Contents

Chapter 1

Introduction

Alscheme is a proper extension of the Scheme programming language. Scheme is a dialect of the Lisp programming language [23]. Lisp was developed by John McCarthy in the 1960's. Scheme was developed in the late 1970's by Jay Sussman and Guy Steele. Scheme is now an ANSI/IEEE standard [5], [13]. Alscheme is the basic language of `Analysa`.

`Analysa` is a system for analysing physical phenomena. `Analysa` serves as "software glue" to combine standard (or nearly standard) software components. In the past, scientific programmers have combined these components in ad-hoc, system-dependent ways.

In addition, `Analysa` provides small, application-area-specific languages. Specifications written in these small languages can be compiled and used seamlessly in `Analysa`. The perfomance of the compiled code is competitive with hand-crafted programs written in C or Fortran.

The overall effect is that simulation programs created using `Analysa` are 10 to 100 times smaller than simulations prepared in traditional ways. This makes it possible for analysts to prepare, in a timely way, analysis programs for designers to use.

The base language of `Analysa` is a minimal extension to Scheme, which we call Alscheme to avoid confusion with standard Scheme. This manual explains the details of Alscheme. It is based extensively on the manual [20] for Scheme, and differs only by specifying some options and explaining the extensions.

The principal extension to Scheme in Alscheme is the introduction of an "object" or "class" system, to support object-oriented programming. Our implementation derives from the Common Lisp [23] object system, known as CLOS [14] [15]. More precisely, it is based on "tiny CLOS" or `tclos`, and it was developed in `stk` [12].

This manual is not intended as an introduction to `Analysa`, nor is it intended as a textbook on Scheme. The `Analysa` Quick Start Guide is an attempt at the former, and there are numerous texts on Scheme, e.g., [1] [4] [5] [6] [7] [8] [10] [11] [17] [22].

If you are beginning to use Scheme (or any Lisp-like language) for the first time, you will notice the large number of parentheses. For a while. After some time, they will begin to appear as the simple placeholders that they are. Almost like "whitespace" in other languages. Some languages don't even require spaces between things. In Fortran, the statements

```
DO1I=7
DO1I=7,11
```

are both legitimate. Clearly some spaces might make it clearer that the second is a loop, and that the first one is not. The parentheses in Scheme insure that the code is easily read, and you will find that they quickly fade into the background after a bit of experience.

Scheme traces its roots through Lisp to the "lambda calculus" of mathematics [16]. This is a key part of the logical foundations of mathematics, and is at once simple, elegant and powerful. Thus one can be assured that Scheme is built on a firm and long-lived foundation. It is increasingly being used in industrial software in computational design.

Chapter 2

Overview of Scheme

2.1 Semantics

This chapter gives an overview of Scheme's semantics. A detailed informal semantics is the subject of chapters 4 through 7. For reference purposes, section 8.2 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 Alscheme programs do not (usually!) run out of storage is that Alscheme reclaims the storage occupied by an object if it 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 7.9.

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

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

2.2 Syntax

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

The `read` procedure performs syntactic as well as lexical decomposition of the data it reads. The `read` procedure parses its input as data (section 8.1.2), not as program.

The formal syntax of Scheme is described in section 8.1.

2.3 Notation and terminology

2.3.1 Entry format

Chapters 5, 7 and 9 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 *category*

followed by an explanation and example in most cases. In chapter 9, the format for the categories becomes more complex, but it is fairly uniform in chapters 5 and 7, as we now explain.

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

$\langle\text{thing}_1\rangle \ldots$

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

$\langle\text{thing}_1\rangle\ \langle\text{thing}_2\rangle \ldots$

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*`)` procedure

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

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

indicate that in Alscheme, the `make-vector` procedure is defined to take one or two arguments.

It is an error for an operation to be presented with an argument that it is not specified to handle. For succinctness, we follow the convention that if an argument name is also the name of a type listed in section 4.5, then that argument must be of the named type. For example, the header line for `vector-ref` given above dictates that the first argument to `vector-ref` must be a vector. The following naming conventions also imply type restrictions:

obj	any object
$list$, $list_1$, … $list_j$, …	list (see section 7.3)
z, z_1, … z_j, …	complex number
x, x_1, … x_j, …	real number
y, y_1, … y_j, …	real number
q, q_1, … q_j, …	rational number
n, n_1, … n_j, …	integer
k, k_1, … k_j, …	exact non-negative integer

2.3.2 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 4.4 for a discussion of external representations of objects.

2.3.3 Naming conventions

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

By convention, the names of procedures that store values into previously allocated locations (see section 4.6) usually end in "`!`". Such procedures are called mutation procedures. By convention, the value returned by a mutation procedure is `void`.

By convention, when "`->`" appears within the names of a procedure, it indicates that the procedure takes an object of one type and returns an analogous object of another type. For example, `list->vector` takes a list and returns a vector whose elements are the same as those of the list.

Chapter 3

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 8.1.

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`.

3.1 Identifiers

Most identifiers allowed by other programming languages are also acceptable to Scheme. In Alscheme, any sequence of letters, digits, and "extended alphabetic characters" that begins with a character that cannot begin a number is an identifier. In addition, `+`, `-`, and `...` are identifiers. Here are some examples of identifiers:

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

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

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

See section 8.1.1 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 4.1).
- When an identifier appears as a literal or within a literal (see section 5.1.2), it is being used to denote a *symbol* (see section 7.4).

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
```

3.2 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))))))
```

3.3 Other notations

For a description of the notations used for numbers, see section 7.5.

`. + -` 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 7.3), and to indicate a rest-parameter in a formal parameter list (section 5.1.5). A delimited sequence of three successive periods is also an identifier.

`( )` Parentheses are used for grouping and to notate lists (section 7.3).

`'` The single quote character is used to indicate literal data (section 5.1.2).

`` ` `` The backquote character is used to indicate almost-constant data (section 5.2.6).

`, ,@` The character comma and the sequence comma at-sign are used in conjunction with backquote (section 5.2.6).

`"` The double quote character is used to delimit strings (section 7.7).

`\` Backslash is used in the syntax for character constants (section 7.6) and as an escape character within string constants (section 7.7).

`[ ] { }` 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 7.1).

`#\` This introduces a character constant (section 7.6).

`#(` This introduces a vector constant (section 7.8). Vector constants are terminated by `)` .

`#e #i #b #o #d #x` These are used in the notation for numbers (section 7.5.3).

`#'` This introduces a semantic keyword (section 5.1.3).

`#!` This introduces a comment.

Chapter 4

Basic concepts

4.1 Variables and regions

Any identifier that is not a syntactic keyword (see section 3.1) may be used as a variable. A variable may name a location where a value can be stored. A variable that does so is said to be *bound* to the location. The set of all visible bindings in effect at some point in a program is known as the *environment* in effect at that point. The value stored in the location to which a variable is bound is called the variable's value. By abuse of terminology, the variable is sometimes said to name the value or to be bound to the value. This is not quite accurate, but confusion rarely results from this practice.

Certain expression types are used to create new locations and to bind variables to those locations. The most fundamental of these *binding constructs* is the lambda expression, because all other binding constructs can be explained in terms of lambda expressions. The other binding constructs are `let`, `let*`, `letrec`, and `do` expressions (see sections 5.1.5, 5.2.2, and 5.2.4).

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 7); if there is no binding for the identifier, it is said to be *unbound*.

4.2 True and false

Any Scheme value can be used as a boolean value for the purpose of a conditional test. As explained in section 7.1, all values count as true in such a test except for `#f`. This manual uses the word "true" to refer to any Scheme value that counts as true, and the word "false" to refer to `#f`.

4.3 Void

The empty set is of critcal importance in mathematics [2]. For this reason we include `#void` to represent this important concept.

4.4 External representations

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

The external representation of an object is not necessarily unique. The integer 28 also has representations "`#e28.000`" and "`#x1c`", and the list in the previous paragraph also has the representations "`( 08 13 )`" and "`(8 . (13 . ()))`" (see section 7.3).

Many objects have standard external representations, but some, such as procedures, do not have standard representations.

An external representation may be written in a program to obtain the corresponding object (see `quote`, section 5.1.2).

External representations can also be used for input and output. The procedure `read` (section 7.10.2) parses external representations, and the procedure `write` (section 7.10.3) generates them. Together, they provide an elegant and powerful input/output facility.

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

The syntax of external representations of various kinds of objects accompanies the description of the primitives for manipulating the objects in the appropriate sections of chapter 7.

4.5 Disjointness of types

No object satisfies more than one of the following predicates:

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

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

4.6 Storage model

Variables and objects such as pairs, vectors, and strings implicitly denote locations or sequences of locations. A string, for example, denotes as many locations as there are characters in the string.

(These locations need not correspond to a full machine word.) A new value may be stored into one of these locations using the `string-set!` procedure, but the string continues to denote the same locations as before.

An object fetched from a location, by a variable reference or by a procedure such as `car`, `vector-ref`, or `string-ref`, is equivalent in the sense of `eqv?` (section 7.2) to the object last stored in the location before the fetch.

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

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

Chapter 5

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 8.3. They are redundant in the strict sense of the word, but they capture common patterns of usage, and are therefore provided as convenient abbreviations.

5.1 Primitive expression types

5.1.1 Variable references

⟨variable⟩ syntax

An expression consisting of a variable (section 4.1) 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
```

5.1.2 Literal expressions

`(quote` ⟨datum⟩`)` syntax

`'`⟨datum⟩ syntax

⟨constant⟩ syntax

`(quote` ⟨datum⟩`)` evaluates to ⟨datum⟩. ⟨Datum⟩ may be any external representation of a Scheme object (see section 4.4). 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` $\langle$datum$\rangle$`)` may be abbreviated as `'`$\langle$datum$\rangle$. The two notations are equivalent in all respects.

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

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

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

As noted in section 4.6, 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!`.

5.1.3 Semantic keywords

Alscheme introduces semantic keywords in the form

```
#'keygoeshere
```

together with some predefined operations. They are used in both the object (class) system in chapter 9. Semantic keywords evaluate to themselves, and values cannot be bound to them.

One operation defined on keywords is

```
(keyword? x)          ⟹ #t
```

if indeed `x` is one; else

```
(keyword? x)          ⟹ #f
```

Another operation can create a keyword:

```
(string->keyword "ex")
```

will return `#'ex`.

5.1.4 Procedure calls

(⟨operator⟩ ⟨operand$_1$⟩ ...) syntax

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

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

A number of procedures are available as the values of variables in the initial environment; for example, the addition and multiplication procedures in the above examples are the values of the variables + and *. New procedures are created by evaluating lambda expressions (see section 5.1.5).

Procedure calls are also called *combinations*.

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

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

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

5.1.5 Lambda expressions

(lambda ⟨formals⟩ ⟨body⟩) 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 add4
  (let ((x 4))
    (lambda (y) (+ x y))))
(add4 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.

It is an error for a ⟨variable⟩ to appear more than once in ⟨formals⟩.

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

Each procedure created as the result of evaluating a lambda expression is tagged with a storage location, in order to make `eqv?` and `eq?` work on procedures (see section 7.2).

5.1.6 Conditionals

`(if` ⟨test⟩ ⟨consequent⟩ ⟨alternate⟩`)` 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 7.1), 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 `void`.

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

5.1.7 Assignments

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

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

5.2 Derived expression types

For reference purposes, section 8.3 gives rewrite rules that will convert constructs described in this section into the primitive constructs described in the previous section.

5.2.1 Conditionals

`(cond` ⟨clause$_1$⟩ ⟨clause$_2$⟩ ...`)` 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 7.1). 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 `void`; 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
```

`(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 7.2) 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 `void`.

```
(case (* 2 3)
  ((2 3 5 7) 'prime)
  ((1 4 6 8 9) 'composite)) ==>  composite
(case (car '(c d))
  ((a) 'a)
  ((b) 'b))                 ==>  void
(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 7.1) 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 7.1) 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)
```

5.2.2 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, all the bindings are in effect while their initial values are being computed, thus allowing mutually recursive definitions.

`(let` ⟨bindings⟩ ⟨body⟩`)` 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. It is an error for a ⟨variable⟩ to appear more than once in the list of variables being bound.

Semantics: The ⟨init⟩s are evaluated in the current environment (in some unspecified order), the ⟨variable⟩s are bound to fresh locations holding the results, the ⟨body⟩ is evaluated in the extended environment, and the value of the last expression 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 ((x 7)
        (z (+ x y)))
    (* z x)))                ⟹  35
```

See also named `let`, section 5.2.4.

(`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 2) (y 3))
  (let* ((x 7)
         (z (+ x y)))
    (* z x)))              ⟹  70
```

(`letrec` ⟨bindings⟩ ⟨body⟩) syntax

Syntax: ⟨Bindings⟩ should have the form

((⟨variable$_1$⟩ ⟨init$_1$⟩) ...),

and ⟨body⟩ should be a sequence of one or more expressions. It is an error for a ⟨variable⟩ to appear more than once in the list of variables being bound.

Semantics: The ⟨variable⟩s are bound to fresh locations holding undefined values, the ⟨init⟩s are evaluated in the resulting environment (in some unspecified order), each ⟨variable⟩ is assigned to the result of the corresponding ⟨init⟩, the ⟨body⟩ is evaluated in the resulting environment, and the value of the last expression in ⟨body⟩ is returned. Each binding of a ⟨variable⟩ has the entire `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 assigning or referring to the value of any ⟨variable⟩. If this restriction is violated, then it is an error. The restriction is necessary because Scheme passes arguments by value rather than by name. In the most common uses of `letrec`, all the ⟨init⟩s are lambda expressions and the restriction is satisfied automatically.

5.2.3 Sequencing

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

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

```
(define x 0)

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

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

5.2.4 Iteration

(do ((⟨variable$_1$⟩ ⟨init$_1$⟩ ⟨step$_1$⟩)
    ...)
   (⟨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 7.1), 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 void.

The region of the binding of a ⟨variable⟩ consists of the entire do expression except for the ⟨init⟩s. It is an error for a ⟨variable⟩ to appear more than once in the list of do variables.

A ⟨step⟩ may be omitted, in which case the effect is the same as if (⟨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

Alscheme permits 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))
```

5.2.5 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 7.9) for a more complete description of `delay`.

5.2.6 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⟩`)`.

```
(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.

Chapter 6

Program structure

6.1 Programs

A Scheme program consists of a sequence of expressions and definitions. Expressions are described in chapter 5; 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 can be interpreted declaratively. They cause bindings to be created in the top level environment. Expressions occurring at the top level of a program are interpreted imperatively; they are executed in order when the program is invoked or loaded, and typically perform some kind of initialization.

6.2 Definitions

Definitions are valid in all contexts where expressions are allowed.

A definition should have one of the following forms:

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

  ⟨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⟩`)`

  ⟨Formal⟩ should be a single variable. This form is equivalent to

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

- `(begin` ⟨definition$_1$⟩ `...)`

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

6.2.1 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.

6.2.2 Internal definitions

Alscheme permits definitions to occur within 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 portion of the ⟨body⟩ which follows the definition. 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))))
```

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

Chapter 7

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.

7.1 Booleans

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

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

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

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

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

`(not` *obj*`)` 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 '())             ⟹  #f
(not (list))          ⟹  #f
(not 'nil)            ⟹  #f
```

`(boolean? `*obj*`)` procedure

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

```
(boolean? #f)          ⟹  #t
(boolean? 0)           ⟹  #f
(boolean? '())         ⟹  #f
```

7.2 Equivalence predicates

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

`(eqv? `obj_1 obj_2`)` procedure

The `eqv?` procedure defines a useful equivalence relation on objects. Briefly, it returns `#t` if obj_1 and obj_2 should normally be regarded as the same object. The following specification of `eqv?` holds for Alscheme.

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

- obj_1 and obj_2 are both `#t` or both `#f`.
- obj_1 and obj_2 are both symbols and

```
(string=? (symbol->string obj1)
          (symbol->string obj2))
                               ⟹  #t
```

- obj_1 and obj_2 are both numbers, are numerically equal (see `=`, section 7.5), and are either both exact or both inexact.
- obj_1 and obj_2 are both characters and are the same character according to the `char=?` procedure (section 7.6).
- both obj_1 and obj_2 are the empty list.
- obj_1 and obj_2 are pairs, vectors, or strings that denote the same locations in the store (section 4.6).
- obj_1 and obj_2 are procedures whose location tags are equal (section 5.1.5).

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

- obj_1 and obj_2 are of different types (section 4.5).
- one of obj_1 and obj_2 is `#t` but the other is `#f`.
- obj_1 and obj_2 are symbols but

```
(string=? (symbol->string obj1)
          (symbol->string obj2))
                          ⟹  #f
```

- one of obj_1 and obj_2 is an exact number but the other is an inexact number.
- obj_1 and obj_2 are numbers for which the `=` procedure returns `#f`.
- obj_1 and obj_2 are characters for which the `char=?` procedure returns `#f`.
- one of obj_1 and obj_2 is the empty list but the other is not.
- obj_1 and obj_2 are pairs, vectors, or strings that denote distinct locations.
- obj_1 and obj_2 are procedures that would behave differently (return a different value or have different side effects) for some arguments.

```
(eqv? 'a 'a)                  ⟹  #t
(eqv? 'a 'b)                  ⟹  #f
(eqv? 2 2)                    ⟹  #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 which fall outside the scope of the above rules.

```
(eqv? "" "")                  ⟹  #f
(eqv? '#() '#())              ⟹  #f
(eqv? (lambda (x) x)
      (lambda (x) x))         ⟹  #f
(eqv? (lambda (x) x)
      (lambda (y) y))         ⟹  #f
```

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))
                             ⟹ void

(letrec ((f (lambda () (if (eqv? f g) 'both 'f)))
         (g (lambda () (if (eqv? f g) 'both 'g)))
  (eqv? f g))
                             ⟹ void

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

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

(eq? *obj*$_1$ *obj*$_2$) 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))              ⟹ #f
(eq? (list 'a) (list 'a))    ⟹ #f
(eq? "a" "a")                ⟹ #f
(eq? "" "")                  ⟹ #f
(eq? '() '())                ⟹ #t
(eq? 2 2)                    ⟹ #t
(eq? #\A #\A)                ⟹ #t
(eq? car car)                ⟹ #t
(let ((n (+ 2 3)))
  (eq? n n))                 ⟹ #t
(let ((x '(a)))
  (eq? x x))                 ⟹ #t
(let ((x '#()))
  (eq? x x))                 ⟹ #t
(let ((p (lambda (x) x)))
  (eq? p p))                 ⟹ #t
```

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

`(equal?` *obj*$_1$ *obj*$_2$`)` 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))     ⟹  #f
```

7.3 Pairs and lists

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

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

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

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

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

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

The most general notation (external representation) for Scheme pairs is the "dotted" notation `(`c_1 `.` c_2`)` where c_1 is the value of the car field and c_2 is the value of the cdr field. For example `(4 . 5)` is a pair whose car is 4 and whose cdr is 5. Note that `(4 . 5)` is the external representation of a pair, not an expression that evaluates to a pair.

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

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

and

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

are equivalent notations for a list of symbols.

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

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

is equivalent to

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

Whether a given pair is a list depends upon what is stored in the cdr field. When the `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)
(list? y)            ⟹  #t
(set-cdr! x 4)       ⟹  void
x                    ⟹  (a . 4)
(eqv? x y)           ⟹  #t
y                    ⟹  (a . 4)
(list? y)            ⟹  #f
(set-cdr! x x)       ⟹  void
(list? x)            ⟹  #f
```

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 8.1.2). Among other things, this permits the use of the `read` procedure to parse Scheme programs. See section 4.4.

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

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

```
(define (f) (list 'not-a-constant-list))
(define (g) '(constant-list))
(set-car! (f) 3)         ⟹  void
(set-car! (g) 3)         ⟹  void
```

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

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

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

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

(`list?` *obj*) procedure

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

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

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

Returns a newly allocated list of its arguments.

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

(`length` *list*) procedure

Returns the length of *list*.

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

(`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 the last argument 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
(list-ref '(a b c d)
          (inexact->exact (round 1.8)))
          ⟹  c
```

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

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

```
(memq 'a '(a b c))          ⟹  (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))   ⟹  (101 102)
(memv 101 '(100 101 102))   ⟹  (101 102)
```

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

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

```
(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)))
                           ⟹  (5 7)
(assv 5 '((2 3) (5 7) (11 13)))
                           ⟹  (5 7)
```

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

7.4 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 Alscheme uses 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 3.1 and 8.1.1.

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.

`(symbol?` *obj*`)` procedure

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

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

`(symbol->string` *symbol*`)` 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 5.1.2) or by a call to the `read` procedure, and its name contains alphabetic characters, then the string returned will contain characters in 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.

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

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

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

```
(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
```

7.5 Numbers

Numerical computation has traditionally been neglected by the Lisp community. Until Common Lisp there was no carefully thought out strategy for organizing numerical computation, and with the exception of the MacLisp system [19] little effort was made to execute numerical code efficiently. The Scheme report [20], upon which this manual is based, recognizes the excellent work of the Common Lisp committee and accepts many of their recommendations. That report simplifies and generalizes their proposals in a manner consistent with the purposes of Scheme.

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

7.5.1 Numerical types

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

number
complex
real
rational
integer

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

There is no simple relationship between a number's type and its representation inside a computer. Although Alscheme offers at four different representations of 3, these different representations denote the same integer.

Scheme's numerical operations treat numbers as abstract data, as independent of their representation as possible. The machine-dependent bit pattern representing a number in Alscheme should not be apparent to a casual programmer writing simple programs.

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

7.5.2 Exactness

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

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

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

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

7.5.3 Syntax of numerical constants

The syntax of the written representations for numbers is described formally in section 8.1.1.

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

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

In Alscheme, numerical constants may be written with an exponent marker that indicates the desired precision of the inexact representation. The letters `s`, `f`, `d`, and `l` specify the use of *short*, *single*, *double*, and *long* precision, respectively.

```
3.14159265358979F0
        Round to single — 3.14159265358979
0.6L0
        Extend to long — 0.6
```

7.5.4 Numerical operations

The reader is referred to section 2.3.1 for a summary of the naming conventions used to specify restrictions on the types of arguments to numerical routines. The examples used in this section assume that any numerical constant written using an exact notation is indeed represented as an exact number. Some examples also assume that certain numerical constants written using an inexact notation can be represented without loss of accuracy; the inexact constants were chosen so that this is likely to be true in implementations that use flonums to represent inexact numbers.

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

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

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

```
(complex? 3)          ⟹  #t
(real? 3)             ⟹  #t
(real? #e1e10)        ⟹  #t
(integer? 3.0)        ⟹  #t
```

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

`(exact?` z`)` procedure
`(inexact?` z`)` procedure

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

`(=` z_1 z_2 z_3 `...)` procedure
`(<` x_1 x_2 x_3 `...)` procedure
`(>` x_1 x_2 x_3 `...)` procedure
`(<=` x_1 x_2 x_3 `...)` procedure
`(>=` x_1 x_2 x_3 `...)` procedure

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

These predicates are required to be transitive.

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

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

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

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

`(max` x_1 x_2 `...)` procedure
`(min` x_1 x_2 `...)` procedure

These procedures return the maximum or minimum of their arguments.

```
(max 3 4)          ⟹  4    ; exact
(max 3.9 4)        ⟹  4.0  ; inexact
```

Note: If any argument is inexact, then the result will also be inexact.

`(+` z_1 `...)` 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`)` procedure
`(-` z`)` procedure
`(-` z_1 z_2 `...)` procedure

`(/` z_1 z_2`)` procedure
`(/` z`)` 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` x`)` procedure

`Abs` returns the magnitude of its argument.

```
(abs -7)                ⟹  7
```

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

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

```
(quotient n1 n2)        ⟹  n3
(remainder n1 n2)       ⟹  n4
(modulo n1 n2)          ⟹  n4
```

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

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

provided all numbers involved in that computation are exact.

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

```
(modulo 13 4)           ⟹  1
(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

(remainder -13 -4.0)    ⟹  -1.0  ; inexact
```

(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 32.0 -36)          ⟹  288.0  ; inexact
(lcm)                   ⟹  1
```

(numerator q) procedure
(denominator q) procedure

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

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

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

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

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

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

```
(floor -4.3)            ⟹  -5.0
(ceiling -4.3)          ⟹  -4.0
(truncate -4.3)         ⟹  -4.0
(round -4.3)            ⟹  -4.0

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

(round 7/2)             ⟹  4    ; exact
(round 7)               ⟹  7
```

`(rationalize` x y`)` procedure

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

```
(rationalize
  (inexact->exact .3) 1/10) ⟹ 1/3    ; exact
(rationalize .3 1/10)       ⟹ #i1/3  ; inexact
```

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

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

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 formulæ:

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

$$\cos^{-1} z = \pi/2 - \sin^{-1} z$$

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

The above specification follows [23], which in turn cites [18]; refer to these sources for more detailed discussion of branch cuts, boundary conditions, and implementation of these functions. When it is possible these procedures produce a real result from a real argument.

`(sqrt` z`)` procedure

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

`(expt` 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.

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

`Exact->inexact` returns an inexact representation of z. The value returned is the inexact number that is numerically closest to the argument.

`Inexact->exact` returns an exact representation of z. The value returned is the exact number that is numerically closest to the argument.

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

7.5.5 Numerical input and output

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

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

```
(let ((number number)
      (radix radix))
  (eqv? number
        (string->number (number->string number
                                        radix)
                        radix)))
```

is true. It is an error if no possible result makes this expression true.

If *number* is inexact, the radix is 10, and the above expression can be satisfied by a result that contains a decimal point, then the result contains a decimal point and is expressed using the minimum number of digits (exclusive of exponent and trailing zeroes) needed to make the above expression true [24, 3]; otherwise the format of the result is unspecified.

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

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

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

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

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

```
(string->number "100")        ⟹  100
(string->number "100" 16)     ⟹  256
(string->number "1e2")        ⟹  100.0
(string->number "15##")       ⟹  1500.0
```

7.6 Characters

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

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

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

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

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

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

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

`(char=? ` $char_1$ $char_2$`)` procedure
`(char<? ` $char_1$ $char_2$`)` procedure
`(char>? ` $char_1$ $char_2$`)` procedure
`(char<=? ` $char_1$ $char_2$`)` procedure
`(char>=? ` $char_1$ $char_2$`)` 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.

(`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

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

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

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

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

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

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

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

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

(`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.

7.7 Strings

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

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

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

A string constant may continue from one line to the next;

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

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

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

`(string?` *obj*`)` procedure

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

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

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

`(string` *char* ...`)` procedure

Returns a newly allocated string composed of the arguments.

`(string-length` *string*`)` procedure

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

`(string-ref` *string k*`)` 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 `void`.

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

`(string=? ` *string*$_1$ *string*$_2$`)` 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*$_2$`)` procedure
`(string>? ` *string*$_1$ *string*$_2$`)` procedure
`(string<=? ` *string*$_1$ *string*$_2$`)` procedure
`(string>=? ` *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
`(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 numerical predicates.

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

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

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

`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* ...`)` procedure

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

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

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

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

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

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

Stores *char* in every element of the given *string* and returns `void`.

7.8 Vectors

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

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

Vectors are written using the notation `#(`*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*`)` procedure

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

`(make-vector` *k*`)` 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 random.

`(vector` *obj* ...`)` 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*`)` procedure

Returns the number of elements in *vector*.

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

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

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

`(vector-set! `*vector k obj*`)` 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 `void`.

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

(vector-set! '#(0 1 2) 1 "doe")
        ⟹ error ; constant vector
```

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

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

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

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

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

7.9 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*`)` 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*`)` procedure
`(apply `*proc* arg_1 … *args*`)` procedure
Proc must be a procedure and *args* must be a list. The first () 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* $list_1$ $list_2$ ...) 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, in order from left to right. The dynamic order in which *proc* is applied to the elements of the *lists* is unspecified.

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

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

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

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

(for-each *proc* $list_1$ $list_2$ ...) 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 `void`.

```
(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 5.2.5). If no value has been computed for the promise, then a value is computed and returned. The value of the promise is cached (or "memoized") so that if it is forced a second time, the previously computed value is returned.

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

```
                              ⟹  (3 3)

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

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

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

```
(define count 0)
(define p
  (delay (begin (set! count (+ count 1))
                (if (> count x)
                    count
                    (force p)))))
(define x 5)
p                             ⟹  a promise
(force p)                     ⟹  6
p                             ⟹  a promise, still
(begin (set! x 10)
       (force p))             ⟹  6
```

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

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

We define the expression

```
(delay ⟨expression⟩)
```

to have the same meaning as the procedure call

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

where `make-promise` is defined as follows:

```
(define make-promise
  (lambda (proc)
    (let ((result-ready? #f)
          (result #f))
      (lambda ()
```

```
      (if result-ready?
          result
          (let ((x (proc)))
            (if result-ready?
                result
                (begin (set! result-ready? #t)
                       (set! result x)
                       result)))))))
```

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

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

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

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

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

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

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

(list-length '(1 2 3 4))    ⟹  4

(list-length '(a b . c))    ⟹  #f
```

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

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

Most programming languages incorporate one or more special-purpose escape constructs with names like `exit`, `return`, or even `goto`. In 1965, however, Peter Landin [16] invented a general purpose escape operator called the J-operator. John Reynolds [21] 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.

7.10 Input and output

7.10.1 Ports

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

`(call-with-input-file` *string proc*`)` procedure
`(call-with-output-file` *string proc*`)` procedure
Proc should be a procedure of one argument, and *string* should be a string naming a file. For `call-with-input-file`, the file must already exist; for `call-with-output-file`, the effect is unspecified if the file already exists. These procedures call *proc* with one argument: the port obtained by opening the named file for input or output. If the file cannot be opened, an error is signalled. If the procedure returns, then the port is closed automatically and the value yielded by the procedure is returned. If the procedure does not return, then the port will not be closed automatically unless it is possible to prove that the port will never again be used for a read or write operation.
Rationale: Because Scheme's escape procedures have unlimited extent, it is possible to escape from the current continuation but later to escape back in. If the port were to be closed 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*`)` procedure
`(output-port?` *obj*`)` procedure
Returns `#t` if *obj* is an input port or output port respectively, otherwise returns `#f`.

`(current-input-port)` procedure
`(current-output-port)` 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*.

`(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 `void`.

7.10.2 Input

`(read)` procedure
`(read` *port*`)` procedure
`Read` converts external representations of Scheme objects into the objects themselves. That is, it is a parser for the nonterminal ⟨datum⟩ (see sections 8.1.2 and 7.3). `Read` returns the next object parsable from the given input *port*, updating *port* to point to the first character past the end of the external representation of the object.

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

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

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

`(peek-char)` procedure
`(peek-char` *port*`)` procedure

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

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

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

Returns `#t` if *obj* is an end of file object, otherwise returns `#f`. No end of file object can be read in using `read`.

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

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

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

7.10.3 Output

`(write` *obj*`)` procedure
`(write` *obj port*`)` procedure

Writes a written 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 `void`. The *port* argument may be omitted, in which case it defaults to the value returned by `current-output-port`.

`(display` *obj*`)` procedure
`(display` *obj port*`)` 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. Character objects appear in the representation as if written by `write-char` instead of by `write`. `Display` returns `void`. 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)` procedure
`(newline` *port*`)` procedure

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

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

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

7.10.4 System interface

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

`(load` *filename*`)` 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. The results of the expressions are not printed. The `load` procedure does not affect the values returned by `current-input-port` and `current-output-port`. `Load` returns `void`.

Chapter 8

Formal syntax and semantics

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

8.1 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⟩.

8.1.1 Lexical structure

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

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

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

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

⟨identifier⟩ ⟶ ⟨initial⟩ ⟨subsequent⟩*

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

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

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

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

⟨number⟩ ⟶ ⟨num 2⟩| ⟨num 8⟩
    | ⟨num 10⟩| ⟨num 16⟩

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

⟨num R⟩ ⟶ ⟨prefix R⟩ ⟨complex R⟩
⟨complex R⟩ ⟶ ⟨real R⟩ | ⟨real R⟩ @ ⟨real R⟩
    | ⟨real R⟩ + ⟨ureal R⟩ i | ⟨real R⟩ - ⟨ureal R⟩ i
    | ⟨real R⟩ + i | ⟨real R⟩ - i
    | + ⟨ureal R⟩ i | - ⟨ureal R⟩ i | + i | - i
⟨real R⟩ ⟶ ⟨sign⟩ ⟨ureal R⟩
⟨ureal R⟩ ⟶ ⟨uinteger R⟩
    | ⟨uinteger R⟩ / ⟨uinteger R⟩
    | ⟨decimal R⟩

⟨decimal 10⟩ ⟶ ⟨uinteger 10⟩ ⟨suffix⟩
| . ⟨digit 10⟩$^+$ #* ⟨suffix⟩
| ⟨digit 10⟩$^+$. ⟨digit 10⟩* #* ⟨suffix⟩
| ⟨digit 10⟩$^+$ #$^+$. #* ⟨suffix⟩
⟨uinteger R⟩ ⟶ ⟨digit R⟩$^+$ #*
⟨prefix R⟩ ⟶ ⟨radix R⟩ ⟨exactness⟩
| ⟨exactness⟩ ⟨radix R⟩

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

8.1.2 External representations

⟨Datum⟩ is what the `read` procedure (section 7.10.2) successfully parses. Note that any string that parses as an ⟨expression⟩ will also parse as a ⟨datum⟩.

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

8.1.3 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⟩)
    | (⟨test⟩)
    | (⟨test⟩ => ⟨recipient⟩)

⟨recipient⟩ ⟶ ⟨expression⟩
⟨case clause⟩ ⟶ `((`⟨datum⟩`*)` ⟨sequence⟩`)`

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

8.1.4 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⟩`)`
  | `'`⟨template 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 either an ⟨unquotation D⟩ or a ⟨splicing unquotation D⟩. The interpretation as an ⟨unquotation⟩ or ⟨splicing unquotation D⟩ takes precedence.

8.1.5 Programs and definitions

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

8.2 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 [25]; 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
$s \mathbin{\S} t$	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 $\mathtt{D}$

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

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

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

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

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

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

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

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

8.2.1 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

$$\begin{aligned}
\mathrm{Exp} \longrightarrow\ & \mathrm{K} \mid \mathrm{I} \mid \texttt{(}\mathrm{E}_0\ \mathrm{E}^*\texttt{)}\\
& \mid \texttt{(lambda (}\mathrm{I}^*\texttt{) }\Gamma^*\ \mathrm{E}_0\texttt{)}\\
& \mid \texttt{(lambda (}\mathrm{I}^*\texttt{ . }\mathrm{I}\texttt{) }\Gamma^*\ \mathrm{E}_0\texttt{)}\\
& \mid \texttt{(lambda }\mathrm{I}\ \Gamma^*\ \mathrm{E}_0\texttt{)}\\
& \mid \texttt{(if }\mathrm{E}_0\ \mathrm{E}_1\ \mathrm{E}_2\texttt{)} \mid \texttt{(if }\mathrm{E}_0\ \mathrm{E}_1\texttt{)}\\
& \mid \texttt{(set! }\mathrm{I}\ \mathrm{E}\texttt{)}
\end{aligned}$$

8.2.2 Domain equations

$\alpha \in \mathtt{L}$		locations
$\nu \in \mathtt{N}$		natural numbers
$\mathtt{T}$	$= \{\mathit{false},\ \mathit{true}\}$	booleans
$\mathtt{Q}$		symbols
$\mathtt{H}$		characters
$\mathtt{R}$		numbers
$\mathtt{E_p}$	$= \mathtt{L} \times \mathtt{L} \times \mathtt{T}$	pairs
$\mathtt{E_v}$	$= \mathtt{L}^* \times \mathtt{T}$	vectors
$\mathtt{E_s}$	$= \mathtt{L}^* \times \mathtt{T}$	strings
$\mathtt{M}$	$= \{\mathit{false},\ \mathit{true},\ \mathit{null},\ \mathit{undefined},\ \mathit{unspecified}\}$	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

8.2.3 Semantic functions

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

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

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

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

$$
\begin{aligned}
&\mathcal{E}[\![\texttt{(}\mathrm{E}_0\ \mathrm{E}^*\texttt{)}]\!] = \\
&\quad \lambda\rho\kappa\,.\,\mathcal{E}^*(\mathit{permute}(\langle\mathrm{E}_0\rangle \mathbin{\S} \mathrm{E}^*)) \\
&\qquad\quad \rho \\
&\qquad\quad (\lambda\epsilon^*\,.\,((\lambda\epsilon^*\,.\,\mathit{applicate}\,(\epsilon^* \downarrow 1)\,(\epsilon^* \dagger 1)\,\kappa) \\
&\qquad\qquad\quad (\mathit{unpermute}\,\epsilon^*)))
\end{aligned}
$$

$$
\begin{aligned}
&\mathcal{E}[\![\texttt{(lambda (}\mathrm{I}^*\texttt{)}\ \Gamma^*\ \mathrm{E}_0\texttt{)}]\!] = \\
&\quad \lambda\rho\kappa\,.\,\lambda\sigma\,. \\
&\qquad \mathit{new}\,\sigma \in \mathtt{L} \to \\
&\qquad\quad \mathit{send}\,(\langle \mathit{new}\,\sigma\,|\,\mathtt{L}, \\
&\qquad\qquad \lambda\epsilon^*\kappa'\,.\,\#\epsilon^* = \#\mathrm{I}^* \to \\
&\qquad\qquad\qquad \mathit{tievals}(\lambda\alpha^*\,.\,(\lambda\rho'\,.\,\mathcal{C}[\![\Gamma^*]\!]\rho'(\mathcal{E}[\![\mathrm{E}_0]\!]\rho'\kappa')) \\
&\qquad\qquad\qquad\qquad (\mathit{extends}\,\rho\,\mathrm{I}^*\,\alpha^*))
\end{aligned}
$$

$$
\begin{array}{l}
\qquad\qquad\qquad\qquad\quad \epsilon^*, \\
\qquad\qquad\qquad\quad \mathit{wrong}\ \text{“wrong number of arguments”}\rangle \\
\qquad\qquad\ \text{in}\ \mathtt{E}) \\
\qquad\qquad \kappa \\
\qquad\qquad (\mathit{update}\ (\mathit{new}\,\sigma \mid \mathtt{L})\ \mathit{unspecified}\,\sigma), \\
\qquad \mathit{wrong}\ \text{“out of memory”}\ \sigma
\end{array}
$$

$$
\begin{array}{l}
\mathcal{E}[\![\mathtt{(lambda\ (}\mathrm{I}^*\ \mathtt{.}\ \mathrm{I}\mathtt{)}\ \Gamma^*\ \mathrm{E}_0\mathtt{)}]\!] = \\
\quad \lambda\rho\kappa\,.\,\lambda\sigma\,. \\
\qquad \mathit{new}\,\sigma \in \mathtt{L} \rightarrow \\
\qquad\quad \mathit{send}\ (\langle \mathit{new}\,\sigma \mid \mathtt{L}, \\
\qquad\qquad\quad \lambda\epsilon^*\kappa'\,.\,\#\epsilon^* \geq \#\mathrm{I}^* \rightarrow \\
\qquad\qquad\qquad\qquad \mathit{tievalsrest} \\
\qquad\qquad\qquad\qquad\quad (\lambda\alpha^*\,.\,(\lambda\rho'\,.\,\mathcal{C}[\![\Gamma^*]\!]\rho'(\mathcal{E}[\![\mathrm{E}_0]\!]\rho'\kappa')) \\
\qquad\qquad\qquad\qquad\qquad\quad (\mathit{extends}\,\rho\ (\mathrm{I}^*\ \S\ \langle\mathrm{I}\rangle)\ \alpha^*)) \\
\qquad\qquad\qquad\qquad\quad \epsilon^* \\
\qquad\qquad\qquad\qquad\quad (\#\mathrm{I}^*), \\
\qquad\qquad\qquad\qquad \mathit{wrong}\ \text{“too few arguments”}\rangle\ \text{in}\ \mathtt{E}) \\
\qquad\qquad \kappa \\
\qquad\qquad (\mathit{update}\ (\mathit{new}\,\sigma \mid \mathtt{L})\ \mathit{unspecified}\,\sigma), \\
\qquad \mathit{wrong}\ \text{“out of memory”}\ \sigma
\end{array}
$$

$$
\mathcal{E}[\![\mathtt{(lambda}\ \mathrm{I}\ \Gamma^*\ \mathrm{E}_0\mathtt{)}]\!] = \mathcal{E}[\![\mathtt{(lambda\ (.}\ \mathrm{I}\mathtt{)}\ \Gamma^*\ \mathrm{E}_0\mathtt{)}]\!]
$$

$$
\begin{array}{l}
\mathcal{E}[\![\mathtt{(if}\ \mathrm{E}_0\ \mathrm{E}_1\ \mathrm{E}_2\mathtt{)}]\!] = \\
\quad \lambda\rho\kappa\,.\,\mathcal{E}[\![\mathrm{E}_0]\!]\ \rho\ (\mathit{single}\ (\lambda\epsilon\,.\,\mathit{truish}\,\epsilon \rightarrow \mathcal{E}[\![\mathrm{E}_1]\!]\rho\kappa, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathcal{E}[\![\mathrm{E}_2]\!]\rho\kappa))
\end{array}
$$

$$
\begin{array}{l}
\mathcal{E}[\![\mathtt{(if}\ \mathrm{E}_0\ \mathrm{E}_1\mathtt{)}]\!] = \\
\quad \lambda\rho\kappa\,.\,\mathcal{E}[\![\mathrm{E}_0]\!]\ \rho\ (\mathit{single}\ (\lambda\epsilon\,.\,\mathit{truish}\,\epsilon \rightarrow \mathcal{E}[\![\mathrm{E}_1]\!]\rho\kappa, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{send}\ \mathit{unspecified}\ \kappa))
\end{array}
$$

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

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

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

$$
\begin{array}{l}
\mathcal{E}^*[\![\mathrm{E}_0\ \mathrm{E}^*]\!] = \\
\quad \lambda\rho\kappa\,.\,\mathcal{E}[\![\mathrm{E}_0]\!]\ \rho\ (\mathit{single}(\lambda\epsilon_0\,.\,\mathcal{E}^*[\![\mathrm{E}^*]\!]\ \rho\ (\lambda\epsilon^*\,.\,\kappa\ (\langle\epsilon_0\rangle\ \S\ \epsilon^*))))
\end{array}
$$

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

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

8.2.4 Auxiliary functions

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

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

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

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

$single : (\mathtt{E} \to \mathtt{C}) \to \mathtt{K}$
$single =$
$\quad\lambda\psi\epsilon^* \,.\, \#\epsilon^* = 1 \to \psi(\epsilon^* \downarrow 1),$
$\qquad\qquad wrong$ "wrong number of return values"

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

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

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

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

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

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

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

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

$truish : \mathtt{E} \to \mathtt{T}$
$truish = \lambda\epsilon \,.\, \epsilon = false \to false, true$

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

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

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

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

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

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

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

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

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

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

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

$$\begin{array}{l}
\mathit{setcar} : \mathtt{E}^* \to \mathtt{K} \to \mathtt{C} \\
\mathit{setcar} = \\
\quad \mathit{twoarg}\,(\lambda \epsilon_1 \epsilon_2 \kappa \,.\, \epsilon_1 \in \mathtt{E_p} \to \\
\qquad\qquad\qquad (\epsilon_1 \mid \mathtt{E_p} \downarrow 3) \to \mathit{assign}\,(\epsilon_1 \mid \mathtt{E_p} \downarrow 1)
\end{array}$$

$$
\begin{array}{l}
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \epsilon_2 \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\mathit{send}\ \mathit{unspecified}\ \kappa), \\
\qquad\qquad\quad \mathit{wrong}\ \text{“immutable argument to \texttt{set-car!}”}, \\
\qquad\qquad\quad \mathit{wrong}\ \text{“non-pair argument to \texttt{set-car!}”})
\end{array}
$$

$$
\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} \wedge \epsilon_2 \in \mathtt{M}) \to \\
\qquad\qquad\qquad\qquad \mathit{send}\,(\epsilon_1 \mid \mathtt{M} = \epsilon_2 \mid \mathtt{M} \to \mathit{true}, \mathit{false})\kappa, \\
\qquad\qquad\qquad (\epsilon_1 \in \mathtt{Q} \wedge \epsilon_2 \in \mathtt{Q}) \to \\
\qquad\qquad\qquad\qquad \mathit{send}\,(\epsilon_1 \mid \mathtt{Q} = \epsilon_2 \mid \mathtt{Q} \to \mathit{true}, \mathit{false})\kappa, \\
\qquad\qquad\qquad (\epsilon_1 \in \mathtt{H} \wedge \epsilon_2 \in \mathtt{H}) \to \\
\qquad\qquad\qquad\qquad \mathit{send}\,(\epsilon_1 \mid \mathtt{H} = \epsilon_2 \mid \mathtt{H} \to \mathit{true}, \mathit{false})\kappa, \\
\qquad\qquad\qquad (\epsilon_1 \in \mathtt{R} \wedge \epsilon_2 \in \mathtt{R}) \to \\
\qquad\qquad\qquad\qquad \mathit{send}\,(\epsilon_1 \mid \mathtt{R} = \epsilon_2 \mid \mathtt{R} \to \mathit{true}, \mathit{false})\kappa, \\
\qquad\qquad\qquad (\epsilon_1 \in \mathtt{E}_\mathrm{p} \wedge \epsilon_2 \in \mathtt{E}_\mathrm{p}) \to \\
\qquad\qquad\qquad\qquad \mathit{send}\,((\lambda p_1 p_2 \,.\, ((p_1 \downarrow 1) = (p_2 \downarrow 1) \wedge \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad (p_1 \downarrow 2) = (p_2 \downarrow 2)) \to \mathit{true}, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{false}) \\
\qquad\qquad\qquad\qquad\qquad (\epsilon_1 \mid \mathtt{E}_\mathrm{p}) \\
\qquad\qquad\qquad\qquad\qquad (\epsilon_2 \mid \mathtt{E}_\mathrm{p})) \\
\qquad\qquad\qquad\qquad\qquad \kappa, \\
\qquad\qquad\qquad (\epsilon_1 \in \mathtt{E}_\mathrm{v} \wedge \epsilon_2 \in \mathtt{E}_\mathrm{v}) \to \ldots, \\
\qquad\qquad\qquad (\epsilon_1 \in \mathtt{E}_\mathrm{s} \wedge \epsilon_2 \in \mathtt{E}_\mathrm{s}) \to \ldots, \\
\qquad\qquad\qquad (\epsilon_1 \in \mathtt{F} \wedge \epsilon_2 \in \mathtt{F}) \to \\
\qquad\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\qquad\qquad \kappa, \\
\qquad\qquad\qquad\qquad \mathit{send}\ \mathit{false}\,\kappa)
\end{array}
$$

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

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

$$
\begin{array}{l}
\mathit{cwcc} : \mathtt{E}^* \to \mathtt{K} \to \mathtt{C} \qquad [\texttt{call-with-current-continuation}] \\
\mathit{cwcc} = \\
\quad \mathit{onearg}\,(\lambda \epsilon \kappa \,.\, \epsilon \in \mathtt{F} \to \\
\qquad\qquad\qquad (\lambda \sigma \,.\, \mathit{new}\,\sigma \in \mathtt{L} \to \\
\qquad\qquad\qquad\qquad\quad \mathit{applicate}\,\epsilon \\
\qquad\qquad\qquad\qquad\qquad\qquad \langle\langle \mathit{new}\,\sigma \mid \mathtt{L}, \lambda \epsilon^* \kappa' \,.\, \kappa \epsilon^* \rangle \text{ in } \mathtt{E} \rangle \\
\qquad\qquad\qquad\qquad\qquad\qquad \kappa \\
\qquad\qquad\qquad\qquad\qquad\qquad (\mathit{update}\,(\mathit{new}\,\sigma \mid \mathtt{L}) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{unspecified}
\end{array}
$$

$$\sigma),$$
$$\textit{wrong}\ \text{“out of memory”}\ \sigma),$$
$$\textit{wrong}\ \text{“bad procedure argument”})$$

8.3 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⟩)
      ⟨clause_2⟩ ...)
  ≡  (if ⟨test⟩
         (begin ⟨sequence⟩)
         (cond ⟨clause_2⟩ ...))

(cond (⟨test⟩)
      ⟨clause_2⟩ ...)
  ≡  (or ⟨test⟩ (cond ⟨clause_2⟩ ...))

(cond (⟨test⟩ => ⟨recipient⟩)
      ⟨clause_2⟩ ...)
  ≡  (let ((test-result ⟨test⟩)
           (thunk2 (lambda () ⟨recipient⟩))
           (thunk3 (lambda () (cond ⟨clause_2⟩ ...))))
       (if test-result
           ((thunk2) test-result)
           (thunk3)))

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

(cond)
  ≡  ⟨some expression returning void⟩

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

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

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

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

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

(or)              ≡  #f
(or ⟨test⟩)        ≡  ⟨test⟩
(or ⟨test1⟩ ⟨test2⟩ ...)
  ≡  (let ((x ⟨test1⟩)
           (thunk (lambda () (or ⟨test2⟩ ...))))
       (if x 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 $\langle\text{temp}_1\rangle$, $\langle\text{temp}_2\rangle$, ... are variables, distinct from $\langle\text{variable}_1\rangle$, ..., that do not free occur in the original ⟨init⟩ expressions, and ⟨undefined⟩ is an expression which returns something that when stored in a location makes it an error to try to obtain the value stored in the location. (No such expression is defined, but one is assumed to exist for the purposes of this rewrite rule.) The second `let` expression in the expansion is not strictly necessary, but it serves to preserve the property that the ⟨init⟩ expressions are evaluated in an arbitrary order.

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

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

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

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

```
(do ((⟨variable_1⟩ ⟨init_1⟩ ⟨step_1⟩)
     ...)
    (⟨test⟩ ⟨sequence⟩)
  ⟨command_1⟩ ...)
  ≡  (letrec ((⟨loop⟩
               (lambda (⟨variable_1⟩ ...)
                 (if ⟨test⟩
                     (begin ⟨sequence⟩)
                     (begin ⟨command_1⟩
                            ...
                            (⟨loop⟩ ⟨step_1⟩ ...))))))
       (⟨loop⟩ ⟨init_1⟩ ...))
```

where ⟨loop⟩ is any variable which is distinct from ⟨variable_1⟩, . . . , and which does not occur free in the `do` expression.

```
(let ⟨variable_0⟩ ((⟨variable_1⟩ ⟨init_1⟩) ...)
  ⟨body⟩)
  ≡  ((letrec ((⟨variable_0⟩ (lambda (⟨variable_1⟩ ...)
                                ⟨body⟩)))
        ⟨variable_0⟩)
      ⟨init_1⟩ ...)
```

```
(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 7.9.

Chapter 9

The Class System

The class system used in `Analysa` is derived from the Common Lisp Object System (CLOS) [14], [15]. It is called "tiny CLOS" and the acronym "tc" will be used to denote it. As in CLOS, a class is an Alscheme object that determines the structure and behavior of a set of other Alscheme objects, which are called its instances.

9.1 Class entry format

This chapter, similar to chapters 5 and 7, is organized into entries (cf. section 2.3.1). Recall that 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. Many entries begins with one or more header lines of the usual form

template *category*

followed by an explanation and example in most cases. In this chapter, the format for the categories becomes more complex, including "slot" and "generic" as well as "procedure" and "syntax". Slots are explained in section 9.3.1, and "generic" refers to generic methods, see section 9.5. Moreover, the classes themselves have similar entries of the form

class-name *class-derived-from* *instance-of-which-class*

The first name on the left is of course the name of the class being described. The second from the left is the class from which this one is derived. The last, on the right, is the class of which it is an "instance" (see section 9.3).

9.2 `tc.top:`

Everything must start somewhere, and in the Alscheme object systems it is the class `tc.top:`. Everything will be derived from this class. Its use will emerge as we describe everything else. Thus its class entry format takes the form

tc.top: *tc.class:*

That is, `tc.top:` is not derived from anything, but it is an instance of `tc.class`.

9.3 Instances

It may seem backward to describe instances before describing classes. But, as the reader will see, classes are themselves instances of a class called `tc.class:`. Instances are sometimes called objects. But they should not be confused with Scheme objects.

`tc.instance: tc.top:` tc.class:

We see that `tc.instance` is derived from the top class `tc.top`, and it is an instance of `tc.class`. Anything that is an instance of something will be derived from `tc.instance`. Note that `tc.top` is not derived from anything.

We make new classes with the following.

`(tc.make` *class* ⟨option⟩ ...`)`

Syntax: ⟨Option⟩ should be a pair of expressions of the form

name obj

where *name* is an Alscheme semantic keyword (section 3.1) and *obj* is any Scheme object.

Semantics: ⟨Class⟩ should the an instance of class `tc.class:` or of a class derived from `tc.class:` as described in the section 9.4. `Tc.make` does the following. First, it creates an instance of ⟨class⟩. Next, it calls the generic method (section 9.5.1) `tc.initialize` with the newly created instance and the list of options as arguments. Finally, `tc.make` returns the initialized instance.

9.3.1 Slots

Each instance has zero or more direct slots. A slot is either allocated or virtual. An allocated slot always has a storage location allocated to it, whereas a virtual slot points to a procedure to be executed. Slots are effectively the "indices" of a class if we think of a class as a "record." This will be made clear by looking at how one can access slots.

The following procedures allow access to slots.

`(slot-ref` *instncs slaht*`)` procedure

For allocated slots, `slot-ref` returns the Scheme object stored in the slot called *slaht* of *instncs*. *instncs* should be an instance and *slaht* should be a Scheme symbol.

For virtual slots, `slot-ref` returns the result of applying a user-supplied procedure to *instncs*.

`(slot-set!` *instncs slaht obj*`)` procedure

For allocated slots, `slot-set!` stores the Scheme object *obj* in the slot called *slaht* of *instncs*. *instncs* should be an instance and *slaht* should be a Scheme symbol.

For virtual slots, `slot-set!` calls a user-supplied procedure with *instncs* and *obj* as arguments.

9.3.2 An example

The following example shows how to construct a class of complex numbers. It illustrates the use of slots. It uses some concepts yet to be defined, but this will help motivate the full definitions. In section 9.5, we will continue this example by defining an addition operation `c+` and a multiplication operation `c*`.

```
(tc.define-class complex: (tc.top:)
  ([real-slot #'init-keyword #'real-part #'accessor Re]
   [imag-slot #'init-keyword #'imag-part #'accessor Im]
   [phas-slot #'allocation #'virtual  #'accessor Arg
              #'slot-ref (lambda (z) (atan (/ (Im z) (Re z))))
              #'slot-set! (lambda (z angl)
                    (slot-set! z 'real-slot (cos angl))
                    (slot-set! z 'imag-slot (sin angl)))]))
```

In this example, `real-slot` and `imag-slot` are allocated slots, whereas `phas-slot` is a virtual slot implemented by the corresponding lambda expression.

Although it is not striclty controlled in Alscheme, it is recommended to reference a slot by the "accessor" as shown in the example. That is, we write `(Re a)` as opposed to the corresonding slot. One could achieve the same result using `slot-ref` and `slot-set!`, but it is recommended to avoid this style of programming. However, we have illustrated this in the definition of `slot-set!` for `phas-slot`, where we refer to `z 'real-slot` instead of `Re z`. By using the "accessor" only, the keyword access can be made "public" whereas the slot name is kept private (and could be changed without changing the public name).

The phase, or `Arg` of `z` is accessed just like the `Re` and `Im` parts, i.e., `(Arg z)`. Note that divide by zero has not been accounted for the in the definition of `Arg`, and indeed it is not possible to define this uniquely in a continuous way for all complex numbers, cf. [18].

9.4 Classes

Classes are instances of class `tc.class:` which is an instance of itself. By convention, class names end in a colon. The complete specification of `tc.class:` is as follows.

`tc.class:  tc.instance:` — tc.class:

- `name  #'keyword #'name #'getter tc.name` — slot
- `direct-supers  #'keyword #'dsupers #'getter tc.direct-supers` — slot
- `direct-slots  #'keyword #'dslots #'getter tc.direct-slots` — slot
- `slots  #'getter tc.slots` — slot
- `cpl  #'getter tc.cpl` — slot
- `nfields  #'getter tc.nfields` — slot
- `initializers  #'getter tc.initializers` — slot
- `getters-n-setters  #'getter tc.getters-n-setters` — slot

Many of the above slots are self-explanatory. `cpl` stands for "class precedence list." `nfields` is the number of slots.

New classes can be defined by calling `tc.make` and specifying the appropriate slots, but the object system includes a special form `tc.define-class` which fills in the class slots.

`(tc.define-class` ⟨name⟩ ⟨supers⟩ ⟨slots⟩`)` — syntax
`(tc.define-class` ⟨name⟩ ⟨supers⟩ ⟨slots⟩ ⟨option⟩ ...`)` — syntax

Syntax: ⟨Name⟩ should be a Scheme identifier. ⟨Supers⟩ should be of the form

```
(⟨super⟩ ...)
```

where ⟨super⟩ is the name of a previously defined class. ⟨slots⟩ should be of the form

```
(⟨slot⟩ ...)
```

where ⟨slot⟩ is a Scheme identifier or of the form

```
(⟨slot name⟩ ⟨slot proprerties⟩)
```

where ⟨slot name⟩ is a Scheme identifier and ⟨slot properties⟩ is a sequence of pairs of the form

```
⟨property name⟩ ⟨value⟩
```

where ⟨property name⟩ is `#'init-keyword`,
`#'default`,
`#'getter`,
`#'setter`,
`#'accessor`,
`#'allocation`,
`#'slot-ref`, or
`#'slot-set!`.

Semantics: ⟨Name⟩ will be the name of the defined class. ⟨Supers⟩ will be the list of direct super-classes of the defined class; the order of the classes in this list is significant. ⟨Slots⟩ will be the direct slots of the defined class. ⟨Slot properties⟩ will determine the semantics of slot creation and access. The meaning of ⟨slot properties⟩ for a slot ⟨slot⟩ is as follows.

A slot property of the form

```
#'init-keyword key
```

where *key* is an Alscheme semantic keyword provides that, when an instance of ⟨class⟩ is made, `tc.make` can be given an option of the form

```
#'key obj
```

where *obj* is any Scheme object. And the meaning of such an option to make is that the value of ⟨slot⟩ will be initialized to *obj*.

A slot property of the form

```
#'default obj
```

where *obj* is any Scheme object, indicates that, in the absence of a `#'init-keyword` option for ⟨slot⟩ to `tc.make`, the value of ⟨slot⟩ will be initialized to *obj*.

A slot property of the form

```
#'getter ⟨name⟩
```

where ⟨name⟩ is any Scheme identifier means that `tc.define-class` will define a generic procedure called ⟨name⟩ which, when called with an instance *ins* of ⟨class⟩ as its single argument, will then call `tc.slot-ref` with arguments *ins* and ⟨slot⟩. The procedure called ⟨name⟩ is be defined in the scope where `tc.define-class` is called.

A slot property of the form

```
#'setter ⟨name⟩
```

where ⟨name⟩ is any Scheme identifier means that `tc.define-class` will define a generic procedure called ⟨name⟩ which, when called with an instance *ins* of ⟨class⟩ and a Scheme object *obj* as its two arguments (in this order), will call `tc.slot-set!` with arguments *ins*, ⟨slot⟩ and *obj*.

A slot property of the form

`#'accessor` ⟨name⟩

where ⟨name⟩ is any Scheme identifier means that `tc.define-class` will generate both a getter and a setter called ⟨name⟩ for ⟨slot⟩.

A slot property of the form

`#'allocation #'virtual`

means that no storage is allocated for ⟨slot⟩. Slots with this property are called virtual slots. For virtual slots, the following two slot properties are required.

A slot property of the form

`#'slot-ref` *proc*

where *proc* is a Scheme procedure with one parameter. When `tc.slot-ref` is called for ⟨slot⟩ of instance *ins* of ⟨class⟩, *proc* will be called with *ins* as argument. This property is allowed only for virtual slots.

A slot property of the form

`#'slot-set!` *proc*

where *proc* is a Scheme procedure with two parameters means that when `tc.slot-set!` is called for ⟨slot⟩ of instance *ins* of ⟨class⟩ and object *obj*, *proc* will be called with *ins* and *obj* as arguments (in that order). This property is allowed only for virtual slots.

Example: For an example of `tc.define-class`, see the class defined in section 9.3.2.

9.5 Methods

A method is an object which contains a procedure and type requirements for the arguments of that procedure. Methods are instances of class `tc.method:`.

`tc.method: tc.instance:` tc.class:

`name #'init-keyword #'name` slot

`Name` is the name of the method.

`specializers #'init-keyword #'specializers #'getter tc.specializers` slot

`Specializers` is a list of classes.

`procedure #'init-keyword #'procedure #'getter tc.procedure` slot

`Procedure` is a Scheme procedure.

9.5.1 Generic methods

A *generic method* is a special Scheme procedure. When a generic method is applied, it uses the classes of its arguments to choose a method apply its procedure to the arguments. Generic methods are instances of class `tc.generic:`.

`tc.generic:` `tc.instance:` tc.class:

`name` `#'init-keyword #'name` slot

`Name` is the name of the generic method.

`methods` `#'init-keyword #'methods` slot

`Methods` is a list of methods.

The class system includes special syntax for defining generic methods.

(`tc.define-method` *(*⟨name⟩ ⟨parameter⟩ ...*)*) syntax
(`tc.define-method` *(*⟨name⟩ ⟨parameter⟩ .... ⟨rest⟩*)*) syntax
Syntax: ⟨Name⟩ should be a Scheme identifier. ⟨Parameter⟩ should be a Scheme identifier or a list of the form

(*class* ⟨parm⟩)

where *class* is a class and ⟨parm⟩ is a Scheme identifier. ⟨Rest⟩ should be a Scheme identifier.

Semantics: If ⟨name⟩ is unbound, then it will be bound to a new (location containing an) object of class `tc.generic:`. If ⟨name⟩ is bound to an object of class other than `tc.generic:`, then it will be set to a new object of class `tc.generic:`. Once ⟨name⟩ is bound to a generic, a method with name ⟨name⟩ and specializers, determined by the parameter list given to `tc.define-method`, is added to the list of methods for that generic. The list of specializers is a list of classes determined by the parameter list as follows. For each parameter in the parameter list, there is a class c in the list of specializers. If a parameter is a pair, then c is the first element of the pair. If a parameter is a Scheme identifier, then c is `tc.top:`.

9.5.2 The `complex:` example, continued

Some natural "methods" for complex numbers are addition and multiplication. These can be implemented using the class defined in section 9.3.2 as follows.

```
(tc.define-method (c+ (complex: a) (complex: b))
  (make complex: #'real-part (+ (Re a) (Re b))
                 #'imag-part (+ (Im a) (Im b))))
(tc.define-method (c* (complex: a) (complex: b))
  (make complex:
     #'real-part (- (* (Re a) (Re b)) (* (Im a) (Im b)))
     #'imag-part (+ (* (Re a) (Im b)) (* (Im a) (Re b)))))
```

9.6 Specifity and Application of Generics

A specificity relation among classes is defined by the following two rules. *A class is more specific than its direct superclasses. Among the direct superclasses of a class, those listed earlier are more specific than those listed later.*

A specificity relation among methods is defined as follows. *A method f is more specific than a method g if every class on the list of specifiers for f is either equal to or more specific than the corresponding class on the list of specifiers for g.*

When a generic method is applied, a method, called the effective method, is chosen from among its methods. The procedure of the effective method is then called with the arguments originally passed to the generic method. The effective method is the most specific method among the methods applicable to the list of arguments. A method is applicable to a list of arguments if its list of specifiers has the same length as the list of arguments.

9.7 The Scheme classes

In order to access standard Scheme types (see section 7), we have to put class "wrappers" around them. These are written as follows.

`tc.boolean: tc.top:` tc.class:

The set of objects of class `tc.boolean:` is the same as the set of Scheme objects distinguished by `boolean?`.

`tc.char: tc.top:` tc.class:

The set of objects of class `tc.char:` is the same as the set of Scheme objects distinguished by `char?`.

`tc.list: tc.pair:` tc.class:

The set of objects of class `tc.list:` is the same as the set of Scheme objects distinguished by `list?`.

`tc.pair: tc.top:` tc.class:

The set of objects of class `tc.pair:` is the same as the set of Scheme objects distinguished by `pair?`.

`tc.null: tc.top:` tc.class:

The set of objects of class `tc.null:` is the same as the set of Scheme objects distinguished by `null?`.

`tc.string: tc.top:` tc.class:

The set of objects of class `tc.string:` is the same as the set of Scheme objects distinguished by `string?`.

`tc.symbol: tc.top:` tc.class:

The set of objects of class `tc.symbol:` is the same as the set of Scheme objects distinguished by `symbol?`.

`tc.vector: tc.top:` tc.class:
The set of objects of class `tc.vector:` is the same as the set of Scheme objects distinguished by `vector?`.

`tc.number: tc.top:` tc.class:
The set of objects of class `tc.number:` is the same as the set of Scheme objects distinguished by `number?`.

`tc.real: tc.number:` tc.class:
The set of objects of class `tc.real:` is the same as the set of Scheme objects distinguished by `real?`.

`tc.integer: tc.real:` tc.class:
The set of objects of class `tc.integer:` is the same as the set of Scheme objects distinguished by `integer?`.

`tc.keyword: tc.top:` tc.class:
The set of objects of class `tc.keyword:` is the same as the set of Scheme objects distinguished by `keyword?`.

`tc.void: tc.top:` tc.class:
The set of objects of class `tc.void:` is the same as the set of Scheme objects distinguished by `void?` (namely the single value `void`).

`tc.procedure: tc.top:` tc.class:
The set of objects of class `tc.procedure:` is the same as the set of Scheme objects distinguished by `procedure?`.

`tc.port: tc.top:` tc.class:
The set of objects of class `tc.port:` is the same as the set of Scheme objects `x` satisfying

```
(or (input-port? x) (output-port? x))
```

`tc.output-port: tc.port:` tc.class:
The set of objects of class `tc.output-port:` is the same as the set of Scheme objects distinguished by `output-port?`.

`tc.input-port: tc.port:` tc.class:
The set of objects of class `tc.input-port:` is the same as the set of Scheme objects distinguished by `input-port?`.

9.8 Constrained Classes

`tc.constrained-class: tc.class:` tc.class:

A constrained class is a class containing objects satisfying constraints. When you update one of the objects in a constraint, others involved in the constraint will automatically be updated. It allows the addition of "constraints" on slots of the form

`\var{oneslot}#'=\var{anotherslot}`

The following defines constrained classes.

`(tc.define-constrained-class` ⟨name⟩ ⟨supers⟩ ⟨slots⟩`)` syntax

Syntax: ⟨Name⟩ should be a Scheme identifier.

Example: A line in the plane is an example of a constrained set. We can implement this as follows, for the line $\{(x, y) : y = 2x\}$.

```
(tc.define-constrained-class line: ()
  (x)
  (y #'= (* 2 x))
)
```

The use of this constrained class is illustrated in the following.

```
(define p (make line: ))
(slot-set! p 'x 1)
(slot-ref p 'y)
```

By setting the `x` coordinate of the line `p`, we implicitly update the `y` coordinate of the point. Note that we could not create `line:` as a subclass of `math:r^2` since this would require lower-level access to the data structures. Constraints may only be applied to explicit slots.

Constrained slots and virtual slots are very closely related. In the `complex:` number example, we could have defined `Arg` as a constrained slot instead of a virtual slot. The constraints are "one-way" constraints, in that the slot on the left of the constraint symbol `#'= is slaved to (a function of) the other slot. Thus we can slot-set! p 'y and it will not update the value of slot-ref p 'x.`

9.9 Foreign objects

`There is a special class which supports the addition of C++ objects.`

`c++.top: tc.top:` `tc.constrained-class:`

`rep #'init-keyword #'rep #'getter C++.rep` slot

`This slot constains an Alscheme object which is a representation of a low-level foreign object.`

Let us give an example based again on complex numbers. Suppose we define a C++ class complex as follows in a file called complex.h.

```
   class complex{
     Real reel;
     Real imag;
   public:
     Real re().{return reel};
//al: Re~complex
     Real Im() {return imag};
//al: Im~complex
}
```

The ‘‘structured comments’’ beginning with //al: are an indicator to the program ptog which will generate appropriate ‘‘glue’’ to allow the linking of the compiled C++ code and Alscheme. More precisely, the command

```
ptog complex.h
```

will create a file complex_g.C which, when compiled, will link correctly with Alscheme.

The form of a structured comment is, in general, the stuctured comment symbol //al: followed by the name of a procedure followed by template information as needed. Above, a particular convention has been followed in that the name is taken to be of the form of a pair joined by a ‘‘tilde’’ where the first part of the name is the name of the function in the class, and the second half of the name is the name of the class itself.

To access the C++ complex number class in Alscheme, one uses the name given in the structured comment. So for example

```
(C++:Re~complex (c++.rep z))
```

Note that

```
        C++:Re~complex
```

operates on the low-level C++ representation of z, given by (c++.rep z).

Chapter 10

Some Global Aliases

There are several global aliases used in Alscheme that could cause potential conflict. We describe these here.

The following represent new display options.

(display *(tc.top: x) (tc.output-port: p)*) generic

This allows display of any class x to the specified output port p.

(display *(tc.top: x)*) generic

This allows display of any class x to the current output port. It simply calls the former display with the appropriate tc.output-port.

(display *(math.matrix: x) (tc.output-port: p)*) generic

This allows display of any matrix x to the specified output port p. Note that it is also possible to display (math.matrix: x) without speicifying tc.output-port: since the previous definition of (generic) display applies.

The name of a method can be either a simple symbol or a pair of the form setter <symbol>, as follows.

((setter car) *(pair: l) x*) generic

((setter cdr) *(pair: l) x*) generic

These are used to allow the definition of expressions such as

```
(set! (car list) 3)
```

Similarly, the following are used in algebraic objects.

((setter math._) *(math.vector: v) (tc.integer: i) x*) generic

((setter math._) *(math.matrix: m) (tc.integer: i) (tc.integer: j) x*) generic

These last two ''setters'' are used to implement an expression of the form

```
(set! (math._ v i) 17.4)
```

where v is a matrix or vector.

(void? x) procedure

This returns #t if x is #void; otherwise it returns #f.

(version) procedure

This returns the version of Alscheme that you are using.

Bibliography

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

[2] Carl deBoor. An empty exercise. *SIGNUM Newsletter* 25(4):2--6, 1990.

[3] William Clinger. How to read floating point numbers accurately. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 92--101. Proceedings published as *SIGPLAN Notices* 25(6), June 1990.

[4] R. Kent Dybvig. *The Scheme Programming Language.* Prentice-Hall, 1995.

[5] R. Kent Dybvig. *The Scheme Programming Language: ANSI Scheme* Prentice-Hall, 1996.

[6] Michael Eisenberg. Harold Abelson, editor. *Programming In Scheme.* Scientific Press, Redwood City, California, 1988.

[7] Michael Eisenberg and Harold Abelson. *Programming In Scheme.* MIT Press, 1990.

[8] Michael Eisenberg, with William Clinger and Anne Hartheimer. Harold Abelson, editor. *Programming In MacScheme.* Scientific Press, San Francisco, 1990.

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

[10] Daniel P. Friedman, Matthias Felleisen, Duane Bibby (Illustrator) *The Little Schemer*, 4th Edition. MIT Press, 1996.

[11] Daniel P. Friedman, Matthias Felleisen, Duane Bibby (Illustrator) *The Seasoned Schemer*, 4th Edition. MIT Press, 1996.

[12] Erick Gallesio. STk is a R4RS Scheme interpreter which can access the Tk graphical package. All of the commands defined by the Tk toolkit are available to the STk interpreter, and Tk variables are reflected back into Scheme as Scheme variables. Callback is expressed in Scheme. Includes a CLOS-like OO extension called STklos, which provides multiple inheritance, generic functions, multi methods, and a true meta-object protocol. A set of classes have been defined to manipulate Tk commands (menu, buttons, scales, canvas, canvas items) as Scheme objects. STk is available by anonymous ftp from kaolin.unice.fr:/pub/STk-2.1.tar.gz [193.48.229.225]

[13] *IEEE Standard 1178-1990. IEEE Standard for the Scheme Programming Language.* IEEE, New York, 1991.

[14] Sonya E. Keene. *Object-Oriented Programming in Common Lisp: A Programmer's Guide to CLOS.* Addison-Wesley, 1989.

[15] Jo A. Lawless and Molly M. Miller. *Understanding CLOS: the Common Lisp Object System.* Digital Press, 1991.

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

[17] Vincent S. Manis, James J. Little. *The Schematics of Computation (An Alan R. Apt Book)*. Prentice Hall, 1995.

[18] 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.

[19] Kent M. Pitman. The revised MacLisp manual (Saturday evening edition). MIT Laboratory for Computer Science Technical Report 295, May 1983.

[20] William Clinger, Jonathan Rees, *et al.*, editors. Revised^{4} Report on the Algorithmic Language Scheme. *Lisp Pointers* 4(3), 1991.

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

[22] George Springer and Daniel P. Friedman. *Scheme and the Art of Programming.* MIT Press and McGraw-Hill, 1989.

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

[24] Guy Lewis Steele Jr. and Jon L White. How to print floating point numbers accurately. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 112--126. Proceedings published as *SIGPLAN Notices* 25(6), June 1990.

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

Index

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