Revised\(^3\) Report on the Algorithmic Language Scheme

JONATHAN REES AND WILLIAM CLINGER (Editors)

H. ABELSON R. K. DYBVIG C. T. HAYNES G. J. ROZAS
N. I. ADAMS IV D. P. FRIEDMAN E. KOHLBECKER G. J. SUSSMAN
D. H. BARTLEY R. HALSTEAD D. OXLEY M. WAND
G. BROOKS C. HANSON K. M. PITMAN

Dedicated to the Memory of ALGOL 60

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.

The report concludes with an example of the use of the language and an alphabetic index.

CONTENTS

<table>
<thead>
<tr>
<th>CONTENTS</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduction</td>
<td>2</td>
</tr>
<tr>
<td>1. Overview of Scheme</td>
<td>3</td>
</tr>
<tr>
<td>1.1. Semantics</td>
<td>3</td>
</tr>
<tr>
<td>1.2. Syntax</td>
<td>3</td>
</tr>
<tr>
<td>1.3. Notation and terminology</td>
<td>3</td>
</tr>
<tr>
<td>2. Lexical conventions</td>
<td>4</td>
</tr>
<tr>
<td>2.1. Identifiers</td>
<td>4</td>
</tr>
<tr>
<td>2.2. Whitespace and comments</td>
<td>5</td>
</tr>
<tr>
<td>2.3. Other notations</td>
<td>5</td>
</tr>
<tr>
<td>3. Basic concepts</td>
<td>5</td>
</tr>
<tr>
<td>3.1. Variables and regions</td>
<td>5</td>
</tr>
<tr>
<td>3.2. True and false</td>
<td>6</td>
</tr>
<tr>
<td>3.3. External representations</td>
<td>6</td>
</tr>
<tr>
<td>4. Expressions</td>
<td>6</td>
</tr>
<tr>
<td>4.1. Primitive expression types</td>
<td>6</td>
</tr>
<tr>
<td>4.2. Derived expression types</td>
<td>8</td>
</tr>
<tr>
<td>5. Program structure</td>
<td>11</td>
</tr>
<tr>
<td>5.1. Programs</td>
<td>11</td>
</tr>
<tr>
<td>5.2. Definitions</td>
<td>11</td>
</tr>
<tr>
<td>6. Standard procedures</td>
<td>12</td>
</tr>
<tr>
<td>6.1. Booleans</td>
<td>12</td>
</tr>
<tr>
<td>6.2. Equivalence predicates</td>
<td>13</td>
</tr>
<tr>
<td>6.3. Pairs and lists</td>
<td>14</td>
</tr>
<tr>
<td>6.4. Symbols</td>
<td>17</td>
</tr>
<tr>
<td>6.5. Numbers</td>
<td>17</td>
</tr>
<tr>
<td>6.6. Characters</td>
<td>23</td>
</tr>
<tr>
<td>6.7. Strings</td>
<td>24</td>
</tr>
<tr>
<td>6.8. Vectors</td>
<td>25</td>
</tr>
<tr>
<td>6.9. Control features</td>
<td>26</td>
</tr>
<tr>
<td>6.10. Input and output</td>
<td>28</td>
</tr>
<tr>
<td>7. Formal syntax and semantics</td>
<td>30</td>
</tr>
<tr>
<td>7.1. Formal syntax</td>
<td>30</td>
</tr>
<tr>
<td>7.2. Formal semantics</td>
<td>32</td>
</tr>
<tr>
<td>7.3. Derived expression types</td>
<td>35</td>
</tr>
<tr>
<td>Notes</td>
<td>36</td>
</tr>
<tr>
<td>Example</td>
<td>37</td>
</tr>
<tr>
<td>Bibliography and references</td>
<td>38</td>
</tr>
<tr>
<td>Alphabetic index of definitions of concepts,</td>
<td>41</td>
</tr>
<tr>
<td>keywords, and procedures</td>
<td></td>
</tr>
</tbody>
</table>
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 has influenced the evolution of Lisp. Scheme was one of the first programming languages to incorporate first class procedures as in the lambda calculus, thereby proving the usefulness of static scope rules and block structure in a dynamically typed language. Scheme was the first major dialect of Lisp to distinguish procedures from lambda expressions and symbols, to use a single lexical environment dialect of Lisp to distinguish procedures from lambda expressions and symbols, to use a single lexical environment.

As might be expected of a language used primarily for education and research, Scheme has always evolved rapidly. This was no problem when Scheme was used only within MIT, but 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. Participating in this workshop were Hal Abelson, Norman Adams, David Bartley, Gary Brooks, William Clinger, Daniel Friedman, Robert Halstead, Chris Hanson, Christopher Haynes, Eugene Kohlbecker, Don Oxley, Jonathan Rees, Guillermo Rozas, Gerald Jay Sussman, and Mitchell Wand. Kent Pitman made valuable contributions to the agenda for the workshop but was unable to attend the sessions.

Subsequent electronic mail discussions and committee work completed the definition of the language. Gerry Sussman drafted the section on numbers, Chris Hanson drafted the sections on characters and strings, and Gary Brooks and William Clinger drafted the sections on input and output. William Clinger recorded the decisions of the workshop and compiled the pieces into a coherent document. The “Revised revised report on Scheme” [4] was published at MIT and Indiana University in the summer of 1985. Another round of revision in the spring of 1986, again accomplished almost entirely by electronic mail, resulted in the present report.

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 comments and criticisms: Alan Bawden, George Carrette, Andy Cromarty, Andy Freeman, Richard Gabriel, Yekta Gürsel, Ken Haase, Paul Hudak, Richard Kelsey, Chris Lindblad, Mark Meyer, Jim Miller, Jim Philbin, John Ransdell, Guy Lewis Steele Jr., Julie Sussman, Perry Wagle, Daniel Weise, and Henry Wu. 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 and the Computer Science Department of Indiana University 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.
DESCRIPTION OF THE LANGUAGE

1. Overview of Scheme

1.1. Semantics

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

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

Scheme has latent as opposed to manifest types. 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 SASL, or the call-by-name semantics of Algol 60, where an argument expression is not evaluated unless its value is needed by the procedure.

1.2. Syntax

Scheme employs a parenthesized-list Polish notation to describe programs and (other) data. The syntax of Scheme, like that of most Lisp dialects, provides for great expressive power, largely due to its simplicity. An important consequence of this simplicity is the susceptibility of Scheme programs and data to uniform treatment by other Scheme programs. As with other Lisp dialects, the read primitive parses its input; that is, it performs syntactic as well as lexical decomposition of what it reads.

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.

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 other arguments.

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

```
(append list1 list2) essential procedure
(append list . . .) procedure
```

indicate that in all implementations, the append procedure must be defined to take two arguments, and some implementations will extend it to take zero or more 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, then this implies a restriction on the type of that argument to the procedure. For example, the header line for vector-ref given above dictates that first argument to vector-ref must be a vector. The following naming conventions also imply type restrictions:

```
obj any object
z, z₁, . . ., zₖ . . . complex, real, rational, integer
x, x₁, . . ., xₖ . . . real, rational, integer
y, y₁, . . ., yₖ . . . real, rational, integer
q, q₁, . . ., qₖ . . . rational, integer
n, n₁, . . ., nₖ . . . integer
k, k₁, . . ., kₖ . . . 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.

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 - (which can begin numbers) are identifiers. Here are some examples of identifiers:

```
lambda q
list->vector soup
+ V17a
<=? a34kTNMs
the-word-recursion-has-many-meanings
```

Extended alphabetic characters may be used in identifiers exactly 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. Here are 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.

The characters `?` and `!` have no special properties—they are extended alphabetic characters. By convention, however, most predicate procedures (those that return boolean values) are named by identifiers that end in `?`, and most data mutation procedures are named by identifiers that end in `!`.

### 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.

```scheme
;;; The FACT procedure computes the factorial
;;; of a non-negative integer.
(define fact
  (lambda (n)
    (if (= n 0)
        1 ;Base case: return 1
        (* n (fact (- n 1))))))
```

### 2.3. Other notations

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

`()` Parentheses are used for grouping and to notate 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 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 #l #s #b #o #d #x` These are used in the notation for numbers (section 6.5.3).

### 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 such 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 \texttt{let}, \texttt{let*}, \texttt{letrec}, and \texttt{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 and the empty list, which count as false. This report uses the word “true” to refer to any Scheme value that counts as true, and the word “false” to refer to any Scheme value that counts as false.

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 “28.000” and "\texttt{#x1c}", and the list in the previous paragraph also has the representations “( 08 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 \texttt{quote}, section 4.1.2).

External representations can also be used for input and output. The procedure \texttt{read} (section 6.10.2) parses external representations, and the procedure \texttt{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 \textit{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 which 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 which 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.

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

\begin{verbatim}
(variable)
\end{verbatim}

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

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

\begin{verbatim}
x ⇒ 28
\end{verbatim}
4.1.2. Literal expressions

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

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

\[
\begin{align*}
\text{(quote a)} & \implies a \\
\text{(quote #(a b c))} & \implies #(a b c) \\
\text{(quote (+ 1 2))} & \implies (+ 1 2)
\end{align*}
\]

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

\[
\begin{align*}
\text{'}a & \implies a \\
\text{'#(a b c)} & \implies #(a b c) \\
\text{'+( 1 2)} & \implies (+ 1 2) \\
\text{'(quote a)} & \implies (quote a) \\
\text{''a} & \implies (quote a)
\end{align*}
\]

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

\[
\begin{align*}
\text{''"abc"} & \implies "abc" \\
\text{"abc"} & \implies "abc" \\
\text{''145932} & \implies 145932 \\
\text{145932} & \implies 145932 \\
\text{''#t} & \implies #t \\
\text{#t} & \implies #t
\end{align*}
\]

It is an error to alter a constant (i.e. the value of a literal expression) using a mutation procedure like set-car! or string-set!.

4.1.3. Procedure calls

\[ \text{(operator) (operand\textsubscript{1}) . . .)} \]

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

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

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

4.1.4. Lambda expressions

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

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

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

\[
\begin{align*}
\text{(lambda (x) (+ x x)} & \implies a \text{ procedure} \\
\text{((lambda (x) (+ x x)) 4)} & \implies 8 \\
\text{(define reverse-subtract) \\
\text{(lambda (x y) (- y x))}) & \implies 3 \\
\text{(define foo) \\
\text{(let ((x 4)) \\
\text{(lambda (y) (+ x y)))}} & \implies 10 \\
\text{foo 6) & \implies 10
\end{align*}
\]

(Formals) should have one of the following forms:

- (variable\textsubscript{1} . . .): The procedure takes a fixed number of arguments; when the procedure is called, the arguments will be stored in the bindings of the corresponding variables.

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

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

\[
\begin{align*}
\text{((lambda x x) 3 4 5 6)} & \implies (3 4 5 6) \\
\text{((lambda (x y . z) z) 3 4 5 6)} & \implies (5 6)
\end{align*}
\]
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.

(test) => yes
(test) => no
(if (> 3 2) (+ x 1)) => 3
(+ x 1) => 5

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)
(define x 3)
(+ x 4) => unspecified
(+ x 1) => 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)) => greater
(cond ((> 3 3) 'greater) ((< 3 3) 'less) (else 'equal)) => equal

Some implementations support an alternative (clause) syntax, (test => (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))) => cadr) (else #f)) => 2

4.2.2. Case expressions

(case (key) (clause1) (clause2) ...)  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 eqv?; 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) => composite
(case (car '(c d)) ((a) 'a) ((b) 'b) => unspecified
(case (car '(c d)) ((a e i o u) 'vowel) ((u y) 'semivowel)
(else 'consonant)) => consonant
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 expressions evaluate to true values, the value of the last expression is returned. If there are no expressions then #t is returned.

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

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

The three binding constructs let, let*, and letrec 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 initial values are computed, and evaluations are performed sequentially; while in a letrec expression, the bindings are in effect while their expressions are evaluated from left to right, and the value of the last expression in the resulting environment, and the value of the last expression in the binding is visible, and so on.

\[\text{(let} \ (\text{bindings}) \ (\text{body})\]\ Syntax: (Bindings) should have the form

\[\text{((variable}_1) \ (\text{init}_1)) \ldots\]

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.

\[\text{(let*} \ (\text{bindings}) \ (\text{body})\]\ Syntax: (Bindings) should have the form

\[\text{((variable}_1) \ (\text{init}_1)) \ldots\]

and (body) should be a sequence of one or more expressions.

**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 (body) as its region.

\[\text{(letrec} \ (\text{bindings}) \ (\text{body})\]\ essential syntax

**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 <body> as its region.

\[
\begin{align*}
\text{(let} ((x 2) \ (y 3)) \\
(* \ x \ y)) & \rightarrow 6 \\
\text{(let} ((x 2) \ (y 3)) \\
(let ((foo \ (lambda \ (z) (+ \ x \ y \ z))) \ (x \ 7)) \\
(foo \ 4))) & \rightarrow 9
\end{align*}
\]

See also named let, section 4.2.4.

\[
\begin{align*}
\text{(let*} \ (\text{bindings}) \ (\text{body})\)
\end{align*}
\]

\[
\begin{align*}
\text{Syntax: (Bindings) should have the form}
\end{align*}
\]

\[
\begin{align*}
\text{((variable}_1) \ (\text{init}_1)) \ldots,
\end{align*}
\]

and (body) should be a sequence of one or more expressions.

**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 (body) as its region, making it possible to define mutually recursive procedures.
One restriction on letrec is very important: it must be possible to evaluate each (init) without referring to the value of any (variable). If this restriction is violated, then the effect is undefined, and an error may be signaled during evaluation of the (init)s. The restriction is necessary because Scheme passes arguments by value rather than by name. In the most common uses of letrec, all the (init)s are lambda expressions and the restriction is satisfied automatically.

4.2.3. Sequencing

(begin (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.

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

(begin (display "4 plus 1 equals ") (display (+ 4 1))) \(\Rightarrow\) unspecified and prints 4 plus 1 equals 5

Note: [1] uses the keyword sequence instead of begin.

4.2.4. Iteration

(do ((variable₁) (init₁) (step₁)) ...)

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

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

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

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

The region of the binding of a (variable) consists of the entire do expression except for the (init)s.

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

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

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

(let (variable) (bindings) (body)) syntax

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).

(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 1 3) (-5 -2))

4.2.5. Delayed evaluation

(delay (expression)) syntax

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

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

`(quasiquote (template))` syntax
`
'(template) syntax

"Backquote" or "quasiquote" expressions are useful for constructing a list or vector structure when most but not all of the desired structure is known in advance. If no commas appear within the `(template)`, the result of evaluating `'(template)` is equivalent 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)) b
     ⇒   (a 3 4 5 6 b)
'(foo ,(- 10 3),@(cdr '(c)) ,,(car 'cons))
    ⇒   ((foo 7) . cons)
'(#(10 5 ,,@(sqrt 4),@map sqrt ,(16 9)) 8)
⇒   #(10 5 2 4 3 8)
```

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

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

The notations `'(template)` and `(quasiquote (template))` are identical in all respects. `,expression is identical to `(unquote (expression))`, and `,@expression is identical to `(unquote-splicing (expression))`. The external syntax generated by `write` for two-element lists whose car is one of these symbols 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 the present 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 are 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))).
  ```

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))
```
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, 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 and the empty list count as false in conditional expressions. Everything else, including #t, pairs, symbols, numbers, strings, vectors, and procedures, counts as true.

The empty list counts as false for compatibility with existing programs and implementations that assume this to be the case.

Programmers accustomed to other dialects of Lisp should beware that Scheme distinguishes false and the empty list from the symbol nil.

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

\[
\begin{align*}
\text{(not #t)} & \quad \Rightarrow \quad #f \\
\text{(not #f)} & \quad \Rightarrow \quad #t \\
\end{align*}
\]

\[
\begin{align*}
\text{(not 0)} & \quad \Rightarrow \quad #f \\
\text{(not (list))} & \quad \Rightarrow \quad #t \\
\end{align*}
\]

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

\[
\begin{align*}
\text{(boolean? obj)} & \quad \text{essential procedure} \\
\text{Boolean? returns #t if obj is either #t or #f and returns #f otherwise.} \\
\end{align*}
\]

\[
\begin{align*}
\text{(boolean? #f)} & \quad \Rightarrow \quad #t \\
\text{(boolean? 0)} & \quad \Rightarrow \quad #f \\
\end{align*}
\]

Some implementations provide variables nil and t whose values in the initial environment are #f and #t respectively.

\[
\begin{align*}
\text{nil} & \quad \Rightarrow \quad #f \\
\text{t} & \quad \Rightarrow \quad #t \\
\end{align*}
\]
### 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, **eqv?** is the finest or most discriminating, and **equal?** is the coarsest. **Eqv?** is slightly less discriminating than **eq?**.

Two objects are **operationally equivalent** if and only if there is no way that they can be distinguished, using Scheme primitives other than **eqv?** or **eq?** or those like **memq** and **assv** whose meaning is defined explicitly in terms of **eqv?** or **eq?**. It is guaranteed that objects maintain their operational identity despite being named by variables or fetched from or stored into data structures.

This definition can be interpreted in the following ways for various kinds of objects:

- The two boolean values, **#t** and **#f**, are operationally distinct because they behave in opposite ways in conditionals.
- Two symbols are operationally equivalent if they print the same way.
- Two numbers are operationally equivalent if they are numerically equal (see **=**, section 6.5.4) and are either both exact or both inexact (see section 6.5.2).
- Two characters are operationally equivalent if they are the same character according to **char=?** (section 6.6).
- Two structured mutable objects—pairs, vectors, or strings—are operationally equivalent if they have operationally equivalent values in corresponding positions, and applying a mutation procedure to one causes the other to change as well. (A mutation procedure is one like **set-car!** which alters a data structure.) For example, two pairs are not operationally equivalent if a **set-car!** operation on one does not change the car field of the other.
- There is only one empty list, and it is operationally equivalent to itself. All empty vectors are operationally equivalent to each other. All empty strings are operationally equivalent to each other. Whether there is more than one empty vector or string is implementation-dependent.
- Two procedures are operationally equivalent if, when called on operationally equivalent arguments, they return the same value and perform the same side effects.

\[
\text{(eqv? obj1 obj2)}
\]

The **eqv?** procedure implements an approximation to the relation of operational equivalence. It returns **#t** if it can prove that **obj1** and **obj2** are operationally equivalent. If it can't, it always errs on the conservative side and returns **#f**.

The only situation in which it might fail to prove is when **obj1** and **obj2** are operationally equivalent procedures that were created at different times. In general, operational equivalence of procedures is uncomputable, but it is guaranteed that **eqv?** can recognize a procedure created at a given time by a given lambda expression as "being itself." This is useful for applications in which procedures are being used to implement objects with local state.

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

The following examples illustrate cases in which **eqv?** is permitted to fail to prove operational equivalence, depending on the implementation. (In every case, it will return either **#t** or **#f**, but which one it returns is implementation-dependent.) Compare with the last example in the previous set.

\[
\begin{align*}
\text{(eqv? (lambda (x) x) (lambda (x) x))} & \Rightarrow \text{unspecified} \\
\text{(eqv? (lambda (x) x) (lambda (y) y))} & \Rightarrow \text{unspecified}
\end{align*}
\]

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.

\[
\begin{align*}
\text{(define gen-counter}
\quad \text{(lambda ()}
\quad \quad \text{(let ((n 0))}
\quad \quad \quad \text{(lambda () (set! n (+ n 1)) n)))))
\quad \text{(let ((g (gen-counter)))}
\quad \quad \text{(eqv? g g))} & \Rightarrow \text{#t} \\
\quad \text{(eqv? (gen-counter) (gen-counter))} & \Rightarrow \text{#f}
\end{align*}
\]

\[
\begin{align*}
\text{(define gen-loser}
\quad \text{(lambda ()}
\quad \quad \text{(let ((n 0))}
\quad \quad \quad \text{(lambda () (set! n (+ n 1)) 27)))))
\quad \text{(let ((g (gen-loser)))}
\quad \quad \text{(eqv? g g))} & \Rightarrow \text{#t} \\
\quad \text{(eqv? (gen-loser) (gen-loser))} & \Rightarrow \text{unspecified}
\end{align*}
\]
Objects of distinct types are never operationally equivalent, except that false and the empty list are permitted to be identical, and the character type need not be disjoint from other types.

\[(\text{eqv? } '() \ #\text{f})\quad\Rightarrow\quad \text{unspecified}\]
\[(\text{eqv? } 57 \ #\text{\#A})\quad\Rightarrow\quad \text{unspecified}\]

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 \text{eqv?} on constants is sometimes implementation-dependent.\[\text{(let \((x \ 'a)\))}
\text{(eqv? x x))\quad\Rightarrow\quad \#t\]
\text{(eqv? 'a 'a))\quad\Rightarrow\quad \text{unspecified}\]
\text{(eqv? 'a "a")}\quad\Rightarrow\quad \text{unspecified}\]
\text{(eqv? 'b (cdr 'a b)))\quad\Rightarrow\quad \text{unspecified}\]

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

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

\text{Eq?} and \text{eqv?} are guaranteed to have the same behavior on symbols, booleans, the empty list, pairs, and non-empty strings and vectors. \text{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 \text{eqv?} would also return true. \text{Eq?} may also behave differently from \text{eqv?} on empty vectors and empty strings.

\[(\text{eq? } 'a \ 'a)\quad\Rightarrow\quad \#t\]
\text{(eq? } 'a (\text{'}a)\text{) }\Rightarrow\quad \#t\]
\text{(eq? \ (list 'a) (list 'a)) }\Rightarrow\quad \#t\]
\text{(eq? "a" "a") }\Rightarrow\quad \text{unspecified}\]
\text{(eq? == ==) }\Rightarrow\quad \text{unspecified}\]
\text{(eq? () ()) }\Rightarrow\quad \#t\]
\text{(eq? 2 2) }\Rightarrow\quad \#t\]
\text{(eq? #\text{\#A} #\text{\#A}) }\Rightarrow\quad \text{unspecified}\]
\text{(eq? car car) }\Rightarrow\quad \#t\]
\text{(let \((n (+ 2 3)))}
\text{(eq? n n)) }\Rightarrow\quad \text{unspecified}\]
\text{(let \((x 'a))}
\text{(eq? x x)) }\Rightarrow\quad \#t\]
\text{(let \((x 'a))}
\text{(eq? x x)) }\Rightarrow\quad \#t\]
\text{(let \((x 'a))}
\text{(eq? x x)) }\Rightarrow\quad \#t\]
\text{(let \((p \ (\lambda \ (x) \ x)))}
\text{(eq? p p)) }\Rightarrow\quad \#t\]

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

\text{(equal? obj1 obj2)}\quad\text{essential procedure}\]
\text{Equal?} recursively compares the contents of pairs, vectors, and strings, applying \text{eqv?} on other objects such as numbers and symbols. A rule of thumb is that objects are generally \text{equal?} if they print the same. \text{Equal?} may fail to terminate if its arguments are circular data structures.

\[(\text{equal? 'a 'a) }\Rightarrow\quad \#t\]
\text{(equal? 'a (\text{'a}) ) }\Rightarrow\quad \#t\]
\text{(equal? '(a b) c) }\Rightarrow\quad \#t\]
\text{(equal? "abc" "abc") }\Rightarrow\quad \#t\]
\text{(equal? 2 2) }\Rightarrow\quad \#t\]
\text{(equal? (make-vector 5 'a) (make-vector 5 'a)) }\Rightarrow\quad \#t\]
\text{(equal? (\lambda \ (x) \ x) (\lambda \ (y) \ y)) }\Rightarrow\quad \text{unspecified}\]

6.3. Pairs and lists

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

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 both representations of the same 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:

\[
\begin{align*}
\text{(define x (list 'a 'b 'c))} \\
\text{(define y x)} \\
\text{(set-cdr! x 4)} \\
\text{(eqv? x y)} \\
\text{y}
\end{align*}
\]

\[
\begin{array}{c}
\text{⇒ (a b c)} \\
\text{⇒ unspecified} \\
\text{⇒ (a . 4)} \\
\text{⇒ #t} \\
\text{⇒ (a . 4)}
\end{array}
\]

It is often convenient to speak of a homogeneous list of objects of some particular data type, as for example \((1 \ 2 \ 3)\) is a list of integers. To be more precise, suppose \(D\) is some data type. (Any predicate defines a data type consisting of those objects of which the predicate is true.) Then

- The empty list is a list of \(D\).
- If \texttt{list} is a list of \(D\), then any pair whose \texttt{cdr} is \texttt{list} and whose \texttt{car} is an element of the data type \(D\) is also a list of \(D\).
- There are no other lists of \(D\).

Within literal expressions and representations of objects read by the \texttt{read} procedure, the forms \(,\text{datum}\), \(\text{datum},\), \(\text{datum,}\), and \(\text{datum,datum}\) denote two-element lists whose first elements are the symbols \texttt{quote}, \texttt{quas quo te}, \texttt{unquote}, and \texttt{unquote-splicing}, respectively. The second element in each case is \(\text{datum}\). 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 \(\text{datum}\) (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.
Arbitrary compositions, up to four deep, are provided. There are twenty-eight of these procedures in all.

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

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

Returns \#t if \(obj\) is the empty list, otherwise returns \#f. (In implementations in which the empty list is the same as \#f, \text{null?} will return \#t if \(obj\) is \#f.)

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

Returns a list of its arguments.

\[(\text{list} \ \text{a} \ (+ \ 3 \ 4) \ \text{c}) \quad \Rightarrow \ (\text{a} \ 7 \ \text{c})\]
\[(\text{list}) \quad \Rightarrow \ ()\]

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

Returns the length of \(list\).

\[(\text{length} \ (\text{a} \ \text{b} \ \text{c})) \quad \Rightarrow \ 3\]
\[(\text{length} \ (\text{a} \ (\text{b} \ (\text{c} \ d \ e)))) \quad \Rightarrow \ 3\]
\[(\text{length} \ ()) \quad \Rightarrow \ 0\]

\[(\text{append} \ list_1 \ list_2) \quad \text{essential procedure}\]

\[(\text{append} \ list \ \ldots) \quad \text{procedure}\]

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

\[(\text{append} \ (\text{x}) \ (\text{y})) \quad \Rightarrow \ (\text{x} \ \text{y})\]
\[(\text{append} \ (\text{a}) \ (\text{b} \ \text{c} \ \text{d})) \quad \Rightarrow \ (\text{a} \ \text{b} \ \text{c} \ \text{d})\]
\[(\text{append} \ (\text{a} \ (\text{b})) \ (\text{(c)} \ (\text{c}))) \quad \Rightarrow \ (\text{a} \ (\text{b}) \ (\text{c} \ (\text{c})))]\]

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 it is not a proper list.

\[(\text{append} \ (\text{a} \ \text{b}) \ (\text{c} \ . \ \text{d})) \quad \Rightarrow \ (\text{a} \ \text{b} \ \text{c} \ . \ \text{d})\]
\[(\text{append} \ () \ \text{a}) \quad \Rightarrow \ \text{a}\]

\[(\text{reverse} \ list) \quad \text{procedure}\]

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

\[(\text{reverse} \ (\text{a} \ \text{b} \ \text{c})) \quad \Rightarrow \ (\text{c} \ \text{b} \ \text{a})\]
\[(\text{reverse} \ (\text{a} \ (\text{b} \ \text{c}) \ \text{d} \ (\text{e} \ (\text{f})))) \quad \Rightarrow \ ((\text{e} \ (\text{f})) \ \text{d} \ (\text{b} \ \text{c}) \ \text{a})\]

\[(\text{list-tail} \ list \ k) \quad \text{procedure}\]

Returns the sublist of \(list\) obtained by omitting the first \(k\) elements. \text{List-tail} could be defined by

\[(\text{define list-tail} \ (\lambda (x \ k) \ (\text{if} \ (\text{zero?} \ k) \ x \ (\text{list-tail} \ (\text{cdr} \ x) \ (- \ k \ 1))))).\]

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

Returns the \(k\)th element of \(list\). (This is the same as the car of \(\text{list-tail} \ list \ k\).)

\[(\text{last-pair} \ list) \quad \text{procedure}\]

Returns the last pair in the nonempty, possibly improper, list \text{list}. \text{Last-pair} could be defined by

\[(\text{define last-pair} \ (\lambda (x) \ (\text{if} \ (\text{pair?} \ (\text{cdr} \ x)) \ (\text{last-pair} \ (\text{cdr} \ x) \ x))))\]

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

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

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

These procedures return the first sublist of \(list\) whose car is \(obj\). If \(obj\) does not occur in \(list\), \#f (n.b.: not the empty list) is returned. \text{Memq} uses \text{eq?} to compare \(obj\) with the elements of \(list\), while \text{memv} uses \text{eqv?} and \text{member} uses \text{equal?}.

\[(\text{memq} \ \text{a} \ (\text{a} \ \text{b} \ \text{c})) \quad \Rightarrow \ (\text{a} \ \text{b} \ \text{c})\]
\[(\text{memq} \ \text{b} \ (\text{a} \ \text{b} \ \text{c})) \quad \Rightarrow \ (\text{b} \ \text{c})\]
\[(\text{memq} \ \text{a} \ (\text{b} \ \text{c} \ \text{d})) \quad \Rightarrow \ \#f\]
\[(\text{memq} \ (\text{list} \ \text{a}) \ (\text{a} \ \text{b} \ \text{c})) \quad \Rightarrow \ \#f\]
\[(\text{member} \ (\text{list} \ \text{a}) \ (\text{b} \ \text{a} \ \text{c})) \quad \Rightarrow \ ((\text{a}) \ \text{c})\]
\[(\text{memq} \ 101 \ (100 \ 101 \ 102)) \quad \Rightarrow \ \text{unspecifie}\]
\[(\text{memv} \ 101 \ (100 \ 101 \ 102)) \quad \Rightarrow \ (101 \ 102)\]

\[(\text{assq} \ obj \ \text{alist}) \quad \text{essential procedure}\]

\[(\text{assv} \ obj \ \text{alist}) \quad \text{essential procedure}\]

\[(\text{assoc} \ obj \ \text{alist}) \quad \text{essential procedure}\]

\text{Alist} (for “association list”) must be a list of pairs. These procedures find the first pair in \text{alist} whose car field is \(obj\), and returns that pair. If no pair in \text{alist} has \(obj\) as its car, \#f is returned. \text{Assq} uses \text{eq?} to compare \(obj\) with the car fields of the pairs in \text{alist}, while \text{assv} uses \text{eqv?} and \text{assoc} uses \text{equal?}.

\[(\text{define e} \ (\text{a} \ (\text{a} \ (\text{b} \ (\text{c} \ (\text{d}))))))\]
\[(\text{assq} \ \text{a} \ \text{e}) \quad \Rightarrow \ (\text{a} \ \text{e})\]
\[(\text{assq} \ \text{b} \ \text{e}) \quad \Rightarrow \ (\text{b} \ \text{e})\]
\[(\text{assq} \ \text{d} \ \text{e}) \quad \Rightarrow \ \#f\]
\[(\text{assq} \ (\text{list} \ \text{a}) \ ((\text{a}) \ ((\text{b})) \ ((\text{c}))))\]
6. Standard procedures

6.4. Symbols

Symbols are objects whose usefulness rests on the fact that two symbols are identical (in the sense of eqv?) if and only if their names are spelled the same way. This is exactly the property needed to represent identifiers in programs, and so most implementations of Scheme use them internally for that purpose. Symbols are useful for many other applications; for instance, they may be used the way enumerated types 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 read procedure, and subsequently written out using the write procedure, will read back in as the identical symbol (in the sense of eqv?). The string->symbol procedure, however, can create symbols for which this write/read invariance may not hold because their names contain special characters or letters in the non-standard case.

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.

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

(string->symbol 'foo) => #t
(string->symbol (car '(a b))) => #t
(string->symbol "bar") => #f

Note: 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.

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

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

(string->symbol string) essential procedure

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

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

(eq? 'mISSISSIppi 'mississippi) => #t
(eq? 'bitBlt (string->symbol "bitBlt")) => #f
(eq? 'JollyWog (string->symbol (symbol->string 'JollyWog))) => #t

6.5. Numbers

Numerical computation has traditionally been neglected by the Lisp community. Until Common Lisp there has been no carefully thought out strategy for organizing numerical computation, and with the exception of the MacLisp system [28] there has been little effort to execute numerical code efficiently. We applaud the excellent work of the Common Lisp committee and we accept many of their recommendations. In some ways we simplify and generalize their proposals in a manner consistent with the purposes of Scheme.
Scheme’s numerical operations treat numbers as abstract data, as independent of their representation as is possible. Thus, the casual user should be able to write simple programs without having to know that the implementation may use fixed-point, floating-point, and perhaps other representations for his data. Unfortunately, this illusion of uniformity can be sustained only approximately—the implementation of numbers will leak out of its abstraction whenever the user must be in control of precision, or accuracy, or when he must construct especially efficient computations. Thus the language must also provide escape mechanisms so that a sophisticated programmer can exercise more control over the execution of his code and the representation of his data when necessary.

It is important to distinguish between the abstract numbers, their machine representations, and their written representations. We will use mathematical terms number, complex, real, rational, and integer for properties of the abstract numbers, the names fixnum, bignum, ratnum, and flonum for machine representations, and the names int, fix, flo, sci, rat, polar, and rect for input/output formats.

### 6.5.1. Numbers

A Scheme system provides data of type number, which is the most general numerical type supported by that system. Number is likely to be a complicated union type implemented in terms of fixnums, bignums, flonums, and so forth, but this should not be apparent to a naive user. What the user should see is that the usual operations on numbers produce the mathematically expected results, within the limits of the implementation. Thus if the user divides the exact number 3 by the exact number 2, he should get something like 1.5 (or the exact fraction 3/2).

If he adds that result to itself, and the implementation is good enough, he should get an exact 3.

Mathematically, numbers may be arranged into a tower of subtypes with projections and injections relating adjacent levels of the tower:

```
number
  complex
  real
  rational
  integer
```

We impose a uniform rule of downward coercion—a number of one type is also of a lower type if the injection (up) of the projection (down) of a number leaves the number unchanged. Since this tower is a genuine mathematical structure, Scheme provides predicates and procedures to access the tower.

Not all implementations of Scheme must provide the whole tower, but they must implement a coherent subset consistent with both the purposes of the implementation and the spirit of the Scheme language.

#### 6.5.2. Exactness

Numbers are either exact or inexact. A number is exact if it was derived from exact numbers using only exact operations. A number is inexact if it models a quantity (e.g., a measurement) known only approximately, if it was derived using inexact ingredients, or if it was derived using inexact operations. Thus inexactness is a contagious property of a number. Some operations, such as the square root (of non-square numbers), must be inexact because of the finite precision of our representations. Other operations are inexact because of implementation requirements.

We emphasize that exactness is independent of the position of the number on the tower. It is perfectly possible to have an inexact integer or an exact real; 355/113 may be an exact rational or it may be an inexact rational approximation to pi, depending on the application.

Operationally, it is the system’s responsibility to combine exact numbers using exact methods, such as infinite precision integer and rational arithmetic, where possible. An implementation may not be able to do this (if it does not use infinite precision integers and rationals), but if a number becomes inexact for implementation reasons there is likely to be an important error condition, such as integer overflow, to be reported. Arithmetic on inexact numbers is not so constrained. The system may use floating point and other ill-behaved representation strategies for inexact numbers. This is not to say that implementors need not use the best known algorithms for inexact computations—only that approximate methods of high quality are allowed.

In a system that cannot explicitly distinguish exact from inexact numbers the system must do its best to maintain precision. Scheme systems must not burden users with numerical operations described in terms of hardware and operating-system dependent representations such as fixnum and flonum, however, because these representation issues are hardly ever germane to the user’s problems.

We highly recommend that the IEEE 32-bit and 64-bit floating-point standards be adopted for implementations that use floating-point representations internally. To minimize loss of precision we adopt the following rules: If an implementation uses several different sizes of floating-point formats, the results of any operation with a floating-point result must be expressed in the largest format used to express any of the floating-point arguments to that operation. It is desirable (but not required) for potentially irrational 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 an exact number (or an inexact number represented as
6. Standard procedures

6.5.3. Number syntax

Scheme allows the traditional ways of writing numerical constants, though any particular implementation may support only some of them. These syntaxes are intended to be purely notational; any kind of number may be written in any form that the user deems convenient. Of course, writing 1/7 as a limited-precision decimal fraction will not express the number exactly, but this approximate form of expression may be just what the user wants to see.

The syntax of numbers is described formally in section 7.1.1. See section 6.5.6 for many examples of representations of numbers.

A numerical constant may be represented in binary, octal, decimal, or hex by the use of a radix prefix. The radix prefixes are #b (binary), #o (octal), #d (decimal), and #x (hex). 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 representation of a numerical constant has no exactness prefix, the constant may be assumed to be exact or inexact at the discretion of the implementation, except that integers expressed without decimal points and without use of exponential notation are assumed exact.

In systems with both single and double precision flonums we may want to specify which size we want to use to represent a constant internall. For example, we may want a constant that has the value of pi rounded to the single precision length, or we might want a long number that has the value 6/10. In either case, we are specifying an explicit way to represent an inexact number. For this purpose, we may express a number with a prefix that indicates short or long flonum representation:

```
#S3.14159265358979
```

Round to short — 3.141593

```
#L.6
```

Extend to long — .600000000000000

6.5.4. 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.

A fixnum, a bignum, or a ratnum) is operated upon so as to produce an inexact result (as by sqrt), and if the result is represented as a flonum, then the largest available flonum format must be used; but if the result is expressed as a ratnum then the rational approximation must have at least as much precision as the largest available flonum.

These numerical type predicates can be applied to any kind of argument, including non-numbers. They return true if the object is of the named type. In general, if a type predicate is true of a number then all higher type predicates are also true of that number. Not every system supports all of these types; for example, it is entirely possible to have a Scheme system that has only integers. Nonetheless every implementation of Scheme must have all of these predicates.

```
(zero? z)
(positive? x)
(negative? x)
(even? n)
(odd? n)
(exact? z)
(inexact? z)
```

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

```
(= z1 z2)
(< x1 x2)
(> x1 x2)
(<= x1 x2)
(>= x1 x2)
```

Some implementations allow these procedures to take many arguments, to facilitate range checks. These procedures return #t if their arguments are (respectively): numerically equal, monotonically increasing, monotonically decreasing, monotonically nondecreasing, or monotonically nonincreasing. Warning: on inexact numbers the equality tests will give unreliable results, and the other numerical comparisons will be useful only heuristically; when in doubt, consult a numerical analyst.

```
(max x1 x2)
(max x1 x2 ...)
(min x1 x2)
(min x1 x2 ...)
```

These procedures return the maximum or minimum of their arguments.

```
(+ z1 z2)
(+ z1 ...)
(*) z1 z2
(*) z1 ...)
```

These procedures return the sum or product of their arguments.
\[ \text{(gcd 32 -36)} \quad \Rightarrow \quad 4 \]
\[ \text{(gcd)} \quad \Rightarrow \quad 0 \]
\[ \text{(lcm 32 -36)} \quad \Rightarrow \quad 288 \]
\[ \text{(lcm)} \quad \Rightarrow \quad 1 \]

These procedures create integers and rationals. Their results are exact if and only if their arguments are exact.

Floor returns the largest integer not larger than \( x \). Ceiling returns the smallest integer not smaller than \( x \). Truncate returns the integer of maximal absolute value not larger than the absolute value of \( x \). Round returns the closest integer to \( x \), rounding to even when \( x \) is halfway between two integers. With two arguments, rationalize produces the rational number with smallest denominator differing from \( x \) by no more than \( y \). With one argument, rationalize produces the best rational approximation to \( x \), preserving all of the precision in its representation.

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

\[ \text{(exp } z) \quad \text{procedure} \]
\[ \text{(log } z) \quad \text{procedure} \]
\[ \text{(sin } z) \quad \text{procedure} \]
\[ \text{(cos } z) \quad \text{procedure} \]
\[ \text{(tan } z) \quad \text{procedure} \]
\[ \text{(asin } z) \quad \text{procedure} \]
\[ \text{(acos } z) \quad \text{procedure} \]
\[ \text{(atan } z) \quad \text{procedure} \]
\[ \text{(atan } y x) \quad \text{procedure} \]

These procedures are part of every implementation that supports real numbers; they compute the usual transcendental functions. Log computes the natural logarithm of \( z \) (not the base 10 logarithm). Asin, acos, and atan...
compute arcsine (sin\(^{-1}\)), arccosine (cos\(^{-1}\)), and arctangent (tan\(^{-1}\)), respectively. The two-argument variant of atan computes (angle (make-rectangular x y)) (see below), even in implementations that don’t support complex numbers.

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

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

With log defined this way, the values of sin\(^{-1}\) z, cos\(^{-1}\) z, and tan\(^{-1}\) z are according to the following formulae:

\[
\sin^{-1} z = -i \log(iz + \sqrt{1 - z^2})
\]
\[
\cos^{-1} z = -i \log(z + i\sqrt{1 - z^2})
\]
\[
\tan^{-1} z = -i \log((1 + iz)\sqrt{1/(1 + z^2)})
\]

The above specification follows [43], which in turn follows [26]; refer to these sources for more detailed discussion of branch cuts, boundary conditions, and implementation of these functions.

\[(\text{sqrt } z)\] procedure

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

\[(\text{expt } z_1 \ z_2)\] procedure

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

\[z_1^{z_2} = e^{z_2 \log z_1}\]

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

\[(\text{make-rectangular } x_1 \ x_2)\] procedure

\[(\text{make-polar } x_3 \ x_4)\] procedure

\[(\text{real-part } z)\] procedure

\[(\text{imag-part } z)\] procedure

\[(\text{magnitude } z)\] procedure

\[(\text{angle } z)\] procedure

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

\[z = x_1 + x_2i = x_3 \cdot e^{ix_4}\]

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

Note: Magnitude is the same as abs, but abs must be present in all implementations, whereas magnitude will only be present in implementations that support complex numbers.

\[(\text{exact->inexact } z)\] procedure

\[(\text{inexact->exact } z)\] procedure

Exact->inexact returns an inexact representation of \(z\), which is a fairly harmless thing to do. Inexact->exact returns an exact representation of \(z\). Since the law of “garbage in, garbage out” remains in force, inexact->exact should not be used casually.

6.5.5. Numerical input and output

\[(\text{number->string } number \ \text{format})\] procedure

The conventions used to produce the printed representation of a number can be specified by a format, as described in section 6.5.6. The procedure number->string takes a number and a format and returns as a string the printed representation of the given number in the given format. This procedure will mostly be used by sophisticated users and in system programs. In general, a naive user will need to know nothing about the formats because the system printer will have reasonable default formats for all types of numbers.

\[(\text{string->number } string \ \text{exactness} \ \text{radix})\] procedure

The system reader will construct reasonable default numerical types for numbers expressed in each of the formats it recognizes. A user who needs control of the coercion from strings to numbers will use string->number. Exactness must be a symbol, either E (for exact) or I (for inexact). Radix must also be a symbol: B for binary, O for octal, D for decimal, and X for hexadecimal. Returns a number of the maximally precise representation expressed by the given string. It is an error if string does not express a number according to the grammar in section 7.1.1.

6.5.6. Formats

A format is a list beginning with a format descriptor, which is a symbol such as sci. Following the descriptor are parameters used by that descriptor, such as the number of significant digits to be used. Default values are supplied for any parameters that are omitted. Modifiers may appear after the parameters, such as the radix and exactness formats described below, which themselves take parameters. Details of particular formats such as sci and fix are given in section 6.5.7.

For example, the format (sci 5 2 (exactness s)) specifies that a number is to be expressed in Fortran scientific format with 5 significant places and two places after
the radix point, and that its exactness prefix is to be suppressed.

In the following examples, the comment shows the format that was used to produce the output shown:

```
123 +123 -123 ; (int)
123456789012345678901234567 ; (int)
355/113 +355/113 -355/113 ; (rat)
+123.45 -123.45 ; (fix 2)
3.14159265358979 ; (fix 14)
3.14159265358979 ; (flo 15)
123.450 ; (flo 6)
-123.45e-1 ; (sci 5 2)
355/113 +355/113 -355/113 ; (rat)
123e3 123e-3 -123e-3 ; (sci 3 0)
-1+2i ; (rect (int) (int))
1.2@1.570796 ; (polar (fix 1)
```

A format may specify that a number should be expressed in a particular radix. The radix prefix may also be suppressed. For example, one may express a complex number in polar form with the magnitude in octal and the angle in decimal as follows:

```
#o1.2@#d1.570796327 ; (polar (flo 2 (radix o)) ; (flo (radix d s))
```

A format may specify that a number should be expressed with an explicit exactness prefix ((exactness e)), or it may force the exactness to be suppressed ((exactness s)). For example, the following are ways to express an inexact value for pi:

```
#i355/113 ; (rat (exactness e))
355/113 ; (rat (exactness s))
#i3.1416 ; (fix 4 (exactness e))
```

An attempt to produce more digits than are available in the internal machine representation of a number will be marked with a "#" filling the extra digits. This is not a statement that the implementation knows or keeps track of the significance of a number, just that the machine will flag attempts to produce 20 digits of a number that has only 15 digits of machine representation:

```
3.14158265358979##### ; (flo 20 (exactness s))
```

### 6.5.7. Details of formats

The format descriptors are:

- **(int)**
  - Express as an integer. The radix point is implicit. If there are not enough significant places then insignificant digits will be flagged. For example, an inexact integer 6.0238·10^23 (represented internally as a 7 digit flonum) would be printed as
    
    60238000000000000000

- **(rat n)**
  - Express as a rational fraction. n specifies the largest denominator to be used in constructing a rational approximation to the number being expressed. If n is omitted it defaults to infinity.

- **(fix n)**
  - Express with a fixed radix point. n specifies the number of places to the right of the radix point. n defaults to the size of a single-precision flonum. If there are not enough significant places, then insignificant digits will be flagged. For example, an inexact 6.0238·10^23 (represented internally as a 7 digit flonum) would be printed with a (fix 2) format as

    60238000000000000000

- **(flo n)**
  - Express with a floating radix point. n specifies the total number of places to be displayed. n defaults to the size of a single-precision flonum. If the number is out of range, it is converted to (sci). (flo h) expresses n in floating point format heuristically for human consumption.

- **(sci n m)**
  - Express in exponential notation. n specifies the total number of places to be displayed. n defaults to the size of a single-precision flonum. m specifies the number of places to the right of the radix point. m defaults to n − 1. (sci h) does heuristic expression.

- **(rect r i)**
  - Express as a rectangular form complex number. r and i are formats for the real and imaginary parts respectively. They default to (heur).

- **(polar m a)**
  - Express as a polar form complex number. m and a are formats for the magnitude and angle respectively. m and a default to (heur).

- **(heur)**
  - Express heuristically using the minimum number of digits required to get an expression that when coerced back to a number produces the original machine representation. Exact numbers are expressed as (int) or (rat). Inexact numbers are expressed as (flo h) or (sci h) depending
on their range. Complex numbers are expressed in (rect). This is the normal default of the system printer.

The following modifiers may be added to a numerical format specification:

(exactness s) format
This controls the expression of the exactness prefix of a number. s must be a symbol, either E or S, indicating whether the exactness is to be expressed or suppressed, respectively. If no exactness modifier is specified for a format then the exactness is by default suppressed.

(radix r s) format
This forces a number to be expressed in the radix r. r may be the symbol B (binary), O (octal), D (decimal), or X (hex). s must be a symbol, either E or S, indicating whether the radix prefix is to be expressed or suppressed, respectively. If no radix modifier is specified for a format then the default is decimal and the prefix is suppressed.

6.6. Characters
Characters are objects that represent printed characters such as letters and digits. There is no requirement that the data type of characters be disjoint from other data types; implementations are encouraged to have a separate character data type, but may choose to represent characters as integers, strings, or some other type.

Characters are written using the notation \#\langle character\rangle or \#\langle character name\rangle. 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 \#\langle character\rangle, but not in \#\langle character name\rangle. If \langle character\rangle in \#\langle character\rangle is alphabetic, then the character following \langle character\rangle must be a delimiter character such as a space or parenthesis. This rule resolves the ambiguous case where, for example, the sequence of characters “\#\space” could be taken to be either a representation of the space character or a representation of the character “\#\s” followed by a representation of the symbol “\space.”

Characters written in the \#\langle notation are self-evaluating. That is, they do not have to be quoted in programs. The \#\langle notation is not an essential part of Scheme, however. Even implementations that support the \#\langle notation for input do not have to support it for output.

Some of the procedures that operate on characters ignore the difference between upper case and lower case. The procedures that ignore case have the suffix “-ci” (for “case insensitive”). If the operation is a predicate, then the “-ci” suffix precedes the “?” at the end of the name.

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

(char=? char1 char2) procedure
(char<? char1 char2) procedure
(char>? char1 char2) procedure
(char<=? char1 char2) procedure
(char=>? char1 char2) 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.

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

(char-ci=? char1 char2) procedure
(char-ci<? char1 char2) procedure
(char-ci>? char1 char2) procedure
(char-ci<=? char1 char2) procedure
(char-ci=>? char1 char2) 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 numeric predicates.

(char-alphabetic? char) procedure
(char-numeric? char) procedure
(char-whitespace? char) procedure

These procedures return #t if their arguments are alphabetic, numeric, or whitespace 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 white space characters are space, tab, line feed, form feed, and carriage return.

- **(char-upper-case? letter)** procedure
- **(char-lower-case? letter)** procedure

*Letter* must be an alphabetic character. These procedures return *#t* if their arguments are upper case or lower case characters, respectively, otherwise they return *#f*.

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

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

\[(char<=? a b) \Rightarrow \#t \quad \text{and} \quad (\leq (char->integer a) (char->integer b)) \Rightarrow \#t\]

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

\[(\leq (char->integer x) (char->integer y)) \Rightarrow \#t\]

- **(char-upcase char)** procedure
- **(char-downcase char)** 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. In some implementations of Scheme they are immutable; other implementations provide destructive procedures such as string-set! that alter string objects.

Strings are written as sequences of characters enclosed within double quotes (*"*). 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 may continue from one line to the next, but this is usually a bad idea because the exact effect may vary from one computer system to another.

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 the suffix “-ci” (for “case insensitive”). If the operation is a predicate, then the “-ci” suffix precedes the “?” at the end of the name.

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

- **(make-string k)** procedure
- **(make-string k char)** procedure
  *k* must be a non-negative integer, and *char* must be a character. 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-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)** procedure
  *k* must be a valid index of *string*. String-set! stores *char* in element *k* of *string* and returns an unspecified value.

- **(string=? string1 string2)** essential procedure
- **(string-ci=? string1 string2)** procedure
  Returns #t if the two strings are the same length and contain the same characters in the same positions, otherwise
returns #f. String-ci=? treats upper and lower case letters as though they were the same character, but string=? treats upper 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)  procedure
(string-ci<=? string1 string2)  procedure
(string-ci=>? string1 string2)  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 numeric 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 string1 string2)  essential procedure
(string-append string ... )  procedure
Returns a new string whose characters form the concatenation of the given strings.

(string->list string)  essential procedure
(list->string chars)  essential procedure
String->list returns a newly allocated list of the characters that make up the given string. List->string returns a string formed from the characters in the list chars. String->list and list->string are inverses so far as equal? is concerned. Implementations that provide destructive operations on strings should ensure that the result of list->string is newly allocated.

(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 mutable structures whose elements are indexed by integers.

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

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

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

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

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

⇒ #(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.

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

(vector 'a 'b 'c)  ⇒ #(a b c)

(vector-length vector)  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 2 3 5 8 13 21) 5)  ⇒ 8

(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->list vector)  essential procedure
(list->vector list)  essential procedure

Vector->list returns a newly created list of the objects contained in the elements of vector. List->vector returns a newly created vector initialized to the elements of the list.

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

(vector-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? (lambda (x) (* x x)))  ⇒ #t
(procedure? (lambda (x) (expt 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 1 2 3))  ⇒ 7

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

((compose sqrt *) 12 75)  ⇒ 30

(map proc list)  essential procedure
(map proc list1 list2 ...)  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. The 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 (n) (set! count (+ count 1))
         count)
       '(a b c)))  ⇒ unspecified

(for-each proc list)  essential procedure
(for-each proc list1 list2 ...)  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 without any recomputation.

(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)))
6. Standard procedures

(define head car)
(define tail
  (lambda (stream) (force (cdr stream))))
(head (tail (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 the value of a promise is computed at most once.

(define count 0)
(define p (delay (begin (set! count (+ count 1)) (* x 3))))
(define x 5)
count =⇒ 0
p =⇒ a promise
p (force p) =⇒ 15
p =⇒ a promise, still
count =⇒ 1
(p (force p)) =⇒ 15
(count) =⇒ 1

Here is a possible implementation of delay and force. 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 ((already-run? #f) (result #f))
      (lambda ()
        (cond ((not already-run?)
               (set! result (proc))
               (set! already-run? #t)))
        result))))

Promises are implemented here as procedures of no arguments, and force simply calls its argument.

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

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:

(call-with-current-continuation proc)

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 created by call-with-current-continuation 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 will take the result, print it on the screen, prompt for the next input, evaluate it, and so on forever. Most of the time the continuation includes actions specified by user code, as in a continuation that will take the result, multiply it by the value stored in a local variable, add seven, and give the answer to the top level continuation to be printed. Normally 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 [21] invented a general purpose escape operator called the J-operator. John Reynolds [32] 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 device is a Scheme object that can deliver characters upon command, while an output device 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 Scheme will not close the port unless it can 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.

(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)  procedure

Takes a string naming an existing file and returns an input port capable of delivering characters from the file. If the file cannot be opened, an error is signalled.

(open-output-file filename)  procedure

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

### 6.10.2. Input

**{read}** essential procedure

_{read port} _

_{read-char port}_ essential procedure

**Rationale:** *Read* converts written representations of Scheme objects into the objects themselves. That is, it is a parser for the nonterminal ⟨datum⟩ (see section 7.1.2). *Read* returns the next object parsable from the given input _port_, updating _port_ to point to the first character past the end of the written 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 written representation, but the written 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.

### 6.10.3. Output

**{write}** essential procedure

_{write obj}_

_{write obj 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.

_{newline}_ essential procedure

_{newline port}_

_{write-char char}_ essential procedure

_{write-char char 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.

_{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 a character or an object that can be read in using **read**.
6.10.4. User interface

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

(load filename) essential procedure

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.

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

(transcript-on filename) procedure

(transcript-off) procedure

Filename must be a string naming an output file to be created. The effect of transcript-on is to open the named file for output, and to cause a transcript of subsequent interaction between the user and the Scheme system to be written to the file. The transcript is ended by a call to transcript-off, which closes the transcript file. Only one transcript may be in progress at any time, though some implementations may relax this restriction. The values returned by these procedures are unspecified.

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, characters, and dot) may be terminated by any (delimiter), but not necessarily by anything else.

(token) → ⟨identifier⟩ | ⟨boolean⟩ | ⟨number⟩
    | ⟨character⟩ | ⟨string⟩
    | ⟨digit⟩ | ⟨$⟩ | ⟨#⟩ | ⟨'⟩ | ⟨' '⟩ | ⟨@⟩ | ⟨\rangle.
(delimiter) → ⟨whitespace⟩ | ⟨⟩ | ⟨⟩
(whitespace) → ⟨space or newline⟩
(comment) → ⟨⟨space⟩⟩
(atmosphere) → ⟨⟨space⟩⟩ | ⟨⟨comment⟩⟩
(intertoken space) → ⟨⟨atmosphere⟩⟩

(peculiar identifier) → ⟨⟨identifier⟩⟩

(letters) → a | b | c | ... | z
(special initial) → ! | $ | % | & | * | / | : | < | = |
    | > | ? | \ | [ | ]
(subsequent) → ⟨⟨digit⟩⟩
(digit) → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
(special subsequent) → . | + | -
(peculiar identifier) → + | -
(expression keyword) → ⟨⟨expression keyword⟩⟩

| else | => | define
| unquote | unquote-splicing
(expression keyword) → quote | lambda | if
7. Formal syntax and semantics

7.1.2. External representations

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

(datum) → (simple datum) | (compound datum)

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

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

(boolean) → #t | #f

(character) → #\ ⟨any character⟩

(character name) → space | newline

(string) → " ⟨string element)* "

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

| " | \n
(number) → ⟨real⟩ | (real) + ⟨real⟩ i

| (real) - (real) i | (real) @ (real)

(real) → ⟨sign⟩ ⟨real⟩

(ureal) → ⟨ureal 10⟩ | ⟨ureal 2⟩ | ⟨ureal 8⟩ | ⟨ureal 10⟩ | ⟨ureal 16⟩

The following rules for ⟨ureal R⟩, ⟨uinteger R⟩, and ⟨prefix R⟩ should be replicated for R = 2, 8, 10, and 16:

⟨ureal R⟩ → ⟨prefix R⟩ ⟨uinteger R⟩ ⟨suffix⟩

| (prefix R) ⟨uinteger R⟩ / ⟨uinteger R⟩ ⟨suffix⟩

| (prefix R) . (digit R) + #* ⟨suffix⟩

| (prefix R) . (digit R) + . (digit R) * #* ⟨suffix⟩

| (prefix R) . (digit R) + #* . #* ⟨suffix⟩

(prefix R) → (radix R) ⟨exactness⟩ ⟨precision⟩

| (radix R) ⟨precision⟩ ⟨exactness⟩

| (exactness) (radix R) ⟨precision⟩

| (exactness) ⟨precision⟩ (radix R)

| (precision) (radix R) ⟨exactness⟩

| (precision) ⟨exactness⟩ (radix R)

(uinteger R) → ⟨digit R⟩+ #*

(sign) → ⟨empty⟩ | + | -

(suffix) → ⟨empty⟩ | e ⟨sign⟩ ⟨digit⟩+

(exactness) → ⟨empty⟩ | #i | #e

(precision) → ⟨empty⟩ | #s | #l

(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⟩ | a | b | c | d | e | f

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⟩

| ⟨symbol⟩ → ⟨identifier⟩

(compound datum) → ⟨list⟩ | ⟨vector⟩

(list) → ((datum)* | ((datum) + . ⟨datum⟩)

(abbreviation) → (abbrev prefix ⟨datum⟩)

(abbrev prefix) → \ | , | @

(vector) → #((datum)*)
7.1.4. Quasiquotations

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

\[
\begin{align*}
\langle \text{quasiquotation} \rangle & \rightarrow \langle \text{quasiquotation} \rangle \ 1 \\
\langle \text{template} \rangle & \rightarrow \langle \text{expression} \rangle \\
\langle \text{quasiquotation} \rangle & \rightarrow * \langle \text{template} \rangle \\
\langle \text{template} \rangle & \rightarrow \langle \text{simple datum} \rangle \\
& \ | \langle \text{list template} \rangle \\
& \ | \langle \text{vector template} \rangle \\
& \ | \langle \text{unquotation} \rangle \\
\langle \text{list template} \rangle & \rightarrow \langle (\text{template or splice} \ D)^* \rangle \\
& \ | \langle (\text{template or splice} \ D)^* \rangle \ * \langle \text{template} \rangle \\
& \ | \langle \text{quasiquotation} \rangle \ \langle \text{quasiquotation} \rangle \\
\langle \text{vector template} \rangle & \rightarrow \# \langle (\text{template or splice} \ D)^* \rangle \\
\langle \text{unquotation} \rangle & \rightarrow , \langle \text{template} \ D \ 1 \rangle \\
& \ | \langle \text{unquote} \ \langle \text{template} \ D \ 1 \rangle \rangle \\
\langle \text{template or splice} \rangle & \rightarrow \langle \text{template} \rangle \\
& \ | \langle \text{splicing unquotation} \rangle \\
\langle \text{splicing unquotation} \rangle & \rightarrow , \$ \langle \text{template} \ D \ 1 \rangle \\
& \ | \langle \text{unquote-splicing} \ \langle \text{template} \ D \ 1 \rangle \rangle \\
\end{align*}
\]

In \( \langle \text{quasiquotation} \rangle \)'s, a \( \langle \text{list template} \ D \rangle \) can sometimes be confused with an \( \langle \text{unquotation} \ D \rangle \) or \( \langle \text{splicing unquotation} \ D \rangle \). The interpretation as an \( \langle \text{unquotation} \rangle \) or \( \langle \text{splicing unquotation} \ D \rangle \) takes precedence.

7.1.5. Programs and definitions

\( \langle \text{program} \rangle \rightarrow \langle \text{command or definition} \rangle \)

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 [50]; the notation is summarized below:

\[
\begin{align*}
\langle \ldots \rangle & \text{ sequence formation} \\
s \downarrow k & \text{ kth member of the sequence } s \text{ (1-based)} \\
\# s & \text{ length of sequence } s \\
s \underline k t & \text{ concatenation of sequences } s \text{ and } t \\
s \uparrow k & \text{ drop the first } k \text{ members of sequence } s \\
t \rightarrow a, b & \text{ McCarthy conditional “if } t \text{ then } a \text{ else } b” \\
\rho[x/i] & \text{ substitution “} p \text{ with } x \text{ for } i” \\
x \in D & \text{ injection of } x \text{ into domain } D \\
x | D & \text{ projection of } x \text{ to domain } D
\end{align*}
\]

To avoid special treatment for a top-level environment, the semantics assumes that environments assign locations to all variables.

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 order of evaluation within a call is unspecified. We mimic that here by applying arbitrary permutations \textit{permute} and \textit{unpermute}, which must be inverses, to the arguments in a call before and after they are evaluated. This still requires that the order of evaluation be constant throughout a program (for any given number of arguments), but it is a closer approximation to the intended semantics than a left-to-right evaluation would be.

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

The semantics in this section was translated by machine from an executable version of the semantics written in Scheme itself.

7.2.1. Abstract syntax

\[
\begin{align*}
K & \in \text{Con} \quad \text{ constants, including quotations} \\
I & \in \text{Id} \quad \text{ identifiers (variables)} \\
E & \in \text{Exp} \quad \text{ expressions} \\
\Gamma & \in \text{Com} = \text{ Exp} \quad \text{ commands}
\end{align*}
\]

\( \text{Exp} \rightarrow \text{K} \ | \ I \ | \ \langle \text{E}_0 \ 	ext{E}^* \rangle \\
\ | \ \langle \text{lambda} \ (I^*) \ 	ext{I}^* \ \text{E}_0 \rangle \\
\ | \ \langle \text{lambda} \ (I^* \ I) \ 	ext{I}^* \ 	ext{E}_0 \rangle \)
7.2.3. Semantic functions

\[ \kappa \text{ (update (new } \sigma \mid L) \text{ unspecified } \sigma) } \]
\[ \text{wrong “out of memory” } \sigma \]

\[ \varepsilon[(\lambda (I^* \cdot I) \Gamma^* E_0)] = \lambda \rho k \cdot \lambda \sigma \cdot \text{new } \sigma \in L \rightarrow \]
\[ \text{send } ((\text{new } \sigma \mid L), \]
\[ \lambda \epsilon^* \cdot (\lambda \rho' \cdot C[I^*] \rho' (C[E_0] \rho' \epsilon')) \]
\[ (\text{extends } \rho \Gamma^* \Gamma' (\lambda \epsilon')) \]
\[ \epsilon^* \]
\[ (\# \Gamma^*), \]
\[ \text{wrong “too few arguments” in } E \]

\[ \varepsilon[(\lambda (I^* \cdot I) \Gamma^* E_0)] = \varepsilon[(\lambda (I^* \cdot I) \Gamma^* E_0)] \]

\[ \varepsilon[(I^* E_0)] = \lambda \rho k \cdot \text{single } (\lambda \epsilon \cdot \text{truish } \epsilon \rightarrow E[E_1] \rho k, \]
\[ E[E_2] \rho k) \]

\[ \varepsilon[(\set! I E)] = \lambda \rho k \cdot \text{single } (\lambda \epsilon \cdot \text{assign } (\text{lookup } \rho I) \]
\[ (\text{send unspecified } \kappa)) \]

\[ \kappa \]
\[ \text{Definition of } K \text{ deliberately omitted.} \]

\[ \varepsilon[K] = \lambda \rho k \cdot \text{send } (K[K],) \]
\[ \kappa \]

\[ \varepsilon[I] = \lambda \rho k \cdot \text{hold } (\text{lookup } \rho I) \]
\[ (\text{single } (\lambda \epsilon \cdot \epsilon = \text{undefined } \rightarrow \text{undefined variable}, \]
\[ \text{send } \epsilon \kappa)) \]

\[ \varepsilon[(E_0, E_1)] = \lambda \rho k \cdot \text{permute } (\langle E_0 \rangle \mid E_1, \rangle \]
\[ \kappa \]

\[ \rho \]
\[ (\lambda \epsilon^* \cdot (\lambda \epsilon^* \cdot \text{apply } (\epsilon^* \downarrow 1) (\epsilon^* \uparrow 1) \kappa) \]
\[ (\text{unpermute } \epsilon^*)) \]

\[ \varepsilon[(\lambda (I^* \cdot I) \Gamma^* E_0)] = \lambda \rho k \cdot \lambda \sigma \cdot \text{new } \sigma \in L \rightarrow \]
\[ \text{send } ((\text{new } \sigma \mid L), \]
\[ \lambda \epsilon^* \cdot \text{tievals } (\lambda \epsilon^* \cdot (\lambda \rho' \cdot C[I^*] \rho' (C[E_0] \rho' \epsilon')) \]
\[ (\text{extends } \rho \Gamma^* \Gamma' (\lambda \epsilon')) \]
\[ \epsilon^*, \]
\[ \text{wrong “wrong number of arguments”}) \]
\[ \text{in } E \]

7.2.4. Auxiliary functions

lookup : U → Ide → L
lookup = \lambda \rho I \cdot I \rho

extends : U → Ide* → L* → U
extends = \lambda \rho I^* \alpha^* \cdot \#I^* = 0 \rightarrow \rho, \]
\[ \text{extends } \rho (\alpha (\downarrow 1)) (I^* \downarrow 1) (\lambda \rho (\alpha (\downarrow 1))) \]
\[ \text{wrong } : X \rightarrow C \text{ [implementation-dependent]} \]

send : E → K → C
send = \lambda \rho k \cdot \kappa(\epsilon)

single : (E → C) → K
single = \lambda \rho I^* \alpha^* \cdot \#\epsilon^* = 1 \rightarrow I^* (\psi (\epsilon^* \downarrow 1), \]
\[ \text{wrong “number of return values”} \]
new : S → (L + {error}) [implementation-dependent]
hold : L → K → C
hold = λα . send(σ ↓ 1)α
assign : L → E → C → C
assign = λε . (update α)ε
update : L → E → S → S
update = λαε . σ[(ε, true)/α]
tievals : (L* → C) → E* → C
tievals =
λψεσ. #e = 0 → ψ(σ)σ
new σ ∈ L → tievals(λα . ψ((new σ | L) $ α))
(σ* ↑ 1)
(update(new σ | L)(e* ↓ 1)σ),
wrong “out of memory” σ
tievalsrest : (L* → C) → E* → N → C
tievalsrest =
λψeν. list (dropfirst e*ν)
(singlε(λε . tievals ψ ((takefirst e*ν) $ (ε) )))
dropfirst = λn . n = 0 → l, dropfirst(l ↑ 1)(n - 1)
takefirst = λn . n = 0 → l, takefirst(l ↓ 1)(n - 1)
true : E → T
true = λε . (ε = false ∨ ε = null) → false, true
permute : Exp* → Exp* [implementation-dependent]
unpermute : E* → E* [inverse of permute]
apply : E → E* → K → C
apply =
λεeκ. ε ∈ F → (ε | F ↓ 2)e*κ, wrong “bad procedure”
onearg : (E → K → C) → (E* → K → C)
onearg =
λεeκ. #e = 1 → ζ(e* ↓ 1)κ,
wrong “wrong number of arguments”
twoarg : (E → E → K → C) → (E* → K → C)
twoarg =
λεeκ. #e = 2 → ζ(e* ↓ 1)(e* ↓ 2)κ,
wrong “wrong number of arguments”
list : E* → K → C
list =
λεκ. #e = 0 → send null κ,
list(e* ↑ 1) = (singlε(λε . cons(e* ↓ 1, ε)κ))
cons : E* → K → C
cons =
twoarg(λε1e2κ . new σ ∈ L →
λα . new σ' ∈ L →
send((new σ | L, new σ' | L) in E))κ
wrong “out of memory” σ)
(update(new σ | L)e2κ),
wrong “out of memory” σ)
less : E* → K → C
less =
twoarg(λε1e2κ . (ε1 ∈ R ∧ e2 ∈ R) →
send(ε1 | R < e2 | true, false)κ,
wrong “non-numeric argument to <”)
add : E* → K → C
add =
twoarg(λε1e2κ . (ε1 ∈ R ∧ e2 ∈ R) →
send((ε1 | R + e2 | R) in E)κ,
wrong “non-numeric argument to +”)
car : E* → K → C
car =
onearg(λεκ . ε ∈ Eρ → hold(ε | Eρ ↓ 1)κ,
wrong “non-pair argument to car”)
cdr : E* → K → C [similar to car]
setcar : E* → K → C
setcar =
twoarg(λε1e2κ . ε1 ∈ Eρ → assign(ε1 | Eρ ↓ 1)
ε2
(send unspecified κ),
wrong “non-pair argument to set-car!”)
eqv : E* → K → C
eqv =
twoarg(λε1e2κ . (ε1 ∈ M ∧ e2 ∈ M) →
send(ε1 | M = e2 | M → true, false)κ,
(ε1 ∈ Q ∧ e2 ∈ Q) →
send(ε1 | Q = e2 | Q → true, false)κ,
(ε1 ∈ H ∧ e2 ∈ H) →
send(ε1 | H = e2 | H → true, false)κ,
(ε1 ∈ R ∧ e2 ∈ R) →
send(ε1 | R = e2 | R → true, false)κ,
(ε1 ∈ Eρ ∧ e2 ∈ Eρ) →
send((λp1p2 . ((p1 ↓ 1) = (p2 ↓ 1) ∧
(p1 ↓ 2) = (p2 ↓ 2)) → true,
false)
(ε1 | Eρ)
(ε2 | Eρ))κ,
(ε1 ∈ Eρ ∧ e2 ∈ Eρ) → . . .
(ε1 ∈ Eρ ∧ e2 ∈ Eρ) → . . .
(ε1 ∈ F ∧ e2 ∈ F) →
send(ε1 | F ↓ 1) = (ε2 | F ↓ 1) → true, false)κ,
send false κ)
apply : E* → K → C
apply =
twoarg(λε1e2κ . ε1 ∈ F → valueslist(ε2)(λε . apply ε1e*κ),
wrong “bad procedure argument to apply”)
7. Formal syntax and semantics

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.

\[
\begin{align*}
\text{cond} & ((\text{test}) \ (\text{sequence}) \\
& \quad \text{clause}_2 \ldots) \\
& \equiv (\text{if} \ (\text{test}) \\
& \quad (\text{begin} \ (\text{sequence}) \\
& \quad \text{clause}_2 \ldots))
\end{align*}
\]

\[
\begin{align*}
\text{cond} & ((\text{test}) \Rightarrow (\text{recipient}) \\
& \quad \text{clause}_2 \ldots) \\
& \equiv (\text{or} \ (\text{test}) \ (\text{cond} \ (\text{clause}_2) \ldots))
\end{align*}
\]

\[
\begin{align*}
\text{cond} & ((\text{test}) \Rightarrow (\text{recipient}) \\
& \quad \text{clause}_2 \ldots) \\
& \equiv (let \ ((\text{test-result} \ (\text{test})) \\
& \quad (\text{thunk2} \ (\text{lambda} \ () \ (\text{recipient}))) \\
& \quad (\text{thunk3} \ (\text{lambda} \ () \ (\text{cond} \ (\text{clause}_2) \ldots)))) \\
& \quad (\text{if} \ \text{test-result} \\
& \quad (\text{thunk2} \ \text{test-result}) \\
& \quad (\text{thunk2})))
\end{align*}
\]

\[
\begin{align*}
\text{cond} & ((\text{test}) \Rightarrow (\text{recipient}) \\
& \quad \text{clause}_2 \ldots) \\
& \equiv (\text{begin} \ (\text{sequence}))
\end{align*}
\]

\[
\begin{align*}
\text{cond} & \equiv (\text{some expression returning an unspecified value})
\end{align*}
\]

\[
\begin{align*}
\text{case} & (\text{key}) \\
& ((d_1 \ldots) \ (\text{sequence})) \\
& \ldots \\
& \equiv (let \ ((\text{key} \ (\text{key}))) \\
& \quad (\text{thunk1} \ (\text{lambda} \ () \ (\text{sequence}))) \\
& \quad \ldots) \\
& \quad (\text{cond} \ ((\text{memv} \ \text{key} \ '(d_1 \ldots) \ (\text{thunk1}))) \\
& \quad \ldots))
\end{align*}
\]

\[
\begin{align*}
\text{case} & (\text{key}) \\
& ((d_1 \ldots) \ (\text{sequence})) \\
& \ldots \\
& \equiv (\text{let} \ ((\text{key} \ (\text{key})))
\end{align*}
\]

where \(\text{temp1}, \text{temp2}, \ldots\) are variables, distinct from \(\text{variable1}, \ldots\), which do not free occur in the original \(\text{init}\) expressions, and \(\text{undefined}\) is an expression which returns something which 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 \text{let} expression in the expansion is not strictly necessary, but it serves to preserve the property that the \text{init} expressions are evaluated in an arbitrary order.

\[
\begin{align*}
\text{let*} & () \ (\text{body}) \\
& \equiv (\text{lambda} \ () \ (\text{body}))
\end{align*}
\]

\[
\begin{align*}
\text{let*} & ((\text{variable1} \ (\text{init1})) \\
& \quad (\text{variable2} \ (\text{init2})) \\
& \quad \ldots) \ (\text{body}) \\
& \equiv (\text{let} \ ((\text{variable1} \ (\text{init1}))) \\
& \quad (\text{let*} \ ((\text{variable2} \ (\text{init2})) \\
& \quad \ldots) \ (\text{body})))
\end{align*}
\]

\[
\begin{align*}
\text{letrec} & ((\text{variable1} \ (\text{init1})) \\
& \quad \ldots) \ (\text{body}) \\
& \equiv (\text{let} \ ((\text{variable1} \ (\text{undefined}))) \\
& \quad \ldots) \\
& \quad (\text{let} \ ((\text{temp1} \ (\text{init1}))) \\
& \quad \ldots) \\
& \quad (\text{set!} \ (\text{variable1} \ (\text{temp1}))) \\
& \quad \ldots) \ (\text{body})
\end{align*}
\]
(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))))

(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) ...) (body))))
       (variable0))
       (init1) ...)

(delay (expression))
≡ ((make-promise) (lambda () (expression)))

where `(make-promise)` is an expression evaluating to some procedure which behaves appropriately with respect to the `force` procedure; see section 6.9.

**NOTES**

**Language changes**

This section enumerates the changes that have been made to Scheme since the "Revised revised report" [4] was published.

- The character ^ (circumflex) is now an extended alphabetic character
- The objects returned by literal expressions are permitted to be immutable
- The list to which a rest-argument becomes bound must be newly allocated
- Do variables are updated by rebinding rather than by assignment
- New expression type: `delay`
- Quasiquote (backquote) has been improved in several ways: vectors are allowed; nesting is allowed; and an external syntax for quasiquote expressions (analogous to that for `quote`) has been defined
- The semantics of definitions of the form `(define (variable (formals)) (body))` no longer involves an implicit `rec` or `letrec`
- The "curried" definition syntax has been removed
- The boolean constants are now written `#t` and `#f` instead of `#!true` and `#!false`
- The syntax `#!null` (for the empty list) has been removed
- New procedures: `boolean?`, `procedure?`, and `force`
- The value of `eq?` on numbers and characters is now unspecified
- `Eq?` and `eqv?` now explicitly permit operationally equivalent procedures to be identified
- `Eqv?` distinguishes exact numbers from inexact ones, even if they are equal according to `=`
- List, string, and vector indexes must be `exact` integers
- `Atan` now admits either one or two arguments
- Expression types removed: `named-lambda`, `rec`, `sequence`
- Procedures removed: `append!`, `string-null?`, `substring-fill!`, `substring-move-left!`, `substring-move-right!`, `object-hash`, `object-unhash`, `1+`, `-1+`
Keywords as variable names

Some implementations allow arbitrary syntactic keywords to be used as variable names, instead of reserving them, as this report would have it. But this creates ambiguities in the interpretation of expressions: for example, in the following, it’s not clear whether the expression \((\text{if } 1 \ 2 \ 3)\) should be treated as a procedure call or as a conditional.

\[
\text{(define if list)}
\]
\[
\text{(if } 1 \ 2 \ 3) \quad \Rightarrow \ 2 \ or \ (1 \ 2 \ 3)
\]

These ambiguities are usually resolved in some consistent way within any given implementation, but no particular treatment stands out as being clearly superior to any other, so these situations were excluded for the purposes of this report.

Macros

Scheme does not have any standard facility for defining new kinds of expressions.

The ability to alter the syntax of the language creates numerous problems. All current implementations of Scheme have macro facilities that solve those problems to one degree or another, but the solutions are quite different and it isn’t clear at this time which solution is best, or indeed whether any of the solutions are truly adequate. Rather than standardize, we are encouraging implementations to continue to experiment with different solutions.

The main problems with traditional macros are: They must be defined to the system before any code using them is loaded; this is a common source of obscure bugs. They are usually global; macros can be made to follow lexical scope rules, but many people find the resulting scope rules confusing. Unless they are written very carefully, macros are vulnerable to inadvertent capture of free variables; to get around this, for example, macros may have to generate code in which procedure values appear as quoted constants. There is a similar problem with syntactic keywords if the keywords of special forms are not reserved. If keywords are reserved, then either macros introduce new reserved words, invalidating old code, or else special forms defined by the programmer do not have the same status as special forms defined by the system.

EXAMPLE

\text{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 \text{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 \text{initial-state} provides an initial system state, and \(h\) is an initial guess for the length of the integration step.

The value returned by \text{integrate-system} is an infinite stream of system states.

\[
\text{(define integrate-system}
\begin{align*}
&\text{(lambda (system-derivative initial-state h))} \\
&\text{(let ((next (runge-kutta-4 system-derivative h)))} \\
&\text{(letrec ((states}
&\text{(cons initial-state} \\
&\text{(delay (map-streams next states))))}
&\text{states))))}
\end{align*}
\]

\text{Runge-Kutta-4} takes a function, \(f\), that produces a system derivative from a system state. \text{Runge-Kutta-4} produces a function that takes a system state and produces a new system state.

\[
\text{(define runge-kutta-4}
\begin{align*}
&\text{(lambda (f h))} \\
&\text{(let ((+h (scale-vector h))} \\
&\text{(+2 (scale-vector 2)))} \\
&\text{(+1/2 (scale-vector (/ 1 2)))} \\
&\text{(+1/6 (scale-vector (/ 1 6))))} \\
&\text{(lambda (y)} \\
&\text{;; \(y\) is a system state} \\
&\text{(let* ((k0 (*h (f y)))} \\
&\text{(k1 (*h (f (add-vectors y (*1/2 k0))))))} \\
&\text{(k2 (*h (f (add-vectors y (*1/2 k1))))))} \\
&\text{(k3 (*h (f (add-vectors y k2))))))} \\
&\text{(add-vectors y} \\
&\text{(*1/6 (add-vectors k0} \\
&\text{(*2 k1))} \\
&\text{(*2 k2))} \\
&\text{k3))))))))
\end{align*}
\]

\[
\text{(define elementwise}
\begin{align*}
&\text{(lambda (f)} \\
&\text{(lambda vectors} \\
&\text{generate-vector} \\
&\text{(vector-length (car vectors))} \\
&\text{(lambda (i)} \\
&\text{(apply f} \\
&\text{(map (lambda (v) (vector-ref v i))} \\
&\text{vectors))))))
\end{align*}
\]

\[
\text{(define generate-vector}
\begin{align*}
&\text{(lambda (size proc)}
\end{align*}
\]
Revised Scheme

(let ((ans (make-vector size)))
  (letrec ((loop
              (lambda (i)
                (cond ((= i size) ans)
                      (else
                       (vector-set! ans i (proc i))
                       (loop (+ i 1)))))))
    (loop 0))))

(define add-vectors (elementwise +))

(define scale-vector
  (lambda (s)
    (elementwise (lambda (x) (* x s))))

Map-streams is analogous to map: it applies its first argument (a procedure) to all the elements of its second argument (a stream).

(define map-streams
  (lambda (f s)
    (cons (f (head s))
          (force (map-streams f (tail s))))))

Infinite streams are implemented as pairs whose car holds the first element of the stream and whose cdr holds a promise to deliver the rest of the stream.

(define head car)
(define tail
  (lambda (stream) (force (cdr stream))))

The following illustrates the use of integrate-system in integrating the system

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

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

which models a damped oscillator.

(define damped-oscillator
  (lambda (R L C)
    (lambda (state)
      (let ((Vc (vector-ref state 0))
            (Il (vector-ref state 1)))
        (vector (- 0 (+ (/ Vc (* R C)) (/ Il C))
                 (/ Vc L))))))

(define the-states
  (integrate-system
    (damped-oscillator 10000 1000 .001)
    '#(1 0)
    .01))

BIBLIOGRAPHY AND REFERENCES


[40] 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.


ALPHABETIC INDEX OF DEFINITIONS OF CONCEPTS, KEYWORDS, AND PROCEDURES

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

! 5
', 7; 15
* 19; 7
+ 19; 4, 7, 12, 27, 34
, 11; 15
- 20; 4
/ 20
; 5
< 19; 34
<= 19; 24
= 19; 13, 36
=> 8
> 19
>= 19
? 5

abs 20; 12, 21
acos 20
and 9; 12
angle 21
append 16
apply 26; 34
asin 20
assoc 16; 17
assq 16; 17
assv 16; 13, 17
at-sign 11
atan 20; 21, 36

#b 19, 31
backquote 11
begin 10; 36
binding 5
binding construct 6
boolean? 12; 36
bound 6
caar 15
caddr 15
cadr 15
call 7
call by need 10
call-with-current-continuation 27; 28, 34
call-with-input-file 28
call-with-output-file 28
call/cc 28
car 15; 14, 34
case 8
catch 28
cdddar 15
cdddr 15
cdr 15; 14, 27
ceiling 20
char->integer 24
char-alphabetic? 23
char-ci<=? 23
char-ci<? 23
char-ci=? 23
char-ci>=? 23
char-ci>? 23
char-downcase 24
char-lower-case? 24
char-numeric? 23
char-ready 29
char-ready? 29
char-upcase 24
char-upper-case? 24
char-whitespace? 23
char<=? 23; 24
char<? 23; 25
char=? 23; 13
char=>? 23
char>? 23
char? 23
close-input-port 29
close-output-port 29
combination 7
comma 11
comment 5; 30
complex? 19
cond 8; 12
cons 15; 14
cos 20
current-input-port 28; 29, 30
current-output-port 28; 29, 30

#d 19
define 11; 12
delay 10; 26, 27, 36
denominator 20
display 29
do 10; 6, 12, 36
dotted pair 14

d
#e 19, 31
else 8
empty list 14; 12, 13, 15, 16, 36
eof-object? 29
eq? 14; 13, 16, 36
equal? 14; 13, 16, 25
equivalence predicate 13
eqv? 13; 8, 14, 15, 16, 17, 36
error  3
escape procedure  27
essential  3
even?  19
exact->inexact  21
exact?  19
exactness  18
exactness  23
exp  20
expt  21

#f  12
false  6; 12
fix  22
flo  22
floor  20
foo  4, 11
for-each  26
force  26; 10, 27, 36

gcd  20
gen-counter  13
gen-loser  13

heur  22

#i  19, 22, 31
identifier  4; 5, 17, 30
if  8; 12, 33, 35
imag-part  21
improper list  15
inexact->exact  21
inexact?  19
initial environment  12
input-port?  28
int  22
integer->char  24
integer?  19
integrate-system  37, 38
internal definition  12

keyword  4, 5, 30, 37

#l  19, 31
lambda  7; 12, 33, 35
lambda expression  6
last-pair  16
lazy evaluation  10
lcm  20

length  16
let  9, 10; 6, 12, 35
let*  9; 6, 12
letrec  9; 6, 10, 12
list  16; 25
list->string  25
list->vector  26

list-ref  16
list-tail  16
load  30
log  20

macros  37
magnitude  21
make-polar  21
make-promise  27
make-rectangular  21
make-string  24
make-vector  25
map  26; 38
map-streams  38
max  19
member  16; 17
memq  16; 13, 17
memv  16; 17, 35
min  19
modulo  20

mutation procedure  13

negative?  19
newline  29
nil  12
not  12
null?  16
number  17
number->string  21
number?  19
numerator  20

#o  19, 22, 31
odd?  19
open-input-file  28
open-output-file  28
operationally equivalent  13
or  9; 12
output-port?  28

pair  14
pair?  15
polar  22
positive?  19
predicate  13
procedure call  7
procedure?  26; 36
promise  10; 26

quasiquoite  11; 15
quote  7; 6, 15, 36
quotient  20

radix  23
rat  22
rational?  19
rationalize 20
read 29; 3, 6, 15, 17, 31
read-char 29
real-part 21
real? 19
rect 22
region 6; 8, 9, 10
remainder 20
return 28
reverse 16
round 20
runge-kutta-4 37

#s 19, 31
sci 22
sequence 10
set! 8; 11, 12, 33, 35
set-car! 15; 7, 13, 14, 34
set-cdr! 15; 14
sin 20
sqrt 21; 18, 19
string->list 25
string->number 21
string->symbol 17
string-append 25
string-ci<=? 25
string-ci>? 25
string-ci=? 24; 25
string-ci>=? 25
string-copy 25
string-fill! 25
string-length 24
string-ref 24
string-set! 24; 7, 17
string<=? 25
string<=? 25
string=? 24; 25
string>=? 25
string>=? 25
string? 24
substring 25
symbol->string 17
symbol? 17
syntactic keyword 4, 5, 30, 37

#t 12
t 12
tan 20
token 30
top level environment 12; 6
transcript-off 30
transcript-on 30
ture 6; 8, 12
truncate 20

unbound 6; 12
unquote 11, 15
unquote-splicing 11, 15
valid indexes 24; 25
values-list 34
variable 5; 4, 6, 31, 37
vector 25
vector->list 26
vector-fill! 26
vector-length 25
vector-ref 25
vector-set! 25
vector? 25
whitespace 5
with-input-from-file 28
with-output-to-file 28
write 29; 6, 11, 17
write-char 29

#x 19, 31

zero? 19