_q$:

From this we can conclude that:

\[
\begin{align*}
\text{ceiling} & \quad n_q = \lceil n_1/n_2 \rceil \\
\text{floor} & \quad n_q = \lfloor n_1/n_2 \rfloor \\
\text{truncate} & \quad n_q = \text{truncate}(n_1/n_2) \\
\text{round} & \quad n_q = \text{round}(n_1/n_2) \\
\text{euclidean} & \quad \text{if } n_2 > 0, n_q = \lfloor n_1/n_2 \rfloor; \text{ if } n_2 < 0, n_q = \lceil n_1/n_2 \rceil \\
\text{centered} & \quad \text{choose } n_q \text{ such that } -\lfloor |n_2|/2 \rfloor \leq n_r < \lfloor |n_2|/2 \rfloor
\end{align*}
\]

For any of the operators and for integers $n_1$ and $n_2$ with $n_2$ not equal to 0,

\[
\begin{align*}
\text{(# =)} \quad n_1 & \quad \text{( operator) } \quad \text{(operator)-quotient } \quad n_1 \quad \Rightarrow \quad \text{module } \quad n_q \\
\text{(operator)-remainder } \quad n_1 \quad \Rightarrow \quad n_r
\end{align*}
\]

provided all numbers involved in that computation are exact.


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

The quotient and remainder procedures are equivalent to truncate-quotient and truncate-remainder, respectively, and modulo is equivalent to floor-remainder.

(remainder 13 4) \quad \Rightarrow \quad 1
\]
(remainder -13 4) \quad \Rightarrow \quad 1
(modulo 13 4) \quad \Rightarrow \quad 1
(modulo -13 4) \quad \Rightarrow \quad 3
(remainder 13 -4) \quad \Rightarrow \quad 1
(modulo 13 -4) \quad \Rightarrow \quad -3
(remainder 13 -4) \quad \Rightarrow \quad -1
(modulo -13 -4) \quad \Rightarrow \quad -1
(remainder -13 -4.0) \quad \Rightarrow \quad -1.0 ; \text{ inexact}

Note: These procedures are provided for backward compatibility with earlier versions of this report.

(gcd $n_1$ ...) procedure
(lcm $n_1$ ...) procedure
These procedures return the greatest common divisor or the least common multiple of their arguments. The result is $n_r$, always non-negative.

(gcd 32 -36) \quad \Rightarrow \quad 4
(gcd) \quad \Rightarrow \quad 0
(lcm 32 -36) \quad \Rightarrow \quad 288
(lcm 32.0 -36) \quad \Rightarrow \quad 288.0 ; \text{ inexact}
(lcm) \quad \Rightarrow \quad 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)) \quad \Rightarrow \quad 3
(denominator (/ 6 4)) \quad \Rightarrow \quad 2
(denominator (exact->inexact (/ 6 4))) \quad \Rightarrow \quad 2.0

(floor $x$) procedure
(ceiling $x$) procedure
(truncate $x$) procedure
(round $x$) procedure
These procedures return integers. The Floorfloor procedure returns the largest integer not larger than $x$. The Ceilingceiling procedure returns the smallest integer not smaller than $x$. Truncate truncate returns the integer closest to $x$ whose absolute value is not larger than the absolute value of $x$. and Round round returns the closest integer to $x$, rounding to even when $x$ is halfway between two integers.
Rationale: The **round** procedure rounds to even for consistency with the default rounding mode specified by the IEEE floating-point standard.

Note: If the argument to one of these procedures is inexact, then the result will also be inexact. If an exact value is needed, the result can be passed to the inexact->exact procedure.

\[
\begin{align*}
\text{floor } -4.3 & \Rightarrow -5.0 \\
\text{ceiling } -4.3 & \Rightarrow -4.0 \\
\text{truncate } -4.3 & \Rightarrow -4.0 \\
\text{round } -4.3 & \Rightarrow -4.0 \\
\text{floor } 3.5 & \Rightarrow 3.0 \\
\text{ceiling } 3.5 & \Rightarrow 4.0 \\
\text{truncate } 3.5 & \Rightarrow 3.0 \\
\text{round } 3.5 & \Rightarrow 4.0 \ ; \text{ inexact} \\
\text{round } 7/2 & \Rightarrow 4 \ ; \text{ exact} \\
\text{round } 7 & \Rightarrow 7
\end{align*}
\]

**(rationalize \(x\) \(y\))**

The **rationalize** procedure returns the simplest rational number differing from \(x\) by no more than \(y\). A rational number \(r_1\) is simpler than another rational number \(r_2\) if \(r_1 = p_1/q_1\) and \(r_2 = p_2/q_2\) (in lowest terms) and \(|p_1| \leq |p_2|\) and \(|q_1| \leq |q_2|\). Thus \(3/5\) is simpler than \(4/7\). Although not all rationals are comparable in this ordering (consider \(2/7\) and \(3/5\)) any interval contains a rational number that is simpler than every other rational number in that interval (the simpler \(2/5\) lies between \(2/7\) and \(3/5\)). Note that \(0 = 0/1\) is the simplest rational of all.

\[
\begin{align*}
\text{rationalize } (\text{inexact->exact } .3) 1/10 & \Rightarrow 1/3 \ ; \text{ exact} \\
\text{rationalize } .3 1/10 & \Rightarrow #i1/3 \ ; \text{ inexact}
\end{align*}
\]

\(\text{exp } z\), \(\text{log } z\), \(\text{sin } z\), \(\text{cos } z\), \(\text{tan } z\), \(\text{asin } z\), \(\text{acos } z\), \(\text{atan } z\), \(\text{atan y x}\)

These procedures are part of every implementation that supports general real numbers; they compute the usual transcendental functions. The **log** procedure computes the natural logarithm of \(z\) (not the base ten logarithm). The **asin**, **acos**, and **atan** procedures 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, the **complex** library.

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

\[
\begin{align*}
\sin^{-1} z &= -i \log(iz + \sqrt{1 - z^2}) \\
\cos^{-1} z &= \pi/2 - \sin^{-1} z \\
\tan^{-1} z &= (\log(1 + iz) - \log(1 - iz))/2i
\end{align*}
\]

The above specification follows [23], which in turn cites [25]; 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.

\(\text{(sqrt } z\)\)

Inexact library procedure

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

\(\text{(exact-integer-sqrt } k\)\)

Procedure

Returns two non-negative exact integers \(s\) and \(r\) where \(k = s^2 + r\) and \(k < (s+1)^2\).

\[
\begin{align*}
\text{(exact-integer-sqrt } 4\) & \Rightarrow 2.0 \\
\text{(exact-integer-sqrt } 5\) & \Rightarrow 2.1
\end{align*}
\]

\(\text{(exp } z\)\)

Procedure

Returns \(z\) raised to the power \(\pi\). For \(z \neq 0\) nonzero \(z\), this is

\[
z^\pi = e^{\pi \log z}
\]

0\(^\pi\) is 1 if \(z = 0\) and 0 otherwise. 0\(^\pi\) is 1.0 if \(z = 0.0\), and 0.0 if \(\text{real-part } z\) is positive. For other cases in which the first argument is zero, either an error is signalled or an unspecified number is returned.

\(\text{(make-rectangular } x_1 \ x_2\)\)

Complex library procedure

\(\text{(make-polar } x_3 \ x_4\)\)

Complex library procedure

\(\text{(real-part } z\)\)

Complex library procedure

\(\text{(imag-part } z\)\)

Complex library procedure

\(\text{(magnitude } z\)\)

Complex library procedure

\(\text{(angle } z\)\)

Complex library 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

\[
z = x_1 + x_2 i = x_3 \cdot e^{ix_4}
\]

Then all of
(make-rectangular \( x_1 \), \( x_2 \)) \( \Rightarrow \) \( z \)
(make-polar \( x_3 \), \( x_4 \)) \( \Rightarrow \) \( z \)
(real-part \( z \)) \( \Rightarrow \) \( x_1 \)
(imag-part \( z \)) \( \Rightarrow \) \( x_2 \)
(magnitude \( z \)) \( \Rightarrow \) \( |x_3| \)
(angle \( z \)) \( \Rightarrow \) \( x_{angle} \)

are true, where \(-\pi < x_{angle} \leq \pi\) with \( x_{angle} = x_4 + 2\pi n \)
for some integer \( n \).

The make-polar procedure may return an inexact complex number even if its arguments are exact.

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

(exact->inexact \( z \)) procedure
(inexact->exact \( z \)) procedure

The procedure exact->inexact returns an inexact representation of \( z \). The value returned is the inexact number that is numerically closest to the argument. For inexact arguments, the result is the same as the argument. For exact complex numbers, the result is a complex number whose real and imaginary parts are the result of applying exact->inexact to the real and imaginary parts of the argument, respectively. If an exact argument has no reasonably close inexact equivalent, then a violation of an implementation restriction may be reported.

The procedure inexact->exact returns an exact representation of \( z \). The value returned is the exact number that is numerically closest to the argument. For exact arguments, the result is the same as the argument. For inexact non-integral real arguments, the implementation may return a rational approximation, or may report an implementation violation. For inexact complex arguments, the result is a complex number whose real and imaginary parts are the result of applying inexact->exact to the real and imaginary parts of the argument, respectively. If an inexact argument has no reasonably close exact equivalent, then a violation of an implementation restriction may be reported.

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

6.2.7. Numerical input and output

(number->string \( z \)) procedure
(number->string \( z \) radix) procedure

must be an exact integer, either It is an error if radix is not one of 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

(\( \text{let } ((\text{number number})\)
\( (\text{radix radix}))\)
\( (\text{eqv? number}\)
\( (\text{string->number (number->string number})\)
\( \text{radix}))\)

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

If \( z \) is inexact, the radix is 10, and the above expression can be satisfied by a result that contains a decimal point, then the result contains a decimal point and is expressed using the minimum number of digits (exclusive of exponent and trailing zeroes) needed to make the above expression true \[6, 8\]; otherwise the format of the result is unspecified.

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

Note: The error case can occur only when \( z \) is not a complex number or is a complex number with a non-rational real or imaginary part.

Rationale: If \( z \) is an inexact number represented using \#\(\text{f/0} \), 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-finite unusual representations.

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

Returns a number of the maximally precise representation expressed by the given string. must be an exact integer, or is a complex number with a non-rational real or imaginary part. It is an error if radix is not 2, 8, 10, or 16. If supplied, radix is a default radix that may will be overridden by an explicit radix prefix in string (e.g. \#\(\text{f17} \)). 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") \( \Rightarrow \) 100
(string->number "100" 16) \( \Rightarrow \) 256
(string->number "1e2") \( \Rightarrow \) 100.0
(string->number "1500.0")

Note: The domain of string->number may be restricted by implementations in the following ways. Whenever string contains an explicit radix prefix, string->number is permitted to return #f whenever 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 digit or an inexact marker appears in place of a digit. If all inexact numbers are integers, then `string->number` may return `#f` whenever a decimal point is used.

6.3. Booleans

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

6.3.1. Booleans

The standard boolean objects for true and false are written as `#t` and `#f`. Alternatively, they may be written `true` and `false`, respectively. What really matters, though, are the objects that the Scheme conditional expressions (if, cond, and, or, when, unless, do) treat as true or false. The phrase “a true value” (or sometimes just “true”) means any object treated as true by the conditional expressions, and the phrase “a false value” (or “false”) means any object treated as false by the conditional expressions.

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

Note: Programmers accustomed to Lisp should be aware that Scheme distinguishes both `true` and the empty list from each other and from the symbol `nil`.

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

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

The `not` procedure returns `#t` if `obj` is false, and returns `#f` otherwise.

```
(not  #t)  ⇒  #f
(not  #f)  ⇒  #t
(not  ' #f)  ⇒  #t
(not  (list 3))  ⇒  #f
(not  (list 3))  ⇒  #f
(not  (list))  ⇒  #f
(not  'nil)  ⇒  #f
```

6.4. 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 if it is not a pair—it has no elements and its length is zero.

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

The most general notation (external representation) for Scheme pairs is the “dotted” notation `(c1 . e2)` where `c1` is the value of the car field and `e2` is the value of the cdr field. For example `(#1 5)` is a pair whose car is 4 and whose cdr is 5. Note that `(#1 5)` is the external representation of a pair, not an expression that evaluates to a pair.

A more streamlined notation can be used for lists: the elements of the list are simply enclosed in parentheses and separated by spaces. The empty list is written `( )`. For example,

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

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

are equivalent notations for a list of symbols.

A chain of pairs not ending in the empty list is called an improper list. Note that an improper list is not a list. The list and dotted notations can be combined to represent improper lists:

(a b c . d)

is equivalent to

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

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

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

Within literal expressions and representations of objects read by the read procedure, the forms '(datum), 'datum, 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-possibly be represented as lists. That is, according to Scheme’s grammar, every ⟨expression⟩ is also a ⟨datum⟩ (see section 7.1.2). Among other things, this permits the use of the read procedure to parse Scheme programs. See section 3.3.

(pair? obj) procedure

The Pairpair? predicate 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 obj1 obj2) procedure

Returns a newly allocated pair whose car is obj1 and whose cdr is obj2. The pair is guaranteed to be different (in the sense of eqv?) from every 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.

(set-car! (g) 3)
(define (g) '(constant-list))

(set-car! (f) 3)
(define (f) (list 'not-a-constant-list))

(set-car! (f) (constant-list))

(set-cdr! pair obj) procedure

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

(set-cdr! (f) 3)
(set-cdr! (g) 3)

(set-cdr! pair obj) procedure

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

(define cddr (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, it returns #f. By definition, all lists have finite length and are terminated by the empty list.
(list? '(a b c))  ⇒  #t
(list? '())  ⇒  #t
(list? '(a . b))  ⇒  #f
(let ((x (list 'a)))
  (set-cdr! x x))  ⇒  #f

(make-list k)  procedure
(make-list k fill)  procedure

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

(make-list 2 3)  ⇒  (3 3)

(list obj ...)  procedure

Returns a newly allocated list of its arguments.

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

(length list)  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))

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 kth element of list. (This is the same as the result returned by list-tail list k.) It is an error if list has fewer than k elements.

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

(list-set! list k obj)  procedure

It is an error if k is not a valid index of list. The list-set! procedure stores obj in element k of list. The value returned by list-set! is unspecified.

(list-set! '([ (a b c d) 2 ] (list-set! ls 2 'three))  ⇒  (one two three)
(list-set! '(0 1 2) 1 "oops")  ⇒  error : constant list

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

These procedures return the first sublist of list whose car is obj, where the sublists of list are the non-empty lists returned by (list-tail list k) for k less than the length of list. If obj does not occur in list, then #f (not the empty list) is returned. The memq memv procedures use eq? to compare obj with the elements of list, while member uses compare, if given, and equal? otherwise.
(assoc obj alist) procedure
(assv obj alist) procedure
(assoc obj alist) procedure
(assoc obj alist compare) procedure

It is an error if alist (for “association list”) is not a list of pairs. These procedures find the first pair in alist whose car field is obj, and returns that pair. If no pair in alist has obj as its car, then #f (not the empty list) is returned. The \texttt{assoc} procedure uses eq? to compare obj with the car fields of the pairs in alist, while \texttt{assv} uses eqv? and \texttt{assoc} uses compare if \texttt{given} and \texttt{equal?} otherwise.

\begin{verbatim}
(define e '(((a) (b) (c)) (d) (e)))
(assoc 'a e) ⇒ (a 1)
(assoc 'b e) ⇒ (b 2)
(assoc 'd e) ⇒ #f
(assoc (list 'a) '(((a)) ((b)) ((c)))) ⇒ #f
(assoc (list 'a) '(((a)) ((b)) ((c)))) ⇒ (a)
\end{verbatim}

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

### 6.4.1. Symbols

\texttt{(list-copy list)} procedure

Returns a newly allocated copy of the given list. Only the pairs themselves are copied; the cars of the result are the same (in the sense of \texttt{eqv?}) as the cars of list. If the last pair of list has a cdr which is not the empty list, the last pair of the result does too. As a degenerate case, an argument which is not a list is returned unchanged.

### 6.5. Symbols

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

The rules for writing a symbol are exactly the same as the rules for writing an identifier; see sections 2.1 and 7.1.

It is guaranteed that any symbol that has been returned as part of a literal expression, or read using the \texttt{read} procedure, and subsequently written out using the \texttt{write} procedure, will read back in as the identical symbol (in the sense of \texttt{eqv?}). The \texttt{string->symbol} procedure, however, can create symbols for which this write/read invariance may not hold because their names contain special characters or letters in the non-standard case.

\textit{Note:} Some implementations of Scheme have a feature \texttt{have_values} known as “classification” in order to guarantee write/read invariance for all symbols, but historically the most important use of this feature has been to compensate for the lack of a string data type.

Some implementations also have “uninterned symbols” \texttt{uninterned_symbols} which defeat write/read invariance even in implementations with classification, and also generate exceptions to violate the rule that two symbols are the same if and only if their names are spelled the same.

\begin{verbatim}
(symbol->string 'foo) ⇒ #t
(symbol? 'car '(a b))) ⇒ #t
(symbol? "bar") ⇒ #f
(symbol? 'nil) ⇒ #t
(symbol? '()) ⇒ #f
(symbol? #f) ⇒ #f
\end{verbatim}

\texttt{(symbol->string symbol)} procedure

Returns the name of \texttt{symbol} as a string. If the symbol was part of an object returned as the value of a literal expression (section 4.1.2) or by a call to the \texttt{readprocedure}, 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 \texttt{string-symbol}, the case of characters in the string returned will be the same as the case in the string that was passed to \texttt{string-symbol}. It is an error to apply mutation procedures like \texttt{string-set!} to strings returned by this procedure.

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

\begin{verbatim}
(symbol->string 'flying-fish) ⇒ "flying-fish"
(symbol->string 'Martin) ⇒ "martin Martin"
(symbol->string (string->symbol "Malvina")) ⇒ "Malvina"
\end{verbatim}

\texttt{(string->symbol string)} procedure

Returns the symbol whose name is \texttt{string}. This procedure can create symbols with names containing special
characters or letters in the non-standard case, but it is usually a bad idea to create such symbols because in some implementations of Scheme they cannot be read as themselves. See symbol-string, that would require escaping when written.

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

\[ \text{(eq? 'mISSISSIppi 'missionippi) \text{-} (string->symbol "mISSISSIppi") \Rightarrow \#t} \]
\[ \text{(eq? 'JollyWog (symbol->string 'JollyWog)) \Rightarrow \#t} \]
\[ \text{(string=? "K. Harper, M.D." (symbol->string "K. Harper, M.D.")) \Rightarrow \#t} \]

6.5.1. Characters

6.6. Characters

Characters are objects that represent printed characters such as letters and digits. All Scheme implementations must support at least the ASCII character repertoire: that is, Unicode characters U+0000 through U+007F. Implementations may support any other Unicode characters they see fit, and may also support non-Unicode characters as well. Except as otherwise specified, the result of applying any of the following procedures to a non-Unicode character is implementation-dependent.

Characters are written using the notation \#\langle character\rangle or \#\langle character name\rangle. For example or \#\langle hex scalar value\rangle.

Here are some examples:

\#\( \text{name} \) ; lower case letter
\#\( \text{A} \) ; upper case letter
\#\( \text{a} \) ; left parenthesis
\#\( \text{ } \) ; the space character
\#\( \text{j} \) ; \( \text{iota} \) (if supported)
\#\( \text{x03b8} \) ; \( \text{lambda} \) (if supported)

The following character names must be supported by all implementations:

\#\( \text{alarm} \) : U+0007
\#\( \text{backspace} \) : U+0008
\#\( \text{delete} \) : U+001f
\#\( \text{escape} \) : U+001b
\#\( \text{newline} \) : the linefeed character, U+000a
\#\( \text{null} \) : the null character, U+0000
\#\( \text{return} \) : the return character, U+000d
\#\( \text{space} \) : the preferred way to write a space
\#\( \text{tab} \) : the newline character, tab character, U+0009

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

Characters written in the \#\ notation are self-evaluating. That is, they do not have to be quoted in programs.

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

\( \text{(char? obj)} \) procedure

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

\( \text{(char=? char1 char2 char3 ...)} \) procedure
\( \text{(char<? char1 char2 char3 ...)} \) procedure
\( \text{(char>? char1 char2 char3 ...)} \) procedure
\( \text{(char<=? char1 char2 char3 ...)} \) procedure
\( \text{(char>=? char1 char2 char3 ...)} \) procedure

These procedures return \#t if the Unicode codepoints corresponding to their arguments are (respectively): equal, monotonically increasing, monotonically decreasing, monotonically nondecreasing, or monotonically nonincreasing.

These predicates are required to be transitive.

These procedures impose a total ordering on the set of characters. It is guaranteed that under this ordering, which is the same as the Unicode code point ordering. This is true whether or not the implementation uses the Unicode representation internally.

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

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

(char-ci=? char₁ char₂ char₃ ...)
(char-ci<? char₁ char₂ char₃ ...)
(char-ci>? char₁ char₂ char₃ ...)
(char-ci<=? char₁ char₂ char₃ ...)
(char-ci>=? char₁ char₂ char₃ ...)

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

Specifically, these procedures behave as if char-foldcase were applied to their arguments before they were compared.

(char-alphabetic? char)
(char-numeric? char)
(char-whitespace? char)
(char-upper-case? letter)
(char-lower-case? letter)

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

Specifically, they must return #\t when applied to characters with the Unicode properties Alphabetic, Numeric SUBSCRIPTNBDigit, White SUBSCRIPTNBSpace, Uppercase, and Lowercase respectively, and #\f when applied to any other Unicode characters. Note that many Unicode characters are alphabetic but neither upper nor lower case.

(digit-value char)

This procedure returns the numeric value (0 to 9) of its argument if it is a numeric digit (that is, if char-numeric? returns #\t), or #\f on any other character.

(digit-value \3)  =>  3
(digit-value \x0664)  =>  4
(digit-value \x0EA6)  =>  0

(char->integer char)
(integer->char n)

Given a Unicode character, char->integer returns an exact integer representation of the between 0 and #\x0DFF or between #\x0E000 and #\x11FFFF which is equal to the Unicode code point of that character. Given a non-Unicode character, it returns an exact integer greater than #\x10FFFF. This is true independent of whether the implementation uses the Unicode representation internally.

Given an exact integer that is the image of a character under-value returned by a character when char->integer is applied to it, integer->char returns that character. These procedures implement an order-preserving isomorphisms between the set of characters under the ordering and some subset of the integers under the ordering. That is, if

(char=? id) (char-upcase char)

(char-downcase char)
(char-foldcase char)

and and are in the domain of The integer->char, then char-upcase procedure given an argument that is the lower case part of a Unicode casing pair, returns the uppercase member of the pair, provided that both characters are supported by the Scheme implementation. Note that language-sensitive casing pairs are not used. If the argument is not the lowercase member of such a pair, it is returned.

(id (charinteger) (charinteger))
(char=? (integerchar) (integerchar))

These procedures return a character such that The \(\text{char-ci=? char-downcase} \) In addition, if is alphabetic, then the result of procedure, given an argument that is the upper case part of a Unicode casing pair, returns the lowercase member of the pair, provided that both characters are supported by the Scheme implementation. Note that language-sensitive casing pairs are not used. If the argument is not the upper case member of such a pair, it is returned.

The char-upcase is upper-case and the result of char-downcase char-foldcase is lower case procedure applies the Unicode simple case-folding algorithm
Strings are sequences of characters. Strings are written as sequences of characters enclosed within double-quotes (" "). A doublequote can be written inside a string only by escaping it. Within a string literal, various escape sequences represent characters other than themselves. Escape sequences always start with a backslash (\) as in:

- \a: alarm, U+0007
- \b: backspace, U+0008
- \t: character tabulation, U+0009
- \n: linefeed, U+000A
- \r: return, U+000D
- \": doublequote, U+0022
- \\: backslash, U+006C
- \ associative whitespace\line ending
  (intraline whitespace) : nothing
- \x{hex scalar value}; \ specified character (note the terminating semi-colon).

The result is unspecified if any other character in a string occurs after a backslash.

Except for a line ending, any character outside of an escape sequence stands for itself in the string literal. A line ending which is preceded by \(\text{(intraline whitespace)}\) expands to nothing (along with any trailing intraline whitespace), and can be used to indent strings for improved legibility. Any other line ending has the same effect as inserting a \n character into the string.

Examples:

"The word \"recursion\" has many meanings."
"Another example:\ntwo lines of text\nHere's a text
containing just one line\"x03B1; is named GREEK_SANS
\ref procedure returns character k of

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

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

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

In phrases such as “the characters of string beginning with index start and ending with index end,” it is understood that the index start is inclusive and the index end is exclusive. Thus if start and end are the same index, a null substring is referred to, and if start is zero and end is the length of string, then the entire string is referred to. It is an error if start is less than end.

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

Implementations may forbid certain characters from appearing in strings. For example, an implementation might support the entire Unicode repertoire, but only allow characters U+0000 to U+00FF (the Latin-1 repertoire) in strings. It is an error to pass such a forbidden character to make-string, string, string-set!, or string-fill!.

(string? obj) procedure
Returns #t if obj is a string, otherwise returns #f.

(make-string k) procedure
(make-string k char) procedure
The make-string procedure returns a newly allocated string of length k. If char is given, then all elements the characters 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. It is analogous to list.

(string-length string) procedure
Returns the number of characters in the given string.

(string-ref string k) procedure
It is an error if k is not a valid index of string. The string-ref procedure returns character k of
string using zero-origin indexing. There is no requirement for this procedure to execute in constant time.

(string-set! string k char) procedure
It is an error if k is not a valid index of string. The String-set! procedure stores char in element k of string and returns an unspecified value. There is no requirement for this procedure to execute in constant time.

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

(string=? char1 char2 char3 ...) procedure
Returns #t if the two all the strings are the same length and contain exactly the same characters in the same positions, otherwise returns #f.

(string-ci=? char1 char2 char3 ...) char library procedure
Returns #t if, after case-folding, all the strings are the same length and contain the same characters in the same positions, otherwise returns #f. Specifically, these procedures behave as if String-ci-string-foldcase were applied to their arguments before comparing them.

(string-ni=? char1 char2 char3 ...) procedure
Returns #t if, after an implementation-defined normalization, all the strings are the same length and contain the same characters in the same positions, otherwise returns #f. The intent is to provide a means of comparing strings that are considered equivalent in some situations but are represented by a different sequence of characters.

Specifically, an implementation which supports Unicode should use Unicode normalization NFC or NFDC as specified by Unicode TR#15. Implementations which only support ASCII or some other character set which provides no unambiguous representations of character sequences may define the normalization to be the identity operation, in which case string-ni=?treats upper and lower case letters as though they were the same character, but is equivalent to string=?treats upper and lower case as distinct characters.

(string<? string1 string2 string3 ...) procedure
(string-ci<? string1 string2 string3 ...) procedure
(string-ni<? string1 string2 string3 ...) procedure
(string->char string) char library procedure
(string-ci->char string) char library procedure
(string-ni->char string) char library procedure

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

These predicates are required to be transitive.

These procedures compare strings in an implementation-defined way. One approach is to make them the lexicographic extensions to strings of the corresponding orderings on characters. For example in that case, strings? = would be the lexicographic ordering on strings induced by the ordering char=? on characters. If and if the two strings differ in length but are the same up to the length of the shorter string, the shorter string in would be considered to be lexicographically less than the longer string. However, it is also permitted to use the natural ordering imposed by the internal representation of strings, or a more complex locale-specific ordering.

Implementations may generalize those and the In all cases, a pair of strings must satisfy exactly one of string<?, string>? and string-ci>? and must satisfy string<=? procedures to take more than two arguments, as with the corresponding numerical predicates, if and only if they do not satisfy string>? and string>=? if and only if they do not satisfy string<.

The "-ci" procedures behave as if they applied string-foldcase to their arguments before invoking the corresponding procedures without "-ci".

The "-ni" procedures behave as if they applied the implementation-defined normalization used by string-ni=? to their arguments before invoking the corresponding procedures without "-ni".

(string-upcase string) char library procedure
(string-downcase string) char library procedure
(string-foldcase string) char library procedure
These procedures apply the Unicode full string uppercasing, lowercasing, and case-folding algorithms to their arguments and return the result. If the result is equal to the argument, a new string need not be allocated. Note that language-sensitive mappings and foldings are not used. The result may differ in length from the argument. What is more, a few characters have case-mappings that depend on the surrounding context. For example, Greek capital sigma normally lowercases to Greek small sigma, but at the end of a word it downcases to Greek small final sigma instead. See UAX #29 (part of the Unicode Standard) for details.

(substring string start end) procedure
It is an error if start and end are not exact integers satisfying the inequality

\[ 0 \leq \text{start} \leq \text{end} \leq (\text{string-length} \text{ string}). \]

The Substrings substring procedure 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 are the concatenation of the characters in the given strings.

(string->list string) procedure
(list->string list) procedure
Stringstring->list returns a newly allocated list of the characters that make up the given string. Listlist->string returns a newly allocated string formed from the characters elements in the list list, which must be a list of characters. It is an error if any element is not a character. Stringstring->list and list->string are inverses so far as equal? is concerned.

(string-copy string) procedure
Returns a newly allocated copy of the given string.

(string-fill! string char) procedure
(string-fill! string char start end) procedure
Stores in every element of the given. If start and end are given, string-fill! stores fill in all the elements of string between start (inclusive) and end (exclusive). It is an error if fill is not a character or is forbidden in strings, or if start is less than end. If start and returns an unspecified value, endy are omitted, fill is stored in all the elements of string. In either case, an unspecified value is returned.

6.7.1. Vectors

6.8. Vectors

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

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

Vectors are written using the notation #((obj ...)). For example, a vector of length 3 containing the number zero in element 0, the list (2 2 2) in element 1, and the string "Anna" in element 2 can be written as following:

#(0 (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. It is an error to quote a constant:

'#(0 (2 2 2) "Anna")

⇒ #(0 (2 2 2) "Anna")

(vector? obj) procedure
Returns #t if obj is a vector, otherwise returns #f.

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

(vector obj ...) procedure
Returns a newly allocated vector whose elements contain the given arguments. Analogous. It is analogous to list.

(vector 'a 'b 'c) ⇒ (a b c)

(vector-length vector) procedure
Returns the number of elements in vector as an exact integer.

(vector-ref vector k) procedure
It is an error if k is not a valid index of vector. Vector-ref vector-ref returns the contents of element k of vector.
(vector-ref #'(1 1 2 3 5 8 13 21) 5) ➞ 8
(vector-ref #'(1 1 2 3 5 8 13 21) (let ((i (inexact->exact i))
 returned of by (if (inexact? i)
(list->vector
:::
(in
(vector->list
of the objects contained in the elements of the list-
Vector-
String
(vector->string
::::::::
elements
:::
:::::
(vector-copy a vector
::
::::::
:::::::::
newly character

(let ((vec (vector 0 '(2 2 2 2) "Anna"))
(vector-ref '#(1 1 2 3 5 8 13 21)
:::
(vector-copy vector start end fill)
procedure
Returns a newly allocated copy of the given vector. The value returned by elements of the new vector are the same (in the sense of eqv?) as the elements of the old.
The arguments start, end, and fill default to 0, the length of vector, and an implementation-specified value

(vector-copy vector)
procedure
(vector-copy vector start)
procedure
(vector-copy vector start end)
procedure
(vector-copy vector start end fill)
procedure

( round (* 2 (acos -1))) (if (inexact? i)
(vector-fill! vector fill)
procedure
(vector-fill! vector fill start end)
procedure

If start and end are given, vector-fill! stores fill (which can be any object) in all the elements of vector between start (inclusive) and end (exclusive). It is an error if start is less than end. If they are omitted, fill is stored in all the elements of vector. In either case, an unspecified value is returned.

6.9. Bytevectors

Bytevectors represent blocks of binary data. They are fixed-length sequences of bytes, where a byte is an exact integer in the range [0, 255]. A bytevector is typically more space-efficient than a vector containing the same values.

The length of a bytevector is the number of elements that it contains. This number is a non-negative integer that is fixed when the bytevector is created. The valid indexes of a bytevector are the exact non-negative integers less than the length of the bytevector, starting at index zero as with vectors.

(bytevector? obj) procedure

Returns #t if obj is a bytevector. Otherwise, #f is returned.

(make-bytevector k) procedure

(make-bytevector k byte) procedure

make-bytevector returns a newly allocated bytevector of length k. If byte is given, then all elements of the bytevector are initialized to byte, otherwise the contents of each element are unspecified.

(bytevector-length bytevector) procedure

Returns the length of bytevector in bytes as an exact integer.

(bytevector-u8-ref bytevector k) procedure

Returns the kth byte of bytevector.

(bytevector-u8-set! bytevector k byte) procedure

Stores byte as the kth byte of bytevector. The value returned by bytevector-u8-set! is unspecified.

(bytevector-copy bytevector) procedure
6.10. Control features

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

(procedure? obj)  procedure

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

(apply proc arg1 ... args)  procedure

Calls apply calls proc with the elements of the list (append (list arg1 ...) args) as the actual arguments.

(map proc list1 list2 ...)  procedure

It is an error if proc does not accept as many arguments as there are lists and return a single value. If more than one list is given and not all lists have the same length, map terminates when the shortest list runs out. Map(map) applies proc element-wise to the elements of the lists and returns a list of the results, in order. It is an error for proc to mutate any of the lists. The dynamic order in which proc is applied to the elements of the lists is unspecified. If multiple returns occur from map, the values returned by earlier returns are not mutated.

(string-map proc string1 string2 ...)  procedure

The arguments to It is an error if proc does not accept as many arguments as there are strings and return a single character. If more than one string is given and not all strings have the same length, string-map terminates when the shortest string runs out.
are like the arguments to map, but for each call 
for its side effects rather than for its values. Unlike 
map for each guaranteed to call on element-wise to the 
elements of the in order from the first element(s) to the 
last, and the value returned by strings and returns a string 
of the results in order. The dynamic order in which proc 
is applied to the elements of the strings is unspecified. 

If multiple 

Forces the value of (see section 6.2.5). If no value has 
been computed for the promise, then a value is computed 
and returned. The value of the promise is cached (or “memorized”) so that if it is forced a second time, 
the previously computed value is returned. It is an error if 
proc does not accept as many arguments as there are vec-
tors and return a single value. If more than one vector is 
given and not all vectors have the same length, 
vector-map terminates when the shortest vector runs out. vector-map 

Forces the value of (see section 6.2.5). If no value has 
been computed for the promise, then a value is computed 
and returned. The value of the promise is cached (or “memorized”) so that if it is forced a second time, 
the previously computed value is returned. It is an error if 
proc does not accept as many arguments as there are vec-
tors and return a single value. If more than one vector is 
given and not all vectors have the same length, vector-map 

Forces the value of (see section 6.2.5). If no value has 
been computed for the promise, then a value is computed 
and returned. The value of the promise is cached (or “memorized”) so that if it is forced a second time, 
the previously computed value is returned. It is an error if 
proc does not accept as many arguments as there are vec-
tors and return a single value. If more than one vector is 
given and not all vectors have the same length, vector-map 

Forces the value of (see section 6.2.5). If no value has 
been computed for the promise, then a value is computed 
and returned. The value of the promise is cached (or “memorized”) so that if it is forced a second time, 
the previously computed value is returned. It is an error if 
proc does not accept as many arguments as there are vec-
tors and return a single value. If more than one vector is 
given and not all vectors have the same length, vector-map 

Forces the value of (see section 6.2.5). If no value has 
been computed for the promise, then a value is computed 
and returned. The value of the promise is cached (or “memorized”) so that if it is forced a second time, 
the previously computed value is returned. It is an error if 
proc does not accept as many arguments as there are vec-
tors and return a single value. If more than one vector is 
given and not all vectors have the same length, vector-map 

It may be the case that there is no means by which a promise can be operationally distinguished from its forced value. That is, expressions like the following may evaluate to either or to, depending on the implementation:

\[
\text{let cc } \lambda (x) = (\text{if } (\text{negative? } x) \text{ then } (\text{exit } x)) \text{ in '}(54 \ 0 \ 37 \ -3 \ 245 \ 19)\]

\[
\text{#t}) \quad \Rightarrow \quad -3
\]

Some implementations may implement ``implicit forcing,'' whereby the value of a promise is forced by primitive procedures. The following examples show only the \text{call-with-current-continuation} procedure is for structured, non-local exits from loops or procedure bodies, but in fact \text{call-with-current-continuation} is extremely useful for implementing a wide variety of advanced control structures.

A common use of \text{call-with-current-continuation} is when controlling who is given credit for the computation. If the expression is evaluated at top level, for example, then the continuation might take the result, print it on the screen, prompt for the next input, evaluate it, and so on forever. Most of the time the continuation includes actions specified by user code, as in a continuation that will take the result, multiply it by the value stored in a local variable, add seven, and give the answer to the top level continuation to be printed. Normally these ubiquitous continuations are hidden behind the scenes and programmers do not think much about them. On rare occasions, however, a programmer may need needs to deal with continuations explicitly. The \text{Call-with-current-continuation} procedure allows Scheme programmers to do that by creating a procedure that acts just like the current continuation.

Most programming languages incorporate one or more special-purpose escape constructs with names like \text{exit}, \text{return}, or even \text{goto}. In 1965, however, Peter Landin [21] invented a general purpose escape operator called the J-operator. John Reynolds [30] described a simpler but equally powerful construct in 1972. The \text{catch} construct described by Sussman and Steele in the 1975 report on Scheme is exactly the same as Reynolds’s construct, though its name came from a less general construct in MacLisp. Several Scheme implementors noticed that the full power of the \text{catch} construct could be provided by a procedure instead of by a special syntactic construct, and the name \text{call-with-current-continuation} was
coined in 1982. This name is descriptive, but opinions differ on
the merits of such a long name, and some people prefer the
name call/cc instead.

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

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

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

(call-with-values (lambda () (values 4 5))
  (lambda (a b) b))
  ⇒ 5

(call-with-values * -)  ⇒ -1

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

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

• It is also exited when execution is within the dynamic
  extent and a continuation is invoked that was captured
  while not within the dynamic extent.

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

If a second call to dynamic-wind occurs within the dy-
namic extent of the call to thunk and then a continu-
ation is invoked in such a way that the before from these
two invocations of dynamic-wind are both to be called,
then the before associated with the first (inner) call to
dynamic-wind is called first.

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

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

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

6.11. Exceptions

This section describes Scheme’s exception-handling and
exception-raising procedures. For the concept of Scheme
exceptions, see section 4.3.2. See also 12.7 for the guard
syntax.

Exception handlers are one-argument procedures that
determine the action the program takes when an
exceptional situation is signalled. The system implicitly
maintains a current exception handler.

The program raises an exception by invoking the current
exception handler, passing it an object encapsulating
information about the exception. Any procedure accepting one argument may serve as an exception handler and any object may be used to represent an exception.

(with-exception-handler handler thunk) procedure

It is an error if handler does not accept one argument. It is also an error if thunk does not accept zero arguments. The with-exception-handler procedure returns the results of invoking thunk. Handler is installed as the current exception handler for the dynamic extent (as determined by dynamic-wind) of the invocation of thunk.

(raise obj) procedure

Raises an exception by invoking the current exception handler on obj. The handler is called with a continuation whose dynamic extent is that of the call to raise, except that the current exception handler is the one that was in place when the handler being called was installed. If the handler returns, an exception is raised in the same dynamic extent as the handler.

(raise-continuable obj) procedure

Raises an exception by invoking the current exception handler on obj. The handler is called with a continuation that is equivalent to the continuation of the call to raise-continuable, except that: (1) the current exception handler is the one that was in place when the handler being called was installed, and (2) if the handler being called returns, then it will again become the current exception handler. If the handler returns, the values it returns become the values returned by the call to raise-continuable.

(with-exception-handler (lambda (x) 100))

(error message obj . . .) procedure

Message should be a string. Raises an exception as if by calling raise on a newly allocated implementation-defined object which encapsulates the information provided by message, as well as any objs, known as the irritants. The procedure error-object? must return #t on such objects.

(define (null-list? l) (cond ((pair? l) #f) (else #t)))

(error-object? obj) procedure

Returns #t if obj is an object created by error or one of an implementation-defined set of objects, otherwise returns #f.

(error-object-message error-object) procedure

Returns the message encapsulated by error-object.

(error-object-irritants error-object) procedure

Returns a list of the irritants encapsulated by error-object.

6.12. Eval

(eval expression environment-specifier) eval library procedure

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

(eval '(* 7 3) (scheme-report-environment 6 7 ))

⇒ 21

(let ((f (eval ' (lambda (f x) (f x x))
 (null-environment 6 7 ))))

(f + 10))

⇒ 20

(scheme-report-environment version)

(eval ' (string? con) (display con))

(if (else (display "a warning has been issued")))

The procedure error-report-environment returns a specifier for an environment that is empty except for all bindings defined in this report that are either required or both optional and supported by the implementation. Null environment returns a specifier for an environment that is empty except for the (syntactic) bindings for all syntactic keywords defined in contains (null-environment), #t. It is defined either in the error library or in "null-list?" the other libraries of this report that are either required or both optional and supported by the implementation. Implementations must support this value of version.

Other implementations may also support other values of version can be used to specify environments matching past revisions of this report, but their support is not required. An implementation will signal an error if in which case
they should return an environment containing bindings corresponding to the corresponding version of the report. If version is neither 7 nor another value supported by the implementation, an error is signalled.

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

(setq environment (null-environment version)) eval library procedure

The null-environment procedure returns a specifier for an environment that contains only the bindings for all syntactic keywords defined either in the base library or in the other libraries of this report, provided that the implementation supports them.

(setq environment (environment list1 ... )) eval library procedure

This procedure returns a specifier for the environment that contains results by starting with an empty environment and then importing each list, considered as an import set, into it. (See section 5.3 for a description of import sets.) The bindings of the environment represented by the specifier are immutable.

(setq environment (interaction-environment)) repl library procedure

This procedure returns a specifier for an environment that contains an implementation-defined set of bindings, typically a superset of those listed in the report exported by (scheme base). The intent is that this procedure will return the environment in which the implementation would evaluate expressions dynamically typed entered by the user into a REPL.

6.13. Input and output

6.13.1. Ports

Ports represent input and output devices. To Scheme, an input port is a Scheme object that can deliver characters data upon command, while an output port is a Scheme object that can accept characters data. Whether the input and output port types are disjoint is implementation-dependent.

Different port types operate on different data. Scheme implementations are required to support textual ports and binary ports, but may also provide other port types.

A textual port supports reading or writing of individual characters from or to a backing store containing characters using read-char and write-char below, as well as operations defined in terms of characters such as read and write.

A binary port supports reading or writing of individual bytes from or to a backing store containing bytes using read-u8 and write-u8 below, as well as operations defined in terms of bytes. Whether the textual and binary port types are disjoint is implementation-dependent.

Ports can be used to access files, devices, and similar things on the host system on which the Scheme program is running.

(setq environment (call-with-input-file string proc)) file library procedure

(setq environment (call-with-output-file string proc)) file library procedure

It is an error if proc should be a procedure that accepts does not accept one argument. For call-with-input-file, the file named by string should already exist; for call-with-output-file, the effect is unspecified if the file already exists. These procedures call proc with one argument: the textual port obtained by opening the named file for input or output as if by open-input-file or open-output-file. If the file cannot be opened, an error is signalled. If proc returns, then the port is closed automatically and the value(s)-values yielded by the proc is(are) returned. If proc does not return, then the port will never 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.

(setq environment (call-with-port port proc)) procedure

Returns: It is an error if proc does not accept one argument.

The call-with-port procedure calls proc with port as an argument. If proc returns, port is closed automatically and the values returned by proc are returned.

(setq environment (input-port? obj)) procedure

(setq environment (output-port? obj)) procedure

(setq environment (textual-port? obj)) procedure

(setq environment (binary-port? obj)) procedure

(setq environment (port? obj)) procedure

These procedures return #t if obj is an input port, output port, respectively, otherwise returns , output port, textual port, binary port, or any kind of port, respectively. Otherwise they return #f.
(port-open? port) procedure

Returns #t if port is still open and capable of performing input or output, and #f otherwise.

(current-input-port) procedure

(current-output-port) procedure

(current-error-port) procedure

Returns the current default input or output port, port, output port, or error port (an output port), respectively. These procedures are parameter objects, which can be overridden with parameterize (see section 1.2.5). The initial bindings for these are system-defined textual ports.

(with-input-from-file string thunk) file library procedure
(with-output-to-file string thunk) file library procedure

should be a string naming a file, and should be a procedure of no arguments. It is an error if thunk does not accept zero arguments. For with-input-from-file, the file should be an input file. If the file named by string does not already exist, for with-output-to-file, the effect is unspecified if the file already exists. The file is opened for input or output—an input or output port connected to it as if by open-input-file or open-output-file, and the new port is made the default value returned by current-input-port or current-output-port (and is used by (read), (write obj), and so forth). The thunk is then called with no arguments. When the thunk returns, the port is closed and the previous default is restored. With-input-from-file and with-output-to-file return (s) the value(s) the values yielded by thunk. If an escape procedure is used to escape from the continuation of these procedures, their behavior is implementation-dependent.

(open-input-string string) file library procedure

Takes a string and returns a textual input port that delivers characters from the string.

(open-output-string) procedure

Returns a textual output port that will accumulate characters for retrieval by get-output-string.

(get-output-string port) procedure

It is an error if port was not created with open-output-string. Returns a string consisting of the characters that have been output to the port so far in the order they were output.

(open-input-bytevector bytevector) procedure

Takes a bytevector and returns a binary input port that delivers bytes from the bytevector.

(open-output-bytevector) procedure

Returns a binary output port that will accumulate bytes for retrieval by get-output-bytevector.

(get-output-bytevector port) procedure

It is an error if port was not created with open-output-bytevector. Returns a bytevector consisting of the bytes that have been output to the port so far in the order they were output.
6.13.2. Input

(read) read library procedure
(read port) read library procedure

Read read converts external representations of Scheme objects into the objects themselves. That is, it is a parser for the nonterminal (datum) (see sections 7.1.2 and 6.4). Read returns the next object parsable from the given textual 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 argument Port 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 textual 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 textual input port, without updating the port to point to the following character. If no more characters are available, an end of file object is returned. Port may be omitted, in which case it defaults to the value returned by current-input-port.

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

(read-line) procedure
(read-line port) procedure

Returns the next line of text available from the textual input port, updating the port to point to the following character. If an end of line is read, a string containing all of the text up to (but not including) the end of line is returned, and the port is updated to point just past the end of line. If an end of line is encountered before any end of line is read, but some characters have been read, a string containing those characters is returned. If an end of file is encountered before any characters are read, an end-of-file object is returned. For the purpose of this procedure, an end of line consists of either a linefeed character, a carriage return character, or a sequence of a carriage return character followed by a linefeed character. Port may be omitted, in which case it defaults to the value returned by current-input-port.

(eof-object? obj) procedure

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

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

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

Rationale: char-ready? exists to make it possible for a program to accept characters from interactive ports without getting stuck waiting for input. Any input editors associated with such ports must ensure that characters whose existence has been asserted by char-ready? cannot be rubbed out. If char-ready? were to return #f at end of file, a port at end of file would be indistinguishable from an interactive port that has no ready characters.

(read-u8) procedure
(read-u8 port) procedure

Returns the next byte available from the binary input port, updating the port to point to the following byte. If no more bytes 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-u8) procedure
(peek-u8 port) procedure

Returns the next byte available from the binary input port, without updating the port to point to the following byte. If no more bytes 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.
Reads the next length bytes, or as many as are available before the end of file, from the binary input port into a newly allocated bytevector in left-to-right order and returns the bytevector. If no bytes 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.

Reads the next end - start bytes, or as many as are available before the end of file, from the binary input port into bytevector in left-to-right order beginning at the start position. Returns the number of bytes read. If no bytes 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.

6.13.3. Output

Writes a written representation of obj to the given textual output port. Strings that appear in the written representation are enclosed in doublequotes, and within those strings backslash and doublequote characters are escaped by backslashes. Symbols that contain non-ASCII characters are escaped either with inline hex escapes or with vertical bars. Character objects are written using the \#\ notation. Shared list structure is represented using datum labels. write-write returns an unspecified value. The argument Port may be omitted, in which case it defaults to the value returned by current-output-port.

(u8-ready?) procedure
(u8-ready? port) procedure

Returns #t if a byte is ready on the binary input port and returns #f otherwise. If u8-ready? returns #t then the next read-u8 operation on the given port is guaranteed not to hang. If the port is at end of file then u8-ready? returns #t. Port may be omitted, in which case it defaults to the value returned by current-input-port.

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

Returns an unspecified value. Port may be omitted, in which case it defaults to the value returned by current-output-port.

Returns the byte to the given binary output port argument and returns an unspecified value. Port may be omitted, in which case it defaults to the value returned by current-output-port.

Returns the bytes of bytevector in left-to-right order to the binary output port. Port may be omitted, in which case it defaults to the value returned by current-output-port.

Returns an unspecified value. Port may be omitted, in which case it defaults to the value returned by current-output-port.

write-simple is the same as write, except that shared structure is not represented using datum labels. This may cause write-simple not to terminate if obj contains circular structure.
(write-partial-bytevector bytevector start end port) procedure

Writes the bytes of bytevector from start (inclusive) to end (exclusive) in left-to-right order to the binary output port. Port may be omitted, in which case it defaults to the value returned by current-output-port.

(flush-output-port) procedure

Flushes any buffered output from the buffer of output-port to the underlying file or device and returns an unspecified value. Port may be omitted, in which case it defaults to the value returned by current-output-port.

6.13.4. System interface

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

(load filename) load library procedure

(load filename environment-specifier) load library procedure

should be a string naming an implementation-dependent operation used to transform filename into the name of an existing file containing Scheme source code. The load procedure reads expressions and definitions from the file and evaluates them sequentially in the environment specified by environment-specifier. If environment-specifier is omitted, (interaction-environment) is assumed.

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

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

(file-exists? filename) file library procedure

The effect of (file-exists? filename) is not a string. The transcript on file-exists? to open procedure returns #t if the named file exists at the time the procedure is called, #f otherwise.

(delete-file filename) file library procedure

It is an error if filename is not a string. The delete-file procedure deletes the named file for output, and to cause a transcript of subsequent interaction between the user and the Scheme system to be written to the file. The transcript is ended by a call to if it exists and can be deleted, and returns an unspecified value. If the file does not exist or cannot be deleted, an error is signalled.

(command-line) process-context library procedure

Returns the command line passed to the process as a list of strings. The first string corresponds to the command name, and is implementation-dependent. It is an error to mutate any of these strings.

(exit) process-context library procedure

Exits the running program and communicates an exit value to the operating system. If no argument is supplied, the transcript-off exit procedure should communicate to the operating system that the program exited normally. If an argument is supplied, the exit procedure should translate the argument into an appropriate exit value for the operating system. If obj is #f, the exit is assumed to be abnormal.

(get-environment-variable name) process-context library procedure

Most operating systems provide each running process with an environment consisting of environment variables. (This environment is not to be confused with the Scheme environments that can be passed to eval; see section 6.1.2.) Both the name and value of an environment variable are strings. The procedure get-environment-variable returns the value of the environment variable name, or #f if the named environment variable is not found. get-environment-variable may use locale-setting information to encode the name and decode the value of the environment variable. It is an error if get-environment-variable can’t decode the value. It is also an error to mutate the resulting string.

\[ \text{(get-environment-variable "PATH")} \rightarrow \\text{"/usr/local/bin:/usr/bin:/bin"} \]

Returns the names and values of all the environment variables as an alist, where the car of each entry is the name of an environment variable and the cdr is its value, both as strings. The order of the list is unspecified. It is an error to mutate any of these strings.

\[ \text{(get-environment-variables)} \rightarrow \text{(("USER" ; "root") ("HOME" ; "/"))} \]
which restriction. at any transcript are returned implementations one by one:

(time library procedure

Returns an inexact number representing time on the International Atomic Time (TAI) scale. The value 0.0 represents ten seconds after midnight on January 1, which closes the transcript file. Only one transcript may be in progress at any time, though some implementations may relax this restriction. The values returned by these procedures are unspecified. 1970_TAI (equivalent to midnight Universal Time) and the value 1.0 represents one TAI second later. Neither high-accuracy nor high-precision values are required: in particular, returning Coordinated Universal Time plus a suitable constant may be the best an implementation can do.

(time library procedure

Returns the number of jiffies that have elapsed since an arbitrary, implementation-defined epoch. A jiffy is an implementation-defined fraction of a second which is defined by the return value of the jiffies-per-second procedure. The starting epoch is guaranteed to be constant during a run of the program, but may vary between different runs.

(time library procedure

Returns an exact integer representing the number of jiffies per SI second. This value is an implementation-specified constant.

(define (time-length) (let ((list (make-list 100000))) (/ (- (current-jiffy) start) (jiffies-per-second))))

7. Formal syntax and semantics
This chapter provides formal descriptions of what has already been described informally in previous chapters of this report.

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

All spaces in the grammar are for legibility. Case is insignificant; for example, #x1A and #X1A are equivalent. (empty) stands for the empty string.

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

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

(Inter-token space) may occur on either side of any token, but not within a token.

Identifiers, dot, numbers, characters, and booleans are terminated by a (delimiter) or by the end of the input.

The following four characters from the ASCII repertoire are reserved for future extensions to the language: [ ] { }

In addition to the identifier characters of the ASCII repertoire specified below, Scheme implementations may permit any additional repertoire of Unicode characters to be employed in identifiers, provided that each such character has a Unicode general category of Lu, Li, Lt, Lm, Lo, Mn, Mc, Me, Nd, Ni, No, Pd, Pc, Po, Sc, Sm, Sk, So, or Co, or is U+200C or U+200D (the zero-width non-joiner and joiner, respectively, which are needed for correct spelling in Persian, Hindi, and other languages). It is an error to use a non-Unicode character in symbols or identifiers.

All Scheme implementations must permit the escape sequence \x<hexdigits>; to appear in Scheme identifiers. If the character with the given Unicode scalar value is supported by the implementation, identifiers containing such a sequence are equivalent to identifiers containing the corresponding character.

(token) → (identifier) | (boolean) | (number) | (character) | (string) | ( | ) | #(| | #u8(| | \ | .
(delimiter) → (whitespace) | ( | ) | " | ;
(intraline whitespace) → (space or tab)
7. Formal syntax and semantics

The following rules for \langle num \rangle, \langle complex \rangle, \langle real \rangle, 
\langle uren \rangle, \langle uninteger \rangle, and \langle prefix \rangle are implicitly replicated for \(R = 2, 8, 10, \) and \(16.\) There are no rules for 
\langle decimal 2 \rangle, \langle decimal 8 \rangle, and \langle decimal 16 \rangle, which means that 
numbers containing decimal points or exponents are always in decimal radix. Although not shown below, all 
alphabetic characters used in the grammar of numbers may 
appear in either upper or lower case.

\langle num \rangle \rightarrow \langle prefix \rangle \langle complex \rangle 
\langle complex \rangle \rightarrow \langle real \rangle | \langle real \rangle \@ \langle real \rangle 
\langle real \rangle + \langle uren \rangle \@ - \langle uren \rangle \i \| \langle real \rangle \@ - \langle uren \rangle \i 
\langle uren \rangle + \langle uren \rangle \i \| \langle real \rangle \@ - \langle uren \rangle \i 
\langle uren \rangle + \langle uren \rangle \i \| \langle real \rangle \@ - \langle uren \rangle \i 
\langle real \rangle \rightarrow \langle sign \rangle \langle uren \rangle 
\langle sign \rangle \rightarrow \langle infinity \rangle \| \langle infinity \rangle \| + \langle infinity \rangle \| - \langle infinity \rangle 
\langle uren \rangle \rightarrow \langle uninteger \rangle \langle radix \rangle \langle exactness \rangle 
\langle exactness \rangle \rightarrow \langle infinity \rangle \| \langle infinity \rangle \| + \langle infinity \rangle \| - \langle infinity \rangle 
\langle complex \rangle \rightarrow \langle real \rangle \| \langle real \rangle \| \langle real \rangle \| \langle real \rangle 
\langle real \rangle \rightarrow \langle digit \rangle \langle digit \rangle \langle digit \rangle \langle digit \rangle \langle digit \rangle \langle digit \rangle \langle digit \rangle 
\langle digit \rangle \rightarrow 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 
\langle hex digit \rangle \rightarrow \langle digit \rangle 
\langle digit \rangle \rightarrow a | A | b | B | c | C | d | D | e | E | f | F 
\langle explicit sign \rangle \rightarrow + | - 
\langle special subsequent \rangle \rightarrow \langle explicit sign \rangle \| . \| @ 
\langle inline hex escape \rangle \rightarrow \x \langle hex scalar value \rangle 
\langle hex scalar value \rangle \rightarrow \langle hex digit \rangle *+ 
\langle explicit identifier \rangle \rightarrow \langle explicit sign \rangle 
\langle explicit sign \rangle \rightarrow \langle sign subsequent \rangle \langle subsequent \rangle *+ 
\langle explicit sign \rangle . \langle dot subsequent \rangle \langle subsequent \rangle *+ 
\langle non-digit \rangle \langle subsequent \rangle *+ 
\langle dot subsequent \rangle \rightarrow \langle dot \rangle \langle dot \rangle \langle dot \rangle \langle dot \rangle \langle dot \rangle 
\langle sign subsequent \rangle \rightarrow \langle initial \rangle | \langle explicit sign \rangle \| @ 
\langle symbol element \rangle \rightarrow 
\langle any character other than \langle vertical bar \rangle or \langle \| \rangle \}

\langle boolean \rangle \rightarrow #t | #f | #true | #false 

\langle character \rangle \rightarrow \#\| \langle any character \rangle 
\langle character name \rangle \rightarrow \#\| \langle character name \rangle 
\langle character sequence not containing \langle nested comment \rangle or \#| 
\langle character name \rangle \rightarrow \langle escape \rangle | \langle newline \rangle | \langle null \rangle | \langle return \rangle | \langle space \rangle | \langle tab \rangle 

\langle string \rangle \rightarrow " \langle string element \rangle * " 
\langle string element \rangle \rightarrow \langle any character other than " or \| \rangle 
\langle escape \rangle | \langle newline \rangle | \langle null \rangle | \langle return \rangle | \langle space \rangle | \langle tab \rangle

7.1.2. External representations

\langle Datum \rangle is what the \texttt{read} procedure (section \ref{sec:read}) 
successfully parses. Note that any string that parses as an 
expression will also parse as a \langle datum \rangle.
7.1.3. Expressions

The definitions in this and the following subsections assume that all the syntax keywords defined in this report have been properly imported from their libraries, and that none of them have been redefined or shadowed.

\[
\text{(expression)} \rightarrow \text{(identifier)} \\
| \text{(literal)} \\
| \text{(procedure call)} \\
| \text{(lambda expression)} \\
| \text{(conditional)} \\
| \text{(assignment)} \\
| \text{(derived expression)} \\
| \text{(macro use)} \\
| \text{(macro block)}
\]

\[
\text{(literal)} \rightarrow \text{(quotation)} | \text{(self-evaluating)} \\
\text{(self-evaluating)} \rightarrow \text{(boolean)} | \text{(number)} \\
| \text{(character)} | \text{(string)} | \text{(symbol)} | \text{(bytevector)} \\
\text{(quotation)} \rightarrow \text{'}(\text{datum}) | \text{(quote} \text{'}(\text{datum})\text{)} \\
\text{(procedure call)} \rightarrow (\text{operator} \text{'}(\text{operand})*) \\
\text{(operator)} \rightarrow \text{(expression)} \\
\text{(operand)} \rightarrow \text{(expression)}
\]

\[
\text{(lambda expression)} \rightarrow (\text{lambda} \text{'}(\text{formals}) \text{'(body)}) \\
\text{(formals)} \rightarrow (\text{identifier})* | \text{(identifier)} \\
| (\text{identifier}*) | \text{(identifier)} \\
\text{(body)} \rightarrow \text{'}(\text{definition})* \text{'(sequence)} \\
\text{(sequence)} \rightarrow \text{(command)}* \text{(expression)} \\
\text{(command)} \rightarrow \text{(expression)}
\]

\[
\text{(conditional)} \rightarrow \text{(if} \text{'}(\text{test}) \text{'}(\text{consequent}) \text{'}(\text{alternate})\text{)} \\
\text{(test)} \rightarrow \text{(expression)} \\
\text{(consequent)} \rightarrow \text{(expression)} \\
\text{(alternate)} \rightarrow \text{(expression)} | \text{empty} \\
\text{(assignment)} \rightarrow \text{(set}! \text{'}(\text{identifier}) \text{'(expression)})
\]

\[
\text{(derived expression)} \rightarrow \text{(cond} \text{'(cond clause)*)} \\
| \text{(cond} \text{'(cond clause)*) \text{'}(else} \text{'}(\text{sequence})\text{)}
\]

\[
\text{(case} \text{'(expression)} \\
\text{'}(case clause)*) \\
\text{(case} \text{'(expression)} \\
\text{'}(case clause)*) \\
\text{'}(else} \text{'}(\text{sequence})) \\
\text{(case} \text{'(expression)} \\
\text{'}(case clause)*) \\
\text{'}(else} \text{'}(\text{recipient})\text{)} \\
\text{(and} \text{'(test)*)} \\
\text{(or} \text{'(test)*)} \\
\text{(when} \text{'(test) \text{'(sequence))} \\
\text{(unless} \text{'(test) \text{'(sequence))} \\
\text{(let} \text{'}(\text{binding spec}) \text{'(body))} \\
\text{'}(let*} \text{'}(\text{binding spec}) \text{'(body))} \\
\text{(letrec} \text{'}(\text{binding spec}) \text{'(body))} \\
\text{(letrec*} \text{'}(\text{binding spec}) \text{'(body))} \\
\text{(let-values} \text{'}(\text{mv binding spec}) \text{'(body))} \\
\text{(let*values} \text{'}(\text{mv binding spec}) \text{'(body))} \\
\text{(begin} \text{'(sequence))} \\
\text{(do} \text{'(iteration spec)*)} \\
| \text{'}(test) \text{'(do result)} \\
\text{'(command)*)} \\
\text{(delay} \text{'(expression))} \\
\text{(lazy} \text{'(expression))} \\
\text{(parameterize} \text{'}(\text{expression}) \text{'}(\text{expression})) \text{'(body))} \\
\text{(guard} \text{'}(\text{identifier}) \text{'}(\text{cond clause}) \text{'(body))} \\
\text{(quasiquote) \text{'(expression))} \\
\text{(case-lambda} \text{'}(\text{case-lambda clause}) \text{'(body))}
\]

\[
\text{(cond clause)} \rightarrow \text{'}(\text{test}) \text{'}(\text{sequence}) \\
| \text{'}(test) \\
| \text{'}(test) \text{'}(\text{recipient}) \\
\text{(recipient)} \rightarrow \text{(expression)} \\
\text{(case clause)} \rightarrow \text{'}(\text{(datum)}) \text{'}(\text{sequence}) \\
| \text{'}(\text{(datum)}) \text{'}(\text{recipient}) \\
\text{(binding spec)} \rightarrow \text{'}(\text{identifier}) \text{'}(\text{expression}) \\
\text{(mv binding spec)} \rightarrow \text{'}(\text{formals}) \text{'}(\text{expression}) \\
\text{(iteration spec)} \rightarrow \text{'}(\text{identifier}) \text{'}(\text{init}) \text{'}(\text{step}) \\
| \text{'}(\text{identifier}) \text{'}(\text{init}) \\
\text{(case-lambda clause)} \rightarrow \text{'}(\text{formals}) \text{'}(\text{body}) \\
\text{(init)} \rightarrow \text{(expression)} \\
\text{(step)} \rightarrow \text{(expression)} \\
\text{(do result)} \rightarrow \text{'}(\text{sequence}) | \text{empty} \\
\text{(macro use)} \rightarrow \text{'}(\text{keyword}) \text{'}(\text{datum}) \\
\text{(keyword)} \rightarrow \text{'}(\text{identifier}) \\
\text{(macro block)} \rightarrow \text{'}(\text{let-syntax}) \text{'}(\text{syntax spec}) \text{'}(\text{body}) \\
| \text{'}(\text{letrec-syntax}) \text{'}(\text{syntax spec}) \text{'}(\text{body}) \\
\text{(syntax spec)} \rightarrow \text{'}(\text{keyword}) \text{'}(\text{transformer spec})
7.1.4. Quasiquotations

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

\[
\begin{align*}
\text{quotation} & \rightarrow \text{quotation} 1 \\
\text{qq template 0} & \rightarrow \text{expression} \\
\text{quotation D} & \rightarrow \text{qq template D} \\
\text{qq template D} & \rightarrow \text{(simple datum)} \\
\text{list qq template D} & \rightarrow \text{(qq template or splice D)*} \\
\text{unquotation D} & \rightarrow , \text{(qq template D - 1)} \\
\text{quote qq template D} & \rightarrow \text{(qq template D)} \\
\text{splicing unquotation D} & \rightarrow \text{QQ template D - 1)} \\
\text{unquote-splicing qq template D} & \rightarrow \text{(QQ template D - 1)}
\end{align*}
\]

In (quotation)s, a \( \text{list qq template D} \) can sometimes be confused with either an \( \text{unquotation D} \) or a \( \text{splicing unquotation D} \). The interpretation as an \( \text{unquotation} \) or \( \text{splicing unquotation D} \) takes precedence.

7.1.5. Transformers

\[
\begin{align*}
\text{transformer spec} & \rightarrow \\
\text{syntax rules} & \rightarrow \text{(identifier)*} \text{(syntax rule)*} \\
\text{syntax rules} & \rightarrow \text{(identifier) (identifier)*} \\
\text{syntax rule} & \rightarrow \text{(pattern) (identifier)} \\
\text{pattern} & \rightarrow \text{(pattern identifier)} \\
\text{underscore} & \rightarrow \text{(pattern)*} \\
\text{pattern D} & \rightarrow \text{(pattern)*} \\
\text{pattern* (pattern) (ellipsis) (pattern)*} & \rightarrow \text{(pattern)* (pattern) (ellipsis) (pattern)*} \\
\text{pattern* (pattern) (ellipsis) (pattern)*} & \rightarrow \text{(pattern)* (pattern) (ellipsis) (pattern)*} \\
\text{pattern datum} & \rightarrow \text{string} \\
\text{character} & \rightarrow \text{(pattern identifier)} \\
\text{boolean} & \rightarrow \text{(pattern identifier)} \\
\text{number} & \rightarrow \text{(pattern identifier)} \\
\text{template} & \rightarrow \text{(pattern identifier)} \\
\text{template element} & \rightarrow \text{(template)*} \\
\text{template element+} & \rightarrow \text{(template)} \\
\text{template+} & \rightarrow \text{(template)*} \\
\text{template datum} & \rightarrow \text{(template)} \\
\text{ellipses} & \rightarrow \text{any identifier except ...} \\
\text{identifier defaulting to ...} & \rightarrow \text{an identifier defaulting to ...} \\
\text{underscore} & \rightarrow \text{(the identifier _)}
\end{align*}
\]

7.1.6. Programs and definitions

\[
\begin{align*}
\text{program} & \rightarrow \text{command or definition)*} \\
\text{command or definition} & \rightarrow \text{command} \\
\text{definition} & \rightarrow \text{import (import set)+} \\
\text{begin (command or definition)+} & \rightarrow \text{definition} \rightarrow \text{define (identifier) (expression))} \\
\text{define ((identifier) (def forms) (body))} & \rightarrow \text{syntax definition} \\
\text{define-values (def forms) (body)} & \rightarrow \text{define-record-type (identifier)} \\
\text{constructor (identifier) (field spec)*} & \rightarrow \begin{align*}
\text{begin (definition)*} \\
\text{def forms} & \rightarrow \text{(identifier)*} \\
\text{(identifier)*} & \rightarrow \text{(identifier)*} \\
\text{constructor} & \rightarrow \text{(identifier) (field name)*} \\
\text{field spec} & \rightarrow \text{(field name) (accessor)} \\
\text{accessor} & \rightarrow \text{(identifier) (mutator)} \\
\text{syntax definition} & \rightarrow \text{(define-syntax (keyword) (transformer spec))}
\end{align*}
\]

7.1.7. Libraries

\[
\begin{align*}
\text{library} & \rightarrow \\
\text{define-library (library name) (library declaration)*} \\
\text{library name} & \rightarrow \text{(library name part)+} \\
\text{library name part} & \rightarrow \text{(identifier) | (integer 10)} \\
\text{library declaration} & \rightarrow \text{export (export spec)*} \\
\text{import (import set)*} & \rightarrow \text{begin (command or definition)*} \\
\text{include (string)+} & \rightarrow \text{include-ci (string)+} \\
\text{(cond-expand (cond-expand clause)*} & \rightarrow \text{(cond-expand (cond-expand clause)*} \\
\text{else (library declaration)*) & \rightarrow \text{display spec) \rightarrow (identifier)} \\
\text{rename (identifier) (identifier))} & \rightarrow \text{(import set) \rightarrow (library name)}
\end{align*}
\]
7.2. Formal semantics

This section provides a formal denotational semantics for the primitive expressions of Scheme and selected built-in procedures. The concepts and notation used here are described in [35]; the notation is summarized below:

- \((...)\) sequence formation
- \(s \downarrow k\) \(k\)th member of the sequence \(s\) (1-based)
- \(#s\) length of sequence \(s\)
- \(s \circ t\) concatenation of sequences \(s\) and \(t\)
- \(s \uparrow k\) drop the first \(k\) members of sequence \(s\)
- \(t \rightarrow a, b\) McCarthy conditional "if \(t\) then \(a\) else \(b\)"
- \(\rho[x/i]\) substitution "\(\rho\) with \(x\) for \(i\)"
- \(x\) in \(D\) injection of \(x\) into domain \(D\)
- \(x \uparrow D\) 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 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 

\[ \text{permute} \quad \text{and} \quad \text{unpermute}, \]

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 \(\text{new}\) is implementation-dependent, but it must obey the following axiom: if \(\text{new} \sigma \in L\), then \(\sigma(\text{new} \sigma \uparrow L) \downarrow 2 = \text{false}\).

The definition of \(K\) is omitted because an accurate definition of \(K\) would complicate the semantics without being very interesting.

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

\[ \mathcal{E}[(\lambda \alpha. \mathcal{E}(\text{P}^*)) \langle \text{undefined} \rangle \ldots\ldots] \]

where \(\mathcal{I}^*\) is the sequence of variables defined in \(P\), \(\mathcal{I}'\) is the sequence of expressions obtained by replacing every definition in \(P\) by an assignment, \(\langle \text{undefined} \rangle\) is an expression that evaluates to \text{undefined}, and \(\mathcal{E}\) is the semantic function that assigns meaning to expressions.

7.2.1. Abstract syntax

\[
\begin{align*}
K & \in \text{Con} & \text{constants, including quotations} \\
I & \in \text{Ide} & \text{identifiers (variables)} \\
E & \in \text{Exp} & \text{expressions} \\
\Gamma & \in \text{Com} = \text{Exp} & \text{commands}
\end{align*}
\]

\[
\text{Exp} \rightarrow K \mid I \mid (E_0 \ E^*)
\]

\[
\begin{align*}
& | \ (\lambda \alpha. (I^*) \ E^*) \\
& | \ (\lambda \alpha. (I^* \ . \ I) \ E^*) \\
& | \ (\lambda \alpha. I \ E^*) \\
& | \ (\text{if} \ E_0 \ E_1 \ E_2) \ | \ (\text{if} \ E_0 \ E_1) \\
& | \ (\text{set}! \ I \ E)
\end{align*}
\]

7.2.2. Domain equations

\[
\begin{align*}
\alpha & \in L & \text{locations} \\
\nu & \in N & \text{natural numbers} \\
T & = \{\text{false}, \text{true}\} & \text{booleans} \\
Q & & \text{symbols} \\
H & & \text{characters} \\
R & & \text{numbers} \\
E_p & = L \times L \times T & \text{pairs} \\
E_v & = L^* \times T & \text{vectors} \\
E_s & = L^* \times T & \text{strings} \\
M & = \{\text{false}, \text{true}, \text{null}, \text{undefined}, \text{unspecified}\} & \text{miscellaneous}
\end{align*}
\]

\[
\begin{align*}
\phi & \in F & = L \times (E^* \rightarrow K \rightarrow C) & \text{procedure values} \\
\epsilon & \in E & = Q \uparrow H \uparrow R \uparrow E_p \uparrow E_v \uparrow E_s \uparrow M \uparrow F & \text{expressed values} \\
\sigma & \in S & = L \rightarrow (E \times T) & \text{stores} \\
\rho & \in U & = \text{Ide} \rightarrow L & \text{environments} \\
\theta & \in C & = S \rightarrow A & \text{command continuations} \\
\kappa & \in K & = E^* \rightarrow C & \text{expression continuations}
\end{align*}
\]

\[A \mid X\]

7.2.3. Semantic functions

\[
\begin{align*}
K : \text{Con} \rightarrow E \\
\mathcal{E} : \text{Exp} \rightarrow U \rightarrow K \rightarrow C \\
\mathcal{E}^* : \text{Exp}^* \rightarrow U \rightarrow K \rightarrow C \\
C : \text{Com}^* \rightarrow U \rightarrow C \rightarrow C
\end{align*}
\]

Definition of \(K\) deliberately omitted.

\[
\mathcal{E}[K] = \lambda \alpha. \text{send}(K[K]) \kappa
\]
7. Formal syntax and semantics

7.2.4. Auxiliary functions

lookup : U → Ide → L
lookup = λρl. ρl
extends : U → Ide* → L* → U
extends =
λρlα*. #I* = 0 → ρ,
extends (ρ[(α* ⊕ 1)/(I* ⊕ 1)]) (I* ⊕ 1) (α* ⊕ 1)
wrong : X → C [implementation-dependent]
send : E → K → C
send = λκκ. κε
single : (E → C) → K
single =
λψε*. #ε* = 1 → ψ(ε* ⊕ 1),
wrong “wrong number of return values”
new : S → (L + {error}) [implementation-dependent]
hold : L → K → C
hold = λκσ. send(σa ⊕ 1)κσ
assign : L → E → C → C
assign = λκσ. σ{ϵ, true}/κσ
update : L → E → S → S
update = λκσ. σ{ε, true}/κσ

tievals = (L* → C) → E* → C
tievals =
λψε*. #ε* = 0 → ψ(ε* ⊕ 1),
new σ ∈ L → tievals (λα*, ψ(⟨new σ | L⟩ ⊕ α*))
(ε* ⊕ 1)
(update (new σ | L)(ε* ⊕ 1)σ),
wrong “out of memory” σ

tievalsrest : (L* → C) → E* → N → C
tievalsrest =
λψε*ν. list (dropfirst ε*ν)
(singlε. tievals ψ ((takefirst ε*ν) ⊕ 〈ε〉))
dropfirst = λln. n = 0 → l, dropfirst(l ⊕ 1)(n ⊕ 1)
takefirst = λln. n = 0 → 〈〉, (l ⊕ 1) ⊕ 〈takefirst(l ⊕ 1)(n ⊕ 1)〉
truish : E → T
truish = λε. ε = faalse → false, true
permute : Exp* → Exp* [implementation-dependent]
unpermute : E* → E* [inverse of permute]
appliicate : E → E* → K → C
appliicate =
λκε*κ. ε ∈ F → (ε | F ⊕ 1)ε*κ, wrong “bad procedure”
oneary : (E → K → C) → (E* → K → C)
oneary =
λζκes. #es = 1 → ez(ε* ⊕ 1)κ,
wrong “wrong number of arguments”
twoary : (E → E → K → C) → (E* → K → C)
twoary =
λζκes. #es = 2 → ez(ε* ⊕ 1)(es ⊕ 2)κ,
wrong “wrong number of arguments”
64 Revised\(^7\) Scheme

\[
\text{list} : E^* \rightarrow K \rightarrow C
\]

\[
\text{list} = \lambda e^* . \# e^* = 0 \rightarrow \text{send null } \kappa,
\text{list}(e^* \uparrow 1)(\text{single}(\lambda e . \text{cons}(e^* \downarrow 1, e)\kappa))
\]

\[
\text{cons} : E^* \rightarrow K \rightarrow C
\]

\[
\text{cons} = \text{twoarg}(\lambda \epsilon \kappa. \text{new } \sigma \in L \rightarrow
\langle \lambda \sigma^\prime . \text{new } \sigma^\prime \in L \rightarrow
\text{send}((\text{new } \sigma^\prime | L, \text{new } \sigma^\prime | L, \text{true})\kappa)
\text{in } E)
\rangle (\text{update}(\text{new } \sigma^\prime | L)\epsilon^\sigma),
\text{wrong} \text{“out of memory” } \sigma)
\]

\[
\text{less} : E^* \rightarrow K \rightarrow C
\]

\[
\text{less} = \text{twoarg}(\lambda \epsilon \kappa . \epsilon \in R \land \epsilon \in R) \rightarrow
\text{send}((\epsilon_1 \land \epsilon \in R \rightarrow \text{true}, \false)\kappa),
\text{wrong} \text{“non-numeric argument to } <^\prime\text{”}
\]

\[
\text{add} : E^* \rightarrow K \rightarrow C
\]

\[
\text{add} = \text{twoarg}(\lambda \epsilon \kappa . \epsilon \in R \land \epsilon \in R) \rightarrow
\text{send}((\epsilon_1 \land \epsilon \in R \rightarrow \text{true}, \false)\kappa),
\text{wrong} \text{“non-numeric argument to } +^\prime\text{”}
\]

\[
\text{car} : E^* \rightarrow K \rightarrow C
\]

\[
\text{car} = \text{onearg}(\lambda \kappa \epsilon . \epsilon \in E \rightarrow \text{hold}(\epsilon | E \downarrow 1)\kappa),
\text{wrong} \text{“non-pair argument to } \text{car}^\prime\text{”}
\]

\[
\text{cdr} : E^* \rightarrow K \rightarrow C
\]

\[
\text{cdr} = \text{similar to } \text{car}
\]

\[
\text{setcar} : E^* \rightarrow K \rightarrow C
\]

\[
\text{setcar} = \text{twoarg}(\lambda \epsilon \kappa . \epsilon \in E \rightarrow
\langle \epsilon_1 | E \downarrow 3 \rightarrow \text{assign}(\epsilon_1 | E \downarrow 1)\epsilon_2
\rangle
\text{send unspecified } \kappa,
\text{wrong} \text{“immutable argument to } \text{set-car}^\prime\text{”,}
\text{wrong} \text{“non-pair argument to } \text{set-car}^\prime\text{”}
\]

\[
\text{equiv} : E^* \rightarrow K \rightarrow C
\]

\[
\text{equiv} = \text{twoarg}(\lambda \epsilon \kappa . \epsilon \in M \land \epsilon \in M) \rightarrow
\text{send}((\epsilon_1 | M = \epsilon_2 | M \rightarrow \text{true}, \false)\kappa),
\text{send}((\epsilon_1 | Q = \epsilon_2 | Q \rightarrow \text{true}, \false)\kappa),
\text{send}((\epsilon_1 | H = \epsilon_2 | H \rightarrow \text{true}, \false)\kappa),
\text{send}(\epsilon_1 | R \land \epsilon \in R) \rightarrow
\text{send}((\epsilon_1 | R \land \epsilon \in R \rightarrow \text{true}, \false)\kappa),
\text{send}(\epsilon_1 | E \land \epsilon \in E) \rightarrow
\text{send}(\epsilon_1 | (p_1 \downarrow 1) = (p_2 \downarrow 1) \land
\text{true}, \false)
\rangle
\text{send unspecified } \kappa,
\text{wrong} \text{“non-pair argument to } \text{equiv}^\prime\text{”}
\]

7.3. Derived expression types

This section gives macro definitions for the derived expression types in terms of the primitive expression types (literal, variable, call, lambda, if, and set!). See section 6.19 for a possible definition of \text{except for delay quasiquote}.

\[
\text{define-syntax cond}
\]

\[
\text{(syntax-rules (else =>)}
\text{ ((cond (else result1 result2 ...))}
\text{ (begin result1 result2 ...))}
\text{ ((cond (test => result))}
\text{ (let ((temp test))}
\text{ (if temp (result temp)))}
\text{ ((cond (test => result) clause1 clause2 ...))}
\text{ (let ((temp test))}
\text{ (if temp (result temp)
\text{ (cond clause1 clause2 ...))))}
\]
((cond (test)) test)
((cond (test) clause1 clause2 ...)
 (let ((temp test))
  (if temp
   temp
   (cond clause1 clause2 ...)))))
((cond (test result1 result2 ...))
 (if test (begin result1 result2 ...) 
  (cond clause1 clause2 ...)))
((cond (test result1 result2 ...) clause1 clause2 ...)
 (if test 
  (begin result1 result2 ...) 
  (cond clause1 clause2 ...))))

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

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

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

(define-syntax let
  (syntax-rules ()
    ((let ((name val) ...) body1 body2 ...) val ...))

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

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

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

(define-syntax let
  (syntax-rules ()
    ((let ((name val) ...) body1 body2 ...) val ...))

(define-syntax letrec
  (syntax-rules ()
    ((letrec ([(var1 init1) ...]) body ...) (letrec "generate temp names"
      (var1 ...) ()
      (let ((var1 <undefined>) ...) (set! var1 init1)
        ...
        body ...))
    ((letrec ([(var1 init1) ...]) body ...) (letrec "generate temp names"
      ()
      (let ((var1 <undefined>) ...) (set! var1 init1)
        ...
        body ...)))

(define-syntax let-values
  (syntax-rules ()
    ((let-values ([(binding ...) body0 body1 ...])
      (let-values "bind" () tmps body) (let tmps body))
    ((let-values ([(binding ...) body0 body1 ...])
      (let-values "bind" () tmps body) (let tmps body))))

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

(define-syntax letrec
  (syntax-rules ()
    ((letrec ([(var1 init1) ...]) body ...) (letrec "generate temp names"
      (var1 ...) ()
      (let ((var1 <undefined>) ...) (set! var1 init1)
        ...
        body ...))
    ((letrec ([(var1 init1) ...]) body ...) (letrec "generate temp names"
      ()
      (let ((var1 <undefined>) ...) (set! var1 init1)
        ...
        body ...))))

(define-syntax let-values
  (syntax-rules ()
    ((let-values ([(binding ...) body0 body1 ...])
      (let-values "bind" () tmps body) (let tmps body))
    ((let-values ([(binding ...) body0 body1 ...])
      (let-values "bind" () tmps body) (let tmps body))))

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

(define-syntax letrec
  (syntax-rules ()
    ((letrec ([(var1 init1) ...]) body ...) (letrec "generate temp names"
      (var1 ...) ()
      (let ((var1 <undefined>) ...) (set! var1 init1)
        ...
        body ...))
    ((letrec ([(var1 init1) ...]) body ...) (letrec "generate temp names"
      ()
      (let ((var1 <undefined>) ...) (set! var1 init1)
        ...
        body ...))))

(define-syntax let-values
  (syntax-rules ()
    ((let-values ([(binding ...) body0 body1 ...])
      (let-values "bind" () tmps body) (let tmps body))
    ((let-values ([(binding ...) body0 body1 ...])
      (let-values "bind" () tmps body) (let tmps body))))
The following definition of \texttt{begin} does not make use of the ability to write more than one expression in the body of a lambda expression. In any case, note that these rules apply only if the body of the \texttt{begin} contains no definitions.

\begin{verbatim}
(define-syntax begin
  (syntax-rules ()
    ((begin exp ...)
      (lambda () exp ...))))
\end{verbatim}

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

\begin{verbatim}
(define-syntax do
  (syntax-rules ()
    ((do ((var init step ...) ...) ...)
      (test expr ...) command ...)
    (letrec
      ((loop
        (lambda (var ...) (if test
          (begin (if \#f) expr ...)
          command ...
          (loop (do "step" var step ...) ...))))
      (loop init ...)))(do "step" x y)
      (do "step" x y)
\end{verbatim}
Appendix A. Standard Libraries

This section lists the exports provided by the standard libraries. The libraries are factored so as to separate features which might not be supported by all implementations, or which might be expensive to load.

The scheme library prefix is used for all standard libraries, and is reserved for use by future standards.

Base Library

The (scheme base) library exports many of the procedures and syntax bindings that are traditionally associated with Scheme.

<table>
<thead>
<tr>
<th>Procedure</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
</tr>
<tr>
<td>+</td>
</tr>
<tr>
<td>-</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>/</td>
</tr>
<tr>
<td>&lt;</td>
</tr>
<tr>
<td>&lt;=</td>
</tr>
<tr>
<td>=&gt;</td>
</tr>
<tr>
<td>abs</td>
</tr>
<tr>
<td>and</td>
</tr>
<tr>
<td>append</td>
</tr>
<tr>
<td>apply</td>
</tr>
<tr>
<td>assoc</td>
</tr>
<tr>
<td>assq</td>
</tr>
<tr>
<td>assv</td>
</tr>
<tr>
<td>begin</td>
</tr>
<tr>
<td>binary-port?</td>
</tr>
<tr>
<td>boolean?</td>
</tr>
<tr>
<td>bytevector-copy</td>
</tr>
<tr>
<td>bytevector-copy!</td>
</tr>
<tr>
<td>bytevector-copy-partial</td>
</tr>
<tr>
<td>bytevector-copy-partial!</td>
</tr>
<tr>
<td>bytevector-length</td>
</tr>
<tr>
<td>bytevector-u8-ref</td>
</tr>
<tr>
<td>bytevector-u8-set!</td>
</tr>
<tr>
<td>bytevector?</td>
</tr>
<tr>
<td>caaar</td>
</tr>
<tr>
<td>caadr</td>
</tr>
<tr>
<td>caadr!</td>
</tr>
<tr>
<td>caar</td>
</tr>
<tr>
<td>cadar</td>
</tr>
<tr>
<td>cadar!</td>
</tr>
<tr>
<td>caddr</td>
</tr>
<tr>
<td>caddr</td>
</tr>
<tr>
<td>caddr!</td>
</tr>
<tr>
<td>cadr</td>
</tr>
<tr>
<td>cadr!</td>
</tr>
<tr>
<td>cddar</td>
</tr>
<tr>
<td>cddar</td>
</tr>
<tr>
<td>cddar!</td>
</tr>
<tr>
<td>cdddr</td>
</tr>
<tr>
<td>cddr</td>
</tr>
<tr>
<td>ceiling</td>
</tr>
<tr>
<td>char-&gt;integer</td>
</tr>
<tr>
<td>char-ready?</td>
</tr>
<tr>
<td>char&lt;=?</td>
</tr>
<tr>
<td>char&lt;?</td>
</tr>
<tr>
<td>char=E?</td>
</tr>
<tr>
<td>char=&gt;?</td>
</tr>
<tr>
<td>char?&gt;&gt;</td>
</tr>
<tr>
<td>close-input-port</td>
</tr>
<tr>
<td>close-output-port</td>
</tr>
<tr>
<td>complex?</td>
</tr>
<tr>
<td>cond</td>
</tr>
<tr>
<td>cond-expand</td>
</tr>
<tr>
<td>cons</td>
</tr>
<tr>
<td>current-error-port</td>
</tr>
<tr>
<td>current-input-port</td>
</tr>
<tr>
<td>current-output-port</td>
</tr>
<tr>
<td>define</td>
</tr>
<tr>
<td>define-record-type</td>
</tr>
<tr>
<td>define-syntax</td>
</tr>
<tr>
<td>define-values</td>
</tr>
<tr>
<td>denominator</td>
</tr>
<tr>
<td>do</td>
</tr>
<tr>
<td>dynamic-wind</td>
</tr>
<tr>
<td>else</td>
</tr>
<tr>
<td>eof-object?</td>
</tr>
<tr>
<td>eq?</td>
</tr>
<tr>
<td>equal?</td>
</tr>
<tr>
<td>eqv?</td>
</tr>
<tr>
<td>error</td>
</tr>
<tr>
<td>error-object-irritants</td>
</tr>
<tr>
<td>error-object-message</td>
</tr>
<tr>
<td>even?</td>
</tr>
<tr>
<td>exact-&gt;inexact</td>
</tr>
<tr>
<td>exact?</td>
</tr>
<tr>
<td>expt</td>
</tr>
<tr>
<td>for-each</td>
</tr>
<tr>
<td>flush-output-port</td>
</tr>
<tr>
<td>gcd</td>
</tr>
<tr>
<td>get-output-bytevector</td>
</tr>
<tr>
<td>get-output-string</td>
</tr>
<tr>
<td>guard</td>
</tr>
<tr>
<td>if</td>
</tr>
<tr>
<td>import</td>
</tr>
<tr>
<td>inexact?</td>
</tr>
<tr>
<td>input-port?</td>
</tr>
<tr>
<td>integer-&gt;char</td>
</tr>
<tr>
<td>integer?</td>
</tr>
<tr>
<td>lambda</td>
</tr>
<tr>
<td>lcm</td>
</tr>
<tr>
<td>length</td>
</tr>
<tr>
<td>let</td>
</tr>
<tr>
<td>let*</td>
</tr>
<tr>
<td>let-values</td>
</tr>
<tr>
<td>let-syntax</td>
</tr>
<tr>
<td>list</td>
</tr>
<tr>
<td>list-&gt;string</td>
</tr>
<tr>
<td>list-&gt;vector</td>
</tr>
<tr>
<td>list-copy</td>
</tr>
<tr>
<td>list-ref</td>
</tr>
<tr>
<td>list-set!</td>
</tr>
<tr>
<td>list-tail</td>
</tr>
<tr>
<td>list?</td>
</tr>
<tr>
<td>make-bicycle</td>
</tr>
<tr>
<td>make-parameter</td>
</tr>
<tr>
<td>make-string</td>
</tr>
<tr>
<td>make-vector</td>
</tr>
<tr>
<td>map</td>
</tr>
<tr>
<td>max</td>
</tr>
<tr>
<td>member</td>
</tr>
<tr>
<td>memq</td>
</tr>
<tr>
<td>memv</td>
</tr>
<tr>
<td>min</td>
</tr>
<tr>
<td>modulo</td>
</tr>
<tr>
<td>negative?</td>
</tr>
<tr>
<td>newline</td>
</tr>
<tr>
<td>not</td>
</tr>
<tr>
<td>null?</td>
</tr>
<tr>
<td>number-&gt;string</td>
</tr>
<tr>
<td>number?</td>
</tr>
<tr>
<td>numerator</td>
</tr>
<tr>
<td>odd?</td>
</tr>
<tr>
<td>open-input-bytevector</td>
</tr>
<tr>
<td>open-input-string</td>
</tr>
<tr>
<td>open-output-bytevector</td>
</tr>
<tr>
<td>open-output-string</td>
</tr>
<tr>
<td>or</td>
</tr>
<tr>
<td>output-port?</td>
</tr>
<tr>
<td>pair?</td>
</tr>
<tr>
<td>parameterize</td>
</tr>
<tr>
<td>peek-char</td>
</tr>
<tr>
<td>peek-u8</td>
</tr>
<tr>
<td>port-open?</td>
</tr>
<tr>
<td>port?</td>
</tr>
<tr>
<td>positive?</td>
</tr>
<tr>
<td>procedure?</td>
</tr>
<tr>
<td>raise</td>
</tr>
<tr>
<td>raise-continuable</td>
</tr>
<tr>
<td>rational?</td>
</tr>
<tr>
<td>rationalize</td>
</tr>
<tr>
<td>read-bytevector</td>
</tr>
<tr>
<td>read-bytevector!</td>
</tr>
<tr>
<td>read-char</td>
</tr>
<tr>
<td>read-line</td>
</tr>
<tr>
<td>read-u8</td>
</tr>
<tr>
<td>real?</td>
</tr>
<tr>
<td>remainder</td>
</tr>
<tr>
<td>reverse</td>
</tr>
<tr>
<td>round</td>
</tr>
<tr>
<td>set!</td>
</tr>
<tr>
<td>set-car!</td>
</tr>
<tr>
<td>set-cdr!</td>
</tr>
<tr>
<td>string</td>
</tr>
<tr>
<td>string-&gt;list</td>
</tr>
<tr>
<td>string-&gt;number</td>
</tr>
<tr>
<td>string-&gt;symbol</td>
</tr>
<tr>
<td>string-&gt;utf8</td>
</tr>
<tr>
<td>string-&gt;vector</td>
</tr>
<tr>
<td>string-append</td>
</tr>
<tr>
<td>string-copy</td>
</tr>
<tr>
<td>string-fill!</td>
</tr>
<tr>
<td>string-for-each</td>
</tr>
<tr>
<td>string-length</td>
</tr>
<tr>
<td>string-map</td>
</tr>
<tr>
<td>string-ref</td>
</tr>
<tr>
<td>string-set!</td>
</tr>
<tr>
<td>string&lt;?</td>
</tr>
<tr>
<td>string&lt;=?</td>
</tr>
<tr>
<td>string&gt;=?</td>
</tr>
<tr>
<td>string?&gt;&gt;</td>
</tr>
<tr>
<td>symbol-&gt;string</td>
</tr>
<tr>
<td>symbol?</td>
</tr>
<tr>
<td>syntax-error</td>
</tr>
<tr>
<td>syntax-rules</td>
</tr>
<tr>
<td>textual-port?</td>
</tr>
<tr>
<td>truncate</td>
</tr>
<tr>
<td>u8-ready?</td>
</tr>
<tr>
<td>unless</td>
</tr>
<tr>
<td>unquote</td>
</tr>
<tr>
<td>unquote-splicing</td>
</tr>
<tr>
<td>utf8-&gt;string</td>
</tr>
<tr>
<td>values</td>
</tr>
<tr>
<td>vector</td>
</tr>
<tr>
<td>vector-&gt;list</td>
</tr>
<tr>
<td>vector-copy</td>
</tr>
<tr>
<td>vector-fill!</td>
</tr>
<tr>
<td>vector-for-each</td>
</tr>
<tr>
<td>vector-length</td>
</tr>
<tr>
<td>vector-map</td>
</tr>
<tr>
<td>vector-ref</td>
</tr>
<tr>
<td>vector-set!</td>
</tr>
<tr>
<td>when</td>
</tr>
<tr>
<td>with-exception-handler</td>
</tr>
<tr>
<td>write-bytevector</td>
</tr>
<tr>
<td>write-char</td>
</tr>
<tr>
<td>write-partial-bytevector</td>
</tr>
<tr>
<td>write-u8</td>
</tr>
<tr>
<td>zero?</td>
</tr>
</tbody>
</table>

Inexact Library

The (scheme inexact) library exports procedures which are typically only useful with inexact values.

<table>
<thead>
<tr>
<th>Procedure</th>
</tr>
</thead>
<tbody>
<tr>
<td>acos</td>
</tr>
<tr>
<td>asin</td>
</tr>
<tr>
<td>atan</td>
</tr>
<tr>
<td>cos</td>
</tr>
<tr>
<td>exp</td>
</tr>
<tr>
<td>finite?</td>
</tr>
<tr>
<td>log</td>
</tr>
<tr>
<td>nan?</td>
</tr>
<tr>
<td>sin</td>
</tr>
<tr>
<td>sqrt</td>
</tr>
<tr>
<td>tan</td>
</tr>
</tbody>
</table>
Complex Library

The (scheme complex) library exports procedures which are typically only useful with complex values.

- angle
- imag-part
- magnitude
- make-polar
- make-rectangular
- real-part

Division Library

The (scheme division) library exports procedures for integer division.

- ceiling-quotient
- ceiling-remainder
- centered-quotient
- centered-remainder
- euclidean-quotient
- euclidean-remainder
- floor-quotient
- floor-remainder
- round-quotient
- round-remainder
- truncate-quotient
- truncate-remainder
- truncate/

Lazy Library

The (scheme lazy) library exports procedures and syntax keywords for lazy evaluation.

- delay
- eager
- force
- lazy

Case-Lambda Library

The (scheme case-lambda) library exports the case-lambda syntax.

- case-lambda

Eval Library

The (scheme eval) library exports procedures for evaluating Scheme data as programs.

- environment
- eval
- null-environment
- scheme-report-environment

Repl Library

The (scheme repl) library exports the interaction-environment procedure.

- interaction-environment

Process Context Library

The (scheme process-context) library exports procedures for accessing with the program’s calling context.

- command-line
- exit
- get-environment-variable
- get-environment-variables

Load Library

The (scheme load) library exports procedures for loading Scheme expressions from files.

- load

File Library

The (scheme file) library provides procedures for accessing files.

- call-with-input-file
- call-with-output-file
- delete-file
- file-exists?
- open-binary-input-file
- open-binary-output-file
- open-input-file
- open-output-file
- with-input-from-file
- with-output-to-file

Read Library

The (scheme read) library provides procedures for reading Scheme objects.

- read

Write Library

The (scheme write) library provides procedures for writing Scheme objects.

- display
- write
- write-simple

Char Library

The (scheme char) library provides procedures for dealing with Unicode character operations.

- char-alphabetic?
- char-ci<=?
- char-ci=?
- char-ci>=?
- char-ci?>
- char-downcase
- char-foldcase
- char-lower-case?
- char-numeric?
- char-upcase
- char-upper-case?
- char-whitespace?
- digit-value
- string-ci<=?
- string-ci=?
- string-ci>?
- string-ci>=?
- string-foldcase
- string-upcase

Char Normalization Library

The (scheme char normalization) library provides procedures for dealing with Unicode normalization operations.

- string-ni<=?
- string-ni=?
- string-ni>?
Appendix B. Standard Feature Identifiers

An implementation may provide any or all of the feature identifiers listed below, as well as any others that it chooses, but must not provide a feature identifier if it does not provide the corresponding feature. These features are used by cond-expand.

r7rs
All R7RS Scheme implementations have this feature.

exact-closed
All algebraic operations except / produce exact values given exact inputs.

ratios
/ with exact arguments produces an exact result when the divisor is nonzero.

exact-complex
Exact complex numbers are provided.

ieee-float
Inexact numbers are IEEE 754 floating point values.

full-unicode
All Unicode codepoints are supported as characters (except the surrogates).

windows
This Scheme implementation is running on Windows.

posix
This Scheme implementation is running on a POSIX system.

unix, darwin, linux, bsd, freebsd, solaris, ...
Operating system flags (more than one may apply).

i386, x86-64, ppc, sparc, jvm, clr, llvm, ...
CPU architecture flags.

ilp32, lp64, ilp64, ...
C memory model flags.

big-endian, little-endian
Byte order flags.

⟨name⟩
The name of this implementation.

⟨name-version⟩
The name and version of this implementation.
NOTES

Language changes since R^5RS

This section enumerates the differences between this report and the “Revised^5 report” [2].

The list is incomplete and subject to change while this report is in draft status.

- Various minor ambiguities and unclarities in R^5RS have been cleaned up.
- Libraries have been added as a new program structure to improve encapsulation and sharing of code. Some existing and new identifiers have been factored out into separate libraries. Libraries can be imported into other libraries or main programs, with controlled exposure and renaming of identifiers. The contents of a library can be made conditional on the features of the implementation on which it is to be used.
- Exceptions can now be signalled explicitly with `raise`, `raise-continuable` or `error`, and can be handled with `with-exception-handler` and the `guard` syntax. Any object can specify an error condition; the implementation-defined conditions signalled by `error` have accessor functions to retrieve the arguments passed to `error`.
- New disjoint types supporting access to multiple fields can be generated with SRFI 9’s `define-record-type`.
- Parameter objects can be created with `make-parameter`, and dynamically rebound with `parameterize`.
- Bytevectors, homogeneous vectors of integers in the range [0, 255], have been added as a new disjoint type. A subset of the procedures available for vectors is provided. Bytevectors can be converted to and from strings in accordance with the UTF-8 character encoding. Bytevectors have a datum representation and evaluate to themselves.
- The procedure `read-line` is provided to make line-oriented textual input simpler.
- `Ports` can now be designated as `textual` or `binary` ports, with new procedures for reading and writing binary data. The new predicate `port-open?` returns whether a port is open or closed.
- `String ports` have been added as a way to read and write characters to and from strings, and `byetvector ports` to read and write bytes to and from bytevectors.
- The procedures `current-input-port` and `current-output-port` are now parameter objects, as is the newly introduced `current-error-port`.
- The `syntax-rules` construct now recognizes _ (underscore) as a wildcard, allows the ellipsis symbol to be specified explicitly instead of the default ..., allows template escapes with an ellipsis-prefixed list, and allows tail patterns to follow an ellipsis pattern.
- The `syntax-error` syntax has been added as a way to signal immediate and more informative errors when a macro is expanded.
- Internal `define-syntax` definitions are now allowed wherever internal `defines` are.
- The `letrec*` binding construct has been added, and internal `define` is specified in terms of it.
- Support for capturing multiple values has been enhanced with `define-values`, `let-values`, and `let*values`. Programs are now explicitly permitted to pass zero or more than one value to continuations which discard them.
- The `case` conditional now supports a `=>` syntax analogous to `cond`.
- To support dispatching on the number of arguments passed to a procedure, `case-lambda` has been added in its own library.
- The convenience conditionals `when` and `unless` have been added.
- Positive infinity, negative infinity, NaN, and negative inexact zero have been added to the numeric tower as inexact values with the written representations `+inf.0`, `-inf.0`, `+nan.0`, and `-0.0` respectively.
- The procedures `map` and `for-each` are now required to terminate on the shortest list when inputs have different length.
- The procedures `member` and `assoc` now take an optional third argument specifying the equality predicate to be used.
- The procedures `exact-integer?` and `exact-integer-sqrt` have been added.
- The procedures `make-list`, `list-copy`, `list-set!`, `string-map`, `string-for-each`, `string->vector`, `vector-copy`, `vector-map`, `vector-for-each`, and `vector->string` have been added to round out the sequence operations.
• Implementations may provide any subset of the full Unicode repertoire that includes ASCII, but implementations must support any such subset in a way consistent with Unicode. Various character and string procedures have been extended accordingly. String comparison remains implementation-dependent, and is no longer required to be consistent with character comparison, which is based on Unicode code points. The new `digit-value` procedure is added to obtain the numerical value of a numeric character.

• The procedures `string-ni=?` and related procedures have been added to compare strings as though they had gone through an implementation-defined normalization, without exposing the normalization.

• The case-folding behavior of `read` can now be explicitly controlled, with no folding as the default.

• There are now two additional comment syntaxes: `#;` to skip the next datum, and `#| ... |#` for nestable block comments.

• Data prefixed with datum labels `#<n>=` can be referenced with `#<n>:` allowing for reading and writing of data with shared structure.

• Strings and symbols now allow mnemonic and numeric escape sequences, and the list of named characters has been extended.

• The procedures `file-exists?` and `delete-file` are available in the `(scheme file)` library.

• An interface to the system environment and command line is available in the `(scheme process-context)` library.

• Procedures for accessing the current time are available in the `(scheme time)` library.

• A complete set of integer division operators is available in the `(scheme division)` library.

• The `load` procedure now accepts a second argument specifying the environment to load into.

• The procedures `transcript-on` and `transcript-off` have been removed.

• The semantics of read-eval-print loops are now partly prescribed, allowing the retroactive redefinition of procedures but not syntax keywords.

• The syntax of the library system was deliberately chosen to be syntactically different from R6RS, using `define-library` instead of `library` in order to allow easy disambiguation between R6RS and R7RS libraries.

• The library system does not support phase distinctions, which are unnecessary in the absence of low-level macros (see below), nor does it support versioning, which is an important feature but deserves more experimentation before being standardized.

• Putting an extra level of indirection around the library body allows room for extensibility. The R6RS syntax provides two positional forms which must be present and must have the correct keywords, `export` and `import`, which does not allow for unambiguous extensions. The Working Group considers extensibility to be important, and so chose a syntax which provides a clear separation between the library declarations and the Scheme code which makes up the body.

• The `include` library declaration makes it easier to include separate files, and the `include-ci` variant allows legacy case-insensitive code to be incorporated.

• The `cond-expand` library declaration based on SRFI 0 allows for a more deterministic alternative to the R6RS .impl.sls file naming convention.

• Since the R7RS library system is straightforward, we expect that R6RS implementations will be able to support the `define-library` syntax in addition to their library syntax.

• The grouping of standardized identifiers into libraries is different from the R6RS approach. In particular, procedures which are optional either expressly or by implication in R6RS have been removed from the base library. Only the base library is an absolute requirement.

• Identifier syntax is not provided. This is a useful feature in some situations, but the existence of such macros means that neither programmers nor other macros can look at an identifier in an evaluated position and know it is a reference — this in a sense makes all macros slightly weaker. Individual implementations are encouraged to continue experimenting with this and other extensions before further standardization is done.

• Internal syntax definitions are allowed, but all references to syntax must follow the definition; the `even/odd` example given in R6RS is not allowed.

• The R6RS exception system was incorporated as is, but the condition types have been left unspecified.

Incompatibilities with the main R6RS document

This section enumerates the incompatibilities between R7RS and the “Revised6 report” [I].

The list is incomplete and subject to change while this report is in draft status.
Specific errors that must be signalled in R⁶RS remain errors in R⁷RS, allowing implementations to provide their own extensions. There is no discussion of safety.

- Full Unicode support is not required. Instead of explicit normalization forms this report provides normalization-insensitive string comparisons that use an implementation-defined normalization form (which may be the identity transformation). Character comparisons are defined by Unicode, but string comparisons are implementation-dependent, and therefore need not be the lexicographic mapping of the corresponding character comparisons (an incompatibility with R⁵RS). Non-Unicode characters are permitted.

- The full numeric tower is optional as in R⁵RS, but optional support for IEEE infinities, NaN, and -0.0 was adopted from R⁶RS. Most clarifications on numeric results were also adopted, but the R⁶RS procedures real-valued?, rational-valued?, and integer-valued? were not. The R⁶RS names inexact->exact for exact and exact->inexact for inexact were retained, with a note indicating that their names are historical. The R⁶RS division operators div, mod, div-and-mod, div0, mod0 and div0-and-mod0 have been replaced with a full set of 18 operators describing 6 different rounding semantics.

- When a result is unspecified, it is still required to be a single value, in the interests of R⁵RS compatibility. However, non-final expressions in a body may return any number of values.

- Because of widespread SRFI 1 support and extensive code that uses it, the semantics of map and for-each have been changed to use the SRFI 1 early termination behavior. Likewise assoc and member take an optional equal? argument as in SRFI 1, instead of the separate assp and memp procedures from R⁶RS.

- The R⁶RS quasiquote clarifications have been adopted, but the Working Group has not seen convincing enough examples to allow multiple-argument unquote and unquote-splicing.

- The R⁶RS method of specifying mantissa widths was not adopted.

Incompatibilities with the R⁶RS Standard Libraries document

This section enumerates the incompatibilities between R⁵RS and the R⁶RS [1] Standard Libraries.

The list is incomplete and subject to change while this report is in draft status.

- The low-level macro system and syntax-case were not adopted. There are two general families of macro systems in widespread use — the syntax-case family and the syntactic-closures family — and they have neither been shown to be equivalent nor capable of implementing each other. Given this situation, low-level macros have been left to the large language.

- The new I/O system from R⁶RS was not adopted. Historically, standardization reflects technologies that have undergone a period of adoption, experimentation, and usage before incorporation into a standard. The Working Group was unhappy with the redundant provision of both the new system and the R⁵RS-compatible “simple I/O” system, which relegated R⁶RS code to being a second-class citizen. However, binary I/O was added using binary ports that are at least potentially disjoint from textual ports and use their own parallel set of procedures.

- String ports are compatible with SRFI 6 rather than R⁶RS; analogous bytevector ports are also provided.

- The Working Group felt that the R⁶RS records system was overly complex, and the two layers poorly integrated. The Working Group spent a lot of time debating this, but in the end decided to simply use a generative version of SRFI 9, which has near-universal support among implementations. The Working Group hopes to provide a more powerful records system in the large language.

- Enumerations are not included in the small language.

- R⁶RS-style bytevectors are included, but provide only the “#u8” procedures in the small language. The lexical syntax uses #vu8 for compatibility with SRFI 4, rather than the R⁶RS #vu8 style. With a library system, it’s easier to change names than reader syntax.

- The utility macros when and unless are provided, but since it would be meaningless to try to use their result, it is left unspecified.

- The Working Group could not agree on a single design for hash tables and left them for the large language.

- Sorting, bitwise arithmetic, and enumerations were not considered to be sufficiently useful to include in the small language. They will probably be included in the large language.

- Pair and string mutation are too well-established to be relegated to separate libraries.

ADDITIONAL MATERIAL

The Internet Scheme Repository at
http://www.cs.indiana.edu/scheme-repository/ contains an extensive Scheme bibliography, as well as papers, programs, implementations, and other material related to Scheme.

The Scheme community website at
http://schemers.org/
contains additional resources for learning and programming, job and event postings, and Scheme user group information.

A bibliography of Scheme-related research at
http://library.readscheme.org/
links to technical papers and theses related to the Scheme language, including both classic papers and recent research.

EXAMPLE

The procedure integrate-system integrates the system
\[ y'_k = f_k(y_1, y_2, \ldots, y_n), \quad k = 1, \ldots, n \]
of differential equations with the method of Runge-Kutta.

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

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

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

The procedure Runge-Kutta-4 takes a function, \(f\), that produces a system derivative from a system state. Runge-Kutta-4 produces a function that takes a system state and produces a new system state.

```scheme
(define runge-kutta-4
  (lambda (f h)
    (let ((*h (scale-vector h))
           (*2 (scale-vector 2))
           (*1/2 (scale-vector (/ 1 2)))
           (*1/6 (scale-vector (/ 1 6)))
           (lambda (y)
                ;; y is a system state
                (let* ((k0 (*h (f y)))
                       (k1 (*h (f (add-vectors y (*1/2 k0)))))
                       (k2 (*h (f (add-vectors y (*1/2 k1)))))
                       (k3 (*h (f (add-vectors y k2)))))
                  (add-vectors y
                               (*1/6 (add-vectors k0
define elementwise
  (lambda (f)
    (lambda vectors
      (generate-vector
       (vector-length (car vectors))
       (lambda (i)
         (apply f
            (map (lambda (v) (vector-ref v i))
                 vectors)))))))
```

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

(define add-vectors (elementwise +))

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

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

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

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

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

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

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

\[
L \frac{di_L}{dt} = v_C
\]

which models a damped oscillator.

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

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

REFERENCES


ALPHABETIC INDEX OF DEFINITIONS OF CONCEPTS,
KEYWORDS, AND PROCEDURES

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

! 7
' 12; 38
* 33
+ 33; 61
, 19; 38
,@ 19
- 34
-> 7
. 7
... 21
/ 34
; 8
< 33; 61
<= 33
= 33
=> 14
> 33
>= 33
? 7
. 21
' 20

abs 34; 36
acos 35
and 14; 62
angle 36
append 39
apply 46; 11, 62
asin 35
assoc 40
assq 40
assv 40
atan 35

#b 32; 57
backquote 19
begin 16; 23, 24, 27, 64
binary-port? 51
binding 9
binding construct 9
boolean? 37; 9
bound 9
byte 45
bytevector-copy 46
bytevector-copy! 46
bytevector-copy-partial 46
bytevector-copy-partial! 46
bytevector-length 45; 31
bytevector-u8-ref 45
bytevector-u8-set! 45
bytevector? 45; 9
bytevectors 45
caaaar 38
caaadr 38
caaar 38
cadar 38
caddr 38
cadr 38
caddar 38
cadddr 38
caddr 38
cadr 38
call 12
call by need 17
call-with-current-continuation 47; 11, 49, 62
call-with-input-file 51
call-with-output-file 51
call-with-port 51
call-with-values 48; 11, 62
call/cc 47; 48
car 38; 61
case 14; 62
case-lambda 20; 24, 66
catch 48
cdaaar 38
cdaadr 38
cdaar 38
cdadr 38
cdaddar 38
cdaddr 38
cddar 38
cdddr 38
cdddr 38
cddr 38
cdr 38
cddar 38
cddadr 38
cddar 38
cddadr 38
cdaddr 38
cddadr 38
cdddr 38
cddrr 38
ceiling 35
ceiling-quotient 34
ceiling-remainder 34
celling/ 34
centered-quotient 34
centered-remainder 34
centered/ 34
port? 51; 9
positive? 33
predicate 28
prefix 26
procedure 28
procedure call 12
procedure? 46; 9
program parts 23
promise 17
proper tail recursion 10
quasiquote 19; 38
quote 12; 38
quotient 34
raise 49
raise-continuable 49
rational? 32; 30
rationalize 35
read 52; 38, 57
read-bytevector 53
read-bytevector! 53
read-char 52
read-line 53
read-u8 53
real-part 36
real? 32; 30
record type definitions 25
record types 25
records 25
referentially transparent 20
region 9; 13, 15, 16, 17
remainder 34
rename 26
repl 23
reverse 39
round 35
round-quotient 34
round-remainder 34
round/ 34
scheme-report-environment 50
set! 13; 24, 60
set-car! 38
set-cdr! 38
setcar 61
simplest rational 35
sin 35
sqrt 35
string 43
string->list 44
string->number 37
string->symbol 40
string->utf8 46
string->vector 45
string-append 44
string-ci<=? 43
string-ci<? 43
string-ci=? 43
string-ci>=? 43
string-ci>? 43
string-copy 44
string-downcase 44
string-fill! 44
string-foldcase 44
string-for-each 47
string-length 43; 31
string-map 46
string-ni<=? 43
string-ni<? 43
string-ni=? 43
string-ni>=? 43
string-ni?> 43
string-ref 43
string-set! 43; 40
string-upcase 44
string<=? 43
string<? 43
string=? 43
string>=? 43
string?> 43
string? 42; 9
substring 44
symbol->string 40; 10
symbol? 40; 9
syntactic keyword 9; 7, 20
syntax definition 25
syntax-error 23
syntax-rules 25
#t 37
tail call 11
tan 35
textual-port? 51
token 56
top level environment 28; 9
type 9
u8-ready? 53
unbound 9; 12, 24
unless 15
unquote 38
unquote-splicing 38
unspecified 6
utf8->string 46
valid indexes 42; 44, 45
values 48; 12
variable 9; 7, 12
vector 44
vector->list 45
vector->string 45
vector-copy 45
vector-fill! 45
vector-for-each 47
vector-length 45; 31
vector-map 47
vector-ref 45
vector-set! 45
vector? 44; 9

when 14
whitespace 8
with-exception-handler 49
with-input-from-file 51
with-output-to-file 51
write 54; 20
write-bytevector 54
write-char 54
write-partial-bytevector 54
write-simple 54
write-u8 54

#x 32; 57

zero? 33