MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

AI Memo 848a September 1986

Revised^3 Report on the Algorithmic Language Scheme

JONATHAN REES AND WILLIAM CLINGER (*Editors*)

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

Abstract

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

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

This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology 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 contracts N00014-80-C-0505 and N00014-85-K-0124. Support for the Indiana University research is provided by NSF grants NCS 83-04567 and NCS 83-03325. This report is published as Computer Science Department Technical Report 174, October 1986.

With apologies to J. R. R. Tolkien.

DESCRIPTION OF THE LANGUAGE

1. Overview of Scheme

1.1. Semantics

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

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

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

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

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

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

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

Arguments to Scheme procedures are always passed by value, which means that the actual argument expressions are evaluated before the procedure gains control, whether the procedure needs the result of the evaluation or not. ML, C, and APL are three other languages that always pass arguments by value. This is distinct from the lazy-evaluation semantics of SASL, or the call-by-name semantics of Algol 60, where an argument expression is not evaluated unless its value is needed by the procedure.

1.2. Syntax

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

1.3. Notation and terminology

1.3.1. Essential and non-essential features

It is required that every implementation of Scheme support features that are marked as being *essential*. Features not explicitly marked as essential are not essential. Implementations are free to omit non-essential features of Scheme or to add extensions, provided the extensions are not in conflict with the language reported here.

1.3.2. Error situations and unspecified behavior

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

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

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

1.3.3. Entry format

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

template	essential *category*

if the feature is an essential feature, or simply

template	*category*

if the feature is not an essential feature.

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

⟨thing$_1$⟩ ...

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

⟨thing$_1$⟩ ⟨thing$_2$⟩ ...

indicates one or more occurrences of a ⟨thing⟩.

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

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

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

(append *list*$_1$ *list*$_2$)	essential procedure
(append *list* ...)	procedure

indicate that in all implementations, the append procedure must be defined to take two arguments, and some implementations will extend it to take zero or more arguments.

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

obj	any object
$z, z_1, \ldots z_j, \ldots$	complex, real, rational, integer
$x, x_1, \ldots x_j, \ldots$	real, rational, integer
$y, y_1, \ldots y_j, \ldots$	real, rational, integer
$q, q_1, \ldots q_j, \ldots$	rational, integer
$n, n_1, \ldots n_j, \ldots$	integer
$k, k_1, \ldots k_j, \ldots$	exact non-negative integer

1.3.4. Evaluation examples

The symbol "$\Longrightarrow$" used in program examples should be read "evaluates to." For example,

```
(* 5 8)                    ⟹  40
```

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

2. Lexical conventions

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

Upper and lower case forms of a letter are never distinguished except within character and string constants. For example, `Foo` is the same identifier as `FOO`, and `#x1AB` is the same number as `#X1ab`.

2.1. Identifiers

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

```
lambda                  q
list->vector            soup
+                       V17a
<=?                     a34kTMNs
the-word-recursion-has-many-meanings
```

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

```
* / < = > ! ? : $ % _ & ~ ^
```

See section 7.1.1 for a formal syntax of identifiers.

Identifiers have several uses within Scheme programs:

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

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

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

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

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

2.2. Whitespace and comments

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

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

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

2.3. Other notations

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

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

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

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

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

`, ,@` The character comma and the sequence comma at-sign are used in conjunction with backquote (section 4.2.6).

`"` The double quote character is used to delimit strings (section 6.7).

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

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

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

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

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

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

`#e #i #l #s #b #o #d #x` These are used in the notation for numbers (section 6.5.3).

3. Basic concepts

3.1. Variables and regions

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

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

Like Algol and Pascal, and unlike most other dialects of Lisp except for Common Lisp, Scheme is a statically scoped language with block structure. To each place where a variable is bound in a program there corresponds a *region* of the program text within which the binding is effective. The region is determined by the particular binding construct that establishes the binding; if the binding is established by a lambda expression, for example, then its region is the entire lambda expression. Every reference to or assignment of a variable refers to the binding of the variable that established the innermost of the regions containing the use. If there is no binding of the variable whose region contains the use, then the use refers to the binding for the variable in the top level environment, if any (section 6); if there is no binding for the identifier, it is said to be *unbound.*

3.2. True and false

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

3.3. External representations

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

The external representation of an object is not necessarily unique. The integer 28 also has representations "`28.000`" and "`#x1c`", and the list in the previous paragraph also has the representations "`( 08 13 )`" and "`(8 . (13 . ()))`" (see section 6.3).

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

An external representation may be written in a program to obtain the corresponding object (see `quote`, section 4.1.2).

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

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

The syntax of external representations of various kinds of objects accompanies the description of the primitives for manipulating the objects in the appropriate sections of chapter 6.

4. Expressions

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

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

4.1. Primitive expression types

4.1.1. Variable references

⟨variable⟩ essential syntax

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

```
(define x 28)
x                          ⟹  28
```

Revised^3 Report on the Algorithmic Language Scheme

JONATHAN REES AND WILLIAM CLINGER (*Editors*)

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

Dedicated to the Memory of ALGOL 60

SUMMARY

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

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

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

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

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

Chapter 7 provides a formal syntax for Scheme written in extended BNF, along with a formal denotational semantics.

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

CONTENTS

INTRODUCTION

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

Scheme has influenced the evolution of Lisp. Scheme was one of the first programming languages to incorporate first class procedures as in the lambda calculus, thereby proving the usefulness of static scope rules and block structure in a dynamically typed language. Scheme was the first major dialect of Lisp to distinguish procedures from lambda expressions and symbols, to use a single lexical environment for all variables, and to evaluate the operator position of a procedure call in the same way as an operand position. By relying entirely on procedure calls to express iteration, Scheme emphasized the fact that tail-recursive procedure calls are essentially goto's that pass arguments. Scheme was the first widely used programming language to embrace first class escape procedures, from which all known sequential control structures can be synthesized. A few of these innovations have recently been incorporated into Common Lisp, while others remain to be adopted.

Background

The first description of Scheme was written in 1975 [48]. A revised report [44] appeared in 1978, which described the evolution of the language as its MIT implementation was upgraded to support an innovative compiler [41]. Three distinct projects began in 1981 and 1982 to use variants of Scheme for courses at MIT, Yale, and Indiana University [30,23,10]. 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 dialects began to diverge until students and researchers occasionally found it difficult to understand code written at other sites.

Fifteen representatives of the major implementations of Scheme therefore met in October 1984 to work toward a better and more widely accepted standard for Scheme. Participating in this workshop were Hal Abelson, Norman Adams, David Bartley, Gary Brooks, William Clinger, Daniel Friedman, Robert Halstead, Chris Hanson, Christopher Haynes, Eugene Kohlbecker, Don Oxley, Jonathan Rees, Guillermo Rozas, Gerald Jay Sussman, and Mitchell Wand. Kent Pitman made valuable contributions to the agenda for the workshop but was unable to attend the sessions.

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

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

Acknowledgements

We would like to thank the following people for their comments and criticisms: Alan Bawden, George Carrette, Andy Cromarty, Andy Freeman, Richard Gabriel, Yekta Gürsel, Ken Haase, Paul Hudak, Richard Kelsey, Chris Lindblad, Mark Meyer, Jim Miller, Jim Philbin, John Ramsdell, Guy Lewis Steele Jr., Julie Sussman, Perry Wagle, Daniel Weise, and Henry Wu. We thank Carol Fessenden, Daniel Friedman, and Christopher Haynes for permission to use text from the Scheme 311 version 4 reference manual. We thank Texas Instruments, Inc. for permission to use text from the *TI Scheme Language Reference Manual*. We gladly acknowledge the influence of manuals for MIT Scheme, T, Scheme 84, Common Lisp, and Algol 60.

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

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

4.1.2. Literal expressions

(quote ⟨datum⟩) essential syntax
'⟨datum⟩ essential syntax
⟨constant⟩ essential syntax

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

```
(quote a)              ⟹  a
(quote #(a b c))       ⟹  #(a b c)
(quote (+ 1 2))        ⟹  (+ 1 2)
```

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

```
'a                     ⟹  a
'#(a b c)              ⟹  #(a b c)
'(+ 1 2)               ⟹  (+ 1 2)
'(quote a)             ⟹  (quote a)
''a                    ⟹  (quote a)
```

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

```
'"abc"                 ⟹  "abc"
"abc"                  ⟹  "abc"
'145932                ⟹  145932
145932                 ⟹  145932
'#t                    ⟹  #t
#t                     ⟹  #t
```

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

4.1.3. Procedure calls

(⟨operator⟩ ⟨operand$_1$⟩ ...) essential syntax

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

```
(+ 3 4)                ⟹  7
((if #f + *) 3 4)      ⟹  12
```

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

Procedure calls are also called *combinations*.

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

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

4.1.4. Lambda expressions

(lambda ⟨formals⟩ ⟨body⟩) essential syntax

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

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

```
(lambda (x) (+ x x))       ⟹  a procedure
((lambda (x) (+ x x)) 4)   ⟹  8

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

(define foo
  (let ((x 4))
    (lambda (y) (+ x y))))
(foo 6)                    ⟹  10
```

⟨Formals⟩ should have one of the following forms:

- (⟨variable$_1$⟩ ...): The procedure takes a fixed number of arguments; when the procedure is called, the arguments will be stored in the bindings of the corresponding variables.
- ⟨variable⟩: The procedure takes any number of arguments; when the procedure is called, the sequence of actual arguments is converted into a newly allocated list, and the list is stored in the binding of the ⟨variable⟩.
- (⟨variable$_1$⟩ ... ⟨variable$_{n-1}$⟩ . ⟨variable$_n$⟩): If a space-delimited period precedes the last variable, then the value stored in the binding of the last variable will be a newly allocated list of the actual arguments left over after all the other actual arguments have been matched up against the other formal arguments.

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

4.1.5. Conditionals

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

Syntax: ⟨Test⟩, ⟨consequent⟩, and ⟨alternate⟩ may be arbitrary expressions.

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

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

4.1.6. Assignments

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

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

```
(define x 2)
(+ x 1)                      ⟹  3
(set! x 4)                   ⟹  unspecified
(+ x 1)                      ⟹  5
```

4.2. Derived expression types

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

4.2.1. Conditionals

(cond ⟨clause$_1$⟩ ⟨clause$_2$⟩ ...) essential syntax

Syntax: Each ⟨clause⟩ should be of the form

(⟨test⟩ ⟨expression⟩ ...)

where ⟨test⟩ is any expression. The last ⟨clause⟩ may be an "else clause," which has the form

(else ⟨expression$_1$⟩ ⟨expression$_2$⟩ ...).

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

```
(cond ((> 3 2) 'greater)
      ((< 3 2) 'less))       ⟹  greater
(cond ((> 3 3) 'greater)
      ((< 3 3) 'less)
      (else 'equal))         ⟹  equal
```

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

```
(cond ((assv 'b '((a 1) (b 2))) => cadr)
      (else #f))             ⟹  2
```

(case ⟨key⟩ ⟨clause$_1$⟩ ⟨clause$_2$⟩ ...) syntax

Syntax: ⟨Key⟩ may be any expression. Each ⟨clause⟩ should have the form

((⟨datum$_1$⟩ ...) ⟨expression$_1$⟩ ⟨expression$_2$⟩ ...),

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

(else ⟨expression$_1$⟩ ⟨expression$_2$⟩ ...).

Semantics: A `case` expression is evaluated as follows. ⟨Key⟩ is evaluated and its result is compared against each ⟨datum⟩. If the result of evaluating ⟨key⟩ is equivalent (in the sense of eqv?; see section 6.2) to a ⟨datum⟩, then the expressions in the corresponding ⟨clause⟩ are evaluated from left to right and the result of the last expression in the ⟨clause⟩ is returned as the result of the `case` expression. If the result of evaluating ⟨key⟩ is different from every ⟨datum⟩, then if there is an else clause its expressions are evaluated and the result of the last is the result of the `case` expression; otherwise the result of the `case` expression is unspecified.

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

`(and` ⟨test$_1$⟩ `...)` syntax

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

```
(and (= 2 2) (> 2 1))          ⟹  #t
(and (= 2 2) (< 2 1))          ⟹  #f
(and 1 2 'c '(f g))            ⟹  (f g)
(and)                          ⟹  #t
```

`(or` ⟨test$_1$⟩ `...)` syntax

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

```
(or (= 2 2) (> 2 1))           ⟹  #t
(or (= 2 2) (< 2 1))           ⟹  #t
(or #f #f #f)                  ⟹  #f
(or (memq 'b '(a b c))
    (/ 3 0))                   ⟹  (b c)
```

4.2.2. Binding constructs

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

`(let` ⟨bindings⟩ ⟨body⟩`)` essential syntax

Syntax: ⟨Bindings⟩ should have the form

`((`⟨variable$_1$⟩ ⟨init$_1$⟩`) ...)`,

where each ⟨init⟩ is an expression, and ⟨body⟩ should be a sequence of one or more expressions.

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

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

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

See also named `let`, section 4.2.4.

`(let*` ⟨bindings⟩ ⟨body⟩`)` syntax

Syntax: ⟨Bindings⟩ should have the form

`((`⟨variable$_1$⟩ ⟨init$_1$⟩`) ...)`,

and ⟨body⟩ should be a sequence of one or more expressions.

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

```
(let* ((x 1) (y (+ x 1)))
  y)                           ⟹  2
```

`(letrec` ⟨bindings⟩ ⟨body⟩`)` essential syntax

Syntax: ⟨Bindings⟩ should have the form

`((`⟨variable$_1$⟩ ⟨init$_1$⟩`) ...)`,

and ⟨body⟩ should be a sequence of one or more expressions.

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

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

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

4.2.3. Sequencing

(begin ⟨expression$_1$⟩ ⟨expression$_2$⟩ ...) essential syntax

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

```
(begin (set! x 5)
       (+ x 1))              ⟹  6

(begin (display "4 plus 1 equals ")
       (display (+ 4 1)))    ⟹  unspecified
            and prints  4 plus 1 equals 5
```

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

4.2.4. Iteration

```
(do ((⟨variable1⟩ ⟨init1⟩ ⟨step1⟩)
     ...)
    (⟨test⟩ ⟨expression⟩ ...)
  ⟨command⟩ ...)
```
syntax

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

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

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

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

The region of the binding of a ⟨variable⟩ consists of the entire `do` expression except for the ⟨init⟩s.

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

```
(do ((vec (make-vector 5))
     (i 0 (+ i 1)))
    ((= i 5) vec)
  (vector-set! vec i 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
```

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

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

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

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

4.2.5. Delayed evaluation

(delay ⟨expression⟩) syntax

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

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

4.2.6. Quasiquotation

`(quasiquote ⟨template⟩)` syntax
`` `⟨template⟩ `` syntax

"Backquote" or "quasiquote" expressions are useful for constructing a list or vector structure when most but not all of the desired structure is known in advance. If no commas appear within the ⟨template⟩, the result of evaluating `` `⟨template⟩ `` is equivalent to the result of evaluating `'⟨template⟩`. If a comma appears within the ⟨template⟩, however, the expression following the comma is evaluated ("unquoted") and its result is inserted into the structure instead of the comma and the expression. If a comma appears followed immediately by an at-sign (`@`), then the following expression must evaluate to a list; the opening and closing parentheses of the list are then "stripped away" and the elements of the list are inserted in place of the comma at-sign expression sequence.

```
`(list ,(+ 1 2) 4)          ⟹  (list 3 4)
(let ((name 'a)) `(list ,name ',name))
            ⟹  (list a (quote a))
`(a ,(+ 1 2) ,@(map abs '(4 -5 6)) b)
            ⟹  (a 3 4 5 6 b)
`((foo ,(- 10 3)) ,@(cdr '(c)) . ,(car '(cons)))
            ⟹  ((foo 7) . cons)
`#(10 5 ,(sqrt 4) ,@(map sqrt '(16 9)) 8)
            ⟹  #(10 5 2 4 3 8)
```

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

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

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

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

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

5. Program structure

5.1. Programs

A Scheme program consists of a sequence of expressions and definitions. Expressions are described in chapter 4; definitions are the subject of the rest of the present chapter.

Programs are typically stored in files or entered interactively to a running Scheme system, although other paradigms are possible; questions of user interface lie outside the scope of this report. (Indeed, Scheme would still be useful as a notation for expressing computational methods even in the absence of a mechanical implementation.)

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

5.2. Definitions

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

A definition should have one of the following forms:

- `(define ⟨variable⟩ ⟨expression⟩)`

  This syntax is essential.

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

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

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

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

  This syntax is not essential. ⟨Formal⟩ should be a single variable. This form is equivalent to

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

5.2.1. Top level definitions

At the top level of a program, a definition

```
(define ⟨variable⟩ ⟨expression⟩)
```

has essentially the same effect as the assignment expression

```
(set! ⟨variable⟩ ⟨expression⟩)
```

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

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

All Scheme implementations must support top level definitions.

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

5.2.2. Internal definitions

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

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

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

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

6. Standard procedures

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

6.1. Booleans

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

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

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

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

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

```
#t                ⟹  #t
#f                ⟹  #f
'#f               ⟹  #f
```

`(not obj)` essential procedure

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

```
(not #t)          ⟹  #f
(not 3)           ⟹  #f
(not (list 3))    ⟹  #f
(not #f)          ⟹  #t
(not '())         ⟹  #t
(not (list))      ⟹  #t
```

`(boolean? obj)` essential procedure

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

```
(boolean? #f)     ⟹  #t
(boolean? 0)      ⟹  #f
```

`nil` variable
`t` variable

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

```
t                 ⟹  #t
nil               ⟹  #f
'nil              ⟹  nil
```

6.2. Equivalence predicates

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

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

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

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

(eqv? *obj*$_1$ *obj*$_2$) essential procedure

The eqv? procedure implements an approximation to the relation of operational equivalence. It returns #t if it can prove that *obj*$_1$ and *obj*$_2$ are operationally equivalent. If it can't, it always errs on the conservative side and returns #f.

The only situation in which it might fail to prove is when *obj*$_1$ and *obj*$_2$ are operationally equivalent procedures that were created at different times. In general, operational equivalence of procedures is uncomputable, but it is guaranteed that eqv? can recognize a procedure created at a given time by a given lambda expression as "being itself." This is useful for applications in which procedures are being used to implement objects with local state.

```
(eqv? 'a 'a)                    ⟹  #t
(eqv? 'a 'b)                    ⟹  #f
(eqv? 2 2)                      ⟹  #t
(eqv? '() '())                  ⟹  #t
(eqv? "" "")                    ⟹  #t
(eqv? 100000000 100000000)      ⟹  #t
(eqv? (cons 1 2) (cons 1 2))⟹  #f
(eqv? (lambda () 1)
      (lambda () 2))            ⟹  #f
(eqv? #f 'nil)                  ⟹  #f
(let ((p (lambda (x) x)))
  (eqv? p p))                   ⟹  #t
```

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

```
(eqv? (lambda (x) x)
      (lambda (x) x))           ⟹  unspecified
(eqv? (lambda (x) x)
      (lambda (y) y))           ⟹  unspecified
```

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

```
(define gen-counter
  (lambda ()
    (let ((n 0))
      (lambda () (set! n (+ n 1)) n))))
(let ((g (gen-counter)))
  (eqv? g g))                   ⟹  #t
(eqv? (gen-counter) (gen-counter))
                                ⟹  #f
(define gen-loser
  (lambda ()
    (let ((n 0))
      (lambda () (set! n (+ n 1)) 27))))
(let ((g (gen-loser)))
  (eqv? g g))                   ⟹  #t
(eqv? (gen-loser) (gen-loser))
                                ⟹  unspecified
```

Objects of distinct types are never operationally equivalent, except that false and the empty list are permitted to be identical, and the character type need not be disjoint from other types.

```
(eqv? '() #f)                    ⟹  unspecified
(eqv? 57 #\A)                    ⟹  unspecified
```

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

```
(let ((x '(a)))
  (eqv? x x))                    ⟹  #t
(eqv? '(a) '(a))                 ⟹  unspecified
(eqv? "a" "a")                   ⟹  unspecified
(eqv? '(b) (cdr '(a b)))         ⟹  unspecified
```

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

(eq? *obj*$_1$ *obj*$_2$**)** essential procedure

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

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

```
(eq? 'a 'a)                      ⟹  #t
(eq? '(a) '(a))                  ⟹  unspecified
(eq? (list 'a) (list 'a))        ⟹  #f
(eq? "a" "a")                    ⟹  unspecified
(eq? "" "")                      ⟹  unspecified
(eq? '() '())                    ⟹  #t
(eq? 2 2)                        ⟹  unspecified
(eq? #\A #\A)                    ⟹  unspecified
(eq? car car)                    ⟹  #t
(let ((n (+ 2 3)))
  (eq? n n))                     ⟹  unspecified
(let ((x '(a)))
  (eq? x x))                     ⟹  #t
(let ((x '#()))
  (eq? x x))                     ⟹  #t
(let ((p (lambda (x) x)))
  (eq? p p))                     ⟹  #t
```

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

(equal? *obj*$_1$ *obj*$_2$**)** essential procedure

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

```
(equal? 'a 'a)                   ⟹  #t
(equal? '(a) '(a))               ⟹  #t
(equal? '(a (b) c)
        '(a (b) c))              ⟹  #t
(equal? "abc" "abc")             ⟹  #t
(equal? 2 2)                     ⟹  #t
(equal? (make-vector 5 'a)
        (make-vector 5 'a))      ⟹  #t
(equal? (lambda (x) x)
        (lambda (y) y))          ⟹  unspecified
```

6.3. Pairs and lists

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

Pairs are used primarily to represent lists. A list can be defined recursively as either the empty list or a pair whose cdr is a list. The objects in the car fields of successive pairs of a list are the elements of the list. For example, a two-element list is a pair whose car is the first element and whose cdr is a pair whose car is the second element and whose cdr is the empty list. The length of a list is the number of elements, which is the same as the number of pairs.

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

The most general notation (external representation) for Scheme pairs is the "dotted" notation (*c*$_1$. *c*$_2$) where *c*$_1$ is the value of the car field and *c*$_2$ is the value of the cdr field. For example (4 . 5) is a pair whose car is 4 and whose cdr is 5. Note that (4 . 5) is the external representation of a pair, not an expression that evaluates to a pair.

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

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

and

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

are both representations of the same list of symbols.

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

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

is equivalent to

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

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

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

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

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

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

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

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

```
(pair? '(a . b))        ⟹  #t
(pair? '(a b c))        ⟹  #t
(pair? '())             ⟹  #f
(pair? '#(a b))         ⟹  #f
```

`(cons `*obj*$_1$ *obj*$_2$`)` essential procedure

Returns a newly allocated pair whose car is *obj*$_1$ and whose cdr is *obj*$_2$. The pair is guaranteed to be different (in the sense of `eqv?`) from every existing object.

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

`(car `*pair*`)` essential procedure

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

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

`(cdr `*pair*`)` essential procedure

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

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

`(set-car! `*pair obj*`)` essential procedure

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

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

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

`(caar `*pair*`)` essential procedure
`(cadr `*pair*`)` essential procedure
⋮ ⋮
`(cdddar `*pair*`)` essential procedure
`(cddddr `*pair*`)` essential procedure

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

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

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

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

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

`(list `*obj* `...)` essential procedure

Returns a list of its arguments.

```
    (list 'a (+ 3 4) 'c)         ⟹  (a 7 c)
    (list)                       ⟹  ()
```

`(length `*list*`)` essential procedure

Returns the length of *list*.

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

`(append `*list*$_1$ *list*$_2$`)` essential procedure
`(append `*list* `...)` procedure

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

```
    (append '(x) '(y))           ⟹  (x y)
    (append '(a) '(b c d))       ⟹  (a b c d)
    (append '(a (b)) '((c)))     ⟹  (a (b) (c))
```

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

```
    (append '(a b) '(c . d))     ⟹  (a b c . d)
    (append '() 'a)              ⟹  a
```

`(reverse `*list*`)` procedure

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

```
    (reverse '(a b c))           ⟹  (c b a)
    (reverse '(a (b c) d (e (f))))
              ⟹  ((e (f)) d (b c) a)
```

`(list-tail `*list k*`)` procedure

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

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

`(list-ref `*list k*`)` procedure

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

```
    (list-ref '(a b c d) 2)      ⟹  c
```

`(last-pair `*list*`)` procedure

Returns the last pair in the nonempty, possibly improper, list *list*. `Last-pair` could be defined by

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

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

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

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

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

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

```
    (define e '((a 1) (b 2) (c 3)))
    (assq 'a e)                  ⟹  (a 1)
    (assq 'b e)                  ⟹  (b 2)
    (assq 'd e)                  ⟹  #f
    (assq (list 'a) '(((a)) ((b)) ((c))))
```

```
                          ⟹  #f
(assoc (list 'a) '(((a)) ((b)) ((c))))
                          ⟹  ((a))
(assq 5 '((2 3) (5 7) (11 13)))
                          ⟹  unspecified
(assv 5 '((2 3) (5 7) (11 13)))
                          ⟹  (5 7)
```

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

6.4. Symbols

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

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

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

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

Some implementations also have "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.

(symbol? *obj*) essential procedure

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

```
(symbol? 'foo)            ⟹  #t
(symbol? (car '(a b)))    ⟹  #t
(symbol? "bar")           ⟹  #f
```

(symbol->string *symbol*) essential procedure

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

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

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

(string->symbol *string*) essential procedure

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

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

```
(eq? 'mISSISSIppi 'mississippi)
          ⟹  #t
(string->symbol "mISSISSIppi")
          ⟹  the symbol with name "mISSISSIppi"
(eq? 'bitBlt (string->symbol "bitBlt"))
          ⟹  #f
(eq? 'JollyWog
     (string->symbol
       (symbol->string 'JollyWog)))
          ⟹  #t
(string=? "K. Harper, M.D."
          (symbol->string
            (string->symbol "K. Harper, M.D.")))
          ⟹  #t
```

6.5. Numbers

Numerical computation has traditionally been neglected by the Lisp community. Until Common Lisp there has been no carefully thought out strategy for organizing numerical computation, and with the exception of the MacLisp system [28] there has been little effort to execute numerical code efficiently. We applaud the excellent work of the Common Lisp committee and we accept many of their recommendations. In some ways we simplify and generalize their proposals in a manner consistent with the purposes of Scheme.

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

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

6.5.1. Numbers

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

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

```
number
complex
real
rational
integer
```

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

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

6.5.2. Exactness

Numbers are either `exact` or `inexact`. A number is `exact` if it was derived from `exact` numbers using only `exact` operations. A number is `inexact` if it models a quantity (e.g., a measurement) known only approximately, if it was derived using `inexact` ingredients, or if it was derived using `inexact` operations. Thus `inexact`ness 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`.

6.5.3. Number syntax

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

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

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

A numerical constant may be specified to be either `exact` or `inexact` by a prefix. The prefixes are `#e` for exact, and `#i` for inexact. An exactness prefix may appear before or after any radix prefix that is used. If the representation of a numerical constant has no exactness prefix, the constant may be assumed to be exact or inexact at the discretion of the implementation, except that integers expressed without decimal points and without use of exponential notation are assumed exact.

In systems with both single and double precision `flonums` we may want to specify which size we want to use to represent a constant 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
```

6.5.4. Numerical operations

The reader is referred to section 1.3.3 for a summary of the naming conventions used to specify restrictions on the types of arguments to numerical routines.

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

These numerical type predicates can be applied to any kind of argument, including non-numbers. They return 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 `#t` or `#f`.

(= z_1 z_2**)**	essential procedure
(< x_1 x_2**)**	essential procedure
(> x_1 x_2**)**	essential procedure
(<= x_1 x_2**)**	essential procedure
(>= x_1 x_2**)**	essential procedure

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

(max x_1 x_2**)**	essential procedure
(max x_1 x_2 ...**)**	procedure
(min x_1 x_2**)**	essential procedure
(min x_1 x_2 ...**)**	procedure

These procedures return the maximum or minimum of their arguments.

(+ z_1 z_2**)**	essential procedure
(+ z_1 ...**)**	procedure
(* z_1 z_2**)**	essential procedure
(* z_1 ...**)**	procedure

These procedures return the sum or product of their arguments.

```
(+ 3 4)                  ⟹  7
(+ 3)                    ⟹  3
(+)                      ⟹  0
(* 4)                    ⟹  4
(*)                      ⟹  1
```

(- z_1 z_2) essential procedure
(- z_1 z_2 ...) procedure
(/ z_1 z_2) essential procedure
(/ z_1 z_2 ...) procedure

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

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

(abs z) essential procedure

Abs returns the magnitude of its argument.

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

(quotient n_1 n_2) essential procedure
(remainder n_1 n_2) essential procedure
(modulo n_1 n_2) procedure

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
```

For all integers n_1 and n_2 with n_2 not equal to 0,

```
(= n1 (+ (* n2 (quotient n1 n2))
         (remainder n1 n2)))
                         ⟹  #t
```

The value returned by `quotient` always has the sign of the product of its arguments. `Remainder` and `modulo` differ on negative arguments—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
```

(numerator q) procedure
(denominator q) procedure

These procedures return the numerator or denominator of their argument.

```
(numerator (/ 6 4))      ⟹  3
(denominator (/ 6 4))    ⟹  2
```

(gcd n_1 ...) procedure
(lcm n_1 ...) 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 `exact` if and only if their arguments are `exact`.

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

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

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

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

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

$$\log z = \log \mathrm{magnitude}(z) + i\, \mathrm{angle}(z)$$

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

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

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

`(sqrt` z`)` procedure

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

`(expt` z_1 z_2`)` procedure

Returns z_1 raised to the power z_2:

$$z_1{}^{z_2} = e^{z_2 \log z_1}$$

0^0 is defined to be equal to 1.

`(make-rectangular` x_1 x_2`)` procedure
`(make-polar` x_3 x_4`)` 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

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

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

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

`(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.

6.5.5. Numerical input and output

`(number->string` *number format*`)` procedure

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

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

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

6.5.6. Formats

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

Details of particular formats such as `sci` and `fix` are given in section 6.5.7.

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

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

```
123  +123  -123                ; (int)
1234567890123456789012345667   ; (int)
355/113  +355/113  -355/113    ; (rat)
+123.45  -123.45               ; (fix 2)
3.14159265358979               ; (fix 14)
3.14159265358979               ; (flo 15)
123.450                        ; (flo 6)
-123.45e-1                     ; (sci 5 2)
123e3  123e-3  -123e-3         ; (sci 3 0)
-1+2i                          ; (rect (int) (int))
1.2@1.570796                   ; (polar (fix 1)
                               ;        (flo 7))
```

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

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

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

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

6.5.7. Details of formats

The format descriptors are:

(int) format

Express as an integer. The radix point is implicit. If there are not enough significant places then insignificant digits will be flagged. For example, an inexact integer $6.0238 \cdot 10^{23}$ (represented internally as a 7 digit `flonum`) would be printed as

```
6023800#################
```

(rat *n*) format

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*) format

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

```
6023800#################.##
```

(flo *n*) format

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

(sci *n* *m*) format

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*) format

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*) format

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) format

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

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

`(exactness` *s*`)` format

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

`(radix` *r s*`)` format

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

6.6. Characters

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

Characters are written using the notation #\⟨character⟩ or #\⟨character name⟩. For example:

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

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

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

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

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

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

`(char=?` *char*$_1$ *char*$_2$`)` essential procedure
`(char<?` *char*$_1$ *char*$_2$`)` essential procedure
`(char>?` *char*$_1$ *char*$_2$`)` essential procedure
`(char<=?` *char*$_1$ *char*$_2$`)` essential procedure
`(char>=?` *char*$_1$ *char*$_2$`)` essential procedure

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

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

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

`(char-ci=?` *char*$_1$ *char*$_2$`)` procedure
`(char-ci<?` *char*$_1$ *char*$_2$`)` procedure
`(char-ci>?` *char*$_1$ *char*$_2$`)` procedure
`(char-ci<=?` *char*$_1$ *char*$_2$`)` procedure
`(char-ci>=?` *char*$_1$ *char*$_2$`)` procedure

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

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

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

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

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

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

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

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

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

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

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

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

These procedures return a character $char_2$ such that **(char-ci=?** *char* $char_2$**)**. In addition, if *char* is alphabetic, then the result of **char-upcase** is upper case and the result of **char-downcase** is lower case.

6.7. Strings

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

k must be a valid index of *string*. **String-ref** returns character *k* of *string* using zero-origin indexing.

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

k must be a valid index of *string*. **String-set!** stores *char* in element *k* of *string* and returns an unspecified value.

(string=? $string_1$ $string_2$**)** essential procedure
(string-ci=? $string_1$ $string_2$**)** procedure

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

(string<? *string*$_1$ *string*$_2$**)** essential procedure
(string>? *string*$_1$ *string*$_2$**)** essential procedure
(string<=? *string*$_1$ *string*$_2$**)** essential procedure
(string>=? *string*$_1$ *string*$_2$**)** essential procedure
(string-ci<? *string*$_1$ *string*$_2$**)** procedure
(string-ci>? *string*$_1$ *string*$_2$**)** procedure
(string-ci<=? *string*$_1$ *string*$_2$**)** procedure
(string-ci>=? *string*$_1$ *string*$_2$**)** procedure

These procedures are the lexicographic extensions to strings of the corresponding orderings on characters. For example, **string<?** is the lexicographic ordering on strings induced by the ordering **char<?** on characters. If two strings differ in length but are the same up to the length of the shorter string, the shorter string is considered to be lexicographically less than the longer string.

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

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

String must be a string, and *start* and *end* must be exact integers satisfying

$$0 \leq start \leq end \leq (\texttt{string-length}\ string).$$

Substring returns a newly allocated string formed from the characters of *string* beginning with index *start* (inclusive) and ending with index *end* (exclusive).

(string-append *string*$_1$ *string*$_2$**)** essential procedure
(string-append *string* ...**)** procedure

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

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

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

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

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

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

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

6.8. Vectors

Vectors are heterogenous mutable structures whose elements are indexed by integers.

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

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

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

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

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

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

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

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

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

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

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

```
(vector 'a 'b 'c)           ⟹  #(a b c)
```

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

Returns the number of elements in *vector*.

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

k must be a valid index of *vector*. **Vector-ref** returns the contents of element *k* of *vector*.

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

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

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

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

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

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

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

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

Stores *fill* in every element of *vector*. The value returned by `vector-fill!` is unspecified.

6.9. Control features

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

`(procedure?` *obj*`)` essential procedure

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

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

`(apply` *proc args*`)` essential procedure
`(apply` *proc arg$_1$ ... args*`)` procedure

Proc must be a procedure and *args* must be a list. The first (essential) form calls *proc* with the elements of *args* as the actual arguments. The second form is a generalization of the first that calls *proc* with the elements of the list `(append (list` *arg$_1$* `...)` *args*`)` as the actual arguments.

```
(apply + (list 3 4))        ⟹  7

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

((compose sqrt *) 12 75)    ⟹  30
```

`(map` *proc list*`)` essential procedure
`(map` *proc list$_1$ list$_2$ ...*`)` procedure

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

```
(map cadr '((a b) (d e) (g h)))
          ⟹  (b e h)

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

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

(let ((count 0))
  (map (lambda (ignored)
         (set! count (+ count 1))
         count)
       '(a b c)))           ⟹  unspecified
```

`(for-each` *proc list*`)` essential procedure
`(for-each` *proc list$_1$ list$_2$ ...*`)` procedure

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

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

`(force` *promise*`)` procedure

Forces the value of *promise* (see `delay`, section 4.2.5). If no value has been computed for the promise, then a value is computed and returned. The value of the promise is cached (or "memoized") so that if it is forced a second time, the previously computed value is returned without any recomputation.

```
(force (delay (+ 1 2)))     ⟹  3
(let ((p (delay (+ 1 2))))
  (list (force p) (force p)))
                            ⟹  (3 3)

(define a-stream
  (letrec ((next
            (lambda (n)
              (cons n (delay (next (+ n 1)))))))
    (next 0)))
```

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

(head (tail (tail a-stream)))
                                ⟹  2
```

Force and **delay** are mainly intended for programs written in functional style. The following examples should not be considered to illustrate good programming style, but they illustrate the property that the value of a promise is computed at most once.

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

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

```
(delay ⟨expression⟩)
```

to have the same meaning as the procedure call

```
(make-promise (lambda () ⟨expression⟩)),
```

where `make-promise` is defined as follows:

```
(define make-promise
  (lambda (proc)
    (let ((already-run? #f) (result #f))
      (lambda ()
        (cond ((not already-run?)
               (set! result (proc))
               (set! already-run? #t)))
        result))))
```

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

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

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

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

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

```
(eqv? (delay 1) 1)           ⟹  unspecified
(pair? (delay (cons 1 2)))   ⟹  unspecified
```

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

```
(+ (delay (* 3 7)) 13)       ⟹  34
```

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

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

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

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

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

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

(list-length '(1 2 3 4))        ⟹  4

(list-length '(a b . c))        ⟹  #f
```

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

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

Most programming languages incorporate one or more special-purpose escape constructs with names like `exit`, `return`, or even `goto`. In 1965, however, Peter Landin [21] invented a general purpose escape operator called the J-operator. John Reynolds [32] described a simpler but equally powerful construct in 1972. The `catch` special form described by Sussman and Steele in the 1975 report on Scheme is exactly the same as Reynolds's construct, though its name came from a less general construct in MacLisp. Several Scheme implementors noticed that the full power of the `catch` construct could be provided by a procedure instead of by a special syntactic construct, and the name `call-with-current-continuation` was coined in 1982. This name is descriptive, but opinions differ on the merits of such a long name, and some people use the name `call/cc` instead.

6.10. Input and output

6.10.1. Ports

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

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

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

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

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

Returns `#t` if *obj* is an input port or output port respectively, otherwise returns `#f`.

`(current-input-port)` essential procedure
`(current-output-port)` essential procedure

Returns the current default input or output port.

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

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

`(open-input-file` *filename*`)` 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. These routines have no effect if the file has already been closed. The value returned is unspecified.

6.10.2. Input

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

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

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

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

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

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

(eof-object? *obj***)** essential procedure

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

6.10.3. 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**.

(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. In those implementations that have a distinct character type, character objects appear in the representation as if written by **write-char** instead of by **write**. **Display** returns an unspecified value. The *port* argument may be omitted, in which case it defaults to the value returned by **current-output-port**.

Rationale: **Write** is intended for producing machine-readable output and **display** is for producing human-readable output. Implementations that allow "slashification" within symbols will probably want **write** but not **display** to slashify funny characters in symbols.

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

6.10.4. User interface

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

`(load` *filename*`)` essential procedure

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

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

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

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

7. Formal syntax and semantics

This chapter provides formal descriptions of what has already been described informally in previous chapters of this report.

7.1. Formal syntax

This section provides a formal syntax for Scheme written in an extended BNF. The syntax for the entire language, including features which are not essential, is given here.

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

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

7.1.1. Lexical structure

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

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

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

```
⟨token⟩ ⟶ ⟨identifier⟩ | ⟨boolean⟩ | ⟨number⟩
     | ⟨character⟩ | ⟨string⟩
     | ( | ) | #( | ' | ` | , | ,@ | .
⟨delimiter⟩ ⟶ ⟨whitespace⟩ | ( | ) | " | ;
⟨whitespace⟩ ⟶ ⟨space or newline⟩
⟨comment⟩ ⟶ ; ⟨all subsequent characters up to a
                 line break⟩
⟨atmosphere⟩ ⟶ ⟨whitespace⟩ | ⟨comment⟩
⟨intertoken space⟩ ⟶ ⟨atmosphere⟩*

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

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

⟨variable⟩ ⟶ ⟨any ⟨identifier⟩ which isn't
                    also a ⟨syntactic keyword⟩⟩

⟨boolean⟩ ⟶ #t | #f
⟨character⟩ ⟶ #\ ⟨any character⟩
     | #\ ⟨character name⟩
⟨character name⟩ ⟶ space | newline

⟨string⟩ ⟶ " ⟨string element⟩* "
⟨string element⟩ ⟶ ⟨any character other than " or \⟩
     | \" | \\

⟨number⟩ ⟶ ⟨real⟩ | ⟨real⟩ + ⟨ureal⟩ i
     | ⟨real⟩ - ⟨ureal⟩ i | ⟨real⟩ @ ⟨real⟩
⟨real⟩ ⟶ ⟨sign⟩ ⟨ureal⟩
⟨ureal⟩ ⟶ ⟨ureal 2⟩ | ⟨ureal 8⟩ | ⟨ureal 10⟩ | ⟨ureal 16⟩
```

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

```
⟨ureal R⟩ ⟶ ⟨prefix R⟩ ⟨uinteger R⟩ ⟨suffix⟩
     | ⟨prefix R⟩ ⟨uinteger R⟩ / ⟨uinteger R⟩ ⟨suffix⟩
     | ⟨prefix R⟩ . ⟨digit R⟩+ #* ⟨suffix⟩
     | ⟨prefix R⟩ ⟨digit R⟩+ . ⟨digit R⟩* #* ⟨suffix⟩
     | ⟨prefix R⟩ ⟨digit R⟩+ #* . #* ⟨suffix⟩
⟨prefix R⟩ ⟶ ⟨radix R⟩ ⟨exactness⟩ ⟨precision⟩
     | ⟨radix R⟩ ⟨precision⟩ ⟨exactness⟩
     | ⟨exactness⟩ ⟨radix R⟩ ⟨precision⟩
     | ⟨exactness⟩ ⟨precision⟩ ⟨radix R⟩
     | ⟨precision⟩ ⟨radix R⟩ ⟨exactness⟩
     | ⟨precision⟩ ⟨exactness⟩ ⟨radix R⟩
⟨uinteger R⟩ ⟶ ⟨digit R⟩+ #*

⟨sign⟩ ⟶ ⟨empty⟩ | + | -
⟨suffix⟩ ⟶ ⟨empty⟩ | e ⟨sign⟩ ⟨digit⟩+
⟨exactness⟩ ⟶ ⟨empty⟩ | #i | #e
⟨precision⟩ ⟶ ⟨empty⟩ | #s | #l
⟨radix 2⟩ ⟶ #b
⟨radix 8⟩ ⟶ #o
⟨radix 10⟩ ⟶ ⟨empty⟩ | #d
⟨radix 16⟩ ⟶ #x
⟨digit 2⟩ ⟶ 0 | 1
⟨digit 8⟩ ⟶ 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7
⟨digit 10⟩ ⟶ ⟨digit⟩
⟨digit 16⟩ ⟶ ⟨digit⟩ | a | b | c | d | e | f
```

7.1.2. External representations

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

```
⟨datum⟩ ⟶ ⟨simple datum⟩ | ⟨compound datum⟩
⟨simple datum⟩ ⟶ ⟨boolean⟩ | ⟨number⟩
     | ⟨character⟩ | ⟨string⟩ | ⟨symbol⟩
⟨symbol⟩ ⟶ ⟨identifier⟩
⟨compound datum⟩ ⟶ ⟨list⟩ | ⟨vector⟩
⟨list⟩ ⟶ (⟨datum⟩*) | (⟨datum⟩+ . ⟨datum⟩)
     | ⟨abbreviation⟩
⟨abbreviation⟩ ⟶ ⟨abbrev prefix⟩ ⟨datum⟩
⟨abbrev prefix⟩ ⟶ ' | ` | , | ,@
⟨vector⟩ ⟶ #(⟨datum⟩*)
```

7.1.3. Expressions

```
⟨expression⟩ ⟶ ⟨variable⟩
     | ⟨literal⟩
     | ⟨procedure call⟩
     | ⟨lambda expression⟩
     | ⟨conditional⟩
     | ⟨assignment⟩
     | ⟨derived expression⟩

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

⟨lambda expression⟩ ⟶ (lambda ⟨formals⟩ ⟨body⟩)
⟨formals⟩ ⟶ (⟨variable⟩*) | ⟨variable⟩
     | (⟨variable⟩+ . ⟨variable⟩)
⟨body⟩ ⟶ ⟨definition⟩* ⟨sequence⟩
⟨sequence⟩ ⟶ ⟨command⟩* ⟨expression⟩
⟨command⟩ ⟶ ⟨expression⟩

⟨conditional⟩ ⟶ (if ⟨test⟩ ⟨consequent⟩ ⟨alternate⟩)
⟨test⟩ ⟶ ⟨expression⟩
⟨consequent⟩ ⟶ ⟨expression⟩
⟨alternate⟩ ⟶ ⟨expression⟩ | ⟨empty⟩

⟨assignment⟩ ⟶ (set! ⟨variable⟩ ⟨expression⟩)

⟨derived expression⟩ ⟶
          (cond ⟨cond clause⟩+)
     | (cond ⟨cond clause⟩* (else ⟨sequence⟩))
     | (case ⟨expression⟩
          ⟨case clause⟩+)
     | (case ⟨expression⟩
          ⟨case clause⟩*
          (else ⟨sequence⟩))
     | (and ⟨test⟩*)
     | (or ⟨test⟩*)
     | (let (⟨binding spec⟩*) ⟨body⟩)
     | (let ⟨variable⟩ (⟨binding spec⟩*) ⟨body⟩)
     | (let* (⟨binding spec⟩*) ⟨body⟩)
     | (letrec (⟨binding spec⟩*) ⟨body⟩)
     | (begin ⟨sequence⟩)
     | (do (⟨iteration spec⟩*)
           (⟨test⟩ ⟨sequence⟩)
         ⟨command⟩*)
     | (delay ⟨expression⟩)
     | ⟨quasiquotation⟩

⟨cond clause⟩ ⟶ (⟨test⟩ ⟨sequence⟩)
     | (else ⟨sequence⟩)
     | (⟨test⟩)
     | (⟨test⟩ => ⟨recipient⟩)
⟨recipient⟩ ⟶ ⟨expression⟩
⟨case clause⟩ ⟶ ((⟨datum⟩*) ⟨sequence⟩)
     | (else ⟨sequence⟩)

⟨binding spec⟩ ⟶ (⟨variable⟩ ⟨expression⟩)
⟨iteration spec⟩ ⟶ (⟨variable⟩ ⟨init⟩ ⟨step⟩)
     | (⟨variable⟩ ⟨init⟩)
⟨init⟩ ⟶ ⟨expression⟩
⟨step⟩ ⟶ ⟨expression⟩
```

7.1.4. Quasiquotations

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

```
⟨quasiquotation⟩ ⟶ ⟨quasiquotation 1⟩
⟨template 0⟩ ⟶ ⟨expression⟩
⟨quasiquotation D⟩ ⟶ `⟨template D⟩
     | (quasiquote ⟨template D⟩)
⟨template D⟩ ⟶ ⟨simple datum⟩
     | ⟨list template D⟩
     | ⟨vector template D⟩
     | ⟨unquotation D⟩
⟨list template D⟩ ⟶ (⟨template or splice D⟩*)
     | (⟨template or splice D⟩+ . ⟨template D⟩)
     | '⟨quasiquotation D⟩
     | ⟨quasiquotation D + 1⟩
⟨vector template D⟩ ⟶ #(⟨template or splice D⟩*)
⟨unquotation D⟩ ⟶ ,⟨template D − 1⟩
     | (unquote ⟨template D − 1⟩)
⟨template or splice D⟩ ⟶ ⟨template D⟩
     | ⟨splicing unquotation D⟩
⟨splicing unquotation D⟩ ⟶ ,@⟨template D − 1⟩
     | (unquote-splicing ⟨template D − 1⟩)
```

In ⟨quasiquotation⟩s, a ⟨list template D⟩ can sometimes be confused with an ⟨unquotation D⟩ or ⟨splicing unquotation D⟩. The interpretation as an ⟨unquotation⟩ or ⟨splicing unquotation D⟩ takes precedence.

7.1.5. Programs and definitions

```
⟨program⟩ ⟶ ⟨command or definition⟩*
⟨command or definition⟩ ⟶ ⟨command⟩ | ⟨definition⟩
⟨definition⟩ ⟶ (define ⟨variable⟩ ⟨expression⟩)
     | (define (⟨variable⟩ ⟨def formals⟩) ⟨body⟩)
⟨def formals⟩ ⟶ ⟨variable⟩*
     | ⟨variable⟩* . ⟨variable⟩
```

7.2. Formal semantics

This section provides a formal denotational semantics for the primitive expressions of Scheme and selected built-in procedures. The concepts and notation used here are described in [50]; the notation is summarized below:

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

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

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

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

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

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

7.2.1. Abstract syntax

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

```
Exp ⟶ K | I | (E₀ E*)
       | (lambda (I*) Γ* E₀)
       | (lambda (I* . I) Γ* E₀)
```

$\mid$ **(lambda** I Γ^* E_0**)**
$\mid$ **(if** E_0 E_1 E_2**)** $\mid$ **(if** E_0 E_1**)**
$\mid$ **(set!** I E**)**

7.2.2. Domain equations

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

7.2.3. Semantic functions

$$\begin{array}{l}
\mathcal{K} : \mathrm{Con} \to \mathtt{E} \\
\mathcal{E} : \mathrm{Exp} \to \mathtt{U} \to \mathtt{K} \to \mathtt{C} \\
\mathcal{E}^* : \mathrm{Exp}^* \to \mathtt{U} \to \mathtt{K} \to \mathtt{C} \\
\mathcal{C} : \mathrm{Com}^* \to \mathtt{U} \to \mathtt{C} \to \mathtt{C}
\end{array}$$

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

$$\mathcal{E}[\![\mathrm{K}]\!] = \lambda\rho\kappa\,.\,send\,(\mathcal{K}[\![\mathrm{K}]\!])\,\kappa$$

$$\begin{array}{l}
\mathcal{E}[\![\mathrm{I}]\!] = \lambda\rho\kappa\,.\,hold\,(lookup\,\rho\,\mathrm{I}) \\
\qquad\qquad (single(\lambda\epsilon\,.\,\epsilon = undefined \to \\
\qquad\qquad\qquad\qquad wrong\ \text{“undefined variable”}, \\
\qquad\qquad\qquad send\,\epsilon\,\kappa))
\end{array}$$

$$\begin{array}{l}
\mathcal{E}[\![\texttt{(}\mathrm{E_0}\ \mathrm{E}^*\texttt{)}]\!] = \\
\quad \lambda\rho\kappa\,.\,\mathcal{E}^*(permute(\langle \mathrm{E_0}\rangle\ \S\ \mathrm{E}^*)) \\
\qquad\qquad \rho \\
\qquad\qquad (\lambda\epsilon^*\,.\,((\lambda\epsilon^*\,.\,applicate\,(\epsilon^* \downarrow 1)\,(\epsilon^* \dagger 1)\,\kappa) \\
\qquad\qquad\qquad (unpermute\ \epsilon^*)))
\end{array}$$

$$\begin{array}{l}
\mathcal{E}[\![\texttt{(lambda (}\mathrm{I}^*\texttt{)}\ \Gamma^*\ \mathrm{E_0}\texttt{)}]\!] = \\
\quad \lambda\rho\kappa\,.\,\lambda\sigma\,. \\
\qquad new\,\sigma \in \mathtt{L} \to \\
\qquad\quad send\,(\langle new\,\sigma \mid \mathtt{L}, \\
\qquad\qquad\quad \lambda\epsilon^*\kappa'\,.\,\#\epsilon^* = \#\mathrm{I}^* \to \\
\qquad\qquad\qquad tievals(\lambda\alpha^*\,.\,(\lambda\rho'\,.\,\mathcal{C}[\![\Gamma^*]\!]\rho'(\mathcal{E}[\![\mathrm{E_0}]\!]\rho'\kappa')) \\
\qquad\qquad\qquad\qquad\qquad (extends\,\rho\,\mathrm{I}^*\,\alpha^*)) \\
\qquad\qquad\qquad\qquad \epsilon^*, \\
\qquad\qquad\qquad wrong\ \text{“wrong number of arguments”}\rangle \\
\qquad\qquad \text{in } \mathtt{E}) \\
\qquad\quad \kappa \\
\qquad\quad (update\,(new\,\sigma \mid \mathtt{L})\ unspecified\ \sigma), \\
\qquad wrong\ \text{“out of memory”}\ \sigma
\end{array}$$

$$\begin{array}{l}
\mathcal{E}[\![\texttt{(lambda (}\mathrm{I}^*\ \texttt{.}\ \mathrm{I}\texttt{)}\ \Gamma^*\ \mathrm{E_0}\texttt{)}]\!] = \\
\quad \lambda\rho\kappa\,.\,\lambda\sigma\,. \\
\qquad new\,\sigma \in \mathtt{L} \to \\
\qquad\quad send\,(\langle new\,\sigma \mid \mathtt{L}, \\
\qquad\qquad\quad \lambda\epsilon^*\kappa'\,.\,\#\epsilon^* \geq \#\mathrm{I}^* \to \\
\qquad\qquad\qquad tievalsrest \\
\qquad\qquad\qquad\quad (\lambda\alpha^*\,.\,(\lambda\rho'\,.\,\mathcal{C}[\![\Gamma^*]\!]\rho'(\mathcal{E}[\![\mathrm{E_0}]\!]\rho'\kappa')) \\
\qquad\qquad\qquad\qquad\quad (extends\,\rho\,(\mathrm{I}^*\ \S\ \langle\mathrm{I}'\rangle)\,\alpha^*)) \\
\qquad\qquad\qquad\quad \epsilon^* \\
\qquad\qquad\qquad\quad (\#\mathrm{I}^*), \\
\qquad\qquad\qquad wrong\ \text{“too few arguments”}\rangle\ \text{in } \mathtt{E}) \\
\qquad\quad \kappa \\
\qquad\quad (update\,(new\,\sigma \mid \mathtt{L})\ unspecified\ \sigma), \\
\qquad wrong\ \text{“out of memory”}\ \sigma
\end{array}$$

$$\mathcal{E}[\![\texttt{(lambda I}\ \Gamma^*\ \mathrm{E_0}\texttt{)}]\!] = \mathcal{E}[\![\texttt{(lambda (. I)}\ \Gamma^*\ \mathrm{E_0}\texttt{)}]\!]$$

$$\begin{array}{l}
\mathcal{E}[\![\texttt{(if}\ \mathrm{E_0}\ \mathrm{E_1}\ \mathrm{E_2}\texttt{)}]\!] = \\
\quad \lambda\rho\kappa\,.\,\mathcal{E}[\![\mathrm{E_0}]\!]\,\rho\,(single\,(\lambda\epsilon\,.\,truish\,\epsilon \to \mathcal{E}[\![\mathrm{E_1}]\!]\rho\kappa, \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathcal{E}[\![\mathrm{E_2}]\!]\rho\kappa))
\end{array}$$

$$\begin{array}{l}
\mathcal{E}[\![\texttt{(if}\ \mathrm{E_0}\ \mathrm{E_1}\texttt{)}]\!] = \\
\quad \lambda\rho\kappa\,.\,\mathcal{E}[\![\mathrm{E_0}]\!]\,\rho\,(single\,(\lambda\epsilon\,.\,truish\,\epsilon \to \mathcal{E}[\![\mathrm{E_1}]\!]\rho\kappa, \\
\qquad\qquad\qquad\qquad\qquad\qquad send\ unspecified\ \kappa))
\end{array}$$

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

$$\begin{array}{l}
\mathcal{E}[\![\texttt{(set!}\ \mathrm{I}\ \mathrm{E}\texttt{)}]\!] = \\
\quad \lambda\rho\kappa\,.\,\mathcal{E}[\![\mathrm{E}]\!]\,\rho\,(single(\lambda\epsilon\,.\,assign\,(lookup\,\rho\,\mathrm{I}) \\
\qquad\qquad\qquad\qquad\qquad\qquad \epsilon \\
\qquad\qquad\qquad\qquad\qquad\qquad (send\ unspecified\ \kappa)))
\end{array}$$

$$\mathcal{E}^*[\![\,]\!] = \lambda\rho\kappa\,.\,\kappa\langle\,\rangle$$

$$\begin{array}{l}
\mathcal{E}^*[\![\mathrm{E_0}\ \mathrm{E}^*]\!] = \\
\quad \lambda\rho\kappa\,.\,\mathcal{E}[\![\mathrm{E_0}]\!]\,\rho\,(single(\lambda\epsilon_0\,.\,\mathcal{E}^*[\![\mathrm{E}^*]\!]\,\rho\,(\lambda\epsilon^*\,.\,\kappa\,(\langle\epsilon_0\rangle\ \S\ \epsilon^*))))
\end{array}$$

$$\mathcal{C}[\![\,]\!] = \lambda\rho\theta\,.\,\theta$$

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

7.2.4. Auxiliary functions

$$\begin{array}{l}
lookup : \mathtt{U} \to \mathrm{Ide} \to \mathtt{L} \\
lookup = \lambda\rho\mathrm{I}\,.\,\rho\mathrm{I}
\end{array}$$

$$\begin{array}{l}
extends : \mathtt{U} \to \mathrm{Ide}^* \to \mathtt{L}^* \to \mathtt{U} \\
extends = \\
\quad \lambda\rho\mathrm{I}^*\alpha^*\,.\,\#\mathrm{I}^* = 0 \to \rho, \\
\qquad\qquad extends\,(\rho[(\alpha^* \downarrow 1)/(\mathrm{I}^* \downarrow 1)])\,(\mathrm{I}^* \dagger 1)\,(\alpha^* \dagger 1)
\end{array}$$

$wrong : \mathtt{X} \to \mathtt{C}$ [implementation-dependent]

$$\begin{array}{l}
send : \mathtt{E} \to \mathtt{K} \to \mathtt{C} \\
send = \lambda\epsilon\kappa\,.\,\kappa\langle\epsilon\rangle
\end{array}$$

$$\begin{array}{l}
single : (\mathtt{E} \to \mathtt{C}) \to \mathtt{K} \\
single = \\
\quad \lambda\psi\epsilon^*\,.\,\#\epsilon^* = 1 \to \psi(\epsilon^* \downarrow 1), \\
\qquad\qquad wrong\ \text{“wrong number of return values”}
\end{array}$$

$\mathit{new} : \mathtt{S} \to (\mathtt{L} + \{\mathit{error}\})$ [implementation-dependent]

$$\begin{array}{l}
\mathit{hold} : \mathtt{L} \to \mathtt{K} \to \mathtt{C} \\
\mathit{hold} = \lambda\alpha\kappa\sigma \,.\, \mathit{send}(\sigma\alpha \downarrow 1)\kappa\sigma
\end{array}$$

$$\begin{array}{l}
\mathit{assign} : \mathtt{L} \to \mathtt{E} \to \mathtt{C} \to \mathtt{C} \\
\mathit{assign} = \lambda\alpha\epsilon\theta\sigma \,.\, \theta(\mathit{update}\ \alpha\epsilon\sigma)
\end{array}$$

$$\begin{array}{l}
\mathit{update} : \mathtt{L} \to \mathtt{E} \to \mathtt{S} \to \mathtt{S} \\
\mathit{update} = \lambda\alpha\epsilon\sigma \,.\, \sigma[\langle\epsilon, \mathit{true}\rangle/\alpha]
\end{array}$$

$$\begin{array}{l}
\mathit{tievals} : (\mathtt{L}^* \to \mathtt{C}) \to \mathtt{E}^* \to \mathtt{C} \\
\mathit{tievals} = \\
\quad \lambda\psi\epsilon^*\sigma \,.\, \#\epsilon^* = 0 \to \psi\langle\,\rangle\sigma, \\
\qquad\qquad \mathit{new}\,\sigma \in \mathtt{L} \to \mathit{tievals}\,(\lambda\alpha^* \,.\, \psi(\langle \mathit{new}\,\sigma \mid \mathtt{L}\rangle \mathbin{\S} \alpha^*)) \\
\qquad\qquad\qquad\qquad\qquad\qquad (\epsilon^* \dagger 1) \\
\qquad\qquad\qquad\qquad\qquad\qquad (\mathit{update}(\mathit{new}\,\sigma \mid \mathtt{L})(\epsilon^* \downarrow 1)\sigma), \\
\qquad\qquad \mathit{wrong}\ \text{“out of memory”}\sigma
\end{array}$$

$$\begin{array}{l}
\mathit{tievalsrest} : (\mathtt{L}^* \to \mathtt{C}) \to \mathtt{E}^* \to \mathtt{N} \to \mathtt{C} \\
\mathit{tievalsrest} = \\
\quad \lambda\psi\epsilon^*\nu \,.\, \mathit{list}\,(\mathit{dropfirst}\ \epsilon^*\nu) \\
\qquad\qquad (\mathit{single}(\lambda\epsilon \,.\, \mathit{tievals}\ \psi\ ((\mathit{takefirst}\ \epsilon^*\nu) \mathbin{\S} \langle\epsilon\rangle)))
\end{array}$$

$$\mathit{dropfirst} = \lambda l n \,.\, n = 0 \to l,\ \mathit{dropfirst}\,(l \dagger 1)(n-1)$$

$$\mathit{takefirst} = \lambda l n \,.\, n = 0 \to \langle\,\rangle, \langle l \downarrow 1\rangle \mathbin{\S} (\mathit{takefirst}\,(l \dagger 1)(n-1))$$

$$\begin{array}{l}
\mathit{truish} : \mathtt{E} \to \mathtt{T} \\
\mathit{truish} = \lambda\epsilon \,.\, (\epsilon = \mathit{false} \lor \epsilon = \mathit{null}) \to \mathit{false}, \mathit{true}
\end{array}$$

$\mathit{permute} : \mathrm{Exp}^* \to \mathrm{Exp}^*$ [implementation-dependent]

$\mathit{unpermute} : \mathtt{E}^* \to \mathtt{E}^*$ [inverse of *permute*]

$$\begin{array}{l}
\mathit{applicate} : \mathtt{E} \to \mathtt{E}^* \to \mathtt{K} \to \mathtt{C} \\
\mathit{applicate} = \\
\quad \lambda\epsilon\epsilon^*\kappa \,.\, \epsilon \in \mathtt{F} \to (\epsilon \mid \mathtt{F} \downarrow 2)\epsilon^*\kappa,\ \mathit{wrong}\ \text{“bad procedure”}
\end{array}$$

$$\begin{array}{l}
\mathit{onearg} : (\mathtt{E} \to \mathtt{K} \to \mathtt{C}) \to (\mathtt{E}^* \to \mathtt{K} \to \mathtt{C}) \\
\mathit{onearg} = \\
\quad \lambda\zeta\epsilon^*\kappa \,.\, \#\epsilon^* = 1 \to \zeta(\epsilon^* \downarrow 1)\kappa, \\
\qquad\qquad \mathit{wrong}\ \text{“wrong number of arguments”}
\end{array}$$

$$\begin{array}{l}
\mathit{twoarg} : (\mathtt{E} \to \mathtt{E} \to \mathtt{K} \to \mathtt{C}) \to (\mathtt{E}^* \to \mathtt{K} \to \mathtt{C}) \\
\mathit{twoarg} = \\
\quad \lambda\zeta\epsilon^*\kappa \,.\, \#\epsilon^* = 2 \to \zeta(\epsilon^* \downarrow 1)(\epsilon^* \downarrow 2)\kappa, \\
\qquad\qquad \mathit{wrong}\ \text{“wrong number of arguments”}
\end{array}$$

$$\begin{array}{l}
\mathit{list} : \mathtt{E}^* \to \mathtt{K} \to \mathtt{C} \\
\mathit{list} = \\
\quad \lambda\epsilon^*\kappa \,.\, \#\epsilon^* = 0 \to \mathit{send}\ \mathit{null}\ \kappa, \\
\qquad\qquad \mathit{list}\,(\epsilon^* \dagger 1)(\mathit{single}(\lambda\epsilon \,.\, \mathit{cons}\langle\epsilon^* \downarrow 1, \epsilon\rangle\kappa))
\end{array}$$

$$\begin{array}{l}
\mathit{cons} : \mathtt{E}^* \to \mathtt{K} \to \mathtt{C} \\
\mathit{cons} = \\
\quad \mathit{twoarg}\,(\lambda\epsilon_1\epsilon_2\kappa\sigma \,.\, \mathit{new}\,\sigma \in \mathtt{L} \to \\
\qquad\qquad\qquad (\lambda\sigma' \,.\, \mathit{new}\,\sigma' \in \mathtt{L} \to \\
\qquad\qquad\qquad\qquad \mathit{send}(\langle \mathit{new}\,\sigma \mid \mathtt{L}, \mathit{new}\,\sigma' \mid \mathtt{L}\rangle \text{ in } \mathtt{E}) \\
\qquad\qquad\qquad\qquad\quad \kappa \\
\qquad\qquad\qquad\qquad\quad (\mathit{update}(\mathit{new}\,\sigma' \mid \mathtt{L})\epsilon_2\sigma'), \\
\qquad\qquad\qquad\qquad \mathit{wrong}\ \text{“out of memory”}\sigma') \\
\qquad\qquad\qquad (\mathit{update}(\mathit{new}\,\sigma \mid \mathtt{L})\epsilon_1\sigma), \\
\qquad\qquad\qquad \mathit{wrong}\ \text{“out of memory”}\sigma)
\end{array}$$

$$\begin{array}{l}
\mathit{less} : \mathtt{E}^* \to \mathtt{K} \to \mathtt{C} \\
\mathit{less} = \\
\quad \mathit{twoarg}\,(\lambda\epsilon_1\epsilon_2\kappa \,.\, (\epsilon_1 \in \mathtt{R} \land \epsilon_2 \in \mathtt{R}) \to \\
\qquad\qquad\qquad \mathit{send}(\epsilon_1 \mid \mathtt{R} < \epsilon_2 \mid \mathtt{R} \to \mathit{true}, \mathit{false})\kappa, \\
\qquad\qquad\qquad \mathit{wrong}\ \text{“non-numeric argument to <”})
\end{array}$$

$$\begin{array}{l}
\mathit{add} : \mathtt{E}^* \to \mathtt{K} \to \mathtt{C} \\
\mathit{add} = \\
\quad \mathit{twoarg}\,(\lambda\epsilon_1\epsilon_2\kappa \,.\, (\epsilon_1 \in \mathtt{R} \land \epsilon_2 \in \mathtt{R}) \to \\
\qquad\qquad\qquad \mathit{send}((\epsilon_1 \mid \mathtt{R} + \epsilon_2 \mid \mathtt{R}) \text{ in } \mathtt{E})\kappa, \\
\qquad\qquad\qquad \mathit{wrong}\ \text{“non-numeric argument to +”})
\end{array}$$

$$\begin{array}{l}
\mathit{car} : \mathtt{E}^* \to \mathtt{K} \to \mathtt{C} \\
\mathit{car} = \\
\quad \mathit{onearg}\,(\lambda\epsilon\kappa \,.\, \epsilon \in \mathtt{E_p} \to \mathit{hold}(\epsilon \mid \mathtt{E_p} \downarrow 1)\kappa, \\
\qquad\qquad\qquad \mathit{wrong}\ \text{“non-pair argument to }\mathtt{car}\text{”})
\end{array}$$

$\mathit{cdr} : \mathtt{E}^* \to \mathtt{K} \to \mathtt{C}$ [similar to *car*]

$$\begin{array}{l}
\mathit{setcar} : \mathtt{E}^* \to \mathtt{K} \to \mathtt{C} \\
\mathit{setcar} = \\
\quad \mathit{twoarg}\,(\lambda\epsilon_1\epsilon_2\kappa \,.\, \epsilon_1 \in \mathtt{E_p} \to \mathit{assign}(\epsilon_1 \mid \mathtt{E_p} \downarrow 1) \\
\qquad\qquad\qquad\qquad\qquad\qquad \epsilon_2 \\
\qquad\qquad\qquad\qquad\qquad\qquad (\mathit{send}\ \mathit{unspecified}\ \kappa), \\
\qquad\qquad\qquad \mathit{wrong}\ \text{“non-pair argument to }\mathtt{set\text{-}car!}\text{”})
\end{array}$$

$$\begin{array}{l}
\mathit{eqv} : \mathtt{E}^* \to \mathtt{K} \to \mathtt{C} \\
\mathit{eqv} = \\
\quad \mathit{twoarg}\,(\lambda\epsilon_1\epsilon_2\kappa \,.\, (\epsilon_1 \in \mathtt{M} \land \epsilon_2 \in \mathtt{M}) \to \\
\qquad\qquad\qquad \mathit{send}(\epsilon_1 \mid \mathtt{M} = \epsilon_2 \mid \mathtt{M} \to \mathit{true}, \mathit{false})\kappa, \\
\qquad\qquad (\epsilon_1 \in \mathtt{Q} \land \epsilon_2 \in \mathtt{Q}) \to \\
\qquad\qquad\qquad \mathit{send}(\epsilon_1 \mid \mathtt{Q} = \epsilon_2 \mid \mathtt{Q} \to \mathit{true}, \mathit{false})\kappa, \\
\qquad\qquad (\epsilon_1 \in \mathtt{H} \land \epsilon_2 \in \mathtt{H}) \to \\
\qquad\qquad\qquad \mathit{send}(\epsilon_1 \mid \mathtt{H} = \epsilon_2 \mid \mathtt{H} \to \mathit{true}, \mathit{false})\kappa, \\
\qquad\qquad (\epsilon_1 \in \mathtt{R} \land \epsilon_2 \in \mathtt{R}) \to \\
\qquad\qquad\qquad \mathit{send}(\epsilon_1 \mid \mathtt{R} = \epsilon_2 \mid \mathtt{R} \to \mathit{true}, \mathit{false})\kappa, \\
\qquad\qquad (\epsilon_1 \in \mathtt{E_p} \land \epsilon_2 \in \mathtt{E_p}) \to \\
\qquad\qquad\qquad \mathit{send}((\lambda p_1 p_2 \,.\, ((p_1 \downarrow 1) = (p_2 \downarrow 1) \land \\
\qquad\qquad\qquad\qquad\qquad\qquad (p_1 \downarrow 2) = (p_2 \downarrow 2)) \to \mathit{true}, \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{false}) \\
\qquad\qquad\qquad\qquad (\epsilon_1 \mid \mathtt{E_p}) \\
\qquad\qquad\qquad\qquad (\epsilon_2 \mid \mathtt{E_p})) \\
\qquad\qquad\qquad\quad \kappa, \\
\qquad\qquad (\epsilon_1 \in \mathtt{E_v} \land \epsilon_2 \in \mathtt{E_v}) \to \ldots, \\
\qquad\qquad (\epsilon_1 \in \mathtt{E_s} \land \epsilon_2 \in \mathtt{E_s}) \to \ldots, \\
\qquad\qquad (\epsilon_1 \in \mathtt{F} \land \epsilon_2 \in \mathtt{F}) \to \\
\qquad\qquad\qquad \mathit{send}((\epsilon_1 \mid \mathtt{F} \downarrow 1) = (\epsilon_2 \mid \mathtt{F} \downarrow 1) \to \mathit{true}, \mathit{false}) \\
\qquad\qquad\qquad\quad \kappa, \\
\qquad\qquad\qquad \mathit{send}\ \mathit{false}\ \kappa)
\end{array}$$

$$\begin{array}{l}
\mathit{apply} : \mathtt{E}^* \to \mathtt{K} \to \mathtt{C} \\
\mathit{apply} = \\
\quad \mathit{twoarg}\,(\lambda\epsilon_1\epsilon_2\kappa \,.\, \epsilon_1 \in \mathtt{F} \to \mathit{valueslist}\,\langle\epsilon_2\rangle(\lambda\epsilon^* \,.\, \mathit{applicate}\ \epsilon_1\epsilon^*\kappa), \\
\qquad\qquad\qquad \mathit{wrong}\ \text{“bad procedure argument to }\mathtt{apply}\text{”})
\end{array}$$

$$\begin{array}{l}
\mathit{valueslist} : \mathtt{E}^* \to \mathtt{K} \to \mathtt{C} \\
\mathit{valueslist} = \\
\quad \mathit{onearg}\,(\lambda\epsilon\kappa \,.\, \epsilon \in \mathtt{E_p} \to \\
\qquad\qquad\qquad \mathit{cdr}\langle\epsilon\rangle \\
\qquad\qquad\qquad\quad (\lambda\epsilon^* \,.\, \mathit{valueslist} \\
\qquad\qquad\qquad\qquad\quad \epsilon^* \\
\qquad\qquad\qquad\qquad\quad (\lambda\epsilon^* \,.\, \mathit{car}\langle\epsilon\rangle(\mathit{single}(\lambda\epsilon \,.\, \kappa(\langle\epsilon\rangle \mathbin{\S} \epsilon^*))))), \\
\qquad\qquad \epsilon = \mathit{null} \to \kappa\langle\,\rangle, \\
\qquad\qquad\quad \mathit{wrong}\ \text{“non-list argument to }\mathtt{values\text{-}list}\text{”})
\end{array}$$

```
$cwcc : \mathrm{E}^* \to \mathrm{K} \to \mathrm{C}$     [call-with-current-continuation]
$cwcc =$
  $onearg(\lambda \epsilon \kappa \,.\, \epsilon \in \mathrm{F} \to$
                 $(\lambda \sigma \,.\, new\, \sigma \in \mathrm{L} \to$
                       $applicate\ \epsilon$
                                $\langle\langle new\, \sigma \mid \mathrm{L}, \lambda \epsilon^* \kappa' \,.\, \kappa \epsilon^* \rangle$ in $\mathrm{E}\rangle$
                                $\kappa$
                                $(update\, (new\, \sigma \mid \mathrm{L})$
                                        $unspecified$
                                        $\sigma),$
                       $wrong$ "out of memory" $\sigma),$
                 $wrong$ "bad procedure argument")
```

7.3. Derived expression types

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

```
(cond (⟨test⟩ ⟨sequence⟩)
      ⟨clause₂⟩ ...)
  ≡  (if ⟨test⟩
         (begin ⟨sequence⟩)
         (cond ⟨clause₂⟩ ...))

(cond (⟨test⟩)
      ⟨clause₂⟩ ...)
  ≡  (or ⟨test⟩ (cond ⟨clause₂⟩ ...))

(cond (⟨test⟩ => ⟨recipient⟩)
      ⟨clause₂⟩ ...)
  ≡  (let ((test-result ⟨test⟩)
           (thunk2 (lambda () ⟨recipient⟩))
           (thunk3 (lambda () (cond ⟨clause₂⟩ ...))))
       (if test-result
           ((thunk2) test-result)
           (thunk3)))

(cond (else ⟨sequence⟩))
  ≡  (begin ⟨sequence⟩)

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

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

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

where ⟨memv⟩ is an expression evaluating to the `memv` procedure.

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

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

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

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

(letrec ((⟨variable₁⟩ ⟨init₁⟩)
         ...)
  ⟨body⟩)
  ≡  (let ((⟨variable₁⟩ ⟨undefined⟩)
           ...)
       (let ((⟨temp₁⟩ ⟨init₁⟩)
             ...)
         (set! ⟨variable₁⟩ ⟨temp₁⟩)
         ...)
       ⟨body⟩)
```

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

```
(begin ⟨sequence⟩)
  ≡  ((lambda () ⟨sequence⟩))
```

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

```
(begin ⟨expression⟩)  ≡  ⟨expression⟩
(begin ⟨command⟩ ⟨sequence⟩)
  ≡  ((lambda (ignore thunk) (thunk))
      ⟨command⟩
      (lambda () (begin ⟨sequence⟩)))

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

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

```
(let ⟨variable₀⟩ ((⟨variable₁⟩ ⟨init₁⟩) ...)
  ⟨body⟩)
  ≡  ((letrec ((⟨variable₀⟩ (lambda (⟨variable₁⟩ ...)
                               ⟨body⟩)))
        ⟨variable₀⟩)
      ⟨init₁⟩ ...)

(delay ⟨expression⟩)
  ≡  (⟨make-promise⟩ (lambda () ⟨expression⟩))
```

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

NOTES

Language changes

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

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

Keywords as variable names

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

```
(define if list)
(if 1 2 3)                  ⟹  2 or (1 2 3)
```

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

Macros

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

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

The main problems with traditional macros are: They must be defined to the system before any code using them is loaded; this is a common source of obscure bugs. They are usually global; macros can be made to follow lexical scope rules , but many people find the resulting scope rules confusing. Unless they are written very carefully, macros are vulnerable to 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 syntactic keywords if the keywords of special forms are not reserved. If keywords are reserved, then either macros introduce new reserved words, invalidating old code, or else special forms defined by the programmer do not have the same status as special forms defined by the system.

EXAMPLE

`Integrate-system` integrates the system

$$y_k' = f_k(y_1, y_2, \ldots, y_n),\ k = 1, \ldots, n$$

of differential equations with the method of Runge-Kutta.

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

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

```
(define integrate-system
  (lambda (system-derivative initial-state h)
    (let ((next (runge-kutta-4 system-derivative h)))
      (letrec ((states
                (cons initial-state
                      (delay (map-streams next
                                          states)))))
        states))))
```

`Runge-Kutta-4` takes a function, `f`, that produces a system derivative from a system state. `Runge-Kutta-4` produces a function that takes a system state and produces a new system state.

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

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

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

(define add-vectors (elementwise +))

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

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

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

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

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

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

$$C\frac{dv_C}{dt} = -i_L - \frac{v_C}{R}$$

$$L\frac{di_L}{dt} = v_C$$

which models a damped oscillator.

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

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

BIBLIOGRAPHY AND REFERENCES

[1] Harold Abelson and Gerald Jay Sussman with Julie Sussman. *Structure and Interpretation of Computer Programs.* MIT Press, Cambridge, 1985.

[2] David H. Bartley and John C. Jensen. The implementation of PC Scheme. In *Proceedings of the 1986 ACM Conference on Lisp and Functional Programming*, pages 86–93.

[3] John Batali, Edmund Goodhue, Chris Hanson, Howie Shrobe, Richard M. Stallman, and Gerald Jay Sussman. The Scheme-81 architecture—system and chip. In *Proceedings, Conference on Advanced Research in VLSI*, pages 69–77. Paul Penfield, Jr., editor. Artech House, 610 Washington Street, Dedham MA, 1982.

[4] William Clinger, editor. The revised revised report on Scheme, or an uncommon Lisp. MIT Artificial Intelligence Memo 848, August 1985. Also published as Computer Science Department Technical Report 174, Indiana University, June 1985.

[5] William Clinger. The Scheme 311 compiler: An exercise in denotational semantics. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 356–364.

[6] R. Kent Dybvig, Daniel P. Friedman, and Christopher T. Haynes. Expansion-passing style: Beyond conventional macros. In *Proceedings of the 1986 ACM Conference on Lisp and Functional Programming*, pages 143–150.

[7] Michael A. Eisenberg. Bochser: an integrated Scheme programming system. MIT Laboratory for Computer Science Technical Report 349, October 1985.

[8] Marc Feeley. Deux approches à l'implantation du language Scheme. M.Sc. thesis, Département d'Informatique et de Recherche Opérationelle, University of Montreal, May 1986.

[9] Matthias Felleisen, Daniel P. Friedman, Eugene Kohlbecker, and Bruce Duba. Reasoning with continuations. In *Proceedings of the Symposium on Logic in Computer Science*, pages 131–141. IEEE Computer Society Press, Washigton DC, 1986.

[10] 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. Superceded by [13].

[11] Daniel P. Friedman and Matthias Felleisen. *The Little LISPer.* Science Research Associates, second edition 1986.

[12] Daniel P. Friedman, Christopher T. Haynes, and Eugene Kohlbecker. Programming with continuations. In *Program Transformation and Programming Environments,* pages 263–274. P. Pepper, editor. Springer-Verlag, 1984.

[13] D. Friedman, C. Haynes, E. Kohlbecker, and M. Wand. Scheme 84 interim reference manual. Indiana University Computer Science Technical Report 153, January 1985.

[14] Daniel P. Friedman and Christopher T. Haynes. Constraining control. In *Proceedings of the Twelfth Annual Symposium on Principles of Programming Languages,* pages 245–254. ACM, January 1985.

[15] 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,* pages 293–298.

[16] Christopher T. Haynes. Logic continuations. In *Proceedings of the Third International Conference on Logic Programming,* pages 671–685. Springer-Verlag, July 1986.

[17] Christopher T. Haynes and Daniel P. Friedman. Engines build process abstracions. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming,* pages 18–24.

[18] Peter Henderson. Functional Geometry. In *Conference Record of the 1982 ACM Symposium on Lisp and Functional Programming,* pages 179–187.

[19] Eugene Edmund Kohlbecker Jr. *Syntactic Extensions in the Programming Language Lisp.* PhD thesis, Indiana University, August 1986.

[20] David Kranz, Richard Kelsey, Jonathan Rees, Paul Hudak, James Philbin, and Norman Adams. Orbit: An optimizing compiler for Scheme. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction,* pages 219–233. ACM, June 1986. Proceedings published as *SIGPLAN Notices* 21(7), July 1986.

[21] Peter Landin. A correspondence between Algol 60 and Church's lambda notation: Part I. *Communications of the ACM* 8(2):89–101, February 1965.

[22] Drew McDermott. An efficient environment allocation scheme in an interpreter for a lexically-scoped lisp. In *Conference Record of the 1980 Lisp Conference,* pages 154–162. The Lisp Conference, P.O. Box 487, Redwood Estates CA, 1980. Proceedings reprinted by ACM.

[23] MIT Department of Electrical Engineering and Computer Science. Scheme manual, seventh edition. September 1984.

[24] Steven S. Muchnick and Uwe F. Pleban. A semantic comparison of Lisp and Scheme. In *Conference Record of the 1980 Lisp Conference,* pages 56–64. The Lisp Conference, 1980.

[25] Peter Naur et al. Revised report on the algorithmic language Algol 60. *Communications of the ACM* 6(1):1–17, January 1963.

[26] Paul Penfield, Jr. Principal values and branch cuts in complex APL. In *APL '81 Conference Proceedings,* pages 248–256. ACM SIGAPL, San Francisco, September 1981. Proceedings published as *APL Quote Quad* 12(1), ACM, September 1981.

[27] Kent M. Pitman. Exceptional situations in Lisp. MIT Artificial Intelligence Laboratory Working Paper 268, February 1985.

[28] Kent M. Pitman. The revised MacLisp manual (saturday evening edition). MIT Artificial Intelligence Laboratory Technical Report 295, May 1983.

[29] Kent M. Pitman. Special forms in Lisp. In *Conference Record of the 1980 Lisp Conference,* pages 179–187. The Lisp Conference, 1980.

[30] Jonathan A. Rees and Norman I. Adams IV. T: A dialect of Lisp or, lambda: The ultimate software tool. In *Conference Record of the 1982 ACM Symposium on Lisp and Functional Programming,* pages 114–122.

[31] Jonathan A. Rees, Norman I. Adams IV, and James R. Meehan. The T manual, fourth edition. Yale University Computer Science Department, January 1984.

[32] John Reynolds. Definitional interpreters for higher order programming languages. In *ACM Conference Proceedings,* pages 717–740. ACM, 1972.

[33] Guillermo J. Rozas. Liar, an Algol-like compiler for Scheme. S. B. thesis, MIT Department of Electrical Engineering and Computer Science, January 1984.

[34] Brian C. Smith. Reflection and semantics in a procedural language. MIT Laboratory for Computer Science Technical Report 272, January 1982.

[35] Amitabh Srivastava, Don Oxley, and Aditya Srivastava. An(other) integration of logic and functional programming. In *Proceedings of the Symposium on Logic Programming,* pages 254–260. IEEE, 1985.

[36] Richard M. Stallman. Phantom stacks—if you look too hard, they aren't there. MIT Artificial Intelligence Memo 556, July 1980.

[37] Guy Lewis Steele Jr. and Gerald Jay Sussman. Lambda, the ultimate imperative. MIT Artificial Intelligence Memo 353, March 1976.

[38] Guy Lewis Steele Jr. Lambda, the ultimate declarative. MIT Artificial Intelligence Memo 379, November 1976.

[39] 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*, pages 153–162. ACM, 1977.

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

[41] Guy Lewis Steele Jr. Rabbit: a compiler for Scheme. MIT Artificial Intelligence Laboratory Technical Report 474, May 1978.

[42] Guy Lewis Steele Jr. An overview of Common Lisp. In *Conference Record of the 1982 ACM Symposium on Lisp and Functional Programming*, pages 98–107.

[43] Guy Lewis Steele, Jr. *Common Lisp: The Language*. Digital Press, Burlington MA, 1984.

[44] Guy Lewis Steele, Jr. and Gerald Jay Sussman. The revised report on Scheme, a dialect of Lisp. MIT Artificial Intelligence Memo 452, January 1978.

[45] 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.

[46] Guy Lewis Steele, Jr. and Gerald Jay Sussman. Design of a Lisp-based processor. *Communications of the ACM* 23(11):628–645, November 1980.

[47] 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*, pages 163–172. The Lisp Conference, 1980.

[48] Gerald Jay Sussman and Guy Lewis Steele, Jr. Scheme: an interpreter for extended lambda calculus. MIT Artificial Intelligence Memo 349, December 1975.

[49] Gerald Jay Sussman, Jack Holloway, Guy Lewis Steele, Jr., and Alan Bell. Scheme-79—Lisp on a chip. *IEEE Computer* 14(7):10–21, July 1981.

[50] Joseph E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. MIT Press, Cambridge, 1977.

[51] Texas Instruments, Inc. *TI Scheme Language Reference Manual*. Preliminary version 1.0, November 1985.

[52] Mitchell Wand. Continuation-based program transformation strategies. *Journal of the ACM* 27(1):174–180, 1978.

[53] Mitchell Wand. Continuation-based multiprocessing. In *Conference Record of the 1980 Lisp Conference*, pages 19–28. The Lisp Conference, 1980.

ALPHABETIC INDEX OF DEFINITIONS OF CONCEPTS, KEYWORDS, AND PROCEDURES

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

As the Scheme report was being prepared for publication, it was discovered that a concise language definition of only 43 pages was too thin to be bound properly. The following article has been appended in order to remedy this deficiency.

Computation: An Introduction to Engineering Design

Harold Abelson
Gerald Jay Sussman

Massachusetts Institute of Technology

Computer science is not a science. Its significance has little to do with computers. The real significance of the computer revolution is that it is a revolution in the way we think, and in the way in which we express what we think. The essence of this change is the emergence of what might best be called procedural epistemology—the study of the structure of knowledge from an imperative point of view, as opposed to the more declarative point of view taken by classical mathematical subjects. A computer language, from this perspective, is a novel formal medium for expressing ideas about methodology, not just a way to get a computer to perform operations. Programs are written for people to read, and only incidentally for machines to execute.

The MIT introductory computer-science subject reflects this view. In this subject, the syntax of particular programming-language constructs, clever algorithms for computing particular functions efficiently, or even the mathematical analysis of algorithms and the foundations of computing are only peripheral issues. The essential material is the techniques used to control the complexity of large systems:

1. We create abstractions to hide detail and to separate specification from implementation.
2. We describe a design in terms of a sequence of levels of description. Robust design requires that, at each level, the descriptive framework is rich enough so that small modifications of the design are only slightly different in description.
3. We establish conventional interfaces that enable us to combine standard well-understood pieces in a mix-and-match way.
4. We make languages to describe designs in appropriate ways. Each language emphasizes particular aspects of designs and deemphasizes others.

These techniques are not unique to the organization of computational systems. They are common to all of engineering design. By making these the centerpiece of our presentation, we come to view computer science as an abstract form of engineering. In a computational system, the elementary pieces are idealized and completely specified. Thus, the complexity of a design is limited only by the mind of the designer, not by the approximations inherent in modeling physical reality.

Expressing abstractions as procedures

One way to control complexity is by using abstractions to hide details. A design language provides a set of primitive constructs, means by which these can be combined, and means of abstraction by which combinations may be named and manipulated as if they were primitive. In programming, we often express abstractions as procedures that capture general patterns of operations.

A framework for thinking about design must not arbitrarily limit our ability to make abstractions. In particular, an expressive programming language should draw no distinction between patterns that abstract over procedures and patterns that abstract over other kinds of data. Procedures must be allowed to be passed as arguments, returned as values, and incorporated into data structures. Prodedural data allows us to capture general methods so that we can name, analyze, and build on them.

From a pedagogical point of view, expressing general methods as executable procedures makes them concrete and often easier to understand. Mathematical constructions that involve functional operators are naturally expressed in this way.

Here is a procedure that expresses the concept of summation—summing the values of a given function f evaluated at discrete points on the interval from a to b:

```
(define (sum f a next b)
  (if (> a b)
      0
      (+ (f a)
         (sum f (next a) next b))))
```

Sum takes as its arguments the lower bound a, the upper bound b, the procedure f that computes the value of the function f, and the procedure next that computes the next point of the interval to consider. The above definition expresses the idea that the sum over the interval from a to b is 0 if $a > b$ and otherwise is equal to $f(a)$ plus the sum over the interval from the next value after a until b.

We can use the sum procedure as a building block in expressing further concepts. For instance, the definite integral of a function f between the limits a and b can be approximated numerically using the formula

$$\int_a^b f = \left[f\left(a+\frac{dx}{2}\right)+f\left(a+dx+\frac{dx}{2}\right)+f\left(a+2dx+\frac{dx}{2}\right)+\cdots\right]dx$$

and we can express this formula procedurally in terms of sum:

```
(define (integral f a b dx)
  (define (add-dx x) (+ x dx))
  (* (sum f (+ a (/ dx 2)) add-dx b)
     dx))
```

Now we can use integral as a building block in more general constructions. As an example, consider Picard's method for solving ordinary differential equations of the form $dx/dt = f(x,t)$, where x is a function of t with a given initial value $x(0) = x_0$.

Picard's method for approximating the function x (the solution of the differential equation) uses the technique of iterative improvement. We choose an arbitrary function to serve as an initial approximation to x (for example, the function that has constant value x_0) and repeatedly improve this approximation by applying an operator that, given a function, produces a new function that better approximates x. This operator, when applied to a function y, produces the function whose value at t is

$$x_0 + \int_0^t f(y(s), s)ds$$

This function is a better approximation to the solution of the differential equation than y is.

We can express the Picard operator as the following procedure:

```
(define (((picard-operator f x0) y) t)
  (define (picard-integrand s)
    (f (y s) s))
  (+ x0 (integral picard-integrand 0 t dx)))
```

The picard-operator procedure takes as its arguments a function f and the initial value x_0. It produces an operator, which when applied to a function y, produces a function of t.

The nth-order Picard approximation to the solution x is obtained by beginning with any arbitrary function, and applying the operator n times.

```
(define (picard-method f x0 n)
  (define (arbitrary t) x0)
  ((repeated-application n (picard-operator f x0))
   arbitrary))
```

The repeated-application procedure takes a number and an operator as arguments. It produces the operator composed with itself the given number of times.

```
    (define (repeated-application n operator)
      (if (= n 0)
          identity-operator
          (compose operator
                   (repeated-application (- n 1) operator))))
    (define ((compose f g) x)
      (f (g x)))
```

To the student writing this program Picard's method is no longer a bunch of mathematical formulae, tied together with informal prose. It is actually executable, in a completely formal way. He can try it out on simple cases, examining the dynamic behavior and how the successive approximations relate to one another. This kind of activity leads to more complete understanding.

Levels of language for robust design

A complex design is a sequence of levels of description. Each level describes how parts of the design are constructed from elements that are regarded as primitive at that level. The parts constructed at each level are used as primitives at the next level. Thus, in designing electrical circuits, we use transistors, resistors and capacitors to build amplifiers, filters and oscillators. We assemble these to make various kinds of radios and sound equipment.

If a design is to be robust, its parts must have more generality than is needed for the particular application. The means for combining the parts must allow variations in the design plan, such as substituting similar parts for one another and varying the arrangement in which parts are combined. This generality insures that, at any level, small design changes can be expressed by making small changes in the description. A powerful design framework must therefore be more than just a way to build hierarchies of abstractions. It must be a succession of languages each complete in itself.

We see this succession of languages in electrical circuit design, where there are many conventional kinds of diagrams for describing how circuit elements are interconnected to produce modules, how modules are assembled into subsystems, and how subsystems are combined to make systems. Often, a particularly useful language is standardized as a commercial component family, such as TTL or CMOS to describe interconnection of digital-logic elements or Multibus to describe the assembly of board-level modules.

The idea of stratified design was used by Peter Henderson [2] in a beautiful analysis of the construction of the "Square Limit" woodcut of M.C.Escher. Henderson created a system for easily describing such images. Henderson's system consists of several languages. There is a language of primitive pictures that are constructed from points and lines. Built on top of this is a language of geometric combination that describes how pictures are placed relative to another to make compound pictures. At the next level, there is a language of general patterns of combination.

In Henderson's system, a picture is a procedure that takes a rectangle as argument, and draws an image scaled to fit the rectangle. At the lowest level of description, a picture is described as a collection geometric elements, where each element is specified in terms of (x, y) coordinates with respect to the unit square $(0 \leq x, y \leq 1)$. Figure 1 shows two simple pictures, `diamond` and `leg`, each described as line-segments that connect designated vertices. Besides segments, other picture elements appropriate to this level are circles with specified radii and centers, spline curves through designated points, and so on.

The next level of description in Henderson's system is a language of geometric combinators—placing pictures beside or above one another, or rotating pictures through multiples of 90 degrees. For example, the `beside` combinator horizontally adjoins two pictures so that their widths are in a given ratio. Figure 2 shows two pictures adjoined by the `beside` combinator.

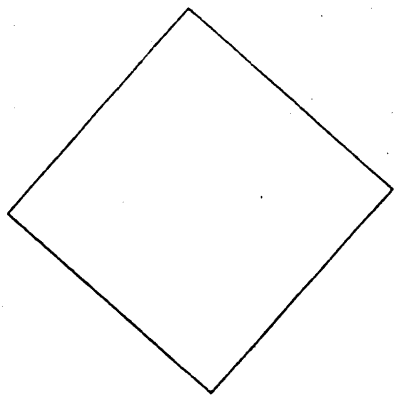
One important feature of these geometric combinators is that pictures are closed under combination: The `beside` of two pictures is itself a picture and thus can be further combined with other pictures. Combinators can be abstracted: We can express common patterns of picture combination as new picture combinators defined in terms of other combinators. For example, a `triangle` combination is formed by placing one picture above two copies of another:

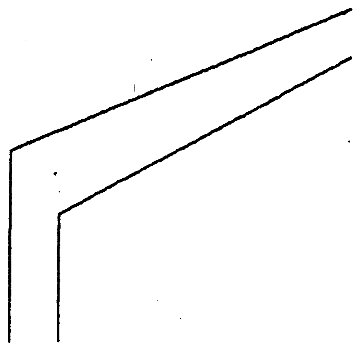
```
(define (triangle pict1 pict2 ratio)
  (above pict1
         (beside pict2 pict2 .5)
         ratio))
```

Moreover, since the combinators are expressed procedurally, we can easily construct complex combinators. The recursively defined combinator

```
(define (right-push pict n ratio)
  (if (= n 0)
      pict
      (beside pict
              (right-push pict (- n 1) ratio)
              ratio)))
```

produces the result of repeatedly adjoining n copies of a given picture, scaled by the given ratio as shown in figure 3. Similarly, using the primitive combi-





```
(define diamond
  (let ((v1 (vertex 0.5 0))
        (v2 (vertex 1 0.5))
        (v3 (vertex 0.5 1))
        (v4 (vertex 0 0.5)))
    (primitive-picture
      (segment v1 v2)
      (segment v2 v3)
      (segment v3 v4)
      (segment v4 v1))))
```

```
(define leg
  (let ((v1 (vertex (/ 1 8) 0))
        (v2 (vertex (/ 1 4) 0))
        (v3 (vertex 1 (/ 3 4)))
        (v4 (vertex 1 (/ 7 8)))
        (v5 (vertex (/ 1 4) (/ 1 3)))
        (v6 (vertex (/ 1 8) (/ 1 2))))
    (primitive-picture
      (segment v1 v6)
      (segment v6 v4)
      (segment v3 v5)
      (segment v5 v2))))
```

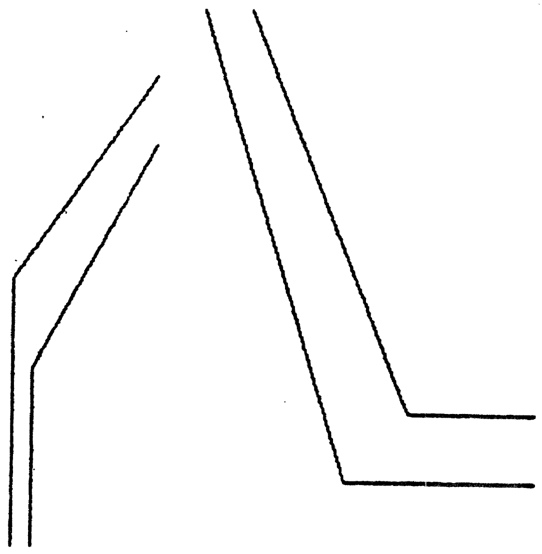
Figure 1. At the lowest level of description in Henderson's language, pictures are specified as collections of individual geometric elements. Here are two simple pictures, diamond and leg.

nator above we can write

```
(define (up-push pict n ratio)
  (if (= n 0)
      pict
      (above pict
             (up-push pict (- n 1) ratio)
             ratio)))
```

The combinator language derives its power from the closure and abstraction properties of the combinators. That is why it can describe seemingly complex figures using only a few simple ideas.

The combinators themselves are manipulated at a third level of description that captures common patterns of combining picture combinators. Thus,



(define legs (beside leg (rotate90 leg) .3))

Figure 2. The picture language includes geometric combinators for adjoining pictures horizontally and vertically, and for rotating pictures by 90 degrees. Here is a picture formed from elements in figure 1.

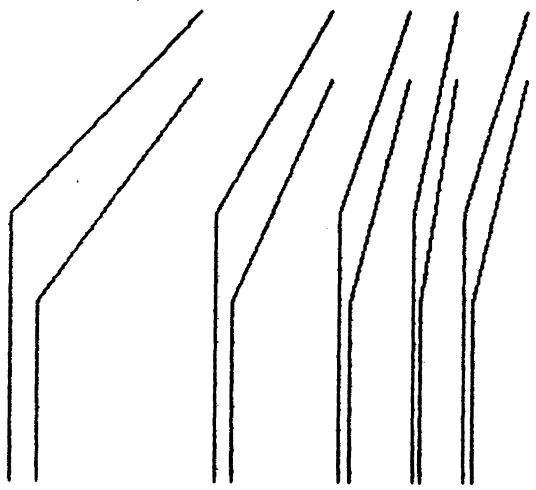
`right-push` and `up-push` are instances of the general pattern of pushing, which we can implement as a repeated application:

```
(define up-push (push above))

(define right-push (push beside))

(define ((push combiner) pict n ratio)
  (define (basic-combination p)
    (combiner pict p ratio))
  ((repeated-application n basic-combination) pict))
```

Push is a higher-order combinator that transforms combinators to more elaborate combinators. Having captured the idea of pushing, we can apply it to arbitrary combinators, for instance (see figure 4):

```
(define pyramid (push triangle))
```

Given such a description of the picture in figure 4, we can easily vary the pieces at any level: We could change the location of a point in the primitive picture `leg`, we could replace `legs` by `leg` in the basic repeated unit, or we could even replace `pyramid` by some other picture plan based upon `triangle`.



(right-push leg 4 .4)

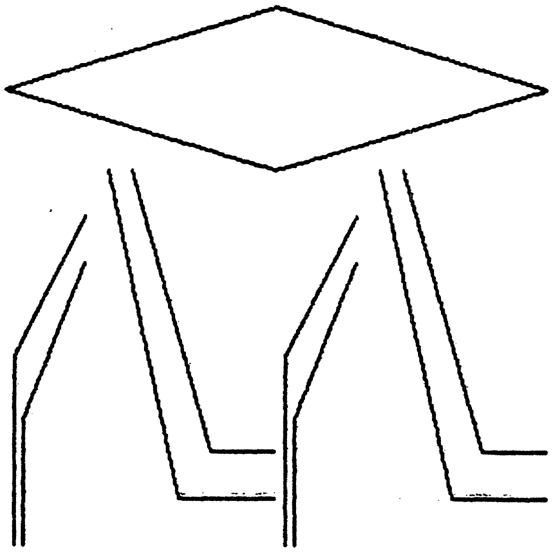
Figure 3. The combinator `right-push` iterates the `beside` combinator. Here we see the result of adjoining `leg` to itself 4 times.

Standard components and conventional interfaces

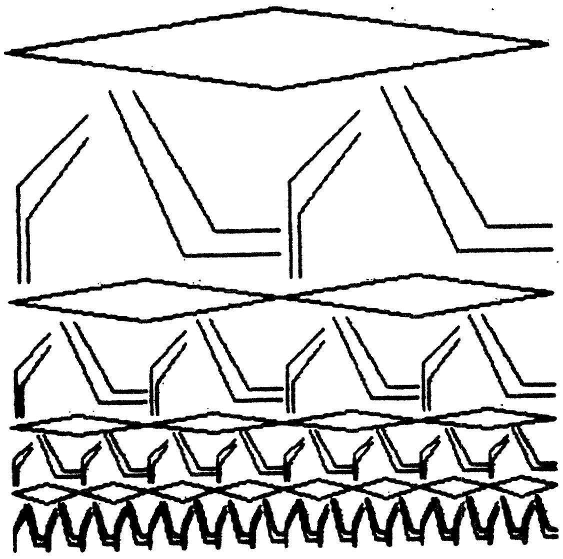
In addition to controlling complexity by building abstractions, we control complexity by making designs modular. A modular design is a combination of relatively independent pieces. We can encourage modular design by providing a library of standard components together with a conventional interface for connecting the components in a variety of ways. In digital design, for example, all components in standard logic families such as TTL have the same power-supply requirements and the same signal definitions.

Electrical engineers often design systems in terms of a language of block diagrams, consisting of blocks connected by lines. The blocks represent functions that process signals. The lines represent the communication paths by which these functional blocks are interconnected. The power in this language is in the assumption that all blocks obey the same interface specifications, so that they can be interconnected in a variety of configurations.

The programming analogue of the block-diagram language is based upon data objects called streams. A stream is a (possibly infinite) sequence of values. For instance, we could represent keyboard input to a program as a stream of characters. In signal-processing applications, we could use streams of numbers to represent the values of signals sampled at discrete intervals.



(define animal (triangle diamond legs .3))



(pyramid animal 3 .5)

Figure 4. The triangle combinator places one picture above two copies of another. We can use triangle to combine diamond and legs. Iterating the triangle combinator by means of the higer-order combinator push produces the pyramid combinator.

For fun, we might have the stream of all positive integers. Functional blocks are procedures that take streams as inputs and produce streams as outputs. We might have a procedure that takes a stream of numbers and puts out a stream of the squares of the numbers in its input stream. The streams are the conventional interface through which such blocks are connected to form systems.

Suppose we have a `filter` block that takes a stream as input and produces the stream of only those values in the input stream that satisfy a given condition; a `map` block that applies a given operation to successive items in a stream, producing the stream of resulting values; and an `add-streams` block that produces the elementwise sum of two streams. Then, given the stream of positive integers we can filter this to form the stream of odd integers, square each result, and add these to the integers themselves:

```
(add-streams (map square
                  (filter odd? integers))
             integers)
```

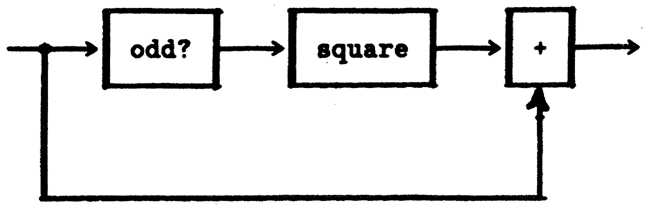
Alternatively, by connecting the blocks in a different configuration, we can form the sum of the integers and the squares of the integers, and filter these to select the odd elements:

```
(filter odd?
        (add-streams integers
                     (map square integers)))
```

The corresponding block diagrams are shown in figure 5.

We can perform other operations on streams. For instance, we can merge two ordered streams to produce an ordered stream. If we have two streams, we can produce the stream of all possible pairs (s,t) where s is in the first stream and t is in the second stream. In fact, working with such streams (and analogous triples, 4-tuples, etc.) corresponds to using nested loops in traditional programming style.

As an illustration of the expressive power of the stream formulation, consider the problem of finding the numbers that can be written as the sum of two cubes in two different ways—the so-called Ramanujan numbers. We assume that we have a stream operation `unique-pairs` that, given a stream of values s_i, produces the stream of all pairs (s_i, s_j) with $i < j$. To generate the Ramanujan numbers, we map the along the stream of pairs of integers, generating the sum of the cubes of the two components in each pair. We then produce the stream of pairs of these sums of cubes and filter this to find the pairs with equal components. The surviving pairs contain the Ramanujan



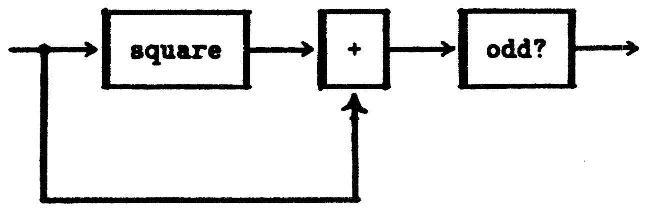


Figure 5. The power of block-diagram design is that we can assemble the same components in different ways to produce a variety of systems.

numbers, which we extract by taking the first component of each pair:

```
(define ramanujan-numbers
  (map first
       (filter equal-components
         (unique-pairs (map sum-of-cubes
                            (unique-pairs integers))))))
(define (equal-components pair)
  (= (first pair) (second pair)))
(define (sum-of-cubes pair)
  (+ (cube (first pair)) (cube (second pair))))
```

Streams are data objects that encapsulate the control structure for the natural iterations on aggregate data. This allows us to write programs that manipulate streams without explicitly representing the details of the control.† The decoupling of data flow from control allows the implementer the freedom to order the computation in any way that is convenient. Infinite streams work because we adopt a lazy evaluation strategy that does no computation until

†APL was the first major programming language to be organised around functional manipulations of aggregate data. As Alan Perlis says: "[People tell me] 'APL is unstructured. There is no while in it.' I say, 'It's got nothing but wiles', which pun they don't get."

the result of that computation is required.‡

Establishing new languages

As we confront increasingly complex problems, we will find that any fixed language is not sufficient for our needs. We must constantly turn to new languages in order to express our ideas more effectively. We can often enhance our ability to deal with a complex problem by describing it in a different way, using new kinds of primitives, means of combination, and means of abstraction. Such a linguitic shift occurs in engineering when one needs to describe function rather than structure. The structure of an analog circuit is described in terms of circuit elements and modules, but analog engineers also use signal-flow graphs to capture the essential functioning of signal-processing systems.

Sometimes it is necessary to completely change the framework by which the descriptions are held together. All of the examples we have shown so far have been constructed by functional composition. But many systems are only awkwardly described in terms of functions. Physcial systems are often described as sets of algebraic constraints among several variables. A constraint such as $x + yz = 3$ is no more a function for computing x given y and z than it is a function for computing z given x and y. A language for manipulating such descriptions must be fundamentally a language about relations, not a language about functions.

The universality of computation allows us to deal with linguistic shifts even of this order, because programming languages can themselves be described in computational terms. We can write interpreters to embed one language within another, and we can write compilers to translate from one language to another.

In our subject we illustrate these principles by developing interpreters for several languages. We can easily implement an interpreter for the procedural language we have been using throughout the subject, and by perturbing this interpreter we illustrate semantic variants of the language. We also implement a vastly different language—a logic-programming language similar to Prolog. In this language, the objects of discourse are relations rather than procedures. Primitive relations are assertions that are posited. Relations are combined by the logical connectives and, or, and not. Relations are abstracted by giving rules that allow one to infer new relations from combinations of others.

We find that interpreters for these different languages have similar structures. Each interpreter has an `eval` phase that classifies expressions, decomposing compound expressions and recursively interpreting the pieces. There

‡Stream processing was invented by Peter Landin [3].

is also an apply phase that combines the results of interpreting the parts by specializing the abstractions to specific situations.

Lisp

In teaching our subject we use the Scheme dialect of the programming language Lisp. It supports (but does not enforce) more of the large-scale strategies of design than any other language we know. We can make procedural and data abstractions, we can use higher-order functions to capture common patterns of usage, we can model local state using assignment and data mutation, we can link parts of a program with streams and delayed evaluation, and we can easily implement embedded languages. All of this is packaged in an interactive environment with support for incremental program design, construction, testing, and debugging.

Summary

The above examples show how the introductory subject in computer science can in fact be an introduction to general principles of engineering design. Robust designs are organized into levels of abstractions. Each level is described in terms of a language that is appropriately constructed for expressing designs at that level. Descriptions at each level are closed under combination at that level, and we can achieve this by providing a library of standard parts with conventional interfaces. Sometimes we must invent vastly different frameworks to express a design, and there are well-understood technologies for erecting new languages.

By using computation as a forum for expressing methodology this subject sets a tone for the rest of the curriculum. Methodology is rarely taught explicitly in classes, or written in texts. With the exception of analytic theories, the working knowledge of professionals is almost universally considered intrinsically informal, hence unteachable except by experience. If we express working knowledge formally, in computational terms, we can manipulate it, reflect on it, and transmit it more effectively to students.

Acknowledgements

These ideas evolved over the past ten years as a result of reformulating the introductory computer science subject at MIT. Details of the subject are available in [1]. The point of view that computer science is abstract engineering and the importace of levels of language in engineering design has long been

stressed by Joel Moses. The significance of procedural epistemology for education is the major thrust of the work of Seymour Papert [4].

References

1. Harold Abelson, Gerald Jay Sussman, Julie Sussman, *Structure and Interpretation of Computer Programs*, MIT Press and McGraw-Hill, 1985.
2. Peter Henderson, "Functional Geometry", in *Proceedings of the 1982 ACM Symposium on on Lisp and Functional Programming.*
3. Landin, Peter, "A correspondence between Algol 60 and Church's lambda notation: Part I", *Communications of the ACM* 8(2):89–101, 1965.
4. Seymour Papert, *Mindstorms*, Basic Books, 1981.