Revised Report on the Algorithmic Language Scheme

\[(Y F) = (F (Y F))\]

MIT Artificial Intelligence Laboratory
Revised\textsuperscript{4} Report on the Algorithmic Language Scheme

William Clinger and Jonathan Rees (Editors)

D. H. Bartley  R. Halstead  D. Oxley  G. J. Sussman
G. Brooks  C. Hanson  K. M. Pitman  M. Wand

Abstract

Data and procedures and the values they amass,
Higher-order functions to combine and mix and match,
Objects with their local state, the messages they pass,
A property, a package, the control point for a catch—
In the Lambda Order they are all first-class.
One Thing to name them all, One Thing to define them,
One Thing to place them in environments and bind them,
In the Lambda Order they are all first-class.

Keywords: Scheme, Lisp, functional programming, computer languages.

This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the laboratory's artificial intelligence research is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-89-J-3202 and by the National Science Foundation under grant number MIP-9001651.
With apologies to J. R. R. Tolkien.
Revised Report on the Algorithmic Language Scheme

WILLIAM CLINGER AND JONATHAN REES (Editors)

H. Abelson
N. I. Adams IV
D. H. Bartley
G. Brooks
R. K. Dybvig
D. P. Friedman
R. Halstead
C. Hanson
C. T. Haynes
E. Kohlbecker
D. Oxley
K. M. Pitman
G. J. Rozas
G. L. Steele Jr.
G. J. Sussman
M. Wand

Dedicated to the Memory of ALGOL 60

2 November 1991

SUMMARY

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

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

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

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

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

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

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

The report concludes with a bibliography and an alphabetic index.

CONTENTS

Introduction ................................................. 2
1. Overview of Scheme .................................... 3
   1.1. Semantics .................................... 3
   1.2. Syntax ..................................... 3
   1.3. Notation and terminology ......................... 3
2. Lexical conventions ................................... 5
   2.1. Identifiers .................................. 5
   2.2. Whitespace and comments ....................... 5
   2.3. Other notations ................................ 5
3. Basic concepts ....................................... 6
   3.1. Variables and regions ......................... 6
   3.2. True and false ................................ 6
   3.3. External representations ...................... 6
   3.4. Disjointness of types .......................... 7
   3.5. Storage model ................................ 7
4. Expressions .......................................... 7
   4.1. Primitive expression types ..................... 7
   4.2. Derived expression types ...................... 9
5. Program structure ..................................... 12
   5.1. Programs .................................... 12
   5.2. Definitions ................................ 12
6. Standard procedures .................................. 13
   6.1. Booleans .................................... 13
   6.2. Equivalence predicates ......................... 13
   6.3. Pairs and lists ................................ 15
   6.4. Symbols ...................................... 18
   6.5. Numbers ...................................... 18
   6.6. Characters ................................... 24
   6.7. Strings ...................................... 25
   6.8. Vectors ...................................... 26
   6.9. Control features ................................ 27
   6.10. Input and output ................................ 29
7. Formal syntax and semantics ......................... 32
   7.1. Formal syntax ................................ 32
   7.2. Formal semantics ................................ 34
   7.3. Derived expression types ....................... 36
Notes .................................................... 38
Example .................................................. 39
Appendix: Macros ......................................... 40
Bibliography and references ............................. 47
Alphabetic index of definitions of concepts,
   keywords, and procedures ............................ 52
INTRODUCTION

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

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

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

Acknowledgements

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

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

The Artificial Intelligence Laboratory of the Massachusetts Institute of Technology, the Computer Science Department of Indiana University, and the Computer and Information Sciences Department of the University of Oregon supported the preparation of this report. Support for the MIT work was provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-80-C-0505. Support for the Indiana University work was provided by NSF grants NCS 83-04567 and NCS 83-03325.
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 Haskell, or the call-by-name semantics of Algol 60, where an argument expression is not evaluated unless its value is needed by the procedure.

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

1.2. Syntax

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

The read procedure performs syntactic as well as lexical decomposition of the data it reads. The read procedure parses its input as data (section 7.1.2), not as program.

The formal syntax of Scheme is described in section 7.1.

1.3. Notation and terminology

1.3.1. Essential and non-essential features

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

1.3.2. Error situations and unspecified behavior

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

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

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

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

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

1.3.3. Entry format

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

```
template essential category
```

if the feature is an essential feature, or simply

```
template category
```

if the feature is not an essential feature.

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

```
(thing1) ...
```

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

```
(thing1) ⟨thing2⟩ ...
```

indicates one or more occurrences of a ⟨thing⟩.

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

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

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

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

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

indicate that in all implementations, the make-vector procedure must be defined to take one argument, and some implementations will extend it to take two arguments.

It is an error for an operation to be presented with an argument that it is not specified to handle. For succinctness, we follow the convention that if an argument name is also the name of a type listed in section 3.4, then that argument must be of the named type. For example, the header line for vector-ref given above dictates that the first argument to vector-ref must be a vector. The following naming conventions also imply type restrictions:

```
obj
list, list1, ..., listj, ...
y, y1, ..., yj, ...
q, q1, ..., qj, ...
n, n1, ..., nj, ...
k, k1, ..., kj, ...
```

any object
list (see section 6.3)
complex number
real number
real number
rational number
integer
exact non-negative integer

1.3.4. Evaluation examples

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

```
(* 5 8) ⇒ 40
```

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

1.3.5. Naming conventions

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

By convention, the names of procedures that store values into previously allocated locations (see section 3.3) usually end in “->”. Such procedures are called mutation procedures. By convention, the value returned by a mutation procedure is unspecified.

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

2. Lexical conventions

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

Upper and lower case forms of a letter are never distinguished except within character and string constants. For example, \texttt{Foo} is the same identifier as \texttt{FOO}, and \texttt{#x1AB} is the same number as \texttt{#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 \ldots are identifiers. Here are some examples of identifiers:

\begin{verbatim}
lambda    q
list->vector soup
+          V17a
<=?         a34kTMNs
the-word-recursion-has-many-meanings
\end{verbatim}

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

\begin{verbatim}
+ - . * / < > ! ? : $ % _ & ~
\end{verbatim}

See section 7.1.1 for a formal syntax of identifiers.

Identifiers have several uses within Scheme programs:

\begin{itemize}
\item Certain identifiers are reserved for use as syntactic keywords (see below).
\item Any identifier that is not a syntactic keyword may be used as a variable (see section 3.1).
\item When an identifier appears as a literal or within a literal (see section 4.1.2), it is being used to denote a \textit{symbol} (see section 6.4).
\end{itemize}

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

\begin{verbatim}
=> do or
and else quasiquote
begin if quote
case lambda set!
cond let unquote
define let* unquote-splicing
delay leterr
\end{verbatim}

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

2.2. Whitespace and comments

\textit{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.

\begin{verbatim}
;; The FACT procedure computes the factorial
;; of a non-negative integer.
(define fact
  (lambda (n)
    (if (= n 0)
      1
      (* n (fact (- n 1))))))
\end{verbatim}

2.3. Other notations

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

\begin{itemize}
\item + - These are used in numbers, and may also occur anywhere in an identifier except as the first character. A delimited plus or minus sign by itself is also an identifier. A delimited period (not occurring within a number or identifier) is used in the notation for pairs (section 6.3), and to indicate a rest-parameter in a formal parameter list (section 4.1.4). A delimited sequence of three successive periods is also an identifier.
\item ( ) Parentheses are used for grouping and to noteate lists (section 6.3).
\item ' The single quote character is used to indicate literal data (section 4.1.2).
\item " The backquote character is used to indicate almost-constant data (section 4.2.6).
\item ,, The character comma and the sequence comma at-sign are used in conjunction with backquote (section 4.2.6).
\end{itemize}
The double quote character is used to delimit strings (section 6.7).

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

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

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

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

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

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

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

3. Basic concepts

3.1. Variables and regions

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

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

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

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

3.2. True and false

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

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

3.3. External representations

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

The external representation of an object is not necessarily unique. The integer 28 also has representations “#e28.000” and “#x1c”, and the list in the previous paragraph also has the representations “(( 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 quote, section 4.1.2).

External representations can also be used for input and output. The procedure read (section 6.10.2) parses external representations, and the procedure write (section 6.10.3) generates them. Together, they provide an elegant and powerful input/output facility.

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

3.4. Disjointness of types

No object satisfies more than one of the following predicates:

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

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

3.5. Storage model

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

An object fetched from a location, by a variable reference or by a procedure such as car, vector-ref, or string-ref, is equivalent in the sense of eqv? (section 6.2) to the object last stored in the location before the fetch.

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

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

4. Expressions

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

Expression types are categorized as primitive or derived. Primitive expression types include variables and procedure calls. Derived expression types are not semantically primitive, but can instead be explained in terms of the primitive constructs as in 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

(variable) essential syntax

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

(define x 28)

x → 28

4.1.2. Literal expressions

(quote (datum)) essential syntax

'(datum) essential syntax

(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.

(quote a) → a

(quote #(a b c)) → #(a b c)

(quote (+ 1 2)) → (+ 1 2)

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

'a → a

'(a b c) → #(a b c)

'() → ()

'(1 2) → (+ 1 2)

'(quote a) → (quote a)

'"a" → (quote a)

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

'"abc" → "abc"

"abc" → "abc"

'145932 → 145932

145932 → 145932

'#t → #t

'#t → #t

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

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

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

\[(+ \: 3 \: 4) \quad \Rightarrow \quad 7\]
\[((\text{if} \: \#f \: +) \: 3 \: 4) \quad \Rightarrow \quad 12\]

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

Procedure calls are also called combinations.

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

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

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

4.1.4. Lambda expressions

\[(\text{lambda} \: \text{(formals)} \: \text{(body)})\] essential syntax

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

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

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

\[(\text{define} \: \text{reverse-subtract} \ldots)\]

\[(\text{reverse-subtract} \: (x \: y) \: (- \: y \: x))\]
\[(\text{add4} \quad \Rightarrow \quad 3)\]
\[\text{(define add4} \: (\ldots) \quad \Rightarrow \quad 10)\]

(Formals) should have one of the following forms:

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

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

- \((\text{variable}_1 \: \ldots) \: (\text{variable} \: \ldots)\): If a space-delimited period precedes the last variable, then the value stored in the binding of the last variable will be a newly allocated list of the actual arguments left over after all the other actual arguments have been matched up against the other formal arguments.

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

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

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

4.1.5. Conditionals

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

\[(\text{if} \: (\text{test}) \: (\text{consequent})\] 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.

\[(\text{if} \: (> \: 3 \: 2) \: \text{'yes} \: \text{'no}) \quad \Rightarrow \quad \text{yes}\]
\[(\text{if} \: (> \: 2 \: 3) \: \text{'yes} \: \text{'no}) \quad \Rightarrow \quad \text{no}\]
\[(\text{if} \: (> \: 3 \: 2) \: \text{'yes} \: \text{'no}) \quad \Rightarrow \quad 1\]
\[(\text{if} \: (> \: 3 \: 2) \: \text{'yes} \: \text{'no}) \quad \Rightarrow \quad 1\]
4.1.6. Assignments

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

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

(define x 2)
(+ x 1)          =>  3
(set! 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

  (if (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) 'lesss))  =>  greater

(cond ((> 3 3) 'greater)
      ((< 3 3) 'less))
      (else 'equal))       =>  equal

Some implementations support an alternative (clause) syntax,
((test) => (recipient), where (recipient) is an expres-
sion. 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

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

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

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

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

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

Semantics: A case expression is evaluated as follows.
(Key) is evaluated and its result is compared against each
(datum). If the result of evaluating (key) is equivalent
(in the sense of 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)
  ((w y) 'semivowel)
  (else 'consonant))     =>  consonant

(and (test1) ...)  essential syntax

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

(and (= 2 2) (> 2 1))  =>  #t

(and (= 2 2) (< 2 1))  =>  #f

(and 1 2 'c '(f g))  =>  (f g)

(and)  =>  #t

(or (test1) ...)  essential syntax

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

4.2.2. Binding constructs

The three binding constructs \texttt{let}, \texttt{let*}, and \texttt{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 \texttt{let} expression, the initial values are computed before any of the variables become bound; in a \texttt{let*} expression, the bindings and evaluations are performed sequentially; while in a \texttt{letrec} expression, all the bindings are in effect while their initial values are being computed, thus allowing mutually recursive definitions.

\texttt{(let (bindings) (body))} \hfill \text{essential syntax}

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

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

where each \texttt{(init)} is an expression, and \texttt{(body)} should be a sequence of one or more expressions. It is an error for a \texttt{(variable)} to appear more than once in the list of variables being bound.

\textit{Semantics}: The \texttt{(init)}s are evaluated in the current environment (in some unspecified order), the \texttt{(variable)}s are bound to fresh locations holding the results, the \texttt{(body)} is evaluated in the extended environment, and the value of the last expression of \texttt{(body)} is returned. Each binding of a \texttt{(variable)} has \texttt{(body)} as its region.

\begin{align*}
(\text{let } ((x \ 2) \ (y \ 3)) \ (\ast x \ y)) & \implies 6 \\
(\text{let } ((x \ 2) \ (y \ 3)) \ (\text{let } ((x \ 7) \ (z \ (+ x \ y)) \ (\ast z \ x))) & \implies 35
\end{align*}

See also named \texttt{let}, section 4.2.4.

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

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

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

and \texttt{(body)} should be a sequence of one or more expressions.

\textit{Semantics}: \texttt{let*} is similar to \texttt{let}, but the bindings are performed sequentially from left to right, and the region of a binding indicated by \texttt{(variable) (init)} is that part of the \texttt{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.

\begin{align*}
(\text{let* } ((x \ 2) \ (y \ 3)) \ (\text{let* } ((x \ 7) \ (z \ (+ x \ y)) \ (* z \ x))) & \implies 70 \\
(\text{letrec } \text{(bindings)} \ (\text{body})) & \text{ essential syntax}
\end{align*}

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

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

and \texttt{(body)} should be a sequence of one or more expressions. It is an error for a \texttt{(variable)} to appear more than once in the list of variables being bound.

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

\begin{align*}
(\text{letrec } (\text{even?}) \ (\lambda n \ (\text{if } \text{(zero? \ n)} \ \#t \ \text{(odd? } (- n 1)))) \\
\ (\text{odd?}) \ (\lambda n \ (\text{if } \text{(zero? \ n)} \ \#f \ \text{(even? } (- n 1)))) \\
\ (\text{even? 88}) & \implies \#t
\end{align*}

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

4.2.3. Sequencing

\texttt{(begin (expression}_1) \ (expression}_2) \ldots) \text{ essential syntax}

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

\begin{align*}
(\text{define } x \ 0) \\
(\text{begin } (\text{set! x 5}) \ (+ x \ 1)) & \implies 6
\end{align*}
4. Expressions

4.2.4. Iteration

\[
\begin{align*}
\text{(do (\{variable\}_1) (\{init\}_1) (\{step\}_1))} & \quad \text{\textit{syntax}} \\
& \ldots \\
& (\{test\} (\{expression\}) \ldots) \\
& (\{command\} \ldots)
\end{align*}
\]

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\}_1s are bound to fresh locations, the results of the \{init\}_1 expressions are stored in the bindings of the \{variable\}_1s, 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\}_1 expressions are evaluated in some unspecified order, the \{variable\}_1s are bound to fresh locations, the results of the \{step\}_1 expressions are stored in the bindings of the \{variable\}_1s, and the next iteration begins.

If \{test\} evaluates to a true value, then the \{expression\}_1 expressions are evaluated from left to right and the value of the last \{expression\}_1 expression is returned as the value of the do expression. If no \{expression\}_1 is 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\}_1. It is an error for a \{variable\} to appear more than once in the list of do variables.

A \{step\}_1 may be omitted, in which case the effect is the same as if (\{variable\} (\{init\} (\{variable\})) had been written instead of ((\{variable\} (\{init\})).

\[
\begin{align*}
\text{(do ((vec (make-vector 5))} & \quad \text{\textit{syntax}} \\
& \quad (i 0 (+ i 1)) \\
& \quad (\text{vector-set! vec i i})) \Rightarrow \#(0 1 2 3 4) \\
\text{(let ((x '(1 3 5 7 9)))} & \quad \text{\textit{syntax}} \\
& \quad (do ((x x (cdr x))} \\
& \quad \quad (\text{sum 0 (+ sum (car x))}) \\
& \quad \quad (\text{null? x sum})) \Rightarrow 25
\end{align*}
\]

Some implementations of Scheme permit a variant on the syntax of do called "named do" 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\}.

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

4.2.5. Delayed evaluation

\[
\begin{align*}
\text{(delay (expression))} & \quad \text{\textit{syntax}} \\
& \quad \text{The \textit{delay} construct is used together with the procedure} \\
& \quad \text{\textit{force} to implement lazy evaluation or call by need. (delay} \\
& \quad \text{(expression)) returns an object called a} \textit{promise} \text{which at} \\
& \quad \text{some point in the future may be asked (by the \textit{force} procedure) to evaluate (expression) and deliver the resulting} \\
& \quad \text{value.} \\
& \quad \text{See the description of \textit{force} (section 6.9) for a more} \\
& \quad \text{complete description of \textit{delay}.}
\end{align*}
\]

4.2.6. Quasiquotation

\[
\begin{align*}
\text{(quasiquote (template))} & \quad \text{\textit{essential syntax}} \\
& \quad \text{\textit{template}} \\
& \quad \text{\textit{template}}\end{align*}
\]

"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 immediately after 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.
5.2. Definitions

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

A definition should have one of the following forms:

- (define (variable) (expression))
  This syntax is essential.

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

- (define ((variable) . (formal)) (body))
  This syntax is not essential. (Formal) should be a single variable. This form is equivalent to
  
  (define (variable)
    (lambda (formal) (body))).

- (begin (definition1) ...)
  This syntax is essential. This form is equivalent to the set of definitions that form the body of the begin.

5.2.1. Top level definitions

At the top level of a program, a definition

  (define (variable) (expression))

has essentially the same effect as the assignment expression

  (set! (variable) (expression))

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

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

All Scheme implementations must support top level definitions.

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

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

\[
\begin{align*}
&\text{(let (x 5))} \\
&(\text{define foo (lambda (y) (bar x y)))} \\
&(\text{define bar (lambda (a b) (+ (* a b) a)))} \\
&(\text{foo (+ x 3))}) \\
&\Rightarrow 45
\end{align*}
\]

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

\[
\begin{align*}
&\text{(let (x 5))} \\
&(\text{letrec ((foo (lambda (y) (bar x y)))} \\
&(\text{bar (lambda (a b) (+ (* a b) a)))} \\
&(\text{foo (+ x 3))}) \\
&\Rightarrow 45
\end{align*}
\]

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

6. Standard procedures

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

6.1. Booleans

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

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

\[
\begin{align*}
\text{#t} &\Rightarrow \text{#t} \\
\text{#f} &\Rightarrow \text{#f} \\
\text{'#f} &\Rightarrow \text{#f}
\end{align*}
\]

\[
\begin{align*}
\text{not obj} &\quad \text{essential procedure} \\
&\Rightarrow \text{not #t if obj is false, and returns #f otherwise.}
\end{align*}
\]

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

\[
\begin{align*}
\text{eqv? obj1 obj2} &\quad \text{essential procedure} \\
&\Rightarrow \text{the eqv? procedure defines a useful equivalence relation on objects. Briefly, it returns #t if obj1 and obj2 should normally be regarded as the same object. This relation is left slightly open to interpretation, but the following partial specification of eqv? holds for all implementations of Scheme. The eqv? procedure returns #t if:}
&\begin{itemize}
&\text{bullet obj1 and obj2 are both #t or both #f.}
&\text{bullet obj1 and obj2 are both symbols and
\end{itemize}
\end{align*}
\]
Revised Scheme

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

\[(\text{symbol->string obj2)})\]  \(\Rightarrow\)  \#t

\text{Note:}  \text{This assumes that neither } obj_1 \text{ nor } obj_2 \text{is an "uninterned symbol" as alluded to in section 6.4. This report does not presume to specify the behavior of eqv? on implementation-dependent extensions.}

\bullet  obj_1 \text{ and } obj_2 \text{ are both numbers, are numerically equal (see \#, section 6.5), and are either both exact or both inexact.}

\bullet  obj_1 \text{ and } obj_2 \text{ are both characters and are the same character according to the char=? procedure (section 6.6).}

\bullet  both } obj_1 \text{ and } obj_2 \text{ are the empty list.}

\bullet  obj_1 \text{ and } obj_2 \text{ are pairs, vectors, or strings that denote the same locations in the store (section 3.5).}

\bullet  obj_1 \text{ and } obj_2 \text{ are procedures whose location tags are equal (section 4.1.4).}

\text{The eqv? procedure returns \#f if:}

\bullet  obj_1 \text{ and } obj_2 \text{ are of different types (section 3.4).}

\bullet  one of } obj_1 \text{ and } obj_2 \text{ is \#t but the other is \#f.}

\bullet  obj_1 \text{ and } obj_2 \text{ are symbols but}

\[\text{(string=? (symbol->string obj1)})\]  \(\Rightarrow\)  \#f

\bullet  one of } obj_1 \text{ and } obj_2 \text{ is an exact number but the other is an inexact number.}

\bullet  obj_1 \text{ and } obj_2 \text{ are numbers for which the = procedure returns \#f.}

\bullet  obj_1 \text{ and } obj_2 \text{ are characters for which the char=? procedure returns \#f.}

\bullet  one of } obj_1 \text{ and } obj_2 \text{ is the empty list but the other is not.}

\bullet  obj_1 \text{ and } obj_2 \text{ are pairs, vectors, or strings that denote distinct locations.}

\bullet  obj_1 \text{ and } obj_2 \text{ are procedures that would behave differently (return a different value or have different side effects) for some arguments.}

\[\text{eqv? 'a 'a)}\]  \(\Rightarrow\)  \#t

\[\text{eqv? 'a 'b)}\]  \(\Rightarrow\)  \#f

\[\text{eqv? 2 2)}\]  \(\Rightarrow\)  \#t

\[\text{eqv? '() '()}}\]  \(\Rightarrow\)  \#t

\[\text{eqv? 100000000 1000000000)}\]  \(\Rightarrow\)  \#t

\[\text{eqv? (cons 1 2) (cons 1 2))}\]  \(\Rightarrow\)  \#f

\[\text{eqv? (lambda () 1)}\]  \(\Rightarrow\)  \#f

\[\text{(lambda () 2))}\]  \(\Rightarrow\)  \#f

\[\text{(let ((p (lambda (x) x))) (eqv? p p))}\]  \(\Rightarrow\)  \#t

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

\[\text{(eqv? "" "")}\]  \(\Rightarrow\)  \text{unspecified}

\[\text{(eqv? '1() '1())}\]  \(\Rightarrow\)  \text{unspecified}

\[\text{(eqv? (lambda (x) x) (lambda (x) x))}\]  \(\Rightarrow\)  \text{unspecified}

\[\text{(eqv? (lambda (x) x) (lambda (y) y))}\]  \(\Rightarrow\)  \text{unspecified}

\text{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.}

\[\text{(define gen-counter)}\]
\[\text{(lambda () (let ((n 0)) (lambda () (set! n (+ n 1)) n)))}\]
\[\text{(let ((g (gen-counter))) (eqv? g g))}\]  \(\Rightarrow\)  \#t

\[\text{(eqv? (gen-counter) (gen-counter))}\]  \(\Rightarrow\)  \#f

\text{(define gen-loser)}
\[\text{(lambda () (let ((n 0)) (lambda () (set! n (+ n 1)) 27)))}\]
\[\text{(let ((g (gen-loser))) (eqv? g g))}\]  \(\Rightarrow\)  \#t

\[\text{(eqv? (gen-loser) (gen-loser))}\]  \(\Rightarrow\)  \text{unspecified}

\[\text{(letrec ((f (lambda () (if (eqv? f g) 'both 'f)) (g (lambda () (if (eqv? f g) 'both 'g)) (eqv? f g)))}\]  \(\Rightarrow\)  \text{unspecified}

\[\text{(letrec ((f (lambda () (if (eqv? f g) 'f 'both)) (g (lambda () (if (eqv? f g) 'g 'both)) (eqv? f g)))}\]  \(\Rightarrow\)  \#f

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

\[\text{(eqv? '(a) '(a))}\]  \(\Rightarrow\)  \text{unspecified}

\[\text{(eqv? "a" "a")}\]  \(\Rightarrow\)  \text{unspecified}

\[\text{(eqv? '(b) (cdr '(a b)))}\]  \(\Rightarrow\)  \text{unspecified}

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

\[(\text{eq? } \text{obj}_1 \text{ obj}_2)\]

essential procedure

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

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

\[(\text{eq? } \text{'}a\text{ '}\text{a})\]
\[\Rightarrow \#t\]

\[(\text{eq? } \text{'}a\text{ '}\text{b})\]
\[\Rightarrow \text{ unspecified}\]

\[(\text{eq? } \text{'}a\text{ '}\text{b})\]
\[\Rightarrow \#f\]

\[(\text{eq? } \text{'}a\text{ '}\text{'a})\]
\[\Rightarrow \text{ unspecified}\]

\[(\text{eq? } \text{'}a\text{ '}\text{c})\]
\[\Rightarrow \text{ unspecified}\]

\[(\text{eq? } \text{'}(\text{a} \text{'}(\text{b} \text{'})\text{a})\text{'})\]
\[\Rightarrow \#t\]

\[(\text{eq? } \text{'}(\text{a} \text{'}\text{b})\text{c})\]
\[\Rightarrow \text{ unspecified}\]

\[(\text{eq? } \text{'}a\text{ '}\text{b})\]
\[\Rightarrow \text{ unspecified}\]

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 cons. The car and cdr fields are accessed by the procedures car and cdr. The car and cdr fields are assigned by the procedures set-car! and set-cdr!.

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

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

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

The empty list is a special object of its own type (it is not a pair); it has no elements and its length is zero.

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

The most general notation (external representation) for Scheme pairs is the “dotted notation (c₁ . c₂) where c₁ is the value of the car field and c₂ is the value of the cdr field. For example (4 . 5) is a pair whose car is 4 and whose cdr is 5. Note that (4 . 5) is the external representation of a pair, not an expression that evaluates to a pair.

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

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

and

\[(a \ (b \ (c \ (d \ (e \ ()))))\)
are equivalent notations for a list of symbols.

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

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

is equivalent to

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

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

```
(define x (list 'a 'b 'c))
(define y x)
(define z (list 'a 'b 'c))
(y (list? y)) => #t
(set-cdr! x 4) => unspecified
x => (a . 4)
(eqv? x y) => #t
set-cdr! x x => unspecified
(set-cdr! x x) => unspecified
(list? x) => #f
```

Within literal expressions and representations of objects read by the `read` procedure, the forms `(datum), `'(datum), ,`(datum), and ,0`(datum)` denote two-element lists whose first elements are the symbols `quote`, `quasiquote`, `unquote`, and `unquote-splicing`, respectively. The second element in each case is `(datum)`. This convention is supported so that arbitrary Scheme programs may be represented as lists. That is, according to Scheme’s grammar, every expression is also a `(datum)` (see section 7.1.2). Among other things, this permits the use of the `read` procedure to parse Scheme programs. See section 3.3.

### (pair? obj) essential procedure

Pair? returns `#t` if `obj` is a pair, and otherwise returns `#f`.

```
(pair? 'a . b)) => #t
(pair? '(a b c)) => #t
(pair? **) => #f
(pair? '#(a b)) => #f
```

### (cons obj1 obj2) essential procedure

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

```
(cons 'a ') => (a)
(cons '(a) '((b c d)) => ((a) b c d)
(cons "a" '(b c)) => ("a" b c)
(cons 'a 3) => (a . 3)
(cons '(a b) 'c) => ((a b) . c)
```

### (car pair) essential procedure

Returns the contents of the car field of `pair`. Note that it is an error to take the car of the empty list.

```
(car '(a b c)) => a
(car '((a) b c d)) => (a)
(car '(1 . 2)) => 1
(car '()) => error
```

### (cdr pair) essential procedure

Returns the contents of the cdr field of `pair`. Note that it is an error to take the cdr of the empty list.

```
(cdr '((a) b c d)) => (b c d)
(cdr '(1 . 2)) => 2
(cdr '()) => error
```

### (set-car! pair obj) essential procedure

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

```
(define (f) (list 'not-a-constant-list))
(define (g) '(constant-list))
(set-car! (f) 3) => unspecified
(set-car! (g) 3) => error
```

### (set-cdr! pair obj) essential procedure

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

```
(caar pair) essential procedure
(cadr pair) essential procedure

(cdddar pair) essential procedure
(cddddr pair) essential procedure
```

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

```
(define caddr (lambda (x) (car (cdr (cdr x))))).
```

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

### (null? obj) essential procedure

Returns `#t` if `obj` is the empty list, otherwise returns `#f`.

```
(null? obj) ➞ #f
```

### (list? obj) essential procedure

Returns `#t` if `obj` is a list, otherwise returns `#f`. By definition, all lists have finite length and are terminated by the empty list.

```
(list? obj) ➞ #f
```
6. Standard procedures

(list? '(a b c))  \implies \#t
(list? '())  \implies \#t
(list? '(a . b))  \implies \#f
(let ((x (list 'a)))
  (set-cdr! x x)
  (list? x))  \implies \#f

(list obj ...)

Returns a newly allocated list of its arguments.

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

(length list)

Returns the length of list.

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

(append list ...)

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

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

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

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

(reverse list)

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

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

(list-tail list k)

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

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

(list-ref list k)  \hfill \text{essential procedure}

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

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

(memq obj list)  \hfill \text{essential procedure}
(memv obj list)  \hfill \text{essential procedure}
(member obj list)  \hfill \text{essential procedure}

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

(memq 'a '(a b c))  \implies (a b c)
(memq 'b '(a b c))  \implies (b c)
(memq 'a '(b c d))  \implies \#f
(memq (list 'a) '(b (a) c))  \implies \#f
(member (list 'a)
  '(b (a) c))  \implies ((a) c)
(memq 101 '(100 101 102))  \implies \text{unspecified}
(memv 101 '(100 101 102))  \implies (101 102)

(assq obj alist)  \hfill \text{essential procedure}
(assv obj alist)  \hfill \text{essential procedure}
(assoc obj alist)  \hfill \text{essential procedure}

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

(define e '(((a 1) (b 2) (c 3)))
  (assq 'a e)  \implies (a 1)
  (assq 'b e)  \implies (b 2)
  (assq 'd e)  \implies \#f
  (assq (list 'a) '(((a)) ((b)) ((c)))))  \implies \#f
  (assoc (list 'a) '(((a)) ((b)) ((c))))  \implies ((a))
  (assq 5 '((2 3) (5 7) (11 13)))  \implies \text{unspecified}
  (assv 5 '((2 3) (5 7) (11 13)))  \implies (5 7)

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

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

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

It is guaranteed that any symbol that has been returned as part of a literal expression, or read using the 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.

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

Some implementations also have "untinterned 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.

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

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

\[
\begin{align*}
\text{(symbol? 'foo)} & \quad \Rightarrow \ #t \\
\text{(symbol? (car '(a b)))} & \quad \Rightarrow \ #t \\
\text{(symbol? "bar")} & \quad \Rightarrow \ #f \\
\text{(symbol? 'nil)} & \quad \Rightarrow \ #t \\
\text{(symbol? '()}) & \quad \Rightarrow \ #f \\
\text{(symbol? #f)} & \quad \Rightarrow \ #f \\
\end{align*}
\]

\[
\text{(symbol->string symbol)} \quad \text{essential procedure}
\]

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

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

\[
\begin{align*}
\text{(symbol->string 'flying-fish)} & \quad \Rightarrow \ "flying-fish" \\
\text{(symbol->string 'Martin)} & \quad \Rightarrow \ "martin" \\
\text{(symbol->string (string->symbol "Malvina"))} & \quad \Rightarrow \ "Malvina"
\end{align*}
\]

\[
\text{(string->symbol string)} \quad \text{essential procedure}
\]

Returns the symbol whose name is \textit{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:

\[
\begin{align*}
\text{(eq? 'mISSISSippi 'mississippi)} & \quad \Rightarrow \ #t \\
\text{(string->symbol "mISSISSippi")} & \quad \Rightarrow \ \text{the symbol with name "mISSISSippi"} \\
\text{(eq? 'bitBlt (string->symbol "bitBlt")}) & \quad \Rightarrow \ #f \\
\text{(eq? 'JollyWog (string->symbol (symbol->string 'JollyWog)))} & \quad \Rightarrow \ #t \\
\text{(string=? "K. Harper, M.D." (symbol->string (symbol->symbol "K. Harper, M.D."))) & \quad \Rightarrow \ #t}
\end{align*}
\]

6.5. Numbers

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

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

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

- number
- complex
- real
- rational
- integer

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

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

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

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

6.5.2. Exactness

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

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

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

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

6.5.3. Implementation restrictions

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

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

An implementation of Scheme must support exact integers throughout the range of numbers that may be used for indexes of lists, vectors, and strings or that may result from computing the length of a list, vector, or string. The length, vector-length, and string-length procedures must return an exact integer, and it is an error to use anything but an exact integer as an index. Furthermore any integer constant within the index range, if expressed by an exact integer syntax, will indeed be read as an exact integer, regardless of any implementation restrictions that may apply outside this range. Finally, the procedures listed below will always return an exact integer result provided all their arguments are exact integers and the mathematically expected result is representable as an exact integer within the implementation:
Implementations are encouraged, but not required, to support exact integers and exact rationals of practically unlimited size and precision, and to implement the above procedures and the \(/\) procedure in such a way that they always return exact results when given exact arguments. If one of these procedures is unable to deliver an exact result when given exact arguments, then it may either report a violation of an implementation restriction or it may silently coerce its result to an inexact number. Such a coercion may cause an error later.

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

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

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

### 6.5.4 Syntax of numerical constants

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

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

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

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

\[
3.1415926535897990
\]

\[
0.610
\]

\[
\text{Round to single} \rightarrow 3.141593
\]

\[
\text{Extend to long} \rightarrow .6000000000000000
\]

### 6.5.5 Numerical operations

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

\[
\text{(number? obj)}
\]

\[
\text{(complex? obj)}
\]

\[
\text{(real? obj)}
\]

\[
\text{(rational? obj)}
\]

\[
\text{(integer? obj)}
\]

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

If \( z \) is an inexact complex number, then \((\text{real? } z)\) is true if and only if \((\text{zero? } (\text{imag-part } z))\) is true. If \( x \) is an inexact real number, then \((\text{integer? } x)\) is true if and only if \((= x \text{ (round } x))\).

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

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

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

\[
\begin{align*}
(\text{exact? } z) & \implies \text{essential procedure} \\
(\text{inexact? } z) & \implies \text{essential procedure}
\end{align*}
\]

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

\[
\begin{align*}
(= z_1 z_2 z_3 \ldots) & \implies \text{essential procedure} \\
(< z_1 z_2 z_3 \ldots) & \implies \text{essential procedure} \\
(> z_1 z_2 z_3 \ldots) & \implies \text{essential procedure} \\
(\leq z_1 z_2 z_3 \ldots) & \implies \text{essential procedure} \\
(\geq z_1 z_2 z_3 \ldots) & \implies \text{essential procedure}
\end{align*}
\]

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

These predicates are required to be transitive.

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

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

\[
\begin{align*}
(\text{zero? } z) & \implies \text{essential procedure} \\
(\text{positive? } z) & \implies \text{essential procedure} \\
(\text{negative? } z) & \implies \text{essential procedure}
\end{align*}
\]

\[
\begin{align*}
(\text{odd? } n) & \implies \text{essential procedure} \\
(\text{even? } n) & \implies \text{essential procedure}
\end{align*}
\]

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

\[
\begin{align*}
(\text{max } x_1 x_2 \ldots) & \implies \text{essential procedure} \\
(\text{min } x_1 x_2 \ldots) & \implies \text{essential procedure}
\end{align*}
\]

These procedures return the maximum or minimum of their arguments.

\[
\begin{align*}
(\text{max } 3 4) & \implies 4 \text{ ; exact} \\
(\text{max } 3.9 4) & \implies 4.0 \text{ ; inexact}
\end{align*}
\]

**Note:** If any argument is inexact, then the result will also be inexact (unless the procedure can prove that the inaccuracy is not large enough to affect the result, which is possible only in unusual implementations). If \text{min} or \text{max} is used to compare numbers of mixed exactness, and the numerical value of the result cannot be represented as an inexact number without loss of accuracy, then the procedure may report a violation of an implementation restriction.

\[
\begin{align*}
(\text{+ } z_1 \ldots) & \implies \text{essential procedure} \\
(\text{* } z_1 \ldots) & \implies \text{essential procedure}
\end{align*}
\]

These procedures return the sum or product of their arguments.

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

\[
\begin{align*}
(- z_1 z_2) & \implies \text{essential procedure} \\
(- z) & \implies \text{essential procedure} \\
(- z_1 z_2 \ldots) & \implies \text{essential procedure} \\
(/ z_1 z_2) & \implies \text{essential procedure} \\
(/ z) & \implies \text{essential procedure} \\
(/ z_1 z_2 \ldots) & \implies \text{essential procedure}
\end{align*}
\]

With two or more arguments, these procedures return the difference or quotient of their arguments, associating to the left. With one argument, however, they return the additive or multiplicative inverse of their argument.

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

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

Abs returns the magnitude of its argument.

\[
(\text{abs } -7) \implies 7
\]
(quotient n₁ n₂) essential procedure
(remainder n₁ n₂) essential procedure
(modulo n₁ n₂) essential procedure

These procedures implement number-theoretic (integer) division: For positive integers n₁ and n₂, if n₃ and n₄ are integers such that n₁ = n₂n₃ + n₄ and 0 ≤ n₄ < n₂, then

(quotient n₁ n₂) ⊢ n₃
(remainder n₁ n₂) ⊢ n₄
(modulo n₁ n₂) ⊢ n₄

For integers n₁ and n₂ with n₂ not equal to 0,

(\(=\) n₁ (\(+\) (\(*\) n₂ (quotient n₁ n₂)))
  (remainder n₁ n₂))) ⊢ #t

provided all numbers involved in that computation are exact.

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

(modulo 13 4) ⊢ 1
(modulo -13 4) ⊢ 3
(modulo 13 -4) ⊢ -3
(modulo -13 -4) ⊢ 1
(modulo -13 -4.0) ⊢ -1.0 ; inexact

(gcd n₁ ...) essential procedure
(lcm n₁ ...) essential procedure

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

(gcd 32 -36) ⊢ 4
(gcd) ⊢ 0
(lcm 32 -36) ⊢ 288
(lcm 32.0 -36) ⊢ 288.0 ; inexact
(lcm) ⊢ 1

(numerator q) procedure
(denominator q) procedure

These procedures return the numerator or denominator of their argument; the result is computed as if the argument was represented as a fraction in lowest terms. The denominator is always positive. The denominator of 0 is defined to be 1.

(numerator (/ 6 4)) ⊢ 3
(denominator (/ 6 4)) ⊢ 2
(denominator
  (exact->inexact (/ 6 4))) ⊢ 2.0

(floor x) essential procedure
(ceiling x) essential procedure
(truncate x) essential procedure
(round x) essential procedure

These procedures return integers. Floor returns the largest integer not larger than x. Ceiling returns the smallest integer not smaller than x. Truncate returns the integer closest to x whose value is 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.

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

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

(floor -4.3) ⊢ -5.0
(ceiling -4.3) ⊢ -4.0
(truncate -4.3) ⊢ -4.0
(round -4.3) ⊢ -4.0

(floor 3.5) ⊢ 3.0
(ceiling 3.5) ⊢ 4.0
(truncate 3.5) ⊢ 3.0
(round 3.5) ⊢ 4.0 ; inexact

(round 7/2) ⊢ 4 ; exact
(round 7) ⊢ 7

(rationalize x y) procedure

Rationalize returns the simplest rational number differing from x by no more than y. A rational number r₁ is simpler than another rational number r₂ if r₁ = p₁/q₁ and r₂ = p₂/q₂ (in lowest terms) and \(|p₁| ≤ |p₂|\) and \(|q₁| ≤ |q₂|\). Thus 3/5 is simpler than 4/7. Although not all rationals are comparable in this ordering (consider 2/7 and 3/5) any interval contains a rational number that is simpler than every other rational number in that interval (the simpler 2/5 lies between 2/7 and 3/5). Note that 0 = 0/1 is the simplest rational of all.

(rationalize
  (inexact->exact .3) 1/10) ⊢ 1/3 ; exact
(rationalize .3 1/10) ⊢ #i1/3 ; inexact
6. Standard procedures

(exp z) procedure
(log z) procedure
(sin z) procedure
(cos z) procedure
(tan z) procedure
(asin z) procedure
(acos z) procedure
(atan z) procedure
(atan y x) procedure

These procedures are part of every implementation that supports general real numbers; they compute the usual transcendental functions. Log computes the natural logarithm of z (not the base ten 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 general 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 −π (exclusive) to π (inclusive). log 0 is undefined. The value of log z when z is complex is defined according to the formula

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

With log defined this way, the values of sin⁻¹ z, cos⁻¹ z, and tan⁻¹ z are according to the following formulas:

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

The above specification follows [83], which in turn cites [60]; refer to these sources for more detailed discussion of branch cuts, boundary conditions, and implementation of these functions. When it is possible these procedures produce a real result from a real argument.

(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.

(expt z1 z2) procedure

Returns z1 raised to the power z2:

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

0⁰ is defined to be equal to 1.

(make-rectangular x1 x2) procedure

(make-polar x3 x4) procedure

(real-part z) procedure

(imag-part z) procedure

(magnitude z) procedure

(angle z) procedure

These procedures are part of every implementation that supports general complex numbers. Suppose x1, x2, x3, and x4 are real numbers and z is a complex number such that

\[ z = x_1 + x_2 i = x_3 e^{i x_4} \]

Then make-rectangular and make-polar return z, real-part returns x₁, imag-part returns x₂, magnitude returns x₃, and angle returns x₄. 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 −π (exclusive) to π (inclusive).

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

(exact->inexact z) procedure

(inexact->exact z) procedure

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

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

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

6.5.6. Numerical input and output

(number->string number) essential procedure

(number->string number radix) essential procedure

Radix must be an exact integer, either 2, 8, 10, or 16. If omitted, radix defaults to 10. The procedure number->string takes a number and a radix and returns as a string an external representation of the given number in the given radix such that

(let ((number number) (radix radix)) (eqv? number (string->number (number->string number radix))))
is true. It is an error if no possible result makes this expression true.

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

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

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

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

\[
\begin{align*}
\text{string->number string} & \quad \text{essential procedure} \\
\text{string->number string radix} & \quad \text{essential procedure}
\end{align*}
\]

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

\[
\begin{align*}
\text{(string->number "100")} & \quad \Rightarrow 100 \\
\text{(string->number "100" 16)} & \quad \Rightarrow 256 \\
\text{(string->number "1e2")} & \quad \Rightarrow 100.0 \\
\text{(string->number "15##")} & \quad \Rightarrow 1500.0
\end{align*}
\]

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

6.6. Characters

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

\[
\begin{align*}
\text{\#a} & \quad ; \text{lower case letter} \\
\text{\#A} & \quad ; \text{upper case letter} \\
\text{\#\(} & \quad ; \text{left parenthesis} \\
\text{\#\}} & \quad ; \text{the space character} \\
\text{\#space} & \quad ; \text{the preferred way to write a space} \\
\text{\#newline} & \quad ; \text{the newline character}
\end{align*}
\]

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

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

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

\[
\begin{align*}
\text{(char? obj)} & \quad \text{essential procedure}
\end{align*}
\]

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

\[
\begin{align*}
\text{(char=? char1 char2)} & \quad \text{essential procedure} \\
\text{(char< char1 char2)} & \quad \text{essential procedure} \\
\text{(char> char1 char2)} & \quad \text{essential procedure} \\
\text{(char<=? char1 char2)} & \quad \text{essential procedure} \\
\text{(char=> char1 char2)} & \quad \text{essential procedure}
\end{align*}
\]

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

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

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

```
(char-ci=? char₁ char₂) essential procedure
(char-ci<=? char₁ char₂) essential procedure
(char-ci>? char₁ char₂) essential procedure
(char-ci=? char₂ char₁) essential procedure
```

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

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

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

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

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

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

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

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

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

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-lowercase is lower case.

6.7. Strings

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

"The word \"recursion\" has many meanings."

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

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

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

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

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

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

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

```
Make-string returns a newly allocated string of length k.
```

If char is given, then all elements of the string are initialized to char, otherwise the contents of the string are unspecified.

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

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

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

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

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

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

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

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

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

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

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

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

6.8. Vectors

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

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

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

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

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

'(#(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 unspec-
ified.
6. Standard procedures

6.9. Control features

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

(procedure? obj)  essential procedure

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

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

(apply proc args)  essential procedure

(apply proc args1 ... 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 args1 ...) args) as the actual arguments.

(apply + (list 3 4))  => 7

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

((compose sqrt *) 12 75)  => 30

(map proc list1 list2 ...)  essential procedure

The lists must be lists, and proc must be a procedure taking as many arguments as there are lists. If more than one list is given, then they must all be the same length. Map applies proc element-wise to the elements of the lists and returns a list of the results, in order from left to right. The dynamic order in which proc is applied to the elements of the lists is unspecified.

(map cadr '((a b) (d e) (g h)))  => (b e h)

(map (lambda (n) (expt n n))
  '(1 2 3 4 5))  => (1 4 27 256 3125)

(map + '(1 2 3) '(4 5 6))  => (5 7 9)

(let ((count 0))
  (map (lambda (ignored)
        (set! count (+ count 1))
        count)
       '(a b c)))  => unspecified
(for-each proc list1 list2 ...)    essential procedure
The arguments to for-each are like the arguments to map, but for-each calls proc for its side effects rather than for its values. Unlike map, for-each is guaranteed to call proc on the elements of the lists in order from the first element to the last, and the value returned by for-each is unspecified.

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

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

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

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

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

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

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

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

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

We define the expression

(delay (expression))

to have the same meaning as the procedure call

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

where make-promise is defined as follows:

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

Rationale: A promise may refer to its own value, as in the last example above. Forcing such a promise may cause the promise to be forced a second time before the value of the first force has been computed. This complicates the definition of make-promise. Various extensions to this semantics of delay and force are supported in some implementations:

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

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

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

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

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

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

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

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

```
(call-with-current-continuation
  (lambda (exit)
    (for-each (lambda (x)
       (if (negative? x)
          (exit x)))
          '(54 0 37 -3 245 19))
    #t))  \[\rightarrow\] -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))  \[\rightarrow\] 4
(list-length '(a b . c))  \[\rightarrow\] #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 contin-
uation 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 contin-
uation might take the result, print it on the screen, prompt
for the next input, evaluate it, and so on forever. Most of the
time the continuation includes actions specified by user code,
as in a continuation that will take the result, multiply it by
the value stored in a local variable, add seven, and give the
answer to the top level continuation to be printed. Normally
these ubiquitous continuations are hidden behind the scenes and
programmers don’t think much about them. On rare occasions,
however, a programmer may need to deal with continuations ex-
plicitly. Call-with-current-continuation allows Scheme pro-
grammers to do that by creating a procedure that acts just like
the current continuation.

Most programming languages incorporate one or more special-
purpose escape constructs with names like exit, return, or
even goto. In 1965, however, Peter Landin [55] invented a
general purpose escape operator called the J-operator. John
Reynolds [69] described a simpler but equally powerful con-
struct 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 in-
stead.

### 6.10. Input and output

#### 6.10.1. Ports

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

```
(call-with-input-file string proc)
```

**essential procedure**

```
(call-with-output-file string proc)
```

**essential procedure**

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

**Rationale:** Because Scheme's escape procedures have un-
limited extent, it is possible to escape from the current con-
tinuation 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 respec-
tively, otherwise returns #f.
(current-input-port) essential procedure
(current-output-port) essential procedure

Returns the current default input or output port.

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

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

(open-input-file filename) essential procedure

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

(open-output-file filename) essential procedure

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

(close-input-port port) essential procedure
(close-output-port port) essential procedure

Closes the file associated with *port*, rendering the *port* incapable of delivering or accepting characters. These routines have no effect if the file has already been closed. The value returned is unspecified.

6.10.2. Input

(read) essential procedure
(read port) essential procedure

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

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

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

(read-char) essential procedure
(read-char port) essential procedure

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

(peek-char) essential procedure
(peek-char port) essential procedure

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

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

(eof-object? obj) essential procedure

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

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

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

Rationale: Char-ready? exists to make it possible for a program to accept characters from interactive ports without getting
6. Standard procedures

6.10.3. Output

\texttt{(write obj)} \hspace{1cm} \textit{essential procedure}
\texttt{(write obj port)} \hspace{1cm} \textit{essential procedure}

Writes a written representation of \textit{obj} to the given \textit{port}. Strings that appear in the written representation are enclosed in doublequotes, and within those strings backslash and doublequote characters are escaped by backslashes. \texttt{Write} returns an unspecified value. The \textit{port} argument may be omitted, in which case it defaults to the value returned by \texttt{current-output-port}.

\texttt{(display obj)} \hspace{1cm} \textit{essential procedure}
\texttt{(display obj port)} \hspace{1cm} \textit{essential procedure}

Writes a representation of \textit{obj} to the given \textit{port}. Strings that appear in the written representation are not enclosed in doublequotes, and no characters are escaped within those strings. Character objects appear in the representation as if written by \texttt{write-char} instead of by \texttt{write}. \texttt{Display} returns an unspecified value. The \textit{port} argument may be omitted, in which case it defaults to the value returned by \texttt{current-output-port}.

\textit{Rationale:} \texttt{Write} is intended for producing machine-readable output and \texttt{display} is for producing human-readable output. Implementations that allow “slashification” within symbols will probably want \texttt{write} but not \texttt{display} to slashify funny characters in symbols.

\texttt{(newline)} \hspace{1cm} \textit{essential procedure}
\texttt{(newline port)} \hspace{1cm} \textit{essential procedure}

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

\texttt{(write-char char)} \hspace{1cm} \textit{essential procedure}
\texttt{(write-char char port)} \hspace{1cm} \textit{essential procedure}

Writes the character \textit{char} (not an external representation of the character) to the given \textit{port} and returns an unspecified value. The \textit{port} argument may be omitted, in which case it defaults to the value returned by \texttt{current-output-port}.

6.10.4. System interface

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

\texttt{(load filename)} \hspace{1cm} \textit{essential procedure}

\textit{Filename} should be a string naming an existing file containing Scheme source code. The \texttt{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 \texttt{load} procedure does not affect the values returned by \texttt{current-input-port} and \texttt{current-output-port}. \texttt{Load} returns an unspecified value.

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

\texttt{(transcript-on filename)} \hspace{1cm} \textit{procedure}
\texttt{(transcript-off)} \hspace{1cm} \textit{procedure}

\textit{Filename} must be a string naming an output file to be created. The effect of \texttt{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 \texttt{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, etc.) are formed from sequences of characters. The following sections describe how expressions and programs are formed from sequences of tokens.

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

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

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

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

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

(boolean) → #t | #f
(character) → #\ {any character}
  | #\ {character name}
(character name) → space | newline
(string) → " {string element}*
(string element) → (any character other than " or \)
  | \ | \"
(number) → (num 2) | (num 8)
  | (num 10) | (num 16)

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

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

(suffix) → (empty)
  | (exponent marker) (sign) (digit 10)
(exponent marker) → e | s | f | d | l
(sign) → (empty) | + | -
(exactness) → (empty) | #i | #e
(radix 2) → #b
(radix 8) → #o
(radix 10) → (empty) | #d
(radix 16) → #x
(digit 2) → 0 | 1
(digit 8) → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7
(digit 10) → (digit)
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 that parses as an (expression) will also parse as a (datum).

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

7.1.3. Expressions

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

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

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

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

(assignment) --- (set! (variable) (expression))

(derived expression) ---
    | (cond (cond clause)*)
    | (cond (cond clause) (else (sequence)))
    | (case (expression)
        (case clause)*)
    | (and (test)*)
    | (or (test)*)
    | (let ((binding spec) (body))
    | (let (variable) (binding spec) (body))
    | (let* (binding spec) (body))
    | (letrec (binding spec) (body))
    | (begin (sequence))
    | (do (iteration spec)*)
        (test) (sequence)
        (command)*
    | (delay (expression))
    | (quasiquote)

(cond clause) --- ((test) (sequence))
    | (test)
    | (test => (recipient))
(recipient) --- (expression)
(case clause) --- ((datum)* (sequence))
(binding spec) --- (variable) (expression)
(iteration spec) --- (variable) (init) (step)
    | (variable) (init)
(init) --- (expression)
(step) --- (expression)

7.1.4. Quasiquotations

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

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

### 7.1.5. Programs and definitions

\[(\text{program}) \rightarrow (\text{command or definition})^*\]

\[(\text{command or definition}) \rightarrow (\text{command}) \mid (\text{definition})\]

\[(\text{definition}) \rightarrow (\text{define} \ (\text{variable}) \ (\text{expression}))\]

\[(\text{define} \ ((\text{variable}) \ (\text{def forms})) \ (\text{body}))\]

\[(\text{def forms}) \rightarrow (\text{variable})^*\]

\[(\text{variable})^* . \ (\text{variable})\]

### 7.2. Formal semantics

This section provides a formal denotational semantics for the primitive expressions of Scheme and selected built-in procedures. The concepts and notation used here are described in [93]; the notation is summarized below:

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

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

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

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

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

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

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

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

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

#### 7.2.1. Abstract syntax

- \(K \in \text{Con}\) constants, including quotations
- \(I \in \text{Ide}\) identifiers (variables)
- \(E \in \text{Exp}\) expressions
- \(\Gamma \in \text{Com} = \text{Exp}\) commands

\[\text{Exp} \rightarrow K \mid I \mid (\text{E}_0 \ E^*)\]

\[\mid (\text{lambda} \ ((I^*) \ \Gamma^* \ E_0)\]

\[\mid (\text{lambda} \ ((I^* \ . \ I) \ \Gamma^* \ E_0)\]

\[\mid (\text{lambda} \ I \ \Gamma^* \ E_0)\]

\[\mid (\text{if} \ E_0 \ E_1 \ E_2) \mid (\text{if} \ E_0 \ E_1)\]

\[\mid (\text{set!} \ I \ E)\]

#### 7.2.2. Domain equations

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

#### 7.2.3. Semantic functions

- \(\mathcal{K} : \text{Con} \rightarrow E\)
- \(\mathcal{E} : \text{Exp} \rightarrow U \rightarrow K \rightarrow C\)
- \(\mathcal{E}^* : \text{Exp}^* \rightarrow U \rightarrow K \rightarrow C\)
- \(\mathcal{C} : \text{Com}^* \rightarrow U \rightarrow C \rightarrow C\)

Definition of \(\mathcal{K}\) deliberately omitted.

\[\mathcal{E}[K] = \lambda \rho \ . \ \text{send}(\mathcal{K}^{[K]} \ k)\]
7. Formal syntax and semantics

7.2.4. Auxiliary functions

\textit{lookup}: \( U \rightarrow \text{Ide} \rightarrow L \)
\[ \text{lookup} = \lambda \rho \Pi . \rho \Pi \]
\textit{extends}: \( U \rightarrow \text{Ide}^* \rightarrow L^* \rightarrow U \)
\[ \text{extends} = \lambda \rho \Pi^* \alpha^*. \#!I^* = 0 \rightarrow \rho, \]
\[ \text{extends}(\rho((\alpha^* \downarrow 1)/(!^* \downarrow 1)))(!^* \downarrow 1)(\alpha^* \downarrow 1) \]
\textit{wrong}: \( X \rightarrow C \) [implementation-dependent]
\textit{send}: \( E \rightarrow K \rightarrow C \)
\[ \text{send} = \lambda k \kappa . \kappa(k) \]
\textit{single}: \( (E \rightarrow C) \rightarrow K \)
\[ \text{single} = \lambda \psi \kappa^* . \#e^* = 1 \rightarrow (\psi^* \downarrow 1), \]
\textit{wrong “wrong number of arguments”}
\textit{new}: \( S \rightarrow (L + \{ \text{error} \}) \) [implementation-dependent]
\textit{hold}: \( L \rightarrow K \rightarrow C \)
\[ \text{hold} = \lambda \alpha \kappa \sigma . \text{send}(\sigma \alpha \downarrow 1)\kappa \sigma \]
\textit{assign}: \( L \rightarrow E \rightarrow C \rightarrow C \)
\[ \text{assign} = \lambda \alpha \theta \sigma . \theta(\text{update} \alpha \sigma) \]
\textit{update}: \( L \rightarrow E \rightarrow S \rightarrow S \)
\textit{update} = \( \lambda \alpha \sigma \sigma . \sigma((\epsilon, \text{true})/\alpha) \)
\textit{tievals}: \( (L^* \rightarrow C) \rightarrow E^* \rightarrow C \)
\[ \text{tievals} = \lambda \psi \kappa^* . \#e^* = 0 \rightarrow \psi(\epsilon), \]
\[ \text{new} \sigma \in L \rightarrow \text{tievals}(\lambda \alpha^* . \psi((\text{new} \sigma \mid L) \alpha^*)) \]
\[ \text{tievals}(e^* \downarrow 1) \]
\[ \text{update(new} \sigma \mid L)(e^* \downarrow 1)\sigma, \]
\[ \text{wrong “out of memory”} \]
\textit{tievalrest}: \( (L^* \rightarrow C) \rightarrow (E^* \rightarrow C) \)
\[ \text{tievalrest} = \lambda \psi \kappa^* . \text{list(dropfirst}(e^* \nu)) \]
\[ \text{dropfirst} = \lambda n . n = 0 \rightarrow l, \text{dropfirst}(l \downarrow 1)(n - 1) \]
\textit{takefirst}: \( (E^* \rightarrow C) \rightarrow (E^* \rightarrow C) \)
\[ \text{truish} : E \rightarrow T \]
\[ \text{truish} = \lambda e . e = \text{false} \rightarrow \text{false}, \text{true} \]
\textit{permute}: \( \text{Exp}^* \rightarrow \text{Exp}^* \) [implementation-dependent]
\textit{unpermute}: \( E^* \rightarrow E^* \) [inverse of permute]
\textit{apply}: \( E \rightarrow E^* \rightarrow K \rightarrow C \)
\[ \text{apply} = \lambda e^* \kappa \epsilon \in F \rightarrow (\epsilon \uparrow F \downarrow (e^* \nu \kappa), \text{wrong “bad procedure”} \]
\textit{oneary}: \( (E \rightarrow K \rightarrow C) \rightarrow (E^* \rightarrow K \rightarrow C) \)
\[ \text{oneary} = \lambda \epsilon^* \kappa . \#e^* = 1 \rightarrow (\epsilon^* \downarrow 1)\kappa, \]
\[ \text{wrong “wrong number of arguments”} \]
\textit{twoary}: \( (E \rightarrow E \rightarrow K \rightarrow C) \rightarrow (E^* \rightarrow K \rightarrow C) \)
\[ \text{twoary} = \lambda \epsilon^* \kappa . \#e^* = 2 \rightarrow (\epsilon^* \downarrow 1)(\epsilon^* \downarrow 2)\kappa, \]
\[ \text{wrong “wrong number of arguments”} \]
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.

\[(e_1 \in E_3 \land e_2 \in E_3) \rightarrow \ldots,\]
\[(e_1 \in F \land e_2 \in F) \rightarrow \]
\[send((e_1 \mid F \downarrow 1) = (e_2 \mid F \downarrow 1) \rightarrow \text{true}, \text{false})\]
\[\kappa,\]
\[send \text{false} \kappa\]

\[\text{apply} : E^* \rightarrow K \rightarrow C\]
\[\text{apply} = \]
\[\text{twoarg}(\lambda e_1 e_2 k. e_1 \in F \rightarrow \text{valueslist}(e_2)(\lambda e^*. \text{apply} e_1 e^* k),\]
\[\text{wrong "bad procedure argument to apply"}\]
\[\text{valueslist} = \]
\[\text{onearg}(\lambda e k. e \in E_p \rightarrow \]
\[\text{cdr}(e)\]
\[\text{\lambda e^*. \text{valueslist}}\]
\[\text{\epsilon}^*\]
\[\text{\lambda e^*. \text{car} (e) (\text{values} (\lambda e. \kappa ((\epsilon \downarrow \kappa^*))))},\]
\[\epsilon = \text{null} \rightarrow \kappa (\text{),}\]
\[\text{wrong "non-list argument to values-list"}\]
\[\text{cwcc} : E^* \rightarrow K \rightarrow C \quad \text{[call-with-current-continuation]}\]
\[\text{cwcc} = \]
\[\text{onearg}(\lambda e k. e \in F \rightarrow \]
\[\text{\lambda} \sigma. \text{new} \sigma \in L \rightarrow \]
\[\text{\text{apply} e}\]
\[\text{\langle \text{new} \sigma \mid L, \lambda e^* \kappa^* \rangle \in E}\]
\[\kappa\]
\[\text{\text{update} (new \sigma \mid L)}\]
\[\text{unspecified} \sigma,\]
\[\text{wrong "out of memory" \sigma),}\]
\[\text{wrong "bad procedure argument"}\]

\[\text{list} : E^* \rightarrow K \rightarrow C\]
\[\text{list} = \]
\[\lambda e^* \kappa. \# e^* = 0 \rightarrow \text{send null} \kappa,\]
\[\text{list} (e^* \downarrow 1) \rightarrow (\text{single} (\lambda e. \text{cons} (e^* \downarrow 1, e) \kappa))\]

\[\text{cons} : E^* \rightarrow K \rightarrow C\]
\[\text{cons} = \]
\[\text{twoarg}(\lambda e_1 e_2 k \sigma. \text{new} \sigma \in L \rightarrow \]
\[\text{\text{\lambda} \sigma^*. \text{new} \sigma^* \in L \rightarrow \]
\[\text{send} ((\text{new} \sigma \mid L, \text{new} \sigma^* \mid L, \text{true})\]
\[\text{in} E)\]
\[\kappa\]
\[\text{\text{update} (new \sigma^* \mid L) e_2 \sigma^*,}\]
\[\text{wrong "out of memory" \sigma^*}\]
\[\text{(update} (\text{new} \sigma \mid L) e_1 \sigma),\]
\[\text{wrong "out of memory" \sigma)}\]

\[\text{less} : E^* \rightarrow K \rightarrow C\]
\[\text{less} = \]
\[\text{twoarg}(\lambda e_1 e_2 k. e_1 \in R \land e_2 \in R \rightarrow \]
\[\text{send} ((e_1 \mid R \downarrow e_2 \mid R \rightarrow \text{true}, \text{false}) \kappa,\]
\[\text{wrong "non-numeric argument to <"}\]

\[\text{add} : E^* \rightarrow K \rightarrow C\]
\[\text{add} = \]
\[\text{twoarg}(\lambda e_1 e_2 k. e_1 \in R \land e_2 \in R \rightarrow \]
\[\text{send} ((e_1 \mid R \downarrow e_2 \mid R \rightarrow \text{true}, \text{false}) \kappa,\]
\[\text{wrong "non-numeric argument to +"}\]

\[\text{car} : E^* \rightarrow K \rightarrow C\]
\[\text{car} = \]
\[\text{onearg}(\lambda e k. e \in E_p \rightarrow \text{hold} (e \mid E_p \downarrow 1) \kappa,\]
\[\text{wrong "non-pair argument to car"}\]

\[\text{cdr} : E^* \rightarrow K \rightarrow C \quad \text{[similar to car]}\]
\[\text{setcor} : E^* \rightarrow K \rightarrow C\]
\[\text{setcor} = \]
\[\text{twoarg}(\lambda e_1 e_2 k. e_1 \in E_p \rightarrow \]
\[\text{assign} (e_1 \mid E_p \downarrow 3) \rightarrow (e_2 \]
\[\text{send unspecified} \kappa),\]
\[\text{wrong "immutable argument to set-car!"},\]
\[\text{wrong "non-pair argument to set-car!"}\]

\[\text{equiv} : E^* \rightarrow K \rightarrow C\]
\[\text{equiv} = \]
\[\text{twoarg}(\lambda e_1 e_2 k. e_1 \in M \land e_2 \in M \rightarrow \]
\[\text{send} ((e_1 \mid M \downarrow E_2 \mid M \rightarrow \text{true}, \text{false}) \kappa,\]
\[\text{send} ((e_1 \mid Q \downarrow e_2 \mid Q \rightarrow \text{true}, \text{false}) \kappa,\]
\[\text{send} ((e_1 \mid H \downarrow e_2 \mid H \rightarrow \text{true}, \text{false}) \kappa,\]
\[\text{send} ((e_1 \mid R \downarrow e_2 \mid R \rightarrow \text{true}, \text{false}) \kappa,\]
\[\text{send} ((e_1 \mid E_p \downarrow e_2 \mid E_p \rightarrow \]
\[\text{send} ((\lambda p. p_2. ((p_1 \downarrow 1) = (p_2 \downarrow 1) \land \]
\[\text{\lambda} \lambda \text{false})\]
\[\text{false})\]
\[\text{false})\]
\[\kappa,\]
\[\text{(e_1 \mid E_p)}\]
\[\text{(e_2 \mid E_p)}\]
\[\kappa,\]
\[\text{(e_2 \in E_r)}\]
7. Formal syntax and semantics

\[(\text{cond} \ (\text{else} \ \{\text{sequence}\}))\]
\[\equiv \ (\text{begin} \ \{\text{sequence}\})\]

\[(\text{cond})\]
\[\equiv \ \{\text{some expression returning an unspecified value}\}\]

\[(\text{case} \ \{\text{key}\})\]
\[\{(d_1 \ldots) \ \{\text{sequence}\}\}
\[\ldots\]
\[\equiv \ (\text{let} \ ((\{\text{key} \ (\text{key})\})
\ (\text{thunk} \ (\lambda x \ (\text{x (thunk) x}))))\]

\[\text{(case) \ \{\text{key}\})\]
\[\{(d_1 \ldots) \ \{\text{sequence}\}\}
\[\ldots\]
\[\text{(else} \ f_1 \ f_2 \ldots)\]
\[\text{(sequence)}\]
\[\ldots\]
\[\equiv \ (\text{let} \ ((\{\text{key} \ (\text{key})\})
\ (\text{thunk} \ (\lambda x \ (\text{x (thunk) x}))))\]

\[(\{\text{sequence}\})\]
\[\equiv \ (\text{begin} \ \{\text{sequence}\})\]

\[\text{where} \ \{\text{memv}\} \text{is an expression evaluating to the memv procedure.}\]

\[\text{(and)} \quad \equiv \ #t\]
\[\text{and} \ \{\text{test}\} \quad \equiv \ \{\text{test}\}\]
\[\text{and} \ \{\text{test}_1\} \ \{\text{test}_2\} \ldots\]
\[\equiv \ (\text{let} \ ((x \ \{\text{test}_1\}))
\ (\text{thunk} \ (\lambda x \ (\text{x (thunk) x}))))\]

\[\text{(or)} \quad \equiv \ #f\]
\[\text{or} \ \{\text{test}\} \quad \equiv \ \{\text{test}\}\]
\[\text{or} \ \{\text{test}_1\} \ \{\text{test}_2\} \ldots\]
\[\equiv \ (\text{let} \ ((x \ \{\text{test}_1\}))
\ (\text{thunk} \ (\lambda x \ (\text{x (thunk) x}))))\]

\[\text{let} \ ((\{\text{variable}_1\} \ \{\text{init}_1\}) \ldots)\]
\[\{\text{body}\} \quad \equiv \ ((\lambda x \ (\text{variable}_1 \ldots) \ \{\text{body}\} \ \{\text{init}_1\} \ldots)\]

\[\text{let*} \ () \ \{\text{body}\} \quad \equiv \ ((\lambda x \ (\text{variable}_1 \ldots) \ \{\text{body}\} \ \{\text{init}_1\} \ldots)\]

\[\text{let*} \ ((\{\text{variable}_1\} \ \{\text{init}_1\})
\ (\{\text{variable}_2\} \ \{\text{init}_2\}) \ldots)\]
\[\{\text{body}\} \quad \equiv \ (\text{let} \ ((\{\text{variable}_1\} \ \{\text{init}_1\}))
\ (\text{let*} \ ((\{\text{variable}_2\} \ \{\text{init}_2\}) \ldots)
\ (\text{body}))\]

\[\text{letrec} \ ((\{\text{variable}_1\} \ \{\text{init}_1\}) \ldots)\]
\[\{\text{body}\} \quad \equiv \ ((\text{letrec} \ ((\{\text{variable}_1\} \ldots)
\ (\text{lambda} \ ((\text{variable}_1) \ldots)
\ (\text{if} \ \{\text{test}\} \ (\text{begin} \ \{\text{sequence}\})
\ (\text{begin} \ \{\text{command}\} \ldots)
\ (\text{loop} \ \{\text{step}_1\} \ldots))))\]

\[\text{where} \ \{\text{loop}\} \text{is any variable which is distinct from}\]
\[\{\text{variable}_1\}, \ldots, \text{and which does not occur free in the do expression.}\]

\[\text{let} \ \{\text{variable}_0\} \ ((\{\text{variable}_1\} \ \{\text{init}_1\}) \ldots)
\ (\text{body})\]
\[\equiv \ ((\text{letrec} \ ((\{\text{variable}_0\} \ (\text{lambda} \ ((\text{variable}_1) \ldots)\)
\ (\text{if} \ \{\text{test}\} \ (\text{begin} \ \{\text{sequence}\})
\ (\text{begin} \ \{\text{command}\} \ldots)
\ (\text{loop} \ \{\text{step}_1\} \ldots))))\]

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

\[\text{begin} \ (\{\text{expression}\}) \quad \equiv \ \{\text{expression}\}\]
\[\text{begin} \ (\text{command}) \ (\text{sequence})\]
\[\equiv \ ((\text{lambda} \ (\text{ignore thunk}) \ (\text{thunk})
\ (\text{command}))
\ (\text{lambda} \ () \ (\text{begin} \ \{\text{sequence}\})))\]

\[\text{The following expansion for do is simplified by the assumption that no}\]
\[\{\text{step}\} \text{is omitted. Any do expression in which a}\]
\[\{\text{step}\} \text{is omitted can be replaced by an equivalent do expression in which the corresponding}\]
\[\{\text{variable}\} \text{appears as the}\]
\[\{\text{step}\}.\]

\[\text{do} \ ((\{\text{variable}_1\} \ \{\text{init}_1\} \ \{\text{step}_1\}) \ldots)
\ (\{\text{test}\} \ (\text{sequence})
\ (\text{command}_1) \ldots)
\equiv \ ((\text{letrec} \ ((\text{loop}))
\ (\text{lambda} \ ((\text{variable}_1) \ldots)
\ (\text{if} \ \{\text{test}\} \ (\text{begin} \ \{\text{sequence}\})
\ (\text{begin} \ \{\text{command}_1\} \ldots)
\ (\text{loop} \ \{\text{step}_1\} \ldots))))\]
(body))
  (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" report [68] was published.

- Although implementations may extend Scheme, they must offer a syntactic mode that adds no reserved words and preempts no lexical conventions of Scheme.

- Implementations may report violations of implementation restrictions.

- It is no longer specified whether the empty list counts as true or as false in conditional expressions. It should be noted that the IEEE standard for Scheme requires the empty list to count as true [50].

- The sets defined by boolean?, pair?, symbol?, number?, char?, string?, vector?, and procedure? are required to be disjoint.

- The variables bound by a lambda, let, letrec, and do must not contain duplicates.

- Nested begin expressions containing definitions are treated as a sequence of definitions.

- The eqv? procedure is no longer required to be true of any two empty strings or two empty vectors.

- The syntax of numerical constants has been changed, and the exactness implied by each syntax has been specified.

- The semantics of many numerical procedures have been clarified.

- Rationalize has been restricted to two arguments and its specification clarified.

- The number->string and string->number procedures have been changed.

- Integer->char now requires an exact integer argument.

- The specification of the force procedure has been weakened. The previous specification was unimplementable.

- Variables removed: t, nil.

- Procedures removed: approximate, last-pair.

- Procedures added: list?, peek-char.

- Syntaxes made essential: case, and, or, quasiquote.

- Procedures made essential:

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>reverse</td>
<td>char-ci=?</td>
</tr>
<tr>
<td>max</td>
<td>char-ci&lt;&gt;?</td>
</tr>
<tr>
<td>min</td>
<td>char-ci&lt;&lt;?</td>
</tr>
<tr>
<td>modulo</td>
<td>char-ci&lt;=?</td>
</tr>
<tr>
<td>gcd</td>
<td>char-ci=&gt;?</td>
</tr>
<tr>
<td>lcm</td>
<td>char-alphabetic?</td>
</tr>
<tr>
<td>floor</td>
<td>char-numeric?</td>
</tr>
<tr>
<td>ceiling</td>
<td>char-white-space?</td>
</tr>
<tr>
<td>truncate</td>
<td>char-lower-case?</td>
</tr>
<tr>
<td>round</td>
<td>char-upper-case?</td>
</tr>
<tr>
<td>number-&gt;string</td>
<td>char-upcase</td>
</tr>
<tr>
<td>string-&gt;number</td>
<td>char-downcase</td>
</tr>
<tr>
<td>make-string</td>
<td>string-set!</td>
</tr>
<tr>
<td>string-\set!</td>
<td>string-ci=?</td>
</tr>
<tr>
<td>open-input-file</td>
<td></td>
</tr>
<tr>
<td>open-output-file</td>
<td></td>
</tr>
<tr>
<td>close-input-port</td>
<td></td>
</tr>
<tr>
<td>close-output-port</td>
<td></td>
</tr>
</tbody>
</table>

- Procedures required to accept more general numbers of arguments: append, +, *, - (one argument), / (one argument), =, <, >, <=, >=, map, for-each.

- A macro facility has been added as an appendix to this report.
EXAMPLE

Integrate-system integrates the system
\[ y'_k = f_k(y_1, y_2, \ldots, y_n), \quad k = 1, \ldots, n \]
of differential equations with the method of Runge-Kutta.

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

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

\begin{verbatim}(define integrate-system
  (lambda (system-derivative initial-state h)
    (let ((next (runge-kutta-4 system-derivative h)))
      (letrec ((states
          (cons initial-state
                (delay (map-streams next states))))))
        states))))
\end{verbatim}

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

\begin{verbatim}(define runge-kutta-4
  (lambda (f h)
    (let* ((k0 (*h (scale-vector h)))
           (k1 (*h f (add-vectors y (*1/2 k0))))
           (k2 (*h f (add-vectors y (*1/2 k1))))
           (k3 (*h f (add-vectors y k2))))
      (add-vectors y
                   (*1/6 (add-vectors k0
                           (*2 k1)
                           (*2 k2)
                           k3))))))
\end{verbatim}

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

\begin{verbatim}(define map-streams
  (lambda (f s)
    (cons f (head s))
    (delay (map-streams f (tail s))))))
\end{verbatim}

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.

\begin{verbatim}(define head car)
(define tail
  (lambda (stream) (force (cdr stream))))
\end{verbatim}

The following illustrates the use of \texttt{integrate-system} in integrating the system
\[ \frac{dV_C}{dt} = -i_L - \frac{V_C}{R} \]
\[ L \frac{di_L}{dt} = V_C \]

which models a damped oscillator.

\begin{verbatim}(define damped-oscillator
  (lambda (R L C)
    (lambda (state)
      (let ((Vc (vector-ref state 0))
            (Il (vector-ref state 1))
            (v0 (- 0 (+ (/ Vc (* R C)) (/ Il C))))
            (v1 (+ (/ Il C) v0))
            (v2 (+ (/ Il C) v1))
            (v3 (+ (/ Il C) v2)))
        (vector v0 v1 v2 v3)))))
\end{verbatim}

\begin{verbatim}(define elementwise
  (lambda (f)
    (lambda vectors
      (generate-vector
       (vector-length (car vectors))
       (lambda (i)
         (apply f
               (map (lambda (v) (vector-ref v i))
                    vectors)))))))
\end{verbatim}

\begin{verbatim}(define generate-vector
  (lambda (size proc)
    (let ((ans (make-vector size)))
      (letrec ((loop
                    (lambda (i)
                      (cond ((= i size) ans)
                            (else
                             (vector-set! ans i (proc i))
                             (loop (+ i 1)))))))
        (loop 0))))
\end{verbatim}
APPENDIX: MACROS

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

Derived expression types introduced using this extension have the syntax

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

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

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

The extension described here consists of three parts:

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

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

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

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

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

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

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

Binding syntactic keywords

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

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

These new expression types and the pattern language described in section 7.3 are added to Scheme by augmenting the BNF in section 7.1 with the following new productions. Note that the identifier \(...\) used in some of these productions is not a metasymbol.

\[
\begin{align*}
  \{\text{expression}\} & \rightarrow \{\text{macro use}\} \\
  & \quad \{\text{macro block}\} \\

  \{\text{macro use}\} & \rightarrow \{(\text{keyword})\ \{\text{datum}\}^*\} \\

  \{\text{keyword}\} & \rightarrow \{\text{identifier}\} \\

  \{\text{macro block}\} & \rightarrow \\
  & \quad \{\text{let-syntax} \ (\{\text{syntax spec}\}^* \ \{\text{body}\})\} \\
  & \quad \{\text{letrec-syntax} \ (\{\text{syntax spec}\}^* \ \{\text{body}\})\} \\

  \{\text{syntax spec}\} & \rightarrow \{(\text{keyword})\ \{\text{transformer spec}\}\} \\

  \{\text{transformer spec}\} & \rightarrow \\
  & \quad \{\text{syntax-rules} \ \{\text{identifier}\}^* \ \{\text{syntax rule}\}^*\} \\

  \{\text{syntax rule}\} & \rightarrow \{(\text{pattern}) \ \{\text{template}\}\} \\

  \{\text{pattern}\} & \rightarrow \{(\text{identifier})\} \\
  & \quad \{(\text{pattern})^* \ \{\text{pattern}\}\} \\
  & \quad \{(\text{pattern})^* \ \{\text{pattern}\} \ \{\text{ellipsis}\}\}
\end{align*}
\]
Appendix: Macros

| (pattern datum) → (vector) |
| (string) |
| (character) |
| (boolean) |
| (number) |
| (template) → (pattern identifier) |
| ((template element)*) |
| ((template element)⁺ . (template)) |
| (template datum) |
| (template element) → (template) |
| (template) (ellipsis) |
| (template datum) → (pattern datum) |
| (pattern identifier) → (any identifier except ...) |
| (ellipsis) → (the identifier ...) |
| (command or definition) → (syntax definition) |
| (syntax definition) → |
| (define-syntax (keyword) (transformer spec)) |
| (begin (syntax definition)⁺) |

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

```scheme
(define define 3)
(begin (define begin list))
```

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

```scheme
(let-syntax (bindings) (body))
Syntax: (Bindings) should have the form
(((keyword) (transformer spec) ...)

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

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

```scheme
(let-syntax (when (syntax-rules ()
        ((when test stmt1 stmt2 ...)
       (if test
        (begin stmt1
        stmt2 ...))))))
  (let ((if #t))
    (when if (set! if 'now)
    if))
    ⇒ now

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

(letrec-syntax (bindings) (body))
Syntax: Same as for let-syntax.

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

```scheme
(letrec-syntax
  ((or (syntax-rules ()
        ((or #f)
       ((or e) e)
       ((or e1 e2 ...) (let ((temp e1))
        (if temp
        temp
        (or e2 ...)))))))))
```
(define-syntax let*
 (syntax-rules ()
  ((let* () body1 body2 ...)
   (let () body1 body2 ...))
  ((let* ((name val1) (name2 val2) ...) body1 body2 ...)
   (let ((name val1))
     (let ((name2 val2) ...) body1 body2 ...))))

Pattern language

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

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

  ((pattern) (template))

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

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

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

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

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

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

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

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

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

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

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

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

(define-syntax let
 (syntax-rules ()
  ((let ((name val) ...) body1 body2 ...)
   ((lambda (name ...) body1 body2 ...)
     val ...))
  ((let rec ((tag name) ...) body1 body2 ...)
   ((letrec ((tag (lambda (name ...) body1 body2 ...))
     body1 body2 ...)))
   tag)
The low-level macro system also adds the following procedures:

\[
\begin{align*}
\text{unwrap-syntax} & \quad \text{identifier} \to \text{symbol} \\
\text{identifier?} & \quad \text{generate-identifier} \\
\text{free-identifier=} & \quad \text{construct-identifier} \\
\text{bound-identifier=} & \\
\end{align*}
\]

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

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

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

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

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

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

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

\textbf{Syntax:} The \text{datum} may be any external representation of a Scheme object.

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

\[(\text{symbol? } (\text{syntax x})) \quad \Rightarrow \#f\]
\[(\text{let-syntax } ((\text{car } (\text{lambda } (x) (\text{syntax car}))))
\quad (\text{car} )' 0)\quad \Rightarrow 0\]
\[(\text{let-syntax}
\quad ((\text{quote-quote})
\quad (\text{lambda } (x) (\text{list } (\text{syntax quote} ) ' \text{quote} )))
\quad (\text{quote-quote})\quad \Rightarrow \text{quote}
\quad (\text{let-syntax}
\quad ((\text{quote-quote})
\quad (\text{lambda } (x) (\text{list } ' \text{quote} ' \text{quote} )))
\quad (\text{quote-quote})\quad \Rightarrow \text{error})\]

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

\[(\text{let-syntax} ((\text{quote-me}
\quad (\text{lambda } (x)
\quad (\text{list } (\text{syntax quote} ) x)))
\quad (\text{quote-me please}))\quad \Rightarrow (\text{quote-me please}))\]
\[(\text{let } ((x 0))
\quad (\text{let-syntax} ((\text{alpha } (\text{lambda } (e) (\text{syntax x}))
\quad (\text{alpha})))\quad \Rightarrow 0\]
\[(\text{let } ((x 0))
\quad (\text{let-syntax} ((\text{alpha } (\text{lambda } (x) (\text{syntax x}))
\quad (\text{alpha}))))\quad \Rightarrow \text{error}\]
\[(\text{let-syntax} ((\text{alpha}
\quad (\text{let-syntax} ((\text{beta}
\quad (\text{syntax-rules } ()
\quad ((\text{beta} )' 0)))
\quad (\text{lambda } (x) (\text{syntax } (\text{beta})))))))
\quad (\text{alpha})\quad \Rightarrow \text{error})\]

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

\[(\text{let-syntax} ((\text{alpha } (\text{syntax-rules } ()
\quad ((\text{alpha} )' 0))))\]
\[(\text{let-syntax} ((\text{beta } (\text{lambda } (x) (\text{alpha}))))
\quad (\text{beta}))\quad \Rightarrow 0\]
\[(\text{let } ((\text{list } 0))
\quad (\text{let-syntax} ((\text{alpha } (\text{lambda } (x) (\text{list } 0))))
\quad (\text{alpha}))\quad \Rightarrow \text{error})\]

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

\textbf{Note:} It has been suggested that \#'\text{datum} and \#\text{datum} would be felicitous abbreviations for \text{syntax } (\text{datum})\text{ and quasisyntax } (\text{datum}), respectively, where quasisyntax, which is not described in this appendix, would bear the same relationship to syntax that quasiquote bears to quote.

\[(\text{identifier? syntax-object})\quad \text{procedure}\]
Returns \#t if syntax-object represents an identifier, otherwise returns \#f.

\[(\text{identifier? syntax x})\quad \Rightarrow \#t\]
\[(\text{identifier? quote x})\quad \Rightarrow \#f\]
\[(\text{identifier? 3})\quad \Rightarrow \#f\]

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

\[(\text{identifier? unwrap-syntax syntax x})\quad \Rightarrow \#t\]
\[(\text{identifier? car unwrap-syntax (syntax x)))\quad \Rightarrow \#t\]
\[(\text{unwrap-syntax cdr (unwrap-syntax (syntax x)))\quad \Rightarrow ()}\]

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

(free-identifier? (syntax x) (syntax y))  
⇒ #f

(let ((x (syntax x)))
  (free-identifier? x (syntax x)))  
⇒ #f

(let-syNTAX  
  (alpha
   (lambda (x)
     (free-identifier? (car (unwrap-syNTAX x)) (syntax alpha))))))  
⇒ #f

(letrec-syNTAX  
  (alpha
   (lambda (x)
     (free-identifier? (car (unwrap-syNTAX x)) (syntax alpha))))))  
⇒ #t

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

(letrec-syNTAX  
  (alpha
   (lambda (x)
     (bound-identifier? (car (unwrap-syNTAX x)) (syntax alpha))))))  
⇒ #f

(identifier->symbol id)  
⇒ x

(generate-identifier)  
⇒ #t

(generate-identifier symbol)  
⇒ #f

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

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

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

(identifier->symbol (generate-identifier 'x))  
⇒ x

(bound-identifier? (generate-identifier 'x))  
⇒ #f

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

(construct-identifier id symbol)  
⇒ #t

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

Another example is a macro that implicitly binds exit:

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

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

Acknowledgements

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

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

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

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


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


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.

; ;

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

#b 20; 32
backquote 11
begin 10; 11, 12, 37, 38
binding 6
binding construct 6
boolean? 13; 38
bound 6
bound-identifier? 45
caar 16
caddr 16
cadr 16
call 8
call by need 11
call-with-current-continuation 28; 29, 36
call-with-input-file 29
call-with-output-file 29
call/cc 29
car 16; 7, 15, 36
case 9; 38
catch 29
cddddar 16
cddddd 16
cdr 16; 15, 28
ceiling 22
char->integer 25
char-alphabetic? 25
char-ci<=? 25
char-ci<? 25
char-ci=? 25
char-ci<=? 25
char-ci>? 25
char-downcase 25
char-lower-case? 25
char-numeric? 25
char-ready 30
char-ready? 30; 31
char-upcase 25
char-upper-case? 25
char-whitespace? 25
char<=? 24; 25
char< 24; 26
char=? 24; 14, 25
char>=? 24
char> 24
char? 24; 38
char->integer 25
close-input-port 30
close-output-port 30
combination 8
comma 11
comment 5; 32
complex? 20; 19, 21
cond 9; 13
cons 16; 15
constant 7
construct-identifier 45
continuation 29
cos 23
current-input-port 30; 31
current-output-port 30; 31

#d 20
d 20
define 12; 13
define-syntax 41
definition 12
delay 11; 28
denominator 22
number 18
number->string 23; 38
number? 20; 19, 21, 38
number->string 23; 24
numerator 22
numerical types 19

#o 20; 32
object 3
odd? 21
open-input-file 30
open-output-file 30
or 9, 13, 38
output-port? 29

pair 15
pair? 16; 38
peek-char 30; 38
port 29
positive? 21
predicate 13
procedure call 8
procedure? 27; 38
promise 11; 28

quasiquote 11; 12, 16, 38
quote 7; 6, 16
quotient 22

rational? 20; 19, 21
rationalize 22; 38
read 30; 3, 6, 16, 18, 33
read-char 30
real-part 23
real? 20; 19, 21
referentially transparent 40
region 6; 9, 10, 11
remainder 22
return 29
reverse 17
round 22
runge-kutta-4 30

s 20
sequence 11
set! 9; 12, 35, 36
set-car! 16; 7, 15, 36
set-cdr! 16; 15
simplest rational 22
sin 23
sqrt 23; 20
string 25
string->list 26
string->number 24; 38
string->symbol 18
string-append 26

string-ci<=? 26
string-ci<? 26
string-ci=? 26
string-ci=>? 26
string-ci>? 26
string-copy 26
string-fill! 26
string-length 25; 19
string-ref 25; 7
string-set! 26; 7, 18
string<=? 26
string<? 26
string=? 26
string=>? 26
string>? 26
string? 25; 38
string->list 26
string->symbol 18
substring 26
symbol->string 18; 7
symbol? 18; 38
symbol->string 18
syntactic keyword 5; 6, 32, 40
syntax 44
syntax-rules 42

#t 13; 38
t 38
tan 23
token 32
top level environment 13; 6
transcript-off 31
transcript-on 31
true 6; 8, 9, 13
truncate 22
type 7

unbound 6; 7, 12
unquote 12; 16
unquote-splicing 12; 16
unspecified 4
unwrap-syntax 44

valid indexes 25; 26
values-list 36
variable 6; 5, 7, 32
vector 27
vector->list 27
vector-fill! 27
vector-length 27; 19
vector-ref 27; 7
vector-set! 27
vector? 26; 38

whitespace 5
with-input-from-file 30
with-output-to-file 30
write 31; 6, 12, 18
write-char 31

#x 20; 32

zero? 21
Macros in Scheme

William Clinger

Although macros are even older than Lisp, Scheme has just become the first block-structured programming language to support a reliable macro system. This macro system is described in the appendix to the Revised Report on the Algorithmic Language Scheme [1]. This technical report contains three more articles on the subject of macros in Scheme, counting the article you are reading. I wrote this as an introduction to the other two articles, because it is very hard to understand their purpose without some further background on macros in Scheme.

Macros are hardly new in the Lisp world, so I can explain what is new about the Scheme macro system by comparing it to Common Lisp's [5]. Consider a simple push macro that is like the push macro in Common Lisp except that the place being pushed is required to be a variable. In Scheme we can write this macro as

```
(define-syntax push
  (syntax-rules ()
    ((push item place)
     (set! place (cons item place)))))
```

and we can use push in an expression such as

```
(let* ((cons (lambda (name)
               (case name
                   ((phil) '("three-card monte")
                    ((dick) '("secret plan to end the war"
                                 "agnes"
                                 "not a crook")
                    ((jimmy) '("why not the best")
                     ((ron) '("abolish the draft"
                                 "balance the budget")
                        (else '())))
                    (scams (cons 'phil)))
                (push (car (cons 'jimmy)) scams)
                (push (cadr (cons 'ron)) scams)
                scams))

This expression will evaluate to

("balance the budget" "why not the best" "three-card monte")
```
That, at least, is what this expression will evaluate to in Scheme.

We're not supposed to write expressions like this in Common Lisp, according to \texttt{LISP-SYMBOL-REDEFINTION:MAR89-X3J13}. (The "consequences are undefined"). One reason for this decision by X3J13 is that the Common Lisp macro system is unreliable, so a lexical binding of \texttt{cons} as in the above example can break macros such as \texttt{push}. The decision by X3J13 doesn't really solve the problem, because it doesn't address macros that refer to global functions written by the user of a Common Lisp system, but it does make Common Lisp significantly less unreliable in practice.

Recognizing that \texttt{LISP-SYMBOL-REDEFINTION:MAR89-X3J13} doesn't completely solve the problem, let's push forward with our example as though X3J13 didn't exist. (Otherwise I would have to construct a different example!)

If we translate everything to pre-X3J13 Common Lisp we will get an error, because in Common Lisp the first use of \texttt{push} would expand into

\begin{verbatim}
(setq scams (cons (cons 'jimmy) scams))
\end{verbatim}

within a context where \texttt{cons} is bound to a one-argument procedure. There are various ways one might try to work around this problem in Common Lisp, but none are fully general. If a macro needs to refer to a global variable or function (other than those predefined in the \texttt{common-lisp} package, which X3J13 has effectively made into reserved words), then it is quite impossible to write that macro reliably using the Common Lisp macro system.

I'm as willing as anyone to pick on Common Lisp, but that's not what I'm doing here. When Common Lisp was being designed Scheme had no macro system at all, for this very reason: Macros were inherently unreliable, or so we thought then. Even today, except for Scheme and a couple of research languages that employ a Scheme-style macro system, Common Lisp probably has the most sophisticated macros of any programming language. The fact is that macros have been unreliable in block-structured programming languages for thirty years, and that unreliability has just been accepted as the state of the art.

How does Scheme's new macro system manage to avoid this problem? In effect, the macro expander systematically renames local variables to avoid all inadvertent captures of bound variables, so the macro-expanded form of the expression above will be something like

\begin{verbatim}
(let+ ((cons i (lambda (name)
                    (case name
                        ((phil) 'three-card-monte)
                        ((dick) 'secret-plan-to-end-the-war
                             agnew
                             not-a-crook)
                        ((jimmy) 'why-not-the-best)
                        ((ron) 'abolish-the-draft)

2
\end{verbatim}
"balance the budget")

(else '(())))

(scams (cons 1 'phil)))

(set! scams (cons (car (cons 1 'jimmy)) scams))

(set! scams (cons (cadr (cons 1 'ron)) scams))

scams)

A macro system that avoids inadvertent captures through systematic renaming is said to be hygienic. Eugene Kohlbecker developed and implemented the first hygienic macro system for his 1986 PhD dissertation [4]. In 1988 the Scheme community asked a four-person committee to design a hygienic macro system for inclusion in the Revised4 Report. In 1990 Jonathan Rees and I developed a more general and efficient form of Kohlbecker’s algorithm, which Jonathan used to implement a prototype of the Scheme macro system [2].

The Revised4 Report specifies the syntax and semantics of a hygienic macro system, but does not prescribe any particular algorithm. The Revised4 Report does describe a non-hygienic, low-level macro facility that could be used to implement the hygienic macro system I have been describing, but goes on to note that this particular low-level facility “is but one of several low-level facilities that have been designed and implemented to complement” Scheme’s hygienic macro system. The following articles describe two more of them.

Before moving on to the low-level facilities, I want to show you more of the power of Scheme’s hygienic macro system. First let us consider a simplified form of Common Lisp’s setf macro. With the Scheme macro system there are no reserved words, so we can redefine set! locally as in

(let-syntax
  (isyntax-rules (car cdr vector-ref)
    ((set! (car x) y) (set-car! x y))
    ((set! (cdr x) y) (set-cdr! x y))
    ((set! (vector-ref x e) y) (vector-set! x e y))
    ((set! x y) (set! x y)))

(let* ((days (list 'monday 'wednesday 'friday))
       (day1 'sunday))
  (set! (car days) 'tuesday)
  (set! day1 (car days))
  day1))

The (car cdr vector-ref) following syntax-rules means that car, cdr, and vector-ref are not pattern variables. They can match only themselves. The use of let-syntax instead of letrec-syntax means that the let-syntax macro is not recursive, so it can refer to the outer definition of set! in the last rule without circularity.

In Common Lisp, (setf (car days) 'tuesday) will not be expanded if it occurs within a local binding for car. (At least that is so according to
FUNCTION-NAME: SMALL. On the other hand, the consequences are supposed to be undefined if car is bound locally. Perhaps some Common Lisp wizard can explain this to me.) The purpose of this rule in Common Lisp is to ensure that the scope of a setf method that is associated with a function does not exceed the scope of the function itself.

This limitation of the scope of a set! method will be enforced automatically by the Scheme macro system, because the car in the pattern will match a car in a use only if the two occurrences of car are within the scope of the same binding of car. Since this scope is lexical (whether local or global), the Scheme macro system generalizes Common Lisp's rule for setf in the way that is appropriate for block-structured languages like Scheme.

Of course, none of this is special to setf. Scheme's hygienic macro system is a general mechanism for defining syntactic transformations that reliably obey the rules of lexical scope.

Another (rather silly) example might help to make the point:

(let ((car cdr)
   (set-car! set-cdr!)
   (cdr car)
   (set-cdr! set-car!))
(let-syntax
  ((set! (syntax-rules (car cdr vector-ref)
   ((set! (car x) y) (set-car! x y))
   ((set! (cdr x) y) (set-cdr! x y))
   ((set! (vector-ref x e) y) (vector-set! x e y))
   ((set! x y) (set! x y))))))
(let ((days (list 'monday 'wednesday 'friday))
   (set-car! (lambda () 17))
   (set! (car days) 'tuesday)
   (cons (set-car!) days)))

In Scheme this evaluates to (17 monday . tuesday). This is easy to see if you forget everything you thought you knew about macros, and rely only on the fact that all (non-pattern-variable) identifiers in the macro definition, whether they occur on the left or on the right hand side of a syntax rule, are resolved using Scheme's familiar lexical scope rules.

Alas, I cannot easily implement an analogue of Common Lisp's setf in Scheme using the high-level macro system. Although local macros can be either recursive (letrec-syntax) or non-recursive (let-syntax), global macros are always recursive (define-syntax). This should be fixed.

Another problem with macros in Scheme is that some macros are awkward or impossible to describe using the pattern language. Suppose we want to define a set*! macro to perform assignments in parallel, with

(set*! i1 e1 i2 e2 ...)
expanding into

\[(\text{let } ((t_1 \, e_1) \, (t_2 \, e_2) \, \ldots)\)
\[(\text{set! } i_1 \, t_1)\)
\[(\text{set! } i_2 \, t_2)\)
\[\ldots)\]

This is hard to express for two reasons. The first is that the variables to be assigned need to be paired up with the expressions giving their new values. The second difficulty is that the macro must generate an indefinite number of temporary variables.

The \texttt{set*!} macro can be defined using an auxiliary macro to perform the pairing and to generate the temporaries. Although it might appear that the same temporary is being used for each assignment, the hygienic macro system automatically renames each binding occurrence that is inserted by a macro, together with all occurrences within its scope. Since each temporary that is inserted by the auxiliary macro eventually becomes a binding occurrence, each will be renamed. (Amazing but true.)

\[(\text{define-syntax set*!})\]
\[(\text{syntax-rules ()})\]
\[(\text{(set*! i1 e1 more \ldots)})\]
\[(\text{(set*!-aux () i1 e1 more \ldots)})\])\]

\[(\text{define-syntax set*!-aux})\]
\[(\text{syntax-rules ()})\]
\[(\text{(set*!-aux ((i1 e1 t1) \ldots)})\]
\[(\text{(let ((t1 e1) \ldots)})\]
\[(\text{(set! i1 t1) \ldots)})\]
\[(\text{(set*!-aux ((i1 e1 t1) \ldots) i2 e2 more \ldots)})\]
\[(\text{(set*!-aux ((i1 e1 t1) \ldots (i2 e2 newtemp)) more \ldots)})\])\]

This definition of \texttt{set*!} may be compared with the \texttt{set*!} macro that is defined in the Revised\textsuperscript{4} Report using the low-level macro system. Although \texttt{set*!} was put forth as an example of a macro that would be easier to write in a low-level macro system, I prefer the high-level definition above.

This definition would be more elegant if the auxiliary macro were local to each use of \texttt{set*!}. That would be less efficient because the auxiliary macro would have to be re-compiled for each use, but the real reason I didn’t use a local macro is that I can’t express the macro I want using \texttt{syntax-rules}!

The problem is annoyingly syntactic: If I tried to use \texttt{syntax-rules} to say that \texttt{set*!} expands into a \texttt{letrec-syntax} that defines \texttt{set*!-aux}, then the seven ellipses that appear in the definition of \texttt{set*!-aux} would appear on the right hand side of the syntax rule for \texttt{set*!}. The macro expander would try
to expand these seven ellipses when transcribing the use of \texttt{set*!}, just as it expands the one ellipsis that appears on the right hand side of the syntax rule for \texttt{set*!} above. Somehow the macro expander needs to know to expand that one ellipsis, while waiting until the local macro is used to expand the other seven ellipses. A similar problem is solved in \LaTeX{} by an escape character, and this solution should work in Scheme as well.

The low-level system described in the Revised\textsuperscript{4} Report uses \texttt{syntax} as an escape character. Some such mechanism needs to be added to the high-level system.

Despite these problems, \texttt{syntax-rules} with recursion is reasonably expressive. It can express all recursive functions on lists, where \texttt{cons}, \texttt{car}, \texttt{cdr}, \texttt{null?}, and \texttt{equal?} on non-list elements are the base functions. The pattern language cannot do much with non-lists, however. It cannot take the successor of a numeral, for example.

The primary limitation of the hygienic macro system is that it is thoroughly hygienic, and thus cannot express macros that bind identifiers implicitly. Common Lisp's \texttt{defstruct} is an example of a non-hygienic macro, since it implicitly defines accessor functions for the structure. The \texttt{loop-until-exit} macro that is used as an example of the low-level macro system in the Revised\textsuperscript{4} Report is also a non-hygienic macro. It may usefully be compared with similar macros in the following articles.

In conclusion, the hygienic macro system described in the Revised\textsuperscript{4} Report is a considerable advance over previous macro systems. It has a few bugs yet, for which solutions can easily be imagined. Even if these bugs are fixed, however, there will still be non-hygienic macros and perhaps even a few hygienic macros that cannot be written using the hygienic macro system. That is why the Revised\textsuperscript{4} Report also describes a compatible low-level macro system.

The low-level macro system described in the Revised\textsuperscript{4} Report is experimental, and is likely to evolve considerably or be replaced altogether. Two alternative low-level macro systems are described in the following articles, and I am sure that even more will be designed during the next few years.

Scheme's hygienic macro system uses a stable technology, and Scheme programmers can use it to define most macros in a portable way. I am sure that the hygienic macro system will be extended, but I do not expect it to change very much. The hygienic macro system should certainly be considered for inclusion when the IEEE standard for Scheme [3] is next revised.

The low-level macro facilities provide an escape for those few macros that cannot be written using the hygienic macro system, but macros written using the low-level facilities will not be portable until the Scheme community has gained enough experience with these low-level macro systems to design and adopt one low-level system as an informal standard.
References


A Syntactic Closures Macro Facility

by Chris Hanson

9 November 1991

This document describes syntactic closures, a low-level macro facility for the Scheme programming language. The facility is an alternative to the low-level macro facility described in the Revised~4 Report on Scheme. This document is an addendum to that report.

The syntactic closures facility extends the BNF rule for \(<\text{transformer spec}\>) to allow a new keyword that introduces a low-level macro transformer:

\[<\text{transformer spec}\> \mapsto (\text{transformer} <\text{expression}\>)\]

Additionally the following procedures are added:

\begin{verbatim}
make-syntactic-closure
capture-syntactic-environment
identifier?
identifier=?
\end{verbatim}

The description of the facility is divided into three parts. The first part defines basic terminology. The second part describes how macro transformers are defined. The third part describes the use of identifiers, which extend the syntactic closure mechanism to be compatible with \texttt{syntax-rules}.

Terminology

This section defines the concepts and data types used by the syntactic closures facility.

- \textit{Forms} are the syntactic entities out of which programs are recursively constructed. A form is any expression, any definition, any syntactic keyword, or any syntactic closure. The variable name that appears in a \texttt{set!} special form is also a form. Examples of forms:
17
#t
car
(+ x 4)
(lamba (x) x)
(define pi 3.14159)
if
define

• An alias is an alternate name for a given symbol. It can appear anywhere in a form that the symbol could be used, and when quoted it is replaced by the symbol; however, it does not satisfy the predicate symbol?. Macro transformers rarely distinguish symbols from aliases, referring to both as identifiers.

• A syntactic environment maps identifiers to their meanings. More precisely, it determines whether an identifier is a syntactic keyword or a variable. If it is a keyword, the meaning is an interpretation for the form in which that keyword appears. If it is a variable, the meaning identifies which binding of that variable is referenced. In short, syntactic environments contain all of the contextual information necessary for interpreting the meaning of a particular form.

• A syntactic closure consists of a form, a syntactic environment, and a list of identifiers. All identifiers in the form take their meaning from the syntactic environment, except those in the given list. The identifiers in the list are to have their meanings determined later.

A syntactic closure may be used in any context in which its form could have been used. Since a syntactic closure is also a form, it may not be used in contexts where a form would be illegal. For example, a form may not appear as a clause in the cond special form.

A syntactic closure appearing in a quoted structure is replaced by its form.

Transformer Definition

This section describes the transformer special form and the procedures make-syntactic-closure and capture-syntactic-environment.

transformer expression

Syntax: It is an error if this syntax occurs except as a <transformer spec>.

Semantics: The expression is evaluated in the standard transformer environment to yield a macro transformer as described below. This macro transformer is bound to a macro keyword by the special form in which the transformer expression appears (for example, let-syntax).

A macro transformer is a procedure that takes two arguments, a form and a syntactic environment, and returns a new form. The first argument, the input form, is the form in which the macro keyword occurred. The second argument, the usage environment,
is the syntactic environment in which the input form occurred. The result of the transformer, the output form, is automatically closed in the transformer environment, which is the syntactic environment in which the transformer expression occurred.

For example, here is a definition of a push macro using syntax-rules:

```
(define-syntax push
 (syntax-rules ()
  ((push item list)
   (set! list (cons item list)))))
```

Here is an equivalent definition using transformer:

```
(define-syntax push
 (transformer
  (lambda (exp env)
   (let ((item
            (make-syntactic-closure env '() (cadr exp)))
     (list
      (make-syntactic-closure env '() (caddr exp)))
    ' (set! ,list (cons ,item ,list))))))
```

In this example, the identifiers set! and cons are closed in the transformer environment, and thus will not be affected by the meanings of those identifiers in the usage environment env.

Some macros may be non-hygienic by design. For example, the following defines a loop macro that implicitly binds exit to an escape procedure. The binding of exit is intended to capture free references to exit in the body of the loop, so exit must be left free when the body is closed:

```
(define-syntax loop
 (transformer
  (lambda (exp env)
   (let ((body (cdr exp)))
    '(call-with-current-continuation
     (lambda (exit)
      (let f ()
       ,@ (map (lambda (exp)
                 (make-syntactic-closure env ' (exit) exp))
              body)
      (f))))))
```

To assign meanings to the identifiers in a form, use `make-syntactic-closure` to close the form in a syntactic environment.

`make-syntactic-closure` environment free-names form

Procedure

`Environment` must be a syntactic environment, `free-names` must be a list of identifiers, and `form` must be a form. `make-syntactic-closure` constructs and returns a syntactic closure of `form` in `environment`, which can be used anywhere that `form` could have been used. All the identifiers used in `form`, except those explicitly excepted by `free-names`, obtain their meanings from `environment`.

Here is an example where `free-names` is something other than the empty list. It is instructive to compare the use of `free-names` in this example with its use in the `loop` example above: the examples are similar except for the source of the identifier being left free.

```scheme
(define-syntax let1
  (transformer
   (lambda (exp env)
    (let ((id (cadr exp))
          (init (caddr exp))
          (exp (cadddr exp)))
     '((lambda (,id)
         ,(make-syntactic-closure env (list id) exp))
       ,(make-syntactic-closure env '() init))))))
```

`let1` is a simplified version of `let` that only binds a single identifier, and whose body consists of a single expression. When the body expression is syntactically closed in its original syntactic environment, the identifier that is to be bound by `let1` must be left free, so that it can be properly captured by the `lambda` in the output form.

To obtain a syntactic environment other than the usage environment, use `capture-syntactic-environment`.

`capture-syntactic-environment` procedure

Procedure

`capture-syntactic-environment` returns a form that will, when transformed, call `procedure` on the current syntactic environment. `Procedure` should compute and return a new form to be transformed, in that same syntactic environment, in place of the form.

An example will make this clear. Suppose we wanted to define a simple `loop-until` keyword equivalent to
(define-syntax loop-until
  (syntax-rules ()
    ((loop-until id init test return step)
     (letrec ((loop
                (lambda (id)
                  (if test return (loop step))))
             (loop init))))))

The following attempt at defining loop-until has a subtle bug:

(define-syntax loop-until
  (transformer
   (lambda (exp env)
     (let ((id (cadr exp))
           (init (caddr exp))
           (test (cadddr exp))
           (return (cadddr (cdr exp)))
           (step (cadddr (cddr exp)))
           (close
            (lambda (exp free)
               (make-syntactic-closure env free exp)))
           (letrec ((loop
                      (lambda (,id)
                        (if ,test return ,list id)
                        ,(close return ,list id)
                        ,(loop ,step (list id))))))
          (loop ,(close init '())))))

This definition appears to take all of the proper precautions to prevent unintended captures. It carefully closes the subexpressions in their original syntactic environment and it leaves the id identifier free in the test, return, and step expressions, so that it will be captured by the binding introduced by the lambda expression. Unfortunately it uses the identifiers if and loop within that lambda expression, so if the user of loop-until just happens to use, say, if for the identifier, it will be inadvertently captured.

The syntactic environment that if and loop want to be exposed to is the one just outside the lambda expression: before the user's identifier is added to the syntactic environment, but after the identifier loop has been added. capture-syntactic-environment captures exactly that environment as follows:
(define-syntax loop-until
  (transformer
    (lambda (exp env)
      (let ((id (cadr exp))
          (init (caddr exp))
          (test (cadddr exp))
          (return (cadddr (cdr exp)))
          (step (cadddr (cdar exp)))
          (close
            (lambda (exp free)
              (make-syntactic-closure env free exp))))
        '(letrec ((loop
                    ,,(capture-syntactic-environment
                      (lambda (env)
                        '(lambda (,id)
                          ,,(make-syntactic-closure env '() 'if)
                          ,,(close test (list id))
                          ,,(close return (list id))
                          ,,(make-syntactic-closure env '() 'loop)
                          ,,(close step (list id)))))
                    (loop ,,(close init '()))))))
    (lambda (exp env)
      (lambda (transformer-env
                   ...)))))

In this case, having captured the desired syntactic environment, it is convenient to construct syntactic closures of the identifiers if and the loop and use them in the body of the lambda.

A common use of capture-syntactic-environment is to get the transformer environment of a macro transformer:

(definition (transformer
                (lambda (exp env)
                  (capture-syntactic-environment
                    (lambda (transformer-env
                                 ...)))))

Identifiers

This section describes the procedures that create and manipulate identifiers. Previous syntactic closure proposals did not have an identifier data type—they just used symbols. The identifier data type extends the syntactic closures facility to be compatible with the high-level syntax-rules facility.

As discussed earlier, an identifier is either a symbol or an alias. An alias is implemented as a syntactic closure whose form is an identifier:
(make-syntactic-closure env '() 'a) ⇒ an alias

Aliases are implemented as syntactic closures because they behave just like syntactic closures most of the time. The difference is that an alias may be bound to a new value (for example by lambda or let-syntax); other syntactic closures may not be used this way. If an alias is bound, then within the scope of that binding it is looked up in the syntactic environment just like any other identifier.

Aliases are used in the implementation of the high-level facility syntax-rules. A macro transformer created by syntax-rules uses a template to generate its output form, substituting subforms of the input form into the template. In a syntactic closures implementation, all of the symbols in the template are replaced by aliases closed in the transformer environment, while the output form itself is closed in the usage environment. This guarantees that the macro transformation is hygienic, without requiring the transformer to know the syntactic roles of the substituted input subforms.

identifier? object

   procedure
   Returns #t if object is an identifier, otherwise returns #f. Examples:

   (identifier? 'a) ⇒ #t
   (identifier? (make-syntactic-closure env '() 'a)) ⇒ #t

   (identifier? "a") ⇒ #f
   (identifier? #\a) ⇒ #f
   (identifier? #?b) ⇒ #f
   (identifier? #f) ⇒ #f
   (identifier? 'a)) ⇒ #f
   (identifier? '#(a)) ⇒ #f

The predicate eq? is used to determine if two identifiers are “the same”. Thus eq? can be used to compare identifiers exactly as it would be used to compare symbols. Often, though, it is useful to know whether two identifiers “mean the same thing”. For example, the cond macro uses the symbol else to identify the final clause in the conditional. A macro transformer for cond cannot just look for the symbol else, because the cond form might be the output of another macro transformer that replaced the symbol else with an alias. Instead the transformer must look for an identifier that “means the same thing” in the usage environment as the symbol else means in the transformer environment.

identifier=? environment1 identifier1 environment2 identifier2

   procedure

   Environment1 and environment2 must be syntactic environments, and identifier1 and identifier2 must be identifiers. identifier=? returns #t if the meaning of identifier1...
in environment1 is the same as that of identifier2 in environment2, otherwise it returns #f. Examples:

\[
\text{let-syntax} \\
\text{((foo)} \\
\text{ (transformer)} \\
\text{ (lambda (form env))} \\
\text{ (capture-syntactic-environment)} \\
\text{ (lambda (transformer-env)} \\
\text{ (identifier=? transformer-env 'x env 'x)))})}) \\
\text{(list (foo)} \\
\text{ (let ((x 3)} \\
\text{ (foo))})} \\
\Rightarrow (#t #f)
\]

\[
\text{let-syntax ((bar foo))} \\
\text{let-syntax} \\
\text{((foo)} \\
\text{ (transformer)} \\
\text{ (lambda (form env))} \\
\text{ (capture-syntactic-environment)} \\
\text{ (lambda (transformer-env)} \\
\text{ (identifier=? transformer-env 'foo env (cadr form)))})}) \\
\text{(list (foo foo)} \\
\text{ (foo bar))})} \\
\Rightarrow (#f #t)
\]

Acknowledgments

The syntactic closures facility was invented by Alan Bawden and Jonathan Rees. The use of aliases to implement syntax-rules was invented by Alan Bawden (who prefers to call them "synthetic names"). Much of this proposal is derived from an earlier proposal by Alan Bawden.
Hygienic Macros Through Explicit Renaming

William Clinger

This paper describes an alternative to the low-level macro facility described in the Revised\(^4\) Report on the Algorithmic Language Scheme [1]. The facility described here is based on explicit renaming of identifiers, and was developed for the first implementation of the hygienic macro expansion algorithm described in [2]. It was the first low-level macro facility to be designed for compatibility with a high-level hygienic macro system, and it remains one of the easiest to understand.

Whereas the low-level macro facility described in the Revised\(^4\) Report renames identifiers automatically, so that hygienic macros are obtained by default, the facility described here requires that identifiers be renamed explicitly in order to maintain hygiene.

Another difference is that, as originally implemented and as described here, there is no way to define certain hygienic macros that define other hygienic macros. The problem is that the transformation procedure for the defined macro may need to compare pieces of its first argument with the denotation of an identifier, but the only way for the defining macro to pass that denotation along to the defined macro is as part of the code for the defined macro. This problem can be solved by introducing \texttt{syntax} expressions as in the Revised\(^4\) Report. \texttt{Syntax} is like \texttt{quote}, except that the denotation of an identifier quoted by \texttt{syntax} is preserved as part of the quoted value.

As with the low-level macro facility based on syntactic closures [3], the explicit renaming facility adds a new production for (\texttt{transformer spec}):

\begin{verbatim}
(transformer) \rightarrow (transformer (expression))
\end{verbatim}

The (expression) is expanded in the syntactic environment of the \texttt{transformer} expression, and the expanded expression is evaluated in the standard transformer environment to yield a \textit{transformation procedure}. The transformation procedure takes an expression and two other arguments and returns a transformed expression. For example, the transformation procedure for a \texttt{call} macro such that (\texttt{call proc ary ...}) expands into (\texttt{proc ary ...}) can be written as

\begin{verbatim}
(lambda (exp rename compare)
    (transformer (expression))
)
\end{verbatim}
(cdr exp))

Expressions are represented as lists in the traditional manner, except that identifiers may be represented by objects other than symbols. Transformation procedures may use the predicate identifier? to determine whether an object is the representation of an identifier.

The second argument to a transformation procedure is a renaming procedure that takes the representation of an identifier as its argument and returns the representation of a fresh identifier that occurs nowhere else in the program. For example, the transformation procedure for a simplified version of the let macro might be written as

(lambda (exp rename compare)
  (let ((vars (map car (cdr exp)))
       (inits (map cdr (cdr exp)))
       (body (cddr exp)))
   '(((lambda ,vars ,@body)
      ,@inits))))

This would not be hygienic, however. A hygienic let macro must rename the identifier lambda to protect it from being captured by a local binding. The renaming effectively creates a fresh alias for lambda, one that cannot be captured by any subsequent binding:

(lambda (exp rename compare)
  (let ((vars (map car (cdr exp)))
       (inits (map cdr (cdr exp)))
       (body (cddr exp)))
   '(((rename 'lambda) ,vars ,@body)
      ,@inits)))

The expression returned by the transformation procedure will be expanded in the syntactic environment obtained from the syntactic environment of the macro application by binding any fresh identifiers generated by the renaming procedure to the denotations of the original identifiers in the syntactic environment in which the macro was defined. This means that a renamed identifier will denote the same thing as the original identifier unless the transformation procedure that renamed the identifier placed an occurrence of it in a binding position.

The renaming procedure acts as a mathematical function in the sense that the identifiers obtained from any two calls with the same argument will be the same in the sense of eqv?. It is an error if the renaming procedure is called after the transformation procedure has returned.

The third argument to a transformation procedure is a comparison predicate that takes the representations of two identifiers as its arguments and returns true if and only if they denote the same thing in the syntactic environment that will be used to expand the transformed macro application. For example,
the transformation procedure for a simplified version of the cond macro can be written as

\[
\text{(lambda (exp rename compare)}
\text{ (let ((clauses (cdr exp)))}
\text{ (if (null? clauses)}
\text{ '(', (rename 'quote) unspecified)}
\text{ (let* ((first (car clauses))}
\text{ (rest (cdr clauses))}
\text{ (test (car first)))}
\text{ (cond ((and (identifier? test)}
\text{ (compare test (rename 'else)))}
\text{ '(', (rename 'begin) ,@ (cdr first))))}
\text{ (else '(', (rename 'if)}
\text{ ,test}
\text{ '(', (rename 'begin) ,@ (cdr first)))}
\text{ (cond ,@ (rest)))))})
\]

In this example the identifier else is renamed before being passed to the comparison predicate, so the comparison will be true if and only if the test expression is an identifier that denotes the same thing in the syntactic environment of the expression being transformed as else denotes in the syntactic environment in which the cond macro was defined. If else were not renamed before being passed to the comparison predicate, then it would match a local variable that happened to be named else, and the macro would not be hygienic.

Some macros are non-hygienic by design. For example, the following defines a loop macro that implicitly binds exit to an escape procedure. The binding of exit is intended to capture free references to exit in the body of the loop, so exit is not renamed.

\[
\text{(define-syntax loop)}
\text{ (transformer}
\text{ (lambda (x r c)}
\text{ (let ((body (cdr x)))}
\text{ '(', (r 'call-with-current-continuation)}
\text{ '(', (r 'lambda) (exit)}
\text{ '(', (r 'let) ,r 'f) () ,@ (body ,r 'f))))))}
\]

Suppose a while macro is implemented using loop, with the intent that exit may be used to escape from the while loop. The while macro cannot be written as

\[
\text{(define-syntax while)}
\text{ (syntax-rules ()}
\text{ ((while test body ...)}
\text{ (loop (if (not test) (exit #f))}
\text{ body ...)))}
\]

3
because the reference to exit that is inserted by the while macro is intended to be captured by the binding of exit that will be inserted by the loop macro. In other words, this while macro is not hygienic. Like loop, it must be written using the transformer syntax:

(define-syntax while
  (transformer
   (lambda (x r c)
     (let ((test (cadr x))
           (body (cddr x)))
       '(', (r 'loop)
       ,r 'if) ,r 'not) ,test (exit #f)
       ,@body))))

References

