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 DYBVIK, MATTHEW FLATT, AND ANTON VAN STRAATEN
(Editors, Revised\textsuperscript{6} Report on the Algorithmic Language Scheme)

Dedicated to the memory of John McCarthy

18 December 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.

*** DRAFT***

December 18, 2011
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 procedures 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^6RS [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^8RS, 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^8RS, 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^8RS compatibility, and a “large” language focused on the practical needs of mainstream software development which would evolve to become a replacement for R^8RS. 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"ursel, 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.

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

Scheme is a dynamically typed language. Types are associated with values (also called objects) rather than with variables. 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.

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.

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.

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. 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 that Scheme programs and data 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. Base and optional features

Every identifier defined in this report appears in one of several libraries. Identifiers defined in the base library are not marked 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 provide or omit the other 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 mode 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 if 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. 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 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 is returned.

Finally, the words and phrases “must,” “must not,” “shall,” “shall not,” “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 4 and 6 are organized into entries. Each entry describes one language feature or a group of related features, where a feature is either a syntactic construct or a built-in procedure. An entry begins with one or more header lines of the form

```
template category
```

for identifiers in the base library, or

```
template library category
```

where library is the short name of a library as defined in 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, \langle expression \rangle, \langle variable \rangle. Syntactic variables are intended to denote segments of program text; for example, \langle expression \rangle stands for any string of characters which is a syntactically valid expression. The notation

\[
\langle \text{thing}_1 \rangle \ldots
\]

indicates zero or more occurrences of a \langle thing \rangle, and

\[
\langle \text{thing}_1 \rangle \langle \text{thing}_2 \rangle \ldots
\]

indicates one or more occurrences of a \langle thing \rangle.

If category is “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

```
(vector-ref vector k)
```

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

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

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 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 is a vector. The following
naming conventions also imply type restrictions:

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

1.3.4. Evaluation examples

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

\[
(* \ 5 \ 8) \quad \Rightarrow \quad 40
\]

means that the expression \( (* \ 5 \ 8) \) evaluates to the object 40. Or, more precisely: the expression given by the sequence of characters “\( (* \ 5 \ 8) \)” evaluates, in the initial environment, to an object that 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, ? is the final character of the names of procedures that always return a boolean value. Such procedures are called predicates.

Similarly, ! 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. 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, \( \text{list} \rightarrow \text{vector} \) 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.

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:

\[ \{ \text{! \ % \ & \ * \ + \ - \ . \ / \ : \ < \ = \ > \ ? \ @ \ ^ \ _ \ ~} \] 

In addition, any character supported by an implementation can be used within an identifier when specified using an \{inline hex escape\}. For example, the identifier \( \text{H\x65;Hello} \) is the same as the identifier \( \text{Hello} \), and in an implementation that supports the appropriate Unicode character the identifier \( \text{\x3BB;\lambda} \) is the same as the identifier \( \lambda \).

As a convenience, identifiers may also be written as a sequence of 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. 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 \( \text{|foo \ bar|} \) is the same as the identifier \( \text{foo\x20;bar} \). Note that \( | | \) is a valid identifier that is not equal to any other identifier.

Here are some examples of identifiers:

- \( \text{lambda} \)
- \( \text{list} \rightarrow \text{vector} \)
- \( + \)
- \( <=? \)
- \( <\rightarrow \text{string} \)
- \( |\text{two words}| \)
- \( \text{the-word-recursion-has-many-meanings} \)

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 [2], 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.

```scheme
#!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. Whitespace and comments

*Whitespace* characters include the space and newline characters. (Implementations 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 can occur between any two tokens, but not within a token. Whitespace occurring inside a string 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>` 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.

```scheme
| The FACT procedure computes the factorial of a non-negative integer.
|#
| (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. 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). Note that a sequence of two or more periods is an identifier.

### ( )

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

### ’

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

###`

The backquote character is used to indicate 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), 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 `)`.

[36x735]8 Revised
[36x735]7
[85x739]Scheme

[36x698]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.

```scheme
#!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. Whitespace and comments

*Whitespace* characters include the space and newline characters. (Implementations 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 can occur between any two tokens, but not within a token. Whitespace occurring inside a string 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>` 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.

```scheme
| The FACT procedure computes the factorial
| of a non-negative integer.
|#
| (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. 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). Note that a sequence of two or more periods is an identifier.

### ( )

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

### ’

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

###`

The backquote character is used to indicate 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), 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 `)`.
3. Basic concepts

3.1. Variables, syntactic keywords, and regions

An identifier names either a type of syntax 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.

Scheme is a 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?
- port?
- null?
- pair?
- number?
- string?
- bytevector?
- procedure?

These predicates define the types boolean, pair, symbol, number, char (or character), string, vector, bytevector, port, procedure 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, 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 “#e28.0000” 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 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 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. A new value 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.

Every object that denotes locations is either mutable or immutable. Specifically: literal constants, the strings returned by 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 might still return. Note that this includes calls that might 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 [11].

Rationale:

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

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

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

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

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

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

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

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

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

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

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

  (when ⟨test⟩) (tail sequence)
  (unless ⟨test⟩) (tail sequence)

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

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

  (begin ⟨tail sequence⟩)

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

where

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

  ⟨tail body⟩ → (⟨definition⟩* ⟨tail sequence⟩)
  ⟨tail sequence⟩ → (expression)* (tail expression)

- If a cond or case expression is in a tail context, and has a clause of the form ⟨expression1⟩ =⇒ ⟨expression2⟩) then the (implied) call to the procedure that results from the evaluation of ⟨expression2⟩ is in a tail context. ⟨expression2⟩ itself is not in a tail context.

- Note that ⟨cond clause⟩s appear in guard expressions as well as cond expressions.

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

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

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

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

Expression types are categorized as primitive or derived. Primitive expression types include variables and procedure calls. Derived expression types are not semantically primitive, but can instead be defined as macros. Suitable definitions of some of the derived expressions are given in section 7.3.

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

4.1. Primitive expression types

4.1.1. Variable references

\[(\text{variable})\]

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.

\[
\text{(define x 28)}
\]

\[
\Rightarrow \text{28}
\]

4.1.2. Literal expressions

\[
\text{(quote (datum))}
\]

\[
\text{'(datum)}
\]

\[
\text{(constant)}
\]

\[
\text{(quote (datum)) evaluates to (datum). (Datum) may be any external representation of a Scheme object (see section 3.3). This notation is used to include literal constants in Scheme code.}
\]

\[
\text{(quote a)}
\Rightarrow \text{a}
\]

\[
\text{(quote #(a b c))}
\Rightarrow \text{(#(a b c)}
\]

\[
\text{(quote (+ 1 2))}
\Rightarrow \text{(+ 1 2)}
\]

\[
\text{(quote (datum)) may be abbreviated as '(datum). The two notations are equivalent in all respects.}
\]

\[
\text{'a}
\Rightarrow \text{a}
\]

\[
\text{'(a b c)}
\Rightarrow \text{(#(a b c))}
\]

\[
\text{'}()
\Rightarrow \text{()}
\]

\[
\text{'+(1 2)}
\Rightarrow \text{(+ 1 2)}
\]

\[
\text{'(quote a)}
\Rightarrow \text{(quote a)}
\]

\[
\text{''a}
\Rightarrow \text{(quote a)}
\]

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

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 set-car! or string-set!.

4.1.3. Procedure calls

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

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 unspecified order) and the resulting procedure is passed the resulting arguments.

\[
\text{(+ 3 4)}
\Rightarrow \text{7}
\]

\[
\text{(if #f + *) 3 4)}
\Rightarrow \text{12}
\]

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

Procedure calls may return any number of values (see values in section 6.10). Most 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.

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 list, (), is a legitimate expression evaluating to itself. In Scheme, it is an error.

4.1.4. Procedures

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

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
body of the lambda expression will be evaluated in the ex-
tended environment. The results of the last expression in
the body will be returned as the results of the procedure
call.

\[
\text{lambda (x) (} + x x) \quad \longrightarrow \quad \text{a procedure}
\]
\[
((\text{lambda (x) (} + x x)) 4) \quad \longrightarrow \quad 8
\]

\[
\text{(define reverse-subtract}
\]
\[
\text{ (lambda (x y) (- y x))}
\]
\[
(\text{reverse-subtract 7 10}) \quad \longrightarrow \quad 3
\]

\[
\text{(define add4}
\]
\[
\text{ (let ((x 4))}
\]
\[
\text{ (lambda (y) (+ x y)))}
\]
\[
\text{(add4 6)} \quad \longrightarrow \quad 10
\]

(Formals) should have one of the following forms:

- (variable_1 ...): The procedure takes a fixed num-
  ber of arguments; when the procedure is called, the
  arguments will be stored in newly allocated locations
  that are bound to the corresponding variables.

- (variable): The procedure takes any number of argu-
  ments; when the procedure is called, the sequence of
  actual arguments is converted into a newly allocated
  list, and the list is stored in a newly allocated location
  that is bound to (variable).

- (variable_1 ... variable_n . variable_{n+1}): If a
  space-delimited period precedes the last variable, then
  the procedure takes n or more arguments, where n is
  the number of formal arguments before the period (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 allo-
  cated 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).

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

4.1.5. Conditionals

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

Syntax: (Test), (consequent), and (alternate) should be
expressions.

Semantics: An if expression is evaluated as follows: first,
(test) is evaluated. If it yields a true value (see section[6.3],
then (consequent) is evaluated and its values are returned.
Otherwise (alternate) is evaluated and its values are re-
turned. If (test) yields a false value and no (alternate) is
specified, then the result of the expression is unspecified.

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

4.1.6. Assignments

\[
\text{(set! (variable) (expression))}
\]

<Expression) is evaluated, and the resulting value is stored
in the location to which (variable) is bound. 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.

\[
\text{(define x 2)}
\]
\[
\text{(+ x 1)} \quad \Rightarrow \quad 3
\]
\[
\text{(set! x 4)} \quad \Rightarrow \quad \text{unspecified}
\]
\[
\text{(+ x 1)} \quad \Rightarrow \quad 5
\]

4.2. Derived expression types

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

4.2.1. Conditionals

\[
\text{(cond (clause_1) (clause_2) ...)}
\]
\[
\text{else}
\]
\[
\text{=>}
\]

Syntax: (Clauses) take one of two forms, either:

\[
((\text{test} (\text{expression}_1) ...)
\]

where (test) is any expression, or

\[
((\text{test} => (\text{expression}))
\]

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

\[
\text{(else (expression_1) (expression_2) ...)}
\]
Semantics: A cond expression is evaluated by evaluating the (test) expressions of successive (clause)s in order until one of them evaluates to a true value (see section 6.3). When a (test) evaluates to a true value, then the remaining (expression)s in its (clause) are evaluated in order, and the results of the last (expression) in the (clause) are returned as the results 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. It is an error if its value is not a procedure that accepts one argument. This procedure is then called on the value of the (test) and the values returned by this procedure are returned by the cond expression.

If all (test)s evaluate to #f, 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 values of the last one are returned.

\[
\text{(cond } ((> 3 2) 'greater) \\
((< 3 2) 'less)) \implies \text{greater}
\]

\[
\text{(cond } ((> 3 3) 'greater) \\
((< 3 3) 'less) \\
(else 'equal)) \implies \text{equal}
\]

\[
\text{(cond } ((\text{assv} 'b '((a 1) (b 2))) => \text{cadr}) \\
(else #f)) \implies 2
\]

The following example outputs 12:

\[
\text{(case } (\text{key}) (\text{clause}_1) (\text{clause}_2) \ldots) \ldots \text{ syntax}
\]

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

\[
(((\text{datum}_1) \ldots) (\text{expression}_1) (\text{expression}_2) \ldots),
\]

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

\[
(((\text{datum}_1) \ldots) => (\text{expression}))
\]

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

\[
\text{(else } (\text{expression}_1) (\text{expression}_2) \ldots)
\]

or

\[
\text{(else => (expression))}.
\]

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 in order and the results of the last expression in the (clause) are returned as the results 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 results of the last are the results 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.

\[
\text{(case } (* 2 3) \\
((2 3 5 7) 'prime) \\
((1 4 6 8 9) 'composite)) \implies \text{composite}
\]

\[
\text{(case } (\text{car} '(c d)) \\
((a) 'a) \\
((b) 'b)) \implies \text{unspecified}
\]

\[
\text{(case } (\text{car} '(c d)) \\
((a e i o u) 'vowel) \\
((w y) 'semivowel) \\
(else => (\text{lambda} (x) x))) \implies c
\]

\[
\text{(and } (\text{test}_1) \ldots \text{ syntax}
\]

The (test) expressions are evaluated from left to right, and if any expression evaluates to #f (see section 6.3), #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.

\[
\text{(and } (= 2 2) (> 2 1)) \implies \text{#t}
\]

\[
\text{(and } (= 2 2) (< 2 1)) \implies \text{#f}
\]

\[
\text{(and } 1 2 'c '('#f g)) \implies ('f g)
\]

\[
\text{(and)} \implies \text{#t}
\]

\[
\text{(or } (\text{test}_1) \ldots \text{ syntax}
\]

The (test) expressions are evaluated from left to right, and the value of the first expression that evaluates to a true value (see section 6.3) is returned. Any remaining expressions are not evaluated. If all expressions evaluate to #f or if there are no expressions, #f is returned.

\[
\text{(or } (= 2 2) (> 2 1)) \implies \text{#t}
\]

\[
\text{(or } (= 2 2) (< 2 1)) \implies \text{#t}
\]

\[
\text{(or #f #f #f)} \implies \text{#f}
\]

\[
\text{(or } (\text{memq} 'b '((a b c)) \\
(/ 3 0))) \implies (b c)
\]

\[
\text{(when } (\text{test}) (\text{expression}_1) (\text{expression}_2) \ldots \text{ syntax}
\]

The (test) expression is evaluated, and if it evaluates to a true value, the expressions are evaluated in order. The result of the when expression is unspecified.

The following example outputs 12:
The last expression of evaluated in the extended environment, and the values of environment (in some unspecified order), the bindings are evaluated in order. The result of the unless expression is unspecified.

The following example outputs nothing:

```
(unless (= 1 1.0)
  (display "1")
  (display "2")
⇒ unspecified
```

## 4.2.2. Binding constructs

The binding constructs `let`, `let*`, `letrec`, `let-values`, and `let*-values` give Scheme a block structure, like Algol 60. The syntax of the 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 `letrec` and `let*-values` 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.

### Syntax: (let (bindings) (body))

Where each `(init)` is an expression, 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 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 values of the last expression of `(body)` are returned. Each binding of a `(variable)` has `(body)` as its region.

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

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

See also “named let,” section 4.2.4

### Syntax: (let* (bindings) (body))

```
(let* (bindings) (body))
```

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

```
(let* ((x 7)
    (z (+ x y)))
  (* z x)))
⇒ 70
```

### Syntax: (letrec (bindings) (body))

```
(letrec (bindings) (body))
```

### Syntax: (letrec* (bindings) (body))

```
(letrec* (bindings) (body))
```

```
(letrec* ((even? (lambda (n)
  (if (zero? n)
    #t
    (odd? (- n 1)))))
  (odd? (lambda (n)
    (if (zero? n)
      #f
      (even? (- n 1)))))
  (even? 88))
⇒ #t
```

One restriction on `letrec` is very important: if it is not possible to evaluate each `(init)` without assigning or referring to the value of any `(variable)`, it is an error. The restriction is necessary because `letrec` is defined in terms
of a procedure call where a \texttt{lambda} expression binds the \langle variable \rangle s to the values of the \langle init \rangle s. In the most common uses of \texttt{letrec}, all the \langle init \rangle s are \texttt{lambda} expressions and the restriction is satisfied automatically. Another restriction is that the continuation of each \langle init \rangle should not be invoked more than once.

\texttt{(letrec* (bindings) (body))} \hfill \text{syntax}

\textit{Syntax:} \langle Bindings \rangle should have the form
\[
(((\text{variable}_1) \ (\text{init}_1)) \ldots),
\]
and \langle body \rangle should be a sequence of zero or more definitions followed by one or more expressions as described in section \[1.4.1 \text{.} \] It is an error for a \langle variable \rangle to appear more than once in the list of variables being bound.

\textit{Semantics:} The \langle variable \rangle s are bound to fresh locations, each \langle variable \rangle is assigned in left-to-right order to the result of evaluating the corresponding \langle init \rangle, the \langle body \rangle is evaluated in the resulting environment, and the values of the last expression in \langle body \rangle are returned. Despite the left-to-right evaluation and assignment order, each binding of a \langle variable \rangle has the entire \texttt{letrec*} expression as its region, making it possible to define mutually recursive procedures.

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

\[
\texttt{(letrec* (p (lambda (x)) (+ 1 (q (- x 1)))))}
\]
\[
\texttt{(lambda (y) (if (zero? y) 0 (+ 1 (p (- y 1)))) (x (p 5)) (y x))}
\]
\[
\Rightarrow 5
\]

\texttt{(let-values (mvbindings) (body))} \hfill \text{syntax}

\textit{Syntax:} \langle Mvbindings \rangle should have the form
\[
(((\text{formals}_1) \ (\text{init}_1)) \ldots),
\]
where each \langle init \rangle should be an expression, and \langle body \rangle should be zero or more definitions followed by a sequence of one or more expressions as described in section \[1.4.1 \text{.} \] It is an error for a variable to appear more than once in the set of \langle formals \rangle.

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

It is an error if the \langle formals \rangle do not match the number of values returned by the corresponding \langle init \rangle.

\[
\texttt{(let-values (((\text{root} \ \text{rem}) (\text{exact-integer-sqrt} \ 32))) (* \text{root} \ \text{rem}) \Rightarrow 35}
\]

\texttt{(let*-values (mvbindings) (body))} \hfill \text{syntax}

\textit{Syntax:} \langle Mvbindings \rangle should have the form
\[
(((\text{formals}_1) \ (\text{init}_1)) \ldots),
\]
and \langle body \rangle should be a sequence of zero or more definitions followed by one or more expressions as described in section \[1.4.1 \text{.} \] In each \langle formals \rangle, it is an error if any variable appears more than once.

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

\[
\texttt{(let ((a 'a) (b 'b) (x 'x) (y 'y)) (let*-values (((a b) (values x y))) ((x y) (values a b))) (list a b x y))) \Rightarrow (x y x y)}
\]

### 4.2.3. Sequencing

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

\texttt{(begin \langle expression or definition \rangle \ldots)} \hfill \text{syntax}

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

\textit{Rationale:} This form is commonly used in the output of macros (see section \[4.3 \text{.} \]) which need to generate multiple definitions and splice them into the context in which they are expanded.

\texttt{(begin \langle expression_1 \rangle \langle expression_2 \rangle \ldots)} \hfill \text{syntax}

This form of \texttt{begin} can be used as an ordinary expression. The \langle expression \rangle s are evaluated sequentially from left to right, and the values of the last \langle expression \rangle are returned. This expression type is used to sequence side effects such as assignments or input and output.
(define x 0)
(and (= x 0)
  (begin (set! x 5)
    (+ x 1) \rightarrow 6)
(begin (display "4 plus 1 equals ")
  (display (+ 4 1))) \rightarrow unspecified
  and prints 4 plus 1 equals 5

4.2.4. Iteration

(do ((variable) (init) (step)) ...)
  (⟨test⟩ (expression) ...)
  (⟨command⟩ ...)

A 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 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), then the (command) expressions are evaluated in order for effect, the (step) expressions are evaluated in some unspecified order, the (variable) s are bound to fresh locations, the results of the (step) s are stored in the bindings of the (variable) s, and the next iteration begins.

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

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

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

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

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

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

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

(let loop ((numbers '3 -2 1 6 -5))
  (nonneg '0)
  (neg '0))
  (cond ((null? numbers) (list nonneg neg))
    ((>= (car numbers) 0)
     (loop (cdr numbers)
       (cons (car numbers) nonneg neg))
       (cons (car numbers) neg))))

\rightarrow ((6 1 3) (-5 -2))

4.2.5. Delayed evaluation

(delay (expression)) lazy library syntax

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

(lazy (expression)) lazy library syntax

The lazy construct is similar to 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.

(force promise) lazy library procedure

The force procedure forces the value of a promise created by delay or 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.

(force (delay (+ 1 2))) \rightarrow 3
(let ((p (delay (+ 1 2))))
  (list (force p) (force p))) \rightarrow (3 3)
The following example is a mechanical transformation of a lazy stream-filtering algorithm into Scheme. Each call to a constructor is wrapped in \texttt{delay}, and each argument passed to a deconstructor is wrapped in \texttt{force}. The use of \texttt{(lazy ...)} instead of \texttt{(delay (force ...))} around the body of the procedure ensures that an ever-growing sequence of pending promises does not exhaust the heap.

\begin{verbatim}
(define (stream-filter p? s)
  (lazy
    (if (null? (force s))
      (delay '())
      (let ((h (car (force s)))
            (t (cdr (force s))))
        (if (p? h)
          (delay (cons h (stream-filter p? t)))
          (stream-filter p? t)))))

(head (tail (tail (stream-filter odd? integers))))
\end{verbatim}

\begin{verbatim}
⇒ 2
\end{verbatim}

The following examples are not intended to illustrate good programming style, as \texttt{delay}, \texttt{lazy}, and \texttt{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{verbatim}
(define count 0)
(define p
  (delay (begin (set! count (+ count 1))
              count (force p))))

(define x 5)
(p => a promise)
(force p) => 6
P => a promise, still
(begin (set! x 10)
       (force p)) => 6
\end{verbatim}

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

- 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 \texttt{#t} or to \texttt{#f}, depending on the implementation:

  \begin{verbatim}
  (eqv? (delay 1) 1) => unspecified
  (pair? (delay (cons 1 2))) => unspecified
  \end{verbatim}

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

  \begin{verbatim}
  (+ (delay (* 3 7)) 13) => 34
  \end{verbatim}

\texttt{(eager obj)} lazy library procedure

The \texttt{eager} procedure returns a promise which when forced will return \texttt{obj}. It is similar to \texttt{delay}, but does not delay its argument: it is a procedure rather than syntax.

4.2.6. Dynamic bindings

\begin{verbatim}
(make-parameter init) procedure
(make-parameter init converter) procedure
\end{verbatim}

Returns a newly allocated \texttt{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 \texttt{(converter init)}, or of \texttt{init} if the conversion procedure \texttt{converter} is not specified. The associated value can be temporarily changed using \texttt{parameterize}, which is described below.

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

\begin{verbatim}
(parameterize ((\langle param \rangle \langle value \rangle) ...)
  \langle body \rangle)
\end{verbatim}

A \texttt{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 \texttt{(param)} expression is not a parameter object. The \texttt{(param)} and \texttt{(value)} expressions are evaluated in an unspecified order. The \texttt{(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 \texttt{(body)} are returned as the results of the entire \texttt{parameterize} expression.

\textbf{Note:} If the conversion procedure is not idempotent, the results of \texttt{(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 \texttt{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.

\[
\begin{align*}
\text{(define } \text{radix} & \text{ (make-parameter 10} \\
& \text{(lambda (x)} \\
& \text{ (if (and (integer? x) (<= 2 x 16) x \\
& \text{ (error "invalid radix"))))})
\end{align*}
\]

\[
\begin{align*}
\text{(define } \text{f n (number->string n (radix)}) & \\
\text{(f 12) } & \Rightarrow \text{"12"} \\
\text{(parameterize ((radix 2)) (f 12)) } & \Rightarrow \text{"1100"} \\
\text{(radix 16) } & \Rightarrow \text{unspeical} \\
\text{(parameterize ((radix 0)) (f 12)) } & \Rightarrow \text{error}
\end{align*}
\]

### 4.2.7. Exception Handling

\[
\begin{align*}
\text{(guard ((variable) (cond clause}_1) \text{ (cond clause}_2) \ldots) (body))
\end{align*}
\]

\textbf{Syntax:} Each \texttt{(cond clause)} is as in the specification of \texttt{cond}.

\textbf{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 \texttt{cond} expression. That implicit \texttt{cond} expression is evaluated with the continuation and dynamic environment of the \texttt{guard} expression. If every \texttt{(cond clause)’s \{test\} evaluates to \texttt{#f} and there is no \texttt{else} clause, then \texttt{raise-continuable} is re-invoked on the raised object within the dynamic environment of the original call to \texttt{raise} except that the current exception handler that is of the \texttt{guard} expression.

See section 6.11 for a more complete discussion of exceptions.

### 4.2.8. Quasiquotation

\[
\begin{align*}
\text{(quasquote } \text{qq template}) & \text{ syntax} \\
\text{'(qq template)} & \text{ syntax} \\
\text{unquote} & \text{ auxiliary syntax} \\
\text{unquote-splicing} & \text{ auxiliary syntax}
\end{align*}
\]

“Backquote” or “quasiquote” expressions are useful for constructing a list or vector structure when some but not all of the desired structure is known in advance. If no commas appear within the \{qq template\}, the result of evaluating \texttt{'(qq template)} is equivalent to the result of evaluating \texttt{(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 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\}.

\[
\begin{align*}
\text{`(list ,(+ 1 2) 4) } & \Rightarrow \text{(list 3 4)} \\
\text{(let ((name 'a)) `(list ,name ,name)) } & \Rightarrow \text{(list a (quote a))} \\
\text{`(a ,(+ 1 2) ,@ (map abs '(4 -5 6)) b) } & \Rightarrow \text{(a 3 4 5 6 b)} \\
\text{`(( foo ,(− 10 3)) ,@ (cdr '(c)) , (car '(cons)) ) } & \Rightarrow \text{((foo 7) . cons)} \\
\text{`#(10 5,(sqrt 4) ,@ (map sqrt '(16 9)) 8) } & \Rightarrow \text{#(10 5 2 4 3 8)}
\end{align*}
\]

Quasiquote 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.

\[
\begin{align*}
\text{`(a `b ,(+ 1 2) ,(foo ,(+ 1 3) d) e) f) } & \Rightarrow \text{(a `(b ,(+ 1 2) ,(foo 4 d) e) f)} \\
\text{(let ((name1 'x) (name2 'y)) `(a `(b ,name1 ,name2 d) e)) } & \Rightarrow \text{(a `(b ,x ,y d) e)}
\end{align*}
\]

A quasiquote 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,

\[
\text{(let ((a 3)) `(((1 2) ,a 4 ,five 6))}
\]

may be equivalent to either of the following expressions:

\[
\text{`((1 2) 3 4 five 6)}
\]
It is an error for the arguments not to agree with the
\textit{formals} of the \textit{lambda} expression. The results
are stored in those locations, the \textit{body} of which is called, the first
\textit{formals} expression. When the \textit{procedure}
resulting from a \textit{lambda} expression is lexically scoped in the same manner as a \textit{procedure} that accepts a variable number of arguments and
\textit{procedure} semantics: A \textit{case-lambda} expression evaluates to a pro-
duction types, called \textit{macros}. Program-defined \textit{expression}
types have the syntax
\[
(\text{define range} \ (\text{range} \ 0 \ e))
\]
\textit{define-syntax} is identical to \textit{letrec-syntax} bindings, and local variable bindings may shadow keyword
bindings.

\subsection{Case-lambda}
\begin{verbatim}
(case-lambda (clause1) (clause2) ...)
\end{verbatim}

\textbf{Syntax:} Each (\textit{clause}) should be of the form ((\textit{formals}) (\textit{body})), where (\textit{formals}) and (\textit{body}) have the same syntax
as in a \textit{lambda} expression.

\textbf{Semantics:} A \textit{case-lambda} expression evaluates to a pro-
cEDURE that accepts a variable number of arguments and
is lexically scoped in the same manner as a procedure result-
expressions. When the procedure is called, the first \textit{clause} for which the arguments agree with (\textit{formals}) is selected, where agreement is specified as for
the \textit{formals} of a \textit{lambda} expression. The \textit{formals}
are bound to fresh locations, the values of the arguments are stored in those locations, the \textit{body} is evaluated in the extended environment, and the results of (\textit{body}) are returned as the results of the procedure call.

However, it is not equivalent to this expression:
\[
(\text{let ((a 3)) (list (list 1 2) a 'five 6))}
\]
The two notations `\textquote (qq template)` and \textquote (\textit{qq template}) are identical in all respects. \textquote (\textit{expression}) is identical to \textquote (expression), and \textquote (\textit{expression}) is identical to \textquote (\textit{expression}). The external syntax generated by \textit{write} for two-element lists
whose car is one of these symbols may vary between im-
plementations.

\[
(\text{quasiquote (list (unquote (+ 1 2)) 4))}
\Rightarrow (\text{list 3 4})
\]
\[
'(\text{quasiquote (list (unquote (+ 1 2)) 4))}
\Rightarrow ' (\text{list ,(+ 1 2) 4})
i.e., (\text{quasiquote (list (unquote (+ 1 2)) 4))
\]

It is an error if any of the identifiers \textquote, \textquote, or \textquote appear in positions within a \textquote otherwise than as described above.

\section{4.3. Macros}

Scheme programs can define and use new derived expression types, called \textit{macros}. Program-defined \textit{expression}
types have the syntax
\[
(\text{define range} \ (\text{range} \ 0 \ e))
\]

\textit{define-syntax} is identical to \textit{letrec-syntax} bindings, and local variable bindings may shadow keyword
bindings.

\begin{verbatim}
(define range
 (case-lambda
  ((e) (range 0 e))
  ((b e) (do ((r '()) (cons e r))
     (e (- e 1) (- e 1)))
   ((< e b) r))))
\end{verbatim}

\subsection{4.3.1. Binding constructs for syntactic keywords}

The \textit{let-syntax} and \textit{letrec-syntax} binding constructs are analogous to \textit{let} and \textit{letrec}, but they bind syn-
tactic keywords to macro transformers instead of binding
variables to locations that contain values. Syntactic key-
words may also be bound at top level or elsewhere with
\textit{define-syntax}; see section 5.3.

\[
(\text{let-syntax} \ (\text{bindings}) \ (\text{body}))
\]

\textbf{Syntax:} (\textit{Bindings}) should have the form

\begin{verbatim}
(define range
 (case-lambda
  ((e) (range 0 e))
  ((b e) (do ((r '()) (cons e r))
     (e (- e 1) (- e 1)))
   ((< e b) r)))
\end{verbatim}
(let-syntax (((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 (body) as its region.

(let-syntax (((when syntax-rules ()
   ((when test stmt1 stmt2 ...)
    (if test
     (begin stmt1 stmt2 ...)))))

(let ((if #t))
 (when if (set! if 'now))
  if))
  ⇒ now

(let ((x 'outer))
 (let-syntax ((m (syntax-rules () ((m) x))))
    (let ((x 'inner))
     (m)))
    ⇒ outer

(letrec-syntax (bindings) (body))

Syntax: Same as for let-syntax.

Semantics: The (body) is expanded in the syntactic environment obtained by extending the syntactic environment of the letrec-syntax expression with macros whose keywords are the (keyword)s, bound to the specified transformers. Each binding of a (keyword) has the (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.

(letrec-syntax
  ((my-or (syntax-rules ()
      ((my-or #f)
       ((my-or e) e)
       ((my-or el e2 ...) 
        (let ((temp el))
         (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 one of the following forms:

(syntax-rules (literal ...) syntax
  (syntax-rule ...) auxiliary syntax
(syntax-rules (ellipsis) (literal ...) syntax
  (syntax-rule ...) auxiliary syntax

Syntax: 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) is not of the form

((pattern) (template))

The (pattern) in a (syntax rule) is a list (pattern) whose first element is an identifier.

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

((pattern) ...
  (pattern) ... . (pattern))
((pattern) ... (pattern) (ellipsis) (pattern) ...)
((pattern) ... (pattern) (ellipsis) (pattern) ... . (pattern))
#(pattern) ...
#(pattern) ... (pattern) (ellipsis) (pattern) ...

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

((element) ...)
((element) (element) ... . (template))
((ellipsis) (template))
#(element) ...)

where an (element) is a (template) optionally followed by an (ellipsis). An (ellipsis) is the identifier specified in the second form of syntax-rules, or the default identifier ... (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 appearing within a (pattern) may be an underscore (_), a literal identifier listed in the list of (literal)s, or the (ellipsis). All other identifiers appearing within a (pattern) are pattern variables.

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).

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).

Identifiers that appear in (literal ... ) are interpreted as literal identifiers to be matched against corresponding elements of the input. A 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 identifier if and only if it is an identifier and either both its

Identifiers that appear in a list, then that takes precedence and underscores in the pattern are replaced by the elements they match in the input. Pattern variables that occur in subpatterns followed by one or more instances of the identifier (ellipsis) are allowed only in subtemplates that are followed by as many instances of (ellipsis). They are replaced in the output by all of the elements 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 (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 ((ellipsis) (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 ((ellipsis) (ellipsis)) produces a single (ellipsis). This allows syntactic abstractions to expand into code containing ellipses.

{(define-syntax be-like-begin
  (syntax-rules ()
    ((be-like-begin name)
      (define-syntax name
        (syntax-rules ()
          ((name expr (.... . . . . .))
            (begin expr (.... . . . . .)))))))
  (be-like-begin sequence)
  (sequence 1 2 3 4) => 4

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

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

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

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

\[
\begin{align*}
(\text{let ((=> #f)}) \\
(\text{let ((temp #t)}) \\
(\text{if temp ('ok temp))}))
\end{align*}
\]

which would result in an invalid procedure call.

### 4.3.3. Signalling errors in macro transformers

\[
\text{(syntax-error \{message\} \{args\} ...)}
\]

\text{syntax-error} behaves similarly to \text{error} (6.11) except that implementations with an expansion pass separate from evaluation should signal an error as soon as \text{syntax-error} is expanded. This can be used as a \text{syntax-rules} (template) for a (pattern) that is an invalid use of the macro, which can provide more descriptive error messages. \text{(message)} should be a string literal, and \text{(args)} arbitrary expressions providing additional information. Applications cannot count on being able to catch syntax errors with exception handlers or guards.

\[
\begin{align*}
(\text{define-syntex simple-let} \\
(\text{syntax-rules ()} \\
(\_, \text{(head ... ((x . y) val) . tail)} \text{ body1 body2 ...}) \\
(\text{syntax-error} \\
\text{"expected an identifier but got"} \\
(x . y))) \\
(\_, \text{((name val) ... body1 body2 ...}) \\
(\text{(lambda (name ...) body1 body2 ...)} \\
\text{val ...))))))
\end{align*}
\]

### 5. Program structure

#### 5.1. Programs

A Scheme program consists of a sequence of \textit{program parts}: expressions, definitions, syntax definitions, record type definitions, imports, cond-expands, and includes. A collection of program parts may be encapsulated in a library to be reused by multiple programs. Expressions are described in chapter 4, 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, and other paradigms are possible. Implementations which store libraries in files should document the mapping from the name of a library to its location in the file system.

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 \textit{import}, so further bindings can only be introduced with \textit{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 (\texttt{begin \{form\} \ldots}) is equivalent to the sequence of expressions, definitions, and syntax definitions in the \texttt{begin}. Macros can expand into such \texttt{begins}.

Implementations may provide an interactive session called a \texttt{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 not empty, but must start out 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 \texttt{abs} is bound to a procedure of one argument that computes the absolute value of a number, and the variable \texttt{+} is bound to a procedure that computes sums. The full list of \texttt{(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 \texttt{(program)} and at the beginning of a \texttt{(body)}.
In a \(<body>\) \(<\text{begin} \langle\text{definition}_1\rangle \ldots\) is equivalent to the sequence of definitions \(<\text{definition}_1\rangle \ldots\). Macros can expand into such \(<\text{begin}\) s.

A definition takes one of the following forms:

- \(<\text{define} \langle\text{variable}\rangle \langle\text{expression}\rangle)\)

- \(<\text{define} \langle\langle\text{variable}\rangle \langle\text{formals}\rangle\rangle \langle\text{body}\rangle)\)

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

\[\text{(define \langle\text{variable}\rangle)}\]
\[\quad \text{(lambda \langle\langle\text{formals}\rangle\rangle \langle\text{body}\rangle))}.\]

- \(<\text{define} \langle\langle\text{variable}\rangle \langle\text{formal}\rangle\rangle \langle\text{body}\rangle)\)

\(\langle\text{Formal}\rangle\) should be a single variable. This form is equivalent to

\[\text{(define \langle\text{variable}\rangle)}\]
\[\quad \text{(lambda \langle\langle\text{formal}\rangle\rangle \langle\text{body}\rangle))}.\]

### 5.2.1. Top level definitions

At the top level of a program, a definition

\[\text{(define \langle\text{variable}\rangle \langle\text{expression}\rangle)}\]

has essentially the same effect as the assignment expression

\[\text{(set! \langle\text{variable}\rangle \langle\text{expression}\rangle)}\]

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

\[\text{(define \text{add3}}\]
\[\quad \text{(lambda \langle\langle\text{x}\rangle\rangle \langle\langle\text{+} \langle\text{x} \langle\text{3}\rangle\rangle\rangle\rangle)\]  \(\Rightarrow\)  \(\text{6}\)
\[\text{(define \text{first} \text{car})}\]
\[\quad \text{(first \langle\langle\text{1} \text{2} \rangle\rangle)\]  \(\Rightarrow\)  \(\text{1}\)

Implementations are permitted to provide an initial environment in which all possible variables are bound to locations, most of which contain 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 \texttt{lambda}, \texttt{let}, \texttt{let*}, \texttt{letrec}, \texttt{letrec*}, \texttt{let-values}, \texttt{let-values*}, \texttt{let-syntax}, \texttt{letrec-syntax}, \texttt{parameterize}, \texttt{guard}, or \texttt{case-lambda} expression or that of a definition of an appropriate form). Such definitions are known as \textit{internal definitions} as opposed to the top level definitions described above. The variable defined by an internal definition is local to the \(\langle\text{body}\rangle\). That is, \(\langle\text{variable}\rangle\) is bound rather than assigned, and the region of the binding is the entire \(<\text{body}\rangle\). For example,

\[\text{(let \langle\langle\text{(x 5)}\rangle\rangle)}\]
\[\quad \text{(define \text{foo} \langle\langle\text{lambda} \langle\langle\text{y}\rangle\rangle (\langle\langle\text{bar} \langle\langle\text{x}\rangle\rangle \langle\langle\text{y}\rangle\rangle)\rangle\rangle\rangle)}\]
\[\quad \text{(define \text{bar} \langle\langle\text{lambda} \langle\langle\text{a} \text{b}\rangle\rangle (\langle\langle\text{+} (\langle\langle\text{a} \text{b}\rangle\rangle) \langle\langle\text{a}\rangle\rangle)\rangle\rangle\rangle)}\]
\[\quad \text{(\langle\langle\text{foo} \langle\langle\text{+} \langle\langle\text{x}\rangle\rangle \langle\langle\text{3}\rangle\rangle)\rangle\rangle)\]  \(\Rightarrow\)  \(\text{45}\)

An expanded \(<\text{body}\) containing internal definitions can always be converted into a completely equivalent \texttt{letrec*} expression. For example, the \texttt{let} expression in the above example is equivalent to

\[\text{(let \langle\langle\text{(x 5)}\rangle\rangle)}\]
\[\quad \text{(letrec* \langle\langle\text{(define \text{foo} \langle\langle\text{lambda} \langle\langle\text{y}\rangle\rangle \langle\langle\text{bar} \langle\langle\text{x} \text{y}\rangle\rangle)\rangle\rangle\rangle\rangle)}\]
\[\quad \quad \text{(\langle\langle\text{foo} \langle\langle\text{+} \langle\langle\text{x} \text{3}\rangle\rangle\rangle)\rangle\rangle)}\]

Just as for the equivalent \texttt{letrec*} expression, it is an error if it is not possible to evaluate each \langle\text{expression}\rangle of every internal definition in a \langle\text{body}\rangle without assigning or referring to the value of the corresponding \langle\text{variable}\rangle or the \langle\text{variable}\rangle of any of the definitions that follow it in \langle\text{body}\rangle.

It is an error to define the same \textit{identifier} more than once in the same \langle\text{body}\rangle.

Wherever an internal definition may occur, \(<\text{begin} \langle\text{definition}_1\rangle \ldots\) is equivalent to the sequence of definitions that form the body of the \texttt{begin}.

### 5.2.3. Multiple-value definitions

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

\[\text{(define-values \langle\langle\text{formals}\rangle\rangle \langle\text{expression}\rangle)\]

\text{syntax}

It is an error if a variable appears more than once in the set of \langle\text{formals}\rangle.

\textit{Semantics:} \langle\text{Expression}\rangle is evaluated, and the \langle\text{formals}\rangle are bound to the return values in the same way that the \langle\text{formals}\rangle in a \texttt{lambda} expression are matched to the arguments in a procedure call.

\[\text{(let \langle\langle()\rangle\rangle)}\]
\[\quad \text{(define-values \langle\langle\text{x} \text{y}\rangle\rangle \langle\langle\text{values} \langle\langle\text{1} \text{2}\rangle\rangle\rangle)\]
\[\quad \quad \langle\langle\text{+} \langle\langle\text{x} \text{y}\rangle\rangle)\rangle\rangle\]
\[\quad \Rightarrow\)  \(\text{3}\)
5.3. Syntax definitions

Syntax definitions are valid wherever definitions are. They have the following form:

\[
\text{(define-syntax } \langle \text{keyword} \rangle \langle \text{transformer spec} \rangle \text{)}
\]

\langle \text{keyword} \rangle \text{ is an identifier, and the } \langle \text{transformer spec} \rangle \text{ should be an instance of syntax-rules. If the } \text{define-syntax} \text{ occurs at the top level, then the top-level syntactic environment is extended by binding the } \langle \text{keyword} \rangle \text{ to the specified transformer, but existing references to any top-level binding for } \langle \text{keyword} \rangle \text{ remain unchanged. Otherwise, it is an internal syntax definition, and is local to the } \langle \text{body} \rangle \text{ in which it is defined.}

\[
\begin{align*}
\text{(let } ((x 1) (y 2)) \\
\text{(define-syntax swap!)} \\
\text{ (syntax-rules ()} \\
\text{ (let ((a b))} \\
\text{ (set! a b))} \\
\text{ (swap! x y))} \\
\text{(list x y))} \\
\Rightarrow \langle 2 1 \rangle
\end{align*}
\]

Macros can expand into definitions in any context that permits them. However, it is an error for a definition to define an identifier whose binding has to be known in order to determine the meaning of the 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 internal definitions and the expressions of the body it belongs to. For example, the following are errors:

\[
\begin{align*}
\text{(define define 3)} \\
\text{(begin (define begin list))}
\end{align*}
\]

\[
\begin{align*}
\text{(let-syntax (\langle foo \rangle \text{syntax-rules ()} \\
\text{ (\langle foo \rangle proc args ... body ...) \\
\text{ (define proc} \\
\text{ (lambda (args ...)} \\
\text{ body ...)))))} \\
\text{(let ((x 3))} \\
\text{ (foo (plus x y) (+ x y))} \\
\text{ (define foo x)} \\
\text{ (plus foo x))}
\end{align*}
\]

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.

\[
\text{(define-record-type } \langle \text{name} \rangle \text{ syntax} \\
\text{ (constructor } \langle \text{pred} \rangle \langle \text{field} \rangle \ldots \text{)}
\]

\langle \text{name} \rangle \text{ and } \langle \text{pred} \rangle \text{ should be identifiers. The } \langle \text{constructor} \rangle \text{ should be of the form}

\[
\text{((\langle constructor name \rangle \langle field name \rangle \ldots))}
\]

and each \langle field \rangle should be either of the form

\[
\text{((\langle field name \rangle \langle accessor name \rangle))}
\]
or of the form

\[
\text{((\langle field name \rangle \langle accessor name \rangle \langle modifier name \rangle))}
\]

It is an error for the same identifier to occur more than once as a field name.

define-record-type is generative: each use creates a new record type that is distinct from all existing types, including Scheme’s predefined types and other record types — even record types of the same name or structure.

An instance of define-record-type is equivalent to the following definitions:

- \langle name \rangle is bound to a representation of the record type itself. This may be a run-time object or a purely syntactic representation.
- \langle constructor name \rangle is bound to a procedure that takes as many arguments as there are \langle field name \rangle\text{s in the } \langle \text{constructor name} \rangle \ldots \text{ subform and returns a new record of type } \langle \text{name} \rangle. Fields whose names are listed with \langle constructor name \rangle have the corresponding argument as their initial value. The initial values of all other fields are unspecified.
- \langle pred \rangle is bound to a predicate that returns \#t when given a value returned by the procedure bound to \langle constructor name \rangle and \#f for everything else.
- Each \langle accessor name \rangle is bound to a procedure that takes a record of type \langle name \rangle and returns the current value of the corresponding field. It is an error to pass an accessor a value which is not a record of the appropriate type.
- Each \langle modifier name \rangle is bound to a procedure that takes a record of type \langle name \rangle and a value which becomes the new value of the corresponding field; an unspecified value is returned. It is an error to pass a modifier a first argument which is not a record of the appropriate type.

For instance, the following definition
defines `kons` to be a constructor, `kar` and `kdr` to be ac-

seers, `set-kar!` to be a modifier, and `pare?` to be a predicate for instances of `<pare>.

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

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 declaration>`. 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 #t, 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)s 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)
  (rename put! set!))

(import (scheme base))
(begi
  ;; Create an N x M grid.
  (define (make n m)
    (let ((grid (make-vector n)))
      (do ((i 0 (+ i 1)))
        ((= i n) grid)
        (let ((v (make-vector m #f)))
          (vector-set! grid i v)))))

(define (rows grid)
  (vector-length grid))
(define (cols grid)
  (vector-length (vector-ref grid 0)))

;; Return #false if out of range.
(define (ref grid n m)
  (and (< -1 n (rows grid))
       (< -1 m (cols grid)))
  (vector-ref (vector-ref grid n) m)))

(define (put! grid n m v)
  (vector-set! grid n m v))

(define (each grid proc)
  (do ((j 0 (+ j 1)))
      ((= j (rows grid))
       (do ((k 0 (+ k 1)))
          ...)))
;;; Scheme

((= k (cols grid)))
(proc j k (ref grid j k)))))))

(define-library (example life)
(export life)
(import (except (scheme base) set!)
(scheme write)
(example grid))
(begin
 (define (life-count grid i j)
 (define (count i j)
 (if (ref grid i j) 1 0))
 (+ (count (- i 1) (- j 1))
 (count (- i 1) j)
 (count (- i 1) (+ j 1))
 (count i (- j 1))
 (count i (+ j 1))
 (count (+ i 1) (- j 1))
 (count (+ i 1) j))
 (count (+ i 1) (+ j 1))))
(define (life-alive? grid i j)
 (case (life-count grid i j)
 ((3) #true)
 ((2) (ref grid i j))
 (else #false)))
(define (life-print grid)
 (display "\x1B;[1H\x1B;[J") ; clear vt100
 (each grid
 (lambda (i v)
 (display (if v "* " " ”))
 (when (= j (- (cols grid) 1))
 (newline)))))
(define (life grid iterations)
 (do ((i 0 (+ i 1))
      (grid0 grid grid1)
      (grid1 (make (rows grid) (cols grid))
             grid0))
     ((= i iterations))
    (each grid0
     (lambda (j k v)
      (let ((a (life-alive? grid0 j k)))
       (set! grid1 j k a)))))
    (life-print grid1)))))))

;;; Main program.
(import (scheme base)
 (only (example life) life)
 (rename (prefix (example grid) grid-)
 (grid-make make-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)
(grid-set! grid 3 0 #true)
(grid-set! grid 3 1 #true)
(grid-set! grid 3 2 #true)

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

6. Standard procedures

This chapter describes Scheme’s built-in procedures.

The procedures force and eager are intimately associated with the expression types delay and lazy, and are described with them in section 4.2.5. In the same way, the procedure make-parameter is intimately associated with the expression type parameterize, and is described with it in section 4.2.6.

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 or most discriminating, 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 are normally 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

\[ \text{(string=?} \text{symbol->string obj1)} \text{symbol->string obj2)} \rightarrow #t \]

Note: This assumes that neither obj1 nor obj2 is an “uninterned symbol” as alluded to in section 6.3. This report does not specify the behavior of eqv? on implementation-dependent extensions.
The `eqv?` procedure returns `#f` if:

- `obj1` and `obj2` are of different types (section 3.2).
- one of `obj1` and `obj2` is `#t` but the other is `#f`.
- `obj1` and `obj2` are symbols but
  
  \[
  (\text{string=? } (\text{symbol->string } obj1))
  (\text{symbol->string } obj2))
  \Rightarrow #f
  \]
- one of `obj1` and `obj2` is an exact number but the other is an inexact number.
- `obj1` and `obj2` are numbers for which the `=` procedure returns `#f`, and `nan?` returns `#f` for both.
- `obj1` and `obj2` are characters for which the `char=?` procedure returns `#f`.
- one of `obj1` and `obj2` is the empty list but the other is not.
- `obj1` and `obj2` are pairs, vectors, bytectors, records, or strings that denote distinct locations.
- `obj1` and `obj2` are procedures that would behave differently (return different values or have different side effects) for some arguments.

\[
\begin{align*}
\text{(eqv? 'a 'a)} & \Rightarrow #t \\
\text{(eqv? 'a 'b)} & \Rightarrow #f \\
\text{(eqv? 2 2)} & \Rightarrow #t \\
\text{(eqv? () '())} & \Rightarrow #t \\
\text{(eqv? 10000000 100000000)} & \Rightarrow #t \\
\text{(eqv? (cons 1 2) (cons 1 2))} & \Rightarrow #f \\
\text{(eqv? (lambda () 1))} & \Rightarrow #f \\
\text{(eqv? (lambda () 2))} & \Rightarrow #f \\
\text{(eqv? #f 'nil)} & \Rightarrow #f
\end{align*}
\]

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.

\[
\begin{align*}
\text{(eqv? "a" "a")} & \Rightarrow \text{unspecified} \\
\text{(eqv? 'a (\text{symbol->string } 'a))} & \Rightarrow \text{unspecified} \\
\text{(eqv? '(a))} & \Rightarrow \text{unspecified} \\
\text{(eqv? 'b (\text{cdr } '(a b)))} & \Rightarrow \text{unspecified} \\
\text{(let ((x '(a)))} & \Rightarrow \text{#t}
\end{align*}
\]

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.

\[ \text{eqv? obj1 obj2} \] procedure
The \text{eqv?} procedure is similar to \text{eqv?} except that in some cases it is capable of discerning distinctions finer than those detectable by \text{eqv?}.

On symbols, booleans, the empty list, pairs, procedures, and non-empty strings, vectors, bytevectors, and records, \text{eq?} and \text{eqv?} are guaranteed to have the same behavior. On numbers and characters, \text{eq?}’s behavior is implementation-dependent, but it will always return either true or false, and will return true only when \text{eqv?} would also return true. On empty vectors and empty strings, \text{eq?} may also behave differently from \text{eqv?}.

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

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

\[ \text{(equal? obj1 obj2)} \] procedure
The \text{equal?} procedure recursively compares the contents of pairs, vectors, strings, bytevectors, and records, applying \text{eqv?} on other objects such as numbers and symbols. If two objects are \text{eqv?}, they must be \text{equal?} as well. Even if its arguments are circular data structures, \text{equal?} must always terminate.

\[
\begin{align*}
\text{(equal? 'a 'a)} & \Rightarrow \text{#t} \\
\text{(equal? '(a) '(a))} & \Rightarrow \text{#t}
\end{align*}
\]

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

6.2. Numbers
It is important to distinguish between 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 \text{number, complex, real, rational, and integer} to refer to both mathematical numbers and Scheme numbers.

6.2.1. Numerical types
Mathematically, numbers are arranged into a tower of subtypes in which each level is a subset of the level above it:

\[
\text{number, complex, real, rational, integer}
\]

For example, 3 is an integer. Therefore 3 is also a rational, a real, and a complex. The same is true of the Scheme numbers that model 3. For Scheme numbers, these types are defined by the predicates \text{number?, complex?, real?, rational?, and integer?}.

There is no simple relationship between a number’s type and its representation inside a computer. Although most implementations of Scheme will offer at least two different representations of 3, these different representations denote the same integer.

Scheme’s numerical operations treat numbers as abstract data, as independent of their representation as possible. Although an implementation of Scheme may use 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 might not be. For example, indexes into data structures must be known exactly, as must some polynomial coefficients in a symbolic algebra system. On the other hand, the results of measurements are inherently inexact, and irrational numbers may
be approximated by rational and therefore inexact approximations. In order to catch uses of inexact numbers where exact numbers are required, Scheme explicitly distinguishes exact from inexact numbers. This distinction is orthogonal to the dimension of type.

6.2.2. Exactness

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

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

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

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, implementations in which all numbers are real, 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 permitted as indexes of lists, vectors, bytevectors, and strings or that result from computing the length of one of these. The length, vector-length, bytevector-length, and string-length procedures must return an exact integer, and it is an error to use anything but an exact integer as an index. Furthermore, any integer constant within the index range, if expressed by an exact integer syntax, will indeed be read as an exact integer, regardless of any implementation restrictions that apply outside this range. Finally, the procedures listed below will always return exact integer results provided all their arguments are exact integers and the mathematically expected results are representable as exact integers within the implementation:

\[
\begin{align*}
+ & \quad - \\
quotient & \quad remainder \quad modulo \\
max & \quad min \quad abs \\
numerator & \quad denominator \quad gcd \\
lcm & \quad floor \quad ceiling \\
truncate & \quad round \quad rationalize \\
expt & \quad exact-integer-sqrt \\
floor/ & \quad ceiling/ \quad centered/ \\
truncate/ & \quad round/ \quad euclidean/ \\
floor-quotient & \quad floor-remainder \\
ceiling-quotient & \quad ceiling-remainder \\
centered-quotient & \quad centered-remainder \\
truncate-quotient & \quad truncate-remainder \\
round-quotient & \quad round-remainder \\
euclidean-quotient & \quad euclidean-remainder
\end{align*}
\]

Implementations are encouraged, but not required, to support exact integers and exact rationals of practically unlimited size and precision, and to implement the above procedures and the / procedure in such a way that they always return exact results when given exact arguments. If one of these procedures is unable to deliver an exact result when given exact arguments, then it may either report a violation of an implementation restriction or it may silently coerce its result to an inexact number; such a coercion can 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 754 standard be followed by implementations that use floating-point representations, and that implementations using other representations should match or exceed the precision achievable using these floating-point standards [17]. In particular, 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. Although it is desirable for potentially inexact operations such as \texttt{sqrt} to produce exact answers when applied to exact arguments, if an exact number is operated upon so as to produce an inexact result, then the most precise representation available must be used. For example, the value of \texttt{(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.

In addition, 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 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 \texttt{(asin 2.0)}.

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 \texttt{-0.0} which is distinct (in the sense of \texttt{eqv?}) from \texttt{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 can be written in binary, octal, decimal, or hexadecimal by the use of a radix prefix. The radix prefixes are \texttt{#b} (binary), \texttt{#o} (octal), \texttt{#d} (decimal), and \texttt{#x} (hexadecimal). With no radix prefix, a number is assumed to be expressed in decimal.

A numerical constant can be specified to be either exact or inexact by a prefix. The prefixes are \texttt{#e} for exact, and \texttt{#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 is inexact if it contains a decimal point or an exponent. Otherwise, it is exact.

In systems with inexact numbers of varying precisions it can be useful to specify the precision of a constant. For this purpose, implementations may accept numerical constants written with an exponent marker that indicates the desired precision of the inexact representation. The letters \texttt{s}, \texttt{f}, \texttt{d}, and \texttt{l}, meaning \texttt{short}, \texttt{single}, \texttt{double}, and \texttt{long} precision respectively, are acceptable in place of \texttt{e}. The default precision has at least as much precision as \texttt{double}, but implementations may allow this default to be set by the user.

\texttt{3.14159265358979F0}
Round to single — \texttt{3.141593}

\texttt{0.6L0}
Extend to long — \texttt{.600000000000000}

The numbers positive infinity, negative infinity and NaN are written \texttt{+inf.0}, \texttt{-inf.0} and \texttt{+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.

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

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

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

Note: In many implementations 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.

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

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

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

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

The finite? procedure returns #t on all real numbers except +inf.0, -inf.0, and +nan.0, and on complex numbers if their real and imaginary parts are both finite. Otherwise it returns #f.

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

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 +nan.0, all the predicates return #f.

These predicates are required to be transitive.

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

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

\[
\text{procedure} \\
\text{(zero? } z) \\
\text{(positive? } x) \\
\text{(negative? } x) \\
\text{(odd? } n) \\
\text{(even? } n) \\
\text{procedure}
\]

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

\[
\text{procedure} \\
\text{(max } x_1 x_2 \ldots) \\
\text{(min } x_1 x_2 \ldots) \\
\text{procedure}
\]

These procedures return the maximum or minimum of their arguments.

\[
\text{(max 3 4) } \Rightarrow 4 \quad ; \text{exact} \\
\text{(max 3.9 4) } \Rightarrow 4.0 \quad ; \text{inexact}
\]

Note: If any argument is inexact, then the result will also be inexact (unless the procedure can prove that the inaccuracy is not large enough to affect the result, which is possible only in unusual implementations). If \text{min} or \text{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.

\[
\text{procedure} \\
\text{(} + z_1 \ldots \text{)} \\
\text{(} * z_1 \ldots \text{)} \\
\text{procedure}
\]

These procedures return the sum or product of their arguments.

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

\[
\text{procedure} \\
\text{(} - z_1 z_2 \text{)} \\
\text{(} - z \text{)} \\
\text{(} - z_1 z_2 \ldots \text{)} \\
\text{(} / z_1 z_2 \text{)} \\
\text{(} / z \text{)} \\
\text{(} / z_1 z_2 \ldots \text{)} \\
\text{procedure}
\]

With two or more arguments, these procedures return the difference or quotient of their arguments, associating to the left. With one argument, however, they return the additive or multiplicative inverse of their argument. It is an error if any argument of \(/\) other than the first is an exact zero.

\[
\begin{align*}
\text{(} - 3 4 \text{) } & \Rightarrow -1 \\
\text{(} - 3 4 5 \text{) } & \Rightarrow -6 \\
\text{(} - 3 \text{) } & \Rightarrow -3 \\
\text{(} / 3 4 5 \text{) } & \Rightarrow 3/20 \\
\text{(} / 3 \text{) } & \Rightarrow 1/3
\end{align*}
\]

\[
\text{procedure} \\
\text{(abs } x) \\
\text{procedure}
\]

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

\[
\text{(abs -7) } \Rightarrow 7
\]

\[
\text{procedure} \\
\text{(floor/ } n_1 n_2) \\
\text{(floor-quotient } n_1 n_2) \\
\text{(floor-remainder } n_1 n_2) \\
\text{(ceiling/ } n_1 n_2) \\
\text{(ceiling-quotient } n_1 n_2) \\
\text{(ceiling-remainder } n_1 n_2) \\
\text{(truncate/ } n_1 n_2) \\
\text{(truncate-quotient } n_1 n_2) \\
\text{(truncate-remainder } n_1 n_2) \\
\text{(round/ } n_1 n_2) \\
\text{(round-quotient } n_1 n_2) \\
\text{(round-remainder } n_1 n_2) \\
\text{(euclidean/ } n_1 n_2) \\
\text{(euclidean-quotient } n_1 n_2) \\
\text{(euclidean-remainder } n_1 n_2) \\
\text{(centered/ } n_1 n_2) \\
\text{(centered-quotient } n_1 n_2) \\
\text{(centered-remainder } n_1 n_2) \\
\text{procedure}
\]

These procedures, all in the division library, implement number-theoretic (integer) division. It is an error if \(n_2\) is zero. The procedures ending in \(/\) 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:

\[
\begin{align*}
\text{((operator)/ } n_1 n_2) & \Rightarrow n_q n_r \\
\text{((operator)-quotient } n_1 n_2) & \Rightarrow n_q \\
\text{((operator)-remainder } n_1 n_2) & \Rightarrow n_r
\end{align*}
\]

The remainder \(n_r\) 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 \(n_q\):

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

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

\[
\begin{align*}
\text{((operator)-quotient } n_1 n_2)) \\
\text{((operator)-remainder } n_1 n_2)) \Rightarrow \#t
\end{align*}
\]

provided all numbers involved in that computation are exact.

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

\[
\begin{align*}
\text{modulo } 13 \ 4 \quad &\Rightarrow \quad 1 \\
\text{remainder } 13 \ 4 \quad &\Rightarrow \quad 1 \\
\text{modulo } -13 \ 4 \quad &\Rightarrow \quad 3 \\
\text{remainder } -13 \ 4 \quad &\Rightarrow \quad -1 \\
\text{modulo } 13 \ -4 \quad &\Rightarrow \quad -3 \\
\text{remainder } 13 \ -4 \quad &\Rightarrow \quad 1 \\
\text{modulo } -13 \ -4 \quad &\Rightarrow \quad -1 \\
\text{remainder } -13 \ -4 \quad &\Rightarrow \quad -1 \\
\text{remainder } -13 \ -4.0 \quad &\Rightarrow \quad -1.0 \ ; \ \text{inexact}
\end{align*}
\]

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

\[
\begin{align*}
\text{gcd } n_1 \ \ldots \quad &\Rightarrow \quad 4 \\
\text{gcd} \quad &\Rightarrow \quad 0 \\
\text{lcm } n_1 \ \ldots \quad &\Rightarrow \quad 288 \\
\text{lcm } 32.0 \ -36 \quad &\Rightarrow \quad 288.0 \ ; \ \text{inexact} \\
\text{lcm} \quad &\Rightarrow \quad 1
\end{align*}
\]

These procedures return the greatest common divisor or least common multiple of their arguments. The result is always non-negative.

\[
\begin{align*}
\text{gcd } 32 \ -36 \quad &\Rightarrow \quad 4 \\
\text{gcd} \quad &\Rightarrow \quad 0 \\
\text{lcm } 32 \ -36 \quad &\Rightarrow \quad 288 \\
\text{lcm } 32.0 \ -36 \quad &\Rightarrow \quad 288.0 \ ; \ \text{inexact} \\
\text{lcm} \quad &\Rightarrow \quad 1
\end{align*}
\]

\[
\begin{align*}
\text{numerator } q \quad &\Rightarrow \quad 3 \\
\text{denominator } q \quad &\Rightarrow \quad 2 \\
\text{denominator} \quad &\Rightarrow \quad 2.0
\end{align*}
\]

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.

\[
\begin{align*}
\text{floor } x \quad &\Rightarrow \quad 3 \\
\text{ceiling } x \quad &\Rightarrow \quad 2 \\
\text{truncate } x \quad &\Rightarrow \quad \text{atn } 3 \\
\text{round } x \quad &\Rightarrow \quad \text{atan } y \ x
\end{align*}
\]

These procedures return integers. The floor procedure returns the largest integer not larger than \(x\). The \(\text{ceiling}\) procedure returns the smallest integer not smaller than \(x\). The \(\text{truncate}\) returns the integer closest to \(x\) whose absolute value is not larger than the absolute value of \(x\), and \(\text{round}\) returns the closest integer to \(x\), rounding to even when \(x\) is halfway between two integers.

**Rationale:** The \(\text{round}\) procedure rounds to even for consistency with the default rounding mode specified by the IEEE 754 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 \text{inexact->exact} procedure.

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

\[
\begin{align*}
\text{rationalize } x \ y \quad &\Rightarrow \quad \text{inexact}
\end{align*}
\]

The \(\text{rationalize}\) 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.

\[
\begin{align*}
\text{rationalize} \quad &\Rightarrow \quad 1/3 \ ; \ \text{exact} \\
\text{rationalize} \quad &\Rightarrow \quad \text{#i1/3} \ ; \ \text{inexact}
\end{align*}
\]

\[
\begin{align*}
\text{exp } z \quad &\Rightarrow \quad \text{inexact library procedure} \\
\text{log } z \quad &\Rightarrow \quad \text{inexact library procedure} \\
\text{sin } z \quad &\Rightarrow \quad \text{inexact library procedure} \\
\text{cos } z \quad &\Rightarrow \quad \text{inexact library procedure} \\
\text{tan } z \quad &\Rightarrow \quad \text{inexact library procedure} \\
\text{asin } z \quad &\Rightarrow \quad \text{inexact library procedure} \\
\text{acos } z \quad &\Rightarrow \quad \text{inexact library procedure} \\
\text{atan } z \quad &\Rightarrow \quad \text{inexact library procedure} \\
\text{atan } y \ x \quad &\Rightarrow \quad \text{inexact library procedure}
\end{align*}
\]

These procedures compute the usual transcendental functions. The \(\text{log}\) procedure computes the natural logarithm of \(z\) (not the base ten logarithm). The \(\text{asin}, \text{acos}, \text{atan}\) procedures are provided for backward compatibility with earlier versions of this report.
acos, and atan procedures compute arcsine ($\sin^{-1}$), arc-
cosine ($\cos^{-1}$), and arctangent ($\tan^{-1}$), respectively.
The two-argument variant of atan computes (angle
(make-rectangular $x$ $y$)) (see below), even in imple-
mentations that don’t support the complex library.

In general, the mathematical functions log, arcsine, arc-
cosine, 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 formulæ:

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

The above specification follows [33], 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.

\[
\begin{align*}
\text{sin}^{-1} z &= \text{angle}(\text{magnitude } z) \\
\text{cos}^{-1} z &= \frac{\pi}{2} - \text{sin}^{-1} z \\
\text{tan}^{-1} z &= \text{rational approximation, or may report an implementation restriction may be reported.}
\end{align*}
\]

Then all of
\[
\begin{align*}
\text{make-rectangular } x_1 x_2 &\quad \Rightarrow z \\
\text{make-polar } x_3 x_4 &\quad \Rightarrow z \\
\text{real-part } z &\quad \Rightarrow x_1 \\
\text{imag-part } z &\quad \Rightarrow x_2 \\
\text{magnitude } z &\quad \Rightarrow |x_3| \\
\text{angle } z &\quad \Rightarrow x_{\text{angle}}
\end{align*}
\]

are true, where $-\pi < x_{\text{angle}} \leq \pi$ with $x_{\text{angle}} = x_4 + 2\pi n$
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 is in the base library, whereas magnitude
is in the optional complex library.

\[
\begin{align*}
\text{exact->inexact } z &\quad \text{procedure} \\
\text{inexact->exact } z &\quad \text{procedure}
\end{align*}
\]

The procedure exact->inexact returns an inexact repre-
sentation 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 representa-
tion 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 corre-
spondence between exact and inexact integers throughout
an implementation-dependent range. See section 6.2.3.
6.2.7. Numerical input and output

\[(\text{number->string } z)\]  \hspace{1cm} \text{procedure}
\[(\text{number->string } z \text{ radix})\]  \hspace{1cm} \text{procedure}

It is an error if \text{radix} is not one of 2, 8, 10, or 16. If omitted, \text{radix} defaults to 10. The procedure \text{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

\[
\text{(let ((number number)}
\hspace{1cm} (\text{radix radix}))
\hspace{1cm} (\text{eqv? number}
\hspace{1cm} (\text{string->number (string->number number radix})
\hspace{1cm} (\text{radix})))
\]

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; otherwise the format of the result is unspecified.

The result returned by \text{number->string} never contains an explicit radix prefix.

\text{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.

\text{Rationale:} If \(z\) is an inexact number 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 unusual representations.

\[(\text{string->number } string)\]  \hspace{1cm} \text{procedure}
\[(\text{string->number } string \text{ radix})\]  \hspace{1cm} \text{procedure}

Returns a number of the maximally precise representation expressed by the given \text{string}. It is an error if \text{radix} is not 2, 8, 10, or 16. If supplied, \text{radix} is a default radix that will be overridden by an explicit radix prefix in \text{string} (e.g. "#o177"). If \text{radix} is not supplied, then the default radix is 10. If \text{string} is not a syntactically valid notation for a number, then \text{string->number} returns \#f.

\[
\begin{align*}
(\text{string->number } "100") & \Rightarrow 100 \\
(\text{string->number } "100" \text{ 16}) & \Rightarrow 256 \\
(\text{string->number } "1e2") & \Rightarrow 100.0
\end{align*}
\]

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

6.3. Booleans

The standard boolean objects for true and false are written as \#t and \#f. Alternatively, they may be written \text{true} and \text{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 Scheme values, only \#f counts as false in conditional expressions. All other Scheme values, including \#t, count as true.

\text{Note:} Unlike some other dialects of Lisp, Scheme distinguishes \#f and the empty list from each other and from the symbol \text{nil}.

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

\[
\begin{align*}
\#t & \Rightarrow \#t \\
\#f & \Rightarrow \#f
\end{align*}
\]

\text{(not \text{obj})} \hspace{1cm} \text{procedure}

The not procedure returns \#t if \text{obj} is false, and returns \#f otherwise.

\[
\begin{align*}
(\text{not } \#t) & \Rightarrow \#f \\
(\text{not } 3) & \Rightarrow \#f \\
(\text{not } (\text{list } 3)) & \Rightarrow \#f \\
(\text{not } \#f) & \Rightarrow \#t \\
(\text{not } ')() & \Rightarrow \#f \\
(\text{not } (\text{list})) & \Rightarrow \#f \\
(\text{not } 'nil) & \Rightarrow \#f
\end{align*}
\]

\text{(boolean? \text{obj})} \hspace{1cm} \text{procedure}

The boolean? predicate returns \#t if \text{obj} is either \#t or \#f and returns \#f otherwise.

\[
\begin{align*}
(\text{boolean? } \#f) & \Rightarrow \#t \\
(\text{boolean? } 0) & \Rightarrow \#f \\
(\text{boolean? } '()) & \Rightarrow \#f
\end{align*}
\]
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. It is not a pair, it has no elements, and its length is zero.

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

The most general notation (external representation) for Scheme pairs is the “dotted” notation \((c_1 . c_2)\) where \(c_1\) is the value of the car field and \(c_2\) is the value of the cdr field. For example \((4 . 5)\) is a pair whose car is 4 and whose cdr is 5. Note that \((4 . 5)\) is the external representation of a pair, not an expression that evaluates to a pair.

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

\[(a \ b \ c \ d \ e)\]

and

\[(a . (b . (c . (d . (e . ()))))\]

are equivalent notations for a list of symbols.

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

\[(a \ b \ c . d)\]

is equivalent to

\[(a . (b . (c . d)))\]

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

\[
\begin{align*}
\text{(define } x \text{ (list } 'a \ 'b \ 'c)\text{)} \\
\text{(define } y \text{ x)} \\
\text{(list? } y \text{)} & \Rightarrow (a \ b \ c) \\
\text{(set-cdr! } x \text{ 4)} & \Rightarrow \#t \\
\text{(eqv? } x \ y \text{)} & \Rightarrow \#t \\
\text{(list? } x \text{)} & \Rightarrow (a \ . \ 4) \\
\text{(list? } y \text{)} & \Rightarrow \#f \\
\text{(set-cdr! } x \ x \text{)} & \Rightarrow \text{unspecified} \\
\text{(list? } x \text{)} & \Rightarrow \#f 
\end{align*}
\]

Within literal expressions and representations of objects read by the read procedure, the forms \(\langle\text{datum}\rangle\), \(\langle\text{datum}, \text{datum}\rangle\), and \(\langle\text{datum}\rangle\) denote two-element lists whose first elements are the symbols quote, quasiquote, unquote, and unquote-splicing, respectively. The second element in each case is \(\langle\text{datum}\rangle\). This convention is supported so that arbitrary Scheme programs can be represented as lists. That is, according to Scheme’s grammar, every \(\langle\text{expression}\rangle\) is also a \(\langle\text{datum}\rangle\) (see section 7.1.2). Among other things, this permits the use of the read procedure to parse Scheme programs. See section 5.3.

\[
\begin{align*}
\text{(pair? } \text{obj}\text{)} & \quad \text{procedure} \\
\text{The pair? predicate returns } \#t \text{ if } \text{obj} \text{ is a pair, and otherwise returns } \#f. \\
\text{(pair? } '(a \ . \ b)\text{)} & \Rightarrow \#t \\
\text{(pair? } '(a \ b \ c)\text{)} & \Rightarrow \#t \\
\text{(pair? } '()\text{)} & \Rightarrow \#f \\
\text{(pair? } '#(a \ b)\text{)} & \Rightarrow \#f
\end{align*}
\]

\[
\begin{align*}
\text{(cons } \text{obj}_1 \ \text{obj}_2\text{)} & \quad \text{procedure} \\
\text{Returns a newly allocated pair whose car is } \text{obj}_1 \text{ and whose cdr is } \text{obj}_2. \text{ The pair is guaranteed to be different (in the sense of eqv?) from every existing object.} \\
\text{(cons } 'a \ '()\text{)} & \Rightarrow (a) \\
\text{(cons } '(a) \ '(b \ c \ d)\text{)} & \Rightarrow ((a) \ b \ c \ d) \\
\text{(cons } "a" \ '(b \ c)\text{)} & \Rightarrow ("a" \ b \ c) \\
\text{(cons } 'a \ 3\text{)} & \Rightarrow (a . 3) \\
\text{(cons } '(a \ b) \ 'c\text{)} & \Rightarrow ((a \ b) . c)
\end{align*}
\]

\[
\begin{align*}
\text{(car } \text{pair}\text{)} & \quad \text{procedure} \\
\text{Returns the contents of the car field of } \text{pair}. \text{ Note that it is an error to take the car of the empty list.}
\end{align*}
\]
(car '(a b c)) ⇒ a
(car '((a) b c d)) ⇒ (a)
(car '(1 . 2)) ⇒ 2
(car '()) ⇒ error

(cdr pair)
Returns the contents of the cdr field of pair. Note that it is an error to take the cdr of the empty list.
(cdr '((a) b c d)) ⇒ (b c d)
(cdr '(1 . 2)) ⇒ 2
(cdr '()) ⇒ error

(set-car! pair obj)
Stores obj in the car field of pair. The value returned by set-car! is unspecified.
(define (f) (list 'not-a-constant-list))
(define (g) '(constant-list))
(set-car! (f) 3) ⇒ unspecified
(set-car! (g) 3) ⇒ error

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

(caar pair)
(cadr pair)
(cddr pair)
(cdddar pair)
These procedures are compositions of car and cdr, where for example caddr could be defined by
(define caddr (lambda (x) (car (cdr (cdr x))))).
Arbitrary compositions, up to four deep, are provided. There are twenty-eight of these procedures in all.

(null? obj)
Returns #t if obj is the empty list, otherwise returns #f.

(list? obj)
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.

(make-list k)
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.

(list obj ...)
Returns a newly allocated list of its arguments.
(length list)
Returns the length of list.

(append list ...)
Returns a list consisting of the elements of the first list followed by the elements of the other lists.
(reverse list)
Returns a newly allocated list consisting of the elements of list in reverse order.
(list-tail list k)
Returns the sublist of list obtained by omitting the first k elements. It is an error if list has fewer than k elements. The list-tail procedure could be defined by
(define list-tail
  (lambda (x k)
    (if (zero? k)
        x
        (list-tail (cdr x) (- k 1)))))

(list-ref list k) procedure

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

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

(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 re-
turned by list-set! is unspecified.

(let ((ls (list 'one 'two 'five!)))
  (list-set! ls 2 'three)
  ls)
⇒ (one two three)

(list-set! '(0 1 2) 1 "oops")
⇒ error ; constant list

(memq obj list) procedure

member uses compare, if given, and equal? otherwise.

(memq 'a '(a b c)) ⇒ (a b c)
(memq 'b '(a b c)) ⇒ (b c)
(memq 'a '(b c d)) ⇒ #f
(memq (list 'a) '(b (a) c)) ⇒ #f
(member (list 'a)
  '(b (a) c)) ⇒ ((a) c)
(member "B"
  "("a" "b" "c")"
  string-ci=?)
⇒ ("b" "c")
(memq 101 '(100 101 102)) ⇒ unspecified
(memv 101 '(100 101 102)) ⇒ (101 102)

(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 assq procedure uses eq? to compare obj with the car
fields of the pairs in alist, while assv uses eqv? and assoc
uses compare if given and equal? otherwise.

(define e '((a 1) (b 2) (c 3)))
(assq 'a e) ⇒ (a 1)
(assq 'b e) ⇒ (b 2)
(assq 'd e) ⇒ #f
(assq (list 'a) '(((a)) ((b)) ((c))))
⇒ #f
(assoc (list 'a) '(((a)) ((b)) ((c))))
⇒ ((a))
(assoc 2.0 '(((1 2) (3 4) (9 2)))
⇒ (2 4)
(assq 5 '((2 3) (5 7) (11 13)))
⇒ unspecified
(assv 5 '((2 3) (5 7) (11 13)))
⇒ (5 7)

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

(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 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 eqv?) if and only
if their names are spelled the same way. For instance, they
can be used the way enumerated values are used in other
languages.

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

It is guaranteed that any symbol that has been returned
as part of a literal expression, or read using the read pro-
cedure, and subsequently written out using the write pro-
procedure, will read back in as the identical symbol (in the
sense of eqv?).
Note: Some implementations have values known as “uninterred symbols,” which defeat write/read invariance, and also violate the rule that two symbols are the same if and only if their names are spelled the same.

(symbol? obj) procedure

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

(symbol? 'foo) ➞ #t
(symbol? (car '(a b))) ➞ #t
(symbol? "bar") ➞ #f
(symbol? 'nil) ➞ #t
(symbol? '()) ➞ #f
(symbol? #f) ➞ #f

(symbol->string symbol) procedure

Returns the name of symbol as a string. It is an error to apply mutation procedures like string-set! to strings returned by this procedure.

(symbol->string 'flying-fish) ➞ "flying-fish"
(symbol->string 'Martin) ➞ "Martin"
(symbol->string (string->symbol "Malvina")) ➞ "Malvina"

(string->symbol string) procedure

Returns the symbol whose name is string. This procedure can create symbols with names containing special characters that would require escaping when written.

(string->symbol "mISSISSIppi") ➞ mISSISSIppi
(eq? 'bitBlt (string->symbol "bitBlt")) ➞ #t
(eq? 'JollyWog (string->symbol "JollyWog")) ➞ #t
(string=? "K. Harper, M.D." (string->symbol "K. Harper, M.D.")) ➞ #t

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 #\(character) or #\(character name) or #\x{hex scalar value}.

Here are some examples:

#\a; lower case letter
#\A; upper case letter
#\(; left parenthesis
#\; the space character
#\iota; ι (if supported)
#\x03BB; λ (if supported)

The following character names must be supported by all implementations:

#\alarm; U+0007
#\backspace; U+0008
#\delete; U+007F
#\escape; U+001B
#\newline; the linefeed character, U+000A
#\null; the null character, U+0000
#\return; the return character, U+000D
#\space; the preferred way to write a space
#\tab; the tab character, U+0009

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

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

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

(char? obj) procedure

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

(char=? char1 char2 char3 ...) procedure
(char>? char1 char2 char3 ...) procedure
(char<=? char1 char2 char3 ...) procedure
(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 which is the same as the Unicode code point ordering. This is true whether or not the implementation uses the Unicode representation internally.

\[(\text{char-ci=} \ char_1 \ char_2 \ char_3 \ ... )\] char library procedure
\[(\text{char-ci<} \ char_1 \ char_2 \ char_3 \ ... )\] char library procedure
\[(\text{char-ci>} \ char_1 \ char_2 \ char_3 \ ... )\] char library procedure
\[(\text{char-ci<=} \ char_1 \ char_2 \ char_3 \ ... )\] char library procedure
\[(\text{char-ci>=} \ char_1 \ char_2 \ char_3 \ ... )\] char library procedure

These procedures are similar to \text{char=} et cetera, but they treat upper case and lower case letters as the same. For example, \[(\text{char-ci=} \text\#\text{A} \text\#\text{a})\] returns \#t.

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

\[(\text{char-alphabetic? char})\] char library procedure
\[(\text{char-numeric? char})\] char library procedure
\[(\text{char-whitespace? char})\] char library procedure
\[(\text{char-upper-case? letter})\] char library procedure
\[(\text{char-lower-case? letter})\] char library procedure

These procedures return \#t if their arguments are alphabetic, numeric, whitespace, upper case, or lower case characters, respectively, otherwise they return \#f.

Specifically, they must return \#t when applied to characters with the Unicode properties Alphabetic, Numeric_Digit, White_Space, 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.

\[(\text{digit-value char})\] char library procedure

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

\[(\text{digit-value \#\text{3})} \Rightarrow 3\]
\[(\text{digit-value \#\text{x0664})} \Rightarrow 4\]
\[(\text{digit-value \#\text{x0EA6})} \Rightarrow 0\]

Given a Unicode character, \text{char->integer} returns an exact integer between 0 and \#x7FFFF or between \#xE0000 and \#x10FFFF 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 value returned by a character when \text{char->integer} is applied to it, \text{integer->char} returns that character.

\[(\text{char-upcase char})\] char library procedure
\[(\text{char-downcase char})\] char library procedure
\[(\text{char-foldcase char})\] char library procedure

The \text{char-upcase} procedure, given an argument that is the lowercase 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.

The \text{char-downcase} procedure, given an argument that is the uppercase 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 uppercase member of such a pair, it is returned.

The \text{char-foldcase} procedure applies the Unicode simple case-folding algorithm to its argument and returns the result. Note that language-sensitive folding is not used. If the argument is an uppercase letter, the result will be either a lowercase letter or the same as the argument if the lowercase letter does not exist or is not supported by the implementation. See UAX \#29 (part of the Unicode Standard) for details.

Note that many Unicode lowercase characters do not have uppercase equivalents.

### 6.7. Strings

Strings are sequences of characters. Strings are written as sequences of characters enclosed within doublequotes ("'). Within a string literal, various escape sequences represent characters other than themselves. Escape sequences always start with a backslash (\):

- \text{\#a} : alarm, U+0007
- \text{\#b} : backspace, U+0008
• \t: character tabulation, U+0009
• \n: linefeed, U+000A
• \r: return, U+000D
• " : doublequote, U+0022
• \:\: backslash, U+005C
• \ ⟨intraline 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 \ ⟨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"
"Here's a text \ "containing just one line"
"\x03B1; is named GREEK SMALL LETTER ALPHA."

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 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 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-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 NFD as specified by Unicode TR#15. Implementations which only support ASCII or some other character set which provides no ambiguous representations of character sequences may define the normalization to be the identity operation, in which case string-ni=? is equivalent to string=?.

In all cases, a pair of strings must satisfy exactly one of string<?, string=?, and string>?; and must satisfy string<? 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 upperscasing, 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 lowers 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

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

The 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 are the concatenation of the characters in the given strings.

(string->list string) procedure
(list->string list) procedure

string->list returns a newly allocated list of the characters that make up the given string. list->string returns a newly allocated string formed from the elements in the list list. It is an error if any element is not a character. string->list and list->string are inverses so far as equal? is concerned.
6. Standard procedures 45

(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
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 and endy are omitted, fill is stored in all the elements of string. In either case, an unspecified value is returned.

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 needed to access a randomly chosen element is typically less for the vector than for the list.

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

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

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

Note that this is the external representation of a vector, not an expression evaluating to a vector. It is an error not to quote a vector constant:

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

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

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

(vector obj ...) procedure
Returns a newly allocated vector whose elements contain the given arguments. 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 returns the contents of element k of vector.

(vector-ref '#(1 1 2 3 5 8 13 21) 5) ⇒ 8
(vector-ref '#(1 1 2 3 5 8 13 21) (inexact->exact (round (* 2 (acos -1))))) ⇒ 13

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

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

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

(vector->list vector) procedure
(list->vector list) procedure
vector->list returns a newly allocated list of the objects contained in the elements of vector. list->vector returns a newly created vector initialized to the elements of the list.

(vector->list '#(dah dah didah))
⇒ (dah dah didah)
(list->vector '(dididit dah))
⇒ #((dididit dah))

(vector->string string) procedure
(string->vector vector) procedure
vector->string returns a newly allocated string of the objects contained in the elements of vector. It is an error if any element is not a character allowed in strings.
string->vector returns a newly created vector initialized to the elements of the string string.

(vector->string "ABC")  ⇒ #(\A \B \C)
(vector->string #(\1 \2 \3)  ⇒ "123"
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.

(vector-copy vector) procedure

(vector-copy vector start) procedure

(vector-copy vector start end) procedure

(vector-copy vector start end fill) procedure

Returns a newly allocated copy of the given vector. The 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 respectively. If end is greater than the length of vector, the fill argument is used to fill the additional elements of the result.

(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.

(bytevector-u8-ref bytevector k) procedure

Returns the \(k\)th byte of bytevector.

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

Stores byte as the \(k\)th byte of bytevector. The value returned by bytevector-u8-set! is unspecified.

(bytevector-copy bytevector) procedure

Returns a newly allocated bytevector containing the same bytes as bytevector.

(bytevector-copy! from to) procedure

Copies the bytes of bytevector from to bytevector to. It is an error if to is shorter than from. The value returned by bytevector-copy! is unspecified.

(bytevector-copy-partial bytevector start end) procedure

Returns a newly allocated bytevector containing the bytes in bytevector between start (inclusive) and end (exclusive).

(bytevector-copy-partial! from start end to at) procedure

Copies the bytes of bytevector from between start and end to bytevector to, starting at at. The order in which bytes are copied is unspecified, except that if the source and destination overlap, copying takes place as if the source is first copied into a temporary bytevector and then into the destination. This can be achieved without allocating storage by making sure to copy in the correct direction in such circumstances.

It is an error if the inequality \((>= (- (bytevector-length to) at) (- end start))\) is false. The value returned by bytevector-copy-partial! is unspecified.

(utf8->string bytevector) procedure

(string->utf8 string) procedure

These procedures translate between strings and bytevectors that encode those strings using the UTF-8 encoding. The utf8->string procedure decodes a bytevector and returns the corresponding string; the string->utf8 procedure encodes a string and returns the corresponding bytevector. It is an error to pass invalid byte sequences or byte sequences representing characters which are forbidden in strings to utf8->string.

(utf8->string #u8(#x41)) \(\Rightarrow \text{"A"}\)

(string->utf8 "\lambda") \(\Rightarrow \text{#u8(#xCE #xBB)}\)
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)

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

(procedure? car) ⇒ #t
(procedure? 'car) ⇒ #f
(procedure? (lambda (x) (* x x))) ⇒ #t
(procedure? '(lambda (x) (* x x))) ⇒ #f

(call-with-current-continuation procedure?) ⇒ #t

(apply proc arg1 ... args)

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

(apply + (list 3 4)) ⇒ 7

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

((compose sqrt *) 12 75) ⇒ 30

(map proc list1 list2 ...) 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 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.

(map cadr '((a b) (d e) (g h))) ⇒ (b e h)
(map (lambda (n) (expt n n)) '(1 2 3 4 5)) ⇒ (1 4 27 256 3125)
(map + '(1 2 3) '(4 5 6 7)) ⇒ (5 7 9)

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

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

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. string-map applies proc element-wise to the elements of the 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 returns occur from string-map, the values returned by earlier returns are not mutated.

(string-map char-foldcase "AbdEgH") ⇒ "abdegh"

(string-map
  (lambda (c)
    (if (eqv? c #\u) char-upcase char-downcase))
  "HAL") ⇒ "IBM"

(string-map
  (lambda (c k)
    ((if (eqv? k #\u) char-upcase char-downcase) c))
  "studlycaps xxx"
  "ululululul") ⇒ "StUdLyCaPs"

(vector-map proc vector1 vector2 ...) procedure

It is an error if proc does not accept as many arguments as there are vectors 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 applies proc element-wise to the elements of the vectors and returns a vector of the results, in order. The dynamic order in which proc is applied to the elements of the vectors is unspecified. If multiple returns occur from vector-map, the values returned by earlier returns are not mutated.

(vector-map cadr '#((a b) (d e) (g h))) ⇒ #b e h
(vector-map (lambda (n) (expt n n)) '#(1 2 3 4 5)) ⇒ #1 27 256 3125
(vector-map + '#(1 2 3) '#(4 5 6 7)) ⇒ #6 7 9

(let ((count 0))
  (vector-map
    (lambda (ignored)
      (set! count (+ count 1))
      count)
    '#(a b))) ⇒ #1 2 or #2 1
The arguments to for-each are like the arguments to map, but for-each calls proc for its side effects rather than for its values. Unlike map, for-each is guaranteed to call proc on the elements of the lists in order from the first element(s) to the last, and the value returned by for-each is unspecified. It is an error for proc to mutate any of the lists. If more than one list is given and not all lists have the same length, for-each terminates when the shortest list runs out.

```
(let ((v (make-vector 5)))
  (for-each (lambda (i)
               (vector-set! v i (* i i)))
             '(0 1 2 3 4))
  v)  ⇒  #(0 1 4 9 16)
```

The arguments to vector-for-each are like the arguments to vector-map, but vector-for-each calls proc for its side effects rather than for its values. Unlike vector-map, vector-for-each is guaranteed to call proc on the elements of the vectors in order from the first element(s) to the last, and the value returned by vector-for-each is unspecified. If more than one vector is given and not all vectors have the same length, vector-for-each terminates when the shortest vector runs out.

```
(let ((v '()))
  (vector-for-each (lambda (x)
                    (exit x))
                   '(54 0 37 -3 245 19))
  v)  ⇒  (101 100 99 98 97)
```

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

However, the continuations of all non-final expressions within a sequence of expressions, such as in lambda, case-lambda, begin, let, let*, letrec, let-values, let*-values, let-syntax, letrec-syntax, parameterize, guard, case, cond, when, and unless expressions, take an arbitrary number of values, because they discard the values passed to them in any event.

The following examples show only the simplest ways in which call-with-current-continuation is used. If all real uses were as simple as these examples, there would be no need for a procedure with the power of call-with-current-continuation.

```
(define list-length
  (lambda (obj)
    (call-with-current-continuation
     (lambda (return)
      (letrec ((r
                   (lambda (obj)
                    (cond ((null? obj) 0)
                           ((pair? obj)
                            (+ (r (cdr obj)) 1))
                           (else (return #f)))))))
       (list-length '(1 2 3 4)))  ⇒  4)
```
(list-length '(a b . c))  ⇒  #f

Rationale:

A common use of call-with-current-continuation is for structured, non-local exits from loops or procedure bodies, but in fact call-with-current-continuation is useful for implementing a wide variety of advanced control structures.

Whenever a Scheme expression is evaluated there is a continuation wanting the result of the expression. The continuation represents an entire (default) future for the computation. If the expression is evaluated at top level, for example, then the continuation might take the result, print it on the screen, prompt for the next input, evaluate it, and so on forever. Most of the time the continuation includes actions specified by user code, as in a continuation that will take the result, multiply it by the value stored in a local variable, add seven, and give the answer to the top level continuation to be printed. Normally these ubiquitous continuations are hidden behind the scenes and programmers do not think much about them. On rare occasions, however, a programmer needs to deal with continuations explicitly. The 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 exit, return, or even 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 MacLisp. Several Scheme implementors noticed Reynolds’s construct, though its name came from a less general construct in MacLisp. Several Scheme implementors noticed that the full power of catch could be provided by a procedure instead of by a special syntactic construct, and the name call-with-current-continuation was coined in 1982. This name is descriptive, but opinions differ on the merits of such a long name, and some people prefer the name call/cc instead.

(values obj ...)  procedure

Delivers all of its arguments to its continuation. Values might be defined as follows:

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

(call-with-values producer consumer)  procedure

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

6. Standard procedures 49

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

(dynamic-wind before thunk after)  procedure

Calls thunk without arguments, returning the result(s) of this call. Before and after are called, also without arguments, as required by the following rules. Note that, in the absence of calls to continuations captured using call-with-current-continuation, the three arguments are called once each, in order. Before is called whenever execution enters the dynamic extent of the call to thunk and after is called whenever it exits that dynamic extent. The dynamic extent of a procedure call is the period between when the call is initiated and when it returns. Before and after are excluded from the dynamic extent. In Scheme, because of call-with-current-continuation, the dynamic extent of a call 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 execution is not within the dynamic extent and a continuation is invoked that was captured (using call-with-current-continuation) during the dynamic extent.
- It is exited when the called procedure returns.
- It is also exited when execution is within the dynamic extent and a continuation is invoked that was captured while not within the dynamic extent.

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

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

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

The effect of using a captured continuation to enter or exit the dynamic extent of a call to before or after is 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 (c0)
               (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 1.3.2. See also 4.2.7 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.

(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.

(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

\( \text{eval expression environment-specifier} \)

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 a datum. Implementations may extend eval to allow non-expression programs such as definitions as the first argument, with the restriction that eval is not allowed to create new bindings in the environments returned by null-environment or scheme-report-environment.

\( \text{(eval '(* 7 3) (scheme-report-environment 7))} \)

⇒ 21

\( \text{(let ((f (eval '(lambda (f x) (f x x)) (null-environment 7)))) (f + 10))} \)

⇒ 20

\( \text{(scheme-report-environment version)} \)

eval library procedure

If version is equal to 7, corresponding to this revision of the Scheme report (the Revised 7 Report on Scheme), scheme-report-environment returns a specifier for an environment that contains only the bindings defined either in the base library or in the other libraries of this report that the implementation supports. Implementations must support this value of version.

Implementations may also support other values of version, 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.

\( \text{(null-environment version)} \)

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.

\( \text{(environment list1 ...)} \)

eval library procedure

This procedure returns a specifier for the environment that results by starting with an empty environment and then importing each list, considered as an import set, into it. (See section 5.5 for a description of import sets.) The bindings of the environment represented by the specifier are immutable.

\( \text{(interaction-environment)} \)

repl library procedure

This procedure returns a specifier for an environment that contains an implementation-defined set of bindings, typically a superset of those exported by (scheme base). The intent is that this procedure will return the environment in which the implementation would evaluate expressions 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 data upon command, while an output port is a Scheme object that can accept 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 characters 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.

\( \text{(call-with-input-file string proc)} \)

file library procedure

\( \text{(call-with-output-file string proc)} \)

file library procedure

It is an error if proc 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 values yielded by the proc are returned. If proc does not return, then the port must not be closed automatically unless it is possible to prove that the port will never again be used for a read or write operation.

**Rationale:** Because Scheme’s escape procedures have unlimited extent, it is possible to escape from the current continuation but later to escape back in. If implementations were permitted to close the port on any escape from the current continuation, then it would be impossible to write portable code using both call-with-current-continuation and call-with-input-file or call-with-output-file.

(call-with-port port proc) procedure

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.

(input-port? obj) procedure

(output-port? obj) procedure

(textual-port? obj) procedure

(binary-port? obj) procedure

(port? obj) procedure

These procedures return #t if obj is an input port, 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 port, output port, or error port (an output port), respectively. These procedures are parameter objects, which can be overridden with parameterize (see section 4.2.6). The initial bindings for these are system-defined textual ports.

(with-input-from-file string thunk) procedure

(with-output-to-file string thunk) procedure

It is an error if thunk does not accept one argument. For with-input-from-file, it is an error 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 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 string) file library procedure

(with-output-to-file string) file library procedure

Takes a string and returns a textual input port or binary input port capable of delivering data from the file. If the file cannot be opened, an error is signalled.

Takes a string naming an output file to be created and returns a textual output port or binary output port capable of writing data to a new file by that name. If the file cannot be opened, an error is signalled. If a file with the given name already exists, the effect is unspecified.

(close-port port) procedure

(close-input-port port) procedure

(close-output-port port) procedure

Closes the resource associated with port, rendering the port incapable of delivering or accepting data. It is an error to apply the last two procedures to a port which is not an input or output port, respectively. Scheme implementations may provide ports which are simultaneously input and output ports, such as sockets; the close-input-port and close-output-port procedures can then be used to close the input and output sides of the port independently.

These routines have no effect if the file has already been closed. The value returned is unspecified.

(open-input-string string) 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 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.

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.

(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

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.

(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.

(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.

(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 file 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.

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

Returns #t if the 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.
ports must ensure that characters whose existence has been asserted by \texttt{char-ready?} cannot be rubbed out. If \texttt{char-ready?} were to return \texttt{#f} at end of file, a port at end of file would be indistinguishable from an interactive port that has no ready characters.

\begin{verbatim}
(read-u8) procedure
(read-u8 port) procedure

Returns the next byte available from the binary input \texttt{port}, updating the \texttt{port} to point to the following byte. If no more bytes are available, an end of file object is returned. \texttt{Port} may be omitted, in which case it defaults to the value returned by \texttt{current-input-port}.

(peek-u8) procedure
(peek-u8 port) procedure

Returns the next byte available from the binary input \texttt{port}, \textit{without} updating the \texttt{port} to point to the following byte. If no more bytes are available, an end of file object is returned. \texttt{Port} may be omitted, in which case it defaults to the value returned by \texttt{current-input-port}.

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

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

(read-bytevector length) procedure
(read-bytevector length port) procedure

Reads the next \texttt{length} bytes, or as many as are available before the end of file, from the binary input \texttt{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. \texttt{Port} may be omitted, in which case it defaults to the value returned by \texttt{current-input-port}.

(read-bytevector! bytevector start end) procedure
(read-bytevector! bytevector start end port) procedure

Reads the next \texttt{end} – \texttt{start} bytes, or as many as are available before the end of file, from the binary input \texttt{port} into \texttt{bytevector} in left-to-right order beginning at the \texttt{start} position. Returns the number of bytes read. If no bytes are available, an end of file object is returned. \texttt{Port} may be omitted, in which case it defaults to the value returned by \texttt{current-input-port}.

\end{verbatim}

\subsection{Output}

\begin{verbatim}
(write obj) write library procedure
(write obj port) write library procedure

\textit{Port} may be omitted, in which case it defaults to the value returned by \texttt{current-output-port}.

(write-simple obj) write library procedure
(write-simple obj port) write library procedure

\texttt{write-simple} is the same as \texttt{write}, except that shared structure is not represented using datum labels. This may cause \texttt{write-simple} not to terminate if \texttt{obj} contains circular structure.

(display obj) write library procedure
(display obj port) write library procedure

\texttt{write} is intended for producing machine-readable output and \texttt{display} for producing human-readable output.

(newline) procedure
(newline port) procedure

Writes an end of line to textual output \texttt{port}. Exactly how this is done differs from one operating system to another. Returns an unspecified value. \texttt{Port} may be omitted, in which case it defaults to the value returned by \texttt{current-output-port}.

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

Writes the character \texttt{char} (not an external representation of the character) to the given textual output \texttt{port} and returns an unspecified value. \texttt{Port} may be omitted, in which case it defaults to the value returned by \texttt{current-output-port}.
\end{verbatim}
(write-u8 byte) procedure
    Writes the byte to the given binary output port and returns an unspecified value. Port may be omitted, in which case it defaults to the value returned by current-output-port.

(write-u8 byte port) procedure

(write-bytevector bytevector) procedure
    Writes 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.

(write-bytevector bytevector port) procedure

(write-partial-bytevector bytevector start end) 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.

(write-partial-bytevector bytevector start end port) procedure

(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.

(flush-output-port port) procedure

(file-exists? filename) file library procedure
    It is an error if filename is not a string. The file-exists? 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 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 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 6.12) 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.

    (get-environment-variable "PATH")
    ⇒ "/usr/local/bin:/usr/bin:/bin"

(get-environment-variables) process-context library procedure
    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.

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
    An implementation-dependent operation is 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. It returns an unspecified value.

    Rationale: For portability, load must operate on source files. Its operation on other kinds of files necessarily varies among implementations.
(get-environment-variables)
⇒ (("USER" . "root") ("HOME" . "/"))

(current-second) 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, 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.

(current-jiffy) 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.

(jiffies-per-second) 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))
        (start (current-jiffy)))
    (length list)
    (/ (- (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⟩
     | ( | ) | #⟨ | #u⟨ | , | , , , , , , , .
⟨delimiter⟩ → ⟨whitespace⟩ | ( | ) | \ | ;
⟨intraline whitespace⟩ → ⟨space or tab⟩
Note that +i, -i and \infty below are exceptions to the
(partial identifier) rule; they are parsed as numbers, not
identifiers.

(identifier) → (initial) (subsequent)*
| (vertical bar) (symbol element)* (vertical bar)
| (partial identifier)

(initial) → (letter) | (special initial)

| (inline hex escape)

(letter) → a | b | c | \ldots | z
| A | B | C | \ldots | Z

(special initial) → ! | $ | % | & | * | / | : | | < | =
| > | ? | ^ | _ | ~

(subsequent) → (initial) | (digit)

| (special subsequent)

(digit) → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

(hex digit) → (digit)
| a | b | c | d | e | f | F

(explicit sign) → + | -

(special subsequent) → (explicit sign) | . | @

(inline hex escape) → \x(x(hex scalar value));

(hex scalar value) → (hex digit)*

(partial identifier) → (explicit sign)
| (explicit sign) (sign subsequent) (subsequent)*
| (explicit sign) . (dot subsequent) (subsequent)*
| . (non-digit) (subsequent)*

(non-digit) → (dot subsequent) | (explicit sign)

(dot subsequent) → (sign subsequent) | .

(sign subsequent) → (initial) | (explicit sign) | @

(symbol element) →

(any character other than (vertical bar) or \}

(boolean) → #t | #f | #true | #false

(character) → #\ (any character)
| #\ (character name)
| #\(hex scalar value)

(character name) → alarm | backspace | delete
| escape | newline | null | return | space | tab

(string) → " (string element)* "

(string element) → (any character other than " or \)

The following rules for \num \Ra, \complex \Ra, \real \Ra, \ureal \Ra, \uinteger \Ra, and \prefix \Ra are implicitly
replicated for \Ra = 2, 8, 10, and 16. There are no rules for
\decimal 2, \decimal 8, and \decimal 16, which means
that numbers containing decimal points or exponents are
always in decimal radix. Although not shown below, all al-
phabetic characters used in the grammar of numbers may
appear in either upper or lower case.

\num \Ra → \prefix \Ra \complex \Ra
\complex \Ra → \real \Ra | \real \Ra \@ \real \Ra
| \real \Ra \+ \ureal \Ra \i | \real \Ra \- \ureal \Ra \i
| \real \Ra \+ \i | \real \Ra \- \i | \real \Ra \infinity \i
| \+ \ureal \Ra \i | \- \ureal \Ra \i
| \infinity \i | + \i | - \i
\real \Ra → \sign \ureal \Ra
| \infinity
\ureal \Ra → \uinteger \Ra
| \untieger \Ra / \uinteger \Ra
| \decimal \Ra
\decimal 10 → \uinteger 10 \suffix
| . \(digit 10)^+ \suffix
| \(digit 10)^+ . \(digit 10)^* \suffix
\uinteger \Ra → \(digit \Ra)^+
\prefix \Ra → \(radix \Ra \exactness
| \exactness \(radix \Ra
\infinity → +inf.0 | -inf.0 | +nan.0

\suffix → \emptyset
| \(exponent marker) \sign \(digit 10)^+
\(exponent marker) → e | s | f | d | l
\sign → \emptyset | + | -
\exactness → \emptyset | #i | #e
\radix 2 → #b
\radix 8 → #o
\radix 10 → \emptyset | #d
\radix 16 → #x
\digit 2 → 0 | 1
\digit 8 → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7
\digit 10 → \digit 0
\digit 16 → \digit 10 | a | b | c | d | e | f

7.1.2. External representations

\Datum is what the \texttt{read} procedure (section 6.13.2) suc-
cessfully parses. Note that any string that parses as an
(expression) will also parse as a ⟨datum⟩.

⟨datum⟩ → ⟨simple datum⟩ | ⟨compound datum⟩
  | ⟨label⟩ = ⟨datum⟩ | ⟨label⟩ ≠

⟨simple datum⟩ → ⟨boolean⟩ | ⟨number⟩
  | ⟨character⟩ | ⟨string⟩ | ⟨symbol⟩ | ⟨bytevector⟩

⟨symbol⟩ → ⟨identifier⟩
⟨compound datum⟩ → ⟨list⟩ | ⟨vector⟩
⟨list⟩ → ⟨((datum)*)⟩ | ⟨(datum)+ . ⟨datum⟩⟩
  | ⟨abbreviation⟩
⟨abbreviation⟩ → ⟨abbrev prefix⟩ ⟨datum⟩
⟨abbrev prefix⟩ → \* | \* | \* | \*
⟨vector⟩ → #⟨(datum)*⟩
⟨label⟩ → #⟨(digit 10)*⟩

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.

⟨expression⟩ → ⟨identifier⟩
  | ⟨literal⟩
  | ⟨procedure call⟩
  | ⟨lambda expression⟩
  | ⟨conditional⟩
  | ⟨assignment⟩
  | ⟨derived expression⟩
  | ⟨macro use⟩
  | ⟨macro block⟩

⟨literal⟩ → ⟨ quoation ⟩ | ⟨ self-evaluating ⟩
⟨self-evaluating⟩ → ⟨ boolean ⟩ | ⟨ number ⟩
  | ⟨ character ⟩ | ⟨ string ⟩ | ⟨ bytevector ⟩
⟨ quoation ⟩ → ‘(datum) | ⟨ quote (datum) ⟩
⟨procedure call⟩ → ((operator) ⟨ operand ⟩)*
⟨operator⟩ → ⟨ expression ⟩
⟨operand⟩ → ⟨ expression ⟩

⟨lambda expression⟩ → (⟨lambda⟩ ⟨formals⟩ ⟨body⟩)
⟨formals⟩ → ⟨(identifier)*⟩ | ⟨ identifier ⟩
  | ⟨ (identifier)+ , ⟨ identifier ⟩ ⟩
⟨body⟩ → ⟨ definition ⟩* ⟨ sequence ⟩
⟨sequence⟩ → ⟨ command ⟩* ⟨ expression ⟩
⟨command⟩ → ⟨ expression ⟩

⟨conditional⟩ → ⟨if ⟨ test ⟩ ⟨ consequent ⟩ ⟨ alternate ⟩⟩
⟨test⟩ → ⟨ expression ⟩
⟨consequent⟩ → ⟨ expression ⟩
⟨alternate⟩ → ⟨ expression ⟩ | ⟨ empty ⟩

⟨assignment⟩ → ⟨ set! ⟨ identifier ⟩ ⟨ expression ⟩ ⟩

⟨derived expression⟩ →
  ⟨ cond ⟨ cond clause ⟩+ ⟩

| ⟨ cond ⟨ cond clause ⟩* ( else ⟨ sequence ⟩ ) ⟩
| ⟨ case ⟨ expression ⟩ ⟩
  ⟨ case clause ⟩+
| ⟨ case ⟨ expression ⟩ ⟩
  ⟨ case clause ⟩*
  ⟨ else ⟨ sequence ⟩ ⟩
| ⟨ case ⟨ expression ⟩ ⟩
  ⟨ case clause ⟩*
  ⟨ else ⟩=
  ⟨ recipient ⟩
| ⟨ and ⟨ test ⟩* ⟩
| ⟨ or ⟨ test ⟩* ⟩
| ⟨ when ⟨ expression ⟩ ⟨ test ⟩ ⟨ sequence ⟩ ⟩
| ⟨ unless ⟨ expression ⟩ ⟨ test ⟩ ⟨ sequence ⟩ ⟩
| ⟨ let ⟨ (binding spec)* ⟩ ⟨ body ⟩ ⟩
| ⟨ let ⟨ identifier ⟩ ⟨ (binding spec)* ⟩ ⟨ body ⟩ ⟩
| ⟨ let* ⟨ (binding spec)* ⟩ ⟨ body ⟩ ⟩
| ⟨ letrec ⟨ (binding spec)* ⟩ ⟨ body ⟩ ⟩
| ⟨ letrec* ⟨ (binding spec)* ⟩ ⟨ body ⟩ ⟩
| ⟨ let-values ⟨ (mv binding spec)* ⟩ ⟨ body ⟩ ⟩
| ⟨ let**-values ⟨ (mv binding spec)* ⟩ ⟨ body ⟩ ⟩
| ⟨ begin ⟨ sequence ⟩ ⟩
| ⟨ do ⟨ (iteration spec)* ⟩
  ⟨ (test) ⟨ do result ⟩ ⟩
  ⟨ command ⟩+
| ⟨ delay ⟨ expression ⟩ ⟩
| ⟨ lazy ⟨ expression ⟩ ⟩
| ⟨ parameterize ⟨ (expression) ⟨ expression ⟩* ⟨ body ⟩ ⟩
| ⟨ guard ⟨ identifier ⟩ ⟨ cond clause ⟩* ⟨ body ⟩ ⟩
  ⟨ quasiquotation ⟩
| ⟨ case-lambda ⟨ case-lambda clause ⟩* ⟩

| ⟨ cond clause ⟩ → ⟨ (test) ⟨ sequence ⟩ ⟩
| ⟨ (test) ⟩
| ⟨ (test) ⇒ ⟨ recipient ⟩ ⟩
⟨recipient⟩ → ⟨ expression ⟩
⟨case clause⟩ → ⟨ ((datum)*) ⟨ sequence ⟩ ⟩
  | ⟨ ((datum)*) ⇒ ⟨ recipient ⟩ ⟩
⟨binding spec⟩ → ⟨ (identifier) ⟨ expression ⟩ ⟩
⟨mv binding spec⟩ → ⟨ (formals) ⟨ expression ⟩ ⟩
⟨iteration spec⟩ → ⟨ (identifier) ⟨ init ⟩ ⟨ step ⟩ ⟩
  | ⟨ (identifier) ⟨ init ⟩ ⟩
⟨case-lambda clause⟩ → ⟨ (formals) ⟨ body ⟩ ⟩
⟨init⟩ → ⟨ expression ⟩
⟨step⟩ → ⟨ expression ⟩
⟨do result⟩ → ⟨ sequence ⟩ | ⟨ empty ⟩

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

⟨macro block⟩ →
  ⟨ let-syntax ⟨ (syntax spec)* ⟩ ⟨ body ⟩ ⟩
  | ⟨ letrec-syntax ⟨ (syntax spec)* ⟩ ⟨ body ⟩ ⟩
⟨syntax spec⟩ → ⟨ (keyword) ⟨ transformer spec ⟩ ⟩
7.1.4. Quasiquotations

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

\[
\text{(quasiquote)} \rightarrow \text{(quasiquote 1)}
\]
\[
\text{(qq template 0)} \rightarrow \text{(expression)}
\]
\[
\text{(quasiquote \( D \))} \rightarrow \text{\&(qq template \( D \))}
\]
\[
\text{(qq template \( D \))} \rightarrow \text{(simple datum)}
\]
\[
\text{(list qq template \( D \))} \rightarrow \text{(qq template or splice \( D \))}^+ \cdot \text{(qq template \( D \))}
\]
\[
\text{(vector qq template \( D \))} \rightarrow \#\left(\text{(qq template or splice \( D \))}^+ \cdot \text{(qq template \( D \))}\right)
\]
\[
\text{(unquotation \( D \))} \rightarrow \cdot \text{(qq template \( D - 1 \))}
\]
\[
\text{(qq template or splice \( D \))} \rightarrow \text{(qq template \( D \))}
\]
\[
\text{(splicing unquotation \( D \))} \rightarrow \cdot \text{(qq template \( D - 1 \))}
\]
\[
\text{(unquote-splicing (qq template \( D - 1 \))}
\]

In \( \text{(quasiquote)} \)'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

\[
\text{(transformer spec)} \rightarrow
\]
\[
\text{(syntax-rules (identifier) \( \text{(syntax rule)} \))}
\]
\[
\text{(syntax-rules (identifier) \( \text{(identifier)} \))}
\]
\[
\text{(syntax rule)} \rightarrow \text{(pattern) \( \text{(template)} \)}
\]
\[
\text{(pattern)} \rightarrow \text{(pattern identifier)}
\]
\[
\text{| (underscore) \( \text{(pattern)} \)}
\]
\[
\text{| (pattern) \( \text{(pattern)} \)}
\]
\[
\text{| (pattern) \( \text{(pattern)} \)}
\]
\[
\text{| (pattern) \( \text{(pattern)} \)}
\]
\[
\#\left(\text{(pattern)}\right)
\]
\[
\text{\#\left(\text{(pattern)}\right) \( \text{(pattern)} \)}
\]
\[
\text{\#\left(\text{(pattern)}\right) \( \text{(pattern)} \)}
\]
\[
\text{\#\left(\text{(pattern)}\right) \( \text{(pattern)} \)}
\]
\[
\text{\#\left(\text{(pattern)}\right) \( \text{(pattern)} \)}
\]
\[
\text{\#\left(\text{(pattern)}\right) \( \text{(pattern)} \)}
\]
\[
\text{\#\left(\text{(pattern)}\right) \( \text{(pattern)} \)}
\]
\[
\text{\#\left(\text{(pattern)}\right) \( \text{(pattern)} \)}
\]
\[
\text{(pattern datum)} \rightarrow \text{\&(string)}
\]
\[
\text{| (character) \( \text{(boolean)} \)}
\]
\[
\text{| (boolean) \( \text{(number)} \)}
\]
\[
\text{| (number) \( \text{(template)} \)}
\]
\[
\text{| \( \text{(pattern identifier)} \)}
\]

7.1.6. Programs and definitions

\[
\text{\&(program)} \rightarrow \text{(command or definition)\( ^+ \)}
\]
\[
\text{\&(command or definition)} \rightarrow \text{(command)\( ^+ \)}
\]
\[
\text{(definition) \rightarrow \text{(define (identifier) \( \text{(expression)} \))\( ^+ \)}
\]
\[
\text{| (define (identifier) \( \text{(def forms)} \) \( \text{(body)} \))\( ^+ \)}
\]
\[
\text{| \( \text{(syntax definition)} \)\( ^+ \)}
\]
\[
\text{| \( \text{(define-values \( \text{(def forms)} \) \( \text{(body)} \))}\( ^+ \)}
\]
\[
\text{| \( \text{(define-record-type (identifier)} \( \text{(constructor)} \( \text{(identifier)} \) \( \text{(field spec)} \))\( ^+ \)}
\]
\[
\text{| \( \text{(begin (definition)} \( ^+ \))\( ^+ \)}
\]
\[
\text{\&(def forms)} \rightarrow \text{(identifier)\( ^+ \)}
\]
\[
\text{| \( \text{(identifier)} \) \( \text{(identifier)} \)\( ^+ \)}
\]
\[
\text{| \( \text{(constructor)} \) \( \text{(identifier)} \) \( \text{(field name)} \))\( ^+ \)}
\]
\[
\text{| \( \text{(field spec)} \) \( \text{(field name)} \) \( \text{(accessor)} \))\( ^+ \)}
\]
\[
\text{| \( \text{(field name)} \) \( \text{(accessor)} \) \( \text{(mutator)} \))\( ^+ \)}
\]
\[
\text{| \( \text{(syntax definition)} \)\( ^+ \)}
\]
\[
\text{| \( \text{(define-syntax (keyword)} \) \( \text{(transformer spec)} \))\( ^+ \)}
\]

7.1.7. Libraries

\[
\text{\&(library)} \rightarrow
\]
\[
\text{\&(define-library (library name) \( \text{(library declaration)} \))\( ^+ \)}
\]
\[
\text{\&(library name)} \rightarrow \text{(library name part)\( ^+ \)}
\]
\[
\text{\&(library name part)} \rightarrow \text{(identifier) | \( \text{(integer 10)} \))\( ^+ \)}
\]
\[
\text{\&(library declaration)} \rightarrow \text{(export \( \text{(export spec)} \))\( ^+ \)}
\]
\[
\text{| \( \text{(import \( \text{(import set)} \))}\( ^+ \)}
\]
\[
\text{| \( \text{(begin (command or definition)} \))\( ^+ \)}
\]
\[
\text{| \( \text{(include \( \text{(string)} \))}\( ^+ \)}
\]
\[
\text{| \( \text{(include-ci \( \text{(string)} \))}\( ^+ \)}
\]
\[
\text{| \( \text{(cond-expand \( \text{(cond-expand clause)} \))}\( ^+ \)}
\]
\[
\text{| \( \text{(cond-expand \( \text{(cond-expand clause)} \))}\( ^+ \)}
\]
\[
\text{| \( \text{(else \( \text{(library declaration)} \))}\( ^+ \)}
\]
\[
\text{| \( \text{(export spec)} \) \( \text{(identifier)} \))\( ^+ \)}
\]
\[
\text{| \( \text{(rename (identifier) \( \text{(identifier)} \))}\( ^+ \)}
\]
\[
\text{| \( \text{(import set)} \) \( \text{(library name)} \))\( ^+ \)}
\]


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:

(\ldots) \quad \text{sequence formation}

s \downarrow k \quad \text{\textit{k}th member of the sequence \textit{s} (1-based)}

\#s \quad \text{length of sequence \textit{s}}

s \uparrow t \quad \text{concatenation of sequences \textit{s} and \textit{t}}

s \downarrow k \quad \text{drop the first \textit{k} members of sequence \textit{s}}

t \rightarrow a, b \quad \text{McCarthy conditional “if \textit{t} then \textit{a} else \textit{b}”}

\rho[x/i] \quad \text{substitution “\textit{\rho} with \textit{x} for \textit{i}”}

x \in D \quad \text{injection of \textit{x} into domain \textit{D}}

x \mid D \quad \text{projection of \textit{x} to domain \textit{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 \textit{permute} and \textit{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 \textit{new} is implementation-dependent, but it must obey the following axiom: if \textit{new} \\sigma \in L, then \sigma (\textit{new} \\sigma \mid L) \downarrow 2 = false.

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

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

\[ \mathcal{E}[((\lambda \alpha \ (I^* \ P^*) \ (\text{undefined}) \ \ldots))] \]

where \( I^* \) is the sequence of values defined in \textit{P}, \( P^* \) is the sequence of values obtained by replacing every definition in \textit{P} by an assignment, \( \langle \text{undefined} \rangle \) is an expression that evaluates to \textit{undefined}, and \( \mathcal{E} \) is the semantic function that assigns meaning to expressions.

7.2.1. Abstract syntax

\begin{align*}
\text{Exp} & \rightarrow K | I | (E_0 \ E^*) \\
& \mid (\lambda \alpha \ (I^* \ G^* \ E_0)) \\
& \mid (\lambda \alpha \ (I^* \ . \ I) \ G^* \ E_0) \\
& \mid (\lambda \alpha \ I \ G^* \ E_0) \\
& \mid (\text{if} \ E_0 \ E_1 \ E_2) \ | \ (\text{if} \ E_0 \ E_1) \\
& \mid (\text{set!} \ I \ E) \\
\end{align*}

7.2.2. Domain equations

\begin{align*}
\alpha & \in L \quad \text{locations} \\
\nu & \in \mathbb{N} \quad \text{natural numbers} \\
T & = \{false, true\} \quad \text{booleans} \\
Q & \quad \text{symbols} \\
H & \quad \text{characters} \\
R & \quad \text{numbers} \\
E_p & = L \times L \times T \quad \text{pairs} \\
E_v & = L^* \times T \quad \text{vectors} \\
E_s & = L^* \times T \quad \text{strings} \\
M & = \{false, true, null, undefined, unspecified\} \quad \text{miscellaneous} \\
\phi & \in F = L \times (E^* \rightarrow K \rightarrow C) \quad \text{procedure values} \\
\epsilon & \in E = Q + H + R + E_p + E_v + E_s + M + F \quad \text{expressed values} \\
\sigma & \in S = L \rightarrow (E \times T) \quad \text{stores} \\
\rho & \in U = \text{Id} \rightarrow L \quad \text{environments} \\
\theta & \in C = S \rightarrow A \quad \text{command continuations} \\
\kappa & \in K = E^* \rightarrow C \quad \text{expression continuations} \\
A & \quad \text{answers} \\
X & \quad \text{errors} \\
\end{align*}

7.2.3. Semantic functions

\begin{align*}
\mathcal{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 \mathcal{K} deliberately omitted.

\[ \mathcal{E}[K] = \lambda \rho \cdot \text{send}(\mathcal{K}[\kappa]) \]
$$\mathcal{E}[\Pi] = \lambda \rho \cdot \text{hold (lookup } \rho \Pi)$$

$$(\text{single}(\lambda \epsilon. \epsilon = \text{undefined} \rightarrow \text{wrong "undefined variable", send } \epsilon \kappa))$$

$$\mathcal{E}[\langle E_0 \rangle \ E^*] = \lambda \rho \cdot \mathcal{E}^*(\text{permute}(\langle E_0 \rangle \ \| \ E^*))$$

$$\rho (\lambda^*. \ (\langle \lambda^*. \ \text{apply} (\epsilon^* \downarrow 1) (\epsilon^* \uparrow 1) \kappa)$$

$$(\text{unpermute } \epsilon^*))$$

$$\mathcal{E}[\lambda \text{ (I* } \Gamma^* \ E_0)] = \lambda \rho \cdot \lambda \gamma. \new \sigma \in L \rightarrow \text{send (new } \sigma \ | L, \ \lambda^* \kappa'. \ \# \epsilon^* = \# I^* \rightarrow \text{tievals}(\lambda^* \cdot \ C[\Gamma^*](\rho (\mathcal{E}[E_0] \ \| \rho \kappa'))$$

$$(\text{extends } \rho \ I^* \ \alpha^*))$$

$$\epsilon^*,$$

$$\text{wrong "wrong number of arguments"} \text{ in } E)$$

$$\kappa$$

$$(\text{update (new } \sigma \ | L) \text{ unspecified } \sigma), \ \text{wrong "out of memory" } \sigma$$

$$\mathcal{E}[\lambda \text{ (I* } \ I) \ \Gamma^* \ E_0] = \lambda \rho \cdot \lambda \gamma. \new \sigma \in L \rightarrow \text{send (new } \sigma \ | L, \ \lambda^* \kappa'. \ \# \epsilon^* \geq \# I^* \rightarrow \text{tievalsrest}$$

$$(\lambda^* \cdot (\lambda^* \cdot \ C[\Gamma^*](\rho (\mathcal{E}[E_0] \ | \rho \kappa')))$$

$$(\text{extends } \rho \ (I^* \ | \ I) \ \alpha^*))$$

$$\epsilon^*$$

$$(\# I^*), \ \text{wrong "too few arguments"}) \text{ in } E)$$

$$\kappa$$

$$(\text{update (new } \sigma | L) \text{ unspecified } \sigma), \ \text{wrong "out of memory" } \sigma$$

$$\mathcal{E}[\lambda \text{ (I* } \ I) \ \Gamma^* \ E_0] = \mathcal{E}[\lambda \text{ (I* } \ E_0)]$$

$$\mathcal{E}[\text{ if } E_0, E_1, E_2] = \lambda \rho \cdot \mathcal{E}[E_0] \ \rho (\text{single}(\lambda \epsilon. \ \text{truish } \epsilon \rightarrow \mathcal{E}[E_1] \ \rho \kappa, \ \mathcal{E}[E_2] \ \rho \kappa))$$

$$\mathcal{E}[\text{ if } E_0, E_1] = \lambda \rho \cdot \mathcal{E}[E_0] \ \rho (\text{single}(\lambda \epsilon. \ \text{truish } \epsilon \rightarrow \mathcal{E}[E_1] \ \rho \kappa, \ \text{send unspecified } \kappa))$$

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

$$\mathcal{E}[\text{ set } I \ E] = \lambda \rho \cdot \mathcal{E}[E] \ \rho (\text{single}(\lambda \epsilon. \ \text{assign (lookup } \rho \ I)$$

$$\epsilon \text{ (send unspecified } \kappa))$$

$$\mathcal{E}^*[\epsilon] = \lambda \rho \cdot \kappa(\epsilon)$$

$$\mathcal{E}[E_0, E^*] = \lambda \rho \cdot \mathcal{E}[E_0] \ \rho (\text{single}(\lambda \kappa_0. \ E^*[E^*] \ \rho (\lambda \epsilon^* \cdot \kappa (\langle \epsilon \rangle \ \| \ \epsilon^*)))$$

$$\mathcal{C}[\epsilon] = \lambda \rho \cdot \theta. \theta$$

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

7. Formal syntax and semantics

7.2.4. Auxiliary functions

$$\text{lookup : } U \rightarrow \text{Ide } \rightarrow L$$

$$\text{lookup } = \lambda \rho \cdot \rho I$$

$$\text{extends : } U \rightarrow \text{Ide}^* \rightarrow L^* \rightarrow U$$

$$\text{extends } = \lambda \rho \cdot \alpha^* \cdot \# I^* = 0 \rightarrow \rho,$$

$$\text{extends } (\rho (\alpha^* \downarrow 1) (I^* \uparrow 1)) (I^* \uparrow 1) (\alpha^* \uparrow 1)$$

$$\text{wrong : } X \rightarrow C \quad \text{[implementation-dependent]}$$

$$\text{send : } E \rightarrow K \rightarrow C$$

$$\text{send } = \lambda \epsilon \cdot \kappa(\epsilon)$$

$$\text{single : } (E \rightarrow C) \rightarrow K$$

$$\text{single } = \lambda \psi^* \cdot \# \epsilon^* = 1 \rightarrow \psi(\epsilon^* \downarrow 1),$$

$$\text{wrong "wrong number of return values"}$$

$$\text{new : } S \rightarrow (L + \{ \text{error} \}) \quad \text{[implementation-dependent]}$$

$$\text{hold : } L \rightarrow K \rightarrow C$$

$$\text{hold } = \lambda \kappa \sigma \cdot \text{send (new } \sigma \ | L \ \kappa \sigma)$$

$$\text{assign : } L \rightarrow E \rightarrow C$$

$$\text{assign } = \lambda \alpha \theta \sigma \cdot \theta (\text{update } \alpha \sigma)$$

$$\text{update : } L \rightarrow E \rightarrow S \rightarrow S$$

$$\text{update } = \lambda \alpha \sigma \cdot \sigma(\epsilon, \text{true})/\alpha$$

$$\text{tievals : } (L^* \rightarrow C) \rightarrow E^* \rightarrow C$$

$$\text{tievals } = \lambda \psi^* \cdot \# \epsilon^* = 0 \rightarrow \psi(\epsilon^* \downarrow 1),$$

$$\text{new } \sigma \in L \rightarrow \text{tievals}(\lambda^* \cdot \psi(\langle \new \sigma \ | L \ \| \ \alpha^*))$$

$$(\epsilon^* \uparrow 1)$$

$$(\text{update (new } \sigma | L) (\epsilon^* \downarrow 1))$$

$$\text{tievalsrest : } (L^* \rightarrow C) \rightarrow E^* \rightarrow N \rightarrow C$$

$$\text{tievalsrest } = \lambda \psi^* \cdot \text{list (dropfirst } \epsilon^*)$$

$$(\text{single}(\lambda \epsilon \cdot \text{tievals } \psi ((\text{takefirst } \epsilon^*) \ | \ \langle \epsilon \rangle)))$$

$$\text{dropfirst } = \lambda \nu. \ n = 0 \rightarrow l, \ \text{dropfirst (l } \downarrow 1)(n - 1)$$

$$\text{takefirst } = \lambda \nu. \ n = 0 \rightarrow l, (l \uparrow 1)(n - 1)$$

$$\text{truish : } E \rightarrow T$$

$$\text{truish } = \lambda \epsilon \cdot \epsilon = \text{false } \rightarrow \text{false, true}$$

$$\text{permute : Exp^* } \rightarrow \text{Exp^* } \quad \text{[implementation-dependent]}$$

$$\text{unpermute : Exp^* } \rightarrow \text{Exp^* } \quad \text{[inverse of permute]}$$

$$\text{applicate : } E \rightarrow E^* \rightarrow K \rightarrow C$$

$$\text{applicate } = \lambda \epsilon^* \cdot \epsilon \in \ F \rightarrow (\epsilon | F \uparrow 2) \epsilon^* \kappa,$$

$$\text{wrong "bad procedure"}$$

$$\text{oneary : } (E \rightarrow K \rightarrow C) \rightarrow (E^* \rightarrow K \rightarrow C)$$

$$\text{oneary } = \lambda \epsilon^* \cdot \# \epsilon^* = 1 \rightarrow \zeta(\epsilon^* \downarrow 1) \kappa,$$

$$\text{wrong "wrong number of arguments"}$$

$$\text{twoary : } (E \rightarrow E \rightarrow K \rightarrow C) \rightarrow (E^* \rightarrow K \rightarrow C)$$

$$\text{twoary } = \lambda \epsilon^* \cdot \# \epsilon^* = 2 \rightarrow \zeta(\epsilon^* \downarrow 1)(\epsilon^* \downarrow 2) \kappa,$$

$$\text{wrong "wrong number of arguments"}$$
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!), except for quasiquote.

```scheme
(define-syntax cond
  (syntax-rules (else =>)
    ((cond (else result1 result2 ...) . . .)
     (else (result1 result2))
     (else (result1 result2) . . .)
     (else (result1 result2) . . .))
    ((cond (test => result))
     (let ((temp test))
       (if temp
         (result temp)
         (cond clause1 clause2 ...)))
     (let ((temp test))
       (if temp
         (result temp)
         (cond clause1 clause2 ...))))
```
((cond (test) test)
  (cond (test) clause1 clause2 ...)
  (let ((temp test))
    (if temp
      temp
      (cond clause1 clause2 ...)))
  (cond (test result1 result2 ...) 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 ...) clause1 clause2 ...)
     (begin result1 result2 ...))
    ((case key ((atoms ...) result1 result2 ...) clause1 clause2 ...)
     (if (memv key '(atoms ...))
       (begin result1 result2 ...
            clause clauses ...))
     (case key ((atoms ...) => result))
     (if (memv key '(atoms ...))
       (result key)
       (case key ((atoms ...) result1 result2 ...
                  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 (or test2 ...))))
    (define-syntax let
      (syntax-rules ()
        ((let ((name val) ...) body1 body2 ...)
         (lambda (name ...) body1 body2 ...)
         val ...))
        ((let tag ((name val) ...) body1 body2 ...)
         (letrec ((tag (lambda (name ...) body1 body2 ...))
           tag)
           val ...))))

(define-syntax let*
  (syntax-rules ()
    ((let* () body1 body2 ...)
      (let () body1 body2 ...))
    ((let* ((name1 val1) (name2 val2) ...) body1 body2 ...)
      (let ((name1 <undefined>) ...
            (name2 val2) ...
            body1 body2 ...)))

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

(define-syntax letrec
  (syntax-rules ()
    ((letrec ((var1 init1) ...) body ...) (letrec "generate_temp_names"
      (var1 ...) () ((var1 init1) ...) body ...)
    ((letrec "generate_temp_names"
      () (temp1 ...) ((var1 init1) ...) body ...)
    (let ((var1 <undefined>) ...) (let ((temp init1) ...) (set! var1 temp1 ...
      body ...)))
    (define-syntax letrec "generate_temp_names"
      (x y ...) (temp ...) ((var1 init1) ...) body ...)
    (letrec "generate_temp_names"
      (var1 init1) ...) body ...)
    (letrec "generate_temp_names"
      (var1 init1) ...) body ...))
(define-syntax letrec*
  (syntax-rules ()
    ((letrec* ((var1 init1) ...) body1 body2 ...)
      (let ((var1 <undefined>) ...)
        (set! var1 init1)
        ...
        (let () body1 body2 ...))))))

(define-syntax let-values
  (syntax-rules ()
    ((let-values (binding ...) body0 body1 ...)
      (let-values "bind"
        (binding ...) () (begin body0 body1 ...)))
    ((let-values "bind" () tmps body)
      (let tmps body))
    ((let-values "bind" ((b0 e0) binding ...) tmps body)
      (let-values "mktmp" b0 e0 ()
        (binding ...) tmps body))
    ((let-values "mktmp" () e0 args
      bindings tmps body)
      (call-with-values
        (lambda () e0)
        (lambda args
          (let-values "bind" bindings tmps body))))
    ((let-values "mktmp" (a . b) e0 (arg ...)
      bindings tmps body)
      (let-values "mktmp" b e0 (arg ... x)
        bindings (tmp ... (a x) body))
    ((let-values "mktmp" a e0 (arg ...)
      bindings (tmp ... body)
      (call-with-values
        (lambda () e0)
        (lambda (arg . . x)
          (let-values "bind" bindings (tmp ... (a x) body))))
    ((define-syntax define-values
      (syntax-rules ()
        ((define-values () expr)
          (define dummy
            (call-with-values
              (lambda () expr)
              (lambda args #f)))))
        ((define-values (var) expr)
          (define var expr))
        ((define-values (var0 var1 ... varn) expr)
          (begin
            (define var0
              (call-with-values
                (lambda () expr)
                (lambda args list))))
            (define var1
              (let ((v (cadr var0)))
                (set-cdr! var0 (cddr var0))
                v) ...)
            (define varn
              (let ((v (cadr var0)))
                (set! var0 (car var0))
                v)))))
        ((define-values var expr)
          (define var
            (call-with-values
              (lambda () expr)
              (lambda args list))))))

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

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

(define-syntax begin
  (syntax-rules ()
    ((begin exp)
      (begin exp))
    ((begin exp1 exp2 ...)
      (begin exp1 exp2 ...))))
The following definition of `do` uses a trick to expand the variable clauses. As with `letrec` above, an auxiliary macro would also work. The expression `(if #f #f)` is used to obtain an unspecific value.

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

Here is a possible implementation of `delay`, `force` and `lazy`. We define the expression

```
lazy (expression)
```

to have the same meaning as the procedure call

```
(make-promise #f (lambda () (expression)))
```

as follows

```
(define-syntax lazy
  (syntax-rules ()
    ((lazy expression)
      (make-promise #f (lambda () expression)))))
```

and we define the expression

```
delay (expression)
```

to have the same meaning as:

```
lazy (make-promise #t (expression))
```

as follows

```
(define-syntax delay
  (syntax-rules ()
    ((delay expression)
      (lazy (make-promise #t expression))))
```

where `make-promise` is defined as follows:

```
(define make-promise
  (lambda (done? proc)
    (list (cons done? proc)))))
```

Finally, we define `force` to call the procedure expressions in promises iteratively using a trampoline technique following [13] until a non-lazy result (i.e. a value created by `delay` instead of `lazy`) is returned, as follows:

```
(define (force promise)
  (if (promise-done? promise)
      (promise-value promise)
      (let ((promise* ((promise-value promise)))
        (unless (promise-done? promise)
          (promise-update! promise* promise)
          (force promise)))))
```

with the following promise accessors:

```
(define promise-done?
  (lambda (x) (car (car x))))

(define promise-value
  (lambda (x) (cdr (car x))))

(define promise-update!
  (lambda (new old)
    (set-car! (car old) (promise-done? new))
    (set-cdr! (car old) (promise-value new))
    (set-car! new (car old))))
```

The following implementation of `make-parameter` and `parameterize` is suitable for an implementation with no threads. Parameter objects are implemented here as procedures, using two arbitrary unique objects `<param-set!>` and `<param-convert>`:

```
(define (make-parameter init . o)
  (let* ((converter
              (if (pair? o) (car o) (lambda (x) x)))
         (value (converter init)))
    (lambda args
      (cond
        ((null? args)
         value)
        ((eq? (car args) <param-set!>)
         (set! value (cadr args)))
        ((eq? (car args) <param-convert>)
         converter)
        (else
         (error "bad parameter syntax")))))
```

Then `parameterize` uses `dynamic-wind` to dynamically rebind the associated value:

```
(define-syntax parameterize
  (syntax-rules ()
    ((parameterize ("step")
                   (param value p old new) ...)
      ()
      body)
      (let ((p param) ...)
        (let ((old (p)) ...)
          (new ((p <param-convert>) value) ...))
```

((define-syntax guard-aux
  (syntax-rules (else =>)
    ((guard-aux reraise (else result1 result2 ...))
     (begin result1 result2 ...))
    ((guard-aux reraise (test => result))
     (begin result test result2 ...))
    ((guard-aux reraise (test => result) clause1 clause2 ...)
     (begin result test result2 ... clause1 clause2 ...))
    ((guard-aux reraise clause1 clause2 ...) reraise)
    ((guard-aux reraise clause1 clause2 ...) (begin result test result2 ... clause1 clause2 ...))
    ((guard-aux reraise clause1 clause2 ...) (begin result test result2 ... clause1 clause2 ...)))

(define-syntax case-lambda
  (syntax-rules ()
    ((case-lambda (params body0 body1 ...) ...) ...
     (let ((len (length args)))
       (let-syntax ((cl (syntax-rules ::: ()
                     ((cl)
                      (error "no matching clause"))
                     ((cl ((p :::) . body) . rest)
                      (if (= len (length `(p :::)))
                       (apply (lambda (p :::)
                                . body)
                              args)
                       (cl . rest))))
                     ((cl ((p ::: . tail) . body) . rest)
                      (if (>= len (length `(p :::)))
                       (apply (lambda (p ::: . tail)
                                . body)
                              args)
                       (cl . rest))))
       (cl (params body0 body1 ...) ...))))))

The following implementation of guard depends on an auxiliary macro, here called guard-aux.

(define-syntax guard
  (syntax-rules ()
    ((guard (var clause ...) e1 e2 ...) (call/cc
      (lambda (guard-k)
        (with-exception-handler
          (lambda (condition)
            (call/cc
              (lambda (handler-k)
                (guard-k
                 (lambda ()
                  (let ((var condition))
                    (guard-aux
                     (handler-k
                     (lambda ()
                      (raise condition))
                     clause ...))))))))
      (lambda ()
        (call-with-values
         (lambda () e1 e2 ...) (lambda args
          (guard-k
           (lambda ()
            (apply values args)))))))))))

(define-syntax guard-aux
  (syntax-rules (else =>)
    ((guard-aux reraise (else result1 result2 ...))
     (begin result1 result2 ...))
    ((guard-aux reraise (test => result))
     (begin result test result2 ...))
    ((guard-aux reraise (test => result) clause1 clause2 ...)
     (begin result test result2 ... clause1 clause2 ...))
    ((guard-aux reraise clause1 clause2 ...) reraise)
    ((guard-aux reraise clause1 clause2 ...) (begin result test result2 ... clause1 clause2 ...))
    ((guard-aux reraise clause1 clause2 ...) (begin result test result2 ... clause1 clause2 ...))
    ((guard-aux reraise clause1 clause2 ...) (begin result test result2 ... clause1 clause2 ...))
    ((guard-aux reraise clause1 clause2 ...) (begin result test result2 ... clause1 clause2 ...)))
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>Scheme</th>
<th>Procedure</th>
</tr>
</thead>
</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 asin atan cos exp finite? log nan? sin sqrt 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`

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-ci>?`
- `string-downcase`
- `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=?`
- `string-ni=>?`
- `string-ni>?`
Time
The (scheme time) library provides access to the system time.

current-jiffy  current-second
jiffies-per-second

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\textsuperscript{5}RS

This section enumerates the differences between this report and the “Revised\textsuperscript{5} report” \[2\].

The list is incomplete and subject to change while this report is in draft status.

- Various minor ambiguities and unclaritys in R\textsuperscript{5}RS 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 \texttt{raise}, \texttt{raise-continuable} or \texttt{error}, and can be handled with \texttt{with-exception-handler} and the \texttt{guard} syntax. Any object can specify an error condition; the implementation-defined conditions signalled by \texttt{error} have accessor functions to retrieve the arguments passed to \texttt{error}.
- New disjoint types supporting access to multiple fields can be generated with SRFI 9’s \texttt{define-record-type}.
- Parameter objects can be created with \texttt{make-parameter}, and dynamically rebound with \texttt{parameterize}.
- \texttt{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 \texttt{read-line} is provided to make line-oriented textual input simpler.
- \texttt{Ports} can now be designated as \texttt{textual} or \texttt{binary} ports, with new procedures for reading and writing binary data. The new predicate \texttt{port-open?} returns whether a port is open or closed.
- \texttt{String ports} have been added as a way to read and write characters to and from strings, and \texttt{bytevector ports} to read and write bytes to and from bytevectors.
- The procedures \texttt{current-input-port} and \texttt{current-output-port} are now parameter objects, as is the newly introduced \texttt{current-error-port}.
- The \texttt{syntax-rules} construct now recognizes \_ (underscore) as a wildcard, allows the ellipsis symbol to be specified explicitly instead of the default \ldots, allows template escapes with an ellipsis-prefixed list, and allows tail patterns to follow an ellipsis pattern.
- The \texttt{syntax-error} syntax has been added as a way to signal immediate and more informative errors when a macro is expanded.
- Internal \texttt{define-syntax} definitions are now allowed wherever internal \texttt{defines} are.
- The \texttt{letrec*} binding construct has been added, and internal \texttt{define} is specified in terms of it.
- Support for capturing multiple values has been enhanced with \texttt{define-values}, \texttt{let-values}, and \texttt{let*-values}. Programs are now explicitly permitted to pass zero or more than one value to continuations which discard them.
- The \texttt{case} conditional now supports a $\Rightarrow$ syntax analogous to \texttt{cond}.
- To support dispatching on the number of arguments passed to a procedure, \texttt{case-lambda} has been added in its own library.
- The convenience conditionals \texttt{when} and \texttt{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 $+\infty.0$, $-\infty.0$, $+\text{nan}.0$, and $-0.0$ respectively.
- The procedures \texttt{map} and \texttt{for-each} are now required to terminate on the shortest list when inputs have different length.
- The procedures \texttt{member} and \texttt{assoc} now take an optional third argument specifying the equality predicate to be used.
- The procedures \texttt{exact-integer?} and \texttt{exact-integer-sqrt} have been added.
- The procedures \texttt{make-list}, \texttt{list-copy}, \texttt{list-set!}, \texttt{string-map}, \texttt{string-for-each}, \texttt{string->vector}, \texttt{vector-copy}, \texttt{vector-map}, \texttt{vector-for-each}, and \texttt{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 “Revised<sup>6</sup> report” [2].

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 R²RS, 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 R³RS). Non-Unicode characters are permitted.
- The full numeric tower is optional as in R³RS, 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 R³RS 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 R³RS and the R⁶RS 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 R³RS-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 #u8 for compatibility with SRFI 4, rather than the R⁸RS #\u8 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.

```
(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. It produces a function that takes a system state and produces a new system state.

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

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

```
(define generate-vector
  (lambda (size proc)
    (lambda vectors
      (elementwise proc vectors
                   (vector-length vectors)
                   (lambda (i)
                     (apply proc
                            (map (lambda (v) (vector-ref v i))
                                 vectors)))))))
```
(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 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

\[
\frac{dv_C}{dt} = -i_L - \frac{v_C}{R}
\]

\[
L \frac{di_L}{dt} = v_C
\]

which models a damped oscillator.

(define damped-oscillator
  (lambda (R L C)
    (lambda (state)
      (let ((Vc (vector-ref state 0))
            (I1 (vector-ref state 1)))
        (vector (- 0 (+ (/ Vc (* R C)) (/ I1 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.
... 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
caaar 38
caaadr 38
caar 38
caadar 38
cadder 38
caddr 38
caddr 38
caddr 38
caddr 38
caddhar 38
caddr 38
cadr 38
call by need 17
call 12
call with current-continuation 47; 11, 49, 62
call-with-current-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
cddar 38
cddadr 38
cdddr 38
cddr 38
cdar 38
cddaar 38
cddadr 38
cddar 38
cdddr 38
cddr 38
cdr 38
celling 35
celling-quotient 34
celling-remainder 34
celling/ 34
centered-quotient 34
centered-remainder 34
centered/ 34
char->integer 42
char-alphabetic? 41
char-ci<=? 41
char-ci<? 41
char-ci=? 41
char-ci>=? 41
char-ci>? 41
char-downcase 42
char-foldcase 42
char-lower-case? 41
char-numeric? 41
char-ready? 53
char-upcase 42
char-upper-case? 41
char-whitespace? 41
char<=? 41
char<? 41
char=? 41
char>=? 41
char>? 41
char? 41; 9
close-input-port 52
close-output-port 52
close-port 52
comma 19
command-line 55
comment 8; 56
complex? 32; 30, 33
cond 13; 22, 62
cond-expand 27
cond-expand 27
cons 38
constant 10
continuation 48
cos 35
current exception handler 49
current-error-port 51
current-input-port 51
current-jiffy 55
current-output-port 51
current-second 55
#d 32
define 24; 20
define-syntax 25
define-values 24; 64
definition 23
delay 17
delete-file 55
denominator 35
digit-value 41
display 54
do 17; 64
dotted pair 37
dynamic-wind 49; 48
#e 32; 57
eager 18
else 14; 27
empty list 37; 9, 38
environment 50; 55, 10
environment variables 55
eof-object? 53
eq? 29
equal? 30
equivalence predicate 28
eqv? 28; 10, 61
error 5; 50
error-object-irritants 50
error-object-message 50
error-object? 50
escape procedure 47
escape sequence 42
euclidean-quotient 34
euclidean-remainder 34
euclidean/ 34
eval 50; 11
even? 33
exact 28
exact->inexact 36
exact-integer-sqrt 35
exact-integer? 33
exact? 33
exactness 30
except 26
exception handler 49
exit 55
exp 35
export 26
expt 36
#f 37
false 9; 37
fields 25
file-exists? 55
finite? 33
floor 35
floor-quotient 34
floor-remainder 34
floor/ 34
flush-output-port 54
fold-case@##fold-case 8
for-each 47
force 17
gcd 34
get-environment-variable 55
get-environment-variables 55
get-output-bytevector 52
get-output-string 52
guard 19: 24
hygienic 20
#i 32; 57
identifier 7; 9, 56
if 13; 60
imag-part 36
immutable 10
implementation extension 31
implementation restriction 6; 31
import 26
improper list 38
include 27
include-ci 27
inexact 28
inexact->exact 36; 30
inexact? 33
initial environment 28
input-port? 51
integer->char 42
integer? 32; 30
interaction-environment 51
internal definition 24
irritants 50
jiffies 55
jiffies-per-second 55

keyword 20
lambda 12; 24, 60
lazy 17
lazy evaluation 17
lcm 34
length 39; 31
let 15; 17, 22, 24, 63
let* 15; 24, 63
let*-values 16; 63
let-syntax 20; 24
let-values 16; 24, 63
let-values* 24
letrec 15; 24, 63
letrec* 16; 24, 63
letrec-syntax 21; 24
libraries 5
list 37; 39
list->string 44
list->vector 45
list-copy 40
list-ref 39
list-set! 39
list-tail 39
list? 38
load 54
location 10
log 35

macro 20
macro keyword 20
macro transformer 20
macro use 20
magnitude 36
make-bytevector 45
make-list 39
make-parameter 18
make-polar 36
make-rectangular 36
make-string 43
make-vector 44
map 46
max 33
member 39
memq 39
memv 39
min 33
modulo 34
mutable 10
nan? 33
negative? 33
newline 54
nil 37
no-fold-case@#!no-fold-case 8
not 37
null-environment 50
null? 38
number 30
number->string 36
number? 32; 9, 30, 33
numerator 35
numerical types 30
#o 32; 57
object 5
odd? 33
only 26
open-binary-input-file 52
open-binary-output-file 52
open-input-bytevector 52
open-input-file 52
open-input-string 52
open-output-bytevector 52
open-output-file 52
open-output-string 52
or 14; 62
output-port? 51

pair 37
pair? 38; 9
parameter object 18
parameterize 18; 24
peek-char 53
peek-u8 53
port 51
port-open? 51
common index 79

Index 79

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?>? 43
string?>? 43
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
true 9; 13, 14, 37
truncate 35
truncate-quotient 34
truncate-remainder 34
truncate/ 34
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