Revised\textsuperscript{4} Report on the Algorithmic Language Scheme

\textbf{SUMMARY}

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

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

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

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

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

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

The appendix describes a macro facility that may be used to extend the syntax of Scheme.

The report concludes with a bibliography and an alphabetic index.
INTRODUCTION

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

Scheme was one of the first programming languages to incorporate first class procedures as in the lambda calculus, thereby proving the usefulness of static scope rules and block structure in a dynamically typed language. Scheme was the first major dialect of Lisp to distinguish procedures from lambda expressions and symbols, to use a single lexical environment for all variables, and to evaluate the operator position of a procedure call in the same way as an operand position. By relying entirely on procedure calls to express iteration, Scheme emphasized the fact that tail-recursive procedure calls are essentially goto's that pass arguments. 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. More recently, building upon the design of generic arithmetic in Common Lisp, Scheme introduced the concept of exact and inexact numbers. With the appendix to this report Scheme becomes the first programming language to support hygienic macros, which permit the syntax of a block-structured language to be extended reliably.

Background

The first description of Scheme was written in 1975 [91]. A revised report [85] appeared in 1978, which described the evolution of the language as its MIT implementation was upgraded to support an innovative compiler [80]. Three distinct projects began in 1981 and 1982 to use variants of Scheme for courses at MIT, Yale, and Indiana University [66, 57, 34]. An introductory computer science textbook using Scheme was published in 1984 [2].

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 [8] was published at MIT and Indiana University in the summer of 1985. Another round of revision took place in the spring of 1986 [68]. The present report reflects further revisions agreed upon in a meeting that preceded the 1988 ACM Conference on Lisp and Functional Programming and in subsequent electronic mail.

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

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

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

1.1. Semantics

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

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

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

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

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

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

One distinguishing feature of Scheme is that continuations, which in most other languages only operate behind the scenes, also have "first-class" status. Continuations are useful for implementing a wide variety of advanced control constructs, including non-local exits, backtracking, and coroutines. See section 6.9.

Arguments to Scheme procedures are always passed by value, which means that the actual argument expressions are evaluated before the procedure gains control, whether the procedure needs the result of the evaluation or not.

ML, C, and APL are three other languages that always pass arguments by value. This is distinct from the lazy-evaluation semantics of Haskell, or the call-by-name semantics of Algol 60, where an argument expression is not evaluated unless its value is needed by the procedure.

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

1.2. Syntax

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

The read procedure performs syntactic as well as lexical decomposition of the data it reads. The read procedure parses its input as data (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. Essential and non-essential features

It is required that every implementation of Scheme support features that are marked as being essential. Features not explicitly marked as essential are not essential. Implementations are free to omit non-essential features of Scheme or to add extensions, provided the extensions are not in conflict with the language reported here. In particular, implementations must support portable code by providing a syntactic mode that preempts no lexical conventions of this report and reserves no identifiers other than those listed as syntactic keywords in section 2.1.

1.3.2. Error situations and unspecified behavior

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

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

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

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

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

1.3.3. Entry format

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

`template essential category`

if the feature is an essential feature, or simply

`template category`

if the feature is not an essential feature.

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

`{thing1} ...`

indicates zero or more occurrences of a `{thing}`, and

`{thing1} {thing2} ...`

indicates one or more occurrences of a `{thing}`.

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

`(vector-ref vector k)` essential procedure

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

`(make-vector k)` essential procedure

`(make-vector k fill)` procedure

indicate that in all implementations, the `make-vector` procedure must be defined to take one argument, and some implementations will extend it to take 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 3.4, then that argument must be of the named type. For example, the header line for `vector-ref` given above dictates that the first argument to `vector-ref` must be a vector. The following naming conventions also imply type restrictions:

- `obj` any object
- `list, list1, ..., listj, ...` list (see section 6.3)
- `z`, `z1`, ..., `zj`, ... complex number
- `x`, `x1`, ..., `xj`, ... real number
- `y`, `y1`, ..., `yj`, ... real number
- `q`, `q1`, ..., `qi`, ... rational number
- `n`, `n1`, ..., `nj`, ... integer
- `k`, `k1`, ..., `kj`, ... exact non-negative integer

1.3.4. Evaluation examples

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

`( * 5 8 )  ⇒  40`

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

1.3.5. Naming conventions

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

By convention, the names of procedures that store values into previously allocated locations (see section 3.5) usually end in “!” . Such procedures are called mutation procedures. By convention, the value returned by a mutation procedure is unspecified.

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

2. Lexical conventions

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

Upper and lower case forms of a letter are never distinguished except within character and string constants. For example, Foo is the same identifier as FOO, and #x1ab is the same number as #X1ab.

2.1. Identifiers

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

```
lambda q
list->vector soup
+ V17a
<=? a34KTMNs
the-word-recursion-has-many-meanings
```

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

```
+ - . * / < - > ! ? : $ % _ \ ^ \`
```

See section 7.1.1 for a formal syntax of identifiers.

Identifiers have several uses within Scheme programs:

- Certain identifiers are reserved for use as syntactic keywords (see below).
- Any identifier that is not a syntactic keyword may be used as a variable (see section 3.1).
- 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 6.4).

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

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

Some implementations allow all identifiers, including syntactic keywords, to be used as variables. This is a compatible extension to the language, but ambiguities in the language result when the restriction is relaxed, and the ways in which these ambiguities are resolved vary between implementations.

2.2. Whitespace and comments

Whitespace characters are spaces and newlines. (Implementations typically provide additional whitespace characters such as tab or page break.) Whitespace is used for improved readability and as necessary to separate tokens from each other, a token being an indivisible lexical unit such as an identifier or number, but is otherwise insignificant. Whitespace may occur between any two tokens, but not within a token. Whitespace may also occur inside a string, where it is significant.

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

```
;;; The FACT procedure computes the factorial
;;; of a non-negative integer.
(define fact
  (lambda (n)
    (if (zero? n) 1
      (* n (fact (- n 1))))))
```

2.3. Other notations

For a description of the notations used for numbers, see section 6.5.

- + - These are used in numbers, and may also occur anywhere in an identifier except as the first character. A delimited plus or minus sign by itself is also an identifier. A delimited period (not occurring within a number or identifier) is used in the notation for pairs (section 6.3), and to indicate a rest-parameter in a formal parameter list (section 4.1.4). A delimited sequence of three successive periods is also an identifier.

( ) Parentheses are used for grouping and to note lists (section 6.3).

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

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

, @ The character comma and the sequence comma at-sign are used in conjunction with backquote (section 4.2.6).
" The double quote character is used to delimit strings (section 6.7).

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

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

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

#t #f These are the boolean constants (section 6.1).

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

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

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

3. Basic concepts

3.1. Variables and regions

Any identifier that is not a syntactic keyword (see section 2.1) may be used as a variable. A variable may name a location where a value can be stored. A variable that does so is said to be bound to the location. The set of all visible bindings in effect at some point in a program is known as the environment in effect at that point. The value stored in the location to which a variable is bound is called the variable's value. By abuse of terminology, the variable is sometimes said to name the value or to be bound to the value. This is not quite accurate, but confusion rarely results from this practice.

Certain expression types are used to create new locations and to bind variables to those locations. The most fundamental of these binding constructs is the lambda expression, because all other binding constructs can be explained in terms of lambda expressions. The other binding constructs are let, let*, letrec, and do expressions (see sections 4.1.4, 4.2.2, and 4.2.4).

Like Algol and Pascal, and unlike most other dialects of Lisp except for Common Lisp, Scheme is a statically scoped language with block structure. To each place where a variable is bound in a program there corresponds a region of the program text within which the binding is effective. The region is determined by the particular binding construct that establishes the binding; if the binding is established by a lambda expression, for example, then its region is the entire lambda expression. Every reference to or assignment of a variable refers to the binding of the variable that established the innermost of the regions containing the use.

If there is no binding of the variable whose region contains the use, then the use refers to the binding for the variable in the top level environment, if any (section 6); if there is no binding for the identifier, it is said to be unbound.

3.2. True and false

Any Scheme value can be used as a boolean value for the purpose of a conditional test. As explained in section 6.1, all values count as true in such a test except for #f. This report uses the word “true” to refer to any Scheme value that counts as true, and the word “false” to refer to #f.

Note: In some implementations the empty list also counts as false instead of true.

3.3. External representations

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

The external representation of an object is not necessarily unique. The integer 28 also has representations "#e28.000" and "#x1c", and the list in the previous paragraph also has the representations "(0 8 13 )" and "(8 . (13 . ()))" (see section 6.3).

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

An external representation may be written in a program to obtain the corresponding object (see quote, section 4.1.2). External representations can also be used for input and output. The procedure read (section 6.10.2) parses external representations, and the procedure write (section 6.10.3) generates them. Together, they provide an elegant and powerful input/output facility.

Note that the sequence of characters "(+ 2 6)" is not an external representation of the integer 8, even though it is an expression evaluating to the integer 8; rather, it is an external representation of a three-element list, the elements of which are the symbol + and the integers 2 and 6. Scheme's syntax has the property that any sequence of characters that is an expression is also the external representation of some object. This can lead to confusion, since it may not be obvious out of context whether a given sequence of characters is intended to denote data or program, but it is also a source of power, since it facilitates writing programs such as interpreters and compilers that treat programs as data (or vice versa).
The syntax of external representations of various kinds of objects accompanies the description of the primitives for manipulating the objects in the appropriate sections of chapter 6.

3.4. Disjointness of types

No object satisfies more than one of the following predicates:

- boolean?
- pair?
- symbol?
- number?
- char?
- string?
- vector?
- procedure?

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

3.5. Storage model

Variables and objects such as pairs, vectors, and strings implicitly denote locations or sequences of locations. A string, for example, denotes as many locations as there are characters in the string. (These locations need not correspond to a full machine word.) A new value may be stored into one of these locations using the \texttt{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 \texttt{car}, \texttt{vector-ref}, or \texttt{string-ref}, is equivalent in the sense of \texttt{eqv?} (section 6.2) to the object last stored in the location before the fetch.

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

In many systems it is desirable for constants (i.e. the values of literal expressions) to reside in read-only-memory. To express this, it is convenient to imagine that every object that denotes locations is associated with a flag telling whether that object is mutable or immutable. The constants and the strings returned by \texttt{symbol->string} are then the 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.

4. Expressions

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

Expression types are categorized as \textit{primitive} or \textit{derived}. Primitive expression types include variables and procedure calls. Derived expression types are not semantically primitive, but can instead be explained in terms of the primitive constructs as in section 7.3. They are redundant in the strict sense of the word, but they capture common patterns of usage, and are therefore provided as convenient abbreviations.

4.1. Primitive expression types

4.1.1. Variable references

\texttt{(variable)}

\texttt{essential syntax}

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

\begin{verbatim}
(define x 28)
x => 28
\end{verbatim}

4.1.2. Literal expressions

\texttt{(quote (datum))}

\texttt{essential syntax}

\texttt{'}(datum)

\texttt{essential syntax}

\texttt{(quote (datum))} evaluates to \texttt{(datum)}. \texttt{(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.

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

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

\begin{verbatim}
\texttt{'a} => a
\texttt{'\(a b c\)} => #\(a b c\)
\texttt{'}() => ()
\texttt{'+ 1 2} => (+ 1 2)
\texttt{'(quote a)} => (quote a)
\texttt{''a} => (quote a)
\end{verbatim}

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

\begin{verbatim}

"abc" => "abc"
"abc" => "abc"
\texttt{'+15932} => 15932
15932 => 15932
\texttt{'+#t} => #t
\texttt{#t} => #t
\end{verbatim}

As noted in section 3.5, it is an error to alter a constant (i.e. the value of a literal expression) using a mutation procedure like \texttt{set-car!} or \texttt{string-set!}.
4.1.3. Procedure calls

((operator) (operand₁) ...) essential syntax

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

( + 3 4 )    => 7
((if #f + *) 3 4) => 12

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

Procedure calls are also called combinations.

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

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

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

4.1.4. Lambda expressions

(lambda (formals) (body)) essential syntax

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

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

(l lambda (x) (+ x x)) => a procedure
((lambda (x) (+ x x)) 4)  =>  8

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

(define add
(let ((x 4))
(l lambda (y) (+ x y)))
(add 4) => 10

(Formals) should have one of the following forms:

- ((variable₁) ...): The procedure takes a fixed number of arguments; when the procedure is called, the arguments will be stored in the bindings of the corresponding variables.

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

- ((variable₁) ... (variableₙ₋₁) . (variableₙ)): If a space-delimited period precedes the last variable, then the value stored in the binding of the last variable will be a newly allocated list of the actual arguments left over after all the other actual arguments have been matched up against the other formal arguments.

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

((lambda x x) 3 4 5 6) => (3 4 5 6)
((lambda (x y . z) z) => (5 6)

Each procedure created as the result of evaluating a lambda expression is tagged with a storage location, in order to make eqv? and eq? work on procedures (see section 6.2).

4.1.5. Conditionals

(if (test) (consequent) (alternate)) essential syntax

(if (test) (consequent)) syntax

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

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

(if (> 3 2) 'yes 'no) => yes
(if (> 2 3) 'yes 'no) => no
(if (> 3 2) (= 3 2)
(+) 3 2)) => 1
4.1.6. Assignments

(set! (variable) (expression)) essential syntax

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

(define x 2)
(+ x 1) \Rightarrow 3
(set! x 4) \Rightarrow unspecified
(+ x 1) \Rightarrow 5

4.2. Derived expression types

For reference purposes, section 7.3 gives rewrite rules that will convert constructs described in this section into the primitive constructs described in the previous section.

4.2.1. Conditionals

(cond (clause₁) (clause₂) ... ) essential syntax

Syntax: Each (clause) should be of the form

(⟨test⟩ ⟨expression⟩ ... )

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

(else ⟨expression₁⟩ ⟨expression₂⟩ ... )

Semantics: A cond expression is evaluated by evaluating the (test) expressions of successive (clause)s in order until one of them evaluates to a true value (see section 6.1). When a (test) evaluates to a true value, then the remaining (expression)s in its (clause) are evaluated in order, and the result of the last (expression) in the (clause) is returned as the result of the entire cond expression. If the selected (clause) contains only the (test) and no (expression)s, then the value of the (test) is returned as the result. If all (test)s evaluate to false values, and there is no else clause, then the result of the conditional expression is unspecified; if there is an else clause, then its (expression)s are evaluated, and the value of the last one is returned.

(cond ((> 3 2) 'greater)
((< 3 2) 'less)) \Rightarrow greater
(cond ((> 3 3) 'greater)
((< 3 3) 'less)
(else 'equal)) \Rightarrow equal

Some implementations support an alternative (clause) syntax, (⟨test⟩ \Rightarrow (recipient)), where (recipient) is an expression. If (test) evaluates to a true value, then (recipient) is evaluated. Its value must be a procedure of one argument; this procedure is then invoked on the value of the (test).

(cond ((assv 'b '((a 1) (b 2))) \Rightarrow cadr)
(else #f)) \Rightarrow 2

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

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

(((datum₁) ... ) (expression₁) (expression₂) ... )

where each (datum) is an external representation of some object. All the (datum)’s must be distinct. The last (clause) may be an “else clause,” which has the form

(else (expression₁) (expression₂) ... ).

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 eql?; see section 6.2) to a (datum), then the expressions in the corresponding (clause) are evaluated from left to right and the result of the last expression in the (clause) is returned as the result of the case expression. If the result of evaluating (key) is different from every (datum), then if there is an else clause its expressions are evaluated and the result of the last is the result of the case expression; otherwise the result of the case expression is unspecified.

(case (* 2 3)
((2 3 5 7) 'prime)
(((1 4 6 8 9) 'composite)) \Rightarrow composite
(case (car '(c d))
((a) 'a)
((b) 'b)) \Rightarrow unspecified
(case (car '(c d))
((a e i o u) 'vowel)
((v y) 'semivowel)
(else 'consonant)) \Rightarrow consonant

(and ⟨test₁⟩ ...) essential syntax

The (test) expressions are evaluated from left to right, and the value of the first expression that evaluates to a false value (see section 6.1) is returned. Any remaining expressions are not evaluated. If all the expressions evaluate to true values, the value of the last expression is returned. If there are no expressions then #t is returned.

(and (= 2 2) (> 2 1)) \Rightarrow #t
(and (= 2 2) (< 2 1)) \Rightarrow #f
(and 1 2 'c '(f g)) \Rightarrow (f g)
(and) \Rightarrow #t

(or ⟨test₁⟩ ...) essential 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 6.1) is returned. Any remaining expressions are not evaluated. If all expressions evaluate to false values, the value of the last expression is returned. If there are no expressions then #f is returned.
4.2.2. Binding constructs

The three binding constructs let, let*, and letrc give Scheme a block structure, like Algol 60. The syntax of the three constructs is identical, but they differ in the regions they establish for their variable bindings. In a let expression, the initial values are computed before any of the variables become bound; in a let* expression, the bindings and evaluations are performed sequentially; while in a letrc expression, all the bindings are in effect while their initial values are being computed, thus allowing mutually recursive definitions.

(let (bindings) (body))  essential syntax

Syntax: (Bindings) should have the form
((variable1) (init1) ...),
where each (init) is an expression, and (body) should be a sequence of one or more expressions. It is an error for a (variable) to appear more than once in the list of variables being bound.

Semantics: The (init)s are evaluated in the current environment (in some unspecified order), the (variable)s are bound to fresh locations holding the results; the (body) is evaluated in the extended environment, and the value of the last expression in (body) is returned. Each binding of a (variable) has the entire letrc expression as its region, making it possible to define mutually recursive procedures.

(let ((x 2) (y 3))
  (* x y))  \(\Rightarrow\) 6

(let ((x 2) (y 3))
  (let ((x 7)
          (z (+ x y)))
       (* z x)))  \(\Rightarrow\) 35

See also named let, section 4.2.4.

(let* (bindings) (body))  syntax

Syntax: (Bindings) should have the form
((variable1) (init1) ...),
and (body) should be a sequence of one or more expressions.

Semantics: Let* is similar to let, but the bindings are performed sequentially from left to right, and the region of a binding indicated by ((variable) (init)) is that part of the let* expression to the right of the binding. Thus the second binding is done in an environment in which the first binding is visible, and so on.

(letrec (bindings) (body))  essential syntax

Syntax: (Bindings) should have the form
((variable1) (init1) ...),
and (body) should be a sequence of one or more expressions. It is an error for a (variable) to appear more than once in the list of variables being bound.

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

(letrec ((even? (lambda (n)
                    (if (zero? n)
                        #t
                        (odd? (- n 1)))))
         (odd? (lambda (n)
                 (if (zero? n)
                     #f
                     (even? (- n 1)))))
        (even? 88))  \(\Rightarrow\) #t

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

4.2.3. Sequencing

(begin (expression1) (expression2) ...) essential syntax

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

(define x 0)

(begin (set! x 5)
       (+ x 1))  \(\Rightarrow\) 6
4. Expressions

4.2.4. Iteration

\[
\text{(do (\langle\text{variable}\rangle \langle\text{init}\rangle \langle\text{step}\rangle) \ldots) (\langle\text{test}\rangle \langle\text{expression}\rangle \ldots) (\langle\text{command}\rangle \ldots)}
\]

Do is an iteration construct. It specifies a set of variables to be bound, how they are to be initialized at the start, and how they are to be updated on each iteration. When a termination condition is met, the loop exits with a specified result value.

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

Each iteration begins by evaluating (test); if the result is false (see section 6.1), then the (command) expressions are evaluated in order for effect, the (step) expressions are evaluated in some unspecified order, the (variable) values are bound to fresh locations, the results of the (step) expressions are stored in the bindings of the (variable) values, and the next iteration begins.

If (test) evaluates to a true value, then the (expression) values are evaluated from left to right and the value of the last (expression) is returned as the value of the do expression. If no (expression) values 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) values. 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 (\langle\text{variable}\rangle \langle\text{init}\rangle \langle\text{variable}\rangle) had been written instead of (\langle\text{variable}\rangle \langle\text{init}\rangle).

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

\[
\text{(let ((x '(1 3 5 7 9))) (do ((x x (cdr x)) (sum 0 (+ sum (car x)))) ((null? x) sum)) \Rightarrow 25)}
\]

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

Some implementations of Scheme permit a variant on the syntax of let called "named let" which provides a more general looping construct than do, and may also be used to express recursions.

Named let has the same syntax and semantics as ordinary let except that (variable) is bound within (body) to a procedure whose formal arguments are the bound variables and whose body is (body). Thus the execution of (body) may be repeated by invoking the procedure named by (variable).

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

4.2.5. Delayed evaluation

\[
\text{(delay (expression))}
\]

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.

See the description of force (section 6.9) for a more complete description of delay.

4.2.6. Quasiquotation

\[
\text{(quasiquote (template))}
\]

"Backquote" or "quasiquote" expressions are useful for constructing a list or vector structure when most but not all of the desired structure is known in advance. If no commas appear within the (template), the result of evaluating '(template) is equivalent to the result of evaluating '(template). If a comma appears within the (template), however, the expression following the comma is evaluated ("unquoted") and its result is inserted into the structure instead of the comma and the expression. If a comma appears followed immediately by an at-sign (@), then the following expression must evaluate to a list; the opening and closing parentheses of the list are then "stripped away" and the elements of the list are inserted in place of the comma at-sign expression sequence.
```
`(list (+ 1 2) 4) => (list 3 4)
(let ((name 'a) `(list ,name ,name))
     => (list a (quote a))
`(a (+ 1 2), ,@ (map abs '(4 -5 6)) b)
     => (a 3 4 5 6 b)
`((foo ,(- 10 3)), ,@ (cdr 'c)) ,,(car 'cons))
     => ((foo 7) . cons)
`#(1 0 5 , (sqrt 4) , @ (map sqrt '(16 9))) 8
     => #((10 5 2 4 3 8)
```

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

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

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

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

Unpredictable behavior can result if any of the symbols quasiquote, unquote, or unquote-splicing appear in positions within a (template) otherwise than as described above.

5. Program structure

5.1. Programs

A Scheme program consists of a sequence of expressions and definitions. Expressions are described in chapter 4; definitions are the subject of the rest of this chapter. Programs are typically stored in files or entered interactively to a running Scheme system, although other paradigms are possible; questions of user interface lie outside the scope of this report. (Indeed, Scheme would still be useful as a notation for expressing computational methods even in the absence of a mechanical implementation.)

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

5.2. Definitions

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

A definition should have one of the following forms:

- `(define (variable) (expression))

  This syntax is essential.

- `(define ((variable) (formals)) (body))

  This syntax is not essential. (Formals) should be either a sequence of zero or more variables, or a sequence of one or more variables followed by a space-delimited period and another variable (as in a lambda expression). This form is equivalent to

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

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

  This syntax is not essential. (Formal) should be a single variable. This form is equivalent to

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

- `(begin (definition1) ...)"

  This syntax is essential. This form is equivalent to the set of definitions that form the body of the begin.

5.2.1. Top level definitions

At the top level of a program, a definition

```
(define (variable) (expression))
```

has essentially the same effect as the assignment expression

```
(set! (variable) (expression))
```

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

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

All Scheme implementations must support top level definitions.

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

Some implementations of Scheme permit definitions to occur at the beginning of a (body) (that is, the body of a `lambda`, `let`, `let*`, `letrec`, or `define` expression). Such definitions are known as internal definitions as opposed to the top level definitions described above. The variable defined by an internal definition is local to the (body). That is, (variable) is bound rather than assigned, and the region of the binding is the entire (body). For example,

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

A (body) containing internal definitions can always be converted into a completely equivalent `letrec` expression. For example, the `let` expression in the above example is equivalent to

```
(let ((x 5))
  (letrec ((foo (lambda (y) (bar x y)))
           (bar (lambda (a b) (+ (* a b) a))))
       (foo (+ x 3)))
```

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

6. Standard procedures

This chapter describes Scheme's built-in procedures. The initial (or “top level”) Scheme environment starts out with a number of variables bound to locations containing useful values, most of which are primitive procedures that manipulate data. For example, the variable `abs` is bound to (a location initially containing) a procedure of one argument that computes the absolute value of a number, and the variable `+` is bound to a procedure that computes sums.

6.1. Booleans

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

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

Note: In some implementations the empty list counts as false, contrary to the above. Nonetheless a few examples in this report assume that the empty list counts as true, as in [50].

Note: Programmers accustomed to other dialects of Lisp should be aware that Scheme distinguishes both `#f` and the empty list from the symbol `nil`.

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

```
#t               => #t
#f               => #f
(not obj)        essential procedure

Not returns #t if obj is false, and returns #f otherwise.

(boolean? obj)   essential procedure

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

6.2. Equivalence predicates

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

`eqv? obj1 obj2`    essential procedure

The `eqv?` procedure defines a useful equivalence relation on objects. Briefly, it returns `#t` if `obj1` and `obj2` should normally be regarded as the same object. This relation is left slightly open to interpretation, but the following partial specification of `eqv?` holds for all implementations of Scheme.

The `eqv?` procedure returns `#t` if:

- `obj1` and `obj2` are both `#t` or both `#f`.
- `obj1` and `obj2` are both symbols and
(string=? (symbol->string obj1) (symbol->string obj2))

Note: This assumes that neither obj1 nor obj2 is an "un-interned symbol" as alluded to in section 6.4. This report does not presume to specify the behavior of eqv? on implementation-dependent extensions.

- obj1 and obj2 are both numbers, are numerically equal (see =, section 6.5), and are either both exact or both inexact.
- obj1 and obj2 are both characters and are the same character according to the char=? procedure (section 6.6).
- both obj1 and obj2 are the empty list.
- obj1 and obj2 are pairs, vectors, or strings that denote the same locations in the store (section 3.5).
- obj1 and obj2 are procedures whose location tags are equal (section 4.1.4).

The eqv? procedure returns #f if:

- obj1 and obj2 are of different types (section 3.4).
- one of obj1 and obj2 is #t but the other is #f.
- obj1 and obj2 are symbols but
  (string=? (symbol->string obj1) (symbol->string obj2))

- 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.
- 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, or strings that denote distinct locations.
- obj1 and obj2 are procedures that would behave differently (return a different value or have different side effects) for some arguments.

(eqv? 'a 'a)  \(\Rightarrow\) #t
(eqv? 'a 'b)  \(\Rightarrow\) #f
(eqv? 2 2)  \(\Rightarrow\) #t
(eqv? '() '())  \(\Rightarrow\) #t
(eqv? 'a)  \(\Rightarrow\) #t
(eqv? 1000000000 1000000000)  \(\Rightarrow\) #t
(eqv? (cons 1 2) (cons 1 2))  \(\Rightarrow\) #f
(eqv? (lambda () 1))  \(\Rightarrow\) #f

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

(eqv? "" "")  \(\Rightarrow\) unspecified
(eqv? '#()' '#())  \(\Rightarrow\) unspecified
(eqv? (lambda (x) x) (lambda (x) x))  \(\Rightarrow\) unspecified
(eqv? (lambda (x) x) (lambda (y) y))  \(\Rightarrow\) unspecified

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

(define gen-counter
  (lambda ()
    (let ((n 0))
      (lambda () (set! n (+ n 1)) n))))

(let ((g (gen-counter))
  (eqv? g g))  \(\Rightarrow\) #t
(eqv? (gen-counter) (gen-counter))  \(\Rightarrow\) #f

(define gen-loser
  (lambda ()
    (let ((n 0))
      (lambda () (set! n (+ n 1)) 27))))

(let ((g (gen-loser))
  (eqv? g g))  \(\Rightarrow\) #f
(eqv? (gen-loser) (gen-loser))  \(\Rightarrow\) unspecified

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

(letrec ((f (lambda () (if (eqv? f g) 'f 'both)))
  (g (lambda () (if (eqv? f g) 'g 'both))))
  (eqv? f g))  \(\Rightarrow\) #f

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

(eqv? '(a) '(a))  \(\Rightarrow\) unspecified
(eqv? "a" "a")  \(\Rightarrow\) unspecified
(eqv? '(b) (cdr '(a b)))  \(\Rightarrow\) unspecified
(let ((x '(a)))
  (eqv? x x))  \(\Rightarrow\) #t
Rationale: The above definition of `eqv?` allows implementations latitude in their treatment of procedures and literals: implementations are free either to detect or to fail to detect that two procedures or two literals are equivalent to each other, and can decide whether or not to merge representations of equivalent objects by using the same pointer or bit pattern to represent both.

\[(eq\ ? \ obj\_1 \ obj\_2)\] essential procedure

`Eq?` is similar to `eqv?` except that in some cases it is capable of discerning distinctions finer than those detectable by `eqv?`.

`Eq?` and `eqv?` are guaranteed to have the same behavior on symbols, booleans, the empty list, pairs, and non-empty strings and vectors. `Eq?`’s behavior on numbers and characters is implementation-dependent, but it will always return either true or false, and will return true only when `eqv?` would also return true. `Eq?` may also behave differently from `eqv?` on empty vectors and empty strings.

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

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

\[(equal? \ obj\_1 \ obj\_2)\] essential procedure

`Equal?` recursively compares the contents of pairs, vectors, and strings, applying `eqv?` on other objects such as numbers and symbols. A rule of thumb is that objects are generally `equal?` if they print the same. `Equal?` may fail to terminate if its arguments are circular data structures.

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

6.3. Pairs and lists

A `pair` (sometimes called a `dotted pair`) is a record structure with two fields called the `car` and `cdr` fields (for historical reasons). Pairs are created by the procedure `cons`. The car and cdr fields are accessed by the procedures `car` and `cdr`. The car and cdr fields are assigned by the procedures `set-car!` and `set-cdr!`.

Pairs are used primarily to represent lists. A list can be defined recursively as either the empty list or a pair whose cdr is a list. More precisely, the set of lists is defined as the smallest set X such that

- The empty list is in X.
- If list is in X, then any pair whose cdr field contains list is also in X.

The objects in the car fields of successive pairs of a list are the elements of the list. For example, a two-element list is a pair whose car is the first element and whose cdr is a pair whose car is the second element and whose cdr is the empty list. The length of a list is the number of elements, which is the same as the number of pairs.

The empty list is a special object of its own type (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 \texttt{set-cdr!} procedure is used, an object can be a list one moment and not the next:

\[
\begin{align*}
\text{(define } x \ (\text{list } 'a \ 'b \ c) \text{)} \\
\text{(define } y \ x \text{)}
\end{align*}
\]

\[
\begin{align*}
y & \quad \Rightarrow \quad (a \ b \ c) \\
(\text{list? } y) & \quad \Rightarrow \quad \#t \\
(\text{set-cdr! } x \ 4) & \quad \Rightarrow \quad \text{unspecified} \\
x & \quad \Rightarrow \quad (a \ . \ 4) \\
(\text{eqv? } x \ y) & \quad \Rightarrow \quad \#t \\
y & \quad \Rightarrow \quad (a \ . \ 4) \\
(\text{list? } y) & \quad \Rightarrow \quad \#f \\
(\text{set-cdr! } x \ x) & \quad \Rightarrow \quad \text{unspecified} \\
(\text{list? } x) & \quad \Rightarrow \quad \#f
\end{align*}
\]

Within literal expressions and representations of objects read by the \texttt{read} procedure, the forms `'\text{datum}', `(\text{datum}), ,\text{datum}, and ,\text{datum} denote two-element lists whose first elements are the symbols \texttt{quote}, \texttt{quasiquote}, \texttt{unquote}, and \texttt{unquote-splicing}, respectively. The second element in each case is \texttt{datum}. This convention is supported so that arbitrary Scheme programs may be represented as lists. That is, according to Scheme's grammar, every \texttt{expression} is also a \texttt{datum} (see section 7.1.2). Among other things, this permits the use of the \texttt{read} procedure to parse Scheme programs. See section 3.3.

\[
(\text{pair? } \text{obj}) \quad \text{essential procedure}
\]

\texttt{Pair?} returns \#t if \text{obj} is a pair, and otherwise returns \#f.

\[
\begin{align*}
(\text{pair? } '(a \ . \ b)) & \quad \Rightarrow \quad \#t \\
(\text{pair? } '(a \ b \ c)) & \quad \Rightarrow \quad \#t \\
(\text{pair? } '()) & \quad \Rightarrow \quad \#f \\
(\text{pair? } '#(a \ b)) & \quad \Rightarrow \quad \#f
\end{align*}
\]

\[
(\text{cons } \text{obj}_1 \ \text{obj}_2) \quad \text{essential procedure}
\]

\texttt{Returns a newly allocated pair whose \texttt{car} is \text{obj}_1 and whose \texttt{cdr} is \text{obj}_2. The pair is guaranteed to be different (in the sense of \texttt{eqv?}) from every existing object.}

\[
\begin{align*}
(\text{cons } 'a \ '()) & \quad \Rightarrow \quad (a) \\
(\text{cons } '(a \ b \ c) \ d)) & \quad \Rightarrow \quad ((a) \ b \ c \ d) \\
(\text{cons } 'a\ ''(b \ c) \ )) & \quad \Rightarrow \quad ("a" \ b \ c) \\
(\text{cons } 'a \ 3) & \quad \Rightarrow \quad (a \ . \ 3) \\
(\text{cons } '(a \ b) \ 'c) & \quad \Rightarrow \quad ((a \ b) \ . \ c)
\end{align*}
\]

\[
(\text{car } \text{pair}) \quad \text{essential procedure}
\]

\texttt{Returns the contents of the \texttt{car} field of \text{pair}. Note that it is an error to take the \texttt{car} of the empty list.}

\[
\begin{align*}
(\text{car } '((a \ b \ c))) & \quad \Rightarrow \quad a \\
(\text{car } '((a \ b \ c) \ d)) & \quad \Rightarrow \quad (a) \\
(\text{car } '((1 \ . \ 2))) & \quad \Rightarrow \quad 1 \\
(\text{car } '()) & \quad \Rightarrow \quad \text{error}
\end{align*}
\]

\[
(\text{cdr } \text{pair}) \quad \text{essential procedure}
\]

\texttt{Returns the contents of the \texttt{cdr} field of \text{pair}. Note that it is an error to take the \texttt{cdr} of the empty list.}

\[
\begin{align*}
(\text{cdr } '((a \ b \ c) \ d)) & \quad \Rightarrow \quad (b \ c \ d) \\
(\text{cdr } '((1 \ . \ 2))) & \quad \Rightarrow \quad 2 \\
(\text{cdr } '()) & \quad \Rightarrow \quad \text{error}
\end{align*}
\]

\[
(\text{set-car! } \text{pair } \text{obj}) \quad \text{essential procedure}
\]

\texttt{Stores \text{obj} in the \texttt{car} field of \text{pair}. The value returned by \texttt{set-car!} is unspecified.}

\[
\begin{align*}
(\text{define } f \ (\text{list } \text{'not-a-constant-list})) \\
(\text{define } g \ (\text{\texttt{constant-list}})) \\
(\text{set-car! } f \ 3) & \quad \Rightarrow \quad \text{unspecified} \\
(\text{set-car! } g \ 3) & \quad \Rightarrow \quad \text{error}
\end{align*}
\]

\[
(\text{set-cdr! } \text{pair } \text{obj}) \quad \text{essential procedure}
\]

\texttt{Stores \text{obj} in the \texttt{cdr} field of \text{pair}. The value returned by \texttt{set-cdr!} is unspecified.}

\[
\begin{align*}
\text{(caar pair)} \quad \text{essential procedure} \\
\text{(cadr pair)} \quad \text{essential procedure} \\
\text{(cdadr pair)} \quad \text{essential procedure} \\
\text{(cddddr pair)} \quad \text{essential procedure}
\end{align*}
\]

\texttt{These procedures are compositions of \texttt{car} and \texttt{cdr}, where for example \texttt{caddr} could be defined by}

\[
(\text{define caddr } (\text{\texttt{lambda} } (x) \ (\text{car } (\text{cdr } (\text{cdr } x))))).
\]

\texttt{Arbitrary compositions, up to four deep, are provided. There are twenty-eight of these procedures in all.}

\[
(\text{null? } \text{obj}) \quad \text{essential procedure}
\]

\texttt{Returns \#t if \text{obj} is the empty list, otherwise returns \#f.}

\[
(\text{list? } \text{obj}) \quad \text{essential procedure}
\]

\texttt{Returns \#t if \text{obj} is a list, otherwise returns \#f. By definition, all lists have finite length and are terminated by the empty list.}
(list? 'a b c)  \rightarrow  \#t
(list? '())      \rightarrow  \#t
(list? 'a . b)   \rightarrow  \#f
(let ((x (list 'a)))
  (set-cdr! x x)
)(list? x)         \rightarrow  \#f

(list obj ...)

Returns a newly allocated list of its arguments.

(list 'a (+ 3 4) 'c) \rightarrow  (a 7 c)
(list)             \rightarrow  ()

(length list)

Returns the length of list.

(length '(a b c)) \rightarrow  3
(length '(a (b) (c d e))) \rightarrow  3
(length '())       \rightarrow  0

(append list ...)

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

(append '(x) '(y)) \rightarrow  (x y)
(append '(a) '(b c d)) \rightarrow  (a b c d)
(append '(a (b)) '((c))) \rightarrow  (a (b) (c))

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

(append '(a b) '(c . d)) \rightarrow  (a b c . d)
(append '() 'a)         \rightarrow  a

(reverse list)

Returns a newly allocated list consisting of the elements of list in reverse order.

(reverse '(a b c)) \rightarrow  (c b a)
(reverse '(a (b c) d (e (f)))) \rightarrow  (((e (f)) d (b c) a)

(list-tail list k)

Returns the sublist of list obtained by omitting the first k elements. List-tail could be defined by

(define list-tail
  (lambda (x k)
    (if (zero? k)
        x
        (list-tail (cdr x) (- k 1)))))

(list-ref list k)

Returns the kth element of list. (This is the same as the car of (list-tail list k).

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

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

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

(memq 'a '(a b c)) \rightarrow  (a b c)
(memq 'b '(a b c)) \rightarrow  (b c)
(memq 'a '(a b c d)) \rightarrow  #f
(memq (list 'a) '(b (a) c)) \rightarrow  #f
(member (list 'a)
  '(b (a) c)) \rightarrow  ((a) c)
(memv 101 '(100 101 102)) \rightarrow  unspecified
(memv 101 '(100 101 102)) \rightarrow  (101 102)

(assq obj alist)    essential procedure
(assv obj alist)    essential procedure
(assoc obj alist)   essential procedure

Alist (for “association list”) must be a list of pairs. These procedures find the first pair in alist whose car field is obj, and returns that pair. If no pair in alist has obj as its car, then #f (not the empty list) is returned. Assq uses eq? to compare obj with the car fields of the pairs in alist, while assv uses eqv? and assoc uses equal?.

(assq 5 5
  '((2 3) (5 7) (11 13))) \rightarrow  unspecified
(assv 5 5
  '((2 3) (5 7) (11 13))) \rightarrow  (5 7)

Rationale: Although they are ordinarily used as predicates, memq, memv, member, assq, assv, and assoc do not have question marks in their names because they return useful values rather than just #t or #f.
6.4. Symbols
Symbols are objects whose usefulness rests on the fact that two symbols are identical (in the sense of \texttt{eqv?}) if and only if their names are spelled the same way. This is exactly the property needed to represent identifiers in programs, and so most implementations of Scheme use them internally for that purpose. Symbols are useful for many other applications; for instance, they may be used the way enumerated values are used in Pascal.

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

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

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

Some implementations also have "uninterned symbols", which defeat write/read invariance even in implementations with slashification, and also generate exceptions to the rule that two symbols are the same if and only if their names are spelled the same.

\begin{verbatim}
(symbol? obj)  essential procedure

Returns \#t if \texttt{obj} is a symbol, otherwise returns \#f.

(symbol? 'foo)  \implies  \#t
(symbol? (car '(a b)))  \implies  \#t
(symbol? "bar")  \implies  \#f
(symbol? 'nil)  \implies  \#t
(symbol? '(')  \implies  \#f
(symbol? '#f)  \implies  \#f
\end{verbatim}

\begin{verbatim}
(symbol->string symbol)  essential procedure

Returns the name of \texttt{symbol} as a string. If the symbol was part of an object returned as the value of a literal expression (section 4.1.2) or by a call to the \texttt{read} procedure, and its name contains alphabetic characters, then the string returned will contain characters in the implementation's preferred standard case—some implementations will prefer upper case, others lower case. If the symbol was returned by \texttt{string->symbol}, the case of characters in the string returned will be the same as the case in the string that was passed to \texttt{string->symbol}. It is an error to apply \texttt{mutation} procedures like \texttt{string-set!} to strings returned by this procedure.

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

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

\begin{verbatim}
(string->symbol string)  essential procedure

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

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

\begin{verbatim}
eq? 'mISSISSIppi 'mississippi)  \implies  \#t
(string->symbol "mISSISSIppi")  \implies  the symbol with name "mISSISSIppi"
(eq? 'bitBlit (string->symbol "bitBlit")\))  \implies  \#f
(eq? 'JollyWog (string->symbol (symbol->string 'JollyWog)))  \implies  \#t
(string->symbol "K. Harper, M.D." (symbol->string (string->symbol "K. Harper, M.D.")))  \implies  \#t
\end{verbatim}
\end{verbatim}

6.5. Numbers
Numerical computation has traditionally been neglected by the Lisp community. Until Common Lisp there was no carefully thought out strategy for organizing numerical computation, and with the exception of the MacLisp system [62] little effort was made to execute numerical code efficiently. This report recognizes the excellent work of the Common Lisp committee and accepts many of their recommendations. In some ways this report simplifies and generalizes their proposals in a manner consistent with the purposes of Scheme.

It is important to distinguish between the mathematical numbers, the Scheme numbers that attempt to model them, the machine representations used to implement the Scheme numbers, and notations used to write numbers. This report uses the types \texttt{number}, \texttt{complex}, \texttt{real}, \texttt{rational}, and \texttt{integer} to refer to both mathematical numbers and Scheme numbers. Machine representations such as fixed point and floating point are referred to by names such as \texttt{fixnum} and \texttt{flonum}.
6.5.1. Numerical types

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

- number
- complex
- real
- rational
- integer

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

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

Scheme's numerical operations treat numbers as abstract data, as independent of their representation as possible. Although an implementation of Scheme may use fixnum, flonum, and perhaps other representations for numbers, this should not be apparent to a casual programmer writing simple programs.

It is necessary, however, to distinguish between numbers that are represented exactly and those that may not be. For example, indexes into data structures must be known exactly, as must some polynomial coefficients in a symbolic algebra system. On the other hand, the results of measurements are inherently inexact, and irrational numbers may be approximated by rational and therefore inexact approximations. In order to catch uses of inexact numbers where exact numbers are required, Scheme explicitly distinguishes exact from inexact numbers. This distinction is orthogonal to the dimension of type.

6.5.2. Exactness

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

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

Rational operations such as + should always produce exact results when given exact arguments. If the operation is unable to produce an exact result, then it may either report the violation of an implementation restriction or it may silently coerce its result to an inexact value. See section 6.5.3.

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

6.5.3. Implementation restrictions

Implementations of Scheme are not required to implement the whole tower of subtypes given in section 6.5.1, but they must implement a coherent subset consistent with both the purposes of the implementation and the spirit of the Scheme language. For example, an implementation in which all numbers are real may still be quite useful.

Implementations may also support only a limited range of numbers of any type, subject to the requirements of this section. The supported range for exact numbers of any type may be different from the supported range for inexact numbers of that type. For example, an implementation that uses flonums to represent all its inexact real numbers may support a practically unbounded range of exact integers and rationals while limiting the range of inexact reals (and therefore the range of inexact integers and rationals) to the dynamic range of the flonum format. Furthermore the gaps between the representable inexact integers and rationals are likely to be very large in such an implementation as the limits of this range are approached.

An implementation of Scheme must support exact integers throughout the range of numbers that may be used for indexes of lists, vectors, and strings or that may result from computing the length of a list, vector, or string. The `length`, `vector-length`, and `string-length` procedures must return an exact integer, and it is an error to use anything but an exact integer as an index. Furthermore any integer constant within the index range, if expressed by an exact integer syntax, will indeed be read as an exact integer, regardless of any implementation restrictions that may apply outside this range. Finally, the procedures listed below will always return an exact integer result provided all their arguments are exact integers and the mathematically expected result is representable as an exact integer within the implementation:
+  -  *
quotient  remainder  modulo
max  min  abs
numerator  denominator  gcd
lcm  floor  ceiling
truncate  round  rationalize

Implementations are encouraged, but not required, to support exact integers and exact rationals of practically unlimited size and precision, and to implement the above procedures and the / procedure in such a way that they always return exact results when given exact arguments. If one of these procedures is unable to deliver an exact result when given exact arguments, then it may either report a violation of an implementation restriction or it may silently coerce its result to an inexact number. Such a coercion may cause an error later.

An implementation may use floating point and other approximate representation strategies for inexact numbers. This report recommends, but does not require, that the IEEE 32-bit and 64-bit floating point standards be followed by implementations that use flonum representations, and that implementations using other representations should match or exceed the precision achievable using these floating point standards [49].

In particular, implementations that use flonum representations must follow these rules: A flonum result must be represented with at least as much precision as is used to express any of the inexact arguments to that operation. It is desirable (but not required) for potentially inexact operations such as sqrt, when applied to exact arguments, to produce exact answers whenever possible (for example the square root of an exact 4 ought to be an exact 2). If, however, an exact number is operated upon so as to produce an inexact result (as by sqrt), and if the result is represented as a flonum, then the most precise flonum format available must be used; but if the result is represented in some other way then the representation must have at least as much precision as the most precise flonum format available.

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

6.5.4. Syntax of numerical constants

The syntax of the written representations for numbers is described formally in section 7.1.1.

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

A numerical constant may be specified to be either exact or inexact by a prefix. The prefixes are #e for exact, and #i for inexact. An exactness prefix may appear before or after any radix prefix that is used. If the written representation of a number has no exactness prefix, the constant may be either inexact or exact. It is inexact if it contains a decimal point, an exponent, or a "#" character in the place of a digit, otherwise it is exact.

In systems with inexact numbers of varying precisions it may be useful to specify the precision of a constant. For this purpose, numerical constants may be written with an exponent marker that indicates the desired precision of the inexact representation. The letters s, f, d, and l specify the use of short, single, double, and long precision, respectively. (When fewer than four internal inexact representations exist, the four size specifications are mapped onto those available. For example, an implementation with two internal representations may map short and single together and long and double together.) In addition, the exponent marker e specifies the default precision for the implementation. The default precision has at least as much precision as double, but implementations may wish to allow this default to be set by the user.

| 3.141592653589799F0  |
| Round to single — 3.141593 |
| 0.6L0                |
| Extend to long — .600000000000000 |

6.5.5. Numerical operations

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

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

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

If $z$ is an inexact complex number, then (real? $z$) is true if and only if (zero? (imag-part $z$)) is true. If $z$ is an inexact real number, then (integer? $x$) is true if and only if ($x$ rounded).

(\[\begin{align*}
\text{complex? 3+4i} & \Leftrightarrow \#t \\
\text{complex? 3} & \Leftrightarrow \#t \\
\text{real? 3} & \Leftrightarrow \#t \\
\text{real? -2.5+0.0i} & \Leftrightarrow \#t \\
\text{real? #e1e10} & \Leftrightarrow \#t \\
\text{rational? 6/10} & \Leftrightarrow \#t \\
\text{rational? 6/3} & \Leftrightarrow \#t \\
\text{integer? 3+0i} & \Leftrightarrow \#t \\
\text{integer? 3.0} & \Leftrightarrow \#t \\
\text{integer? 8/4} & \Leftrightarrow \#t \\
\end{align*}\]

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

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

(exact? $z$) \hspace{1em} essential procedure

(inexact? $z$) \hspace{1em} essential procedure

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

(\[\begin{align*}
\text{= z1 z2 z3 ...} & \Leftrightarrow \#t \\
\text{< x1 x2 x3 ...} & \Leftrightarrow \#t \\
\text{> x1 x2 x3 ...} & \Leftrightarrow \#t \\
\text{<= z1 x2 x3 ...} & \Leftrightarrow \#t \\
\text{>= z1 x2 x3 ...} & \Leftrightarrow \#t \\
\end{align*}\]

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

These predicates are required to be transitive.

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

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

(zero? $z$) \hspace{1em} essential procedure

(positive? $x$) \hspace{1em} essential procedure

(negative? $x$) \hspace{1em} essential procedure

(\[\begin{align*}
\text{odd? n} & \Leftrightarrow \#t \\
\text{even? n} & \Leftrightarrow \#t \\
\end{align*}\]

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

(max $x_1 x_2$ \ldots) \hspace{1em} essential procedure

(min $x_1 x_2$ \ldots) \hspace{1em} essential procedure

These procedures return the maximum or minimum of their arguments.

(\[\begin{align*}
\text{(max 3 4)} & \Rightarrow 4 \hspace{1em} \text{exact} \\
\text{(max 3.9 4)} & \Rightarrow 4.0 \hspace{1em} \text{inexact} \\
\text{Note:} & \hspace{1em} \text{If any argument is inexact, then the result will also be inexact (unless the procedure can prove that the inaccuracy is not large enough to affect the result, which is possible only in unusual implementations). If min or max is used to compare numbers of mixed exactness, and the numerical value of the result cannot be represented as an inexact number without loss of accuracy, then the procedure may report a violation of an implementation restriction.} \\
\text{(+ z1 \ldots)} & \hspace{1em} \text{essential procedure} \\
\text{(* z1 \ldots)} & \hspace{1em} \text{essential procedure} \\
\end{align*}\]

These procedures return the sum or product of their arguments.

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

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

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.

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

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.

(\[\begin{align*}
\text{abs x)} & \hspace{1em} \text{essential procedure} \\
\text{(abs -7)} & \Rightarrow 7 \\
\end{align*}\]

Abs returns the magnitude of its argument.
### Revised Scheme

<table>
<thead>
<tr>
<th>Operation</th>
<th>Procedure</th>
</tr>
</thead>
<tbody>
<tr>
<td>(quotient ( n_1 \ n_2 ))</td>
<td>essential procedure</td>
</tr>
<tr>
<td>(remainder ( n_1 \ n_2 ))</td>
<td>essential procedure</td>
</tr>
<tr>
<td>(modulo ( n_1 \ n_2 ))</td>
<td>essential procedure</td>
</tr>
</tbody>
</table>

These procedures implement number-theoretic (integer) division: For positive integers \( n_1 \) and \( n_2 \), if \( n_3 \) and \( n_4 \) are integers such that \( n_1 = n_2 n_3 + n_4 \) and \( 0 \leq n_4 < n_2 \), then

\[
\begin{align*}
(\text{quotient } n_1 & \ n_2) \quad \Rightarrow \ n_3 \\
(\text{remainder } n_1 & \ n_2) \quad \Rightarrow \ n_4 \\
(\text{modulo } n_1 & \ n_2) \quad \Rightarrow \ n_4
\end{align*}
\]

For integers \( n_1 \) and \( n_2 \) with \( n_2 \) not equal to 0,

\[
(\text{floor } n_1 \ (\star \ n_2 \ (\text{quotient } n_1 \ n_2)))
\]

\[
(\text{remainder } n_1 \ n_2)) \quad \Rightarrow \ \#t
\]

provided all numbers involved in that computation are exact.

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

\[
\begin{align*}
(\text{modulo } 13 & \ 4) \quad \Rightarrow \ 1 \\
(\text{remainder } 13 & \ 4) \quad \Rightarrow \ 1
\end{align*}
\]

\[
\begin{align*}
(\text{modulo } -13 & \ 4) \quad \Rightarrow \ 3 \\
(\text{remainder } -13 & \ 4) \quad \Rightarrow \ -1
\end{align*}
\]

\[
\begin{align*}
(\text{modulo } 13 & \ -4) \quad \Rightarrow \ -3 \\
(\text{remainder } 13 & \ -4) \quad \Rightarrow \ 1
\end{align*}
\]

\[
\begin{align*}
(\text{modulo } -13 & \ -4) \quad \Rightarrow \ -1 \\
(\text{remainder } -13 & \ -4) \quad \Rightarrow \ -1
\end{align*}
\]

\[
\begin{align*}
(\text{remainder } -13 & \ -4.0) \quad \Rightarrow \ -1.0 \ ; \ \text{inexact}
\end{align*}
\]

<table>
<thead>
<tr>
<th>Operation</th>
<th>Procedure</th>
</tr>
</thead>
<tbody>
<tr>
<td>(gcd ( n_1 ))</td>
<td>essential procedure</td>
</tr>
<tr>
<td>(lcm ( n_1 ))</td>
<td>essential procedure</td>
</tr>
</tbody>
</table>

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

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

<table>
<thead>
<tr>
<th>Operation</th>
<th>Procedure</th>
</tr>
</thead>
<tbody>
<tr>
<td>(numerator ( q ))</td>
<td>procedure</td>
</tr>
<tr>
<td>(denominator ( q ))</td>
<td>procedure</td>
</tr>
</tbody>
</table>

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.

\[
\begin{align*}
(\text{numerator } / \ 6 \ 4) \quad \Rightarrow \ 3 \\
(\text{denominator } / \ 6 \ 4) \quad \Rightarrow \ 2 \\
(\text{denominator} \ (\text{exact->inexact } / \ 6 \ 4)) \quad \Rightarrow \ 2.0
\end{align*}
\]

<table>
<thead>
<tr>
<th>Operation</th>
<th>Procedure</th>
</tr>
</thead>
<tbody>
<tr>
<td>(floor ( x ))</td>
<td>essential procedure</td>
</tr>
<tr>
<td>(ceiling ( x ))</td>
<td>essential procedure</td>
</tr>
<tr>
<td>(truncate ( x ))</td>
<td>essential procedure</td>
</tr>
<tr>
<td>(round ( x ))</td>
<td>essential procedure</td>
</tr>
</tbody>
</table>

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

\textbf{Rationale:} \texttt{Round} rounds to even for consistency with the default rounding mode specified by the IEEE floating point standard.

\textbf{Note:} If the argument to one of these procedures is inexact, then the result will also be inexact. If an exact value is needed, the result should be passed to the \texttt{inexact->exact} procedure.

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

\[
(\text{rationalize } x \ y)
\]

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

\[
\begin{align*}
(\text{rationalize} \ (\text{inexact->exact } .3 \ 1/10) \quad \Rightarrow \ 1/3 \ ; \ \text{exact} \\
(\text{rationalize} \ .3 \ 1/10) \quad \Rightarrow \ #i11/3 \ ; \ \text{inexact}
\end{align*}
\]
These procedures are part of every implementation that supports general complex numbers; they compute the usual transcendental functions. \texttt{Log} computes the natural logarithm of \( z \) (not the base ten logarithm). \texttt{Asin}, \texttt{acos},\ and \texttt{atan} compute arcsine (\( \sin^{-1} \)), arccosine (\( \cos^{-1} \)), and arctangent (\( \tan^{-1} \)), respectively. The two-argument variant of \texttt{atan} computes \((\text{angle (make-rectangular } x\ y))\) (see below), even in implementations that don’t support general complex numbers.

In general, the mathematical functions \texttt{log}, arcsine, arccosine, and arctangent are multiply defined. For nonzero real \( x \), the value of \texttt{log} \( x \) is defined to be the one whose imaginary part lies in the range \(-\pi\) (exclusive) to \( \pi\) (inclusive). \texttt{log} 0 is undefined. The value of \texttt{log} \( z \) when \( z \) is complex is defined according to the formula

\[
\text{log} \ z = \text{log}\text{-magnitude}(z) + i \text{ angle}(z)
\]

With \texttt{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 [83], which in turn cites [60]; 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.

\texttt{sqrt} \( z \)

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

\texttt{expt} \( z_1 \ Z_2 \)

Returns \( z_1 \) raised to the power \( z_2 \):

\[
Z_1^{Z_2} = e^{Z_2 \log Z_1}
\]

0\(^0\) is defined to be equal to 1.

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

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

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

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

\texttt{magnitude} \( z \)

\texttt{angle} \( z \)

These procedures are part of every implementation that supports general complex numbers. Suppose \( x_1, x_2, x_3, \) and \( x_4 \) are real numbers and \( z \) is a complex number such that

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

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

\textit{Rationale:} \texttt{Magnitude} is the same as \texttt{abs} for a real argument, but \texttt{abs} must be present in all implementations, whereas \texttt{magnitude} need only be present in implementations that support general complex numbers.

\texttt{exact->inexact} \( z \)

\texttt{inexact->exact} \( z \)

\texttt{Exact->inexact} returns an inexact representation of \( z \). The value returned is the inexact number that is numerically closest to the argument. If an exact argument has no reasonably close inexact equivalent, then a violation of an implementation restriction may be reported.

\texttt{Inexact->exact} returns an exact representation of \( z \). The value returned is the exact number that is numerically closest to the argument. If an inexact argument has no reasonably close exact equivalent, then a violation of an implementation restriction may be reported.

These procedures implement the natural one-to-one correspondence between exact and inexact integers throughout an implementation-dependent range. See section 6.5.3.

6.5.6. Numerical input and output

\texttt{number->string} \( number \)

\texttt{number->string number radix} essential procedure

\texttt{Radix} must be an exact integer, either 2, 8, 10, or 16. If omitted, \texttt{radix} defaults to 10. The procedure \texttt{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

\[
\text{let ((number number) (radix radix)) (eqv? number (string->number (number->string number radix)))}
\]
is true. It is an error if no possible result makes this expression true.

If number is inexact, the radix is 10, and the above expression can be satisfied by a result that contains a decimal point, then the result contains a decimal point and is expressed using the minimum number of digits (exclusive of exponent and trailing zeroes) needed to make the above expression true [89, 10]; otherwise the format of the result is unspecified.

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

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

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

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

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

```
(string->number "100")  =>  100
(string->number "100" 16)  =>  256
(string->number "1.0e2")  =>  100.0
(string->number "15##")  =>  1500.0
```

**Note:** Although string->number is an essential procedure, an implementation may restrict its domain in the following ways. String->number is permitted to return #f whenever string contains an explicit radix prefix. If all numbers supported by an implementation are real, then string->number is permitted to return #f whenever string uses the polar or rectangular notations for complex numbers. If all numbers are integers, then string->number may return #f whenever the fractional notation is used. If all numbers are exact, then string->number may return #f whenever an exponent marker or explicit exactness prefix is used, or if a # appears in place of a digit. If all inexact numbers are integers, then string->number may return #f whenever a decimal point is used.

### 6.6. Characters

Characters are objects that represent printed characters such as letters and digits. Characters are written using the notation #\{character\} or #\{character name\}. For example:

```
#\a   ; lower case letter
#\A   ; upper case letter
#\(   ; left parenthesis
#\   ; the space character
#\space ; the preferred way to write a space
#\newline ; the newline character
```

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

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

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

```
(char? obj)  essential procedure
```

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

```
(char=? char1 char2)  essential procedure
(char?= char1 char2)  essential procedure
(char=> char1 char2)  essential procedure
(char=>? char1 char2)  essential procedure
```

These procedures impose a total ordering on the set of characters. It is guaranteed that under this ordering:

- The upper case characters are in order. For example, (char<? #\A #\B) returns #t.
- The lower case characters are in order. For example, (char<? #\a #\b) returns #t.
- The digits are in order. For example, (char<? #\0 #\9) returns #t.
- Either all the digits precede all the upper case letters, or vice versa.
- Either all the digits precede all the lower case letters, or vice versa.
6. Standard procedures

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

(char-ci=? char1 char2) essential procedure
(char-ci=? char1 char2) essential procedure
(char-ci=? char1 char2) essential procedure
(char-ci=? char1 char2) essential procedure
(char-ci=? char1 char2) essential procedure

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

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

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

(char->integer char) essential procedure
(integer->char n) essential procedure

Given a character, char->integer returns an exact integer representation of the character. Given an exact integer that is the image of a character under char->integer, integer->char returns that character. These procedures implement injective order isomorphisms between the set of characters under the char-ci? ordering and some subset of the integers under the <= ordering. That is, if

(char-ci=? a b) => #t and (<= x y) => #t

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

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

(char-ci=? (integer->char x) (integer->char y)) => #t

(char-upcase char) essential procedure
(char-downcase char) essential procedure

These procedures return a character char2 such that (char-ci=? char char2). In addition, if char is alphabetic, then the result of char-upcase is upper case and the result of char-downcase is lower case.

6.7. Strings

Strings are sequences of characters. Strings are written as sequences of characters enclosed within doublequotes ("). A doublequote can be written inside a string only by escaping it with a backslash (\), as in

"The word "recursion" has many meanings."

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

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

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

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

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

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

(make-string k) essential procedure
(make-string k char) essential procedure

Make-string returns a newly allocated string of length k.
If char is given, then all elements of the string are initialized to char, otherwise the contents of the string are unspecified.

(string char ...) essential procedure
Returns a newly allocated string composed of the arguments.

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

(string-ref string k) essential procedure
k must be a valid index of string. String-ref returns character k of string using zero-origin indexing.
(string-set! string k char) essential procedure
k must be a valid index of string. String-set! stores char in element k of string and returns an unspecified value.

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

(string=? string1 string2) essential procedure
(string-ci=? string1 string2) essential procedure
Returns #t if the two strings are the same length and contain the same characters in the same positions, otherwise returns #f. String-ci=? treats upper and lower case letters as though they were the same character, but string=? treats upper and lower case as distinct characters.

(string<? string1 string2) essential procedure
(string>? string1 string2) essential procedure
(string<=? string1 string2) essential procedure
(string>=? string1 string2) essential procedure
(string-ci<? string1 string2) essential procedure
(string-ci>? string1 string2) essential procedure
(string-ci<=? string1 string2) essential procedure
(string-ci=>? string1 string2) essential procedure
These procedures are the lexicographic extensions to strings of the corresponding orderings on characters. For example, string<? is the lexicographic ordering on strings induced by the ordering char<? on characters. If two strings differ in length but are the same up to the length of the shorter string, the shorter string is considered to be lexicographically less than the longer string.

Implementations may generalize these and the string=? and string-ci=? procedures to take more than two arguments, as with the corresponding numerical predicates.

(substring string start end) essential procedure
String must be a string, and start and end must be exact integers satisfying
0 ≤ start ≤ end ≤ (string-length string).
Substring returns a newly allocated string formed from the characters of string beginning with index start (inclusive) and ending with index end (exclusive).

(string-append string ...) essential procedure
Returns a newly allocated string whose characters form the concatenation of the given strings.

(string->list string) essential procedure
(list->string) essential procedure
String->list returns a newly allocated list of the characters that make up the given string. List->string returns a newly allocated string formed from the characters in the list. String->list and list->string are inverses so far as equal? is concerned.

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

(string-fill! string char) procedure
Stores char in every element of the given string and returns an unspecified value.

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

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

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

#\(\text{(0 2 2 2) "Anna"})

Note that this is the external representation of a vector, not an expression evaluating to a vector. Like list constants, vector constants must be quoted:

'\#\(\text{(0 2 2 2) "Anna"})
\rightarrow #\(\text{(0 2 2 2) "Anna"})

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

(make-vector k) essential 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.
6. Standard procedures

(vector obj ...) essential procedure
(Avec a b c) => #(a b c)

(vector-length vector) essential procedure
Returns the number of elements in vector.

(vector-ref vector k) essential procedure
k must be a valid index of vector. Vector-ref returns the contents of element k of vector.

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

(vector-set! vector k obj) essential procedure
k must be a valid index of vector. Vector-set! stores obj in element k of vector. The value returned by vector-set! is unspecified.

(let ((vec (vector 0 '(2 2 2) "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) essential procedure
(list->vector list) essential procedure
Vector->list returns a newly allocated list of the objects contained in the elements of vector. List->vector returns a newly created vector initialized to the elements of the list list.

(vector->list #'(dah dah didah))
=> (dah dah didah)
(list->vector '(dah dah didah))
=> #'(dah dah didah)

(vector-fill! vector fill) procedure
Stores fill in every element of vector. The value returned by vector-fill! is unspecified.

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

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

(procedure? car) => #t
(procedure? 'car) => #f
(procedure? (lambda (x) (* x x))) => #t
(procedure? '(lambda (x) (* x x))) => #f
(call-with-current-continuation procedure?) => #t

(apply proc args) essential procedure
(apply proc arg1 ... args) procedure
Proc must be a procedure and args must be a list. The first (essential) form calls proc with the elements of args as the actual arguments. The second form is a generalization of the first that 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 ...) essential procedure
The lists must be lists, and proc must be a procedure taking as many arguments as there are lists. If more than one list is given, then they must all be the same length. Map applies proc element-wise to the elements of the lists and returns a list of the results, in order from left to right. The dynamic order in which proc is applied to the elements of the lists is unspecified.

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

(let ((count 0))
  (map (lambda (ignored)
         (set! count (+ count 1))
         count)
       '(a b c))) => unspecified
(for-each proc list1 list2 ...)  essential procedure

The arguments to for-each are like the arguments to map, but for-each calls proc for its side effects rather than for its values. Unlike map, for-each is guaranteed to call proc on the elements of the lists in order from the first element to the last, and the value returned by for-each is unspecified.

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

(force promise)  procedure

Forces the value of promise (see delay, section 4.2.5). If no value has been computed for the promise, then a value is computed and returned. The value of the promise is cached (or “memoized”) so that if it is forced a second time, the previously computed value is returned.

(force (delay (+ 1 2)))  =>  3
(let ((p (delay (+ 1 2))))
  (list (force p) (force p)))  =>  (3 3)

(define a-stream
  (letrec ((next
             (lambda (n)
             (cons n (delay (next (+ n 1))))))
             (next 0)))
  (define head car)
  (define tail
    (lambda (stream) (force (cdr stream))))
  (head (tail a-stream)))  =>  2

Force and delay are mainly intended for programs written in functional style. The following examples should not be considered to illustrate good programming style, but they illustrate the property that only one value is computed for a promise, no matter how many times it is forced.

(define count 0)
(define p
  (delay (begin (set! count (+ count 1))
               (if (> count x)
               count
               (force p))))))

(define x 5)
  p   =>  a promise
  (force p)  =>  6
  p   =>  a promise, still
  (begin (set! x 10)
         (force p))  =>  6

Here is a possible implementation of delay and force. Promises are implemented here as procedures of no arguments, and force simply calls its argument:

(define force
  (lambda (object)
    (object)))

We define the expression

(delay (expression))

to have the same meaning as the procedure call

(make-promise (lambda () (expression)))

where make-promise is defined as follows:

(define make-promise
  (lambda (proc)
    (let ((result-ready? #f)
           (result #f))
      (lambda ()
        (if result-ready?
        result
        (let ((x (proc)))
          (if result-ready?
          result
          (begin (set! result-ready? #t)
                  (set! result x)
                  result))))))))

Rationale: A promise may refer to its own value, as in the last example above. Forcing such a promise may cause the promise to be forced a second time before the value of the first force has been computed. This complicates the definition of make-promise.

Various extensions to this semantics of delay and force are supported in some implementations:

- Calling force on an object that is not a promise may simply return the object.

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

  (eqv? (delay 1) 1)  =>  unspecified
  (pair? (delay (cons 1 2)))  =>  unspecified

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

  (+ (delay (* 3 7)) 13)  =>  34

(call-with-current-continuation proc)  essential procedure

Proc must be a procedure of one argument. The procedure call-with-current-continuation packages up the current continuation (see the rationale below) as an “escape procedure” and passes it as an argument to proc. The escape procedure is a Scheme procedure of one argument
that, if it is later passed a value, will ignore whatever continuation
is in effect at that later time and will give the value instead to the continuation that was in effect when the escape procedure was created.

The escape procedure that is passed to proc has unlimited extent just like any other procedure in Scheme. It may be stored in variables or data structures and may be called as many times as desired.

The following examples show only the most common uses of call-with-current-continuation. If all real programs were as simple as these examples, there would be no need for a procedure with the power of call-with-current-continuation.

(call-with-current-continuation
  (lambda (exit)
    (for-each (lambda (x)
                  (if (negative? x)
                      (exit x)))
               '(54 0 37 -3 245 19))
  #t)) => -3

(define list-length
  (lambda (obj)
    (call-with-current-continuation
      (lambda (return)
        (letrec ((r
                      (lambda (obj)
                        (cond ((null? obj) 0)
                          ((pair? obj)
                           (+ (r (cdr obj)) 1))
                          (else (return #f)))))
                      (r obj))))))

(list-length '(1 2 3 4)) => 4
(list-length '(a b . c)) => #f

Rationale:

A common use of call-with-current-continuation is for structured, non-local exits from loops or procedure bodies, but in fact call-with-current-continuation is extremely useful for implementing a wide variety of advanced control structures. Whenever a Scheme expression is evaluated there is a continuation wanting the result of the expression. The continuation represents an entire (default) future for the computation. If the expression is evaluated at top level, for example, then the continuation might take the result, print it on the screen, prompt for the next input, evaluate it, and so on forever. Most of the time the continuation includes actions specified by user code, as in a continuation that will take the result, multiply it by the value stored in a local variable, add seven, and give the answer to the top level continuation to be printed. Normally these ubiquitous continuations are hidden behind the scenes and programmers don’t think much about them. On rare occasions, however, a programmer may need to deal with continuations explicitly. Call-with-current-continuation allows Scheme programmers to do that by creating a procedure that acts just like the current continuation.

Most programming languages incorporate one or more special-purpose escape constructs with names like exit, return, or even goto. In 1965, however, Peter Landin [55] invented a general purpose escape operator called the J-operator. John Reynolds [69] described a simpler but equally powerful construct in 1972. The catch special form described by Sussman and Steele in the 1975 report on Scheme is exactly the same as Reynolds’s construct, though its name came from a less general construct in MacLisp. Several Scheme implementors noticed that the full power of the catch construct could be provided by a procedure instead of by a special syntactic construct, and the name call-with-current-continuation was coined in 1982. This name is descriptive, but opinions differ on the merits of such a long name, and some people use the name call/cc instead.

6.10. Input and output

6.10.1. Ports

Ports represent input and output devices. To Scheme, an input port is a Scheme object that can deliver characters upon command, while an output port is a Scheme object that can accept characters.

(call-with-input-file string proc) essential procedure
(call-with-output-file string proc) essential procedure

Proc should be a procedure of one argument, and string should be a string naming a file. For call-with-input-file, the file must already exist; for call-with-output-file, the effect is unspecified if the file already exists. These procedures call proc with one argument: the port obtained by opening the named file for input or output. If the file cannot be opened, an error is signalled. If the procedure returns, then the port is closed automatically and the value yielded by the procedure is returned. If the procedure does not return, then the port will not be closed automatically unless it is possible to prove that the port will never again be used for a read or write operation.

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

(input-port? obj) essential procedure
(output-port? obj) essential procedure

Returns #t if obj is an input port or output port respectively, otherwise returns #f.
Returns the current default input or output port.

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

Thunk must be a procedure of no arguments, and string must be a string naming a file. For with-input-from-file, the file must already exist; for with-output-to-file, the effect is unspecified if the file already exists. The file is opened for input or output, an input or output port connected to it is made the default value returned by current-input-port or current-output-port, and the thunk is called with no arguments. When the thunk returns, the port is closed and the previous default is restored. With-input-from-file and with-output-to-file return the value yielded by thunk. If an escape procedure is used to escape from the continuation of these procedures, their behavior is implementation dependent.

(open-input-file filename) essential procedure
Takes a string naming an existing file and returns an input port capable of delivering characters from the file. If the file cannot be opened, an error is signalled.

(open-output-file filename) essential procedure
Takes a string naming an output file to be created and returns an output port capable of writing characters to a new file by that name. If the file cannot be opened, an error is signalled. If a file with the given name already exists, the effect is unspecified.

(close-input-port port) essential procedure
(close-output-port port) essential procedure
Closes the file associated with port, rendering the port incapable of delivering or accepting characters. These routines have no effect if the file has already been closed. The value returned is unspecified.

6.10.2. Input

(read) essential procedure
(read port) essential procedure
Read converts external representations of Scheme objects into the objects themselves. That is, it is a parser for the nonterminal (datum) (see sections 7.1.2 and 6.3). Read returns the next object parsable from the given input port, updating port to point to the first character past the end of the external representation of the object.

If an end of file is encountered in the input before any characters are found that can begin an object, then an end of file object is returned. The port remains open, and further attempts to read will also return an end of file object. If an end of file is encountered after the beginning of an object’s external representation, but the external representation is incomplete and therefore not parsable, an error is signalled.

The port argument may be omitted, in which case it defaults to the value returned by current-input-port. It is an error to read from a closed port.

(read-char) essential procedure
(read-char port) essential procedure
Returns the next character available from the input port, updating the port to point to the following character. If no more characters are available, an end of file object is returned. Port may be omitted, in which case it defaults to the value returned by current-input-port.

(peek-char) essential procedure
(peek-char port) essential procedure
Returns the next character available from the input port, without updating the port to point to the following character. If no more characters are available, an end of file object is returned. Port may be omitted, in which case it defaults to the value returned by current-input-port.

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

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

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

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

### 6.10.3. Output

```lisp
(write obj) (write obj port) 
```

```lisp
(display obj) (display obj port) 
```

### 6.10.4. 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.

```lisp
(load filename) (transcript-on filename) (transcript-off) 
```

*Filename* should be a string naming an existing file containing Scheme source code. The load procedure reads expressions and definitions from the file and evaluates them sequentially. It is unspecified whether the results of the expressions are printed. The load procedure does not affect the values returned by current-input-port and current-output-port. Load returns an unspecified value.

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

```lisp
(newline) (newline port) (write-char char) (write-char char port) 
```

*Rationale:* Write is intended for producing machine-readable output and display is for producing human-readable output. Implementations that allow “slashification” within symbols will probably want write but not display to slashify funny characters in symbols.
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. The syntax for the entire language, including features which are not essential, is given here.

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

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

7.1.1. **Lexical structure**

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

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

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

(token) → (identifier) | (boolean) | (number)
 | (character) | (string)
 | ( ) | # | ; | . | .
(delimiter) → (whitespace) | ( ) | " | ;
(whitespace) → (space or newline)
(comment) → ; (all subsequent characters up to a line break)
( atmosphere) → (whitespace) | (comment)
(intertoken space) → (atmosphere)*

(identifier) → (initial) (subsequent)*
 | (peculiar identifier)
(initial) → (letter) | (special initial)
(letter) → a | b | c | . . . | z
(special initial) → ! | $ | % | & | * | / | : | < | = | > | ? | " | ‘ | ‘
(subsequent) → (initial) | (digit)
 | (special subsequent)
(digit) → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
(special subsequent) → + | -
(peculiar identifier) → + | - | . . .
(syntactic keyword) → (expression keyword)
 | else | => | define
 | unquote | unquote-splicing
(expression keyword) → quote | lambda | if

| set! | begin | cond | and | or | case
| let | let* | letrec | do | delay
| quasiquote

(variable) → (any (identifier) that isn’t also a (syntactic keyword))

(boolean) → #t | #f
(character) → #\ (any character)
 | #\ (character name)
(character name) → space | newline

(string) → " (string element)* "
(string element) → (any character other than " or \)
 | \ " | \ \n(number) → (num 2) | (num 8)
 | (num 10) | (num 16)

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

(num R) → (prefix R) (complex R)
(complex R) → (real R) | (real R) \ (real R)
 | (real R) + (real R) i | (real R) - (real R) i
 | (real R) + i | (real R) - i
 | + (real R) i | - (real R) i | + i | - i
(real R) → (sign) (real R)
(ureal R) → (integer R)
 | (integer R) / (integer R)
 | (decimal R)
(decimal 10) → (integer 10) (suffix)
 | . (digit 10)+ \ #* (suffix)
 | (digit 10)+ . (digit 10)+ #* (suffix)
 | (digit 10)+ #* . #* (suffix)
(integer R) → (digit R)+ #*
(prefix R) → (radix R) (exactness)
 | (exactness) (radix R)

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

7.1.2. External representations

(Datum) is what the read procedure (section 6.10.2) successfully parses. Note that any string that parses as an expression will also parse as a datum.

(datum) → (simple datum) | (compound datum)
(simple datum) → (boolean) | (number)
  | (character) | (string) | (symbol)
(symbol) → (identifier)
(compound datum) → (list) | (vector)
(list) → ((datum)* ) | ((datum) + . (datum))
  | (abbreviation)
(abbreviation) → (abbrev prefix) (datum)
(abbrev prefix) → * | ` | . | .
(vector) → #((datum)*)

7.1.3. Expressions

(expression) → (variable)
  | (literal)
  | (procedure call)
  | (lambda expression)
  | (conditional)
  | (assignment)
  | (derived expression)

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

(lambda expression) → (lambda (formals) (body))
(formals) → (variable)* | (variable)
  | ((variable)* + . (variable))
(body) → (definition)* (sequence)
(sequence) → (command)* (expression)
(command) → (expression)

(conditional) → (if (test) (consequent) (alternate))
(test) → (expression)
(consequent) → (expression)
(alternate) → (expression) | (empty)

(assignment) → (set! (variable) (expression))

(derived expression) →
  | (cond (cond clause)+)
  | (cond (cond clause)+ (else (sequence)))
  | (case (expression)
    (case clause)+)
  | (case (expression)
    (case clause)*)
  | (else (sequence)))
  | (and (test)+)
  | (or (test)+)
  | (let ((binding spec)* ) (body))
  | (let (variable) (binding spec*) (body))
  | (let* ((binding spec)* ) (body))
  | (letrec (binding spec*) (body))
  | (begin (sequence))
  | (do (iteration spec)*
    (test) (sequence)
    (command)*)
  | (delay (expression))
  | (quasiquote)
  | (cond clause) → ((test) (sequence))
  | ((test))
  | ((test) => (recipient))
  | (recipient) → (expression)
  | (case clause) → (((datum)*) (sequence))
  | (binding spec) → (variable) (expression)
  | (iteration spec) → (variable) (init) (step)
  | (variable) (init)
  | (init) → (expression)
  | (step) → (expression)

7.1.4. Quasiquote

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

(quasiquote) → (quasiquote 1)
(template 0) → (expression)
(quasiquote D) → *(template D)
  | (quasiquote (template D))
(template D) → (simple datum)
  | (list template D)
  | (vector template D)
  | (unquotation D)
(list template D) → ((template or splice D)+)
  | ((template or splice D)+ . (template D))
  | *(template D)
  | *(quasiquote D + 1)
(vector template D) → #((template or splice D)+)
(unquotation D) → , (template D - 1)
  | (unquote (template D - 1))
(template or splice D) → (template D)
  | (splicing unquotation D)
(splicing unquotation D) → , (template D - 1)
  | (unquote-splicing (template D - 1))
In (quasi-quotations), a (list template $D$) can sometimes be confused with either an (unquotation $D$) or a (splicing unquotation $D$). The interpretation as an (unquotation) or (splicing unquotation $D$) takes precedence.

### 7.1.5. Programs and definitions

$$\langle \text{program} \rangle \rightarrow \langle \text{command or definition} \rangle^*$$

$$\langle \text{command or definition} \rangle \rightarrow \langle \text{command} \rangle \mid \langle \text{definition} \rangle$$

$$\langle \text{definition} \rangle \rightarrow \langle \text{define} \langle \text{variable} \rangle \langle \text{expression} \rangle \rangle$$

$$\langle \text{def forms} \rangle \rightarrow \langle \text{variable} \rangle^*$$

$$\langle \text{variable} \rangle^+$$

### 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 [93]; the notation is summarized below:

- $\langle \ldots \rangle$ sequence formation
- $s \upharpoonright k$ $k$th member of the sequence $s$ (1-based)
- $\#s$ length of sequence $s$
- $s \upharpoonright t$ concatenation of sequences $s$ and $t$
- $s \upharpoonright k$ drop the first $k$ members of sequence $s$
- $t \rightarrow a, b$ McCarthy conditional “if $t$ then $a$ else $b$”
- $\rho[x/i]$ substitution “$\rho$ with $x$ for $i$”
- $x \in D$ injection of $x$ into domain $D$
- $x \downarrow D$ projection of $x$ to domain $D$

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

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

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

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

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

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

$$E[(\lambda (I^*)P') \langle \text{undefined} \rangle \ldots]$$

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

#### 7.2.1. Abstract syntax

$$\text{K} \in \text{Con} \quad \text{constants, including quotations}$$

$$\text{I} \in \text{Id} \quad \text{identifiers (variables)}$$

$$E \in \text{Exp} \quad \text{expressions}$$

$$\Gamma \in \text{Com} = \text{Exp} \quad \text{commands}$$

$$\text{Exp} \rightarrow \text{K} \mid \text{I} \mid (E_0 E^*)$$

$$\mid (\lambda (I^*)I^*E_0)$$

$$\mid (\lambda (I^*)I^*E_0)$$

$$\mid (\text{if } E_0 E_1 E_2) \mid (\text{if } E_0 E_1)$$

$$\mid (\text{set! } I E)$$

#### 7.2.2. Domain equations

$$\alpha \in L \quad \text{locations}$$

$$\nu \in N \quad \text{natural numbers}$$

$$T = \{\text{false}, \text{true}\} \quad \text{booleans}$$

$$Q \quad \text{symbols}$$

$$H \quad \text{characters}$$

$$R \quad \text{numbers}$$

$$E_0 = L \times L \times T \quad \text{pairs}$$

$$E_0 = L \times T \quad \text{vectors}$$

$$E_0 = L \times T \quad \text{strings}$$

$$M = \{\text{false}, \text{true}, \text{null}, \text{undefined}, \text{unspecified}\} \quad \text{miscellaneous}$$

$$\phi \in F = L \times (E^* \rightarrow K \rightarrow C) \quad \text{procedure values}$$

$$e \in E = Q \times H + R + E_0 + E_v + E_0 + M + F \quad \text{expressed values}$$

$$\sigma \in S = L \rightarrow (E \times T) \quad \text{stores}$$

$$\rho \in U = \text{Id} \rightarrow L \quad \text{environments}$$

$$\theta \in C = S \rightarrow A \quad \text{command continuations}$$

$$\kappa \in K = E^* \rightarrow C \quad \text{expression continuations}$$

$$A \quad \text{answers}$$

$$X \quad \text{errors}$$

#### 7.2.3. Semantic functions

$\mathcal{K} : \text{Con} \rightarrow E$

$\mathcal{E} : \text{Exp} \rightarrow U \rightarrow K \rightarrow C$

$\mathcal{E}^* : \text{Exp}^* \rightarrow U \rightarrow K \rightarrow C$

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

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

$$\mathcal{E}[\mathcal{K}] = \lambda \rho \kappa : \text{send}(\mathcal{K}[\mathcal{K}])\kappa$$
\[ \mathcal{E} = \lambda \rho \cdot \text{hold (lookup } \rho \text{)} \]
\[
\left( \begin{array}{l}
\text{single}(\lambda e. \ c = \text{undefined} \rightarrow \\
\text{wrong "undefined variable",} \\
\text{send } e \kappa) \\
\end{array} \right)
\]
\[
\mathcal{E}[(E_0 \ E^* \ E^*)] = \\
\lambda \rho \cdot \mathcal{E}^*(\text{permute}(E_0) \ E^*) \\
\rho \\
(\lambda e^* \cdot (\text{compose}(e^* \ | \ 1) (e^* \ | \ 1) \ k) \\
(\text{unpermute } e^*)))
\]
\[
\mathcal{E}[(\text{lambda } (I^* . I^* \ E_0)] = \\
\lambda \rho \cdot \text{L} . \sigma \\
\text{new } \sigma \in \text{L} \rightarrow \\
\text{send } ((\text{new } \sigma \ | \ L) \\
\lambda e^* \cdot \#e^* = \#I^* \rightarrow \\
\text{tceval}(\lambda \alpha . \ \mathcal{E}[\Gamma]^* \rho (\mathcal{E}[E_0] \rho \kappa')) \\
(\text{extend } \rho \ | \ 1 \ k) \\
e^* \\
(\#I^*), \\
\text{wrong "out of memory" } \sigma)
\]
\[
\mathcal{E}[(\text{lambda } (I^* . I^* \ E_0)] = \\
\lambda \rho \cdot \text{L} . \sigma \\
\text{new } \sigma \in \text{L} \rightarrow \\
\text{send } ((\text{new } \sigma \ | \ L) \\
\lambda e^* \cdot \#e^* = \#I^* \\
\text{tcevalrst} \\
(\lambda \alpha . \ (\lambda \rho' . \ \mathcal{E}[\Gamma]^* \rho'(\mathcal{E}[E_0] \rho' \kappa')) \\
(\text{extend } \rho \ | \ 1 \ k) \\
e^* \\
(\#I^*), \\
\text{wrong "out of memory" } \sigma)
\]

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

\[
\mathcal{E}[(\text{set! } I \ E)] = \\
\lambda \rho \cdot \mathcal{E}[I] \rho (\text{single}(\lambda e. \ \text{assign (lookup } \rho \text{)}) \\
(\text{send unspecified } e \kappa))
\]
\[
\mathcal{E}[] = \lambda \rho \cdot \text{L} . \kappa \}
\]
\[
\mathcal{E}[] = \lambda \rho \cdot \mathcal{E}[E] \rho (\text{single}(\lambda e. \ \text{assign (lookup } \rho \text{)} \\
(\text{send unspecified } e \kappa))
\]

7. Formal syntax and semantics

7.2.4. Auxiliary functions

\text{lookup}: U \rightarrow \text{Idc} \rightarrow L \\
\text{lookup} = \lambda \rho \cdot \rho \text{L}

\text{extends}: U \rightarrow \text{Idc} \rightarrow L^* \rightarrow U \\
\text{extends} = \\
\lambda \rho \alpha . \#I^* = 0 \rightarrow \rho, \\
(\text{extend } \rho \ | \ 1 \ k) \\
\text{wrong } \text{number of arguments}

\text{send}: E \rightarrow K \rightarrow C \\
\text{send} = \lambda \text{ek} . \text{L} \kappa

\text{single} : (E \rightarrow C) ightarrow K \\
\text{single} = \\
\lambda \text{ek} . \#e^* = 1 \rightarrow \psi(e^* \ | \ 1) \\
\text{wrong "out of memory" } \sigma

\text{new} : S \rightarrow (L + \{ \text{error} \}) \\
\text{new} = \lambda \text{ek} . \text{send } (\sigma \ | \ 1) \kappa

\text{assign} : L \rightarrow E \rightarrow C \\
\text{assign} = \lambda \text{ek} . \theta \text{update } \alpha \sigma

\text{update} : L \rightarrow E \rightarrow S \\
\text{update} = \lambda \text{ek} . \text{L} \kappa

\text{tceval}: (L^* \rightarrow C) \rightarrow E^* \rightarrow C \\
\text{tceval} = \\
\lambda \text{ek} \psi \cdot \#e^* = 0 \rightarrow \psi(e^* \ | \ 1) \\
(\text{new } \sigma \in \text{L} \rightarrow \text{tceval}(\lambda \alpha . \ \psi((\text{new } \sigma \ | \ L) \ | \ 1 \ k) \\
(\#I^*), \\
\text{wrong "out of memory" } \sigma

\text{tcevalrst}: (L^* \rightarrow C) \rightarrow E^* \rightarrow C \\
\text{tcevalrst} = \\
\lambda \text{ek} \psi \cdot \text{list } \text{dropfirst } e^* \psi \\
(\text{single}(\lambda e . \ \text{tceval } ((\text{takefirst } e^* \psi) \ | \ 1)) \\
\text{dropfirst} = \lambda n . n = 0 \rightarrow I, \text{dropfirst } (I \ | \ 1) \rightarrow n \\
\text{takefirst} = \lambda n . n = 0 \rightarrow I, \text{takefirst } (I \ | \ 1) \rightarrow n - 1 \\
\text{truish} : E \rightarrow T \\
\text{truish} = \lambda e . \ e = \text{false} \rightarrow \text{false, true}

\text{permute} : \text{Exp}^* \rightarrow \text{Exp}^* \\
\text{permute = } \lambda \text{ek} \cdot \text{false } \rightarrow \text{false, true}

\text{unpermute} : E^* \rightarrow E^* \\
\text{unpermute = } \lambda \text{ek} \cdot \text{E} \rightarrow \text{E}

\text{applicate} : E \rightarrow E^* \rightarrow K \rightarrow C \\
\text{applicate} = \\
\lambda \text{ek} \cdot \text{E} \rightarrow \text{false, true}

\text{oncarg} : (E \rightarrow K \rightarrow C) \rightarrow (E^* \rightarrow K \rightarrow C) \\
\text{oncarg} = \\
\lambda \text{ek} \cdot \text{E} \rightarrow \text{false, true}

\text{twoarg} : (E \rightarrow E \rightarrow K \rightarrow C) \rightarrow (E^* \rightarrow K \rightarrow C) \\
\text{twoarg} = \\
\lambda \text{ek} \cdot \text{E} \rightarrow \text{false, true}
7.3. Derived expression types

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

\[(\text{cond} \,(\text{test} \,(\text{sequence})) \quad \Rightarrow \quad (\text{clause}_2 \ldots))\]

\[(\text{cond} \,(\text{test}) \,(\text{clause}_2 \ldots)) \quad \Rightarrow \quad (\text{if} \,(\text{test}) \quad (\text{clause}_2 \ldots))\]

\[(\text{cond} \,(\text{test} \Rightarrow \text{recipient})) \quad \Rightarrow \quad (\text{clause}_2 \ldots)\]

\[(\text{let} \,(\text{test-result} \,(\text{test}))) \quad \Rightarrow \quad (\text{thunk2} \,(\text{lambda} () \,(\text{recipient})))\]

\[(\text{thunk3} \,(\text{lambda} () \,(\text{cond} \,(\text{clause}_2 \ldots)))\,)\]

\[(\text{if} \,(\text{test-result}) \quad (\text{thunk2} \,(\text{test-result})) \quad (\text{thunk3}))\]
(cond (else (sequence)))
≡ (begin (sequence))

(cond)
≡ (some expression returning an unspecified value)

(case (key)
  ((d1 ... ) (sequence))
  ...
≡ (let ((key (key))
           (thunk1 (lambda () (sequence)))
           ...
           (cond (((memv) key '(d1 ... )) (thunk1))
                  ...
           )))

(case (key)
  ((d1 ... ) (sequence))
  ...
  (else f1 f2 ... ))
≡ (let ((key (key))
            (thunk1 (lambda () (sequence)))
            (elsethunk (lambda () f1 f2 ... ))
            (cond (((memv) key '(d1 ... )) (thunk1))
                   ...
                   (else (elsethunk)))))

where (memv) is an expression evaluating to the memv procedure.

(and) ≡ #t
(and (test)) ≡ (test)
(and (test1) (test2) ... )
≡ (let ((x (test1))
          (thunk (lambda () (and (test2) ... ))))
     (if x (thunk) x))

(or) ≡ #f
(or (test)) ≡ (test)
(or (test1) (test2) ... )
≡ (let ((x (test1))
          (thunk (lambda () (or (test2) ... ))))
     (if x (thunk) x))

(let (((variable1) (init1)) ...)
  (body))
≡ ((lambda (variable1) ... ) (body) (init1) ...)

(let* () (body))
≡ ((lambda () (body)))
(let* (((variable1) (init1))
       ((variable2) (init2))
       ...)
  (body))
≡ (let (((variable1) (init1)))
     (let* (((variable2) (init2))
            ...
            (body))
     (body))

(letrec (((variable1) (init1)) ...)
  (body))
≡ (letrec (((variable1) (init1))
            (((variable2) (init2))
             ...
             (body)))
    (body))

7. Formal syntax and semantics
37

(let ((temp1) (init1))
  ...
  (set! (variable1) (temp1))
  ...
) (body))

where (temp1), (temp2), ... are variables, distinct from (variable1), ..., that do not free occur in the original (init) expressions, and (undefined) is an expression which returns something that when stored in a location makes it an error to try to obtain the value stored in the location. (No such expression is defined, but one is assumed to exist for the purposes of this rewrite rule.) The second let expression in the expansion is not strictly necessary, but it serves to preserve the property that the (init) expressions are evaluated in an arbitrary order.

(begin (sequence))
≡ ((lambda () (sequence)))

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

(begin (expression)) ≡ (expression)
(begin (command) (sequence))
≡ ((lambda (ignore thunk) (thunk))
  (command)
  (lambda () (begin (sequence))))

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

(do (((variable1) (init1)) (step1))
    ...
  (test) (sequence))
  (command1) ... )
≡ (letrec (((loop)
      (lambda (variable1) ...)
      (if (test)
        (begin (sequence))
        (begin (command1))
        ...
        (loop (step1) ...))))
    (loop) (init1) ... ))

where (loop) is any variable which is distinct from (variable1), ..., and which does not occur free in the do expression.

(let (variable0) (((variable1) (init1)) ...)
  (body))
≡ (((letrec (((variable0) (lambda (variable1) ...)
                 (variable2) (init2))
                  ...
                  (body)))
     (body))
     (letrec (((variable1) (init1)) ...)
     (body))
\begin{itemize}
  \item The specification of the \texttt{force} procedure has been weakened. The previous specification was unimplementable.
  \item Variables removed: \texttt{t}, \texttt{nil}.
  \item Procedures removed: \texttt{approximate}, \texttt{last-pair}.
  \item Procedures added: \texttt{list?}, \texttt{peek-char}.
  \item Syntaxes made essential: \texttt{case}, \texttt{and}, \texttt{or}, \texttt{quasiquote}.
  \item Procedures made essential:
  \begin{verbatim}
  reverse char-ci=? make-string
  max char-ci<? string-set!
  min char-ci>? string-ci<?
  modulo char-ci>=? string-ci>?
  gcd char-ci>=? string-ci>=?
  lcm char-alphabetic? string-ci=?
  floor char-numeric? string-ci>?
  ceiling char-whitespace? string-append
  truncate char-lower-case? open-input-file
  round char-upper-case? open-output-file
  number->string char-upcase close-input-port
  string->number char-downcase close-output-port
  \end{verbatim}
  \item Procedures required to accept more general numbers of arguments: \texttt{append}. +, *, - (one argument), / (one argument), \texttt{=}, \texttt{<}, \texttt{>, <>, >=}, \texttt{map}, \texttt{for-each}.
  \item A macro facility has been added as an appendix to this report.
\end{itemize}
**EXAMPLE**

Integrate-system integrates the system

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

of differential equations with the method of Runge-Kutta.

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

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

```scheme
(define integrate-system
  (lambda (system-derivative initial-state h)
    (letrec ((states
      (cons initial-state
        (delay (map-streams next
          states)))))
      (runge-kutta-4
        (lambda (h)
          (let ((h (scale-vector h))
            (*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))))))))
      (runge-kutta-4
        (lambda (f s)
          (cons (f (head s))
            (delay (map-streams f (tail s)))))))))
      (define map-streams
        (lambda (f s)
          (cons (f (head s))
            (delay (map-streams f (tail s))))))
      (define head car)
      (define tail cdr)
      (letrec ((states
        (cons initial-state
          (delay (map-streams next
            states)))))
        (runge-kutta-4
          (lambda (h)
            (let ((h (scale-vector h))
              (*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 damped-oscillator
        (lambda (R L C)
          (let ((Vc (vector-ref state 0))
            (Il (vector-ref state 1)))
            (vector (~ 0 (+ / Vc (* R C)) (/ Il C)))
            (/ Vc L)))))
      (define the-states
        (integrate-system
          (damped-oscillator 10000 1000 .001)
          '(#(1 0)
          .01)))
      (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))
          (vector-length (car vectors))
          (lambda (i)
            (apply f
              (map (lambda (v) (vector-ref v i))
              vectors)))))))))
```
APPENDIX: MACROS

This appendix describes an extension to Scheme that allows programs to define and use new derived expression types. A derived expression type that has been defined using this extension is called a macro.

Derived expression types introduced using this extension have the syntax

\[(\text{keyword}) ~(\text{datum})*\]

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

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

The extension described here consists of three parts:

- A set of expressions used to establish that certain identifiers are macro keywords, associate them with macro transformers, and control the scope within which a macro is defined,
- a convenient pattern language that makes it easy to write transformers for most macros, and
- a compatible low-level macro facility for writing macro transformers that cannot be expressed by the pattern language.

With this extension, there are no reserved identifiers. The syntactic keyword of a macro may shadow variable bindings, and local variable bindings may shadow keyword bindings. All macros defined using the pattern language are "hygienic" and "referentially transparent":

- If a macro transformer inserts a binding for an identifier (variable or keyword), the identifier will in effect be renamed throughout its scope to avoid conflicts with other identifiers.
- If a macro transformer inserts a free reference to an identifier, the reference refers to the binding that was visible where the transformer was specified, regardless of any local bindings that may surround the use of the macro.

This appendix is divided into three major sections. The first section describes the expressions and definitions used to introduce macros, i.e. to bind identifiers to macro transformers.

The second section describes the pattern language. This pattern language is sufficient to specify most macro transformers, including those for all the derived expression types from section 4.2. The primary limitation of the pattern language is that it is thoroughly hygienic, and thus cannot express macros that bind identifiers implicitly.

The third section describes a low-level macro facility that could be used to implement the pattern language described in the second section. This low-level facility is also capable of expressing non-hygienic macros and other macros whose transformers cannot be described by the pattern language, and is important as an example of a more powerful facility that can co-exist with the high-level pattern language.

The particular low-level facility described in the third section is but one of several low-level facilities that have been designed and implemented to complement the pattern language described in the second section. The design of such low-level macro facilities remains an active area of research, and descriptions of alternative low-level facilities will be published in subsequent documents.

Binding syntactic keywords

Define-syntax, let-syntax, and letrec-syntax are analogous to define, let, and letrec, but they bind syntactic keywords to macro transformers instead of binding variables to locations that contain values. Furthermore, there is no define-syntax analogue of the internal definitions described in section 5.2.2.

Rationale: As discussed below, the syntax and scope rules for definitions give rise to syntactic ambiguities when syntactic keywords are not reserved. Further ambiguities would arise if define-syntax were permitted at the beginning of a (body), with scope rules analogous to those for internal definitions.

These new expression types and the pattern language described in section 7.3 are added to Scheme by augmenting the BNF in section 7.1 with the following new productions.

Note that the identifier ... used in some of these productions is not a metasymbol.

\[(\text{expression}) \rightarrow (\text{macro use})
  \mid (\text{macro block})\]

\[(\text{macro use}) \rightarrow ((\text{keyword}) (\text{datum})* )\]

\[(\text{keyword}) \rightarrow (\text{identifier} )\]

\[(\text{macro block}) \rightarrow
  (\text{let-syntax} ((\text{syntax spec})* ) (\text{body} )
  \mid (\text{letrec-syntax} ((\text{syntax spec})* ) (\text{body} )
  (\text{syntax spec}) \rightarrow ((\text{keyword}) (\text{transformer spec} )
  (\text{transformer spec}) \rightarrow
  (\text{syntax-rule} (\text{identifier} )) (\text{syntax rule} )
  (\text{syntax rule}) \rightarrow ((\text{pattern} ) (\text{template} )
  (\text{pattern}) \rightarrow (\text{pattern identifier} )
  \mid ((\text{pattern} ))
  \mid ((\text{pattern} ) . (\text{pattern} ))
  \mid ((\text{pattern} )^{\cdot} (\text{pattern} ) (\text{ellipsis} ) )\]
(template) → (pattern identifier)
| (template element)*
| (template element)* . (template)
| (template element)
| (template)
| (ellipses)
| (pattern element)
| (string)
| (character)
| (boolean)
| (number)

(command or definition) → (syntax definition)
(syntax definition) →
  (define-syntax (keyword) (transformer spec))
| (begin (syntax definition)*)

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

(define define 3)

(begin (define begin list))

(let-syntax
  (((foo syntax-rules ()
      ((foo (proc args ...) body ...)
        (define proc
          (lambda (args ...)
            body ...))))

  (let ((x 3))
    (foo (plus x y) (+ x y))
    (define foo x)
    (plus foo x))

(let-syntax (bindings) (body)) syntax

Syntax: (Bindings) should have the form

(((keyword) (transformer spec) ...)

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

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

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

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

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

(letrec-syntax (bindings) (body)) syntax

Syntax: Same as for let-syntax.

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

(letrec-syntax
  ((or (syntax-rules ()
        ((or) #f)
        ((or e) e)
        ((or e1 e2 ...) (let ((temp e1))
                         (if temp
                            temp
                            (or e2 ...))))))

  (let ((x #f))
    (y ?)
    (temp 8)
    (let odd?)
    (if even?))

  (or x
    (let temp
      (if y)
      y)))
  ⇒ 7

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

Syntax: The (keyword) is an identifier, and the (transformer spec) should be an instance of syntax-rules.

Semantics: The top-level syntactic environment is extended by binding the (keyword) to the specified transformer.
(define-syntax let*
  (syntax-rules ()
    ((let* () body1 body2 ...) (let () body1 body2 ...)
     (let* ((name1 val1) (name2 val2) ...) body1 body2 ...)
     (let ((name1 val1))
       ((name2 val2) ...)
       body1 body2 ...)))))

Pattern language

(syntax-rules (literals) (syntax rule) ...)

Syntax: (Literals) is a list of identifiers, and each (syntax rule) should be of the form

   ((pattern) (template))

where the (pattern) and (template) are as in the grammar above.

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.

Each pattern begins with the keyword for the macro. This keyword is not involved in the matching and is not considered a pattern variable or literal identifier.

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

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

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

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

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

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

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

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

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

(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 ...) body1 body2 ...))
       tag)
val ...))))

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

  ((cond (test -> result) clause1 clause2 ...) 
    (let ((temp test))
        (if temp
        (result temp)
            (cond clause1 clause2 ...))))

    ((cond (test) test)
    ((cond (test) clause1 clause2 ...) 
      (or test (cond clause1 clause2 ...)))
    ((cond (test result1 result2 ...))
      (if test (begin result1 result2 ...))
      (cond result1 result2 ...
        (if test
          (begin result1 result2 ...) 
          (cond clause1 clause2 ...))))

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

The last example is not an error because the local variable
=> is renamed in effect, so that its use is distinct from uses of the top level identifier => that the transformer for cond
looks for. Thus, rather than expanding into

  (let ((-> #f))
    (let ((temp #t))
        (if temp ('ok temp))))

which would result in an invalid procedure call, it expands
instead into

  (let ((-> #f))
    (if #t (begin -> 'ok)))

A compatible low-level macro facility

Although the pattern language provided by syntax-rules
is the preferred way to specify macro transformers, other
low-level facilities may be provided to specify more complex
macro transformers. In fact, syntax-rules can itself be
defined as a macro using the low-level facilities described
in this section.

The low-level macro facility described here introduces
syntax as a new syntactic keyword analogous to quote,
and allows a (transformer spec) to be any expression. This
is accomplished by adding the following two productions to
the productions in section 7.1 and in section 7.3 above.

  (expression) —— (syntax (datum))
  (transformer spec) —— (expression)

The low-level macro system also adds the following procedures:

  unwrap-syntax identifier->symbol
  identifier? generate-identifier
  free-identifier? construct-identifier
  bound-identifier?

Evaluation of a program proceeds in two logical steps. First
the program is converted into an intermediate language via
macro-expansion, and then the result of macro expansion
is evaluated. When it is necessary to distinguish the second
stage of this process from the full evaluation process, it is
referred to as “execution.”

Syntax definitions, either lexical or global, cause an identifier
to be treated as a keyword within the scope of the binding. The keyword is associated with a transformer,
which may be created implicitly using the pattern language
of syntax-rules or explicitly using the low-level facilities
described below.

Since a transformer spec must be fully evaluated during
the course of expansion, it is necessary to specify the en-
vironment in which this evaluation takes place. A transformer spec is expanded in the same environment as that
in which the program is being expanded, but is executed in
an environment that is distinct from the environment in
which the program is executed. This execution envi-
ronment distinction is important only for the resolution of
global variable references and assignments. In what fol-
ows, the environment in which transformers are executed
is called the standard transformer environment and is as-
sumed to be a standard Scheme environment.

Since part of the task of hygienic macro expansion is to
resolve identifier references, the fact that transformers are
expanded in the same environment as the program means
that identifier bindings in the program can shadow iden-
tifier uses within transformers. Since variable bindings
in the program are not available at the time the transformer is
executed, it is an error for a transformer to reference or as-
sign them. However, since keyword bindings are available
during expansion, lexically visible keyword bindings from
the program may be used in macro uses in a transformer.

When a macro use is encountered, the macro transformer
associated with the macro keyword is applied to a repre-
sentation of the macro expression. The result returned
by the macro transformer replaces the original expression
and is expanded once again. Thus macro expansions may
themselves be or contain macro uses.

The syntactic representation passed to a macro transformer
encapsulates information about the structure of the repre-
sented form and the bindings of the identifiers it contains.
These syntax objects can be traversed and examined using
the procedures described below. The output of a transformer may be built up using the usual Scheme list constructors, combining pieces of the input with new syntactic structures.

**(syntax** (datum)) syntax

**Syntax:** The (datum) may be any external representation of a Scheme object.

**Semantics:** Syntax is the syntactic analogue of quote. It creates a syntactic representation of (datum) that, like an argument to a transformer, contains information about the bindings for identifiers contained in (datum). The binding for an identifier introduced by syntax is the closest lexically visible binding. All variables and keywords introduced by transformers must be created by syntax. It is an error to insert a symbol in the output of a transformation procedure unless it is to be part of a quoted datum.

(symbol? (syntax x)) \(\Rightarrow \#f\)

(let-syntax (((car (lambda (x) (syntax car)))))
  ((car ')) 0)

(let-syntax ((quote-quote
  (lambda (x) (list (syntax quote) 'quote))))
  (quote-quote)) \(\Rightarrow\) quote

(let-syntax ((quote-quote
  (lambda (x) (list 'quote 'quote))))
  (quote-quote)) \(\Rightarrow\) error

The second quote-quote example results in an error because two raw symbols are being inserted in the output. The quoted quote in the first quote-quote example does not cause an error because it will be a quoted datum.

(let-syntax ((quote-me
  (lambda (x))
  (list (syntax quote) x)))
  (quote-me please)) \(\Rightarrow\) (quote-me please)

(let ((x 0))
  (let-syntax ((alpha (lambda (e) (syntax x))))
    (alpha))) \(\Rightarrow\) 0

(let ((x 0))
  (let-syntax ((alpha (lambda (x) (syntax x))))
    (alpha))) \(\Rightarrow\) error

(let-syntax ((alpha
  (let-syntax ((beta
    (syntax-rules ()
      ((beta) 0))))
    (lambda (x) (syntax (beta)))))))
  (alpha)) \(\Rightarrow\) error

The last two examples are errors because in both cases a lexically bound identifier is placed outside of the scope of its binding. In the first case, the variable x is placed outside its scope. In the second case, the keyword beta is placed outside its scope.

(let-syntax ((alpha (syntax-rules ()
  ((beta) 0))))
  (let-syntax ((beta (lambda (x) (alpha))))
    (alpha))) \(\Rightarrow\) 0

(let ((list 0))
  (let-syntax ((alpha (lambda (x) (list 0))))
    (alpha)) \(\Rightarrow\) error

The last example is an error because the reference to list in the transformer is shadowed by the lexical binding for list. Since the expansion process is distinct from the execution of the program, transformers cannot reference program variables. On the other hand, the previous example is not an error because definitions for keywords in the program do exist at expansion time.

**Note:** It has been suggested that #' (datum) and #+ (datum) would be felicitous abbreviations for (syntax (datum)) and (quasi-syntax (datum)), respectively, where quasi-syntax, which is not described in this appendix, would bear the same relationship to syntax that quasiquote bears to quote.

**(identifier? syntax-object) procedure**

Returns #t if syntax-object represents an identifier, otherwise returns #f.

(= identifier? (syntax x)) \(\Rightarrow\) #t

(= identifier? (quote x)) \(\Rightarrow\) #f

(= identifier? 3) \(\Rightarrow\) #f

**(unwrap-syntax syntax-object) procedure**

If syntax-object is an identifier, then it is returned unchanged. Otherwise unwrap-syntax converts the outermost structure of syntax-object into a data object whose external representation is the same as that of syntax-object. The result is either an identifier, a pair whose car and cdr are syntax objects, a vector whose elements are syntax objects, an empty list, a string, a boolean, a character, or a number.

(= identifier? (unwrap-syntax (syntax x))) \(\Rightarrow\) #t

(= identifier? (car (unwrap-syntax (syntax x)))) \(\Rightarrow\) #t

(= (unwrap-syntax (cdr (unwrap-syntax (syntax x)))) \(\Rightarrow\) ()

**(free-identifier=? id1 id2) procedure**

Returns #t if the original occurrences of id1 and id2 have the same binding, otherwise returns #f. free-identifier=? is used to look for a literal identifier in the argument to a transformer, such as else in a cond clause. A macro definition for syntax-rules would use free-identifier=? to look for literals in the input.
(free-identifier=? (syntax x) (syntax x))
  \implies \#t

(free-identifier=? (syntax x) (syntax y))
  \implies \#f

(let ((x (syntax x)))
  (free-identifier=? x (syntax x)))
  \implies \#f

(let-syntax
  ((alpha
     (lambda (x)
       (free-identifier=? (car (unwrap-syntax x))
         (syntax alpha))))))
  (alpha))
  \implies \#f

(letrec-syntax
  ((alpha
     (lambda (x)
       (free-identifier=? (car (unwrap-syntax x))
         (syntax alpha))))))
  (alpha))
  \implies \#t

(bound-identifier=? id1 id2) procedure

Returns \#t if a binding for one of the two identifiers id1 and id2 would shadow free references to the other, otherwise returns \#f. Two identifiers can be free-identifier=? without being bound-identifier=? if they were introduced at different stages in the expansion process. Bound-identifier=? can be used, for example, to detect duplicate identifiers in bound-variable lists. A macro definition of syntax-rules would use bound-identifier=? to look for pattern variables from the input pattern in the output template.

(bound-identifier=? (syntax x) (syntax x))
  \implies \#t

(letrec-syntax
  ((alpha
     (lambda (x)
       (bound-identifier=? (car (unwrap-syntax x))
         (syntax alpha))))))
  (alpha))
  \implies \#t

(identifier->symbol id) procedure

Returns a symbol representing the original name of id. Identifier->symbol is used to examine identifiers that appear in literal contexts, i.e., identifiers that will appear in quoted structures.

(symbol? (identifier->symbol (syntax x)))
  \implies \#t

(identifier->symbol (syntax x))
  \implies x

(generate-identifier) procedure

(generate-identifier symbol) procedure

Returns a new identifier. The optional argument to generate-identifier specifies the symbolic name of the resulting identifier. If no argument is supplied the name is unspecified.

Generate-identifier is used to introduce bound identifiers into the output of a transformer. Since introduced bound identifiers are automatically renamed, generate-identifier is necessary only for distinguishing introduced identifiers when an indefinite number of them must be generated by a macro.

The optional argument to generate-identifier specifies the symbolic name of the resulting identifier. If no argument is supplied the name is unspecified. The procedure identifier->symbol reveals the symbolic name of an identifier.

(identifier->symbol (generate-identifier 'x))
  \implies x

(bound-identifier=? (generate-identifier 'x))
  (generate-identifier 'x))
  \implies \#f

(define-syntax set*!
  (set*! (<identifier> <expression>) ...)
  (lambda (x)
    (letrec
      ((unwrap-exp
         (lambda (x)
           (let ((x (unwrap-syntax x)))
             (if (pair? x)
               (cons (car x)
                 (unwrap-exp (cdr x))))))))
      (let ((sets (map unwrap-exp
                   (cdr (unwrap-exp x)))))
        (let ((ids (map car sets))
          (vals (map cdr sets))
          (temps (map (lambda (x)
                        (generate-identifier)))
                  sets)))
          `((syntax let) ,(map list temps vals)
            ,0(map (lambda (id temp)
                     `((syntax set!) ,id ,temp))
                 ids
temps)
             \#f))))

(constuct-identifier id symbol) procedure

Creates and returns an identifier named by symbol that behaves as if it had been introduced where the identifier id was introduced.
Construct-identifier is used to circumvent hygiene by creating an identifier that behaves as though it had been implicitly present in some expression. For example, the transformer for a structure definition macro might construct the name of a field accessor that does not explicitly appear in a use of the macro, but can be constructed from the names of the structure and the field. If a binding for the field accessor were introduced by a hygienic transformer, then it would be renamed automatically, so that the introduced binding would fail to capture any references to the field accessor that were present in the input and were intended to be within the scope of the introduced binding.

Another example is a macro that implicitly binds exit:

```scheme
(define-syntax loop-until-exit
  (lambda (x)
    (let ((exit (construct-identifier
                  (car (unwrap-syntax x))
                  'exit))
        (body (car (unwrap-syntax
                    (cdr (unwrap-syntax x))))))
      `(,(syntax call-with-current-continuation)
        ,(syntax lambda)
        (,exit)
        (,(syntax letrec)
         (,,(syntax lambda)
          (,,(syntax loop)
           (,,(syntax lambda) ()
            ,body
            (,,(syntax loop))))))
        (,,(syntax loop))))))

(let ((x 0) (y 1000))
  (loop-until-exit
   (if (positive? y)
     (begin (set! x (+ x 3))
            (set! y (- y 1)))
     (exit x)))))) => 3000
```

Acknowledgements

The extension described in this appendix is the most sophisticated macro facility that has ever been proposed for a block-structured programming language. The main ideas come from Eugene Kohlbecker’s PhD thesis on hygienic macro expansion [51], written under the direction of Dan Friedman [52], and from the work by Alan Bawden and Jonathan Rees on syntactic closures [6]. Pattern-directed macro facilities were popularized by Kent Dybvig’s non-hygienic implementation of extend-syntax [15].

At the 1988 meeting of this report’s authors at Snowbird, a macro committee consisting of Bawden, Rees, Dybvig, and Bob Hieb was charged with developing a hygienic macro facility akin to extend-syntax but based on syntactic closures. Chris Hanson implemented a prototype and wrote a paper on his experience, pointing out that an implementation based on syntactic closures must determine the syntactic roles of some identifiers before macro expansion based on textual pattern matching can make those roles apparent. William Clinger observed that Kohlbecker’s algorithm amounts to a technique for delaying this determination, and proposed a more efficient version of Kohlbecker’s algorithm. Pavel Curtis spoke up for referentially transparent local macros. Rees merged syntactic environments with the modified Kohlbecker’s algorithm and implemented it all, twice [13].

Dybvig and Hieb designed and implemented the low-level macro facility described above. Recently Hanson and Bawden have extended syntactic closures to obtain an alternative low-level macro facility. The macro committee has not endorsed any particular low-level facility, but does endorse the general concept of a low-level facility that is compatible with the high-level pattern language described in this appendix.

Several other people have contributed by working on macros over the years. Hal Abelson contributed by holding this report hostage to the appendix on macros.
BIBLIOGRAPHY AND REFERENCES


[79] Guy Lewis Steele Jr. Macaroni is better than spaghetti. In Proceedings of the Symposium on Artificial Intelligence and Programming Languages, pages 60–66. These proceedings were published as a special joint issue of SIGPLAN Notices 12(8) and SIGART Newsletter 64, August 1977.


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

! 4
' 7; 16
* 21; 8, 38
+ 21; 5, 8, 13, 19, 28, 36, 38 , 11; 16
- 21; 5, 38
-> 4; 5 ...
/ 21; 20, 38 ; 5 < 21; 36, 38 <= 21; 25, 38 = 21; 14, 38 => 9 > 21; 38 >= 21; 38 ? 4 ' 12

abs 21; 13, 23 acos 23 and 9; 13, 38 angle 23 append 17; 38 apply 27; 36 approximate 38 asin 23 assoc 17 assq 17 assv 17 at-sign 11 atan 23

#b 20; 32 backquote 11 begin 10; 11, 12, 37, 38 binding 6 binding construct 6 boolean? 13; 38 bound 6 bound-identifier=? 45


#d 20 d 20 define 12; 13 define-syntax 41 definition 12 delay 11; 28 denominator 22
display 31
do 11; 6, 13, 37, 38
dotted pair 15

#e 20; 32
e 20
else 9
empty list 15; 16
eof-object? 30
eq? 15; 8, 13, 17
equal? 15; 13, 17, 26
equivalence predicate 13
eqv? 13; 7, 8, 9, 14, 15, 16, 17, 18, 38
error 3
escape procedure 28
essential 3
even? 21
exact 14
exact->inexact 23
exact? 21
exact->inexact 23
exactness 19
exp 22
expt 23

#f 13; 20
f 20
false 6; 13
floor 22
foo 5; 12
for-each 28; 38
force 28; 11, 38
free-identifier=? 44
gcd 22
gen-counter 14
gen-loser 14
generate-identifier 45

hygienic 40

#i 20; 32
identifier 5; 6, 18, 32
identifier->symbol 45
identifier? 44
if 8; 13, 35, 36
imag-part 23
immutable 7
implementation restriction 4; 19
improper list 16
inexact 14
inexact->exact 23; 19, 22
inexact? 21
inexact->exact 23
initial environment 13
input-port? 29

integer->char 25; 38
integer? 20; 19
integer->char 25
integrate-system 39
internal definition 13

keyword 5; 6, 32, 40

l 20
lambda 8; 13, 34, 35, 36, 38
lambda expression 6
last-pair 38
lazy evaluation 11
lcm 22
length 17; 19
let 10; 11, 6, 13, 37, 38
let* 10; 6, 13
let-syntax 41
letrec 10; 6, 13, 38
letrec-syntax 41
list 17; 27
list->string 26
list->vector 27; 4
list-ref 17
list-tail 17
list? 16; 38
list->string 26
load 31
location 7
log 22; 23

macro 40
macro keyword 40
macro transformer 40
macro use 40
magnitude 23
make-polar 23
make-promise 28
make-rectangular 23
make-string 25
make-vector 26
map 27; 28, 38, 39
map-strings 39
max 21
member 17
memq 17
memv 17; 37
min 21
modulo 22
mutable 7

negative? 21
newline 31
nil 13; 38
not 13
null? 16
with-output-to-file 30
write 31; 6, 12, 18
write-char 31

#x 20; 32

zero? 21