

CONTENTS

2.3	Commands	17
2.3.1	Arguments Optional	18
2.3.2	Extended Commands	19
2.4	Trouble	19
2.5	Getting Help	20
2.6	Typing	21
2.7	Moving through a Buffer	22
2.8	Correcting Simple Mistakes	24
2.9	Undo	25
2.10	The Region	26
2.11	Kills and Yanks	27
2.12	Editing Scheme Programs	28
2.13	Scheme Interaction	30
2.14	Search	32
2.15	Replace	33
2.16	Buffers	35
2.17	Windows	37
2.18	Saving and Loading Files	38
2.19	Directory Edit	39
2.20	Printing	40
2.21	Miscellaneous Commands	40
3	Debugging	41
3.1	Subproblems and Reductions	43
3.2	Debug (The Continuation Browser)	45
3.3	Where (The Environment Browser)	49
3.4	Trace	50
3.5	Break	52
3.6	Error and Breakpoint Procedures	55
4	Scheme Reference	57
4.1	Notation and terminology	57
4.1.1	Notation for this Chapter	57
4.1.2	Evaluation	58
4.1.3	Numbers	58

CONTENTS

4.1.4	Identifiers	59
4.1.5	Whitespace and Comments	59
4.1.6	Special Characters in Scheme Notations	60
4.2	Variables and bindings	61
4.3	Expressions	61
4.3.1	Variable references	61
4.3.2	Literals	62
4.3.3	Procedure calls	63
4.3.4	Lambda expressions	63
4.3.5	Binding constructs	64
4.3.6	Assignments	65
4.3.7	Sequencing	66
4.4	Definitions	66
4.4.1	Top level definitions	67
4.4.2	Internal definitions	67
4.5	Booleans and conditionals	68
4.5.1	Boolean operations	68
4.5.2	Conditionals	70
4.5.3	Testing for equivalence	72
4.6	Pairs and lists	74
4.7	Symbols	81
4.8	Numbers	83
4.9	Strings	87
4.10	Vectors	88
4.11	Control features	89
4.12	Streams and delayed evaluation	90
4.13	Input and output	92
4.13.1	General Terminal and File I/O	92
4.13.2	Advanced Terminal and File I/O	94
4.14	Environments and evaluation	96
4.15	Miscellaneous	97
4.16	Advanced Features	98
4.17	Graphics	98

CONTENTS

A Glossary

101

Chapter 3

Debugging

Even computer software that has been planned carefully and written well may not always work correctly. Mysterious creatures called *bugs* may creep in and wreak havoc, leaving the programmer to clean up the mess. Some have theorized that a program fails only because its author made a mistake, but experienced computer programmers know that bugs are always to blame. This is why the task of fixing broken computer software is called *debugging*.

It is impossible to prove the correctness of any non-trivial program; hence the Cynic's First Law of Debugging:

Programs don't become more reliable as they are debugged; the bugs just get harder to find.

Scheme is equipped with a full complement of special software for finding and removing bugs. The debugging tools include facilities for tracing a program's use of specified procedures, for examining Scheme environments, and for setting *breakpoints*, places where the program will pause for inspection.

Many bugs are detected when programs try to do something which is impossible, like adding a number to a symbol, or using a variable which does not exist; this type of mistake is called an *error*. Whenever an error occurs, Edwin automatically describes the error in the Typein Window and asks whether to start the debugging tools. For example, using a nonexistent variable `foo` will cause Edwin to respond, "Unbound Variable F00 -- Debug?" Typing Y will start the debugging tools; N will return to Edwin.

Sometimes, a bug will never cause an error, but will still cause the program to operate incorrectly. For instance,

```
(prime? 7)            $\implies$  #f
```

In this situation, Scheme does not know that the program is misbehaving. The programmer must notice the problem and, if necessary, start the debugging tools manually.

There are several approaches to finding bugs in a Scheme program:

- Inspect the original Scheme program.
- Use the debugging tools described below to follow your program's progress.
- Edit the program to insert checks and breakpoints.

Only experience can teach how to debug programs, so be sure to experiment with all these approaches while doing your own debugging. Planning ahead is the best way to ward off bugs, but when bugs do appear, be prepared to attack them with all the tools available. Remember that the 6.001 Lab is equipped with some of the most powerful Scheme debugging tools available – 6.001 TAs. If you really get stuck, ask a TA for help.

3.1 Subproblems and Reductions

Understanding the concepts of *reduction* and *subproblem* is essential to good use of the debugging tools. The Scheme interpreter evaluates an expression by *reducing* it to a simpler expression. In general, Scheme's evaluation rules designate that evaluation proceeds from one expression to the next by either starting to work on a *subexpression* of the given expression, or by *reducing* the entire expression to a new (simpler, or reduced) form. Thus, a history of the successive forms processed during the evaluation of an expression will show a sequence of subproblems, where each subproblem may consist of a sequence of reductions.

For example, both `(+ 5 6)` and `(+ 7 9)` are subproblems of the following combination:

```
(* (+ 5 6) (+ 7 9))
```

If `(prime? n)` is true, then `(cons 'prime n)` is a reduction for the following expression:

```
(if (prime? n)
    (cons 'prime n)
    (cons 'not-prime n))
```

This is because the entire subproblem of the `if` combination can be *reduced* to the problem `(cons 'prime n)`, once we know that `(prime? n)` is true; the `(cons 'not-prime n)` can be ignored, because it will never be needed. On the other hand, if `(prime? n)` were false, then `(cons 'not-prime n)` would be the reduction for the `if` combination.

The *subproblem level* is a number representing how far back in the history of the current computation a particular evaluation is. Consider `factorial`:

```
(define (factorial n)
  (if (< n 2)
      1
      (* n (factorial (- n 1)))))
```

If we stop `factorial` in the middle of evaluating `(- n 1)`, the `(- n 1)` is at subproblem level 0. Following the history of the computation “upwards,” `(factorial (- n 1))` is at subproblem level 1, and `(* n (factorial (- n 1)))` is at subproblem level 2. These expressions all have *reduction number* 0. Continuing upwards, the `if` combination has reduction number 1.

Moving backwards in the history of a computation, subproblem levels and reduction numbers increase, starting from zero at the expression currently being evaluated. Reduction numbers increase until the next subproblem, where they start over at zero. The best way to get a feel for subproblem levels and reduction numbers is to experiment with the debugging tools, especially `Debug`.

3.2 Debug (The Continuation Browser)

Debug, otherwise known as the Continuation Browser, is the debugging tool which starts when an error occurs and you answer to Edwin's question, "Debug?" When Debug starts, it records all the information necessary to continue running the Scheme program which caused the error, and lets the programmer inspect this information – hence the name "Continuation Browser."

Debug runs in the Edwin buffer `*Debug`, which is automatically displayed in an Edwin window when Debug starts. A typical starting Debug display looks like this:

```
Subproblem Level: 0  Reduction Number: 0
Expression:
FOB
within the procedure FIB
applied to (10)
```

This tells us that the error occurred while trying to evaluate the expression `FOB` while running `(FIB 10)`. When it starts, Debug always displays the expression which caused the error; this expression is reduction number 0 at subproblem level 0.

Several keys have special functions when the cursor is in the **Debug* buffer:

B	Move to the next reduction.
D	Move down to the next subproblem.
E	Pop up a window (buffer <i>*Eval*</i> for evaluating Scheme expressions in the current environment.
F	Move to the previous reduction.
G	Move to a particular subproblem/reduction.
H	Pop up a menu of subproblems to be chosen among in buffer <i>*Subproblem-Menu</i> .
P	Display a pretty-printed version of the procedure which created the current environment in buffer <i>*Procedure*</i> .
Q	Leave the Continuation Browser, killing its buffer.
U	Move up to the previous subproblem.
V	Evaluate a single Scheme expression in the current environment, prompting for it in the Typein Window.
W	Browse the current environment in buffer <i>*Environment</i> (using Where, described below).

Now, imagine that you are debugging a Scheme program to compute Fibonacci numbers. You want procedure `fib` to accept an integer as its single parameter, returning the corresponding Fibonacci number. But when you run `(fib 10)`, Edwin displays the error message “Unbound Variable F00 -- Debug?” When you answer by pressing **Y**, Edwin starts Debug, which displays the following:

```
Subproblem Level: 0  Reduction Number: 0
Expression:
F0B
within the procedure FIB
applied to (10)
```

This tells you that the error occurred while trying to evaluate the expression `F0B` while running `(FIB 10)`. Now suppose this error confuses you,

because you don't recognize the variable `FOB` – after all, you were running your Fibonacci number program, which looks like this:

```
(define (fib n)
  (if (< n 2)
      1
      (+ (fib (- n 1))
         (fob (- n 2)))))
```

(It turns out that you made a typo, entering “fob” where you wanted “fib,” but you haven't noticed this yet.)

You wonder what Scheme was doing (evaluating) when it encountered `fob`, so you press `U`, telling Debug to move up one subproblem level. The `*Debug` buffer changes to:

```
Subproblem Level: 1 Reduction Number: 0
Expression:
(FOB (- N 2))
within the procedure FIB
applied to (10)
```

Still confused, you go up one more subproblem level by pressing `U` again:

```
Subproblem Level: 2 Reduction Number: 0
Expression:
(+ (FIB (- N 1)) (FOB (- N 2)))
within the procedure FIB
applied to (10)
```

But now, you notice that this code looks suspiciously like your Fibonacci procedure. Returning to the Edwin buffer where you defined it, you see your typo, correct it, and try `(FIB 10)` again. Now, the Fibonacci procedure works.

In this imaginary scenario, only the `U` command was needed to solve the problem, but let's try `H` to see a summary of evaluation history, anyway:

Sub Prb.	Procedure Name	Expression
0 0	FIB	FOB
1 0	FIB	(FOB (- N 2))
2 0	FIB	(+ (FIB (- N 1)) (FOB (- N 2)))
3		(WRITE-LINE (EVAL-WITH-HISTORY (WITH-INPUT-FROM
4 0		(WITH-OUTPUT-TO-CURRENT-POINT (LAMBDA () (WRITE
5 0	^G-INTERCEPTOR	(LET ((VALUE (CALL-WITH-CURRENT-CONTINUATION (L
6	COMMAND-READER-LOOP	(LET ((VALUE (CALL-WITH-CURRENT-CONTINUATION (L
7	COMMAND-READER-LOOP	(BEGIN (LET ((VALUE (CALL-WITH-CURRENT-CONTINUA

The first column in the history lists the subproblem level of each Expression, and the second column lists the reduction number. The Procedure Name column lists the procedure in which the Expression appears.

In this history, subproblems 0, 1, and 2 give the evaluation history of the `fib` procedure. In order to evaluate subproblem 2, Scheme had to evaluate subproblem 1, and in order to evaluate subproblem 1, Scheme had to evaluate subproblem 0, which caused the error. Subproblems 3 through 7 are parts of the history of the Scheme Read-Eval-Print (REP) loop, which is responsible for interpreting your typing, evaluating it, and displaying the results. While debugging your programs, you can ignore the subproblems for the Scheme REP loop.

3.3 Where (The Environment Browser)

Where, the Environment Browser, is the Scheme debugging tool for examining and changing variables in environments. Where is particularly useful for examining the state of a Scheme program when an error occurs, because it can display the current values of all variables in the program's environments.

Start Where from the Debug debugging tool by pressing **W** while the cursor is in the `*Debug` buffer.

The buffer in which Where starts shows the environment which was being examined in Debug, and is called `*Environment`. An `Environment-Browser` buffer is created for each environment being browsed with Where. All Where buffers display “(Environment-Browser)” on the mode line, indicating their major mode.

These keys have special functions when the cursor is in an `Environment-Browser` buffer:

- | | |
|----------|--|
| E | Pop up a window (buffer <code>*Eval*</code>) for evaluating Scheme expressions in the environment being browsed. |
| F | Browse the environment which is the value of the variable that the point is on. |
| P | Browse the parent environment of this environment, creating a new <code>Environment-Browser</code> buffer. |
| Q | Leave the current <code>Environment-Browser</code> buffer, killing it. |
| V | Evaluate a single expression in the current environment, prompting for it in the Typein Window. |
| W | Browse an environment, prompting for it in the typein window and creating a new <code>Environment-Browser</code> buffer. |

3.4 Trace

Trace is a family of debugging procedures for following the usage of particular procedures in a Scheme program.

<code>(trace-entry procedure)</code>	procedure
<code>(trace-exit procedure)</code>	procedure
<code>(trace-both procedure)</code>	procedure
<code>(trace procedure)</code>	procedure
<code>(untrace)</code>	procedure
<code>(untrace procedure)</code>	procedure

`Trace-entry` causes Scheme to display a message whenever *procedure* is entered. `Trace-exit` causes Scheme to display a message whenever *procedure* returns. `Trace-both` performs both `trace-entry` and `trace-exit`. `Trace` is the same as `trace-entry`.

`Untrace` removes any trace messages set for *procedure*. With no argument, it removes traces from all procedures.

```
==> (define (factorial n) (if (< n 2) 1 (* n (factorial (- n 1)))))
FACTORIAL
```

```
==> (trace-entry factorial)
```

```
==> (trace-exit factorial)
```

```
==> (factorial 4)
[Entering <#[COMPOUND-PROCEDURE FACTORIAL] 3>]
[Entering <#[COMPOUND-PROCEDURE FACTORIAL] 2>]
[Entering <#[COMPOUND-PROCEDURE FACTORIAL] 1>]
6
```

```
==> (untrace factorial)
```

```
==> (trace-exit factorial)
```

```
==> (factorial 4)
[1 <== <#[COMPOUND-PROCEDURE FACTORIAL] 1>]
[2 <== <#[COMPOUND-PROCEDURE FACTORIAL] 2>]
[6 <== <#[COMPOUND-PROCEDURE FACTORIAL] 3>]
[24 <== <#[COMPOUND-PROCEDURE FACTORIAL] 4>]
24

==> (untrace factorial)

==> (factorial 5)
120

==> (trace-both factorial)

==> (factorial 2)
[Entering <#[COMPOUND-PROCEDURE FACTORIAL] 2>]
[Entering <#[COMPOUND-PROCEDURE FACTORIAL] 1>]
[1 <== <#[COMPOUND-PROCEDURE FACTORIAL] 1>]
[2 <== <#[COMPOUND-PROCEDURE FACTORIAL] 2>]
2

==> _
```

3.5 Break

Break is a family of debugging procedures for causing particular procedures in a Scheme program to stop for inspection whenever they are used.

Unfortunately, in this release of the 6.001 course software, Break does not work under Edwin; it pays no attention to the careful organization of the Edwin screen, but haphazardly writes its own information over Edwin's. Therefore, before using Break on any procedures, it is recommended that you leave Edwin, entering the "outside" Scheme REP loop; do this by typing `CTRL-X Z`.

The outside REP loop operates similarly to the Edwin Interaction-mode buffer, except that no Edwin commands work; use `BACK SPACE` to correct your typing, and `ENTER` to send your typing to Scheme. The grey `EDIT` key will return you safely to Edwin. We recommend that you stay within Edwin at all times, except when you want to use Break.

Both Debug and Where work in the outside REP loop, but not exactly the same as in Edwin. However, once you learn how to use them in Edwin, the differences will be easy to overcome. To start Debug, type `(debug)` at Scheme.

When the outside Scheme detects an error, it displays a message describing the error, and switches the prompt from the standard `==>` to `2 Error->`, but continues as if nothing had happened. At this point, either type `CTRL-U` to return to the standard prompt, or `(debug)` to start Debug.

If you start Debug, the prompt will change to `3 Debug-->`, and Debug will wait for you to type a one-key command, just like the Debug in Edwin. To see a list of the possible commands, press `?`. Type `CTRL-U` to stop the Debug, returning to the `2 Error->` prompt.

The `W` key in Debug starts Where, whose prompt is `4 Where-->`. Where accepts one-key commands, as well; press `?` to list them. From Where, press `CTRL-U` to stop Where, returning to Debug.

Here is a simple demonstration of the outside REP loop and its debugging tools:

```

==> foo
Unbound Variable F00

2 Error-> (debug)
Subproblem Level: 0  Reduction Number: 0
Expression:
F00
within a MAKE-ENVIRONMENT special form
applied to ()
Debugger

3 Debug--> w
Environment Inspector

4 Where--> c
Frame created by a MAKE-ENVIRONMENT special form
Depth (relative to starting frame): 0
Has bindings:

FACTORIAL = #[COMPOUND-PROCEDURE FACTORIAL]
FIB = #[COMPOUND-PROCEDURE FIB]

4 Where--> _

```

(break-entry <i>procedure</i>)	procedure
(break-exit <i>procedure</i>)	procedure
(break-both <i>procedure</i>)	procedure
(break <i>procedure</i>)	procedure
(unbreak <i>procedure</i>)	procedure

Break-entry causes Scheme to enter a breakpoint whenever *procedure* is entered. **Break-exit** enters a breakpoint whenever *procedure* returns. **Break-both** performs both **break-entry** and **break-exit**. **Break** is the same as **break-entry**. **Unbreak** removes breakpoints from *procedure*; with no parameter, it removes breakpoints from all procedures.

When a Scheme procedure with a breakpoint is used, it runs normally, except that Scheme creates a simple REP loop when the breakpoint is encountered (either on entering the procedure, or leaving it, or both, depending on how the breakpoint was set). When the REP loop starts, Scheme displays a message which includes the actual parameters to the procedure and, for `break-exit` and `break-both`, the value the procedure is returning.

The prompt for the REP loop is `2 Bkpt`. At this prompt, any normal Scheme expression is acceptable, including the special expression `(proceed)`, which tells Scheme to continue running the procedure as if the breakpoint had never happened.

Here is a simple demonstration of the use of breakpoints in the outside REP loop:

```
==> (define (factorial n) (if (< n 2) 1 (* n (factorial (- n 1)))))
FACTORIAL

==> (break-entry factorial)

==> (factorial 3)
[Entering <#COMPOUND-PROCEDURE FACTORIAL] 3>
Breakpoint on entry

2 Bkpt-> (proceed)
[Entering <#COMPOUND-PROCEDURE FACTORIAL] 2>
Breakpoint on entry

2 Bkpt-> (proceed)
[Entering <#COMPOUND-PROCEDURE FACTORIAL] 1>
Breakpoint on entry

2 Bkpt-> (proceed)
6

==> (unbreak factorial)

==> (break-exit factorial)

==> (factorial 2)
```

```
[1 <== <#[COMPOUND-PROCEDURE FACTORIAL] 1>]
Breakpoint on exit

2 Bkpt-> n
1

2 Bkpt-> (proceed)
[2 <== <#[COMPOUND-PROCEDURE FACTORIAL] 2>]
Breakpoint on exit

2 Bkpt-> (proceed)
6

==> _
```

3.6 Error and Breakpoint Procedures

The procedures Scheme uses for signalling errors and breakpoints are available for use in Scheme programs.

`(bkpt message object)` syntax

Sets a breakpoint. Displays *message* and *object* and enters a breakpoint read-eval-print loop in the current environment.

`(breakpoint-procedure message irritant environment)` procedure

Invokes a breakpoint, displaying string *message* followed by *irritant*, giving the debugger *environment*. Used by the special form `bkpt`.

`(error exp1 exp2 ...)` syntax

Signals an error. Displays all the expressions and creates a debugging breakpoint.

`(error-procedure message irritant environment)` procedure

Invokes an error, displaying string *message* followed by *irritant*, giving the debugger *environment*. Used by the special form `error`.

Chapter 4

Scheme Reference

This chapter is a reference manual for the Scheme programming language. Don't read this chapter to learn Scheme, but turn to it when you need more detail about the language than appears in *Structure and Interpretation of Computer Programs* [1].

4.1 Notation and terminology

This section describes the notation used in this chapter. It also discusses the lexical conventions for writing Scheme programs.

4.1.1 Notation for this Chapter

In this chapter, in text describing a procedure or special form, the names of all formal parameters are *italicized*. The names of Scheme procedures, special forms, and variables appear in **typewriter-style type**. Throughout the chapter, whenever a formal parameter has one of the names listed in the left column of the table below, it must be of the type listed in the right column. For example, a formal parameter referred to as n_2 in this chapter must be an integer.

<i>obj</i>	any object
$x, x_1, \dots, x_j, \dots$	real number
$y, y_1, \dots, y_j, \dots$	real number
$n, n_1, \dots, n_j, \dots$	integer
$k, k_1, \dots, k_j, \dots$	nonnegative integer

4.1.2 Evaluation

The symbol “ \implies ” in these examples means “evaluates to.” For example,

`(* 5 8)` \implies 40

means that the expression `(* 5 8)` evaluates to 40.

Where the value returned by an expression is marked *unspecified*, Scheme does not guarantee what the value will be. For example,

`(set! x 3)` \implies *unspecified*

Feel free to experiment to determine what Scheme returns in any particular case, but remember that a new version of Scheme might return a different value. Avoid writing programs that rely on unspecified values, particularly when working on problem sets.

4.1.3 Numbers

Scheme numbers are expressed exactly as normal numbers, except that scientific notation looks slightly different. In Scheme scientific notation, the numbers of the exponent are not raised, but instead are separated from the base number by the letter **e**. For example,

123e10 is Scheme notation for 123×10^{10} , and
 -123.45e-1 is Scheme notation for -123.45×10^{-1} .

4.1.4 Identifiers

A Scheme identifier is a sequence of letters, digits, and punctuation characters that is not a number. Here are some identifiers:

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

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 `F00`.

Identifiers have several uses within Scheme programs:

- Certain identifiers are reserved for use as special forms, and cannot be used as variables. They are:

<code>and</code>	<code>else</code>	<code>or</code>
<code>begin</code>	<code>error</code>	<code>quote</code>
<code>cond</code>	<code>if</code>	<code>sequence</code>
<code>cons-stream</code>	<code>lambda</code>	<code>set!</code>
<code>define</code>	<code>let</code>	<code>the-environment</code>
<code>delay</code>	<code>make-environment</code>	<code>unquote</code>

- Any identifier that does not refer to a special form may be used as a variable (see section 4.2).
- When an identifier appears as a quoted constant or within a quoted constant, it denotes a *symbol* (see section 4.7).

4.1.5 Whitespace and Comments

Whitespace characters are spaces and line breaks (*newlines*). Whitespace is used to improve readability and to separate *tokens* (identifiers and numbers) from each other. Whitespace may also occur inside a string value.

A semicolon starts a *comment*, text which will be seen as whitespace by Scheme. The comment continues from the semicolon to the end of the line. For example:

```

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

```

4.1.6 Special Characters in Scheme Notations

Here are the special characters used in Scheme notations:

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

These characters may be used in identifiers as if they were letters.

. + -

These appear in numbers and identifiers. The period character is also used in the notation for pairs (section 4.6).

()

Parentheses surround expressions and lists (section 4.6).

,

The single quote character indicates literal data (section 4.3.2).

"

Double quote characters surround string values (section 4.9).

\

Backslash is an escape character within string constants (section 4.9).

[] { }

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

#t #f

These are the boolean constants true and false (section 4.5).

#\

Represents the ASCII standard number representing the immediately following character.

#(

Introduces a vector constant (section 4.10).

4.2 Variables and bindings

Any identifier that does not refer to a special form (see section 4.1.4) may be used as a variable. A variable may name a location where a value can be stored. A variable that does so is said to be *bound* to the location. The set of all bindings in effect at some point in a program is the *environment*.

Certain expressions create new locations and bind variables to them. The most fundamental of these *binding constructs* is the lambda expression, because all other binding constructs can be explained in terms of lambda expressions. See section 4.3.5.

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

4.3 Expressions

A Scheme expression is a piece of code that returns a value. Types of expressions include variable references, literals, procedure calls, and special forms (see section 4.1.4). This section describes all Scheme expression types except conditionals and logical operations, which are described in section 4.5.

4.3.1 Variable references

variable

syntax

An expression consisting of a variable (section 4.2) is a variable reference. The value of the variable reference is the value stored in the location to which

the variable is bound. It is an error to reference an unbound variable.

```
(define x 28)
x            $\implies$  28
```

4.3.2 Literals

```
(quote datum)           syntax
' datum                 syntax
constant                syntax
```

`(quote datum)` evaluates to *datum*. *Datum* may be any printed representation of a Scheme object. This notation is used to include literal constants in Scheme code.

```
(quote a)                 $\implies$  a
(quote #(a b c))          $\implies$  #(a b c)
(quote (+ 1 2))           $\implies$  (+ 1 2)
```

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

```
' a                       $\implies$  a
'#(a b c)                  $\implies$  #(a b c)
'+ 1 2)                    $\implies$  (+ 1 2)
'(quote a)                  $\implies$  (quote a)
'' a                        $\implies$  (quote a)
```

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

```
'"abc"                    $\implies$  "abc"
"abc"                      $\implies$  "abc"
'145932                    $\implies$  145932
145932                     $\implies$  145932
'#t                        $\implies$  #t
#t                          $\implies$  #t
```

4.3.3 Procedure calls

`(operator operand1 ...)` syntax

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

```
(+ 3 4)            $\implies$  7
((if #f + *) 3 4)  $\implies$  12
```

Procedure calls are also called *combinations*.

4.3.4 Lambda expressions

`(lambda formals body)` syntax

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

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

```
(lambda (x) (+ x x))    $\implies$  a procedure
((lambda (x) (+ x x)) 4)  $\implies$  8
```

```
(define reverse-subtract
  (lambda (x y) (- y x)))
(reverse-subtract 7 10)  $\implies$  3
```

```
(define foo
  (let ((x 4))
    (lambda (y) (+ x y))))
(foo 6)  $\implies$  10
```

Formals should have one of the following forms:

- $(variable_1 \dots)$: The procedure takes a fixed number of arguments; when the procedure is called, the arguments will be stored in the bindings of the corresponding variables.
- *variable*: The procedure takes any number of arguments; when the procedure is called, the sequence of actual arguments is converted into a newly allocated list, and the list is stored in the binding of the *variable*.
- $(variable_1 \dots variable_{n-1} . variable_n)$: If a period surrounded by spaces precedes the last variable, then the value stored in the binding of the last variable will be a list of the actual arguments left over after all the other actual arguments have been matched up against the formal arguments.

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

4.3.5 Binding constructs

The binding construct `let` gives Scheme a block structure, like Algol 60. The syntax of the three constructs is identical, but they differ in the regions they establish for their variable bindings. In a `let` expression, the initial values are computed before any of the variables become bound.

`(let bindings body)` syntax

Syntax: *Bindings* should have the form

```
((variable1 init1) ...),
```

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

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

```
(let ((x 2) (y 3))
  (* x y))            $\implies$  6
```

```
(let ((x 2) (y 3))
  (let ((foo (lambda (z) (+ x y z)))
        (x 7))
    (foo 4)))        $\implies$  9
```

(let *variable bindings body*) syntax

This is an extension of the syntax of `let`, called “named `let`” which provides a general looping construct and which may also be used to express recursions.

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

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

4.3.6 Assignments

(set! *variable expression*) syntax

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

```

(define x 2)            $\implies$  unspecified
(+ x 1)                $\implies$  3
(set! x 4)             $\implies$  unspecified
(+ x 1)               $\implies$  5

```

4.3.7 Sequencing

(sequence *expression*₁ *expression*₂ ...) syntax

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

```

(sequence (set! x 5)
  (+ x 1))            $\implies$  6

(sequence (princ "4 plus 1 equals ")
  (princ (+ 4 1)))   $\implies$  unspecified
                  and prints 4 plus 1 equals 5

```

Note: Sequence is a synonym for the `begin` sequencing construct of the Scheme standard.

4.4 Definitions

Definitions are used to establish bindings. Definitions are not valid in 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 definition should have one of the following forms:

- (define *variable expression*)
- (define (*variable formals*) *body*)
Formals should be either a sequence of zero or more variables, or a sequence of one or more variables followed by a period surrounded by spaces and another variable (as in a lambda expression). This form is equivalent to

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

- (define (variable . formal) body)
Formal should be a single variable. This form is equivalent to

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

4.4.1 Top level definitions

At the top level of a program, a definition

```
(define variable expression)
```

has essentially the same effect as the assignment expression

```
(set! variable expression)
```

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

```
(define add3
  (lambda (x) (+ x 3)))
(add3 3)                 $\implies$  6
(define first car)
(first '(1 2))           $\implies$  1
```

4.4.2 Internal definitions

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

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

4.5 Booleans and conditionals

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

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

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

<code>#t</code>	\Rightarrow	<code>#t</code>
<code>#f</code>	\Rightarrow	<code>#f</code>
<code>'#f</code>	\Rightarrow	<code>#f</code>

<code>true</code>	variable
<code>false</code>	variable

The variables `true` and `false` are initially bound to the values `#t` and `#f` respectively. Note that these are ordinary variables, whereas `true` and `false` are constants.

<code>true</code>	\Rightarrow	<code>#t</code>
<code>false</code>	\Rightarrow	<code>#f</code>
<code>'false</code>	\Rightarrow	<code>false</code>

4.5.1 Boolean operations

<code>(not obj)</code>	procedure
------------------------	-----------

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

```

(not #t)            $\implies$  #f
(not 3)            $\implies$  #f
(not (list 3))     $\implies$  #f
(not #f)           $\implies$  #t
(not '())          $\implies$  #t
(not (list))       $\implies$  #t

```

`(and test1 ...)` syntax

The *test* expressions are all evaluated. And returns `#f` if any of its arguments is false, and `#t` otherwise.

```

(and (< 3 4) (> 6 5))  $\implies$  #t
(and (> 3 4) (> 6 5))  $\implies$  #f
(and)  $\implies$  #t ; by convention

```

`(or test1 ...)` syntax

The *test* expressions are all evaluated. Or returns `#t` if any of its arguments is true, and `#f` otherwise.

```

(or (eq? 'x 'y) (eq? 'z 'y))  $\implies$  #f
(or (> x 0) (= x 0) (< x 0))  $\implies$  #t
(or)  $\implies$  #f ; by convention

```

`(conjunction test1 ...)` syntax

The *test* expressions are evaluated from left to right, and the value of the first expression that evaluates to a false value 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.

```

(conjunction (= 2 2) (> 2 1))
  ⇒ #t
(conjunction (= 2 2) (< 2 1))
  ⇒ #f
(conjunction 1 2 'c '(f g))
  ⇒ (f g)
(conjunction)
  ⇒ #t

```

`(disjunction test1 ...)` syntax

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

```

(disjunction (= 2 2) (> 2 1))
  ⇒ #t
(disjunction (= 2 2) (< 2 1))
  ⇒ #t
(disjunction #f #f #f)
  ⇒ #f
(disjunction (memq 'b '(a b c))
  (/ 3 0))
  ⇒ (b c)
(disjunction)
  ⇒ #f

```

4.5.2 Conditionals

`(if test consequent alternate)` syntax

`(if test consequent)` syntax

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

Semantics: An if expression is evaluated as follows: first, *test* is evaluated. If it yields a true value (see section 4.5), then *consequent* is evaluated

and its value is returned. Otherwise *alternate* is evaluated and its value is returned. If *test* yields a false value and no *alternate* is specified, then the result of the expression is unspecified.

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

(*cond clause₁ clause₂ ...*) syntax

Syntax: Each *clause* should be of the form

```
(test expression ...)
```

where *test* is any expression. The last *clause* may be an “else clause,” which has the form

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

Semantics: A *cond* expression is evaluated by evaluating the *test* expressions of successive *clauses* in order until one of them evaluates to a true value (see section 4.5). When a *test* evaluates to a true value, then the remaining *expressions* in its *clause* are evaluated in order, and the result of the last *expression* in the *clause* is returned as the result of the entire *cond* expression. If the selected *clause* contains only the *test* and no *expressions*, then the value of the *test* is returned as the result. If all *tests* evaluate to false values, and there is no else clause, then the result of the conditional expression is unspecified; if there is an else clause, then its *expressions* are evaluated, and the value of the last one is returned.

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

4.5.3 Testing for equivalence

The three predicates described in this section are used to test whether objects are “the same”. Of the three predicates, `eq?` is the finest or most discriminating while `equal?` is the coarsest. `eqv?` is slightly less discriminating than `eq?`. In general, you should use `equal?` to test the equivalence of structures and `eqv?` to test the equivalence of atomic objects. See [2] for a careful description of the rationale for these three different predicates.

`(eq? obj1 obj2)` procedure

`Eq?` is the primitive equivalence predicate. It returns true if `obj1` and `obj2` are the same object. Unless you are rather advanced, do not use `eq?` to test identity of objects other than symbols. In particular, two numbers which are equal in value may not be `eq?`; use `=` or `eqv?` to test numerical equality. Also, two lists which have the same elements and print the same may not be `eq?`; use `equal?` to test if lists have the same elements.

For more advanced users only: Two structures are `eq?` if any attempt to modify one of them modifies the other in the same way; i.e., if the two structures share the same location in memory (see chapter 3 of the 6.001 text).

<code>(define x '(a b))</code>	\implies	X
<code>(define y '(a b))</code>	\implies	Y
<code>(define z x)</code>	\implies	Z
<code>(eq? x y)</code>	\implies	#f
<code>(eq? x z)</code>	\implies	#t
<code>(eq? y z)</code>	\implies	#f
<code>(set! x 'foo)</code>	\implies	<i>unspecified</i>
<code>(set! y 'foo)</code>	\implies	<i>unspecified</i>
<code>(set! z x)</code>	\implies	<i>unspecified</i>
<code>(eq? x y)</code>	\implies	#t
<code>(eq? x z)</code>	\implies	#t
<code>(eq? y z)</code>	\implies	#t

`(eqv? obj1 obj2)`

procedure

`Eqv?` returns `#t` if obj_1 and obj_2 are: `eq?` atoms, = numbers, or character strings containing the same characters in the same order with the same capitalization. Two lists are not in general `eqv?` even if `eq?` atom by atom. `Eqv?` is usually the correct predicate for testing the equality of atoms.

```
(eqv? "abc" "abc")           => #t
(eqv? '(a b c) '(a b c))    => #f
(eqv? 67000 67000)         => #t

(let ((a '(1 2 3)))
  (let ((b (cons a a)))
    (eqv? (car b) (cdr b)))) => #t

(eqv? (lambda () 1)
      (lambda () 1))         => #f

(let ((p (lambda () 1)))
  (eqv? p p))                => #t
```

`(equal? obj_1 obj_2)` procedure

`Equal?` returns `#t` if both arguments are `eqv?` atoms, numbers, or character strings. `Equal?` list structures are those whose `CARs` are `equal?` and whose `CDRs` are `equal?`. As illustrated below, `eq?` and `equal?` are quite different.

```
(eq? '(a b) '(a b))         => #f
(equal? '(a b) '(a b))     => #t
(define x '(a b))           => X
(define y '(a b))           => Y
(equal? x y)                 => #t
(equal? 3 3)                 => #t
(equal? 67000 67000)       => #t
(equal? "abc" "abc")       => #t
```

4.6 Pairs and lists

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

Pairs are used primarily to represent lists. A *list* is either empty or a pair whose *car* is the first element of the list, and whose *cdr* is the rest of the list. The length of a list is the number of elements, which is the same as the number of pairs.

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

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

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

`(a b c d e)`

and

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

are both representations of a list of symbols.

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

`(a b c . d)`

is equivalent to

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

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

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

`(atom? obj)` procedure

Returns `#t` if `obj` is an atom, and `#f` otherwise. An atom is anything which cannot be further decomposed via `car` or `cdr`. In other words, anything that is not a pair is an atom.

```
(atom? 3)        ⇒ #t
(atom? 'car)     ⇒ #t
(atom? car)      ⇒ #t
(atom? '(a 3))   ⇒ #f
(atom? '())      ⇒ #t
(atom? nil)      ⇒ #t
```

`(list? obj)` procedure

Returns `#t` if `obj` is a list, and `#f` otherwise.

```
(list? 'a)       ⇒ #f
(list? '())      ⇒ #t
(list? '(a b))   ⇒ #t
(list? '(a . b)) ⇒ #f
```

`(pair? obj)` procedure

Returns `#t` if `obj` is a pair, and `#f` otherwise. Note that the empty list, `'()`, is a list but not a pair.

```

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

```

`(cons obj1 obj2)`

 procedure

Returns a newly allocated pair whose car is *obj*₁ and whose cdr is *obj*₂. The pair is guaranteed to be different (in the sense of `eqv?`) from every existing object.

```

(cons 'a '())        ⇒ (a)
(cons '(a) '(b c d)) ⇒ ((a) b c d)
(cons "a" '(b c))   ⇒ ("a" b c)
(cons 'a 3)         ⇒ (a . 3)
(cons '(a b) 'c)    ⇒ ((a b) . c)

```

`(car pair)` procedure

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

```

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

```

`(cdr pair)` procedure

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

```

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

```

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

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

(caar *pair*) procedure
 (cadr *pair*) procedure
 :
 (caddr *pair*) procedure
 (caddr *pair*) procedure

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

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

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

(first *pair*) procedure
 (second *pair*) procedure
 (third *pair*) procedure
 (fourth *pair*) procedure
 (fifth *pair*) procedure
 (sixth *pair*) procedure
 (seventh *pair*) procedure
 (eighth *pair*) procedure

These procedures return the first, second, third, ..., eighth element of *pair*, respectively, in exactly the same manner as would car, cadr, caddr, caddr,

```
(first '((a b) c d))    => (a b)
(second '((a b) c d))  => c
(eighth '(a b c d e f g h)) => h
```

(null? *obj*) procedure

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

(list *obj*₁ ...) procedure

Returns a list of its arguments.

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

(list* *obj*₁ *obj*₂ ...)

List* conses together an arbitrary number of arguments. It constructs a pair whose car is *obj*₁ and whose cdr is (list* *obj*₂ ...). When it has only two arguments, list* behaves in the same way as cons.

```
(list* 'a 'b 'c)       => (a b . c)
(list* '(a b) 'c '(d e)) => ((a b) c d e)
(list* 'a 'b)          => (a . b)
(list* 'a)              => a
```

(length *list*) procedure

Returns the length of *list*.

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

(append *list*₁ *list*₂) procedure

(append *list* ...) procedure

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

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

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

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

`(reverse list)` procedure
Returns a newly allocated list consisting of the elements of *list* in reverse order.

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

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

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

For historical reasons, `(nthcdr k list)` is the same as `(list-tail list k)`.

```
(list-tail '(a b c d) 2)  ⇒ (c d)
(nthcdr 2 '(a b c d))   ⇒ (c d)
```

`(last list)` procedure
Returns the last pair of *list*.


```

(define e '((a 1) (b 2) (c 3)))
(assq 'a e)           ⇒ (a 1)
(assq 'b e)           ⇒ (b 2)
(assq 'd e)           ⇒ #f
(assq (list 'a) '(((a)) ((b)) ((c))))
                    ⇒ #f
(assq 5 '((2 3) (5 7) (11 13)))
                    ⇒ unspecified
(assv 5 '((2 3) (5 7) (11 13)))
                    ⇒ (5 7)
(assoc (list 'a) '(((a)) ((b)) ((c))))
                    ⇒ ((a))

```

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

4.7 Symbols

Symbols are objects whose usefulness rests on the fact that two symbols are identical (in the sense of `eqv?`) if and only if their names are spelled the same way. The rules for writing a symbol are exactly the same as the rules for writing an identifier; see section 4.1.4.

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

```

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

```

`(alphaless? symbol1 symbol2)` procedure
 Returns `#t` or `#f` according to whether the name of *symbol*₁ alphabetically precedes the name of *symbol*₂.

```
(alphaless? 'ab 'b)    ⇒    #t
(alphaless? 'a 'a)    ⇒    #f
```

(explode *symbol*) procedure

Returns a list of one-character symbols or digits that are the letters in the name of *symbol*.

```
(explode 'gleep35)    ⇒    (G L E E P 3 5)
```

(implode *list*) procedure

The argument must be a list of character symbols or integers. **Implode** returns a symbol whose name is the concatenation of the names of elements in the list.

```
(implode '(g l e e p 3 5)) ⇒    GLEEP35
```

(char *n*) procedure

Returns the ASCII character (symbol) whose ASCII code is *n*.

```
(char 65)            ⇒    A
```

(ascii *symbol*) procedure

Returns the ASCII code of the single-character symbol *symbol*. All alphabetic characters are regarded as upper case, because symbols do not retain the case in which they are typed.

```
(ascii 'a)          ⇒    65
```

`(generate-uninterned-symbol symbol)` procedure

Returns a symbol that is guaranteed to be different (not `eqv?`) from any other symbol. The printed representation of the new symbol will begin with the characters of *symbol*.

Note: The “name” of an uninterned symbol is convenience for the human reader. You cannot reference an uninterned symbol by quoting its name.

```
(define s (generate-uninterned-symbol 'fact))
                                     ⇒ unspecified
s                                     ⇒ something of the form FACT342
(eqv? s s)                             ⇒ #t
(eqv? s 'FACT342)                       ⇒ #f
```

4.8 Numbers

Section 4.1.1 lists the names used here to specify restriction on the types of arguments these numerical procedures will accept.

`(number? obj)` procedure

`(integer? obj)` procedure

These numerical type predicates can be applied to any kind of argument, including non-numbers. They return true if the object is of the named type. In general, if a type predicate is true of a number then all higher type predicates are also true of that number.

`(zero? z)` procedure

`(positive? x)` procedure

`(negative? x)` procedure

`(odd? n)` procedure

`(even? n)` procedure

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

`(= z1 z2)` procedure

`(< x1 x2)` procedure

$(> x_1 x_2)$ procedure
 $(<= x_1 x_2)$ procedure
 $(>= x_1 x_2)$ procedure

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

$(\max x_1 x_2)$ procedure
 $(\max x_1 x_2 \dots)$ procedure
 $(\min x_1 x_2)$ procedure
 $(\min x_1 x_2 \dots)$ procedure

These procedures return the maximum or minimum of their arguments.

$(+ z_1 z_2)$ procedure
 $(+ z_1 \dots)$ procedure
 $(* z_1 z_2)$ procedure
 $(* z_1 \dots)$ procedure

These procedures return the sum or product of their arguments.

$(+ 3 4) \implies 7$
 $(+ 3) \implies 3$
 $(+) \implies 0$
 $(* 4) \implies 4$
 $(*) \implies 1$

$(- z_1 z_2)$ procedure
 $(- z_1 z_2 \dots)$ procedure
 $(/ z_1 z_2)$ procedure
 $(/ z_1 z_2 \dots)$ procedure

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

$(- 3 4) \implies -1$
 $(- 3 4 5) \implies -6$
 $(- 3) \implies -3$
 $(/ 3 4 5) \implies 3/20$
 $(/ 3) \implies 1/3$

`(1+ z)` procedure

This procedure returns a value one greater than its argument.

`(1+ 3)` \implies 4

`(-1+ z)` procedure

This procedure returns a value one less than its argument.

`(-1+ 3)` \implies 2

`(abs z)` procedure

Abs returns the magnitude of its argument.

`(abs -7)` \implies 7

`(abs -3+4i)` \implies 5

`(quotient n_1 n_2)` procedure

`(remainder n_1 n_2)` procedure

`(integer-divide n_1 n_2)` procedure

These are intended to implement number-theoretic (integer) division: For positive integers n_1 and n_2 , if n_3 and n_4 are integers such that $n_1 = n_2n_3 + n_4$ and $0 \leq n_4 < n_2$, then

`(quotient n_1 n_2)` \implies n_3

`(remainder n_1 n_2)` \implies n_4

`(integer-divide n_1 n_2)` \implies (n_3 . n_4)

`integer-divide` returns the cons of what `quotient` and `remainder` would return given the same arguments.

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

`(= n_1 (+ (* n_2 (quotient n_1 n_2))`

`(remainder n_1 n_2)))`

\implies #t

The value returned by `quotient` always has the sign of the product of its arguments. `Remainder` always has the sign of the dividend.

```
(remainder 13 4)       $\implies$  1
(remainder -13 4)     $\implies$  -1
(remainder 13 -4)     $\implies$  1
(remainder -13 -4)    $\implies$  -1
```

```
(gcd  $n_1$  ...) procedure
```

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

```
(gcd 32 -36)           $\implies$  4
(gcd)                  $\implies$  0
```

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

These procedures create integers and rationals.

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

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

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

`Log` computes the natural logarithm of z (not the base 10 logarithm). `Atan` returns the arctangent, in radians, of the quotient of its arguments.

For nonzero x , the value of `log` x is defined to be the one whose imaginary part lies in the range $-\pi$ (exclusive) to π (inclusive). (`Log 0`) is undefined.

`(sqrt z)` procedure
Returns the square root of z .

`(expt z1 z2)` procedure
Returns z_1 raised to the power z_2 :

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

0^0 is defined to be equal to 1.

`(random k)` procedure
Returns an integer chosen at random between 0 and $k - 1$, inclusive. The argument k must be a positive integer.

4.9 Strings

Strings are sequences of characters. Strings are written as sequences of characters enclosed within double quotes (`"`). A double quote inside a string is created by `\`, as in

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

To type `\` on the Chipmunk keyboard, hold down `SHIFT` and press the `)` on the numeric keypad on the right side of the keyboard; `SHIFT` with `0` (zero) will *not* work.

4.10 Vectors

Vectors are structures whose elements are indexed by integers. The first element in a vector is indexed by zero, and the last element is indexed by one less than the length of the vector. The printed representation for vectors is a `#` followed by the sequence of elements, enclosed in parentheses. 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 2) "Anna")
```

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

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

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

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

```
(vector-cons size fill)      procedure
Vector-cons returns a newly allocated vector of size elements, each initialized to fill.
```

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

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

```
(vector-size vector)        procedure
Returns the number of elements in vector.
```

`(vector-ref vector k)` procedure
k must be a nonnegative integer less than `(vector-length vector)`. `Vector-ref` returns the contents of element *k* of *vector*.

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

`(vector-set! vector k obj)` procedure
k must be a nonnegative integer less than `(vector-length vector)`. `Vector-set!` stores *obj* in element *k* of *vector*. The value returned by `vector-set!` is unspecified.

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

4.11 Control features

This chapter describes various primitive procedures that control the flow of program execution in special ways.

`(apply proc args)` procedure
Proc must be a procedure and *args* must be a list. `Apply` calls *proc* with the elements of *args* as the actual arguments.

```
(apply + (list 3 4)) ⇒ 7
```

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

```
((compose sqrt *) 12 75) ⇒ 30
```

`(mapcar proc list1 list2 ...)` procedure
Proc must be a procedure of which takes as many arguments as there are lists supplied to `mapcar`. `Mapcar` returns a list whose first element is `proc` applied to the first element of each list, whose second element is `proc` applied to the second element of the list, and so on. `Mapcar` stops when it reaches the end of the shortest list.

```
(mapcar length '((a b) (c d) (e (f g) h)))
      ⇒ (2 2 3)
(mapcar + '(1 2 3) '(4 5 6))
      ⇒ (5 7 9)
(mapcar * '(1 2) '(3 4 5) '(6 7 8 9))
      ⇒ (18 56)
```

`(mapc proc list1 list2 ...)` procedure
Proc must be a procedure which takes as many arguments as there are lists supplied to `mapc`. `Mapc` applied `proc` to the first element of each list, to the second element of each list, and so on. The difference between `mapc` and `mapcar` is that `mapcar` returns a list of values, while `mapc` does not. The procedure `proc` is usually applied for effect, and the value returned by `mapc` should be ignored.

4.12 Streams and delayed evaluation

A *stream* is a structure with two fields called the head and the tail. Streams are created by `cons-stream`. The head and tail are accessed using the procedures `head` and `tail`. Streams are used for representing sequences of elements, which are retrieved as the heads of successive tails of the stream. In this way, a stream is like a list. The difference is that the items in the stream are *delayed*, that is, they are computed when they are accessed rather than when the stream is constructed. The empty stream is `stream` that has no elements. See chapter 3 of [1] for information on programming with streams.

`(cons-stream obj1 obj2)` syntax

Returns a stream whose head is *obj*₁ and whose tail is *obj*₂. `Cons-stream` is syntax rather than a procedure because it does not evaluate its second argument.

`(head stream)` procedure

Returns the head of the given *stream*.

`(tail stream)` procedure

Returns the tail of the given *stream*.

`the-empty-stream` variable

A variable initially bound to the empty stream.

`(empty-stream? obj)` procedure

Returns `#t` or `#f` depending on whether *obj* is the empty stream.

`(delay expression)` syntax

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

`(force promise)` procedure

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

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

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

```
(define count 0)
(define p (delay (sequence (set! count (+ count 1))
  (* x 3))))
(define x 5)
count            $\implies$  0
p               $\implies$  a promise
(force p)       $\implies$  15
p               $\implies$  a promise, still
count          $\implies$  1
(force p)      $\implies$  15
count         $\implies$  1
```

4.13 Input and output

4.13.1 General Terminal and File I/O

Several of the procedures described in this section accept optional arguments. The optional argument *channel* is a channel objects, as returned by `open-reader-channel` or `open-printer-channel`. If the argument *channel* is omitted, the channel is defaulted to the console. For input operations, the optional argument *eof* is returned on end of file. If `eof` is omitted, an error is signaled on end of file.

`(read-from-keyboard)` procedure

This procedure is used when you want to give input to Scheme while running under Edwin. When you evaluate this procedure, the Scheme interpreter waits to send it an expression, (e.g. by typing `EXECUTE`) and then returns that expression. While Scheme is waiting, you can use all the Edwin commands normally used in editing. In particular, pressing the `ABORT` key will not cause Scheme to stop waiting for input. To force Scheme to stop waiting and return to top level, send it the expression `abort`.

(photo *filename*) procedure
 (tofu) procedure

Photo begins recording the console session to *filename*, and `tofu` stops this recording process.

(read) procedure
 (read *channel eof*) procedure

Returns the next object parsable from the given input *channel*, updating *channel* to point to the first character past the end of the written representation of the object. If an end of file is encountered in the input before any characters are found that can begin an object, then the end of file object *eof* is returned. This procedure should not be used to get input from the keyboard while Scheme is running in combination with Edwin—use `read-from-keyboard` instead.

(newline) procedure
 (newline *channel*) procedure

Writes a carriage return and line feed to *channel*. Returns an unspecified value.

(print *obj*) procedure
 (print *obj channel*) procedure
 (prin1 *obj*) procedure
 (prin1 *obj channel*) procedure
 (princ *obj*) procedure
 (princ *obj channel*) procedure

`Print` prints the value of *obj* on the console, or *channel*, if specified, preceding it with a carriage-return and line-feed. If *obj* is a string, it is output surrounded by double quotes.

`Prin1` behaves exactly the same as `print`, except that it does not precede the output with a carriage-return and line-feed.

`Princ` behaves exactly the same as `prin1`, except that it does not surround output strings with double quotes, and double quotes which are part of the string are not displayed preceded by backslash “\”.

(`print-depth`) procedure
 (`print-depth` *n*) procedure

Returns the depth to which nested list structures will be printed. If null, no restrictions are placed on print depth. If given the argument *n*, `print-depth` changes the print depth to *n*.

(`print-breadth`) procedure
 (`print-breadth` *n*) procedure

Returns the number of elements of list structure that will be printed out. If null, no restrictions are placed on print breadth. If given the argument *n*, `print-breadth` changes the print breadth to *n*.

4.13.2 Advanced Terminal and File I/O

Several of the procedures described in this section accept optional arguments. The optional argument *channel* is a channel object, as returned by `open-read-channel` or `open-printer-channel`. If the argument *channel* is omitted, the channel is defaulted to the console. For input operations, the optional argument *eof* can be included. On end of file, the object *eof* will be returned as the value of the input operation. If omitted, an error is signaled on end of file.

(`close-channel` *channel*) procedure

Closes an opened input or output channel. *channel* is an object created by `open-reader-channel` or `open-printer-channel`.

(`delete-file` *filename*) procedure

Deletes the file specified by character string *filename*.

(`file-exists?` *filename*) procedure

Returns `#t` if a file named *filename* exists, and `#f` otherwise. *Filename* is a character string.

(`open-reader-channel` *filename*) procedure

Returns a channel object which can be used for input from the file specified by character string *filename*. If the file does not exist, an error occurs.

(`open-printer-channel filename`) procedure

Returns a channel object which can be used for output to the file specified by character string *filename*. If that file already exists, its previous contents will be lost.

(`tyi`) procedure

(`tyi channel`) procedure

(`tyi channel eof`) procedure

Reads a character from the terminal or specified reader *channel* and returns its ASCII code, an integer. The optional argument *eof* is an object which will be returned by `tyi` on end of file.

(`tyo n`) procedure

(`tyo n channel`) procedure

Outputs the character whose ASCII value is *n* on the console or specified output *channel*.

(`readch`) procedure

(`readch channel`) procedure

(`readch channel eof`) procedure

Reads the next character from the terminal or specified input *channel* and returns it as a symbol. *Eof* will be returned on end of file.

(`peekch`) procedure

(`peekch channel`) procedure

(`peekch channel eof`) procedure

Peeks at the next character from the terminal or specified input *channel* without actually reading it, and returns it as a symbol. *Eof* will be returned on end of file.

(`tyipeek`) procedure

(`tyipeek channel`) procedure

(`tyipeek channel eof`) procedure

Peeks at the next character from the terminal or specified input *channel* without actually reading it, and returns its ASCII code, an integer. *Eof* will be returned on end of file.

(initialize-floppy *drive name*)

Formats a floppy disk for use. *Drive* is either the symbol `left` or the symbol `right`, specifying which floppy to format. *Name* is a character string specifying the name to give the floppy; it must be ten characters or fewer and end with a colon “:” character. Every floppy must be initialized before it is used the first time, but Edwin automatically formats your disk when you log in for the first time.

```
(initialize-floppy 'right "F00:")
```

(backup-floppy)

Backs up a floppy disk. When you run this command, you will be asked to insert the source disk in one drive and a target disk in the other drive. The program will initialize the target disk if necessary. Be sure to follow the instructions carefully so as not to confuse the source disk and the target disk.

4.14 Environments and evaluation

An *environment* is the set of bindings for variables that is in effect at some point in a program. (See section 4.2.) Environments in Scheme are first-class objects. That is, they may be named by variables, passed as arguments to procedures, and returned as the values of procedures. Scheme includes operations for capturing the environment in effect at any point and for specifying that an expression should be evaluated relative to a particular environment.

(eval *exp env*) procedure

Returns the result of evaluating *exp* in the environment *env*.

(the-environment) syntax

Evaluating the form `the-environment` returns the environment in which the form is evaluated.

Note: Scheme's static-scoping semantics imply that `the-environment` is not a procedure.

`(environment? obj)` procedure
Returns `#t` or `#f` depending on whether `obj` is an environment.

`(make-environment exp1 exp2 ...)` syntax
Make-environment constructs a new environment that is contained in the environment in which this form is evaluated. The expressions `exp1`, `exp2`, ... are evaluated sequentially in this new environment, and the environment is returned. The `expi` are typically `define` expressions. A `make-environment` form is alternate syntax for

```
(let () exp1 exp2 ... (the-environment))
```

`user-initial-environment` variable
A variable that is initially bound to the environment in which expressions entered from the console are evaluated when Scheme is first run.

4.15 Miscellaneous

`(applicable? obj)` procedure
Returns `#t` if `obj` can be applied, and `#f` otherwise. Primitive and compound procedures are applicable.

`(object-type obj)` procedure
Returns the primitive datatype of the object `obj`. The following are the data types that would be encountered in user code:

- `character`
- `character-string`
- `delayed`

- `environment`
- `list`
- `null`
- `number`
- `primitive-procedure`
- `procedure`
- `symbol`
- `true`
- `vector`

4.16 Advanced Features

`(enable-language-features)` procedure
`(disable-language-features)` procedure

The Scheme system includes many more built-in procedures and special forms than are documented in this manual. Most of these are for system programming only. When Scheme is initially loaded, only the procedures in this manual are accessible from the initial user environment. Evaluating `enable-language-features` makes the additional procedures accessible. Evaluating `disable-language-features`, in turn, disables these features.

4.17 Graphics

This section describes the basic graphics primitives available on the Chipmunks.

Graphics on the Chipmunks is displayed separately, but on the same screen as, the standard alphanumeric output. The graphics screen is made up of points which are either on or off. Each point is addressed by its x and

y coordinates on the screen, the origin being the middle of the screen. The x direction is numbered from -256 (left) to 255 (right). The y direction is numbers from 194 (top) to -195 (bottom). Some of the graphics procedures deal with a “pen.” The pen can be moved from its current position to another point, either drawing a line or not.

- (clear-graphics) procedure
Clears all graphics from the graphics screen and places the pen at point (0,0) (i.e., the middle of the screen).
- (draw-point x y) procedure
Plots the point at (x, y) .
- (clear-point x y) procedure
Erases the point at (x, y) .
- (position-pen x y) procedure
Moves the graphics pen to the point specified by the coordinates (x, y) .
- (draw-line-to x y) procedure
Draws a line from the current pen position to (x, y) and leaves the pen positioned at (x, y) .
- (store-graphics *filename*) procedure
Saves the image on the graphics screen in the designated file.
- (load-graphics *filename*) procedure
Interprets the contents of the designated file as a graphics picture, and displays this on the graphics screen.
- (print-graphics-file *filename*) procedure
Interprets the contents of the designated file as a graphics picture, displays this on the graphics screen, and prints the image on an attached graphics printer. *Do not use this command if there is no graphics printer attached to the Chipmunk!*

Appendix A

Glossary

Buffer A *buffer* is a block of text that you may examine and change. Whenever you edit in Edwin, you change the contents of a buffer. To preserve a buffer for another time you will use the Chipmunk, you must *save* the buffer to a *file* on disk.

Commands A *command* is an instruction you give Edwin through special keys. For example, use the **Find File** command to copy a file from a disk into a new buffer. Edwin commands are normally used by pressing the grey function keys across the top of the Chipmunk keyboard. However, if you're used to Emacs, you may want to use control and escape sequences, instead; in Edwin, most of them work exactly as they work in Emacs. (Use the **RUN** key whenever you would normally use **ESC**.)

Cursor The cursor is the blinking underline visible on the screen whenever the Chipmunk is waiting for you to type something. If you type an ordinary character (letter, number, or punctuation), it will be inserted at the current cursor position, and the cursor will move past it.

Extended Commands An *extended command* is a command used by name, rather than by special keys. To use an extended command, press the **EXTEND** key (**K₂**), followed by the name of the command. Your typing will appear at the bottom of the screen, in the typein window (see below). When you've typed the full name of the extended command, press **ENTER**.

File A *file* is a block of text stored on a disk. A file may be copied from a disk into a buffer. A file is created by saving the contents of a buffer on a disk.

Kills The several most recently-deleted regions are kept in the *kill ring*. Killed text can be retrieved with a *yank*.

Mark The *mark* is an invisible point in a buffer. Many commands set it to the buffer location where they were performed. Each buffer has its own mark.

Modes Every buffer has one major mode, and maybe some minor modes. *Major* modes determine special Edwin behavior particular to the language you are using (e.g.,

Scheme or English). *Minor* modes add special features regardless of the language you are using in the buffer.

Point The current cursor location in each buffer is called the *point*. The cursor is just a reflection of the point in the current buffer.

Region Text between the *point* and the *mark* is called the *region*.

Saving To *save* is to create a disk file from a buffer. The file and the buffer are identical, but only the file remains when you logout; the buffer is erased.

Window A *window* is a rectangular area on the screen. Each window displays part of some buffer. Because most buffers are too big to fit in one window, only part of the buffer is visible at any time; you can see other parts by using Edwin commands to scroll through the buffer.

Bibliography

- [1] Harold Abelson and Gerald Jay Sussman with Julie Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, 1985.
- [2] Jonathan Rees and William Clinger, editors. *Revised³ Report on the Algorithmic Language Scheme*. Newsletter of the ACM Special Interest Group on Programming Languages, November 1986.
- [3] Tod Cass. *Don't Panic: A 6.001 User's guide to the Chipmunk System*. January, 1984.
- [4] Harold Abelson. *Go Ahead – Panic: A 6.003 User's Guide to the Bobcat System*. Spring Semester, 1987.
- [5] Richard M. Stallman. *Gnu Emacs manual*. Fourth Edition, Version 17. February 1986.

Index

'() 75
(log 0) 87
* 84
+ 84
- 84
-1+ 85
/ 84
1+ 85
< 83
<= 84
= 83; 72, 73
=> 71
> 84
>= 84

abs 85
alphaless? 81
and 69
append 78
applicable? 97
apply 89
ascii 82
assoc 80; 81
assq 80; 81
assv 80; 81
atan 86; 87
atom? 75

backup-floppy 96; 12
binding 61

binding construct 61
binding constructs 61
bkpt 55
bound 61
break 52
break 53
break-both 53
break-entry 53
break-exit 53
breakpoint-procedure 55

caar 77
caddr 77
caddr 77
cadr 77
call 63
call by need 91
car 76; 73, 74, 75, 77, 78, 80
cdddar 77
cddddr 77
cdr 76; 73, 74, 75, 77, 78
ceiling 86
char 82
character 97
character-string 97
clear-graphics 99
clear-point 99
close-channel 94
combination 63
comment 59
cond 71
conjunction 69
cons 76; 74, 78, 85
cons-stream 91; 90
continuation browser 45
cos 86

debug 45

- define 66; 97
- delay 91; 92
- delayed 97
- delete-file 94
- disable-language-features 98
- disjunction 70
- draw-line-to 99
- draw-point 99

- eighth 77
- else 71
- empty list 74; 68, 76, 78
- empty stream 90
- empty-stream? 91
- enable-language-features 98
- environment 96
- environment 98
- environment browser 49
- environment? 97
- eof 92
- eq? 72; 73, 80
- equal? 73; 72, 80
- eqv? 72; 73, 76, 80, 81, 83
- error 55
- error-procedure 55
- escape commands 17
- eval 96
- even? 83
- exp 86
- explode 82
- expt 87

- false 68
- #false 68
- false 68
- fifth 77
- file-exists? 94

- first 77
- floor 86
- force 91; 92
- fourth 77

- gcd 86
- generate-uninterned-symbol 83
- graphics 98

- head 91; 90

- identifier 59; 61
- if 70
- implode 82
- improper list 74
- initialize-floppy 96
- integer-divide 85
- integer? 83
- internal definition 67

- lambda 63
- lambda expression 61
- last 79
- lazy evaluation 91
- left 96
- length 78
- let 64, 65
- list 74
- list 78; 88, 98
- list* 78
- list-ref 80
- list-tail 79
- list? 75
- load-graphics 99
- load-picture 10
- log 86

- make-environment 97

mapc 90
mapcar 90
max 84
member 80; 81
memq 80; 81
memv 80; 81
min 84

named let 65
negative? 83
newline 93
nil 68
not 68
nth 80
nthcdr 79
null 98
null? 78
number 58, 83
number 98
number? 83

object-type 97
odd? 83
open-printer-channel 95; 92, 94
open-read-channel 94
open-reader-channel 94; 92
or 69

pair 74
pair? 75
peekch 95
photo 93
position-pen 99
positive? 83
primitive-procedure 98
prin1 93
princ 93
print 93

print-breadth 94
print-depth 94
print-graphics-file 99
print-picture-file 10
proc 90
procedure 98
procedure call 63
promise 91

quote 62
quotient 85; 86

random 87
read 93
read-from-keyboard 92
readch 95
reductions 43
region 61; 64, 65
remainder 85; 86
reverse 79
right 96
round 86

second 77
sequence 66
set! 65
set-car! 77; 74
set-cdr! 77; 74, 75
seventh 77
sin 86
sixth 77
special form 59
specialform 61
sqrt 87
store-graphics 99
store-picture 10
stream 90
subproblems 43

symbol 98
symbol? 81

t 68
tail 91; 90
tan 86
the-empty-stream 91
the-environment 96; 97
third 77
tofu 93
tokens 59
top level environment 61
trace 50
trace 50
trace-both 50
trace-entry 50
trace-exit 50
true 68, 70, 71
#true 68
true 68; 98
truncate 86
tyi 95
tyipeek 95
tyo 95

unbound 61; 62, 67
unbreak 53
untrace 50
user-initial-environment 97

variable 61; 59
vector 88; 98
vector-cons 88
vector-ref 89; 88
vector-set! 89; 88
vector-size 88
vector? 88

where 49
whitespace 59
zero? 83