MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

AI Memo No. 848 August 1985

The Revised Revised Report on Scheme
or
An UnCommon Lisp

by

Hal Abelson
Norman Adams
David Bartley
Gary Brooks
William Clinger [editor]
Dan Friedman
Robert Halstead
Chris Hanson
Chris Haynes
Eugene Kohlbecker
Don Oxley
Kent Pitman
Jonathan Rees
Bill Rozas
Gerald Jay Sussman
Mitchell Wand

Keywords: SCHEME, LISP, functional programming, computer languages.

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.

This report describes researach done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology and the Computer Science Department of Indiana University. Support for the MIT research is 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 research is provided by NSF Grants NCS 83-04567 and NCS 83-03325. This report is published at Indiana University as Computer Science Department Technical Report 174, June 1985.

Table of Contents

Acknowledgements

This report is primarily the work of a group of people who met at Brandeis University for two days in October 1984. Participating in that workshop were Hal Abelson, Norman Adams, David Bartley, Gary Brooks, William Clinger, Dan Friedman, Robert Halstead, Chris Hanson, Chris Haynes, Eugene Kohlbecker, Don Oxley, Jonathan Rees, Bill Rozas, Gerald Sussman, and Mitchell Wand. Kent Pitman made valuable contributions to the agenda for the workshop but was unable to attend the sessions.

We would like to thank the following people for their comments and criticisms in the months following the workshop: George Carrette, Kent Dybvig, Andy Freeman, Yekta Gursel, Paul Hudak, Chris Lindblad, John Ramsdell, and Guy Steele Jr.

We thank Carol Fessenden, Dan Friedman, and Chris Haynes for permission to use text from the Scheme 311 Version 4 reference manual. We thank Gerry Sussman for drafting the chapter on numbers, Chris Hanson for drafting the chapters on characters and strings, and Gary Brooks and William Clinger for drafting the chapters on input and output. We gladly acknowledge the influence of manuals for MIT Scheme, T, Scheme 84, and Common Lisp.

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

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.

Editor's note: This report records the unanimous decisions made through a remarkable spirit of compromise at Brandeis, together with the fruits of subsequent committee work and discussions made possible by various computer networks. I have tried to edit these into a coherent document while remaining faithful to the workshop's decisions and the community's consensus. I apologize for any cases in which I have misinterpreted the authors or misjudged the consensus.

William Clinger

Part I: Introduction to Scheme

I.0 Brief history of 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 very few different methods of expression formation.

The first description of Scheme was written in 1975 [28]. A Revised Report [24] appeared in 1978, which described the evolution of the language as its MIT implementation was upgraded to support an innovative compiler [21]. Three distinct projects began in 1981 and 1982 to use variants of Scheme for courses at MIT, Yale, and Indiana University [11, 14, 4]. An introductory computer science textbook using Scheme was published in 1984 [1].

As might be expected of a language used primarily for education and research, Scheme has always evolved rapidly. This was no problem when Scheme was used only within MIT, but as Scheme became more widespread local subdialects began to diverge until students and researchers occasionally found it difficult to understand code written at other sites. Fifteen representatives of the major implementations of Scheme therefore met in October 1984 to work toward a better and more widely accepted standard for Scheme. This paper reports their unanimous recommendations augmented by committee work in the areas of arithmetic, characters, strings, and input/output.

Scheme shares with Common Lisp [23] the goal of a core language common to several implementations. Scheme differs from Common Lisp in its emphasis upon simplicity and function over compatibility with older dialects of Lisp.

I.1 Syntax

Formal definitions of the lexical and context-free syntaxes of Scheme will be included in a separate report.

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 characters that contains no special characters and begins with a character that cannot begin a number is an identifier. There may be other identifiers as well, and in particular the following are identifiers:

```
+  -  1+  -1+
```

It is guaranteed that the following characters cannot begin a number, so identifiers other than the four listed above should begin with one of:

```
a b c d e f g h i j k l m n o p q r s t u v w x y z
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
! $ % & * / : < = > ? ~
```

Subsequent characters of the identifier should be drawn from:

```
a b c d e f g h i j k l m n o p q r s t u v w x y z
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
0 1 2 3 4 5 6 7 8 9
! $ % & * / : < = > ? ^ _ . ~
```

The case in which the letters of an identifier are typed is immaterial. For example, `Foo` is the same identifier as `FOO`.

The following characters are special, and should never be used in an identifier:

```
) ( ] [ } { " ; blank
```

Scheme deliberately does not specify whether the following characters can be used in identifiers:

```
# ' ` , @ \ |
```

Rationale: Some implementations might want to use backslash (\) and vertical bar (|) as in Common Lisp. As for the others there are two schools of thought. One school argues that disallowing special characters in identifiers allows the computer to catch more typing errors. The other school agrees only for special characters that come in pairs, on the grounds that errors involving only the unpaired special characters are easier to see.

Numbers

For a description of the notations used for numbers, see section II.6.

Comments

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.

Other notations

Left and right parentheses are used for grouping and to notate lists as described in section II.4. Left and right square brackets and curly braces are not used in Scheme right now but are reserved for unspecified future uses.

The quote (`'`) and backquote (`` ` ``) characters are used to indicate constant or almost-constant data as described in section II.1. The comma is used together with the backquote, and the atsign (`@`) is used together with the comma.

The doublequote character is used to notate strings as described in section II.8.

The sharp sign (#) is used for a variety of purposes depending on the character that follows it. A sharp sign followed by a left parenthesis signals the beginning of a vector, as described in section II.9. A sharp sign followed by an exclamation point is used to notate one of the special values `#!true`, `#!false`, and `#!null`. A sharp sign followed by a backslash is used to notate characters as described in section II.7. A sharp sign followed by any of a number of letters is used in the notation for numbers as described in section II.6.

Context free grammar for Scheme

The following grammar is ambiguous because a `<special form>` looks like a `<procedure call>`. Some implementations resolve the ambiguity by reserving the identifiers that serve as keywords of special forms, while other implementations allow the keyword meaning of an identifier to be shadowed by lexical bindings.

```
<expression> ::= <constant> | <identifier> |
                 <special form> | <procedure call>
<constant> ::= <numeral> | <string> |
               (quote <datum>) | '<datum> |
               #!true | #!false | #!null
<special form> ::= (<keyword> <syntactic component> ...)
<procedure call> ::= (<operator> <operands>)
<operator> ::= <expression>
<operands> ::= <empty> | <expression> <operands>
```

<datum> stands for any written representation of a Scheme object, as described in the sections that follow. <identifier> has already been described informally. <numeral> is described in section II.6, and <string> is described in section II.8. <special form> stands for one of the special forms whose syntax is described in section II.1. For uniformity the other kinds of expressions are also described in that section as though they were special forms.

I.2 Semantics

A formal definition of the semantics of Scheme will be included in a separate report. The detailed informal semantics of Scheme is the subject of Part II. This section gives a quick review of Scheme's major characteristics.

Scheme is a statically scoped programming language. Each use of an identifier is associated with a lexically apparent binding of that identifier. In this respect Scheme is like Algol 60, Pascal, and C but unlike dynamically scoped languages such as APL and traditional Lisp.

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 all procedures and variables, 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 process in constant space, even if the iterative process 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.

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. Lazy ML passes arguments by name, so that an argument expression is evaluated only if its value is needed by the procedure.

Part II: A catalog of Scheme

II.0 Notational conventions

This part of the report is a catalog of the special forms and procedures that make up Scheme. The special forms are described in section II.1, and the procedures are described in the following sections. Each section is organized into entries, with one entry (usually) for each special form or procedure. Each entry begins with a header line that includes the name of the special form or procedure in **boldface** type within a template for the special form or a call to the procedure. The names of the arguments to a procedure are *italicized*, as are the syntactic components of a special form. A notation such as

expr ...

indicates zero or more occurrences of *expr*. Thus

expr1 expr2 ...

indicates at least one *expr*. At the right of the header line one of the following categories will appear:

special form
constant
variable
procedure
essential special form
essential constant
essential variable
essential procedure

A special form is a syntactic class of expressions, usually identified by a keyword. A constant is something that is lexically recognizable as a constant. A variable is a location in which values (also called objects) can be stored. An identifier may be bound to a variable. Those variables that initially hold procedure values are identified as procedures.

It is guaranteed that every implementation of Scheme will support the essential special forms, constants, variables, and procedures. Implementations are free to omit other features of Scheme or to add extensions, provided the extensions are not in conflict with the language reported here.

Any Scheme value can be used as a boolean expression for the purpose of a conditional test. As explained in section II.2, most values count as true, but a few—notably `#!false`—count as false. This manual uses the word "true"

to refer to any Scheme value that counts as true in a conditional expression, and the word "false" to refer to any Scheme value that counts as false.

When speaking of an error condition, this manual uses the phrase "an error is signalled" to indicate that implementations must detect and report the error. If the magic word "signalled" 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 condition 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 manual. Implementations may extend a procedure's domain of definition to include other arguments.

II.1. Special forms

Identifiers have two uses within Scheme programs. When an identifier appears within a quoted constant (see `quote`), it is being used as data as described in the section on symbols. Otherwise it is being used as a name. There are two kinds of things that an identifier can name in Scheme: *special forms* and *variables*. A special form is a syntactic class of expressions, and an identifier that names a special form is called the *keyword* of that special form. A variable, on the other hand, is a location where a value can be stored. An identifier that names a variable is said to be *bound* to that location. The set of all such bindings in effect at some point in a program is known as the *environment* in effect at that point.

Certain special forms are used to allocate storage for new variables and to bind identifiers to those new variables. The most fundamental of these *binding constructs* is the `lambda` special form, because all other binding constructs can be explained in terms of lambda expressions. The other binding constructs are the `let`, `let*`, `letrec`, internal definition (see `define`), `rec`, `named-lambda`, and `do` special forms.

Like Algol or Pascal, and unlike most other dialects of Lisp except for Common Lisp, Scheme is a statically scoped language with block structure. To each place where an identifier is bound in a program there corresponds a *region* of the program within which the binding is effective. The region varies according to the binding construct that establishes the binding; if the binding is established by a lambda expression, for example, then the region is the entire lambda expression. Every use of an identifier in a variable reference or assignment refers to the binding of the identifier that established the innermost of the regions containing the use. If there is no binding of the identifier whose region contains the use, then the use refers to the binding for the identifier that was in effect when Scheme started up, if any; if there is no binding for the identifier, it is said to be *unbound*.

variable essential special form

An expression consisting of an identifier that is not the keyword of a special form is a variable reference. The value obtained for the variable reference is the value stored in the location to which *variable* is bound. It is an error to reference an unbound *variable*.

(*operator operand1 ...*) essential special form

A list whose first element is not the keyword of a special form indicates a procedure call. The operator and operand expressions are evaluated and the resulting procedure is passed the resulting arguments. In contrast to other dialects of Lisp the order of evaluation is not specified, and the operator expression and the operand expressions are always evaluated with the same evaluation rules.

```
(+ 3 4)                  -->  7
((if #!false + *) 3 4)   -->  12
```

(**quote** *datum*) essential special form

'datum essential special form

Evaluates to *datum*. 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)
```

Numeric constants, string constants, character constants, vector constants, and the constants `#!true`, `#!false`, and `#!null` need not be quoted.

```
'"abc"                   -->  "abc"
"abc"                    -->  "abc"
'145932                  -->  145932
145932                   -->  145932
'#!true                  -->  #!true
#!true                   -->  #!true
```

(**lambda** (*var1 ...*) *expr*) essential special form

Each *var* must be an identifier. The lambda expression evaluates to a procedure with formal argument list *(var1 ...)* and procedure body *expr*. 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 identifiers in the formal argument list to fresh locations, the corresponding actual argument values will be stored in those locations, and *expr* will then be evaluated in the extended environment. The result of *expr* will be returned as the result of the procedure call.

```
(lambda (x) (+ x x))                --> #<PROCEDURE>
((lambda (x) (+ x x)) 4)            --> 8
(define reverse-subtract
  (lambda (x y) (- y x)))           --> unspecified
(reverse-subtract 7 10)             --> 3
(define foo
  (let ((x 4))
    (lambda (y) (+ x y))))          --> unspecified
(foo 6)                             --> 10
```

(lambda (*var1* ...) *expr1 expr2* ...**)** essential special form

Equivalent to `(lambda (`*var1* `...) (begin` *expr1 expr2* `...))`.

(lambda *var expr1 expr2* ...**)** essential special form

Returns a procedure that when later called with some arguments will bind *var* to a fresh location, convert the sequence of actual arguments into a list, and store that list in the binding of *var*.

```
((lambda x x) 3 4 5 6)              --> (3 4 5 6)
```

One last variation on the formal argument list provides for a so-called "rest" argument. If a space/dot/space sequence precedes the last argument in the formal argument list, then the value stored in the binding of the last formal argument will be a list of the actual arguments left over after all the other actual arguments have been matched up against the formal arguments.

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

(if *condition consequent alternative***)** essential special form

(if *condition consequent***)** special form

First evaluates *condition*. If it yields a true value (see section II.2), then *consequent* is evaluated and its value is returned. Otherwise *alternative* is evaluated and its value is returned. If no *alternative* is specified, then the if expression is evaluated only for its effect, and the result of the expression is unspecified.

```
(if (>? 3 2) 'yes 'no)              --> yes
(if (>? 2 3) 'yes 'no)              --> no
(if (>? 3 2) (- 3 2) (+ 3 2))       --> 1
```

(**cond** *clause1 clause2* ...) essential special form

Each *clause* must be a list of one or more expressions. The first expression in each *clause* is a boolean expression that serves as the *guard* for the *clause*. The *guard*s are evaluated in order until one of them evaluates to a true value (see section II.2). When a *guard* evaluates true, then the remaining expressions in its *clause* are evaluated in order, and the result of the last expression in the selected *clause* is returned as the result of the entire expression. If the selected *clause* contains only the *guard*, then the value of the *guard* is returned as the result. If all *guard*s evaluate to false values, then the result of the conditional expression is unspecified.

```
(cond ((>? 3 2) 'greater)
      ((<? 3 2) 'less))          --> greater
```

The keyword or variable `else` may be used as a *guard* to obtain the effect of a guard that always evaluates true.

```
(cond ((>? 3 3) 'greater)
      ((<? 3 3) 'less)
      (else 'equal))             -->  equal
```

The above forms for the *clause*s are essential. Some implementations support yet another form of *clause* such that

```
(cond (form1 => form2) ...)
```

is equivalent to

```
(let ((form1-result form1)
      (thunk2 (lambda () form2))
      (thunk3 (lambda () (cond ...))))
  (if form1-result
      ((thunk2) form1-result)
      (thunk3)))
```

(**case** *expr clause1 clause2* ...) special form

Each *clause* is a list whose first element is a *selector* followed by one or more expressions. Each *selector* should be a list of values. The *selector*s are not evaluated. Instead *expr* is evaluated and its result is compared against successive *selector*s using the `memv` procedure until a match is found. Then the expressions in the selected *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 no *selector* matches then 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
```

The special keyword `else` may be used as a *selector* to obtain the effect of a selector that always matches.

```
(case (car '(c d))
  ((a e i o u) 'vowel)
  ((y) 'y)
  (else 'consonant))             -->  consonant
```

(**and** *expr1* ...) special form

Evaluates the *expr*s from left to right, returning false as soon as one evaluates to a false value (see section II.2). Any remaining expressions are not evaluated. If all the expressions evaluate to true values, the value of the last expression is returned.

```
(and (=? 2 2) (>? 2 1))          -->  #!true
(and (=? 2 2) (<? 2 1))          -->  #!false
(and 1 2 'c '(f g))              -->  (f g)
```

(**or** *expr1* ...) special form

Evaluates the *expr*s from left to right, returning the value of the first *expr* that evaluates to a true value (see section II.2). Any remaining expressions are not evaluated. If all expressions evaluate to false values, false is returned.

```
(or (=? 2 2) (>? 2 1))           -->  #!true
(or (=? 2 2) (<? 2 1))           -->  #!true
(or #!false #!false #!false)     -->  #!false
(or (memq 'b '(a b c)) (/ 3 0)) -->  (b c)
```

(**let** ((*var1 form1*) ...) *expr1 expr2* ...) essential special form

Evaluates the *form*s in the current environment (in some unspecified order), binds the *var*s to fresh locations holding the results, and then evaluates the *expr*s in the extended environment from left to right, returning the value of the last one. Each binding of a *var* has *expr1 expr2* ... as its region.

```
(let ((x 2) (y 3))
  (* x y))                       -->  6
```

```
(let ((x 2) (y 3))
  (let ((foo (lambda (z) (+ x y z)))
        (x 7))
    (foo 4)))                          --> 9
```

`let` and `letrec` give Scheme a block structure. The difference between `let` and `letrec` is that in a `let` the *forms* are not within the region of the *vars* being bound. See `letrec`.

Some implementations of Scheme permit a "named let" syntax in which

(let *name* **((***var1 form1***) ...)** *expr1 expr2* **...)**

is equivalent to

((rec *name* **(lambda (***var1* **...)** *expr1 expr2* **...))** *form1* **...)**

(let* ((*var1 form1***) ...)** *expr1 expr2* **...)** special form

Similar to *let*, but the bindings are performed sequentially from left to right and the region of a binding indicated by (*var form*) is that part of the `let*` expression to the right of the binding. Thus the second binding is done in an environment in which the first binding is visible, and so on.

(letrec ((*var1 form1***) ...)***expr1 expr2* **...)** essential special form

Binds the *vars* to fresh locations holding undefined values, evaluates the *forms* in the resulting environment (in some unspecified order), assigns to each *var* the result of the corresponding *form*, evaluates the *exprs* sequentially in the resulting environment, and returns the value of the last *expr*. Each binding of a *var* has the entire letrec expression as its region, making it possible to define mutually recursive procedures. See `let`.

```
(letrec ((x 2) (y 3))
  (letrec ((foo (lambda (z) (+ x y z))) (x 7))
    (foo 4)))                          --> 14
```

```
(letrec  ((even?
           (lambda (n)
             (if (zero? n)
                 #!true
                 (odd? (-1+ n)))))
          (odd?
           (lambda (n)
             (if (zero? n)
                 #!false
                 (even? (-1+ n))))))
  (even? 88))
                                        -->  #!true
```

One restriction on `letrec` is very important: it must be possible to evaluate each *form* without referring to the value of a *var*. If this restriction is violated, then the effect is undefined, and an error may be reported during evaluation of the *forms*. The restriction is necessary because Scheme passes arguments by value rather than by name. In the most common uses of letrec, all the *forms* are lambda expressions and the restriction is satisfied automatically.

(rec *var expr***)** special form

Equivalent to `(letrec ((`*var expr*`)) `*var*`)`. `rec` is useful for defining self-recursive procedures.

(named-lambda (*name var1 ...*) *expr ...*) special form

Equivalent to **(rec** *name* `(lambda (`*var1 ...*`) `*expr ...*`))`

Rationale: Some implementatations may find it easier to provide good debugging information when `named-lambda` is used instead of `rec`.

(define *var expr*) essential special form

When typed at top level, so that it is not nested within any other expression, this *form* has essentially the same effect as the assignment `(set!` *var expr)* if *var* is bound. If *var* is not bound, however, then the `define` form will bind *var* before performing the assignment, whereas it would be an error to perform a `set!` on an unbound identifier. The value returned by a `define` form is not specified.

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

The semantics just described is essential. Some implementations also allow `define` expressions to appear at the beginning of the body of a `lambda`, `named-lambda`, `let`, `let*`, or `letrec` expression. Such expressions 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 of the `lambda`, `named-lambda`, `let`, `let*`, or `letrec` expression. That is, *var* is bound rather than assigned, and the region set up by the binding is the entire body of the `lambda`, `named-lambda`, `let`, `let*`, or `letrec` expression. For example,

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

Internal definitions can always be converted into an equivalent `letrec` expression. For example, the `let` expression in the above example is equivalent to

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

(define (*var0 var1 ...*) *expr1 expr2 ...*) special form
(define (*form var1 ...*) *expr1 expr2 ...*) special form

The first syntax, where *var0* is an identifier, is equivalent to

(`define` *var0* (`rec` *var0* (`lambda` (*var1 ...*) *expr1 expr2*)))

The second syntax, where *form* is a list, is sometimes convenient for defining a procedure that returns another procedure as its result. It is equivalent to

(`define` *form* (`lambda` (*var1 ...*) *expr1 expr2 ...*)).

(set! *var expr*) essential special form

Stores the value of *expr* in the location to which *var* is bound. *expr* is evaluated but *var* is not. The result of the `set!` expression is unspecified.

```
(set! x 4)          -->  unspecified
(1+ x)              -->  5
```

(**begin** *expr1 expr2 ...*) essential special form

Evaluates the *expr*s sequentially from left to right and returns the value of the last *expr*. Used to sequence side effects such as input and output.

```
(begin (set! x 5)
       (1+ x))                    -->  6
```

Also

```
(begin (display "4 plus 1 equals ")
       (display (1+ 4)))
```

prints `4 plus 1 equals 5`

A number of special forms such as `lambda` and `letrec` implicitly treat their bodies as `begin` expressions.

(**sequence** *expr1 expr2 ...*) special form

`sequence` is synonymous with `begin`.

Rationale: `sequence` was used in the Abelson and Sussman text, but it should not be used in new code.

(**do** *varspecs exit stmt1 ...*) special form

The `do` special form is an extremely general albeit complex iteration macro. The *varspecs* specify variables to be bound, how they are to be initialized at the start, and how they are to be incremented every on every iteration. the general form looks like:

```
(do (( var1  init1  step1) ...)
    ( test  expr1 ...)
     stmt1 ...)
```

Each *var* must be an identifier and each *init* and *step* must be expressions. The *init* expressions are evaluated (in some unspecified order), the *var*s are bound to fresh locations, the results of the *init* expressions are stored in the bindings of the *var*s, and then the iteration phase begins.

Each iteration begins by evaluating *test*; if the result is false (see section II.2), then the *stmt*s are evaluated in order for effect, the *step*s are evaluated (in some unspecified order), the results of the *step* expressions are stored in the bindings of the *var*s, and the next iteration begins.

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

The region set up by the binding of a *var* consists of the entire do expression except for the *inits*.

A *step* may be omitted, in which case the corresponding *var* is not updated. When the *step* is omitted the *init* may be omitted as well, in which case the initial value is not specified.

```
(do ((vec (make-vector 5))
     (i 0 (1+ i)))
    ((=? i 5) vec)
    (vector-set! vec i i))          -->  #(0 1 2 3 4)
(let ((x '(1 3 5 7 9)))
  (do ((x x (cdr x))
       (sum 0 (+ sum (car x))))
      ((null? x) sum)))             -->  25
```

The do special form is essentially the same as the do macro in Common Lisp. The main difference is that in Scheme the identifier `return` is not bound; programmers that want to bind `return` as in Common Lisp must do so explicitly (see `call-with-current-continuation`).

`'`*pattern* special form

The backquote special form is useful for constructing a list structure when most but not all of the desired structure is known in advance. If no commas appear within the *pattern*, the result of evaluating `'`*pattern* is equivalent (in the sense of `equal?`) to the result of evaluating *'pattern*. If a comma appears within the *pattern*, however, the expression following the comma is evaluated and its result is inserted into the structure instead of the comma and the expression. If a comma appears followed immediately by an at-sign (`@`), then the following expression must evaluate to a list; the opening and closing parentheses of the list are then "stripped away" and the elements of the list are inserted in place of the comma/at-sign/expression sequence.

```
`(a ,(+ 1 2) ,@(map 1+ '(4 5 6)) b)     -->  (a 3 5 6 7 b)
`(((foo ,(- 10 3)) ,@(cdr '(c)) cons)) -->  (((foo 7) cons))
```

Scheme does not have any standard facility for defining new special forms.

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

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

II.2. Booleans

The standard boolean objects for truth and falsity are written as `#!true` and `#!false`. What really matters, though, are the objects that the Scheme conditional expressions (`if`, `cond`, `and`, `or`, `do`) will treat as though they were 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. All of the conditional expressions are equivalent in that an object treated as false by any one of them is treated as false by all of them, and likewise for true values.

Of all the standard Scheme values, only `#!false` and the empty list count as false in conditional expressions. `#!true`, pairs (and therefore lists), symbols, numbers, strings, vectors, and procedures all count as true.

The empty list counts as false for historical reasons only, and programs should not rely on this because future versions of Scheme will probably do away with this nonsense.

Programmers accustomed to other dialects of Lisp should beware that Scheme has already done away with the nonsense that identifies the empty list with the symbol `nil`.

 # !false essential constant

`#!false` is the boolean value for falsity. The `#!false` object is self-evaluating. That is, it does not need to be quoted in programs.

```
'#!false        -->  #!false
#!false         -->  #!false
```

 # !true essential constant

`#!true` is the boolean value for truth. The `#!true` object is self-evaluating, and does not need to be quoted in programs.

(not *obj)* essential procedure

Returns `#!true` if *obj* is false and returns `#!false` otherwise.

nil variable

t variable

As a crutch for programmers accustomed to other dialects of Lisp, some implementations provide variables `nil` and `t` whose initial values are `#!null`

and `#!true` respectively. These variables should not be relied upon in new code.

II.3. Equivalence predicates

A predicate is a procedure that always returns `#!true` or `#!false`. Of the equivalence predicates described in this section, `eq?` is the most discriminating while `equal?` is the most liberal. `eqv?` is very slightly less discriminating than `eq?`.

(`eq?` *obj1 obj2*) essential procedure

Returns `#!true` if *obj1* is identical in all respects to *obj2*, otherwise returns `#!false`. If there is any way at all that a user can distinguish *obj1* and *obj2*, then `eq?` will return `#!false`. On the other hand, it is guaranteed that objects maintain their identity despite being fetched from or stored into variables or data structures.

The notion of identity used by `eq?` is stronger than the notions of equivalence used by the `eqv?` and `equal?` predicates. The constants `#!true` and `#!false` are identical to themselves and are different from everything else, except that in some implementations the empty list is identical to `#!false` for historical reasons. Two symbols are identical if they print the same way (except that some implementations may have "uninterned symbols" that violate this rule). For structured objects such as pairs and vectors the notion of sameness is defined in terms of the primitive mutation procedures defined on those objects. For example, two pairs are the same if and only if a `set-car!` operation on one changes the car field of the other. The rules for identity of numbers are extremely implementation-dependent and should not be relied on.

Generally speaking, the `equal?` procedure should be used to compare lists, vectors, and arrays. The `char=?` procedure should be used to compare characters, the `string=?` procedure should be used to compare strings, and the `=?` procedure should be used to compare numbers. The `eqv?` procedure is just like `eq?` except that it can be used to compare characters and exact numbers as well. (See section II.6 for a discussion of exact numbers.)

```
(eq? 'a 'a)                          -->  #!true
(eq? 'a 'b)                          -->  #!false
(eq? '(a) '(a))                      -->   unspecified

(eq? "a" "a")                        -->   unspecified
(eq? 2 2)                            -->   unspecified
(eq? (cons 'a 'b) (cons 'a 'b))      -->  #!false
(let ((x (read)))
  (eq? (cdr (cons 'b x)) x))         -->  #!true
```

(eqv? *obj1 obj2***)** essential procedure

eqv? is just like eq? except that if *obj1* and *obj2* are exact numbers then eqv? is guaranteed to return `#!true` if *obj1* and *obj2* are equal according to the `=?` procedure.

```
(eq? 100000 100000)     -->  unspecified
(eqv? 100000 100000)    -->  #!true
```

See section II.6 for a discussion of exact numbers.

(equal? *obj1 obj2***)** essential procedure

Returns `#!true` if *obj1* and *obj2* are identical objects or if they are equivalent numbers, lists, characters, strings, or vectors. Two objects are generally considered equivalent if they print the same. equal? may fail to terminate if its arguments are circular data structures.

```
(equal? 'a 'a)                  -->  #!true
(equal? '(a) '(a))              -->  #!true
(equal? '(a (b) c) '(a (b) c))  -->  #!true
(equal? "abc" "abc")            -->  #!true
(equal? 2 2)                    -->  #!true
(equal? (make-vector 5 'a)
        (make-vector 5 'a))     -->  #!true
```

II.4. Pairs and lists

Lists are Lisp's—and therefore Scheme's—characteristic data structures.

The empty list is a special object that is written as an opening parenthesis followed by a closing parenthesis: `()` The empty list has no elements, and its length is zero. The empty list is not a pair.

Larger lists are built out of pairs. 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 named `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!`.

The most general notation used 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.

The dotted notation is not often used, because more streamlined notations exist for the common case where the cdr is the empty list or a pair. Thus (*c1* . `()`) is usually written as (*c1*), and (*c1* . (*c2* . *c3*)) is usually written as (*c1 c2* . *c3*). Usually these special notations permit a structure to be written without any dotted pair notation at all. For example

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

would normally be written as `(a b c d e)`.

When all the dots can be made to disappear as in the example above, the entire structure is called a proper list. Proper lists are so common that when people speak of a list, they usually mean a proper list. An inductive definition:

- The empty list is a proper list.
- If *plist* is a proper list, then any pair whose cdr is *plist* is also a proper list.
- There are no other proper lists.

A proper list is therefore either the empty list or a pair from which the empty list can be obtained by applying the `cdr` procedure a finite number of times. Whether a given pair is a proper list depends upon what is stored in the cdr field. When the `set-cdr!` procedure is used, an object can be a proper list

one moment and not the next:

```
(define x '(a b c))      -->  unspecified
(define y x)             -->  unspecified
y                        -->  (a b c)
(set-cdr! x 4)           -->  unspecified
x                        -->  (a . 4)
(eq? x y)                -->  #!true
y                        -->  (a . 4)
```

A pair object, on the other hand, will always be a pair object.

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

- The empty list is a list of *D*.
- If *plist* is a list of *D*, then any pair whose cdr is *plist* and whose car is an element of the data type *D* is also a list of *D*.
- There are no other lists of *D*.

(pair? *obj***)** essential procedure

Returns `#!true` if *obj* is a pair, otherwise returns `#!false`.

```
(pair? '(a . b))         -->  #!true
(pair? '(a b c))         -->  #!true
(pair? '())              -->  #!false
(pair? '#(a b))          -->  #!false
```

(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 eq?) 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*. *pair* must be a 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*. *pair* must be a 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*. *pair* must be a pair. The value returned by `set-car!` is unspecified. This procedure can be very confusing if used indiscriminately.

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

Stores *obj* in the cdr field of *pair*. *pair* must be a pair. The value returned by set-cdr! is unspecified. This procedure can be very confusing if used indiscriminately.

(**caar** *pair*) essential procedure
(**cadr** *pair*) essential procedure
(**cdar** *pair*) essential procedure
(**cddr** *pair*) essential procedure
(**caaar** *pair*) essential procedure
(**caadr** *pair*) essential procedure
(**cadar** *pair*) essential procedure
(**caddr** *pair*) essential procedure
(**cdaar** *pair*) essential procedure
(**cdadr** *pair*) essential procedure
(**cddar** *pair*) essential procedure
(**cdddr** *pair*) essential procedure
(**caaaar** *pair*) essential procedure
(**caaadr** *pair*) essential procedure
(**caadar** *pair*) essential procedure

(caaddr *pair***)** essential procedure
(cadaar *pair***)** essential procedure
(cadadr *pair***)** essential procedure
(caddar *pair***)** essential procedure
(cadddr *pair***)** essential procedure
(cdaaar *pair***)** essential procedure
(cdaadr *pair***)** essential procedure
(cdadar *pair***)** essential procedure
(cdaddr *pair***)** essential procedure
(cddaar *pair***)** essential procedure
(cddadr *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)))))
```

'() essential constant
#!null constant

`'()` and `#!null` are notations for the empty list. The `#!null` notation does not have to be quoted in programs. The `()` notation must be quoted in programs, however, because otherwise it would be a procedure call without a expression in the procedure position.

Rationale: Because many current Scheme interpreters deal with expressions as list structures rather than as character strings, they will treat an unquoted `()` as though it were quoted. It is entirely possible, however, that some implementations of Scheme will be able to detect an unquoted `()` as an error.

(null? *obj***)** essential procedure

Returns `#!true` if *obj* is the empty list, otherwise returns `#!false`.

(list *obj1* ...**)** essential procedure

Returns a proper list of its arguments.

```
(list 'a (+ 3 4) 'c)          -->  (a 7 c)
```

(length *plist***)** essential procedure

Returns the length of *plist*, which must be a proper list.

```
(length '())                    -->  0
(length '(a b c))               -->  3
(length '(a (b) (c d e)))       -->  3
```

(append *plist1 plist2***)** essential procedure
(append *plist ...***)** procedure

All *plist*s should be proper lists. Returns a list consisting of the elements of the first *plist* followed by the elements of the other *plist*s.

```
(append '(x) '(y))              -->  (x y)
(append '(a) '(b c d))          -->  (a b c d)
(append '(a (b)) '((c)))        -->  (a (b) (c))
```

(append! *plist ...***)** procedure

Like append but may side effect all but its last argument.

(reverse *plist***)** procedure

plist must be a proper list. Returns a list consisting of the elements of *plist* in reverse order.

```
(reverse '(a b c))              -->  (c b a)
(reverse '(a (b c) d (e (f))))  -->  ((e (f)) d (b c) a)
```

(list-ref *x n***)** procedure

Returns the car of (`list-tail` *x n*).

(list-tail *x n***)** procedure

Returns the sublist of *x* obtained by omitting the first *n* elements. Could be defined by

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

(last-pair *x*) procedure

Returns the last pair in the nonempty list *x*. Could be defined by

```
(define last-pair
  (lambda (x)
    (if (pair? (cdr x))
        (last-pair (cdr x))
        x)))
```

(memq *obj plist*) essential procedure
(memv *obj plist*) essential procedure
(member *obj plist*) essential procedure

Finds the first occurrence of *obj* in the proper list *plist* and returns the first sublist of *plist* beginning with *obj*. If *obj* does not occur in *plist*, returns `#!false`. `memq` uses `eq?` to compare *obj* with the elements of *plist*, while `memv` uses `eqv?` and `member` uses `equal?`.

```
(memq 'a '(a b c))              -->  (a b c)
(memq 'b '(a b c))              -->  (b c)
(memq 'a '(b c d))              -->  #!false
(memq (list 'a) '(b (a) c))     -->  #!false
(memq 101 '(100 101 102))       -->   unspecified
(memv 101 '(100 101 102))       -->  (101 102)
(member (list 'a) '(b (a) c))   -->  ((a) c)
```

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

alist must be a proper list of pairs. Finds the first pair in alist whose car field is *obj* and returns that pair. If no pair in *alist* has *obj* as its car, returns `#!false`. `assq` uses `eq?` to compare *obj* with the car fields of the pairs in *alist*, while `assv` uses `eqv?` and `assoc` uses `equal?`.

```
(assq 'a '((a 1) (b 2) (c 3)))      -->  (a 1)
(assq 'b '((a 1) (b 2) (c 3)))      -->  (b 2)
(assq 'd '((a 1) (b 2) (c 3)))      -->  #!false
(assq (list 'a)
      '(((a)) ((b)) ((c))))         -->  #!false
(assq 5 '((2 3) (5 7) (11 13)))     -->   unspecified
(assv 5 '((2 3) (5 7) (11 13)))     -->  (5 7)
(assoc (list 'a)
       '(((a)) ((b)) ((c))))        -->  ((a))
```

Rationale: `memq`, `memv`, `member`, `assq`, `assv`, and `assoc` do not have question marks in their names because they return useful values rather than just `#!true`.

II.5. Symbols

Symbols are objects whose usefulness rests entirely on the fact that two symbols are identical (in the sense of `eq?`) 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. Programmers may also use symbols as they use enumerated values in Pascal.

The rules for writing a symbol are the same as the rules for writing an identifier (see section I.2). As with identifiers, different implementations of Scheme use slightly different rules, but it is always the case that a sequence of characters that contains no special characters and begins with a character that cannot begin a number is taken to be a symbol; in addition `+`, `-`, `1+`, and `-1+` are symbols.

The case in which a symbol is written is unimportant. Some implementations of Scheme convert any upper case letters to lower case, and others convert lower case to upper case.

It is guaranteed that any symbol that has been 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 `eq?`). 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.

Rationale: Some implementations of Lisp 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 have "uninterned symbols", which defeat write/read invariance even in implementations with slashification and also generate exceptions to the rule that two symbols are the same if and only if their names are spelled the same. It is questionable whether these features are worth their complexity, so they are not standard in Scheme.

(symbol? *obj*) essential procedure

Returns `#!true` if *obj* is a symbol, otherwise returns `#!false`.

```
(symbol? 'foo)          -->  #!true
(symbol? (car '(a b)))  -->  #!true
(symbol? "bar")         -->  #!false
```

(symbol->string *symbol***)** essential procedure

Returns the name of *symbol* as a string. `symbol->string` performs no case conversion. See `string->symbol`. The following examples assume the `read` procedure converts to lower case:

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

(string->symbol *string***)** essential procedure

Returns the symbol whose name is *string*. `string->symbol` can create symbols with 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`.

```
'mISSISSIppi                          -->  mississippi
(string->symbol "mISSISSIppi")        -->  mISSISSIppi
(eq? 'bitBlt
     (string->symbol "bitBlt"))       -->   unspecified
(eq? 'JollyWog
     (string->symbol
       (symbol->string 'JollyWog))) -->  #!true
(string=?
  "K. Harper, M.D."
  (symbol->string
    (string->symbol
      "K. Harper, M.D.")))           -->  #!true
```

II.6. Numbers

Numerical computation has traditionally been neglected by the Lisp community. Until Common Lisp there has been no carefully thought out strategy for organizing numerical computation, and with the exception of the MacLisp system there has been little effort to execute numerical code efficiently. We applaud the excellent work of the Common Lisp committee and we accept many of their recommendations. In some ways we simplify and generalize their proposals in a manner consistent with the purposes of Scheme.

Scheme's numerical operations treat numbers as abstract data, as independent of their representation as is possible. Thus, the casual user should be able to write simple programs without having to know that the implementation may use fixed-point, floating-point, and perhaps other representations for his data. Unfortunately, this illusion of uniformity can be sustained only approximately – the implementation of numbers will leak out of its abstraction whenever the user must be in control of precision, or accuracy, or when he must construct especially efficient computations. Thus the language must also provide escape mechanisms so that a sophisticated programmer can exercise more control over the execution of his code and the representation of his data when necessary.

It is important to distinguish between the abstract numbers, their machine representations, and their written representations. We will use mathematical words such as NUMBER, COMPLEX, REAL, RATIONAL, and INTEGER for properties of the abstract numbers, names such as FIXNUM, BIGNUM, RATNUM, and FLONUM for machine representations, and names like INT, FIX, FLO, SCI, RAT, POLAR, and RECT for input/output formats.

Numbers

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

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

NUMBER
COMPLEX
REAL
RATIONAL
INTEGER

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

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

Exactness

Numbers are either EXACT or INEXACT. A number is exact if it was derived from EXACT numbers using only EXACT operations. A number is INEXACT if it models a quantity known only approximately, if it was derived using INEXACT ingredients, or if it was derived using INEXACT operations. Thus INEXACTness is a contagious property of a number. Some operations, such as the square root (of non-square numbers) must be INEXACT because of the finite precision of our representations. Other operations are inexact because of implementation requirements. We emphasize that exactness is independent of the position of the number on the tower. It is perfectly possible to have an INEXACT INTEGER or an EXACT REAL; 355/113 may be an EXACT RATIONAL or it may be an INEXACT RATIONAL approximation to pi, depending on the application.

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

the system must do its best to maintain precision. Scheme systems must not burden users with numerical operations described in terms of hardware and operating-system dependent representations such as FIXNUM and FLONUM, however, because these representation issues are hardly ever germane to the user's problems.

We highly recommend that the IEEE 32-bit and 64-bit floating-point standards be adopted for implementations that use floating-point representations internally. To minimize loss of precision we adopt the following rules: If an implementation uses several different sizes of floating-point formats, the results of any operation with a floating-point result must be expressed in the largest format used to express any of the floating-point arguments to that operation. It is desirable (but not required) for potentially irrational operations such as `sqrt`, when applied to EXACT arguments, to produce EXACT answers whenever possible (for example the square root of an exact 4 ought to be an exact 2). If an EXACT number (or an INEXACT number represented as a FIXNUM, a BIGNUM, or a RATNUM) is operated upon so as to produce an INEXACT result (as by `sqrt`), and if the result is represented as a FLONUM, then the largest available FLONUM format must be used; but if the result is expressed as a RATNUM then the rational approximation must have at least as much precision as the largest available FLONUM.

Numerical operations

Scheme provides the usual set of operations for manipulating numbers. In general, numerical operations require numerical arguments. For succintness we let the following meta-symbols range over the indicated types of object in our descriptions, and we let these meta-symbols specify the types of the arguments to numeric operations. It is an error for an operation to be presented with an argument that it is not specified to handle.

obj	any object
z, z1, ... zi, ...	complex, real , rational, integer
x, x1, ... xi, ...	real, rational, integer
q, q1, ... qi, ...	rational, integer
n, n1, ... ni, ...	integer

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

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

(zero? *z*) essential procedure
(positive? *x*) essential procedure
(negative? *x*) essential procedure
(odd? *n*) essential procedure
(even? *n*) essential procedure
(exact? *z*) essential procedure
(inexact? *z*) essential procedure

These numerical predicates test a number for a particular property, returning `#!true` or `#!false`.

(= *z1 z2*) essential procedure
(=? *z1 z2*) essential procedure
(< *x1 x2*) essential procedure
(<? *x1 x2*) essential procedure
(> *x1 x2*) essential procedure
(>? *x1 x2*) essential procedure
(<= *x1 x2*) essential procedure
(<=? *x1 x2*) essential procedure
(>= *x1 x2*) essential procedure
(>=? *x1 x2*) essential procedure

These numerical comparison predicates have redundant names (with and without the terminal "?") to make all user populations happy. Some implementations allow them to take many arguments, as in Common Lisp, to facilitate range checks. These procedures return `#!true` if their arguments are (respectively): numerically equal, monotonically increasing, monotonically decreasing, monotonically nondecreasing, or monotonically nonincreasing. Warning: On INEXACT numbers the equality tests will give unreliable results, and the other numerical comparisons will be useful only heuristically; when in doubt, consult a numerical analyst.

(max *x1 x2*) essential procedure
(max *x1 x2 ...*) procedure
(min *x1 x2*) essential procedure
(min *x1 x2 ...*) procedure

Returns the maximum or minimum of its arguments, respectively.

(+ *z1 z2*) essential procedure
(+ *z1* ...) procedure
(* *z1 z2*) essential procedure
(* *z1* ...) procedure

These procedures return the sum or product of their arguments.

```
(+ 3 4)          -->  7
(+ 3)            -->  3
(+)              -->  0
(* 4)            -->  4
(*)              -->  1
```

(- *z1 z2*) essential procedure
(- *z1 z2* ...) procedure
(/ *z1 z2*) essential procedure
(/ *z1 z2* ...) procedure

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

```
(- 3 4)          -->  -1
(- 3 4 5)        -->  -6
(- 3)            -->  -3
(/ 3 4 5)        -->  3/20
(/ 3)            -->  1/3
```

(1+ *z*) procedure
(-1+ *z*) procedure

These procedures return the result of adding 1 to or subtracting 1 from their argument.

(abs *z*) essential procedure

Returns the magnitude of its argument.

```
(abs -7)         -->  7
(abs -3+4i)      -->  5
```

(quotient *n1 n2*) essential procedure
(remainder *n1 n2*) essential procedure
(modulo *n1 n2*) procedure

In general, these are intended to implement number-theoretic (integer) division: For positive integers n_1 and n_2, if n_3 and n_4 are integers such that

$n_1 = n_2 n_3 + n_4$ and $0 \leq n_4 < n_2$, then

```
(quotient n1 n2)        -->  n3
(remainder n1 n2)       -->  n4
(modulo n1 n2)          -->  n4
```

The value returned by `quotient` always has the sign of the product of its arguments. `Remainder` and `modulo` differ on negative arguments as do the Common Lisp `rem` and `mod` procedures—the `remainder` always has the sign of the dividend, the `modulo` always has the sign of the divisor:

```
(modulo 13 4)           -->  1
(remainder 13 4)        -->  1
(modulo -13 4)          -->  3
(remainder -13 4)       -->  -1
(modulo 13 -4)          -->  -3
(remainder 13 -4)       -->  1
(modulo -13 -4)         -->  -1
(remainder -13 -4)      -->  -1
```

(gcd *n1* ...**)** procedure
(lcm *nl* ...**)** 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)                   -->  1
```

(floor *x***)** procedure
(ceiling *x***)** procedure
(truncate *x***)** procedure
(round *x***)** procedure
(rationalize *x y***)** procedure
(rationalize *x***)** procedure

These procedures create integers and rationals. Their results are not EXACT—in fact, their results are clearly INEXACT, though they can be made EXACT with an explicit exactness coercion.

`Floor` returns the largest integer not larger than x. `Ceiling` returns the smallest integer not smaller than x. `Truncate` returns the integer of maximal absolute value not larger than the absolute value of x. `Round` returns the

closest integer to x, rounding to even when x is halfway between two integers. With two arguments, `rationalize` produces the best rational approximation to x within the tolerance specified by y. With one argument, `rationalize` produces the best rational approximation to x, preserving all of the precision in its representation.

(exp *z***)**	procedure
(log *z***)**	procedure
(expt *z1 z2***)**	procedure
(sqrt *z***)**	procedure
(sin *z***)**	procedure
(cos *z***)**	procedure
(tan *z***)**	procedure
(asin *z***)**	procedure
(acos *z***)**	procedure
(atan *z1 z2***)**	procedure

These procedures are part of every implementation that supports real numbers. Their meanings conform with the Common Lisp standard. (Implementors should be careful of the branch cuts if complex numbers are allowed.)

(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 complex numbers. Suppose x_1, x_2, x_3, and x_4 are real numbers and z is a complex number such that

$$z = x_1 + x_2\mathrm{i} = x_3 \cdot e^{\mathrm{i}x_4}$$

Then `make-rectangular` and `make-polar` return z, `real-part` returns x_1, `imag-part` returns x_2, `magnitude` returns x_3, and `angle` returns x_4.

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

`exact->inexact` returns an INEXACT representation of z, which is a fairly harmless thing to do. `inexact->exact` returns an EXACT representation of z. Since the law of "garbage in, garbage out" remains in force, `inexact->exact` should not be used casually.

Numerical Input and Output

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

Scheme numbers are written according to the grammar described below. In that description, x^* means zero or more occurrences of x. Spaces never appear inside a number, so all spaces in the grammar are for legibility. <empty> stands for the empty string.

```
bit  -->  0  |  1
oct  -->  0  |  1  |  2  |  3  |  4  |  5  |  6  |  7
dit  -->  oct  |  8  |  9
hit  -->  dit  |  a  |  b  |  c  |  d  |  e  |  f
               |  A  |  B  |  C  |  D  |  E  |  F

radix2  -->  #b  |  #B
radix8  -->  #o  |  #O
radix10  -->  <empty>  |  #d  |  #D
radix16  -->  #x  |  #X
exactness  -->  <empty>  |  #i  |  #I  |  #e  |  #E
precision  -->  <empty>  |  #s  |  #S  |  #l  |  #L

prefix2  -->  radix2 exactness precision
           |  radix2 precision exactness
           |  exactness radix2 precision
           |  exactness precision radix2
           |  precision radix2 exactness
           |  precision exactness radix2

prefix8  -->  radix8 exactness precision
           |  radix8 precision exactness
           |  exactness radix8 precision
           |  exactness precision radix8
           |  precision radix8 exactness
           |  precision exactness radix8
```

```
prefix10  -->  radix10 exactness precision
             |  radix10 precision exactness
             |  exactness radix10 precision
             |  exactness precision radix10
             |  precision radix10 exactness
             |  precision exactness radix10

prefix16  -->  radix16 exactness precision
             |  radix16 precision exactness
             |  exactness radix16 precision
             |  exactness precision radix16
             |  precision radix16 exactness
             |  precision exactness radix16

sign  -->  <empty>  |  +  |  -
suffix  -->  <empty>  |  e sign dit dit*    |  E sign dit dit*
ureal  -->  prefix2 bit bit* #* suffix
          |  prefix2 bit bit* #* / bit bit* #* suffix
          |  prefix2 . bit bit* #* suffix
          |  prefix2 bit bit* . bit* #* suffix
          |  prefix2 bit bit* #* . #* suffix

          |  prefix8 oct oct* #* suffix
          |  prefix8 oct oct* #* / oct oct* #* suffix
          |  prefix8 . oct oct* #* suffix
          |  prefix8 oct oct* . oct* #* suffix
          |  prefix8 oct oct* #* . #* suffix

          |  prefix10 dit dit* #* suffix
          |  prefix10 dit dit* #* / dit dit* #* suffix
          |  prefix10 . dit dit* #* suffix
          |  prefix10 dit dit* . dit* #* suffix
          |  prefix10 dit dit* #* . #* suffix

          |  prefix16 hit hit* #* suffix
          |  prefix16 hit hit* #* / hit hit* #* suffix
          |  prefix16 . hit hit* #* suffix
          |  prefix16 hit hit* . hit* #* suffix
          |  prefix16 hit hit* #* . #* suffix
real  -->  sign ureal
number  -->  real  |  real + ureal i  |  real - ureal i
           |  real @ real
```

The conventions used to print a number can be specified by a format, as described later in this section. The system provides a procedure, number-

`>string`, that takes a number and a format and returns as a string the printed expression of the given number in the given format.

(number->string *number format*) procedure

This procedure will mostly be used by sophisticated users and in system programs. In general, a naive user will need to know nothing about the formats because the system printer will have reasonable default formats for all types of NUMBERs. The system reader will construct reasonable default numerical types for numbers expressed in each of the formats it recognizes. If a user needs control of the coercion from strings to numbers he will use `string->number`, which takes a string, an exactness, and a radix and produces a number of the maximally precise applicable type expressed by the given string.

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

The *exactness* is a symbol, either `E` (or `EXACT`) or `I` (or `INEXACT`). The *radix* is also a symbol: `B` (or `BINARY`), `O` (or `OCTAL`), `D` (or `DECIMAL`), and `X` (or `HEXADECIMAL`). Returns a number of the maximally precise representation expressed by the given *string*. It is an error if *string* does not express a number according to the grammar presented above.

Formats

Formats may have parameters. For example, the `(SCI 5 2)` format specifies that a number is to be expressed in Fortran scientific format with 5 significant places and two places after the radix point.

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

```
123  +123  -123                  ; (int)
123456789012345678901234567      ; (int) ; a big one!
355/113  +355/113  -355/113      ; (rat)
+123.45  -123.45                 ; (fix 2)
3.14159265358979                 ; (fix 14)
3.14159265358979                 ; (flo 15)
123.450                          ; (flo 6)
-123.45e-1                       ; (sci 5 2)
123e3  123e-3  -123e-3           ; (sci 3 0)
-1+2i                            ; (rect (int) (int))
1.2@1.570796                     ; (polar (fix 1) (flo 7))
```

A numerical constant may be specified with an explicit radix by a prefix. The prefixes are: #B (binary), #O (octal), #D (decimal), #X (hex). A format may specify that a number should be expressed in a particular radix. The radix prefix may also be suppressed. For example, one may express a complex number in polar form with the magnitude in octal and the angle in decimal as follows:

```
#o1.2@#d1.570796327 ; (polar (flo 2 (radix o)) (flo (radix d)))
#o1.2@1.570796327   ; (polar (flo 2 (radix o)) (flo (radix d s)))
```

A numerical constant may be specified to be either EXACT or INEXACT by a prefix. The prefixes are: #I (inexact), #E (exact). An exactness prefix may appear before or after any radix prefix that is used. A format may specify that a number should be expressed with an explicit exactness prefix, or it may force the exactness to be suppressed. For example, the following are ways to output an inexact value for pi:

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

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

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

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

```
#S3.14159265358979  ; Round to short - 3.141593
#L.6                ; Extend to long - .600000000000000
```

Details of formats

The format of a number is a list beginning with a format descriptor, which is a symbol such as SCI. Following the descriptor are parameters used by that descriptor, such as the number of significant digits to be used. Default values are supplied for any parameters that are omitted. Modifiers may appear

next, such as the RADIX and EXACTNESS descriptors described below, which themselves take parameters. The format descriptors are:

(INT)

Express as an integer. The radix point is implicit. If there are not enough significant places then insignificant digits will be flagged. For example, 6.0238E23 (represented internally as a 7 digit FLONUM) would be printed as

6023800##################

(RAT *n*)

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

(FIX *n*)

Express with a fixed radix point. *n* specifies the number of places to the right of the radix point. *n* defaults to the size of a single-precision FLONUM. If there are not enough significant places, then insignificant digits will be flagged. For example, 6.0238E23 (represented internally as a 7 digit FLONUM) would be printed with a (FIX 2) format as 6023800##################.##

(FLO *n*)

Express with a floating radix point. *n* specifies the total number of places to be displayed. *n* defaults to the size of a single-precision FLONUM. If the number is out of range, it is converted to (SCI). (FLO H) allows the system to express a FLO heuristically for human consumption.

(SCI *n m*)

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

(RECT *r i*)

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

(POLAR *m a*)

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

(HEUR)

Express heuristically using the minimum number of digits required to get an expression that when coerced back to a number produces the original machine representation. EXACT numbers are expressed as (INT) or (RAT). INEXACT numbers are expressed as (FLO H) or (SCI H) depending on their range. Complex numbers are expressed in (RECT). This is the normal default of the system printer.

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

(EXACTNESS *s*)

This controls the expression of the exactness label of a number. *s* indicates whether the exactness is to be E (expressed) or S (suppressed). *s* defaults to *E*. If no exactness modifier is specified for a format then the exactness is by default not expressed.

(RADIX *r s*)

This forces a number to be expressed in the radix *r*. *r* may be the symbol B (binary), O (octal), D (decimal), or X (hex). *s* indicates whether the radix label is to be E (expressed) or S (suppressed). *s* defaults to E. If no radix modifier is specified then the default is decimal and the label is suppressed.

II.7 Characters

Characters are written using the #\ notation of Common Lisp. For example:

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

Characters written in the #\ notation are self-evaluating. That is, they do not have to be quoted in programs. The #\ notation is not an essential part of Scheme, however. Even implementations that support the #\ notation for input do not have to support it for output, and there is no requirement that the data type of characters be disjoint from data types such as integers or strings.

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

(**char?** *obj*) essential procedure

Returns #!true if *obj* is a character, otherwise returns #!false.

(**char=?** *char1 char2*) essential procedure
(**char<?** *char1 char2*) essential procedure
(**char>?** *char1 char2*) essential procedure
(**char<=?** *char1 char2*) essential procedure
(**char>=?** *char1 char2*) essential procedure

Both *char1* and *char2* must be characters. 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 #!true.
- The lower case characters are in order. For example, (char<? #\a #\b) returns #!true.
- The digits are in order. For example, (char<? #\0 #\9) returns #!true.
- Either all the digits precede all the upper case letters, or vice versa.
- Either all the digits precede all the lower case letters, or vice versa.

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

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

Both *char1* and *char2* must be characters. 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 `#!true`. Some implementations may generalize these procedures to take more than two arguments, as with the corresponding arithmetic predicates.

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

Char must be a character These procedures return `#!true` if their arguments are upper case, lower case, alphabetic, numeric, or whitespace characters, respectively, otherwise they return `#!false`. 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 10 decimal digits. The whitespace characters are tab, line feed, form feed, carriage return, and space.

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

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

```
(char<=? a b)                    -->  #!true
(<=? x y)                        -->  #!true
```

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

```
(<=? (char->integer a)
     (char->integer b))          -->  #!true
(char<=? (integer->char x)
         (integer->char y))      -->  #!true
```

(char-upcase *char***)** procedure
(char-downcase *char***)** procedure

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

II.8. Strings

Strings are sequences of characters. In some implementations of Scheme they are immutable; other implementations provide destructive procedures such as `string-set!` that alter string objects.

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

```
"The word \"Recursion\" has many different meanings."
```

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

A string may continue from one line to the next, but this is usually a bad idea because the exact effect varies from one computer system to another.

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

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

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

(string? *obj***)** essential procedure

Returns `#!true` if *obj* is a string, otherwise returns `#!false`.

(string-null? *string***)** essential procedure

string must be a string. Returns `#!true` if *string* has zero length, otherwise returns `#!false`.

(string=? *string1 string2***)** essential procedure
(string-ci=? *string1 string2***)** procedure

Returns `#!true` if the two strings are the same length and contain the same characters in the same positions, otherwise returns `#!false`. `string-ci=?` treats upper and lower case letters as though they were the same character, but `string=?` treats upper and lower case as distinct characters.

(string<? *string1 string2***)** essential procedure
(string>? *string1 string2***)** essential procedure
(string<=? *string1 string2***)** essential procedure
(string>=? *string1 string2***)** essential procedure
(string-ci<? *string1 string2***)** procedure
(string-ci>? *string1 string2***)** procedure
(string-ci<=? *string1 string2***)** procedure
(string-ci>=? *string1 string2***)** procedure

These procedures are the lexicographic extensions to strings of the corresponding orderings on characters. For example, `string<?` is the lexicographic ordering on strings induced by the ordering `char<?` on characters. Some implementations may generalize these and the `string=?` and `string-ci=?` procedures to take more than two arguments.

(make-string *n***)** procedure
(make-string *n char***)** procedure

n must be a non-negative integer, and *char* must be a character. Returns a newly allocated string of length *n*. If *char* is given, then all elements of the string are initialized to *char*, otherwise the contents of the *string* are unspecified.

(string-length *string***)** essential procedure

Returns the number of characters in the given *string*.

(string-ref *string n***)** essential procedure

n must be a nonnegative integer less than the `string-length` of *string*. Returns character *n* using zero-origin indexing.

(substring *string start end***)** essential procedure

string must be a string, and *start* and *end* must be valid indexes of *string* with *start* <= *end*. Returns a newly allocated string formed from the characters of *string* beginning with index *start* and ending with index *end*.

(string-append *string1 string2***)** essential procedure
(string-append *string1* ...**)** procedure

Returns a new string whose characters form the catenation of the given strings.

(string->list *string***)** essential procedure
(list->string *chars***)** essential procedure

`string->list` returns a list of the characters that make up the given string. `list->string` returns a string formed from the proper list of characters *chars*. `string->list` and `list->string` are inverses so far as `equal?` is concerned. Implementations that provide destructive operations on strings should ensure that the results of these procedures are newly allocated objects.

(string-set! *string n char***)** procedure

string must be a string, *n* must be a valid index of *string*, and *char* must be a character. Stores *char* in element *n* of *string* and returns an unspecified value.

(string-fill! *string char***)** procedure

Stores *char* in every element of the given *string* and returns an unspecified value.

(string-copy *string***)** procedure

Returns a newly allocated copy of the given *string*.

(substring-fill! *string start end char***)** procedure

Stores *char* in elements *start* through *end* of the given *string* and returns an unspecified value.

(substring-move-right! *s1 m1 n1 s2 m2***)** procedure
(substring-move-left! *s1 m1 n1* s*2 m2***)** procedure

s1 and *s2* must be strings, *m1* and *n1* must be valid indexes of *s1* with *m1* <= *n1* and *m2* must be a valid index of *s2*. These procedures store the elements *m1* through *n1* of *s1* into the string *s2* starting at element *m2* and return an unspecified value.

The procedures differ only when *s1* and *s2* are `eq?` and the substring being moved overlaps the substring being replaced. In this case, `substring-move-right!` copies serially, starting with the rightmost element and proceeding to the left, while `substring-move-left!` begins with the leftmost element and proceeds to the right.

II.9. Vectors

Vectors are heterogenous mutable structures whose elements are indexed by integers. 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. A vector of length 3 containing the number zero in element 0, the list (2 2 2 2) in element 1, and the string "Anna" in element 2 can be written as `#(0 (2 2 2 2) "Anna")`

Implementations are not required to support this notation.

Vectors are created by the constructor procedure `make-vector`. The elements are accessed and assigned by the procedures `vector-ref` and `vector-set!`.

(vector? *obj***)** essential procedure

Returns `#!true` if *obj* is a vector, otherwise returns `#!false`.

(make-vector *size***)** essential procedure
(make-vector *size fill***)** procedure

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

(vector *obj* ...**)** essential procedure

Returns a newly allocated vector whose elements contain the given arguments. Analogous to `list`.

```
(vector 'a 'b 'c)          -->  #(a b c)
```

(vector-length *vec***)** essential procedure

Returns the number of elements in the vector *vec*.

(vector-ref *vec k***)** essential procedure

Returns the contents of element *k* of the vector *vec*. *k* must be a nonnegative integer less than (`vector-length` *vec*).

```
(vector-ref '#(1 1 2 3 5 8 13 21) 5)  -->  8
```

(vector-set! *vec k obj***)** essential procedure

Stores *obj* in element *k* of the vector *vec*. *k* must be a nonnegative integer less than (`vector-length` *vec*). The value returned by `vector-set!` is not

specified.

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

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

Returns a list of the objects contained in the elements of *vec*. See `list->vector`.

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

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

Returns a newly created vector whose elements are initialized to the elements of the proper list *elts*.

```
(list->vector '(dididit dah))    -->  #(dididit dah)
```

(vector-fill! *vec fill***)** procedure

Stores *fill* in every element of the vector *vec*. The value returned by `vector-fill!` is not specified.

II.10. The object table

(object-hash *obj***)** procedure
(object-unhash *n***)** procedure

`object-hash` associates an integer with *obj* in a global table and returns *obj*. `object-hash` guarantees that distinct objects (in the sense of `eq?`) are associated with distinct integers. `object-unhash` takes an integer and returns the object associated with that integer if there is one, returning `#!false` otherwise.

Rationale: `object-hash` and `object-unhash` can be implemented using association lists and the `assq` procedure, but the intent is that they be efficient hash functions for general objects. Furthermore it is intended that the Scheme system is free to destroy and reclaim the storage of objects that are accessible only through the object table. It follows that `object-unhash` is of questionable utility, as illustrated by the following scenario.

```
>>> (define x (cons 0 0))
x
>>> (object-hash x)
77
>>> (set! x 0)
...
>>> (gc)          ; garbage collection occurs for some reason
...
>>> (object-unhash 77)
 ???              ; ill-defined: #!false or (0 . 0)
```

II.11. Procedures

Procedures are created when lambda expressions are evaluated. Procedures do not have a standard printed representation.

The most common thing to do with a procedure is to call it with zero or more arguments. A Scheme procedure may also be stored in data structures or passed as an argument to procedures such as those described below.

(**apply** *proc args*) essential procedure
(**apply** *proc arg1 ... args*) procedure

proc must be a procedure and *args* must be a proper list of arguments. 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 (append (*list arg1 ...*) *args*) as the actual arguments.

```
(apply + (list 3 4))                    -->  7

(define compose
  (lambda (f g)
    (lambda args
      (f (apply g args)))))             -->  unspecified

((compose 1+ *) 3 4)                    -->  13
```

(**map** *f plist*) essential procedure
(**map** *f plist1 plist2 ...*) procedure

f must be a procedure of one argument and the *plists* must be proper lists. If more than one *plist* is given, then they should all be the same length. Applies f element-wise to the elements of the *plists* and returns a list of the results. The order in which *f* is applied to the elements of the *plists* is not specified.

```
(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 (1+ count))
         count)
       '(a b c)))                       -->  unspecified
```

(for-each *f plist***)** essential procedure
(for-each *f plist1 plist2* ...**)** procedure

The arguments to `for-each` are like the arguments to `map`, but `for-each` calls *f* for its side effects rather than for its values. Unlike `map`, `for-each` is guaranteed to call *f* on the elements of the *plists* in order from the first element to the last, and the value returned by `for-each` is not specified.

```
(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)
```

(call-with-current-continuation *f***)** essential procedure

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

The escape procedure created by `call-with-current-continuation` has unlimited extent just like any other procedure in Scheme. It may be stored in variables or data structures and may be called as many times as desired.

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

```
(call-with-current-continuation
  (lambda (exit)
    (for-each (lambda (x)
                (if (negative? x)
                    (exit x)))
              '(54 0 37 -3 245 19))
    #!true))                            -->  -3
```

```
(define list-length
  (lambda (obj)
    (call-with-current-continuation
      (lambda (return)
        ((rec loop (lambda (obj)
                     (cond ((null? obj) 0)
                           ((pair? obj)
                            (1+ (loop (cdr obj))))
                           (else (return #!false)))))
         obj)))))
                                          -->  list-length
(list-length '(1 2 3 4))                  -->  4
(list-length '(a b . c))                  -->  #!false
```

Rationale: The classic use of `call-with-current-continuation` is for structured, non-local exits from loops or procedure bodies, but in fact `call-with-current-continuation` is extremely useful for implementing a wide variety of advanced control structures.

Whenever a Scheme expression is evaluated there is a *continuation* wanting the result of the expression. The continuation represents an entire (default) future for the computation. If the expression is evaluated at top level, for example, then the continuation will take the result, print it on the screen, prompt for the next input, evaluate it, and so on forever. Most of the time the continuation includes actions specified by user code, as in a continuation that will take the result, multiply it by the value stored in a local variable, add seven, and give the answer to the top level continuation to be printed. Normally these ubiquitous continuations are hidden behind the scenes and programmers don't think much about them. On rare occasions, however, when programmers need to do something fancy, then they may need to deal with continuations explicitly. `call-with-current-continuation` allows Scheme programmers to do that by creating a procedure that acts just like the current continuation.

Most serious programming languages incorporate one or more special purpose escape constructs with names like `exit`, `return`, or even `goto`. In 1965, however, Peter Landin invented a general purpose escape operator called the J-operator. John Reynolds described a simpler but equally powerful construct in 1972. The `catch` special form described by Sussman and Steele in the 1975 report on Scheme is exactly the same as Reynolds's construct, though its name

came from a less general construct in MacLisp. The fact that the full power of Scheme's `catch` could be obtained using a procedure rather than a special form was noticed in 1982 by the implementors of Scheme 311, and the name `call-with-current-continuation` was coined later that year. Although the name is descriptive, some people feel it is too long and have taken to calling the procedure `call/cc`.

II.12. Ports

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

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

Proc is a procedure of one argument, and *string* is 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. Calls *proc* with one argument: the port obtained by opening the named file for input or output. If the file cannot be opened, an error is signalled. If the procedure returns, then the port is closed automatically and the value yielded by the procedure is returned. If the procedure does not return, then Scheme will not close the port unless it can prove that the port will never again be used for a read or write operation.

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

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

Returns `#!true` if *obj* is an input port or output port (respectively), otherwise returns `#!false`.

(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 is a procedure of no arguments, and *string* is 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*. Furthermore, in contrast to `call-with-input-file` and `call-with-output-file`, these procedures will attempt to close the default port and restore the previous default whenever the current continuation changes in such a way as to make it doubtful that the *thunk* will ever return.

(open-input-file *filename***)** procedure

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

(open-output-file *filename***)** procedure

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

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

Closes the file associated with *port*, rendering the *port* incapable of delivering or accepting characters. The value returned is not specified.

II.13. Input

The `read` procedure converts written representations of Scheme objects into the objects themselves. The written representations for Scheme objects are described in the sections devoted to the operations on those objects.

(eof-object? *obj*) essential procedure

Returns `#!true` if *obj* is an end of file object, otherwise returns `#!false`. The precise set of end of file objects will vary among implementations, but in any case no end of file object will ever be a character or an object that can be read in using `read`.

(read) essential procedure
(read *port*) essential procedure

Returns the next object parsable from the given input *port*, updating *port* to point to the first character past the end of the written representation of the object. If an end of file is encountered in the input before any characters are found that can begin an object, then an end of file object is returned. If an end of file is encountered after the beginning of an object's written representation, but the written representation is incomplete and therefore not parsable, an error is signalled. The *port* argument may be omitted, in which case it defaults to the value returned by `current-input-port`.

Rationale: This corresponds to Common Lisp's `read-preserving-whitespace`, but for simplicity it is never an error to encounter end of file except in the middle of an object.

(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`.

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

Returns `#!true` if a character is ready on the input *port* and returns `#!false` otherwise. If `char-ready` returns `#!true` 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 `#!true`. *port* may be omitted, in which case it defaults to the value returned by `current-input-port`.

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

(load *filename*) essential procedure

filename should be a string naming an existing file containing Scheme source code. The `load` procedure reads expressions from the file and evaluates them sequentially as though they had been typed interactively. It is not specified whether the results of the expressions are printed, however. The `load` procedure does not affect the values returned by `current-input-port` and `current-output-port`. `load` returns an unspecified value.

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

II.14. Output

(write *obj***)** essential procedure
(write *obj port***)** essential procedure

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

(display *obj***)** essential procedure
(display *obj port***)** essential procedure

Writes a representation of *obj* to the given *port*. Strings that appear in the written representation are not enclosed in doublequotes, and no characters are escaped within those strings. *display* returns an unspecified value. The *port* argument may be omitted, in which case it defaults to the value returned by `current-output-port`. See `write`.

Rationale: Like Common Lisp's `prin1` and `princ`, `write` is for producing machine-readable output and `display` is for producing human-readable output. Implementations that allow "slashification" within symbols will probably want `write` but not `display` to slashify funny characters in symbols.

(newline) essential procedure
(newline *port***)** essential procedure

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

(write-char *char***)** essential procedure
(write-char *char port***)** essential procedure

Writes the character *char* (not a written representation of the character) to the given *port* and returns an unspecified value. The *port* argument may be omitted, in which case it defaults to the value returned by `current-output-port`.

(transcript-on *filename***)** procedure
(transcript-off) procedure

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

Rationale: These procedures are redundant in some systems, but systems that need them should provide them.

Bibliography and References

1. Harold Abelson and Gerald Jay Sussman with Julie Sussman, *Structure and Interpretation of Computer Programs.* MIT Press, Cambridge MA, 1985.
2. John Batali, Chris Hanson, Neil Mayle, Howard Shrobe, Richard M. Stallman, and Gerald Jay Sussman, The Scheme-81 Architecture—System and Chip. In *Proceedings of the MIT Conference on Advanced Research in VLSI*, Paul Penfield Jr [ed], Artech House, Dedham MA, 1982.
3. William Clinger, The Scheme 311 Compiler: an Exercise in Denotational Semantics. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, August 1984, pages 356-364.
4. Carol Fessenden, William Clinger, Daniel P Friedman, and Christopher Haynes, Scheme 311 Version 4 Reference Manual. Indiana University Computer Science Technical Report 137, February 1983.
5. D Friedman, C Haynes, E Kohlbecker, and M Wand, Scheme 84 Interim Reference Manual. Indiana University Computer Science Technical Report 153, January 1985.
6. Daniel P Friedman and Christopher T Haynes, Constraining Control. In *Proceedings of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, January 1985, pages 245-254.
7. Christopher T Haynes, Daniel P Friedman, and Mitchell Wand, Continuations and coroutines. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, August 1984, pages 293-298.
8. Peter Landin, A Correspondence Between Algol 60 and Church's Lambda Notation: Part I. In *Communications of the ACM*, 8(2), February 1965, pages 89-101.
9. Drew McDermott, An Efficient Environment Allocation Scheme in an Interpreter for a Lexically-scoped Lisp. In *Conference Record of the 1980 Lisp Conference*, August 1980, pages 154-162.
10. Steven S Muchnick and Uwe F Pleban, A Semantic Comparison of Lisp and Scheme. In *Conference Record of the 1980 Lisp Conference*, August 1980, pages 56-64.
11. MIT Scheme Manual, Seventh Edition, September 1984.
12. Peter Naur et al, Revised Report on the Algorithmic Language Algol 60. In *Communications of the ACM*, 6(1), January 1963, pages 1-17.
13. Kent M Pitman, The Revised MacLisp Manual. MIT Artificial Intelligence Laboratory Technical Report 295, 21 May 1983 (Saturday Evening Edition).

14. Jonathan A Rees and Norman I Adams IV, T: A Dialect of Lisp or, LAMBDA: The Ultimate Software Tool. In *Proceedings of the 1982 ACM Symposium on Lisp and Functional Programming*, August 1982, pages 114-122.
15. Jonathan A Rees, Norman I Adams IV, and James R Meehan, The T Manual. Fourth Edition, 10 January 1984.
16. John Reynolds, Definitional Interpreters for Higher Order Programming Languages. In *ACM Conference Proceedings*, 1972, pages 717-740.
17. Richard M Stallman, Phantom Stacks—If You Look Too Hard, They Aren't There. MIT Artificial Intelligence Memo 556, July 1980.
18. Guy Lewis Steele Jr, and Gerald Jay Sussman, Lambda, the Ultimate Imperative. MIT Artificial Intelligence Memo 353, March 1976.
19. Guy Lewis Steele Jr, Lambda, The Ultimate Declarative. MIT Artificial Intelligence Memo 379, November 1976.
20. Guy Lewis Steele Jr, Debunking the "Expensive Procedure Call" Myth, or Procedure Call Implementations Considered Harmful, or Lambda, The Ultimate GOTO. In *ACM Conference Proceedings*, 1977, pages 153-162.
21. Guy Lewis Steele Jr, Rabbit: a Compiler for Scheme. MIT Artificial Intelligence Laboratory Technical Report 474, May 1978.
22. Guy Lewis Steele Jr, An overview of Common Lisp. In Conference Record of the 1982 *ACM Symposium on Lisp and Functional Programming*, August 1982, pages 98-107.
23. Guy Lewis Steele Jr, Common Lisp: the Language. Digital Press, 1984.
24. Guy Lewis Steele Jr and Gerald Jay Sussman, The revised report on Scheme, a dialect of Lisp. MIT Artificial Intelligence Memo 452, January 1978.
25. Guy Lewis Steele Jr and Gerald Jay Sussman, The Art of the Interpreter, or The Modularity Complex (Parts Zero, One, and Two). MIT Artificial Intelligence Memo 453, May 1978.
26. Guy Lewis Steele Jr and Gerald Jay Sussman, Design of a Lisp-Based Processor. In *Communications of the ACM*, 23(11), November 1980, pages 628-645.
27. Guy Lewis Steele Jr and Gerald Jay Sussman, The Dream of a Lifetime: A Lazy Variable Extent Mechanism. In *Conference Record of the 1980 Lisp Conference, August 1980*, pages 163-172.
28. Gerald Jay Sussman and Guy Lewis Steele Jr, Scheme: an Interpreter for Extended Lambda Calculus. MIT Artificial Intelligence Memo 349, December 1975.

29. Gerald Jay Sussman, Jack Holloway, Guy Lewis Steele Jr, and Alan Bell, Scheme-79—Lisp on a chip. In *IEEE Computer*, 14(7), July 1981, pages 10-21.
30. Mitchell Wand, Continuation-based Program Transformation Strategies. In *Journal of the ACM*, 27(1), 1978, pages 174-180.
31. Mitchell Wand, Continuation-based Multiprocessing. In *Conference Record of the 1980 Lisp Conference*, August 1978, pages 19-28.

Index