**Summary**

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

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

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

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

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

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

The report concludes with a list of references and an alphabetic index.
INTRODUCTION

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

Scheme was one of the first programming languages to incorporate first class procedures as in the lambda calculus, thereby proving the usefulness of static scope rules and block structure. A dynamcally typed language. Scheme was the first major dialect of Lisp to distinguish procedures from lambda expressions and symbols, to use a single lexical environment for all variables, and to evaluate the operator position of a procedure call the same way as an operand position. By relying on this feature, procedures can be used as arguments to other procedures, allowing for higher-order functions. Scheme was the first widely used programming language to embrace first class escape procedures, from which all previously known sequential control structures can be synthesized.

Background

The first description of Scheme was written in 1975 [28]. A revised report [25] appeared in 1978, which described the evolution of the language as its MIT implementation was upgraded to support an innovative compiler [26]. Three distinct projects began in 1981 and 1982 to use variants of Scheme for courses at MIT, Yale, and Indiana University [21,17,10]. An introductory computer science textbook using Scheme was published in 1984 [1].

As Scheme became more widespread, local dialects began to diverge until students and researchers occasionally found it difficult to understand code written at other sites. Fifteen representatives of the major implementations of Scheme therefore met in October 1984 to work toward a better and more widely accepted standard for Scheme. The report [4] was published at MIT and Indiana University in the summer of 1985. Further revisions took place in the spring of 1986 [23], and in the spring of 1988 [6]. The present report reflects further revisions agreed upon in a meeting at Xerox PARC in June 1992.

We intend this report to belong to the entire Scheme community, and so we grant perm ss on to copy t n whole or part withouth fee. In particular, we encourage implementors of Scheme to use this report as a start ng po nt for manuals and other documentat on, mod fy ng t as necessary.

Acknowledgements

We would like to thank the following people for their help: Alan Bawden, Mael Bla r, George Carrette, Andy Cromarty, Pavel Curt s, Jeff Dalton, Ol v er Danyv, Ken D ckey, Bruce Duba, Marc Feeley, Andy Freeman, R chard Gab r el, Yekta Gürsel, Ken Haase, Robert H eb, Paul Hudak, Morry Katz, Chr s L ndblad, Mark Meyer, J m M ller, J m Ph lb n, John Ramsdell, M ke Shaff, Jonathan Shap ro, Jul e Sussman, Perry Wagle, Dan el We se, Henry Wu, and Ozan Y g t. We thank Carol Fessenden, Dan el Fr edman, and Chr stopher Haynes for perm ss on to use text from the Scheme 311 vers on 4 reference manual. We thank Texas Instruments, Inc. for perm ss on to use text from the TI Scheme Language Reference Manual [30]. We gladly acknowledge the influence of manuals for MIT Scheme [17], T [22], Scheme 84 [11], Common L sp [27], and Algol 60 [18].

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

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

1.1. Semantics

The detailed informal semantics is the subject of chapters 3 through 6. For reference purposes, see section 7.2 for a formal semantic of Scheme.

Follow ng Algl, Scheme s a stat cally scoped program-
mg language. Each use of a var able s assoc ated w th a
lex cally cally of that var able.

Scheme has latent as opposed to man fest types. Types
are assoc ated w th values (also called objects) rather than
w th var ables. (Some authors refer to languages w th
latent types as weakly typed or dynam cally typed lan-
guages.) Other languages w th latent types are APL,
Snobol, and other d alects of L sp. Languages w th man-
fest types (somet mes referred to as strongly typed or sta-
cally typed languages) n clude Algol 60, Pascal, and C.

All objects created n the course of a Scheme computaton on,
nclud ng procedures and cont mut ons, have unl m ted ex-
ten. No Scheme object s ever destroyed. The reason that
implementat ons of Scheme do not (usually!) run out of
storage s that they are perm tted to recla m the storage
occup ed by an object f they can prove that the object
cannot poss bly matter to any future computaton on. Other
languages n wh ch most objects have unl m ted extent n-
clude APL and other L sp d alects.

Implementat ons of Scheme are requred to be properly
la-recurs ve. Th s allows the execut on of an terat ve
computaton on n constant space, even f the terat ve comput-
tag on s descr bed by a syntact cally recurs ve procedure.
Thus w th a properly la-recurs ve mplementaton on, ter-
at on can be expressed us ng the ord nary procedure-call
mechan cs, so that spec al terat on construct s are useful
only as syntact c sugar. See sect on 3.5.

Scheme procedures are objects n the r own r gh. Pro-
cedures can be created dynam cally, stored n data struc-
tures, returned as results of procedures, and so on. Other
languages w th these propert es n clude Common L sp and
ML.

One d st ngu sh ng feature of Scheme s that cont mut ons,
wh ch n most other languages only oper ate beh nd the
scenes, also have “first-class” status. Cont mut ons are
useful for implement ng a w de var ety of advanced control
construct s, n clude ng non-local ex ts, backtrac ng, and
coroutnes. See sect on 6.4.

Arguments to Scheme procedures are always passed by
value, wh ch means that the actual argument express ons
are evalu ated before the procedure g a ns control, whether
the procedure needs the result of the evaluat on or not.
1.3.2. Error situations and unspecified behavior

When speaking of an error s tu at on, th s report uses the phrase “an error s gna led to” nd cate that implement at on ons must detect and report the error. If such word ng does not appear, n the d scuss on of an error, then implement at on ons are not requ red to detect or report the error, though they are encouraged to do so. An error s tu at on that implement at on ons are not requ red to detect s usually referred to as an “error.”

For example, t s an error for a procedure to be passed an argument that the procedure s not expl c tly spec fie d to handle, even though such doma n errors are seldom men tioned in the report. Implement at on ons may extend a procedure’s doma n of defin t on to include such arguments.

Th s report uses the phrase “may report a violation of an implementation on restr ct on” to nd cate c rcumstances un der wh ch an implement at on on s perm it ed to report that t s unable to continue execut on of a correct program because of some restr ct on imposed by the implement at on. Implement at on on restr ct ons are of course d scour ed, but implement at on ons are encouraged to report v olat ons of implement at on on restr ct ons.

For example, an implement at on on may report a vol at on of an implement at on on restr ct on f t does not have enough storage to run a program.

If the value of an express on on s sa d to be “unspec fie d,” then the express on on must evaluate to some object w thout s gna led an error, but the value depends on the implement at on on; th s report expl c tly does not say what value should be returned.

1.3.3. Entry format

Chapters 4 and 6 are organ ized into entries. Each entry des c rs one language feature or a group of related features, where a feature s e ther a syntac c constr uct or a bu lin procedure. An entry begins with one or more header lines of the form

\[ \text{template} \quad \text{category} \]

for requ red, pr m ve features, or

\[ \text{template} \quad \text{qual fier} \quad \text{category} \]

where qual fier s e ther “librar y” or “opt onal” as defined in section 1.3.1.

If category s “syntax”, the entry des c rs an express on on type, and the template g ves the syntac c of the express on on type. Components of express ons are des c ned by syntac c c var ables, wh ch are wr tten us ng angle brackets, for example, \( \langle \text{express on} \rangle \). Var ables. Syntac c c var ables should be understood to denote segments of program text; for example, \( \langle \text{express on} \rangle \) stands for any str ng of characters wh ch s a syntac c cally val d express on. The notat on

\[ \langle \text{th ng}_1 \ldots \text{th ng}_n \rangle \]

nd cate zero or more occurrences of a \( \langle \text{th ng} \rangle \), and

\[ \langle \text{th ng}_1 \langle \text{th ng}_2 \ldots \rangle \rangle \]

nd cate one or more occurrences of a \( \langle \text{th ng} \rangle \). If category s “procedure”, then the entry des c rs a procedure, and the header l ne s a template for a call to the procedure. Argument names n the template are tal c ed. Thus the header l ne

\[ \text{(vector-ref vector k)} \]

procedure

nd cate that the bu lin procedure vector-ref takes two arguments, a vector and an exact non-negative integer k (see below). The header l ne

\[ \text{(make-vector k)} \]

procedure

\[ \text{(make-vector k fill)} \]

procedure

nd cate that the make-vector procedure must be defined to take either one or two arguments.

It s an error for an operat on on to be presented with an argument that t s not specifie d to handle. For succinctness, we follow the conven on on that f an argument name s also the name of a type l sted in sect on 3.2, then that argument must be of the named type. For example, the header l ne for vector-ref g ven above d cate s that the first argument to vector-ref must be a vector. The follow ng nam ng conven ons also mply type restr ct ons:

\[ \text{obj} \]

any object

\[ \text{l st, l st}_1, \ldots \text{l st}_j, \ldots \]

l st (see sect on 6.3.2)

\[ \text{z, z}_1, \ldots \text{z}_j, \ldots \]

complex number

\[ \text{x, x}_1, \ldots \text{x}_j, \ldots \]

real number

\[ \text{y, y}_1, \ldots \text{y}_j, \ldots \]

real number

\[ \text{q, q}_1, \ldots \text{q}_j, \ldots \]

rat onal number

\[ \text{n, n}_1, \ldots \text{n}_j, \ldots \]

nteger

\[ \text{k, k}_1, \ldots \text{k}_j, \ldots \]

exact non-negative nteger

1.3.4. Evaluating examples

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

\[ (\ast 5 8) \quad \Rightarrow 40 \]

means that the express on \( (\ast 5 8) \) evaluates to the object 40. Or, more precise, the express on on g ven by the sequence of characters “(\ast 5 8)” evaluates, n the env ronement, to an object that may be represented externally by the sequence of characters “40”. See sect on 3.3 for a discussion on external representation of objects.

1.3.5. Named convention names

By conven on, the names of procedures that always return a boolean value usually end in “?” Such procedures are called pred cates.
By convention, the names of procedures that store values into previously allocated locations (see section 3.4) usually end in "!". Such procedures are called mutation procedures. By convention, the value returned by a mutation on procedure s unspecified.

By convention, "->" appears within the names of procedures that take an object of one type and return an analogous object of another type. For example, 1st->vector takes a list and returns a vector whose elements are the same as those of the list.

2. Lexical conventions

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

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

2.1. Identifiers

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

lambda q
1st->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 alphabet characters:

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

See sect on 7.1.1 for a formal syntax of identifiers.

Identifiers have two uses within Scheme programs:

- Any identier may be used as a variable or as a syntactic keyword (see sections 3.1 and 4.3).
- When an identier appears as a literal or within a literal (see section 4.1.2), the sequence is used to denote a symbol (see section 6.3.3).

2.2. Whitespace and comments

Whitespace characters are spaces and newlines. (Implementations typically provide additional whitespace characters such as tab or page break.) Whitespace is used for improved readability and as necessary to separate tokens from each other, a token being an analogous object of another type. For example, 1st->vector takes a list and returns a vector whose elements are the same as those of the list.

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

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

2.3. Other notations

For a description of the notations used for numbers, see section 6.2.

+ - These are used for numbers, and may also occur anywhere an identier fier except as the first character. A del m ted plus or m nus sgn by itself is also an identier fier. A del m ted per od (not occurring w th n a number or identier fier) is used in the notation for pairs (sect on 6.3.2), and to nd cate a rest-parameter in a formal parameter list (sect on 4.1.4). A del m ted sequence of three success ve per ods is also an identier fier.

( ) Parentheses are used for group ng and to ndate list s (sect on 6.3.2).

' The single quote character is used to ndate literal data (sect on 4.1.2).

The backquote character is used to ndate almost-constant data (sect on 4.2.6).

The character comma and the sequence comma at sgn are used in conjunct on w th backquote (sect on 4.2.6).

" The double quote character is used to del m t strs (sect on 6.3.5).
Backslash is used in the syntax for character constants (sect on 6.3.4) and as an escape character within a string constant (sect on 6.3.5).

Left and right square brackets and curly braces and vertical bar are reserved for possible future extensions to the language.

Sharp signs are used for a variety of purposes depending on the character that immediately follows:

- These are the boolean constants (sect on 6.3.1).
- Ths introduces a character constant (sect on 6.3.4).
- Ths introduces a vector constant (sect on 6.3.6). Vector constants are named natively by.
- These are used n the notation for numbers (sect on 6.2.4).

### 3. Basic Concepts

#### 3.1. Variables, Syntax Constants, and Regions

A variable may name a type of syntax, or a variable may name a location where a value can be stored. An variable that names a type of syntax is called a syntax constant and is said to be bound to that syntax. A variable that names a location on a variable and is said to be bound to that location. The set of all values and ngs of effect at some point in a program is known as the environment.

The value stored n the location to which a variable is bound is called the variable's value. By abuse of terminology, the value of a variable at a location is called the variable's value. The value stored at a location is the variable bound at that location.

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 "#e28.000". The internal representation of an object is not necessarily unique. The integer 28 also has representations "#o12", "#x1c", and the list in the previous paragraph also has the external representation "(8 13)".

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

An external representation of an object obtained from the program text within which the binding is visible.

### 3.2. Disjointness of Types

No object satisfies more than one of the following predicates:

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

These predicates define the types boolean, symbol, number, char, string, pair, vector, and procedure. The empty list is a special object of its own type; its satisfies none of the above predicates.

Although there is a separate boolean type, any Scheme value can be used as a boolean value for the purpose of a conditional test. As explained in section 6.3.1, all values count as true such a test except for #f. Ths report uses the word "true" to refer to any Scheme value except #f, and the word "false" to refer to #f.

### 3.3. External Representations

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

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

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

An external representation of a program may be obtained n a program to obtain the corresponding object (see quote, sect on 4.1.2).
3. Basic concepts

3.4. Storage model

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

An object fetched from a locat on, by a var able reference or by a procedure as car, vector-ref, or string-ref, is equivalent in the sense of eqv? (sect on 6.1) to the object last stored in the locat on before the fetch.

Every locat on is marked to show whether it is in use. No var able or object ever refers to a locat on that is not in use. Whenever a th element of a storage structure is allocated for a var able or object, what is meant is that an appro piate number of locat ons are chosen from the set of locat ons that are not in use, and the chosen locat ons are marked to ndicate that they are now n use before the var able or object is made to denote them.

In many systems, the des rible rable for constants (i.e., the values of literal expressions) to reside in read-only-memory. To express that the subexpressions of a symbol are mmutable objects, while all objects created by the other procedures listed n th s report are mutable. It s an error to attempt to store a new value into a locat on that is denoted by an mmutable object.

3.5. Proper tal recurs on

Implenentat ons of Scheme are required to be properly tail-recursive. Procedure calls that occur in certain contexts defined below are ‘tail calls’. A Scheme implemenentat on s properly tail-recurs ve f it supports an unbounded number of act ve tal calls. A call is act ve f the called procedure may st ll return. Note that th s includes calls that may be returned from elsewhere by the current cont nuat on or by cont nuat ons captured earl er by call-w th-current-cont nuat on that are later invoked. In the absence of captured cont nuat ons, s calls could return at most once and the act ve calls would be those that had not yet returned. A formal defin on of proper tail recurs ons can be found n [8].

Rat onal:

Intu tively, no space is needed for an act ve tal call because the cont nuat on on that s used n the tal call has the same semant cs as the cont nuat on on passed to the procedure conta ng the call. Although an improper implementat on might use a new cont nuat on on the call, a return to th s new cont nuat on on would be followed nmately by a return to the cont nuat on on passed to the procedure. A properly tail-recurs ve implementat on returns to that cont nuat on ng.

Proper tal recurs ons was one of the central ideas n Steele and Sussman’s or gnal vers on of Scheme. The rst Scheme inter nterpreter implemented both funct ons and actors. Control ow was expressed using gn actors, wh ch d d ered from funct ons that they passed the results on to another actor instead of returning ng to a caller. In the terminology of th s sect on, each actor n shed wh a tal call to another actor.

Steele and Sussman later observed that n the r inter nterpreter the code for deal ng w th actors was dent al ng to that for funct ons and thus there was no need to include both n the language.

A tal call s a procedure call that occurs n a tal context. Tal contexts are defined ndrectly. Note that a tal context s always determined with respect to a part cular lambda express on.

• The last express on on w th n the body of a lambda express on, shown as (tal express on below, occurs n a tal context.

(lamb da (formals
  (defin t on * (express on * (tal express on )

• If one of the follow ng express ons s n a tal context, then the subexpress ons shown as (tal express on are n a tal context. These were derived from rules n
the grammar given in chapter 4.1.2. Literal expressions

4. Expressions

4.1. Primitive expression types

4.1.1. Variable references

4.1.2. Literal expressions

- If a cond expression on s n a t a l context, and has a clause of the form \((\text{expression on}_1 \Rightarrow \text{expression on}_2)\) then the (mpl ed) call to the procedure that results from the evalut on of \(\text{expression on}_1\) s n a t a l context.

\(\text{expression on}_2\) ts self s not n a t a l context.

- If a cond expression on s n a t a l context, and has a clause of the form \((\text{expression on}_1 \Rightarrow \text{expression on}_2)\) then the (mpl ed) call to the procedure that results from the evalut on of \(\text{expression on}_1\) s n a t a l context.

\(\text{expression on}_2\) ts self s not n a t a l context.

Cerra bu lt- n procedures are also requ red to perform t a l calls. The first argument passed to apply and to call-w th-current-cont nuat on, and the second argument passed to call-w th-values, must be called v a a t a l call. S m larly, eval must evaluate ts argument as f t were n t a l pos t on w th n the eval procedure.

In the follow ng example the only t a l call s the call to f. None of the calls to g or h are t a l calls. The reference to x s n a t a l context, but t s not a call and thus s not a t a l call.

\(\text{lambda }()\)

\(\text{(f g)}\)

\(\text{(let ((x (h)))}\)

\(\text{(and (g) (f)))}\)

\(\text{Note: Implementat ons are allowed, but not requ red, to recogn ze that some non-t a l calls, such as the call to h above, can be evaluated as though they were t a l calls. In the example above, the let express on could be comp led as a t a l call to h. (The poss b l ty of h return ng an unexpected number of values can be ignor ed, because n that case the effect of the let s expl c tly unspec fied and implementat on-dependent.)}\)

4.1. Primitive expression types

4.1.1. Variable references

\(\text{var able}\)

\(\text{synt}x\)

An express on on cons st ng of a var able (sect on 3.1) s a var able reference. The value of the var able reference s the value stored n the locat on to wh ch the var able s bound. It s an error to reference an unbound var able.

\(\text{def n s} 28\)

\(x\) \(\Rightarrow 28\)

4.1.2. Literal expression.

\(\text{quote (datum )}\)

\(\text{quote # (a b c )}\)

\(\text{quote (+ 1 2 )}\)
(quote (datum ) may be abbreviated as ’(datum ). The
two notat ons are equ valent in all respects.

’a               ⇒ a
’#(a b c)          ⇒ #(a b c)
’()              ⇒ ()
’(+ 1 2)         ⇒ (+ 1 2)
’(quote a)       ⇒ (quote a)
’’a              ⇒ (quote a)

Numerical constants, str ng constants, character constants,
and boolean constants evaluate “to themselves”; they need
not be quoted.

"abc"            ⇒ "abc"
"abc"            ⇒ "abc"
145932           ⇒ 145932
’t              ⇒ t
’t              ⇒ t
As noted in sect on 3.4, t s an error to alter a constant
(. e. the value of a literal express on) us ng a mutat on pro-
cedure like set-car! or str ng-set!.

4.1.3. Procedure calls

((operator (operand1 ...) syntax
A procedure call is written by simply enclosing in paren-
theses the procedure to be called and the arguments to be
passed to it. The operator and operand expressions are evalu-
ated (in an unspecified order) and the result ng procedure
is passed the resulting arguments.

’(+ 3 4)          ⇒ 7
((if #f + *) 3 4) ⇒ 12

A number of procedures are ava lable as the values of vari-
able s n the n tal env roment; for example, the add t on
and mult pl cat on procedures n the above examples are
the values of the var able s + and *. New procedures are
created by evaluat ng lambda express ons (see sect on 4.1.4).

Procedure calls may return any number of values (see values
sect on 6.4). W th the except on of values the procedures ava lable n the n tal env roment return one
value or, for procedures such as apply, pass on the values
returned by a call to one of the r arguments.

Procedure calls are also called comb nat ons.

Note: In many dialects of L sp, the empty comb nat on, (),
s a leg t mate express on. In Scheme, comb nat ons must have
at least one subexpress on, so () s not a syntact cally val d
express on.

4.1.4. Procedures

(lambda (formals (body ) syntax
Semant cs: A lambda express on evaluates to a procedure.
The env roment n effect when the lambda express on was
evaluated s remembered as part of the procedure. When the
procedure s later called with some actual arguments, the
env roment s which the lambda express on was evaluated
will be extended by n ng the var able s the formal
argument s to fresh locat ons, the correspond ng actual
argument values will be stored n ng those locat ons, and the
express ons the body of the lambda express on will be
evaluated sequent ally n the extended env roment. The
result(s) of the last express on n the body will be returned
as the result(s) of the procedure call.

(lambda (x) (+ x x))  ⇒ a procedure
(((lambda (x) (+ x x)) 4)  ⇒ 8

(def ne reverse-subtract
(lambda (x y) (− y x)))
(reverse-subtract 7 10)  ⇒ 3

(def ne add4
(let ((x 4))
  (lambda (y) (+ x y))))
(add4 6)  ⇒ 10

(Formals should have one of the follow ng forms:

• ((var able1 ...): The procedure takes a fixed num-
er of arguments; when the procedure s called, the
arguments w ll be stored n ng the correspond ng var able s.

• (var able : The procedure takes any number of arg-
ument s; when the procedure s called, the sequence
of actual arguments s converted nto a newly allo-
cated list, and the list s stored n ng the
(var able .

• ((var able1 ... (var ablen ) (var ablen+1)): If a
space-del m ted per od preceds the last var able, then
the procedure takes n or more arguments, where n
s the number of formal arguments before the per od
(there must be at least one). The value stored n ng the
n ng of the last var able w ll 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.)
4.1.5. Conditionals

\[
\text{Syntax:} \quad (f \langle \text{test} \rangle \langle \text{consequent} \rangle \langle \text{alternate} \rangle)
\]

\[
\text{Semantics: if test evaluates to true, the \text{consequent} is evaluated; if test evaluates to false, the \text{alternate} is evaluated.}
\]

\[
\begin{align*}
(f \langle \langle \text{test} \rangle \rangle \langle \langle \text{ consequent} \rangle \rangle \langle \langle \text{ alternate} \rangle \rangle) & \quad \Rightarrow \quad \text{true condition evaluates} \\
(f \langle \langle \text{test} \rangle \rangle \langle \langle \text{consequent} \rangle \rangle \langle \langle \text{ alternate} \rangle \rangle) & \quad \Rightarrow \quad \text{false condition evaluates, alternate evaluated}
\end{align*}
\]

4.1.6. Assignments

\[
\text{Syntax:} \quad (\text{set!} \langle \text{variable} \rangle \langle \text{expresson} \rangle)
\]

\[
\text{Semantics: assign variable to expression result.}
\]

\[
\begin{align*}
(\text{def ne x 2}) & \quad \Rightarrow \quad 3 \\
(\text{set!} x 1) & \quad \Rightarrow \quad 3 \\
(\text{set!} x 4) & \quad \Rightarrow \quad \text{undefined}
\end{align*}
\]

4.2. Derived expressions

The constructs in these sections are used in Scheme, as discussed in section 4.3. For reference purposes, section 7.3 gives macro definitions that will convert most of the constructs described in these sections to the previous section.
the (case s) is returned as the result(s) of the case expression. If the result of evaluating the last expression is not evaluated. If all expressions evaluate to false values, the value of the last expression is returned.

\[(\text{case} \ ((2 \ 3) \ 'pr me) \ (
(1 \ 4 \ 6 \ 8 \ 9) \ 'compos te)) \Rightarrow \text{compos te}
\]

\[(\text{case} \ ((\text{car} \ '(c \ d))) \ ((a) \ 'a) \ ((b) \ 'b)) \Rightarrow \text{unspec fied}
\]

\[(\text{case} \ ((\text{car} \ '(c \ d))) \ ((a \ e \ o \ u) \ 'vowel) \ ((w \ y) \ 'sem \ vowel) \ (\text{else} \ 'consonant)) \Rightarrow \text{consonant}
\]

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

The \(\text{test}\) expressions are evaluated from left to right, and the value of the first expression that evaluates to a true value (see sect on 6.3.1) 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.

\[(\text{and} \ ((= \ 2 \ 2) \ (> \ 2 \ 1)) \Rightarrow \#t
\]

\[(\text{and} \ ((= \ 2 \ 2) \ (< \ 2 \ 1)) \Rightarrow \#f
\]

\[(\text{and} \ 1 \ 2 \ 'c \ '(f \ g)) \Rightarrow \ (f \ g)
\]

\[(\text{and}) \Rightarrow \#t
\]

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

The \(\text{test}\) expressions are evaluated from left to right, and the value of the first expression that evaluates to a true value (see sect on 6.3.1) is returned. Any remaining expressions are not evaluated. If all the expressions evaluate to false values, the value of the last expression is returned. If there are no expressions then \#f is returned.

\[(\text{or} \ ((= \ 2 \ 2) \ (> \ 2 \ 1)) \Rightarrow \#t
\]

\[(\text{or} \ ((= \ 2 \ 2) \ (< \ 2 \ 1)) \Rightarrow \#t
\]

\[(\text{or} \ (#f \ #f) \Rightarrow \#f
\]

\[(\text{or} \ (\text{memq} \ 'b \ '(a \ b \ c)) \ ((/= \ 3 \ 0)) \Rightarrow \ (b \ c)
\]

4.2.2. B nd ng constructs

The three b nd ng constructs let, let*, and letrec give Scheme a block structure, like Algol 60. The syntax of the three constructs is different from the three constructs \(s\) dent cal, but they differ \(n\) the region \(s\) they establish for the \(r\) variable \(b\) nd ng. In a let expression, the \(n\) tal values are computed before any of the variables become bound; \(n\) a let* expression, the b nd ng and evaluat ng \(s\) are performed sequent ally; while a \(n\) letrec expression, all the b nd ng are \(n\) effect \(w\) le the \(r\) n tal values are be ng computed, thus allow ng mutually recurs ve definons.

\[\text{let} \ (b \ nd \ ngs \ 〈\text{body}〉) \Rightarrow \text{brary syntax}
\]

\[\text{Syntax:} \ 〈\text{body}〉 \should have the form \((\text{variable} \ 〈\text{n} \ t \ 〈\ldots〉)\),
\]

where each \(〈\text{n} \ t \ 〈\ldots〉)\ is an expression, and \(〈\text{body}〉\ should be a sequence of one or more expressions. It s an error for a \(〈\text{variable}〉\ to appear more than once \(n\) the lst of var ables be ng bound.

\[\text{Semant cs:} \ The \ (〈\text{n} \ t \ s) \ are evaluated \(n\) the current environ ment \(n\) some unspec fied order), the \(〈\text{variable}〉\ are bound to fresh locals hold ng the results, the \(〈\text{body}〉\ s are evaluated \(n\) the extended environ ment, and the value(s) of the last expression on of \(〈\text{body}〉\ s\) are returned. Each b nd ng of a \(〈\text{variable}〉\ has \(〈\text{body}〉\ as \(ts\) region.
\]

\[\text{let} \ ((x \ 2) \ (y \ 3)) \Rightarrow \text{brary syntax}
\]

\[\text{Syntax:} \ 〈\text{body}〉 \should have the form \((\text{variable} \ 〈\text{n} \ t \ 〈\ldots〉)\),
\]

and \(〈\text{body}〉\ should be a sequence of one or more expressions.

\[\text{Semant cs: Let} \ s\ is \ lar to \text{let}, but the \(b\ nd\ ng\ s\ are performed sequent ally from left to right, and the reg on of a \(b\ nd\ ng\ nd\ cated by \((\text{variable} \ 〈\text{n} \ t \ 〈\ldots〉)\) that part of the \text{let} expression on to the rght of the \(b\ nd\ ng\). Thus the second \(b\ nd\ ng\ s\ done \(n\) an environ ment \(n\) wh ch the first \(b\ nd\ ng\ s\ v s\ ble, and so on.
\]

\[\text{let} \ ((x \ 2) \ (y \ 3)) \Rightarrow \text{brary syntax}
\]

\[\text{let*} \ ((x \ 7)) \Rightarrow \text{brary syntax}
\]

\[\text{let*} \ ((x \ 7)) \Rightarrow \text{brary syntax}
\]

\[\text{let} \ ((x \ 2) \ (y \ 3)) \Rightarrow \text{brary syntax}
\]

\[\text{let*} \ ((x \ 7)) \Rightarrow \text{brary syntax}
\]

\[\text{let*} \ ((x \ 7)) \Rightarrow \text{brary syntax}
\]

\[\text{letrec} \ (b \ nd \ ngs \ 〈\text{body}〉) \Rightarrow \text{brary syntax}
\]

\[\text{Syntax:} \ 〈\text{body}〉 \should have the form \((\text{variable} \ 〈\text{n} \ t \ 〈\ldots〉)\),
\]

and \(〈\text{body}〉\ should be a sequence of one or more expressions.

\[\text{Semant cs:} \ The \(〈\text{variable}〉\ s\ are bound to fresh locals hold ng undefined values, the \(〈\text{n} \ t \ s)\ are evaluated \(n\) the
resulting environment, each binding of a variable is assigned to the result of the corresponding expression. Each iteration begins by evaluating the body of an iteration construct. It specifies a set of variables and their update on each iteration. When a variable is bound to a fresh location, the results of the expression as its region, making it possible to define mutually recursive procedures.

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

⇒ #t

One restriction on letrec is important: t must be possible to evaluate each (n t w thout assigning or referring to the value of any variable. If this restriction on s variable is violated, then s 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 (n t s are lambda expressions and the restriction is satisfied automatically.

### 4.2.3. Sequencing

(beg n (express on1 (express on2 ...) 1 brary syntax)

The (express on s are evaluated sequentially from left to right, and the value(s) of the last (express on s are returned. Th s express on type s used to sequence s de effects such as input and output.

(def ne x 0)

(beg n (set! x 5) (+ x 1))

⇒ 6

(beg n (d splay "4 plus 1 equals ")

d splay (+ 4 1))

⇒ unspecified and p rnts 4 plus 1 equals 5

### 4.2.4. Iteration

(let (var able (b nd ngs (body) 1 brary syntax "Named let" s a var ant on the syntax of let wh ch prov es a more general loop construct than do and may also be used to express recurs ons. It has the same syntax and s mans as ord nary let except that (var able s bound w th n (body) to a procedure whose formal arguments are the bound var ables and whose body s (body). Thus the exec ut on of (body) may be repeated by nvoke ng the procedure named by (var able).

(let loop ((numbers '(3 -2 1 6 -5)) (nonneg ')()) (neg '())

(cond ((null? numbers) (1 st nonneg neg)) ((>= (car numbers) 0)

(loop (cdr numbers) (cons (car numbers) nonneg) neg))) (<= (car numbers) 0))
4.2.5. Delayed evaluation

(delay (express on ) library syntax

The delay construct used together with the procedure force to implement lazy evaluation or call by need.

(delay (express on ) returns an object called a promise when called at some point in the future may be asked by the force procedure to evaluate (express on ) and deliver the result ng value. The effect of (express on return ng multivalued values unspecied.

See the description of force (sect on 6.4) for a more complete descrpt on of delay.

4.2.6. Quasiquotation

(quas quote (qq template ) syntax

“Backquote” or “quasquote” expres ons 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 with the (qq template , the result of evaluating the (qq template is equivalent to the result of evaluating the (qq template . If a comma appears with the (qq template , however, the express on on must evaluate to a list; the open ng and clos ng parentheses of the list are then “stripped away” and the elements of the list are inserted into the structure instead of the comma and the express on. If a comma appears followed mmled ately by an at sign (@), then the follow ng express on must evaluate to a list; the open ng and clos ng parentheses of the list are then “stripped away” and the elements of the list are inserted into the structure instead of the comma and the expression on.

A comma at-s gn should only appear with a nonsens list or vector (qq template .

`(1 st ,(+ 1 2) 4) \⇒ (1 st 3 4)

(let ((name 'a)) `(1 st ,name ',name))

\⇒ (1 st 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 24 3 8)

Quasiquote forms may be nested. Subst tions are made only for unquoted components appearing at the same nesting level as the outermost backquote. The nesting level increases by one ns de each success ng quasiquot on, and decreases by one ns de each unquot on.

`(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 two notations ‘qq template and (quas quote (qq template ) are dent cal all values. (express on s dent cal to (unquote (express on ) , and (express on s dent cal to (unquote-spl c ng (express on ). The external syntax generated by wr te for two-element lists whose car s one of these symbols may vary between implementations.

(quas quote (1 st (unquote (+ 1 2) )) 4))

\⇒ (1 st 3 4)

'((quas quote (1 st (unquote (+ 1 2) )) 4))

\⇒ '((1 st ,(+ 1 2) ))

.. , (quas quote (1 st (unquote (+ 1 2) )))

Unpredictable behavior can result if any of the symbols quas quote, unquote, or unquote-spl c ng appear n pos t ons with a (qq template otherw se than as descr bed ab ove.

4.3. Macros

Scheme programs can define and use new derived expres ons on types, called macros. Program-defined express on types have the syntax

`(keyword (datum ... )

where (keyword s an dent fier that un quely determnes the express on type. Th s dent fier s called the syntax c keyword, or s imply keyword, of the macro. The number of the (datum s, and the r syntax, depends on the express on type.

Each instance of a macro s called a use of the macro. The set of rules that spec fies how a use of a macro s transcr bed into a more pr m t ve express on s called the transformer of the macro.

The macro defin t on fact is ty on c sts of two parts:

- A set of express ons used to establish that certain dent nfers are macro keywords, assoc ate them with macro transformers, and control the scope w th n wh ch a macro s defined, and
- A pattern language for spec fy ng macro transformers.

The syntax c keyword of a macro may shadow var able b nd ns, and local var able b nd ns may shadow keyword b nd ns. All macros defined us ng the pattern language are “hygien c” and “referent ally transparent” and thus preserve Scheme’s lexical scope ng.
4.3.1. B nd ng constructs for syntactc keywords

Let-syntax and letrec-syntax are analogous to let and letrec, but they b nd syntactc keywords to macro transformers instead of b nd ng var ables to locat ons that contain values. Syntactc keywords may also be bound at top level; see sect on 5.3.

(let-syntax (b nd ng s (body ))
Syntac: B nd ng s should have the form

((keyword (transformer spec ...))
Each keyword s an dent fier, each (transformer spec s an instance of syntax-rules, and (body should be a sequence of one or more exp ress ons. It s an error for a keyword to appear more than once in the list of keywords be ng bound.

Semantics: The (body s expanded n the syntactc env ronment obta ned by extend ng the syntactc env ronment of the let-syntax express on w th macros whose keywords are the (keyword s, bound to the spec fie d transformers. Each b nd ng of a (keyword has (body as ts reg on.

(let-syntax ((when (syntax-rules ()
  ((when test stmt1 stmt2 ...)
   (f test
    (beg n stmt1 stmt2 ...)...)))))
  (let ((f #t))
    (when f (set! f 'now))
    f)) =⇒ now

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

(letrec-syntax (b nd ng s (body ))
Syntac: Same as for let-syntax.

Semantics: The (body s expanded n the syntactc env ronment obta ned by extend ng the syntactc env ronment of the letrec-syntax express on w th macros whose keywords are the (keyword s, bound to the spec fie d transformers. Each b nd ng of a (keyword has the (b nd ng s as well as the (body w th n ts reg on, so the transformers can transcribe expres ons nto uses of the macros nt roduced by the letrec-syntax express on.

(letrec-syntax
  ((my-or (syntax-rules ()
      ((my-or #f)
       ((my-or e) e)
       ((my-or e1 e2 ...) (let ((temp e1))
           (f temp
            temp
            (my-or e2 ...)))))))))
  (let ((x #f)
        (y 7)
        (temp 8)
        (let odd?)
            (f even?))
    (my-or x
          (let temp)
          (f y)
          y))) =⇒ 7

4.3.2. Pattern language

A (transformer spec has the follow ng form:

(syntax-rules (l terals (syntax rule ...)))

Syntax: L terals s a lst of dent fiers and each (syntax rule should be of the form

((pattern (template ))

The (pattern s a (syntax rule s a lst (pattern that begins w th the keyword for the macro.

A (pattern s ether an dent fier, a constant, or one of the follow ng

((pattern ...)
 ((pattern (pattern ... (pattern ))
 (pattern ... (pattern )
 #((pattern ...)
 #((pattern ... (pattern )

and a template s ether an dent fier, a constant, or one of the follow ng

((element ...)
 ((element (element ... (template ))
 #((element ...)

where an (element s a (template opt onally followed by an (ell ps s and an (ell ps s the dent fier “...” (wh ch cannot be used as an dent fier n ether a template or a pattern).

Semantics: An instance of syntax-rules produces a new macro transformer by spec fy ng a sequence of hygien c
More formally, an input form \langle syntax-rules \rangle s matched against the patterns contained in the \langle syntax rule \rangle s, beginnging with the leftmost \langle syntax rule \rangle. When a match is found, the macro use is transcribed beddy ey onally accord ng to the template.

An dent fier that appears in the pattern of a \langle syntax rule \rangle s a pattern var able, unless t s the keyword that begins the pattern, s lsted n (lterals , or s the dent fier “...”. Pattern var ables match arbitrary nput elements and are used to refer to elements of the nput n the template. It is an error for the same pattern var able to appear more than once n a (pattern).

The keyword at the beginng of the pattern n a \langle syntax rule \rangle s not involved n the match ng and s not cons dered a pattern var able or lteral dent fier.

*Rationale:* The scope of the keyword is determined by the \langle syntax rule \rangle.

Identiers that appear in \langle syntax rule \rangle s not involved in the matchng and s not cons dered a pattern var able or lteral dent fier.

A subform in the nput matches a lteral dent fier f and only f t s an dent fier and e ther both ts occurrence n the macro express on and ts occurrence n the macro derin t on have the same lex cal b nd ng, or the two dentiers are equal and both have no lex cal b nd ng.

A subpattern followed by ... can match zero or more elements of the nput. It is an error for ... to appear n (lters . W th n a pattern the dent fier ... must follow the last element of a nonempty sequence of subpatterns.

More formally, an nput form F matches a pattern P f and only f:

- \( P \) s a non-literal lteral dent fier; or
- \( P \) s a literal lteral dent fier and \( F \) s an dent fier w th the same b nd ng; or
- \( P \) s a list \((P_1 \ldots P_n)\) and \( F \) s a list of \( n \) forms that match \( P_1 \) through \( P_n \), respectvely; or
- \( P \) s an improper list \((P_1 P_2 \ldots P_n P_{n+1})\) and \( F \) s a list or improper list of \( n \) or more forms that match \( P_1 \) through \( P_n \), respectvely, and whose nth “cdr” matches \( P_{n+1} \); or
- \( P \) s of the form \((P_1 \ldots P_n P_{n+1})\) where \( \langle \text{ell ps s s the dent fier ... and F s a proper list of at least n forms, the first n of wh ch match P_j through P_{n+1}, respectvely, and each rema} n ng element of F matches P_{n+1}; or

- \( P \) s a vector of the form \#(\( P_1 \ldots P_n \)) and \( F \) s a vector of \( n \) forms that match \( P_1 \) through \( P_n \); or
- \( P \) s of the form \#(\( P_1 \ldots P_n P_{n+1} \langle \text{ell ps s s the dentier ... and F s a vector of n or more forms the first n of wh ch match P_j through P_{n+1}, respectvely, and each rema} n ng element of F matches P_{n+1}; or

- \( P \) s a datum and \( F \) s equal to \( P \) n the sense of the equal? procedure.

It is an error to use a macro keyword, w th n the scope of ts b nd ng, n an express on that does not match any of the patterns.

When a macro use s transcribe beddy ey on the template of the match ng \langle syntax rule \rangle, pattern var ables that occur n the template are replaced by the subforms they match n the nput. Pattern var ables that occur n subpatterns followed by one or more instances of the dent fier ... are allowed only n subtemplates that are followed by as many instances of .... They are replaced n the output by all of the subforms they match n the nput, d str buted as nd cated. It s an error f the output cannot be buit up as spec fied.

Identiers that appear n the template but are not pattern var ables or the dent fier ... are inserted nto the output as lteral dentiers. If a lteral dent fier s inserted as a free dent fier then t refers to the b nd ng of that dent fier w th n whose scope the instance of \langle syntax rules \rangle appears. If a lteral dent fier s inserted as a bound dent fier then t s n effect renamed to prevent nadvertent captures of free dentiers.

As an example, \( \text{let} \) and \( \text{cond} \) are defined as n sect on 7.3 then they are hyg en c (as requ red) and the follow ng s not an error.

\[
\begin{align*}
(\text{let } ((\Rightarrow \#f)) & \quad (\text{cond } (\#t \Rightarrow 'ok)))
\end{align*}
\]

The macro transformer for \( \text{cond} \) recog nes \( \Rightarrow \) as a local var able, and hence an express on, and not as the top-level dent fier \( \Rightarrow \), wh ch the macro transformer treats as a syntact c keyword. Thus the example expands nto

\[
\begin{align*}
(\text{let } ((\Rightarrow \#f)) & \quad (\text{if } \#t \text{ beg n } 'ok))
\end{align*}
\]

Instead of

\[
\begin{align*}
(\text{let } ((\Rightarrow \#f)) & \quad (\text{let } ((\text{temp } \#t))
\end{align*}
\]

\[
\begin{align*}
(\text{if } \text{temp } 'ok \text{ temp}))
\end{align*}
\]

wh ch would result n an nal d procedure call.
5. Program structure

5.1. Programs

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

Programs are typically stored in files or entered interactively, 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.)

Defin t ons and syntax definitions occurring at the top level of a program can be interpreted declaratively. They cause bindings to be created in the top level environment or modified; they are executed in order when the program is invoked or loaded, and typically perform some kind of computation.

At the top level of a program (beg n (form1 ... )) s equivalent to the sequence of expressions, definitions, and syntax definitions that form the body of the beg n.

5.2. Defin t ons

Defin t ons are valid in some but not all contexts where expressions are allowed. They are valid only at the top level of a program and at the beginning of a body. A defin t on should have one of the following forms:

- (def ne (var able (express on )
- (def ne ((var able (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 on). Th s form s equavalent to

  (def ne (var able (lambda ((formals ) (body ))).

- (def ne ((var able . (formal ) (body )
  (Formal should be a single variable. Th s form s equavalent to

  (def ne (var able (lambda (formal (body ))).

5.2.1. Top level defin t ons

At the top level of a program, a defin t on

  (def ne (var able (express on )
has essentially the same effect as the assignment expression

  (set! (var able (express on )
if (var able s bound. If (var able s not bound, however, then the defin t on will bind (var able to a new location before performing the assignment, whereas the would be an error to perform a set! on an unbound variable.

  (def ne add3
    (lambda (x) (+ x 3)))
  (add3 3) => 6
  (def ne f rst car)
  (f rst '(1 2)) => 1

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

5.2.2. Internal defin t ons

Defin t ons may occur at the beg nn of a body (that is, the body of a lambda, let, let*, letrec, let-syntax, or letrec-syntax) express on or that of a defin t on of an appropriate form). Such defin t ons are known as internal defin t ons as opposed to the top level defin t ons described above. The variable defined by an internal defin t on is local to the body. Th s, (var able s bound rather than assigned, and the region of the body. For example,

  (let ((x 5))
    (def ne foo (lambda (y) (bar x y)))
    (def ne bar (lambda (a b) (+ (* a b) a)))
    (foo (+ x 3))) => 45

A body containg internal defin t ons can always be converted into a completely equivalent letrec express on. For example, the let express on in the above example s equivalent to

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

Just as for the equavalent letrec express on, t must be possible to evaluate each express on of every internal defin t on a body without assigning or referring to the value of any variable be ng defined.

Wherever an internal defin t on may occur (beg n (defin t on1 ... ) s equavalent to the sequence of definitions that form the body of the beg n.
5.3. Syntax definitions

Syntax definitions are valid only at the top level of a program. They have the following form:

```
(define-syntax ⟨keyword⟩ ⟨transformer spec⟩)
```

(Keyword is an identifier, and the ⟨transformer spec⟩ should be an instance of syntax-rules. The top-level syntactic environment is extended by binding the ⟨keyword⟩ to the specified transformer.

There is no define-syntax analogue of internal definitions.

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

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

6. Standard procedures

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

A program may use a top-level define t on to b nd ng any var - able. It may subsequently alter any such b nd ng by an ass gnment (see §4.1.6). These operat ons do not mod fy the behav or of Scheme’s bu lt- n procedures. Alter ng any top-level b nd ng that has not been int rod uc ed by a defin - t on has an unspec i ed effect on the behav or of the bu lt- n procedures.

6.1. Equivalence predicates

A pred cate s a procedure that always returns a boolean value (#t or #f). An equivalence predicate s the computa t onal analogue of a mathemat cal equa l valuet on on ( t s symmetr c, reflex ve, and trans t ve). Of the equa l valuet on predicates descr bed n th s sect on, eq? s the finest or most d scr m nat ng, and equal? s the coarsest. Eqv? s sl ghtly less d scr m nat ng than eq?.

```
(eqv? obj1 obj2)
```

procedure

The eqv? procedure defines a useful equivalence relation on objects. Br ely, t returns #t f obj1 and obj2 should normally be regarded as the same object. Th s relat on s left sl ghtly open to int erpretat on, but the follow ng part al spec fic at on of eqv? holds for all implementat ons of Scheme.

The eqv? procedure returns #f f:

- obj1 and obj2 are both #t or both #f.
- obj1 and obj2 are both symbols and

```
(str ng= (symbol->str ng obj1)
  (symbol->str ng obj2))
```

⇒ #t

Note: Th s assumes that ne ther obj1 nor obj2 s an “un- interned symbol” as alluded to n sect on 6.3.3. Th s re port does not presume to spec fy the behav or of eqv? on implementat on-depend ent extens ons.

- obj1 and obj2 are both numbers, are numer cally equal (see =, sect on 6.2), and are e ther both exact or both nexact.
- obj1 and obj2 are both characters and are the same character accord ng to the char=? procedure (sect on 6.3.4).
- both obj1 and obj2 are the empty l st.
- obj1 and obj2 are pa rs, vec tors, or str ng s that denote the same locat ons n the store (sect on 3.4).
- obj1 and obj2 are procedures whose locat on on tags are equal (sect on 4.1.4).

The eqv? procedure returns #f f:

- obj1 and obj2 are of dfferent types (sect on 3.2).
• one of \( \text{obj}_1 \) and \( \text{obj}_2 \) is \#t but the other is \#f.

• \( \text{obj}_1 \) and \( \text{obj}_2 \) are symbols but

\[
\text{(str \ ng=? \ (symbol->str \ ng \ \text{obj}_1))}
\text{(symbol->str \ ng \ \text{obj}_2))} \implies \#f
\]

• one of \( \text{obj}_1 \) and \( \text{obj}_2 \) is an exact number but the other is an inexact number.

• \( \text{obj}_1 \) and \( \text{obj}_2 \) are numbers for which the \( = \) procedure returns \#f.

• \( \text{obj}_1 \) and \( \text{obj}_2 \) are characters for which the \text{char=?} procedure returns \#f.

• one of \( \text{obj}_1 \) and \( \text{obj}_2 \) is the empty list but the other is not.

• \( \text{obj}_1 \) and \( \text{obj}_2 \) are pairs, vectors, or str ngs that denote d st nct locat ons.

• \( \text{obj}_1 \) and \( \text{obj}_2 \) are procedures that would behave d fferently (return d fferent value(s) or have d fferent s de effects) for some arguments.

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

The follow ng examples illustrate cases n wh ch the above rules do not fully spec fy the behav or of \text{eqv?}. All that can be sa d about such cases s that the value returned by \text{eqv?} must be a boolean.

\[
\begin{align*}
\text{(eqv? "\#\#")} & \implies \text{unspec fied} \\
\text{(eqv? '(\#\#) '(\#\#))} & \implies \text{unspec fied} \\
\text{(eqv? (lambda (x) x))} & \implies \text{unspec fied} \\
\text{(eqv? (lambda (x) x))} & \implies \text{unspec fied}
\end{align*}
\]

The next set of examples shows the use of \text{eqv?} w th procedures that have local state. Gen-counter must return a d st nct procedure every t m, s nce each procedure has ts own nternal counter. Gen-loser, however, returns equv alent procedures each t m, s nce the local state does not affect the value or s de effects of the procedures.

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

\[
\begin{align*}
\text{(eqv? #f 'nil)} & \implies \#f \\
\text{(let ((p (lambda (x) x))) (eqv? p p))} & \implies \#t
\end{align*}
\]

The next set of examples shows the use of \text{eqv?} w th procedures that have local state. Gen-counter must return a d st nct procedure every t m, s nce each procedure has ts own nternal counter. Gen-loser, however, returns equv alent procedures each t m, s nce the local state does not affect the value or s de effects of the procedures.

\[
\begin{align*}
\text{Gen-counter} \quad &\text{def ne}\quad \text{(lambda () )} \\
\text{Gen-loser} \quad &\text{def ne}\quad \text{(lambda () )} \\
\text{Gen-counter} \quad &\text{def ne}\quad \text{(lambda () )}
\end{align*}
\]

Rationale: The above definiton of \text{eqv?} allows implementat ons latude n the r treatment of procedures and l terals: implementat ons are free ether to detect or to fail to detect that two procedures or two l terals are equ valent to each other, and can dec de whether or not to merge representat ons of equ valent objects by us ng the same po nter or b t pattern to represent both.

\[
\begin{align*}
\text{(eqv? 'a) 'a)} & \implies \text{unspec fied} \\
\text{(eqv? "a" "a")} & \implies \text{unspec fied} \\
\text{(eqv? '(b) (cdr '(a b)))} & \implies \text{unspec fied} \\
\text{(let ((x '(a))) (eqv? x x))} & \implies \#t
\end{align*}
\]

Eq? s s m lar to \text{eqv?} except that n some cases t s capable of d scern ng d st nct ons finer th an those detectable by \text{eqv?}.

Eq? and \text{eqv?} are guaranteed to have the same behav or on symbols, booleans, the empty l st, pa rs, procedures, and non-empty str ngs and vectors. Eq?’s behav or on numbers and characters s implementat on-depdendent, but t w ll always return e ther true or false, and w ll return true only when \text{eqv?} would also return true. Eq? may also behave d fferently from \text{eqv?} on empty vectors and empty str ngs.
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. Ths report uses the types `number`, `complex`, `real`, `rational`, and `integer` to refer to both mathematical numbers and Scheme numbers. Mach ne representation ons such as fixed point and float ng points are referred to by names such as `fixnum` and `flonum`.

6.2.1. Numerical types
Mathematically, numbers may be arranged into a tower of subtypes n which each level s a subset of the level above t:

- `number`
- `complex`
- `real`
- `rational`
- `integer`

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

There s no simple relationship between a number’s type and its representation on a modern computer. Although most implementations of Scheme will offer at least two different representations for numbers, the Scheme numbers that model numbers in practice may be approximated by rational and therefore inexact approximations. In order to catch uses of inexact numbers where exactness is desired, the report uses the names `number?`, `number?`, `rational?`, `rational?`, and `nteger?`.

Scheme’s numerical operations treat numbers as abstract data, as independent of the representation on a computer. Although an implementation of Scheme may use `fixnum`, `flonum`, and perhaps other representations for numbers, the report 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 to be represented exactly and those that may not be. For example, indexes into data structures must be known to be represented exactly and those that may not be. For example, indexes into data structures must be known to be represented exactly and those that may not be. For example, indexes into data structures must be known to be represented exactly and those that may not be. For example, indexes into data structures must be known to be represented exactly and those that may not be.
the whole tower of subtypes given in section Implementations of Scheme are not required to implement an exact zero may produce an exact zero result, even if the arguments. For example, multiplication of any number by value of the result is unaffected by the inexactness of its however, return an exact result if it can prove that the implementation must generally return inexact results inexact->exact With the exception of the inexact->exact procedure reported the violation of an implementation restriction or it is unable to produce an exact result, then t may either return exact results when given exact arguments. If one of these procedures s unable to deliver an exact result when g ven exact arguments, then t may either report a value on an implementat on or t may s lently coerce ts result to an nleaf exact value. See section 6.2.3.

W th the except on of nleaf->exact, the operat ons described in section must generally return zero exact results when g ven any nleaf exact arguments. An operat on may, however, return an exact result f t can prove that the value of the result s unaffected by the nleafness of ts arguments. For example, multiple pl at on of any number by an exact zero may produce an exact zero result, even f the other argument s nleaf.

6.2.3. Implementat on restr ct ons

Implementat ons of Scheme are not required to implement the whole tower of subtypes g ven n sect on 6.2.1, but they must implement a coherent subset consistent with both the purposes of the implementat on and the str t of the Scheme language. For example, an implementat on on n wh ch all numbers are real may st ll be qu te useful.

Implementat ons may also support only a limited range of numbers of any type, subject to the requirements of th s sect on. The supported range for exact numbers of any type may be different from the supported range for nleaf exact numbers of that type. For example, an implementat on on that uses flonums to represent all ts nleaf real numbers may support a prac tally unbounded range of exact ntegers and rat onals wh l m t ng the range of exact real numbers (and therefore the range of nleaf ntegers and rat onals) to the dynam c range of the flonum format. Furthermore the gaps between the representable nleaf ntegers and rat onals are l kely to be very large n such an implementat on as the l m ts of th s range are approached.

An implementat on on of Scheme must support exact ntegers throughout the range of numbers that may be used for nx exes of l st s, vectors, and str ng s or that may result from comput ng the length of a l st, vector, or str ng. The length, vector-length, and str ng-length procedures must return an exact nteger, and t s an error to use anything but an exact nteger as an index. Furthermore any nteger constant w th n the index range, f expressed by an exact nteger syntax, w ll indeed be read as an exact nteger, regardless of any implementat on on restr ct ons that may apply outs de th s range. F nally, the procedures l st ed below w ll always return an exact nteger result prov ded all the r arguments are exact ntegers and the mathemat cally expected result s representable as an exact nteger w th n the implementat on:

\[
\begin{align*}
\text{+} & \quad \text{quote} & \quad \text{-} & \quad \text{ent} \\
\text{*} & \quad \text{remainder} & \quad \text{modulo} \\
\text{max} & \quad \text{f} & \quad \text{abs} \\
\text{numerator} & \quad \text{denominator} & \quad \text{gcd} \\
\text{lcm} & \quad \text{floor} & \quad \text{ceil} \\
\text{truncat} & \quad \text{round} & \quad \text{rationalize} \\
\text{expt} & \quad \text{exact} \\
\end{align*}
\]

Implementat ons are encouraged, but not required, to support exact ntegers and exact rat onals of pract cally unlimited size and precision, and to implement the above procedures and the \text{/} procedure n such a way that they always return exact results when g ven exact arguments. If one of these procedures s unable to deliver an exact result when g ven exact arguments, then t may either report a value on an implementat on or t may s lently coerce ts result to an nleaf number. Such a coer c on may cause an error later.

An implementat on on may use float ng po nt and other appro ximate represent at on on str at eg es for nleaf numbers. Th s report recommends, but does not require, that the IEEE 32-b t and 64-b t float ng po nt standards be followed by implementat ons that use flonum represent at on ons, and that implementat ons use ng other represent at on ons should match or exceed the prec s on ach cvable us ng these float ng po nt standards [12].

In part cular, implementat ons that use flonum represent at on ons must follow these rules: A flonum result must be represented w th at least as much prec s on as s used to express any of the nleaf arguments to that operat on. It s des rable (but not required) for potent al ally nleaf operat ons such as \text{sqrt}, when applied to exact arguments, to produce exact answers whenever possible (for example the square root of an exact 4 ought to be an exact 2). If, however, an exact number s operated upon so as to produce an nleaf result (as by \text{sqrt}), and f the result s represented as a flonum, then the most prec s flonum format ava lable must be used; but f the result s represented n some other way then the representat on on must have at least as much prec s on as the most prec s flonum format ava lable.

Although Scheme allows a var ety of wr tten nat on ons for numbers, any part cular implementat on on may support only some of them. For example, an implementat on on n wh ch all numbers are real need not support the rectangular and
6.2.4. Syntax of numerical constants

The syntax of the wr tten representat ons for numbers s described formally n sect on 7.1.1. Note that case s not s gn ficant n numerical cal constants.

A number may be wr tten n b nary, octal, dec mal, or hexadecimal mal by the use of a rad x prefix. The rad x prefixes are #b (b nary), #o (octal), #d (dec mal), and #x (hexadecimal). W th no rad x prefix, a number s assumed to be expressed n dec mal.

A numerical cal constant may be speciﬁ ed to be either exact or inexact by a prefix. The prefixes are #e for exact, and # for inexact. An exactness preﬁ x may appear before or after any rad x preﬁ x that s used. If the wr tten representat on of a number has no exactness preﬁ x, the constant may be e ther exact or inexact. It s inexact f t conta n a dec mal po nt, an exponent, or a “#” character n the place of a d g t, oth erwise it s exact.

In systems w th inexact numbers of vary ng prec ons t may be useful to spec f y the prec on of a constant. For th s purpose, numerical cal constants may be wr tten w th an exponent marker that ndates the des red prec on of the inexact representat on. The letters s, f, d, and 1 spec f y the use of short, s ngle, double, and long prec ons, respec tvely. (When fewer than four internal inexact representat ons ex st, the four s ze speciﬁ cat ons are mapped onto those ava lable. For example, an implementat on w th two internal representat ons may map short and s ngle together and long and double to gether.) In add t on, the exponent marker e speciﬁ es the default prec on for the implementat on. The default prec on on has at least as much prec on as double, but implementat ons may w sh to allow th s de ﬂ at to be set by the user.

3.14159265358979F0
Round to s ngle — 3.14159
0.6L0
Extend to long — .600000000000000

6.2.5. Numerical operat ons

The reader s referred to sect on 1.3.3 for a summary of the nam ng conven ons used to spec f y restr ct ons on the types of arguments to numerical cal rout nes. The examples used n th s sect on assume that any numerical cal constant wr tten us ng an exact notat on on s inde ed represented as an exact number. Some examples also assume that certa n numerical cal constants wr tten us ng an exact notat on on can be represented without loss of accuracy; the exact con stants were chosen so that th s s lely to be true n implementat ons that use ﬂ onums to represent exact numbers.

(number? obj)
(complex? obj)
(real? obj)
(rate? obj)
(integer? obj)
(quotient? obj)

These numerical cal type pred cates can be appl ed to any k nd of argument, include non-numbers. They return #t f the object s of the named type, and otherwise they return #f.

In general, f a type pred cate s true of a number then all h ger type pred cates are also true of that number. Consequently, f a type pred cate s false of a number, then all lower type pred cates are also false of that number.

If z s an inexact numerical constant, then (real? z) s true f and only f (integer? z) s true. If z s an inexact real number, then (integer? z) s true f and only f (= x (round z)).

Note: In many implemen tat ons the rate? procedure w ll be the same as real?, and the complex? procedure w ll be the same as number?, but unusual implemen tat ons may be able to represent some real numbers exactly or may extend the number system to support some k nd of non-complex numbers.

Note: The behav or of these type pred cates on numerical cal numbers s unrel able, s nce any inac curacy may a ffect the result.

These numerical cal pred cates prov de tests for the exactness of a quan ty. For any Scheme number, prec sely one of these pred cates s true.

(exact? z)
(inexact? z)

These procedures return #t f the r arguments are (respec tvely): equal, monoton cally ac reas ng, monoton cally de creas ng, monoton cally non-decreas ng, or monoton cally non ac reas ng.
These predicates are required to be transitive.

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

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 n doubt, consult a numerical analyst.

(abs x)  1 brary procedure

Abs returns the absolute value of x's argument.

(abs -7)  => 7

Note: These predicates are required to be transitive.

(library procedure

(max x1 x2 ...)  1 brary procedure

(min x1 x2 ...)  1 brary procedure

These procedures return the maximum or minimum of their arguments.

Note: If any argument is inexact, then the result will also be inexact (unless the procedure can prove that the inaccuracy is not large enough to affect the result, which is possible only in unusual implementations). If m n or max s used to compare numbers of mixed exactness, and the numerical value of the result cannot be represented as an inexact number without loss of accuracy, then the procedure may report an overflow on an implementation on restrictions.

(library procedure

(+)  1 brary procedure

(*)  1 brary procedure

These procedures return the sum or product of the r arguments.

Note: If any argument is inexact, then the result will also be inexact (unless the procedure can prove that the inaccuracy is not large enough to affect the result, which is possible only in unusual implementations). If m n or max s used to compare numbers of mixed exactness, and the numerical value of the result cannot be represented as an inexact number without loss of accuracy, then the procedure may report an overflow on an implementation on restrictions.

(library procedure

(- z1 z2)  1 brary procedure

(+)  1 brary procedure

These procedures return the difference or quotient of the arguments, associative to the left. With one argument, however, they return the additive inverse of the argument.

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

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.

(abs x)  1 brary procedure

Abs returns the absolute value of x's argument.

(abs -7)  => 7

Note: These predicates are required to be transitive.

(library procedure

(max x1 x2 ...)  1 brary procedure

(min x1 x2 ...)  1 brary procedure

These procedures return the maximum or minimum of their arguments.

Note: If any argument is inexact, then the result will also be inexact (unless the procedure can prove that the inaccuracy is not large enough to affect the result, which is possible only in unusual implementations). If m n or max s used to compare numbers of mixed exactness, and the numerical value of the result cannot be represented as an inexact number without loss of accuracy, then the procedure may report an overflow on an implementation on restrictions.

(library procedure

(+)  1 brary procedure

(*)  1 brary procedure

These procedures return the sum or product of the r arguments.

Note: If any argument is inexact, then the result will also be inexact (unless the procedure can prove that the inaccuracy is not large enough to affect the result, which is possible only in unusual implementations). If m n or max s used to compare numbers of mixed exactness, and the numerical value of the result cannot be represented as an inexact number without loss of accuracy, then the procedure may report an overflow on an implementation on restrictions.

(library procedure

(- z1 z2)  1 brary procedure

(+)  1 brary procedure

These procedures return the difference or quotient of the arguments, associative to the left. With one argument, however, they return the additive inverse of the argument.
These procedures return the greatest common divisor or least common multiple of the r arguments. The result is always non-negative.

\[
\begin{align*}
(gcd \ n_1 \ldots) & \quad \text{1 brary procedure} \\
(lcm \ n_1 \ldots) & \quad \text{1 brary procedure}
\end{align*}
\]

\[
\text{result should be passed to the Round procedure.}
\]

\[
\text{Note: If the argument to one of these procedures is inexact, rounding to even when halfway between two integers is always non-negative.}
\]

\[
\begin{align*}
(gcd \ 32 \ldots -36) & \quad \Rightarrow 4 \\
(gcd) & \quad \Rightarrow 0 \\
(lcm \ 32 \ldots -36) & \quad \Rightarrow 288 \\
(lcm \ 32.0 \ldots -36) & \quad \Rightarrow 288.0 \ ; \ \text{nexact} \\
(lcm) & \quad \Rightarrow 1
\end{align*}
\]

\[
\begin{align*}
\text{(numerator } q) & \quad \text{procedure} \\
\text{(denom nator } q) & \quad \text{procedure}
\end{align*}
\]

\[
\text{These procedures return the numerator or denominator of the } r \text{ argument; the result is computed as if the argument was represented as a fraction in lowest terms. The denominator of 0 is always positive. The denominator of 0 is defined as 1.}
\]

\[
\begin{align*}
\text{(numerator } (/ \ 6 \ 4)) & \quad \Rightarrow 3 \\
\text{(denom nator } (/ \ 6 \ 4)) & \quad \Rightarrow 2 \\
\text{(denom nator} ) & \quad \Rightarrow 2.0 \\
\end{align*}
\]

\[
\begin{align*}
\text{(floor } x) & \quad \text{procedure} \\
\text{(ce l ng } x) & \quad \text{procedure} \\
\text{(truncate } x) & \quad \text{procedure} \\
\text{(round } x) & \quad \text{procedure}
\end{align*}
\]

\[
\text{These procedures return integers. Floor returns the largest integer not larger than } x. \text{ Ceiling returns the smallest integer not smaller than } x. \text{ Truncate returns the integer closest to } x \text{ whose absolute value is not larger than the absolute value of } x. \text{ Round returns the closest integer to } x, \text{ roundng to evn when } x \text{ s halfway between two integers.}
\]

\[
\text{Rat onal : Round rounds to even for cons stency w th the defaut roundng mode spec fied by the IE EE float ng po nt standard.}
\]

\[
\text{Note: If the argument to one of these procedures is nexact, then the result will also be nexact. If an exact value is needed, the result should be passed to the nexact->exact procedure.}
\]

\[
\begin{align*}
\text{(floor } -4.3) & \quad \Rightarrow -5.0 \\
\text{(ce l ng } -4.3) & \quad \Rightarrow -4.0 \\
\text{(truncate } -4.3) & \quad \Rightarrow -4.0 \\
\text{(round } -4.3) & \quad \Rightarrow -4.0 \\
\text{(floor } 3.5) & \quad \Rightarrow 3.0 \\
\text{(ce l ng } 3.5) & \quad \Rightarrow 4.0 \\
\text{(truncate } 3.5) & \quad \Rightarrow 3.0 \\
\text{(round } 3.5) & \quad \Rightarrow 4.0 \ ; \ \text{nexact}
\end{align*}
\]

\[
\begin{align*}
\text{(round } 7/2) & \quad \Rightarrow 4 \ ; \ \text{exact} \\
\text{(round } 7) & \quad \Rightarrow 7
\end{align*}
\]

\[
\text{(rat onal } x y) & \quad \text{1 brary procedure}
\]

\[
\text{Rat onal } x y \text{ returns the s mplest } x \text{ y onal number defined from } x \text{ by no more than } y. \text{ A rational number } r_1 \text{ s mpler than } r_2 \text{ s mpler than } r_2 \text{ s mpler than } r_2 \text{ s mpler than } r_2 \text{ s mpler than } r_2 \text{ s mpler than } r_2 \text{ s mpler than every other rational number } n \text{ that } n \text{ interval (the s mpler 2/5 1es between 2/7 and 3/5). Note that 0 = 0/1 s the s mplest rational of all.}
\]

\[
\begin{align*}
\text{(rat onal } z) & \quad \text{procedure} \\
\text{(log } z) & \quad \text{procedure} \\
\text{as } n \text{ z} & \quad \text{procedure} \\
\text{cos } z & \quad \text{procedure} \\
\text{tan } z & \quad \text{procedure} \\
\text{as } n \text{ z} & \quad \text{procedure} \\
\text{acos } z & \quad \text{procedure} \\
\text{atan } z & \quad \text{procedure} \\
\text{atan y x} & \quad \text{procedure}
\end{align*}
\]

\[
\text{These procedures are part of every implementat on that supports general real numbers; they compute the usual transcendental funct ons. Log computes the natural logarithm of } z (\text{not the base ten logarithm). As n, acos, and atan compute arcs ne (s n^{-1}), arccos ne (s n^{-1}), and arctangent (tan^{-1}), respect vely. The two-argument vari ant of atan computes (angle (make-rectangular x y)) (see below), even n implementat ons that don’t support general complex numbers.}
\]

\[
\text{In general, the mathemat cal funct ons log, arcs ne, arccos ne, and arctangent are mply de ned. The value of log} x \text{ is de ned to be the one whose mag nary part} l e s n \text{ the range from} -\pi (\text{exclus ve}) \text{ to} \pi (\text{clus ve}). \text{Log 0 is unde ned. W th log de ned th s way, the values of } s n^{-1}, \text{ cos}^{-1} z, \text{ and tan}^{-1} z \text{ are accord ng to the follow ng formul e:}
\]

\[
\begin{align*}
\text{s n^{-1} z} & \equiv - \log( z + \sqrt{1 - z^2}) \\
\text{cos}^{-1} z & \equiv \pi / 2 - s n^{-1} z \\
\text{tan}^{-1} z & \equiv (\log(1 + z) - \log(1 - z)) / (2)
\end{align*}
\]

\[
\text{The above spec fcat on follows [27], wh ch n turn c tes [19]; refer to these sources for more deta led scuss on of branch cuts, boundary cond t ons, and implementat on of these funct ons. When t s poss ble these procedures produce a real result from a real argument.}
then an implementation-dependent range. See section
spondence between exact and inexact integers throughout
These procedures implement the natural one-to-one corre-
plementation restriction may be reported.

\[ z_1z_2 = e^{z_2 \log z_1} \]

0^z s 1 f z = 0 and 0 otherw se.

\[
\begin{align*}
\text{(make-rectangular } x_1 x_2) & \quad \text{procedure} \\
\text{(make-polar } x_3 x_4) & \quad \text{procedure} \\
\text{(real-part } z) & \quad \text{procedure} \\
\text{(mag-part } z) & \quad \text{procedure} \\
\text{(magn tude } z) & \quad \text{procedure} \\
\text{(angle } z) & \quad \text{procedure}
\end{align*}
\]

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

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

Then

\[
\begin{align*}
\text{(make-rectangular } x_1 x_2) & \quad \Rightarrow z \\
\text{(make-polar } x_3 x_4) & \quad \Rightarrow z \\
\text{(real-part } z) & \quad \Rightarrow x_1 \\
\text{(mag-part } z) & \quad \Rightarrow x_2 \\
\text{(magn tude } z) & \quad \Rightarrow |x_3| \\
\text{(angle } z) & \quad \Rightarrow x_{angle}
\end{align*}
\]

where \( -\pi < x_{angle} \leq \pi \) w th \( x_{angle} = x_4 + 2\pi n \) for some
nteger \( n \).

\[ \frac{\pi}{2} < x_{angle} \leq \frac{3\pi}{2} \]

**Rat onale:** \text{Mag n tude} is the same as \text{abs} for a real argument,
but \text{abs} must be present n all implementat ons, whereas
\text{magn tude} need only be present n implementat ons that support
general complex numbers.

\[
\begin{align*}
\text{(exact-> nexact } z) & \quad \text{procedure} \\
\text{(n-exact->exact } z) & \quad \text{procedure}
\end{align*}
\]

\[ \text{Exact-> nexact returns an nexact representat on of } z. \]

\[ \text{The value returned } s \text{ the nexact number that } s \text{ numer-
ally closest to the argument. If an exact argument has no
reasonably close nexact equ valent, then a v olat on of an
implementat on on restr ct on may be reported.} \]

\[ \text{Inexact->exact returns an exact representat on of } z. \]

\[ \text{The value returned } s \text{ the exact number that } s \text{ numer cally clos-
est to the argument. If an nexact argument has no
reasonably close exact equ valent, then a v olat on of an
implementat on on restr ct on may be reported.} \]

These procedures implement the natural one-to-one corre-
spondence between exact and nexact ntegers throughout
an implementat on-depen dent range. See sect on 6.2.3.

### 6.2.6. Numer cal input and output

\[
\begin{align*}
\text{(number->str } ng x) & \quad \text{procedure} \\
\text{(str } ng->number } x & \quad \text{procedure} \\
\text{(rad } x rad x) & \quad \text{procedure}
\end{align*}
\]

\[ Rad x \text{ must be an exact nteger, } e \text{ ther } 2, 8, 10, \text{ or } 16. \]

\[ \text{If om tted, } rad x \text{ defaults to } 10. \]

\[ \text{The procedure } \text{number->str } ng \text{ takes a number and a rad x and returns as a str } ng
\text{ an external representat on of the g ven number } n \text{ the g ven rad x such that} \]

\[
\begin{align*}
\text{(let } ((\text{number } number) \\
\text{(rad } x rad x)) \\
\text{(eqv? number } (\text{str } ng->\text{number } (\text{number->str } ng number) \\
\text{rad x)))} \]
\]

\[ s \text{ true. It } s \text{ an error } f \text{ no poss ble result makes } th s \text{ ex-
press on true.} \]

\[ \text{If } z \text{ s nexact, the rad x } s 10, \text{ and the above express on can be sat sfi ed by a result that conta ns a dec mal po nt, then the result conta ns a dec mal po nt and } s \text{ expressed us ng the m n mum number of d g ts (exclus ve of exponent}
\text{ and tra l ng zeroes) needed to make the above express on on}
true [3, 5]; otherw se the format of the result } s \text{ unspec fed.} \]

\[ \text{The result returned by } \text{number->str ng} \text{ never conta ns an expl c t rad x prefix.} \]

\[ \text{Note: The error case can occur only when } z \text{ s not a complex number or } s \text{ a complex number w th a non-rat onal real or mag nary part.} \]

\[ \text{Rat onale: If } z \text{ s an nexact number represented us ng flonums, and the rad x } s 10, \text{ then the above express on on } s \text{ normally sat sfi ed by a result conta ns a dec mal po nt. The unspec fed case}
\text{ allows for } nfn t es, N\text{Ns, and non-flonum representat on ons.} \]

\[
\begin{align*}
\text{(str } ng->\text{number } str ng) & \quad \text{procedure} \\
\text{(str } ng->\text{number } str ng rad x) & \quad \text{procedure}
\end{align*}
\]

\[ \text{Returns a number of the max maly prec se representat on on expressed by the g ven str ng. } Rad x \text{ must be an exact nteger, } e \text{ ther } 2, 8, 10, \text{ or } 16. \]

\[ \text{If suppl ed, } rad x \text{ s a default rad x that may be over d den by an expl c t rad x prefix } n \text{ str ng (e.g. "#1077").} \]

\[ \text{If rad x } s \text{ not suppl ed, then the default rad x } s 10. \]

\[ \text{If str ng } s \text{ not a syntact cally val d notat on on a number, then } \text{str } ng->\text{number} \text{ returns } \text{#f}. \]

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

\[ \text{Note: The dom a } n \text{ of } \text{str } ng->\text{number} \text{ may be restr cted by}
\text{implementat on ons } n \text{ the follow ng ways. } \text{Str } ng->\text{number} \text{ s per-
mitt } s \text{ return } \text{#f} \text{ whenever } str ng \text{ conta ns an expl c t rad x}
\text{prefix. If all numbers supported by an implementat on are real,} \]
then \texttt{string->number} is permitted to return \#f whenever \texttt{string} uses the polar or rectangular notations for complex numbers. If all numbers are \texttt{numbers}, then \texttt{string->number} may return \#f whenever the fractional notation is used. If all numbers are \texttt{exact}, then \texttt{string->number} may return \#f whenever an exponent marker or explicit exactness prefix is used, or if a \# appears in place of a digit. If all inexact numbers are \texttt{integers}, then \texttt{string->number} may return \#f whenever a decimal point is used.

6.3. Other data types

This section describes operations on some of Scheme’s non-numeric data types: \texttt{booleans}, \texttt{pairs}, \texttt{lists}, \texttt{symbols}, \texttt{characters}, \texttt{strings} and \texttt{vectors}.

6.3.1. 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, \texttt{numbers}, \texttt{pairs}, the empty list, symbols, \texttt{strings}, \texttt{vectors}, and procedures, count as true.

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

Boolean constants evaluate to themselves, so they do not need to be quoted in programs.

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

6.3.2. Pairs and lists

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

\begin{itemize}
  \item The empty list is in \(X\).
  \item If \texttt{list} is in \(X\), then any pair whose \texttt{car} field contains \texttt{list} is also in \(X\).
\end{itemize}

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 \texttt{car} is the first element and whose \texttt{cdr} is the empty list. The length of a list is the number of elements, which is 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.

\textit{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,

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

\begin{itemize}
  \item The empty list is \(\texttt{nil}\).
  \item If \texttt{lst} is \(\texttt{nil}\), then any \texttt{pair} whose \texttt{car} field contains \texttt{lst} is also \(\texttt{nil}\).
\end{itemize}

The objects in the car fields of success lists 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 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.

\textit{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,
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:

\[
(a \ b \ c \ . \ d)
\]
s equ valent to

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

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

\[
\begin{align*}
\text{(define } x \text{ (list } 'a \ 'b \ 'c \text{))} \\
\text{(define } y \text{ x}) \\
\text{y} &= \Rightarrow (a \ b \ c) \\
\text{(pair? y)} &= \Rightarrow \texttt{#t} \\
\text{(set-cdr! x 4)} &= \Rightarrow \text{ unspecified} \\
\text{x} &= \Rightarrow (a \ . \ 4) \\
\text{(eqv? x y)} &= \Rightarrow \texttt{#t} \\
\text{(pair? y)} &= \Rightarrow \texttt{#f} \\
\text{(set-cdr! x x)} &= \Rightarrow \text{ unspecified} \\
\text{(pair? x)} &= \Rightarrow \texttt{#f}
\end{align*}
\]

Within literal expressions and representations of objects read by the \texttt{read} procedure, the forms \texttt{'⟨datum1⟩}, \texttt{'⟨datum1⟩,⟨datum2⟩}, \texttt{'⟨datum1⟩}, and \texttt{'⟨datum1⟩denote two-element lists whose first elements are the symbols \texttt{quote}, \texttt{quasiquote}, \texttt{unquote}, and \texttt{unquote-splicing}, respectively. This convention is supported so that arbitrary Scheme programs may be represented as lists. That is, according to Scheme’s grammar, every \texttt{⟨expression⟩} is also a \texttt{⟨datum⟩} (see section 7.1.2). Among other things, this permits the use of the \texttt{read} procedure to parse Scheme programs. See section 3.3.

\[
\begin{align*}
\text{(pair? } '(a . b)\text{)} &= \Rightarrow \texttt{#t} \\
\text{(pair? } '(a \ b \ c)\text{)} &= \Rightarrow \texttt{#t} \\
\text{(pair? } '(a)\text{)} &= \Rightarrow \texttt{#f} \\
\text{(pair? } '#(a b)\text{)} &= \Rightarrow \texttt{#f}
\end{align*}
\]

\[
\begin{align*}
\text{(car } pa r\text{)} & \quad \text{procedure} \\
\text{Returns the contents of the car field of } pa r. \text{ Note that } t \text{ is an error to take the car of the empty list.}
\end{align*}
\]

\[
\begin{align*}
\text{(car } '(a \ b \ c)\text{)} &= \Rightarrow a \\
\text{(car } '((a) \ b \ c \ d)\text{)} &= \Rightarrow (a) \\
\text{(car } '((1 \ . \ 2))\text{)} &= \Rightarrow 1 \\
\text{(car } '()\text{)} &= \Rightarrow \text{ error}
\end{align*}
\]

\[
\begin{align*}
\text{(cdr } pa r\text{)} & \quad \text{procedure} \\
\text{Returns the contents of the cdr field of } pa r. \text{ Note that } t \text{ is an error to take the cdr of the empty list.}
\end{align*}
\]

\[
\begin{align*}
\text{(cdr } '((a) \ b \ c \ d)\text{)} &= \Rightarrow (b \ c \ d) \\
\text{(cdr } '(1 \ . \ 2)\text{)} &= \Rightarrow 2 \\
\text{(cdr } '()\text{)} &= \Rightarrow \text{ error}
\end{align*}
\]

\[
\begin{align*}
\text{(set-car! } pa r \text{ obj) } & \quad \text{procedure} \\
\text{Stores } obj \text{ in the car field of } pa r. \text{ The value returned by } \text{set-car!} s \text{ unspecified.}
\end{align*}
\]

\[
\begin{align*}
\text{(def ne (f) (list not-a-constant-list))} \\
\text{(def ne (g) 'constant-list))} \\
\text{(set-car! (f) 3)} &= \Rightarrow \texttt{unspec fied} \\
\text{(set-car! (g) 3)} &= \Rightarrow \texttt{error}
\end{align*}
\]

\[
\begin{align*}
\text{(set-cdr! } pa r \text{ obj) } & \quad \text{procedure} \\
\text{Stores } obj \text{ in the cdr field of } pa r. \text{ The value returned by } \text{set-cdr!} s \text{ unspecified.}
\end{align*}
\]

\[
\begin{align*}
\text{(caar } pa r\text{)} &= 1 \text{ brary procedure} \\
\text{(cadr } pa r\text{)} &= 1 \text{ brary procedure} \\
\text{(cdddar } pa r\text{)} &= 1 \text{ brary procedure} \\
\text{(cddddr } pa r\text{)} &= 1 \text{ brary procedure}
\end{align*}
\]

These procedures are composites of \texttt{car} and \texttt{cdr}, where for example \texttt{cddadr} could be defined by

\[
\begin{align*}
\text{(define cddadr (lambda (x) (car (cdr (cdr x))))).}
\end{align*}
\]

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

\[
\begin{align*}
\text{(null? } obj\text{)} &= 1 \text{ brary procedure} \\
\text{Returns } \texttt{#t} \text{ if } obj \text{ is the empty list, otherwse returns } \texttt{#f}.
\end{align*}
\]

\[
\begin{align*}
\text{(l st? } obj\text{)} &= 1 \text{ brary procedure} \\
\text{Returns } \texttt{#t} \text{ if } obj \text{ is a list, otherwse returns } \texttt{#f}. \text{ By definition, all lists have finite length and are terminated by the empty list.}
\end{align*}
\]
(1 st? ' (a b c)) ⇒ #t
(1 st? '()) ⇒ #t
(1 st? '(a . b)) ⇒ #f
(let ((x (1 st 'a)))
  (set-cdr! x x)
(1 st? x)) ⇒ #f

(l st? obj ...) 1 brary procedure
Returns a newly allocated l st of ts arguments.

(1 st 'a (+ 3 4) 'c) ⇒ (a 7 c)
(1 st) ⇒ ()

(length l st) 1 brary procedure
Returns the length of l st.

(length '(a b c)) ⇒ 3
(length '(a (b) (c d e))) ⇒ 3
(length '()) ⇒ 0

(append l st ...) 1 brary procedure
Returns a l st cons st ng of the elements of the first l st followed by the elements of the other l sts.

(append '(x) '(y)) ⇒ (x y)
(append '(a) '(b c d)) ⇒ (a b c d)
(append '(a (b)) '(c ())) ⇒ (a (b) (c))

The result ng l st s always newly allocated, except that t shares structure w th the last l st argument. The last argument may actually be any object; an improper l st results f the last argument s not a proper l st.

(append '(a b) '(c . d)) ⇒ (a b c . d)
(append '() 'a) ⇒ a

(reverse l st) 1 brary procedure
Returns a newly allocated l st cons st ng of the elements of l st n reverse order.

(reverse '(a b c)) ⇒ (c b a)
(reverse '(a (b c) d (e (f))))
⇒ ((e (f)) d (b c) a)

(l st-ta 1 l st k) 1 brary procedure
Returns the subl st of l st obta ned by om tt ng the first k elements. It s an error f l st has fewer than k elements. L st-ta 1 could be defined by

(def ne l st-ta l
  (lambda (x k)
    (f (zero? k)
      x
      (l st-ta l (cdr x) (- k 1)))))

(1 st-ref l st k) 1 brary procedure
Returns the kth element of l st. (Th s s the same as the car of (1 st-ta 1 l st k).) It s an error f l st has fewer than k elements.

(1 st-ref '(a b c d) 2) ⇒ c
(1 st-ref '(a b c d)
  (n e x a t -> e x a t (r o u n d 1.8)))
⇒ c

(memq obj l st) 1 brary procedure
(memv obj l st) 1 brary procedure
(member obj l st) 1 brary procedure

These procedures return the first subl st of l st whose car s obj, where the subl sts of l st are the non-empty l sts returned by (1 st-ta 1 l st k) for k less than the length of l st. If obj does not occur n l st, then #f (not the empty l st) s returned. Memq uses eq? to compare obj w th the elements of l st, wh le 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 (1 st 'a) '(b (a) c)) ⇒ #f
(member (1 st 'a) '
  (b (a) c)) ⇒ (a (c)
(memq 101 '((100 101 102)) ⇒ unspec fied
(memv 101 '((100 101 102)) ⇒ (101 102)

(assq obj al st) 1 brary procedure
(assv obj al st) 1 brary procedure
(assoc obj al st) 1 brary procedure

Al st (for “assoc at on l st”) must be a l st of pa rs. These procedures find the first pa r n al st whose car field s obj, and returns that pa r. If no pa r n al st has obj as ts car, then #f (not the empty l st) s returned. Assq uses eq? to compare obj w th the car fields of the pa rs n al st, wh le assv uses eqv? and assoc uses equal?.

(assq obj al st) ⇒ #f
(assoc obj al st) ⇒ #f

(assq obj al st) ⇒ #f
(assoc obj al st) ⇒ #f

(assq obj al st) ⇒ #f
(assoc obj al st) ⇒ #f

(memq (1 st 'a) '(b (a) c)) ⇒ #f
(member (1 st 'a) '
  (b (a) c)) ⇒ (a (c)
(memq 101 '((100 101 102)) ⇒ unspec fied
(memv 101 '((100 101 102)) ⇒ (101 102)

(assq obj al st) ⇒ #f
(assoc obj al st) ⇒ #f

(assq obj al st) ⇒ #f
(assoc obj al st) ⇒ #f

(assq obj al st) ⇒ #f
(assoc obj al st) ⇒ #f

Rat onale: Although they are ord nar ly used as pred cates, memq, memv, member, assq, assv, and assoc do not have quest on marks n the r names because they return useful values rather than just #t or #f.
6.3.3. Symbols

Symbols are objects whose usefulness rests on the fact that two symbols are dent cal (n the sense of eqv?) f and only f the r names are spelled the same way. Th s s exactly the property needed to represent dent fiers n programs, and so most implementat ons of Scheme use them internally for that purpose. Symbols are useful for many other appli- cations, for instance, they may be used the way enumerated values are used n Pascal.

The rules for wr t ng a symbol are exactly the same as the rules for wr t ng an dent fier; see sect ons 2.1 and 7.1.1.

It s guaranteed that any symbol that has been returned as part of a l t eral express on, or read us ng the read proced ure, and subsequently wr t ten out us ng the wr te proced ure, w ll read back n as the dent cal symbol (n the sense of eqv?). The str ng->symbol proced ure, however, can create symbols for wh ch th s wr te/read nvar ance may not hold because the r names conta n spec al characters or letters n the non-standard case.

Note: Some implementat ons of Scheme have a feature known as “slash ficat on” n order to guarantee wr t ng/nvar ance for all symbols, but h stor cally the most important use of th s feature has been to compensate for the lack of a str ng data type.

Some implementat ons also have “un nterned symbols”, wh ch defeat wr t ng/nvar ance even n implementat ons w th slash ficat on, and also generate except ons to the rule that two symbols are the same f and only f the r names are spelled the same.

(symbol? obj) procedure
Returns #t f obj s a symbol, otherw se returns #f.

(symbol? 'foo) == #t
(symbol? (car '(a b))) == #t
(symbol? "bar") == #f
(symbol? 'n l) == #t
(symbol? '()) == #f
(symbol? #f) == #f

(symbol->str ng symbol) procedure
Returns the name of symbol as a str ng. If the symbol was part of an object returned as the value of a l t eral express on (sect on 4.1.2) or by a call to the read proced ure, and ts name conta ns alphabet c characters, then the str ng returned w ll conta n characters n the implementat on’s preferred standard case—some implementat ons w ll prefer upper case, others lower case. If the symbol was returned by str ng->symbol, the case of characters n the str ng returned w ll be the same as the case n the str ng that was passed to str ng->symbol. It s an error to apply mutat on proced ures lke str ng-set! to str ngs returned by th s proced ure.

The follow ng examples assume that the implementat on’s standard case s lower case:

(symbol->str ng 'fly ng-f sh) == "fly ng-f sh"
(symbol->str ng 'Mart n) == "mart n"
(symbol->str ng (str ng->symbol "Malv na")) == "Malv na"

(str ng->symbol str ng) procedure
Returns the symbol whose name s str ng. Th s proced ure can create symbols w th names conta n spec al characters or letters n the non-standard case, but t s usually a bad dea to create such symbols because n some implementat ons of Scheme they cannot be read as themselves. See symbol->str ng.

The follow ng examples assume that the implementat on’s standard case s lower case:

(eq? 'mISSISSIpp 'm ss ss pp) == #t
(str ng->symbol "mISSISSIpp ") == the symbol w th name "mISSISSIpp ",
(eq? 'b tBl (str ng->symbol "b tBl")) == #f
(eq? 'JollyWog (str ng->symbol (symbol->str ng 'JollyWog))) == #t
(str ng=? "K. Harper, M.D." (symbol->str ng (str ng->symbol "K. Harper, M.D."))) == #t

6.3.4. Characters

Characters are objects that represent printed characters such as letters and d g ts. Characters are wr tten us ng the notat on #\(character or #\(character name . For example:

#\a ; lower case letter
#\A ; upper case letter
#\( ; left parentheses s
#\ ; the space character
#\space ; the preferred way to wr te a space
#\newl ne ; the newl ne character

Case ss gn ficat n #\(character , but not n #\(character name . If #\(character n #\(character s alphabet c, then the character follow ng #\(character must be a del m ter character such as a space or parentheses s. Th s rule resolves the ambigu us case where, for example, the sequence of
Characters "\space" could be taken to be ether a represen-
tat on of the space character or a representat on of the character "\s" followed by a representat on of the symbol "pace."

Characters written in the \# notat on are self-evaluat ng. That \( s \), they do not have to be quoted \( n \) programs.

Some of the procedures that operate on characters ignore the difference between upper case and lower case. The pro-
cedures that ignore case have "-c " (for "case nsens t ve") embedded \( n \) the r names.

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

Returns \#t f obj \( s \) a character, otherw se returns \#f.

\[
\text{(char=? char1 char2)}
\]

procedure

\[
\text{(char<? char1 char2)}
\]

procedure

\[
\text{(char>? char1 char2)}
\]

procedure

\[
\text{(char<=? char1 char2)}
\]

procedure

\[
\text{(char>=? char1 char2)}
\]

procedure

\[
\text{(char-ci=? char1 char2)}
\]

procedure

\[
\text{(char-ci>=? char1 char2)}
\]

procedure

\[
\text{(char-ci>? char1 char2)}
\]

procedure

\[
\text{(char-ci<? char1 char2)}
\]

procedure

These procedures impose a total order ng on the set of characters. It s guaranteed that under th s order ng:

- The upper case characters are \( n \) order. For example, \( \text{(char>? \#\A \#\B)} \) returns \#t.
- The lower case characters are \( n \) order. For example, \( \text{(char<? \#\a \#\b)} \) returns \#t.
- The d g ts are \( n \) order. For example, \( \text{(char<? \#\0 \#\9)} \) returns \#t.
- E ther all the d g ts precede all the upper case letters, or v ce versa.
- E ther all the d g ts precede all the lower case letters, or v ce versa.

Some implementat ons may general ze these procedures to take more than two arguments, as w th the correspond ng numer cal pred cates.

\[
\text{(char-c =? char1 char2)}
\]

1 brary procedure

\[
\text{(char-c <? char1 char2)}
\]

1 brary procedure

\[
\text{(char-c >? char1 char2)}
\]

1 brary procedure

\[
\text{(char-c <=? char1 char2)}
\]

1 brary procedure

\[
\text{(char-c =>? char1 char2)}
\]

1 brary procedure

These procedures are s m lar to char=? et cetera, but they treat upper case and lower case letters as the same. For example, \( \text{(char-c =? \#\A \#\a)} \) returns \#t. Some implementat ons may general ze these procedures to take more than two arguments, as w th the correspond ng numer cal pred cates.

\[
\text{(char-alphabet c? char)}
\]

1 brary procedure

\[
\text{(char-numer c? char)}
\]

1 brary procedure

\[
\text{(char-wh tespac e? char)}
\]

1 brary procedure

\[
\text{(char-upper-case? letter)}
\]

1 brary procedure

\[
\text{(char-lower-case? letter)}
\]

1 brary procedure

These procedures return \#t f the r arguments are alpha-
bet c, numer c, wh tespac e, upper case, or lower case char-
acters, respect vely, otherw se they return \#f. The follow-
ng remarks, wh ch are spec fic to the ASCII character set, are ntended only as a gu de: The alphabet c characters are the 52 upper and lower case letters. The numer c characters are the ten dec mal d g ts. The wh tespac e characters are space, tab, l ne feed, form feed, and carr age return.

\[
\text{(char-> nte ger char)}
\]

procedure

\[
\text{(nte ger->char n)}
\]

procedure

\[
\text{Given a character, char-> nte ger returns an exact nte-
ger representat on of the character. Given an exact nte-
ger that s the mage of a character under char-> nte ger,
nte ger->char returns that character. These procedures
implement order-preserv ng somorph sm s between the set
of characters under the char<=? order ng and some subset
of the ntegers under the <= order ng. That \( s, f \)
\[
\text{(char<=? a b)} \iff \#t \text{ and } (<= (char-> nte ger a)) \iff \#t
\]

\[
\text{and } x \text{ and } y \text{ are } n \text{ the doma n of nte ger->char, then}
\]

\[
\text{(char<=? ( char-> nte ger a)) \iff \#t}
\]

\[
\text{and (char-> nte ger (char-> nte ger b))) \iff \#t}
\]

\[
\text{(char<=? ( nte ger->char x)} \text{ ( nte ger->char y)) \iff \#t}
\]

\[
\text{(char-upcase char)}
\]

1 brary procedure

\[
\text{(char-downcase char)}
\]

1 brary procedure

These procedures return a character char\_arg such that
(\text{char-c =? char char}_2). In add t on, f char \( s \) alphabet c, then the result of char-upcase \( s \) upper case and the result of char-downcase \( s \) lower case.

6.3.5. Strngs

Strngs are sequences of characters. Strngs are wrtten as sequences of characters enclosed w th n doublequotes ("."). A doublequote can be wrtten ns de a str ng only by escap ng t w th a backslash (\), as n

"The word "$recurs on" has many mean ng.s."

A backslash can be wrtten ns de a str ng only by escap ng t w th another backslash. Scheme does not spec fy the effect of a backslash w th n a str ng that \( s \) not followed by a doublequote or backslash.
A string constant may continue from one line to the next, but the exact contents of such a string are unspecified.

The length of a string is the number of characters that constitute it. This number is an exact, non-negative integer less than or equal to 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 str begin at index start and end at index end,” t s understood that the index start s inclusive and the index end s exclusive. Thus if start and end are the same, nul substr ng s referred to, and if start s zero and end s the length of str ng, then the entire str ng s referred to.

Some of the procedures that operate on str ng s ignore the difference between upper and lower case. The versions that treat upper and lower case as distinct characters.

(string? obj) procedure
Returns #t if obj s a string, otherw se returns #f.

(make-str ng k) procedure
(make-str ng k char) procedure
Make-str ng returns a newly allocated str ng of length k. If char s g ven, then all elements of the str ng are n-t al zed to char, otherw se the contents of the str ng re un Spec fied.

(str ng char ...) 1 brary procedure
Returns a newly allocated str ng composed of the arguments.

(str ng-length str ng) procedure
Returns the number of characters n the g ven str ng.

(str ng-ref str ng k) procedure
k must be a valid n dex of str ng. Str ng-ref returns character k of str ng us ng zero-or-g n ndex ng.

(str ng-set! str ng k char) procedure
k must be a valid n dex of str ng. Str ng-set! stores char n element k of str ng and returns an unSpec fied value.

(define (make-str ng 3 #\*))
(define (g) "****")
((str ng-set! (f) 0 #\?) => unspec fied)
((str ng-set! (g) 0 #\?)) => error
((str ng-set! (symbol->str ng 'mutable) 0 #\?)) => error

Returns #t if the two str ng s are the same length and co ntain the same characters n the same post on s, otherw se returns #f. Str ng-c =? treats upper and lower case letters as though they were the same character, but str ng=? treats upper and lower case as d st nct characters.

(string-ci=? str ng1 str ng2) 1 brary procedure
(string-ci>? str ng1 str ng2) 1 brary procedure
(string-ci=? str ng1 str ng2) 1 brary procedure
(string-ci>c? str ng1 str ng2) 1 brary procedure
(string-ci>=? str ng1 str ng2) 1 brary procedure
(string-ci>=? str ng1 str ng2) 1 brary procedure
(string-ci>c=? str ng1 str ng2) 1 brary procedure
(string-ci>=? str ng1 str ng2) 1 brary procedure

These procedures are the lexicog rap c ex t nons to str ng s of the correspond ng order ng s on characters. For example, str ng-c=? s the lexicog rap c order ng on ng s Str ng nduced by the order ng char=c? on characters. If two str ng s dffer n length but are the same up to the length of the shorter str ng, the shorter str ng s cons dered to be lex cographicaly less than the longer str ng.

Implementat ons may general ze these and the str ng=? and str ng-c =? procedures to take more than two arguments, as w th the correspond ng numer cal pred cates.

(substr str ng start end) 1 brary procedure
Str ng must be a str ng, and start and end must be exact ntegers sat sfy ng

\[0 \leq start \leq end \leq (str ng-length str ng)\]

Substr ng returns a newly allocated str ng formed from the characters of str begin at index start and end at index end (inclusive).

((str ng-append str ng ...) 1 brary procedure
Returns a newly allocated str ng whose characters form the concatenation of the g ven str ng s.

((str ng->l str ng) 1 brary procedure
(l st->str ng l st) 1 brary procedure
Str ng->l st returns a newly allocated list of the characters that make up the g ven str ng. L st->str ng returns a newly allocated str ng formed from the characters n the list st, wh ch must be a list of characters. Str ng->l st and l st->str ng are inverses so far as equal? s concerned.

((str ng-copy str ng)) 1 brary procedure
Returns a newly allocated copy of the g ven str ng.
6.3.6. 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. Th's number n 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 n a vector is indexed by zero, and the last element is indexed by one less than the length of the vector.

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

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

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

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

6.4. 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)
```

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

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

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

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

Stores fill in every element of vector. The value returned by this procedure is unspecified.

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

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

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

```
(vector-ref '#((1 2 3 5 8 13 21) (let ((i (round (* 2 (acos -1)))) (f (nexact? ) (nexact->exact ))))) 13)
```

```
(vector-set! vector k obj)
```

Stores obj in element k of vector. The value returned by this procedure is unspecified.

```
(vector-set! '#((0 1 2) 1 "doe")
```

```
(= (vector-ref '#((1 1 2 3 5 8 13 21) (let ((i (round (* 2 (acos -1)))) (f (nexact? ) (nexact->exact ))))) 5) 8
```

```
(vector-set! '#((1 2 3 5 8 13 21) (let ((i (round (* 2 (acos -1)))) (f (nexact? ) (nexact->exact )))))
```

```
(vector->list vector)
```

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.

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

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

```
(vector-f ll! vector fill)
```

Stores fill in every element of vector. The value returned by this procedure is unspecified.

```
(vector-f ll!)
```

6.4. Control features

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

```
(procedure? pred cate)
```

Returns #t if obj is a procedure, otherwise returns #f.
(apply proc arg1 ... args)                     procedure
Proc must be a procedure and args must be a list. Calls proc with the elements of the list (append (list arg1 ... args) as the actual arguments.

(apply + (list 3 4)) ⇒ 7
(def ne compose
  (lambda (f g)
    (lambda args
      (f (apply g args)))))
((compose sqrt *) 12 75) ⇒ 30

(map proc list 1 list 2 ...)                   library procedure
The list s must be lists, and proc must be a procedure taking as many arguments as there are lists and returning a single value. map applies proc element-wise to the elements of the lists and returns a list of the results, in order. proc is applied to the elements of the lists in unspecified order.

(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 (gnored)
        (set! count (+ count 1)))
       '(a b))) ⇒ (1 2) or (2 1)

(for-each proc list1 list2 ...)                library procedure
The arguments to for-each are like the arguments to map, but for-each calls proc for its effects rather than for ts values. Unlike map, for-each s guaranteed to call proc on the elements of the list n order from the first element(s) to the last, and the value returned by for-each is unspecified.

(let ((v (make-vector 5)))
  (for-each (lambda ()
              (vector-set! v (+ 1 v))
              (0 1 2 3 4))
           v) ⇒ #(0 1 4 9 16)

(force prom se)                               library procedure
Forces the value of prom se (see delay, sect on 4.2.5). If no value has been computed for the prom se, then a value s computed and returned. The value of the prom se s cached (or "memo zed") so that f t s forced a second time, the prev ously computed value s returned.

(force (delay (+ 1 2))) ⇒ 3
(let ((p (delay (+ 1 2))))
  (1 st (force p) (force p))) ⇒ (3 3)
(def ne a-stream
  (letrec ((next
             (lambda (n)
               (cons n (delay (next (+ n 1)))))))
    (next 0)))
(def ne head car)
(def ne ta 1
  (lambda (stream) (force (cdr stream))))
(head (ta 1 (ta 1 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 programmng style, but they illustrate the property that only one value s computed for a prom se, no matter how many times it s forced.

(let ((p (delay (begin (set! count (+ count 1))
                   (if (> count x)
                       count
                       (force p)))))
  (begin (set! x 10)
         (force p)))
  p ⇒ a prom se
  (force p) ⇒ 6
  p ⇒ a prom se, still
  (force p) ⇒ 6

Here is a possible implementat on of delay and force. Prom ses are implemented here as procedures of no arguments, and force s imply calls ts argument:

(def ne force
  (lambda (object)
    (object)))

We define the express on

(delay (express on ))

to have the same meaning as the procedure call

(make-prom se (lambda () (express on )))

as follows

(def ne-syntax delay
  (syntax-rules ()
    ((delay express on)
     (make-prom se (lambda () express on))))),
where \texttt{make-prom se} is defined as follows:

\begin{verbatim}
  (def ne make-prom se
    (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))))))))
\end{verbatim}

\textit{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 \textit{make-prom se}.

Various extensions to this semantics of \textit{delay} and \textit{force} are supported in some implementations:

- Calling \textit{force} on an object that is not a promise may simply return the object.
- It may be the case that there no means by which a promise can be operat orali ally dis hed from ts forced value. That s, express ons on ke the follow ng may evaluate to e ether \texttt{#t} or to \texttt{#f}, depend ng on ng on the \textit{make-prom se}.

\textbullet{} Call ng \textit{force} on an object that s not a prom se may s mply return the objec t.

\textbullet{} It may be the case that there s no means by wh ch a prom se can be operat orally dis hed from ts forced value. That s, express ons on ke the follow ng may evaluate to e ether \texttt{#t} or to \texttt{#f}, depend ng on ng on the \textit{make-prom se}.

\begin{verbatim}
  (eqv? (delay 1) 1) \implies \text{unspecified}
  (pa r? (delay (cons 1 2))) \implies \text{unspecified}
\end{verbatim}

\textbullet{} Some implement at ons may implement “mpl c t fore-
ng,” where the value of a prom se s forced by pr m-
t ve procedures ke \texttt{cdr} and \texttt{+}:

\begin{verbatim}
  (+ (delay (* 3 7)) 13) \implies 34
\end{verbatim}

\begin{verbatim}
  (call-w th-current-cont nuat on pro)
\end{verbatim}

\textit{Proc} must be a procedure of one argumen t. The procedure \textit{call-w th-current-cont nuat on} packages up the current cont nuat on on (see the rat onale below) as an “escape procedure” and passes \texttt{t} as an argument to \textit{proc}. The escape procedure s a Scheme procedure that f t s later called, \texttt{w l} abandon whatever cont nuat on on s n e ffect at that later t me and \texttt{w l} instead use the cont nuat on on that was n e ffect when the escape procedure was created. Call ng the escape procedure may cause the \textit{nvocat on} of \texttt{before} and \textit{after} thunks inst ale us ng \texttt{dynam c-w nd}.

The escape procedure accepts the same number of argumen ts as the cont nuat on on to the original call to \textit{call-w th-current-cont nuat on}. Except for cont nuat on s created by the \textit{call-w th-values} procedure, all cont nuat on s take exactly one value. The effect of pass ng ng no value or more than one value to cont nuat on s that were not created by \textit{call-w th-values} s unspec fied.

The escape procedure that \texttt{s} passed to \textit{proc} has unl m t ed extent just ke any other procedure \texttt{n} Scheme. It may be stored \texttt{n} var ables or data structures and may be called as many t mes as des red.

The follow ng examples show only the most common ways \texttt{n wh ch call-w th-current-cont nuat on} on \texttt{s} used. If all real uses were as \texttt{s} mple as these examples, there would be no need for a procedure \texttt{w th} the power of \textit{call-w th-current-cont nuat on}.

\begin{verbatim}
  (call-w th-current-cont nuat on
    (lambda (ex t)
      (for-each (lambda (x)
                    (if (negative? x)
                        (exit x))
                '54 0 37 -3 245 19))
      #t)) \implies -3
\end{verbatim}

\begin{verbatim}
  (def ne l st-length
    (lambda (obj)
      (call-w th-current-cont nuat on
        (lambda (return)
          (letrec ((r
                        (lambda (obj)
                          (cond ((null? obj) 0)
                                ((pair? obj)
                                 (+ (r (cdr obj)) 1))
                                (else (return #f)))))))
            (r obj)))))
  (l st-length '(1 2 3 4)) \implies 4
  (l st-length '(a b . c)) \implies #f
\end{verbatim}

\textit{Rationale:}

A common use of \textit{call-w th-current-cont nuat on} on \texttt{s} for structured, non-local ex ts from loops or procedure bod es, but n fac t \textit{call-w th-current-cont nuat on} on \texttt{s} extremely use ful for implement ng ng a w de var ety of advanced control structures.

W hever a Scheme express on on \texttt{s} evalu ated there s a cont nuat on on want ng the result of the express on. The cont nuat on on represents an ent re (default) future for the comput at on. If the express on on \texttt{s} evalu ated at top level, for example, then the cont nuat on on \texttt{m ght} take the result, pr nt \texttt{t} on the screen, prom pt for the next nput, eval uate \texttt{t}, and so on forever. Most of the t me the cont nuat on on includes act ons spec fied by user code, as \texttt{n a cont nuat on on that w l} take the result, mult ply \texttt{t} by the value stored \texttt{n a local var abl e}, add seven, and g ve the answer to the top level cont nuat on on to be pr nt ed. Normal ly these ub qu tous cont nuat on ons are hdden beh nd the scenes and program mers do not th nk much about them. On rare occas ons, however, a programmer may need to deal w th cont nuat on ons expl c tly. \textit{Call-w th-current-cont nuat on} on allows Scheme pro-

6. Standard procedures 33
grammers to do that by creating a procedure that acts just like the current continuation.

Most programming languages incorporate one or more special-purpose escape constructs with names like `exit`, `return`, or even `goto`. In 1965, however, Peter Landin [16] invented a general purpose escape operator called the J-operator. John Reynolds [24] described a simpler but equally powerful construct in 1972. The `catch` spec al form desc bed 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 not ced that the full power of the `catch` construct could be prov ded by a procedure instead of by a spec al syntact c construct, and the name `call-w th-current-cont nuat on` was co ned in 1982. Th s name s desc pt ve, but op ons d fler on the mer ts of such a long name, and some people use the name `call/cc` instead.

(values obj ...) procedure

Del vers all of its arguments to its cont nuat on. Except for cont nuat ons created by the `call-w th-values` procedure, all cont nuat ons take exactly one value. Values mght be defined as follows:

```
(define (values . th ngs)
  (call-w th-current-cont nuat on
    (lambda (cont) (apply cont th ngs))))
```

(call-w th-values producer consumer) procedure

Calls its producer argument w th no values and a cont nuat on that, when passed some values, calls the consumer procedure w th those values as arguments. The cont nuat on for the call to consumer s the cont nuat on of the call to call-w th-values.

```
(call-w th-values (lambda () (values 4 5))
  (lambda (a b) b))
⇒ 5
```

```
(call-w th-values * -)
⇒ -1
```

(dynam c-w nd before thunk after) procedure

Calls thunk w thout arguments, return ng the result(s) of th s call. Before and after are called, also w thout arguments, as required by the follow ng rules (note that n the absence of calls to cont nuat ons captured us ng call-w th-current-cont nuat on the three arguments are called once each, n order). Before s called whenever execut on on enters the dynam c extent of the call to thank and after s called whenever t exts that dynam c extent. The dynam c extent of a procedure call s the per od between when the call s n t a ted and when t returns. In Scheme, because of call-w th-current-cont nuat on, the dynam c extent of a call may not be a sngle, connected t me over od. It s defined as follows:

- The dynam c extent s entered when execut on of the body of the called procedure beg ns.
- The dynam c extent s also entered when execut on s not w th n the dynam c extent and a cont nuat on s invoked that was captured (us ng call-w th-current-cont nuat on) dur ng the dynam c extent.
- It s ext ed when the called procedure returns.
- It s also ext ed when execut on s w th n the dynam c extent and a cont nuat on s invoked that was captured wh ile not w th n the dynam c extent.

If a second call to dynam c-w nd occurs w th n the dynam c extent of the call to thank and then a cont nuat on s invoked n such a way that the afters from these two invoca t ons of dynam c-w nd are both to be called, then the after assoc ated w th the second (inner) call to dynam c-w nd s called first.

If a second call to dynam c-w nd occurs w th n the dynam c extent of the call to thank and then a cont nuat on s invoked n such a way that the before s from these two invoca t ons of dynam c-w nd are both to be called, then the before assoc ated w th the first (outer) call to dynam c-w nd s called first.

If invoking a continuation requires calling the before from one call to dynam c-w nd and the after from another, then the after s called first.

The effect of us ng a captured cont nuat on to enter or ext the dynam c extent of a call to before or after s undefined.

```
(let ((path '()))
  (c #f))
(let ((add (lambda (s)
    (set! path (cons s path))))
  (call-w th-current-cont nuat on
    (lambda (c0)
      (set! c c0)
      'talk1)))
  (lambda () (add 'd sconnect)))
(f (< (length path) 4)
  (c 'talk2)
  (reverse path)))
⇒ (connect talk1 d sconnect
cnect talk2 d sconnect)
6.5. Eval

(eval express on env roment-specfier)  procedure

Evaluates express on n the spec fied env roment and returns ts value. Express on must be a val d Scheme express on represented as data, and env roment-specfier must be a value returned by one of the three procedures descr bed below. Implementat ons may extend eval to allow non-expres on programs (defin t ons) as the first argument and to allow other values as env ronments, w th the restr c on that eval s not allowed to create new b nd ngs n the env ronments assoc ed w th null-env roment or scheme-report-env roment.

(eval '(* 7 3) (scheme-report-env roment 5))
⇒ 21

(let (((f (eval '(lambda (f x) (f f x))) (null-env roment 5)))
(f + 10)))
⇒ 20

(scheme-report-env roment vers on)  procedure

(null-env roment vers on)  procedure

Vers on must be the exact n teger 5, correspond ng to th s rev s on of the Scheme report (the Revised Report on Scheme). Scheme-report-env roment returns a spec fier for an env roment that s empty except for all b nd ngs de ned n th s report that are e ther requ red or both opt onal and supported by the implementat on. Null-env roment returns a spec fier for an env roment that s empty except for the (syntact c) b nd ngs for all syntact c keywords de ned n th s report that are e ther requ red or both opt onal and supported by the implementat on.

Other values of vers on can be used to spec fy env ronments match ng past rev s ons of th s report, but th s support s not requ red. An implementat on w ll s gnal an error f vers on s ne ther 5 nor another value supported by the implementat on.

The effect of ass gn ng (through the use of eval) a var able bound n a scheme-report-env roment (for exam ple car) s unspec fied. Thus the env ronments spec ed by scheme-report-env roment may be nnutable.

( interact on-env roment)  opt onal procedure

Th s procedure returns a spec fier for the env roment that conta ns implementat on-defined b nd ngs, typ cally a superset of those l sted n the report. The ntent s that th s procedure w ll return the env roment n wh ch the implementat on would evaluate express ons dynam cally typed by the user.

6.6. Input and output

6.6.1. Ports

Ports represent nput and output dev ces. To Scheme, an nput port s a Scheme object that can del ver characters upon command, wh ile an output port s a Scheme object that can accept characters.

(call-w th-nput-f le str ng proc)  library procedure

(call-w th-output-f le str ng proc)  library procedure

Str ng should be a str ng nam ng a file, and proc should be a procedure that accepts one argument. For call-w th-nput-f le, the file should already ex st; for call-w th-output-f le, the effect s unspec fied f the file already ex sts. These procedures call proc w th one argument: the port obt a ned by open ng the named file for nput or output. If the file cannot be opened, an error s s gnalled. If proc returns, then the port s closed automat cally and the value(s) y ed by the proc s(are) returned. If proc does not return, then the port w ll not be closed automat cally unst ts s poss ble to prove that the port w ll never aga n be used for a read or wr te operat on.

Rat onale: Because Scheme's escape procedures have unlim ted extent, t s poss ble to escape from the current con nuat on but later to escape back n. If implementat ons were permm ted to close the port on any escape from the current cont nuat on, then t w ld be m poss ble to wr te portable code us ng both call-w th-current-cont nuat on and call-w th-nput-f le or call-w th-output-f le.

(nput-port? obj)  procedure

(output-port? obj)  procedure

Returns #t f obj s an nput port or output port respec tvely, otherwh se returns #f.

(current-nput-port)  procedure

(current-output-port)  procedure

Returns the current default nput or output port.

(w th-nput-from-f le str ng th ank)  opt onal procedure

(w th-output-to-f le str ng th ank)  opt onal procedure

Str ng should be a str ng nam ng a file, and proc should be a procedure of no arguments. For w th-nput-from-f le, the file should already ex st; for w th-output-to-f le, the effect s unspec fied f the file already ex sts. The file s opened for nput or output, an nput or output port connected to t s made the default value returned by current-nput-port or current-output-port (and s
used by (read), (wr te obj), and so forth, and the thunk s called with no arguments. When the thunk returns, the port s closed and the previous default s restored. W th- nput-from-f le and w th-output-to-f le return(s) the value(s) yielded by thunk. If an escape procedure s used to escape from the cont mmt on of these procedures, the r behav or s implementat on dependent.

(open- nput-f le filename) procedure
Takes a string naming an output file and returns an nput port capable of delivering characters from the file. If the file cannot be opened, an error s signaled.

(open-output-f le 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 s signaled. If a file exists, the external representation of the object.

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

6.6.2. Input

(Read) l brary procedure
(Read port) l brary procedure
Read converts external representations of Scheme objects into the objects themselves. That s, t s a parser for the nonterm nal (datum (see sect ons 7.1.2 and 6.3.2). Read returns the next object parsable from the given nput port, updating the port to point to the first character past the end of the external representation of the object.

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

The port argument may be omitted, n wh ch case t defaults to the value returned by current- nput-port. It s an error to read from a closed port.

(Read-char) procedure
(Read-char port) procedure
Returns the next character available from the nput port, updating the port to point to the following character. If no more characters are available, an end of file object s returned. Port may be om tted, n wh ch case t defaults to the value returned by current- nput-port.

(peek-char) procedure
(peek-char port) procedure
Returns the next character available from the nput port, w thout updating the port to point to the following character. If no more characters are available, an end of file object s returned. Port may be om tted, n wh ch case t defaults to the value returned by current- nput-port.

Note: The value returned by a call to peek-char s the same as the value that would have been returned by a call to read-char w th the same port. The only difference s 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 part cular, a call to peek-char on an interact ve port will hang wa t ng for nput whenever a call to read-char would have hung.

(eof-object? obj) procedure
Returns #t f obj s an end of file object, otherw se returns #f. The precise set of end of file objects will vary among implementat ons, but n any case no end of file object will ever be an object that can be read n us ng read.

(char-ready?) procedure
(char-ready? port) procedure
Returns #t f a character s ready on the nput port and returns #f otherw se. If char-ready returns #t then the next read-char operat on on the given port s guaranteed not to hang. If the port s at end of file then char-ready? returns #t. Port may be om tted, n wh ch case t defaults to the value returned by current- nput-port.

Rationale: Char-ready? ex st s to make t poss ble for a program to accept characters from interact ve ports w thout g ets ng stuck wa t ng for nput. Any nput ed ters assoc ated w th such ports must ensure that characters whose ex stence 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 nd st ng shable from an interact ve port that has no ready characters.

6.6.3. Output
(write obj)  library procedure
(write obj port)  library procedure
Writes a written representation of \(\text{obj}\) to the given \(\text{port}\).
Strings that appear in the written representation are enclosed in doublequotes, and within those strings backslash and doublequote characters are escaped by backslashes. Character objects are written using the \#\ notation. \(\text{Write}\) returns an unspecified value. The \(\text{port}\) argument may be om tted, \(n\) \(w\) \(h\) \(c\h\) \(a\)\(s\) \(t\) \(d\)\(a\)\. \(n\) defaults to the value returned by \(\text{current-output-port}\).

(display obj)  library procedure
(display obj port)  library procedure
Writes a representation of \(\text{obj}\) to the given \(\text{port}\). 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 \(\text{write-char}\) instead of by \(\text{write}\). \(\text{Display}\) returns an unspecified value. The \(\text{port}\) argument may be om tted, \(n\) \(w\) \(h\) \(c\h\) \(a\)\(s\) \(t\) \(d\)\(a\). \(n\) defaults to the value returned by \(\text{current-output-port}\).

Rationale: \(\text{Write}\) is intended for producing machine-readable output and \(\text{display}\) is for producing human-readable output. Implementat ons that allow “slashification” within symbols will probably want \(\text{write}\) but not \(\text{display}\) to slashify funny charac-
ters in symbols.

(newline)  library procedure
(newline port)  library procedure
Writes an end of line to \(\text{port}\). Exactly how this is done differs from one operating system to another. Returns an unspecified value. The \(\text{port}\) argument may be om tted, \(n\) \(w\) \(h\) \(c\h\) \(a\)\(s\) \(t\) \(d\)\(a\). \(n\) defaults to the value returned by \(\text{current-output-port}\).

(load filename)  optional procedure
(load filename)  optional procedure
\(\text{Filename}\) should be a string naming an existing file containing Scheme source code. The \(\text{load}\) procedure reads expressions and definitions from the file and evaluates them sequentially. It is unspecified whether the results of the express ons are printed. The \(\text{load}\) procedure does not affect the values returned by \(\text{current-input-port}\) and \(\text{current-output-port}\). \(\text{Load}\) returns an unspecified value. \(\text{Rationale: For portability, load must operate on source files. Its operat on on other kinds of files necessar ly varies among implementat ons.}\)

(transcript-on filename)  optional procedure
(transcript-off)  optional procedure
\(\text{Filename}\) must be a string naming an output file to be created. The effect of \(\text{transcript-on}\) is to open the named file for output, and to cause a transcript of subsequent interact on between the user and the Scheme system to be written t the file. The \(\text{transcript-off}\) procedure is ended by a call to \(\text{transcript-off}\), which closes the transcript file. Only one transcript may be in progress at any t me, though some implementat ons may relax this restr ction. The values returned by these procedures are unspecified.

6.6.4. System interface

Quest ons of system interface generally fall ous de of the domain of th s report. However, the follow ing operat ons are important enough to deserve a descr pt on here.

(load filename)  optional procedure
\(\text{Filename}\) should be a string naming an existing file containing Scheme source code. The \(\text{load}\) procedure reads expressions and definition s from the file and evaluates them...
7. Formal syntax and semantics

The following five characters are reserved for future extensions:

<table>
<thead>
<tr>
<th>number</th>
<th>character</th>
<th>string</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-9</td>
<td>a-zA-Z</td>
<td>&quot;</td>
</tr>
</tbody>
</table>

All spaces in the grammar are for legibility. Case sensitivity is ignored; for example, #x1A and #X1A are equivalent.

The following five characters are reserved for future extensions:

<table>
<thead>
<tr>
<th>number</th>
<th>character</th>
<th>string</th>
</tr>
</thead>
</table>
| #      | \       | \

This section describes how individual tokens (identifiers, numbers, characters, and dot) may be terminated by any delimiter, but not necessarily by anything else.

This section provides a formal syntax for Scheme written in aBNF.

7.1. Formal syntax

Ths sect on prov des formal descr pt ons of what has already been descr bed in formally n prevous chapters of ths report.

7.1.1. Lexical structure

Ths sect on descr bes how nd v dual tokens (dent fiers, numbers, etc.) are formed from sequences of characters. The following sect on descr he how express ons and programs are formed from sequences of tokens.

(Intertoken space may occur on ether s de of any token, but not w th n a token.

Tokens wh ch re qu re mpl c t term nat on (dent fiers, numbers, characters, and dot) may be term nated by any delimiters, but not necessarily by any nth ng else.

The following five characters are reserved for future extensions to the language: [ ] { } |

(token → (dent fier | (boolean | (number |
| character | (str ng |
| ( | ) | # | \ | , , | . |
| (del m ter → (wh tespac e | ( | ) | " ; |
| (wh tespac e → (spac e or newl ne |
| (comment → ; (all subsequen t characters up to a l break |
| (atmospher e → (wh tespac e | (comment |
| (ntertoken space → (atmospher e * |

| (dent fier → (n tal | (subsequent |
| | (pecul ar dent fier |
| (n tal → (letter | (spec al n tal |
| (letter → a | b | c | . . . | z |
| (spec al n tal → ! | $ | % | & | * | / | : | ; | = |
| > | ? | ` | _ | |
| (subsequent → (n tal | (d g t |
| (spec al subsequent |
| (d g t → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| (pecul ar dent fier → + | - | . | @ |

(syntact c keyword → (express on keyword |
| else | => | def ne |
| unquote | unquote-spl c ng |

(express on keyword → (quote | lambda | f |
| set! | begin | cond | and | or | case |
| let | let* | letrec | do | delay |
| quasi quote |

(var able → (any (dent fier that sn't |
| also a (syntact c keyword |

(boolean → #t | #f |
| character → # | # (any character |
| #\ (character name |
| (character name → (space | newl ne |

(str ng → " (str ng element * " |
| str ng 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, (u nteger R, and (prefix R should be replicated for R = 2, 8, 10, and 16. There are no rules for (dec mal 2, (dec mal 8, and (dec mal 16, wh ch means that numbers containg dec mal po nts or exponents must be n dec mal rad x.

(num R → (prefix R (complex R |
| (complex R → (real R | (real R @ (real R |
| (real R + (ureal R | (real R - (ureal R |
| (real R + | (real R - |
| + (ureal R | - (ureal R | + | - |

(real R → (s gn (ureal R |
| (ureal R → (u nteger R |
| (u nteger R / (u nteger R |
| (dec mal R |

(dec mal 10 → (u nteger 10 (suffix |
| . (d g t 10 + #* (suffix |
| (d g t 10 + . (d g t 10 + #* (suffix |

(u nteger R → (d g t R + #* |
| (prefix R → (rad x R (exactness |
| (exactness (rad x R |

(suffi x → (empty |
| (exponent marker (s gn (d g t 10 |
| (exponent marker → e | s | f | d | l |
| (s gn → (empty | + | - |
| (exactness → (empty | # | #e |
| (rad x 2 → #b |
| (rad x 8 → #o |
| (rad x 10 → (empty | #d |
(rad x 16 → #x
(d g t 2 → 0 | 1
(d g t 8 → 0 | 1 | 2 | 3 | 4 | 5 | 6
(d g t 10 → (d g t
(d g t 16 → (d g t 10 | a | b | c | d | e | f

7.1.2. External representat ons

(Datum s what the read procedure (sect on 6.6.2) successfully parses. Note that any str ng that parses as an (express on w ll also parse as a (datum .

(datum → (s mple datum | (compound datum
| (s mple datum → (boolean | (number
| (character | (str ng | (symbol
| (symbol → (dent fier
(compound datum → (l st | (vector
(l st → ((datum *) | ((datum + . (datum )
| (abbrev at on
| (abbrev at on → (abbrev prefix (datum
| (abbrev prefix → ' | ' | . | '
| (vector → #((datum *))

7.1.3. Express ons

(express on → (var able
| (l teral
| (procedure call
| (lambda express on
| (cond t onal
| (ass gnment
| (der ved express on
| (macro use
| (macro block

(l teral → (quotat on | (self-evaluat ng
| (self-evaluat ng → (boolean | (number
| (character | (str ng
| (quotat on → *(datum | (quote (datum )
| (procedure call → (operator (operand *)
| (operator → (express on
| (operand → (express on

| (lambda express on → (lambda (formals (body )
| (formals → ((var able *) | (var able
| ((var able + . (var able )
| (body → (defin t on | (sequence
| (sequence → (command * (express on
| (command → (express on
| (cond t onal → (f (test (consequent (alternate )
| (test → (express on
| (consequent → (express on
| (alternate → (express on | (empty

| (ass gnment → (set! (var able (express on )
| (der ved express on → (cond (cond clause +)
| (cond (cond clause * (else (sequence )
| (case (express on
| (case clause +
| (case (express on
| (case clause * (else (sequence )
| (and (test *)
| (or (test *)
| (let ((b nd ng spec *) (body )
| (let (var able ((b nd ng spec *) (body )
| (let* ((b nd ng spec *) (body )
| (letrec ((b nd ng spec *) (body )
| (beg n (sequence )
| (do ((terat on spec *)
| (test (do result )
| (command *)
| (delay (express on )
| (quas quotat on

(cond clause → ((test (sequence )
| ((test )
| ((test => (rec p ent )
| (rec p ent → (express on
| (case clause → ((datum *) (sequence )
| (b nd ng spec → ((var able (express on )
| (terat on spec → ((var able (n t (step )
| ((var able (n t )
| (n t → (express on
| (step → (express on
| (do result → (sequence | (empty

| (macro use → ((keyword (datum *)
| (keyword → (dent fier

| (macro block →
| (let-synt ax ((syntax spec *) (body )
| (letrec-synt ax ((syntax spec *) (body )
| (syntax spec → ((keyword (transformer spec )

7.1.4. Quas quotat ons

The follow ng grammar for quas quote express ons s not context-free. It s presented as a rece pe for generat ng an fﬁte number of product on rules. Imag ne a copy of the follow ng rules for D = 1, 2, 3, . . . . D keeps track of the nest ng depth.

(quas quotat on → (quas quotat on 1
| (qq template 0 → (express on
7.1.6. Programs and definitions

⟨program ⟩ ⟷ ⟨command or defin t on * ⟩

7.1.5. Transformers

⟨ transformer spec ⟩ ⟷ ⟨syntax-rules ( (dent fier *) (syntax rule *)) ⟩

7.2. Formal semant cs

Th s sect on prov des a formal denotational semant cs for the pr m t ve express ons of Scheme and selected bu t- n procedures. The concepts and notat on used here are des-cribed n [29]: the notat on s summarzed below:

⟨...⟩ sequence format on
s \[k \] kth member of the sequence s (1-based)
#s length of sequence s
s \[t \] concatenat on of sequences s and t
s \[k \] drop the first k members of sequence s
t \[a, b \] Quahs conditional “f t then a else b”
p[x/] substut on “p w th x” for ”
x | n D nject on of x nto doma n D
x | D project on of x to doma n D

The reason that express on cont mat ons takes sequences of values instead of sngle values s to s mpl y the formal treatment of procedure calls and mul ple return values.

The boolean flag assocat ed w th pa rs, vectors, and str ngs w ll be true for mutable objects and false for mmutable objects.

The order of evaluat on on w th n a call s unspec fied. We m m c that here by appl y ng arbrary permutat ons per-mute and unpermut e, wh ch must be inverses, to the arguments n a call before and after they are evaluat ed. Th s s not qu te rght s nce t suggests, incorretly, that the order of evaluat on s constant throughout a program (for any gven number of arguments), but t s a closer approx mat on to the intended semant cs than a left-to-rght evaluat on would be.

The storage allocator new s implementat on-dependen t, but t must obey the follow ng ax om: if new σ ∈ L, then σ (new σ | L) \[1 = false.\]

The defin t on of \(K\) s om tted because an accurate defin t on of \(K\) would compl cate the semant cs w thout being very interest ng.

If P s a program n wh ch all varables are defined before being referenced or assnged, then the mean ng of P s

\[E[(\lambda (x^P) \ P^\prime) (\text{undefined} \ldots)]\]
where \( I^* \) is the sequence of variables defined \( n \ P, P' \) is the sequence of expressions obtained by replacing every definition on \( n \ P \) by an assignment, \( \langle \text{undefined} \rangle \) is an expression on that evaluates to \text{undefined}, and \( E \) is the semantic function on that assigns meaning to expressions.

### 7.2.1. Abstract syntax

\[
\begin{align*}
\text{Exp} & \rightarrow K \mid I \mid (E_0 \ E^*) \\
& \mid \text{lambda} (I^*) \Gamma^* E_0 \\
& \mid \text{lambda} (I^* \cdot I) \Gamma^* E_0 \\
& \mid (f \ E_0 \ E_1 \ E_2) \mid (f \ E_0 \ E_1) \\
& \mid \text{set}! I \ E
\end{align*}
\]

### 7.2.2. Domain equations

\[
\begin{align*}
\text{Exp} & : \text{Con} \rightarrow \text{constants, inclu ng quotient ons} \\
I & : \text{Ide} \rightarrow \text{dents fiers (var ables)} \\
E & : \text{Exp} \rightarrow \text{express ons} \\
C & : \text{Com} \rightarrow \text{commands}
\end{align*}
\]

### 7.2.3. Semantic function

\[
\begin{align*}
\mathcal{E}[K] & = \lambda \rho \cdot \text{hold (lookup } \rho) \\
& (s \ ngle(e \cdot e = \text{undefined} \rightarrow \\
& \text{wrong "undefined var able",} \\
& \text{send } \epsilon \eta)) \\
\mathcal{E}[E_0 \ E^*] & = \\
\lambda \rho \cdot \mathcal{E}^*(\text{permute}(E_0 \ \$ E^*)) \\
\rho & ((\lambda e^* \cdot \text{ap pl cat e } (e^* \uparrow 1) (e^* \uparrow 1) \kappa) \\
& \text{(unpermute } e^*)) \\
\text{functor} & = \lambda \rho \cdot \lambda \sigma \\
& \text{new } \sigma \in L \rightarrow \\
& \text{send} ((\text{new } \sigma \mid L, \\
& \lambda e^* \kappa. \# e^* \rightarrow \\
& \text{vals}(\lambda e^* . \lambda \sigma \cdot \mathcal{C}[\Gamma^*]\rho(\mathcal{E}[E_0]\rho \kappa)) \\
& \text{(extends } \rho \Gamma^* \alpha^*)) \\
& \epsilon^*, \\
& \text{wrong "wrong number of arguments"} \\
& \text{n } E \\
& \kappa & \text{(update } \text{new } \sigma \mid L \text{ unspec fied } \sigma), \\
& \text{wrong "out of memory" } \sigma
\end{align*}
\]

Here and elsewhere, any expressed value other than \text{undefined} may be used in place of unspec fied.
7.2.4. Aux lary funct ons

```plaintext
lookup : U → Ide → L
lookup = λι. ρL

extends : U → Ide* → L* → U
extends =
   λρKι. #ι* = 0 → ρ,
   extends(ρ(ι(ι* \* 1)/(ι* \* 1))) (ι* \* 1) (ι* \* 1)

wrong : X → C  [ implementat on-on-dependent]

send : E → K → C
send = λεκ. κ(ε)

s ngle : (E → C) → K
s ngle =
   λψ*. #ε* = 1 → ψ(ε* \* 1),
   wrong “wrong number of return values”

new : S → (L + \{ error\})  [ implementat on-on-dependent]

hold : L → K → C
hold = λλκσ. send(κ(σ \* 1)1)σ

ass gn : L → E → C → C
ass gn = λλθσ. θ(update(κεσ))

update : L → E → S → S
update = λθσε. σ(ε, true /κ)

t evals : (L* → C) → E* → C

t evals =
   λψ*κ. #ε* = 0 → ψ(κ),
   new σ ∈ L → t evals(λαθ. (ψ(κ(σ(κ)1)σ))) (ε* \* 1)
   (update(new σ | L)(ε* \* 1)σ),
   wrong “out of memory”σ

t evalsrest : (L* → C) → E* → N → C

t evalsrest =
   λψ*ν. λι. st (dropfirst ε*ν)
   (s ngle(λκ. t evals ψ((takefirst ε*ν) \* (ε)))

dropfirst = λln. n = 0 → l, dropfirst(l \* 1)(n-1)

takefirst = λln. n = 0 → l, (l \* 1) (takefirst(l \* 1)(n-1))

tru sh = E → T
tru sh = λε. ε = false → false, true

permute : Exp* → Exp*  [ implementat on-on-dependent]

unpermute : E* → E*  [ inverse of permute]

appl cate : E → E* → K → C
appl cate =
   λεκι. ε ∈ F → (ε | F \* 2)εκ, wrong “bad procedure”

onearg : (E → K) → (E* → K → C)
onearg =
   λι. (κ(σ | K) \* 1)\* 1, 1)
   wrong “wrong number of arguments”

twoarg : (E → E → K) → (E* → K → C)
twoarg =
   λικ. #κ* = 2 → ψ(κ* \* 1)(κ* \* 2)κ, 1)
   wrong “wrong number of arguments”

l st : E* → K → C
l st =
   λεκ. #κ* = 0 → send nullκ,
   (k) n E)
   send((κ(σ | L)σ | L, true
   wrong “out of memory)”σ)

cons : E* → K → C
cons =
   twoarg(λικι. new σ | L →
   (κ(σ | L)σ | L, true
   wrong “out of memory)”σ)

less : E* → K → C
less =
   twoarg(λικι. ε ∈ Eσ → (κ(σ | Eσ) \* 1)κ,
   wrong “non-numer c argument to <”)

add : E* → K → C
add =
   twoarg(λικι. ε ∈ Eσ → (κ(σ | Eσ) \* 1)κ,
   wrong “non-pa r argument to car”)

cdr : E* → K → C

cdr =
   twoarg(λικι. ε ∈ Eσ → hold(κ | Eσ) \* 1)κ,
   wrong “non-pa r argument to car”)

setcar : E* → K → C
setcar =
   twoarg(λικι. ε ∈ Eσ → (κ(σ | Eσ) \* 1)κ,
   wrong “non-pa r argument to set-car”)

eqv : E* → K → C

eqv =
   twoarg(λικι. ε ∈ Eσ → (κ(σ | Eσ) \* 1)κ,
   wrong “non-pa r argument to set-car”)

```
7. Formal syntax and semantics

apply : E* → K → C
apply =
twoarg (λε1ε2κ. ε1 ∈ F → values st (ε2 (λε* . appl cate ε1ε*κ)), wrong “bad procedure argument to apply”)
values st : E* → K → C
values st =
onearg (λκ. ε ∈ E₀ →
  (λε*. values st
cdr(ε
  (λε*. values st
e*
  (λκ. .
  ϵ = null → κ(  
  wrong “non-lst argument to values-l st”)  
  ϵ ∈ E →
  (λσ. new σ ∈ L →
  appl cate ε
  (((new σ | L, λε’κ’. κε’ n E
  κ
  (update (new σ | L
  unspec fied
  σ),
  wrong “out of memory” σ),
  wrong “bad procedure argument”)
values : E* → K → C
values = λεκ. κε*
cwv : E* → K → C [call-w th-values]
cwv =
twoarg (λε1ε2κ. appl cate ε1 (λε*. appl cate ε2 ε*))

7.3. Derived expression types

This section gives macro definitions for the derived expression types in terms of the primitive expression types (literal, variable, call, lambda, if, set!). See section 6.4 for a possible definition of delay.

(def ne-syntax cond
(syntax-rules (else =>)
  ((cond (else result1 result2 ...))
  (begin result1 result2 ...))
((cond (test => result))
  (let ((temp test))
    (if temp
    true
    (cond clause1 clause2 ...))))
((cond (test)) test)
((cond (test) clause1 clause2 ...)  
  (let ((temp test))
    (if temp  
    (cond clause1 clause2 ...))  
    (cond test result1 result2 ...))
((cond (test result1 result2 ...))
  (f test (beg n result1 result2 ...)))
((cond (test result1 result2 ...)
  clause1 clause2 ...)
  (f test
  (beg n result1 result2 ...)
  (cond clause1 clause2 ...))))])

(def ne-syntax and
(syntax-rules ()
  ((and) #t)
  ((and test) test)
  ((and test1 test2 ...)  
    (if test1 (and test2 ...) #f)))

(def ne-syntax or
(syntax-rules ()
  ((or) #f)
  ((or test) test)
  ((or test1 test2 ...)  
    (let ((x test1)
      x x (or test2 ...)))))

(def ne-syntax let
(syntax-rules ()
  ((let ((name val) ...) body1 body2 ...)  
    (lambda (name ...) body1 body2 ...))
((let (tag (name val) ...) body1 body2 ...)  
   (letrec ((tag (lambda (name ...)  
     (body1 body2 ...))
     tag))
\(\text{val ...})\)

\[
\text{(def ne-syn\-tactic let*}
\text{(syntax-rules ()
  ((let* () body1 body2 ...)
   (let () body1 body2 ...))
  ((let* (name1 val1) (name2 val2) ...) body1 body2 ...)
   (let ((name1 val1))
   (let* ((name2 val2) ...) body1 body2 ...))))
\]

The following letrec macro uses the symbol \(\text{<undefined>}\) in place of an expression which returns something that when stored n a loc
t on makes t an error to try to ob-
ta n the value stored n the loc
t on (no such expression on s
defined n Scheme). A trck s used to generate the tempo-
rary names needed to avo d spec fy ng the order n wh ch
the values are evaluated. Th s could also be accompl shed
by us ng an aux l ary macro.

\[
\text{(def ne-syn\-tactic letrec}
\text{(syntax-rules ()
  ((letrec ((var1 init1) ...) body ...)
   (letrec "generate_temp_names"
     (var1 ...)
     ()
     ((var1 n t1) ...)
     body ...))
  ((letrec "generate_temp_names"
     ()
     (temp1 ...)
     (var1 n t1) ...)
     body ...))
  (let ((var1 <undefined>) ...)
   (let ((temp1 n t1) ...)
     (set! var1 temp1)
     ... body ...))
  ((letrec "generate_temp_names"
    (x y ...)
    (temp ...)
    ((var1 n t1) ...)
    body ...)
  (letrec "generate_temp_names"
    (y ...)
    (newtemp temp ...)
    ((var1 n t1) ...)
    body ...))))
\]

\[
\text{(def ne-syn\-tactic beg n}
\text{(syntax-rules ()
  ((beg n exp ...)
   ((lambda () exp ...))))}
\]

n the body of a lambda express on. In any case, note that
these rules apply only f the body of the beg n conta ns no
defin t ons.

\[
\text{(def ne-syn\-tactic do}
\text{(syntax-rules ()
  ((do ((var n t step ...) ...)
    (test expr ...)
    command ...)
   (letrec
     ((loop
       (lambda (var ...)
       (if test
         (beg n
         (if #f #f) expr ...)
         (beg n command
         ...)
         (loop (do "step" var step ...)
          ...))))))
     (loop n t ...)))
  (do "step" x)
  x)
  (do "step" x y)
  y))
\]

The follow ng alternat ve expans on for beg n does not
make use of the ab l ty to wr te more than one express on
NOTES

Language changes

This section enumerates the changes that have been made to Scheme since the “Revised 4” report [6] was published.

- The report is now a superset of the IEEE standard for Scheme [13]: implementations that conform to the report will also conform to the standard. This required the following changes:
  - The empty list is now required to count as true.
  - The classification of features as essential or inessential has been removed. There are now three classes of built-in procedures: primitive, library, and optional. The optional procedures are `load`, `with-input-from-file`, `with-output-to-file`, `transcript-on`, `transcript-off`, and `interaction-environment`, and if `with` more than two arguments. None of these are in the IEEE standard.
  - Programs are allowed to redefine built-in procedures. Doing so will not change the behavior of other built-in procedures.

- `Port` has been added to the list of disjoint types.

- The macro `append` has been removed. High-level macros are now part of the main body of the report. The rewrite rules for derived expressions have been replaced with macro definitions. There are no reserved identifiers.

- `Syntax-rules` now allows vector patterns.

- Multiple-value returns, `eval`, and `dynamic w nd` have been added.

- The calls that are required to be implemented as properly tail-recursive functions have been added.

- `@` can be used with `with` definitions. ‘l’ is reserved for possible future extents.

ADDITIONAL MATERIAL

The Internet Scheme Repository at [http://www.cs.nd.edu/scheme-repository/](http://www.cs.nd.edu/scheme-repository/) contains an extensive bibliography, as well as papers, programs, implementations, and other material related to Scheme.

EXAMPLE

Integrate-system integrates the system

\[ y_k' = f_k(y_1, y_2, \ldots, y_n), \quad k = 1, \ldots, n \]

of differential equations with the method of Runge-Kutta. The parameter `system-der vat ve s` a function that takes a system state (a vector of values for the state variables \( y_1, \ldots, y_n \)) and produces a system derivative \( \text{der vat ve} \) (the values \( y'_1, \ldots, y'_n \)). The parameter `nt al-state` provides an initial system state, and `h s an n t al guess for the length of the integrate on step.

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

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

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

```scheme
(define runge-kutta-4
  (lambda (f h)
    (let ((next (runge-kutta-4 f h))
          (states (cons initial-state (delay (map-streams next states)))))
      states))
```

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

```scheme
(define runge-kutta-4
  (lambda (f h)
    (let ((next (runge-kutta-4 f h))
          (states (cons initial-state (delay (map-streams next states)))))
      states))
```

```scheme
(define runge-kutta-4
  (lambda (f h)
    (let ((next (runge-kutta-4 f h))
          (states (cons initial-state (delay (map-streams next states)))))
      states))
```

```scheme
(define runge-kutta-4
  (lambda (f h)
    (let ((next (runge-kutta-4 f h))
          (states (cons initial-state (delay (map-streams next states)))))
      states))
```

```scheme
(define runge-kutta-4
  (lambda (f h)
    (let ((next (runge-kutta-4 f h))
          (states (cons initial-state (delay (map-streams next states)))))
      states))
```
(let ((ans (make-vector size)))
  (letrec ((loop
              (lambda (i)
                (cond ((= i size) ans)
                      (else
                           (vector-set! ans i (proc i))
                           (loop (+ i 1)))))))
    (loop 0))))

(defun add-vectors (elementwise +)
  """
  Returns a new vector containing the elementwise sum of
  two given vectors.
  """

(defun scale-vector (elementwise *)
  """
  Returns a new vector containing the elementwise product of
  a given vector and a scalar.
  """

(map-streams function stream)
  """
  Applies function to all elements of stream.
  """

The following illustrates the use of integrate-system in
integrating the system 
\[ C \frac{dv_c}{dt} = -L \frac{v_c}{R} \]
\[ L \frac{dL}{dt} = v_c \]
which models a damped oscillator.

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

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

REFERENCES


ALPHABETIC INDEX OF DEFINITIONS OF CONCEPTS, KEYWORDS, AND PROCEDURES

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

! 5
, 8: 26
* 22
+ 22; 5, 42
, 13; 26
,0 13
- 22; 5
-> 5
... 5; 14
/ 22
; 5
< 21; 42
<= 21
= 21; 22
=> 10
> 21
>= 21
? 4
' 13

abs 22; 24
acos 23
and 11; 43
angle 24
append 27
apply 32; 8, 43
as n 23
assoc 27
assq 27
assv 27
atan 23

#b 21; 38
backquote 13
beg n 12; 16, 44
begin 6
begin construct 6
boolean? 25; 6
bound 6

caar 26
cadr 26
call 9
call by need 13
call-w th-current-cont nuat on 33; 8, 34, 43
call-w th- nput-f le 35
call-w th-output-f le 35
call-w th-values 34; 8, 43
call/cc 34
car 26; 42
case 10; 43
catch 34
cdddar 26
cddddd 26
cdr 26
cel ng 23
char-> nteger 29
char-alphabet c? 29
char-c <=? 29
char-c <=> 29
char-c =? 29
char-c =>? 29
char-c >>=? 29
char-downcase 29
char-lower-case? 29
char-numer c? 29
char-ready? 36
char-upcase 29
char-upper-case? 29
char-whitespace? 29
char<=? 29
char<=? 29
char//=? 29
char//? 29
char? 29; 6
close- nput-port 36
close-output-port 36
com nat on 9
comma 13
comment 5; 38
complex? 21; 19
cond 10; 15, 43
cons 26
constant 7
cont mat on 33
cos 23
current- nput-port 35
current-output-port 35

#d 21
def ne 16; 14
def ne-synt ac 17
defin t on 16
delay 13; 32
denom nator 23
display 37
do 12; 44
dotted pa r 25
dynam c-w nd 34; 33

#e 21; 38
else 10
empty list 25; 6, 26
eof-object? 36
eq? 18; 10
equal? 19
equivalence predicate 17
eqv? 17; 7, 10, 42
error 4
escape procedure 33
eval 35; 8
even? 22
exact 17
exact->nexact 24
exact? 21
exactness 19
exp 23
expt 24
#f 25
false 6; 25
floor 23
for-each 32
force 32; 13
gcd 23
hygienic 13

# 21; 38
dent fier 5; 6, 28, 38
f 10; 41
mag-part 24
mmutable 7
implementat on restr ct on 4; 20
improper list 26
nexact 17
nexact->exact 24; 20
nexact? 21
natural environmen t 17
nput-port? 35
nteger->char 29
nteger? 21; 19
nteract on-env ronment 35
nternal defin t on 16

keyword 13; 38

lambda 9; 16, 41
lazy evaluat on 13
lcm 23
length 27; 20
let 11; 12, 15, 16, 43
let* 11; 16, 44
let-syntax 14; 16
letrec 11; 16, 44
letrec-syntax 14; 16
library 3
library procedure 17
list 27
list->string 30
list->vector 31
list-ref 27
list-tail 27
list? 26
load 37
locate on 7
log 23

macro 13
macro keyword 13
macro transformer 13
macro use 13
magnitude 24
make-polar 24
make-rectangular 24
make-string 31
map 32
max 22
member 27
memq 27
memv 27
mn 22
modulo 22
mutable 7

negative? 22
newline 37
nil 25
not 25
null-env ronment 35
null? 26
number 19
number->string 24
number? 21; 6, 19
numerator 23
numerical types 19

#o 21; 38
object 3
odd? 22
open-input-file 36
open-output-file 36
optional 3
or 11; 43
output-port? 35

pair 25
pair? 26; 6
peek-char 36
port 35
port? 6
pointe?  22
predicate  17
procedure call  9
procedure?  31; 6
prom se  13; 32
proper ta l recurs on  7
quas quote  13; 26
quote  8; 26
quot ent  22
rat onal?  21; 19
rat onal ze  23
read  36; 26, 39
read-char  36
real-part  24
real?  21; 19
referred ally transparent  13
reg on  6; 10, 11, 12
remainder  22
reverse  27
round  23

scheme-report-environment  35
set!  10; 16, 41
set-car!  26
set-cdr!  26
setcar  42
simplest rational  23
sqrt  23
str ng  30
str ng->1 st  30
str ng->number  24
str ng->symbol  28
str ng-append  30
str ng-c <=?  30
str ng-c <=?  30
str ng-c =?  30
str ng-c >=?  30
str ng-c >=?  30
str ng-copy  30
str ng-f ll!  31
str ng-length  30; 20
str ng-ref  30
str ng-set!  30; 28
str ng<=?  30
str ng<=?  30
str ng>=?  30
str ng>=?  30
str ng?  30; 6
substr ng  30
symbol->str ng  28; 7
symbol?  28; 6
syntact c keyword  6; 5, 13, 38

syntax defin t on  17
syntax-rules  14; 17
#t  25
tail call  7
tan  23
token  38
top level enironment  17; 6
transcr pt-off  37
transcr pt-on  37
true  6; 10, 25
truncate  23
type  6
unbound  6; 8, 16
unquote  13; 26
unquote-spl c ng  13; 26
unspecified  4
valid indexes  30; 31
values  34; 9
variable  6; 5, 8, 38
vector  31
vector->l st  31
vector-f ll!  31
vector-length  31; 20
vector-ref  31
vector-set!  31
vector?  31; 6
whitespace  5
with-input-from-file  35
with-output-to-file  35
write  37; 13
write-char  37
#x  21; 39
zero?  22