Appendices

A Running Scheme

The precise incantations needed to start Scheme depend on the particular version you're using and the model of computer and operating system you have. It's beyond the scope of this book to teach you the first steps in using a computer; we assume you've already used other programs, if not Scheme. But in this appendix we suggest a few general ideas and point out some knotty details.

One thing that beginners often forget is that a computer generally has many different programs available, and each one has its own capabilities and its own method of operation. If you think of yourself as interacting with "the computer," you're likely to try to use a command suitable for one program when you're actually using a different program. In learning to program in Scheme, you'll probably use at least three programs: Scheme itself, the operating system's *shell* (which is callled *finder* on the Macintosh and *explorer* on Windows), and a text editor. (The text editor may be part of the Scheme package or it may be an entirely separate program.) The shell allows you to run other programs, such as a printing utility or an electronic mail reader.

If you say `(+ 2 3)` to your text editor, it won't respond by printing `5`. Instead, it will insert the seven characters that you typed into the file that you're editing. If you type the same thing to Scheme, it will evaluate the expression.

The Program Development Cycle

Scheme is an interactive language: You can write a program by typing its definition directly into the Scheme interpreter. This ability to interact with Scheme is a great advantage for one-time calculations and for exploratory work, but it's not the best approach for the systematic development of a large program.

There are two issues to consider. First, when writing a large program, you generally don't get it perfect the first time. You make both typing errors and program logic errors, and so you must be able to revise a definition. Typing directly to Scheme, the only way to make such a revision is to retype the entire definition. Second, Scheme does not provide a mechanism to save your work in a permanent file.

For these reasons, programs are generally typed into another program, a text editor, rather than directly at the Scheme prompt. As we'll explain in the next section, there are several ways in which an editing program can be *integrated* with Scheme, so that the work you do in the editor can be communicated easily to Scheme. But the distinction between Scheme and the editor is easiest to understand if we start by considering the worst possible situation, in which the two are not integrated.

Imagine, therefore, that you have two separate programs available on your computer. One program is a Scheme interpreter. When you start Scheme, you may see some initial message, and then you see a prompt, which is a signal from Scheme that it's ready for you to type something. In this book we've used the character "`>`" as the prompt. Then, as we explain in the text, you can type an expression, and Scheme will compute and print the value:

```
> (+ 2 3)
5
```

Your other program is a text editor. This might be a general-purpose word processing program, with facilities for fancy text formatting, or it might be an editor intended specifically for computer programs. Many editors "know" about Lisp programs and have helpful features, such as automatic indentation, selection of complete expressions, and showing you the matching open parenthesis when you type a close parenthesis.

To write a program, you use the editor. Although you are typing Scheme expressions, you're not talking to Scheme itself, and so the expressions are not evaluated as you type them. Instead, they just appear on the screen like any other text. When you're ready to try out your program, you tell the editor to save the text in a file. (The command to save the program text is the same as it would be for any other text; we assume that you already know how to use the editor on your computer.) You can give the file any name you want, although many people like to use names like *`something`*`.scm` to make it easy to recognize files that contain Scheme programs.

Now you switch from the editor to Scheme. To read your program file into Scheme, you enter the expression

```
(load "something.scm")
```

This tells Scheme to read expressions from the specified file.*

Once Scheme has read the program definitions from your file, you can continue typing expressions to Scheme in order to test your program. If this testing uncovers an error, you will want to change some definition. Instead of typing the changed definition directly into Scheme, which would only make a temporary change in your program, you switch back to the editor and make the change in your program file. Then switch back to Scheme, and `load` the corrected file.

This sequence of steps—edit a file, make changes, save the file, switch to Scheme, load the file, test the program, find an error—is called a "development cycle" because what comes after "find an error" is editing the file, beginning another round of the same steps.

Integrated Editing

The development process can become much more convenient if Scheme and the editor "know about" each other. For example, instead of having to reload an entire file when you change one procedure definition, it's faster if your editor can tell Scheme just the one new definition. There are three general approaches to this integration: First, the editor can be in overall charge, with the Scheme interpreter running under control of the editor. Second, Scheme can be in charge, with the editor running under Scheme's supervision. Third, Scheme and the editor can be separate programs, both running under control of a third program, such as a window system, that allows information to be transferred between them.

If you're using a Unix system, you will be able to take a separate editor program and run Scheme from within that editor. The editor can copy any part of your program into the running Scheme, as if you had typed it to Scheme yourself. We use Jove, a free, small, fast version of EMACS. Most people use the more featureful GNU version of EMACS, which is installed on most Unix systems and available at `ftp://prep.ai.mit.edu/pub/gnu/` and many mirror sites for download.

If you're using a Macintosh or Windows version of Scheme, it will probably come with its own text editor and instructions on how to use it. These editors typically provide standard word-processing features such as cut and paste, search and replace, and saving

* If you see an error message about "end of file" or "EOF," it probably means that the file you are trying to load contains unbalanced parentheses; you have started an expression with a left parenthesis, and the file ended before Scheme saw a matching right parenthesis.

files. Also, they typically have a way to ask Scheme to evaluate an expression directly from the editor.

If you're using SCM under DOS, you should read the section "Editing Scheme Code" in the `README` file that comes with the SCM distribution. It will explain that editing can be done in different ways depending on the precise software available to you. You can buy a DOS editor that works like the Unix editors, or you can ask SCM to start a separate editor program while SCM remains active.

Finally, if you're running Scheme under Windows or another windowing operating system (like X or the Macintosh Finder), you can run any editor in another window and use the cut and paste facility to transfer information between the editor and Scheme.

Getting Our Programs

This book uses some programs that we wrote in Scheme. You'll want these files available to you while reading the book:

`simply.scm`	extended Scheme primitives
`functions.scm`	the `functions` program of Chapters 2 and 21
`ttt.scm`	the tic-tac-toe example from Chapter 10
`match.scm`	the pattern matcher example from Chapter 16
`spread.scm`	the spreadsheet program example from Chapter 24
`database.scm`	the beginning of the database project
`copyleft`	the GNU General Public License (see Appendix D)

In particular, the file `simply.scm` must be loaded into Scheme to allow anything in the book to work. Some Scheme systems allow you to load such a "startup" file permanently, so that it'll be there automatically from then on. In other versions of Scheme, you must say

```
(load "simply.scm")
```

at the beginning of every Scheme session.

There are three ways to get these program files:

- If you have access to the Internet, the most recent versions of all these files can be found at `ftp://anarres.cs.berkeley.edu/pub/scheme/`

- If you know someone who already has these files, you may copy them and distribute them freely. (The programs are copyrighted but are provided under a license that allows unlimited redistribution on a nonprofit basis; see Appendix D.)

- If you're stranded on a desert island with nothing but a computer and a copy of this book, you can type them in yourself; complete listings for all six programs, plus the GNU Public License, appear in the text of the book.

Tuning Our Programs for Your System

Almost all of the programs we distribute with this book will work without modification in the popular versions of Scheme. We've included "defensive" procedures that allow our programs to work even in versions that don't conform to current Scheme standards in various ways. However, there are a few details that we couldn't make uniform in all versions.

1. Many versions of Scheme include a `random` procedure to generate random numbers, but the standard does not require it, and so we've provided one just in case. If your Scheme includes a primitive `random`, it's probably better than the one we provide, because we have no way to choose a different starting value in each Scheme session.

Before loading `simply.scm` into Scheme, do the following experiment:

```
> (random 5)
```

If you get an error message, do nothing. If you get a random number as the result, edit `simply.scm` and remove the definition of `random`.

2. Do the following experiment:

```
> (error "Your error is" "string")
```

If the message you get doesn't include quotation marks around the word `string`, then do nothing. But if you do see `"string"` with quotation marks, edit `simply.scm` and change the definition of `error-printform` to

```
(define (error-printform x) x)
```

3. Although the Scheme standard says that the `read` procedure should not read the newline character following an expression that it reads, some old versions of Scheme get this wrong.

After loading `simply.scm`, do the following experiment:

```
> (read-line)
```

End the line with the `return` or `enter` key (whichever is appropriate in your version of Scheme) as usual, but don't type a second `return` or `enter` yet. If Scheme prints `()` right away, skip this paragraph; your version of Scheme behaves correctly. If, on the other hand, nothing happens, type another `return` or `enter`. In this case you must edit `functions.scm` and remove the invocation of `read-line` on the first line of the body of the `functions` procedure.

4. There is a substantial loss of efficiency in treating strings of digits as numbers in some contexts and as text in other contexts. When we're treating 1024 as text, we want to be able to take its `butfirst`, which should be 024. But in Scheme, `024` is the same as `24`, so instead `butfirst` returns a string:

```
> (butfirst 1024)
"024"
```

Yet we want to be able to do arithmetic on this value:

```
> (+ 3 (butfirst 1024))
27
```

To accomplish this, we redefine all of Scheme's arithmetic procedures to accept strings of digits and convert them to numbers. This redefinition slows down all arithmetic, not just arithmetic on strange numbers, and it's only rarely important to the programs we write. Therefore, we've provided a way to turn this part of the package off and on again. If your programs run too slowly, try saying

```
> (strings-are-numbers #f)
```

If you find that some program doesn't work because it tries to do arithmetic on a digit string and gets an error message, you can say

```
> (strings-are-numbers #t)
```

to restore the original behavior of our programs. We recommend that you leave `strings-are-numbers` true while exploring the first few chapters, so that the behavior of the word data type will be consistent. When you get to the large example programs, you may want to change to false.

Loading Our Programs

Scheme's `load` procedure doesn't scan your entire disk looking for the file you want to load. Instead, it only looks in one particular directory (DOS/Unix) or folder (Macintosh/Windows). If you want to load our programs, you have to make sure that Scheme can find them.

The first way to accomplish this is to give the full "path" as part of the argument to `load`. Here are some examples:*

```
UNIX-SCHEME> (load "/usr/people/matt/scheme-stuff/simply.scm")

WINDOWS-SCHEME> (load "c:\\scheme\\simply.scm")

MAC-SCHEME> (load "Hard Disk:Scheme Folder:simply.scm")
```

Under Unix, directories in a path are separated by forward slash characters. Under Windows and DOS, directories are separated by backward slash characters, which have a special meaning to Scheme. So you must use double backslashes as in our example above. On a Macintosh, you separate the parts of a path with colons. (However, most versions of Scheme for the Macintosh or Windows have a load command in one of the menus that opens a standard file selection dialog box, so you can use that instead.)

The other possibility is to put the files in the place where your version of Scheme looks for them. In many versions of Scheme, `load` looks for files in the folder that contains the Scheme program itself. Put our files in that folder.

On Unix, the default loading directory is whatever directory you're in at the moment. If you want to work on different projects in different directories, there's no way to make it so that `load` will always find our files. (But see our suggestion about writing `book-load`.)

---

* Suggestion for instructors: when we teach this class, we define a procedure like

```
(define (book-load filename)
  (load (string-append "/usr/cs3/progs-from-book/" filename)))
```

so that students can just say

```
(book-load "functions.scm")
```

Versions of Scheme

There are lots of them, both free and commercial. Three places to look for pointers are

```
http://swissnet.ai.mit.edu/scheme-home.html
http://www.schemers.org
http://www.cs.indiana.edu/scheme-repository
```

In general, there are four things you should be sure to learn about whatever version of Scheme you choose:

- Most versions of Scheme include a *debugger* to help you find program errors. If you call a primitive with an argument not in its domain, for example, Scheme will start the debugger, which will have features to let you find out where in your program the error occurred. These debuggers vary greatly among versions of Scheme. The first thing you should learn is how to *leave* the debugger, so you can get back to a Scheme prompt!

- Many versions of Scheme will read an *initialization file* if you create one. That is, when you start Scheme, it will look for a file of a particular name (something like `init.scm`, but not usually exactly that), and if there is such a file, Scheme will `load` it automatically. You can copy our `simply.scm` file to the proper filename for your version, and you'll have our added primitives available every time you start Scheme.

- Most versions of Scheme provide a `trace` capability, but the format of the trace results are quite different from one version to another.

- If you are using a Macintosh, one thing to watch out for is that some versions of Scheme expect you to use the ENTER key at the end of an expression, while others expect you to use the RETURN key.

Scheme Standards

The Web sites listed above will provide the latest version of the *Revised^n Report on the Algorithmic Language Scheme.* You can get the document in either Postscript or HTML format.

IEEE Standard 1178-1990, *IEEE Standard for the Scheme Programming Language,* may be ordered from IEEE by calling 1-800-678-IEEE or 908-981-1393 or writing IEEE Service Center, 445 Hoes Lane, P.O. Box 1331, Piscataway, NJ 08855-1331, and using order number SH14209 ($28 for IEEE members, $40 for others). ISBN 1-55937-125-0.

B Common Lisp

The two most popular dialects of Lisp are Scheme and Common Lisp. This appendix, which assumes that you have finished the rest of this book, describes the most important differences between Scheme and Common Lisp so that you will be able to use Common Lisp if you need to. Common Lisp is the most popular language among Artificial Intelligence researchers, so AI courses often use Common Lisp.

Why Common Lisp Exists

Since the beginning of Lisp, many versions of the language were developed. Each dialect reflected different ideas about the most important capabilities to include in the language. This diversity made Lisp an exciting arena for research, but it also meant that a Lisp program written for one dialect couldn't be used elsewhere.

In 1984, a group of Lisp developers decided to define a version of Lisp that would combine the capabilities of all their favorite dialects, so that in the future they would all use the same language; thus the name "Common" Lisp. Common Lisp was not the first attempt at a universal Lisp dialect, but it was more successful than earlier efforts. In 1985 a revision of the language was begun under the aegis of ANSI, the American National Standards Institute. This ANSI sponsorship gave Common Lisp an official status that has contributed to its growing acceptance.

Since Common Lisp was designed by combining the capabilities of many earlier dialects, it's an enormous language with nearly 1000 primitives, including versions of several programs in this book. There is a primitive `sort` procedure, a procedure like `number-name` that spells numbers in English, and a `substitute` procedure identical to the one you wrote in an exercise, to name a few.

If you're writing your own programs in Common Lisp, you can ignore all the extra features and just use the capabilities you already know from Scheme. If you're trying to read someone else's Common Lisp program, we expect that you will have to look up many primitive procedures in a reference manual.

Defining Procedures and Variables

One minor difference between Scheme and Common Lisp is in the way procedures are defined. In Common Lisp,

```
(defun square (x)
  (* x x))
```

means the same as Scheme's

```
(define (square x)
  (* x x))
```

In Scheme, `define` is used both for procedures and for variables whose values aren't procedures. In Common Lisp, procedures are given names by a mechanism separate from the general variable mechanism; `defun` is only for procedures. To define a variable, use `defvar`:

```
common-lisp> (defvar x 6)
6

common-lisp> x
6
```

In Common Lisp, `defvar` returns the name of the variable you define. If a variable has already been defined, `defvar` will not change its value; for that you must use `setq`.

The Naming Convention for Predicates

In Common Lisp, names of predicate procedures end in a "`p`" (for "predicate") instead of a question mark. Unfortunately, this convention isn't followed strictly. For example, Common Lisp's version of the `null?` predicate is just "`null`," not "`nullp`."

No Words or Sentences

We've mentioned that Scheme doesn't really have words and sentences built in; neither does Common Lisp. So none of the following procedures have Common Lisp equivalents: `accumulate`, `appearances`, `before?`, `bf`, `bl`, `butfirst`, `butlast`, `count`, `empty?`, `every`, `first`, `item`, `keep`, `last`, `member?`, `se`, `sentence`, `word`, and `word?`. (Common Lisp does have lists, though, and list-related procedures such as `map`, `reduce`, `append`, and so on *do* have equivalents.)

True and False

Common Lisp doesn't have the Boolean values `#t` and `#f`. Instead, it has a single false value, `nil`, which is also the empty list.

```
common-lisp> (= 2 3)
NIL

common-lisp> (cdr '(one-word-list))
NIL

common-lisp> '()
NIL
```

`Nil` is a strange beast in Common Lisp. It isn't a variable with the empty list as its value; it's a special self-evaluating symbol. There is also `t`, a self-evaluating symbol with a true value.

```
common-lisp> 'nil
NIL

common-lisp> nil
NIL

common-lisp> t
T
```

Like Scheme, Common Lisp treats every non-false (i.e., non-`nil`) value as true. But be careful; in Common Lisp

```
common-lisp> (if (cdr '(one-word-list)) 'yes 'no)
```

has the value NO, because the empty list is nil.

In Common Lisp's cond, there is no equivalent to else; Common Lisp programmers instead use t as the condition for their last clause, like this:

```
(defun sign (n)
  (cond ((> n 0) 'positive)
        ((= n 0) 'zero)
        (t 'negative)))
```

Files

Common Lisp's mechanism for dealing with files is trivially different from Scheme's. What Scheme calls "ports," Common Lisp calls "streams." Also, there is only one procedure for opening streams; the direction is specified this way:

```
common-lisp> (defvar out-stream (open "outfile" :direction :output))
#<OUTPUT STREAM "outfile">

common-lisp> (close out-stream)
T

common-lisp> (defvar in-stream (open "infile" :direction :input))
#<INPUT STREAM "infile">

common-lisp> (close in-stream)
T
```

Note that the close procedure closes both input streams and output streams.

To read from an input stream, you must invoke read with three arguments:

```
common-lisp> (read stream nil anything)
```

The nil indicates that reaching the end of the file should not be an error. If read does reach the end of the file, instead of returning a special end-of-file object it returns its third argument. It's possible to choose any value as the indicator for reaching the end of the file:

```
(let ((next (read stream nil 'xyzzy)))
  (if (equalp next 'xyzzy)
      'done
      (do-something next)))
```

It's important to choose an end-of-file indicator that couldn't otherwise appear as a value in the file.

Arrays

In Common Lisp, vectors are just a special case of the multidimensional *array* data type that you invented in Exercise 23.15. There are quite a few differences between Common Lisp arrays and Scheme vectors, none very difficult, but too numerous to describe here. If you need to use arrays, read about them in a Common Lisp book.

Equivalents to Scheme Primitives

Other than the word and sentence procedures, here is a table of the Scheme primitives from the table on page 553 that have different names, slightly different behavior, or do not exist at all in Common Lisp. Scheme procedures not in this list (other than the word and sentence ones) can be used identically in Common Lisp.

Scheme	Common Lisp
`align`	Common Lisp's `format` primitive has a similar purpose.
`begin`	`progn`
`boolean?`	Doesn't exist; see the section in this appendix about true and false values.
`c...r`	The same, but `(c...r nil)` is `nil` instead of an error.
`children`	You can use our version from Chapter 18.
`close-...-port`	`close`
`close-all-ports`	Doesn't exist.
`cond`	The same, except for `else`; use `t` instead.
`datum`	You can use our version from Chapter 18.
`define`	Either `defun`, for procedures, or `defvar`, otherwise.
`display`	`princ`
`eof-object?`	See the section on files.
`equal?`	`equalp`
`even?`	`evenp`
`filter`	`remove-if-not`
`for-each`	`mapc`
`integer?`	`integerp`
`lambda`	Discussed later in this appendix.
`list?`	`listp`, except that `listp` also returns true for improper lists.
`list-ref`	`nth`, except that the arguments come in reverse order.
`list->vector`	See the section about arrays.
`make-node`	You can use our version from Chapter 18.
`make-vector`	See the section about arrays.

`map`	`mapcar`
`newline`	`terpri`
`null?`	`null`
`number?`	`numberp`
`odd?`	`oddp`
`open-...-file`	See the section on files.
`procedure?`	`functionp`
`quotient`	`truncate`
`read`	Identical except for end of file. See the section on files.
`read-line`	Doesn't exist. (Common Lisp's `read-line` is like our `read-string`.)
`read-string`	`read-line`
`reduce`	The same, but computes `(f (f a b) c)` instead of `(f a (f b c))`.
`remainder`	`rem`
`repeated`	Doesn't exist.
`show`	Doesn't exist but easy to write.
`show-line`	Doesn't exist.
`vector-`*anything*	See the section about arrays.
`write`	`prin1`

A Separate Name Space for Procedures

All of the differences noted in this table are fairly minor ones, in the sense that the translation needed to account for these differences requires little more than renaming. There is one major conceptual difference between the two languages, however, in the way they treat names of procedures. Common Lisp allows a procedure and a variable to have the same name. For example, the program

```
(defun three-copies (list)
  (list list list list))
```

is perfectly legal.

```
common-lisp> (three-copies '(drive my car))
((DRIVE MY CAR) (DRIVE MY CAR) (DRIVE MY CAR))
```

How can Common Lisp tell that one of the `list`s means the primitive procedure, but the other ones mean the formal parameter? Symbols in the first position in a list (right after an open parenthesis) are taken to be names of globally defined procedures.

In Chapter 7 we introduced the image of a blackboard with all the global variables written on it, which all the Scheme little people can see. In Common Lisp, there are *two* blackboards: one for global variables, just as in Scheme, and another one for procedures.

The procedure blackboard contains the primitive procedures and the procedures you define with `defun`. Names in the first position of an expression are looked up on the procedure blackboard.

Therefore, the names of procedures are not variables and cannot be used as actual argument expressions:

```
common-lisp> (sqrt 144)
12

common-lisp> (mapcar sqrt '(9 16 25 36))
ERROR: The variable SQRT is unbound.
```

(Common Lisp's equivalent of `map` is named `mapcar`.)

How, then, do you tell Common Lisp that you want to use the procedure named `sqrt` as data? You must use the `function` special form.*

```
common-lisp> (function sqrt)
#<PROCEDURE>

common-lisp> (mapcar (function sqrt) '(9 16 25 36))
(3 4 5 6)
```

`Function`'s job is to look up names on the procedure blackboard. (`Function` actually has a more general definition, as you'll see in a few paragraphs.)

Lambda

In Common Lisp, as in Scheme, procedures can be named or unnamed. Just as procedure names in Common Lisp are meaningful only in certain contexts, so are `lambda` expressions. They make sense at the beginning of an expression:

```
common-lisp> ((lambda (x) (* x x)) 4)
16
```

* Common Lisp uses the word "function" to mean "procedure," whether or not the procedure implements a function.

or as the argument to `function`:

```
common-lisp> (function (lambda (x) (* x x)))
#<PROCEDURE>

common-lisp> (mapcar (function (lambda (x) (* x x))) '(3 4 5 6))
(9 16 25 36)
```

but they're meaningless on their own:

```
common-lisp> (lambda (x) (* x x))
ERROR: LAMBDA is not a function

common-lisp> (mapcar (lambda (x) (* x x)) '(3 4 5 6))
ERROR: LAMBDA is not a function
```

More about `Function`

The official rule is that `function` returns the "functional interpretation" of its argument. If the argument is a symbol, that means looking up the procedure associated with that name. If the argument is a `lambda` expression, it means creating a new procedure. `Function` uses the same rule that's used to interpret the first element of a procedure invocation.

Since `function` is a very commonly used special form, it has an abbreviation:

```
common-lisp> (mapcar #'(lambda (x) (* x x)) '(3 4 5 6))
(9 16 25 36)

common-lisp> (mapcar #'cdr '((hey jude) (eleanor rigby) (yes it is)))
((JUDE) (RIGBY) (IT IS))
```

Don't confuse

```
#'(lambda (x) (* x x))
```

with

```
'#(lambda (x) (* x x))
```

The first of these is a function that squares its argument; the second is an array containing three elements.

It's unfortunate that the abbreviation for `function` contains a single quote mark, because the job of `function` is nothing like the job of `quote`. You'll just have to get used to the "hashquote" notation.

Writing Higher-Order Procedures

Think about this attempted translation of the `map` procedure:

```
(defun map (fn lst)                              ;; wrong!
  (if (null lst)
      '()
      (cons (fn (car lst))
            (map fn (cdr lst)))))
```

(In Common Lisp, `null` is one of the predicates whose names don't end in "p." Otherwise, this is the same program we showed you in Chapter 19, except for the `defun`, of course.)

According to our rule about names in the front of a list, this procedure doesn't work. Think about what happens when we say

```
(map #'square '(1 2 3 4 5))
```

According to the substitution model, the parameters `fn` and `lst` are replaced in the body with `#'square` and `'(1 2 3 4 5)`. But Common Lisp makes an exception for the first element of a compound expression. It uses the procedure blackboard instead of substitution:

```
(if (null '(1 2 3 4 5))
    '()
    (cons (fn (car '(1 2 3 4 5))
          (map #'square (cdr '(1 2 3 4 5))))))
```

Note that one of the appearances of `fn` was left unchanged. Since there is no global procedure named `fn`, this program will produce an error:

```
common-lisp> (map #'square '(1 2 3 4 5))
ERROR: FN is not a procedure.
```

How, then, do you write higher-order procedures in Common Lisp? The answer is that you must use `funcall`:

```
(defun map (fn lst)
  (if (null lst)
      '()
      (cons (funcall fn (car lst))
            (map fn (cdr lst)))))
```

`Funcall` takes one or more arguments. The first is a procedure and the rest are arguments for that procedure. It applies that procedure to the given arguments.* Since `fn` is no longer at the beginning of a compound expression, the corresponding argument, `#'square`, is substituted for it.

---

* This is a lot like `apply`, you may have noticed. Look at the difference:

```
common-lisp> (funcall #'+ 1 2 3)
6

common-lisp> (apply #'+ '(1 2 3))
6
```

In the first case, each argument to `+` is a separate argument to `funcall`. In the second case, a list of the arguments to `+` is a single argument to `apply`. `Apply` always takes exactly two arguments, the procedure and the argument list.

C Scheme Initialization File

Many of the procedures we talk about in this book aren't part of standard Scheme; we wrote them ourselves. Here is a listing of the definitions of those procedures.

```
;;; simply.scm version 3.13 (8/11/98)

;;; This file uses Scheme features we don't talk about in _Simply_Scheme_.
;;; Read at your own risk.

(if (equal? 'foo (symbol->string 'foo))
    (error "Simply.scm already loaded!!")
    #f)

;; Make number->string remove leading "+" if necessary

(if (char=? #\+ (string-ref (number->string 1.0) 0))
    (let ((old-ns number->string) (char=? char=?) (string-ref string-ref)
          (substring substring) (string-length string-length))
      (set! number->string
            (lambda args
              (let ((result (apply old-ns args)))
                (if (char=? #\+ (string-ref result 0))
                    (substring result 1 (string-length result))
                    result))))))
    'no-problem)

(define number->string
  (let ((old-ns number->string) (string? string?))
    (lambda args
      (if (string? (car args))
          (car args)
          (apply old-ns args)))))
```

```
;; Get strings in error messages to print nicely (especially "")

(define whoops
  (let ((string? string?) (string-append string-append)        (error error)
        (cons cons) (map map) (apply apply))
    (define (error-printform x)
      (if (string? x)
          (string-append "\"" x "\"")
          x))
    (lambda (string . args)
      (apply error (cons string (map error-printform args))))))


;; ROUND returns an inexact integer if its argument is inexact,
;; but we think it should always return an exact integer.
;; (It matters because some Schemes print inexact integers as "+1.0".)
;; The (exact 1) test is for PC Scheme, in which nothing is exact.
(if (and (inexact? (round (sqrt 2))) (exact? 1))
    (let ((old-round round) (inexact->exact inexact->exact))
      (set! round
            (lambda (number)
              (inexact->exact (old-round number)))))
    'no-problem)

;; Remainder and quotient blow up if their argument isn't an integer.
;; Unfortunately, in SCM, (* 365.25 24 60 60) *isn't* an integer.

(if (inexact? (* .25 4))
    (let ((rem remainder) (quo quotient) (inexact->exact inexact->exact)
          (integer? integer?))
      (set! remainder
            (lambda (x y)
              (rem (if (integer? x) (inexact->exact x) x)
                   (if (integer? y) (inexact->exact y) y))))
      (set! quotient
            (lambda (x y)
              (quo (if (integer? x) (inexact->exact x) x)
                   (if (integer? y) (inexact->exact y) y)))))
    'done)
```

```
;; Random
;; If your version of Scheme has RANDOM, you should take this out.
;; (It gives the same sequence of random numbers every time.)

(define random
  (let ((*seed* 1) (quotient quotient) (modulo modulo) (+ +) (- -) (* *) (> >))
    (lambda (x)
      (let* ((hi (quotient *seed* 127773))
             (low (modulo *seed* 127773))
             (test (- (* 16807 low) (* 2836 hi))))
        (if (> test 0)
            (set! *seed* test)
            (set! *seed* (+ test 2147483647))))
      (modulo *seed* x))))


;;; Logo-style word/sentence implementation

(define word?
  (let ((number? number?) (symbol? symbol?) (string? string?))
    (lambda (x)
      (or (symbol? x) (number? x) (string? x)))))

(define sentence?
  (let ((null? null?) (pair? pair?) (word? word?) (car car) (cdr cdr))
    (define (list-of-words? l)
      (cond ((null? l) #t)
            ((pair? l)
             (and (word? (car l)) (list-of-words? (cdr l))))
            (else #f)))
    list-of-words?))

(define empty?
  (let ((null? null?) (string? string?) (string=? string=?))
    (lambda (x)
      (or (null? x)
          (and (string? x) (string=? x ""))))))
```

```
(define char-rank
  ;; 0 Letter in good case or special initial
  ;; 1 ., + or -
  ;; 2 Digit
  ;; 3 Letter in bad case or weird character
  (let ((*the-char-ranks* (make-vector 256 3))
        (= =) (+ +) (string-ref string-ref) (string-length string-length)
        (vector-set! vector-set!) (char->integer char->integer)
        (symbol->string symbol->string) (vector-ref vector-ref))
    (define (rank-string str rank)
      (define (helper i len)
        (if (= i len)
            'done
            (begin (vector-set! *the-char-ranks*
                                (char->integer (string-ref str i))
                                rank)
                   (helper (+ i 1) len))))
      (helper 0 (string-length str)))
    (rank-string (symbol->string 'abcdefghijklmnopqrstuvwxyz) 0)
    (rank-string "!$%&*/:<=>?~_^" 0)
    (rank-string "+-." 1)
    (rank-string "0123456789" 2)
    (lambda (char)                    ;; value of char-rank
      (vector-ref *the-char-ranks* (char->integer char)))))

(define string->word
  (let ((= =) (<= <=) (+ +) (- -) (char-rank char-rank) (string-ref string-ref)
        (string-length string-length) (string=? string=?) (not not)
        (char=? char=?) (string->number string->number)
        (string->symbol string->symbol))
    (lambda (string)
      (define (subsequents? string i length)
        (cond ((= i length) #t)
              ((<= (char-rank (string-ref string i)) 2)
               (subsequents? string (+ i 1) length))
              (else #f)))
      (define (special-id? string)
        (or (string=? string "+")
            (string=? string "-")
            (string=? string "...")))
      (define (ok-symbol? string)
        (if (string=? string "")
            #f
            (let ((rank1 (char-rank (string-ref string 0))))
              (cond ((= rank1 0) (subsequents? string 1 (string-length string)))
                    ((= rank1 1) (special-id? string))
                    (else #f)))))
```

```
      (define (nn-helper string i len seen-point?)
        (cond ((= i len)
               (if seen-point?
                   (not (char=? (string-ref string (- len 1)) #\0))
                   #t))
              ((char=? #\. (string-ref string i))
               (cond (seen-point? #f)
                     ((= (+ i 2) len) #t)  ; Accepts "23.0"
                     (else (nn-helper string (+ i 1) len #t))))
              ((= 2 (char-rank (string-ref string i)))
               (nn-helper string (+ i 1) len seen-point?))
              (else #f)))
      (define (narrow-number? string)
        (if (string=? string "")
            #f
            (let* ((c0 (string-ref string 0))
                   (start 0)
                   (len (string-length string))
                   (cn (string-ref string (- len 1))))
              (if (and (char=? c0 #\-) (not (= len 1)))
                  (begin
                   (set! start 1)
                   (set! c0 (string-ref string 1)))
                  #f)
              (cond ((not (= (char-rank cn) 2)) #f)  ; Rejects "-" among others
                    ((char=? c0 #\.) #f)
                    ((char=? c0 #\0)
                     (cond ((= len 1) #t)  ; Accepts "0" but not "-0"
                           ((= len 2) #f)  ; Rejects "-0" and "03"
                           ((char=? (string-ref string (+ start 1)) #\.)
                            (nn-helper string (+ start 2) len #t))
                           (else #f)))
                    (else (nn-helper string start len #f))))))

      ;; The body of string->word:
      (cond ((narrow-number? string) (string->number string))
            ((ok-symbol? string) (string->symbol string))
            (else string)))))

(define char->word
  (let ((= =) (char-rank char-rank) (make-string make-string) (char=? char=?)
        (string->symbol string->symbol) (string->number string->number))
    (lambda (char)
      (let ((rank (char-rank char))
            (string (make-string 1 char)))
        (cond ((= rank 0) (string->symbol string))
              ((= rank 2) (string->number string))
              ((char=? char #\+) '+)
              ((char=? char #\-) '-)
              (else string))))))
```

```
(define word->string
  (let ((number? number?) (string? string?) (number->string number->string)
        (symbol->string symbol->string))
    (lambda (wd)
      (cond ((string? wd) wd)
            ((number? wd) (number->string wd))
            (else (symbol->string wd))))))

(define count
  (let ((word? word?) (string-length string-length)
        (word->string word->string) (length length))
    (lambda (stuff)
      (if (word? stuff)
          (string-length (word->string stuff))
          (length stuff)))))

(define word
  (let ((string->word string->word) (apply apply) (string-append string-append)
        (map map) (word? word?) (word->string word->string) (whoops whoops))
    (lambda x
      (string->word
       (apply string-append
              (map (lambda (arg)
                     (if (word? arg)
                         (word->string arg)
                         (whoops "Invalid argument to WORD: " arg)))
                   x))))))

(define se
  (let ((pair? pair?) (null? null?) (word? word?) (car car) (cons cons)
        (cdr cdr) (whoops whoops))
    (define (paranoid-append a original-a b)
      (cond ((null? a) b)
            ((word? (car a))
             (cons (car a) (paranoid-append (cdr a) original-a b)))
            (else (whoops "Argument to SENTENCE not a word or sentence"
                          original-a ))))
    (define (combine-two a b)                ;; Note: b is always a list
      (cond ((pair? a) (paranoid-append a a b))
            ((null? a) b)
            ((word? a) (cons a b))
            (else (whoops "Argument to SENTENCE not a word or sentence:" a))))
    ;; Helper function so recursive calls don't show up in TRACE
    (define (real-se args)
      (if (null? args)
          '()
          (combine-two (car args) (real-se (cdr args)))))
    (lambda args
      (real-se args))))
```

```
(define sentence se)

(define first
  (let ((pair? pair?) (char->word char->word) (string-ref string-ref)
        (word->string word->string) (car car) (empty? empty?)
        (whoops whoops) (word? word?))
    (define (word-first wd)
      (char->word (string-ref (word->string wd) 0)))
    (lambda (x)
      (cond ((pair? x) (car x))
            ((empty? x) (whoops "Invalid argument to FIRST: " x))
            ((word? x) (word-first x))
            (else (whoops "Invalid argument to FIRST: " x))))))

(define last
  (let ((pair? pair?) (- -) (word->string word->string) (char->word char->word)
        (string-ref string-ref) (string-length string-length) (empty? empty?)
        (cdr cdr) (car car) (whoops whoops) (word? word?))
    (define (word-last wd)
      (let ((s (word->string wd)))
        (char->word (string-ref s (- (string-length s) 1)))))
    (define (list-last lst)
      (if (empty? (cdr lst))
          (car lst)
          (list-last (cdr lst))))
    (lambda (x)
      (cond ((pair? x) (list-last x))
            ((empty? x) (whoops "Invalid argument to LAST: " x))
            ((word? x) (word-last x))
            (else (whoops "Invalid argument to LAST: " x))))))

(define bf
  (let ((pair? pair?) (substring substring) (string-length string-length)
        (string->word string->word) (word->string word->string) (cdr cdr)
        (empty? empty?) (whoops whoops) (word? word?))
    (define string-bf
      (lambda (s)
      (substring s 1 (string-length s))))
    (define (word-bf wd)
      (string->word (string-bf (word->string wd))))
    (lambda (x)
      (cond ((pair? x) (cdr x))
            ((empty? x) (whoops "Invalid argument to BUTFIRST: " x))
            ((word? x) (word-bf x))
            (else (whoops "Invalid argument to BUTFIRST: " x))))))

(define butfirst bf)
```

```
(define bl
  (let ((pair? pair?) (- -) (cdr cdr) (cons cons) (car car) (substring substring)
        (string-length string-length) (string->word string->word)
        (word->string word->string) (empty? empty?) (whoops whoops) (word? word?))
    (define (list-bl list)
      (if (null? (cdr list))
          '()
          (cons (car list) (list-bl (cdr list)))))
    (define (string-bl s)
      (substring s 0 (- (string-length s) 1)))
    (define (word-bl wd)
      (string->word (string-bl (word->string wd))))
    (lambda (x)
      (cond ((pair? x) (list-bl x))
            ((empty? x) (whoops "Invalid argument to BUTLAST: " x))
            ((word? x) (word-bl x))
            (else (whoops "Invalid argument to BUTLAST: " x))))))

(define butlast bl)

(define item
  (let ((> >) (- -) (< <) (integer? integer?) (list-ref list-ref)
        (char->word char->word) (string-ref string-ref)
        (word->string word->string) (not not) (whoops whoops)
        (count count) (word? word?) (list? list?))
    (define (word-item n wd)
      (char->word (string-ref (word->string wd) (- n 1))))
    (lambda (n stuff)
      (cond ((not (integer? n))
             (whoops "Invalid first argument to ITEM (must be an integer): "
                     n))
            ((< n 1)
             (whoops "Invalid first argument to ITEM (must be positive): "
                     n))
            ((> n (count stuff))
             (whoops "No such item: " n stuff))
            ((word? stuff) (word-item n stuff))
            ((list? stuff) (list-ref stuff (- n 1)))
            (else (whoops "Invalid second argument to ITEM: " stuff))))))
```

```
(define equal?
  ;; Note that EQUAL? assumes strings are numbers.
  ;; (strings-are-numbers #f) doesn't change this behavior.
  (let ((vector-length vector-length) (= =) (vector-ref vector-ref)
        (+ +) (string? string?) (symbol? symbol?) (null? null?) (pair? pair?)
        (car car) (cdr cdr) (eq? eq?) (string=? string=?)
        (symbol->string symbol->string) (number? number?)
        (string->word string->word) (vector? vector?) (eqv? eqv?))
    (define (vector-equal? v1 v2)
      (let ((len1 (vector-length v1))
            (len2 (vector-length v2)))
        (define (helper i)
          (if (= i len1)
              #t
              (and (equal? (vector-ref v1 i) (vector-ref v2 i))
                   (helper (+ i 1)))))
        (if (= len1 len2)
            (helper 0)
            #f)))
    (lambda (x y)
      (cond ((null? x) (null? y))
            ((null? y) #f)
            ((pair? x)
             (and (pair? y)
                  (equal? (car x) (car y))
                  (equal? (cdr x) (cdr y))))
            ((pair? y) #f)
            ((symbol? x)
             (or (and (symbol? y) (eq? x y))
                 (and (string? y) (string=? (symbol->string x) y))))
            ((symbol? y)
             (and (string? x) (string=? x (symbol->string y))))
            ((number? x)
             (or (and (number? y) (= x y))
                 (and (string? y)
                      (let ((possible-num (string->word y)))
                        (and (number? possible-num)
                             (= x possible-num))))))
            ((number? y)
             (and (string? x)
                  (let ((possible-num (string->word x)))
                    (and (number? possible-num)
                         (= possible-num y)))))
            ((string? x) (and (string? y) (string=? x y)))
            ((string? y) #f)
            ((vector? x) (and (vector? y) (vector-equal? x y)))
            ((vector? y) #f)
            (else (eqv? x y))))))
```

```
(define member?
  (let ((> >) (- -) (< <) (null? null?) (symbol? symbol?) (eq? eq?) (car car)
        (not not) (symbol->string symbol->string) (string=? string=?)
        (cdr cdr) (equal? equal?) (word->string word->string)
        (string-length string-length) (whoops whoops) (string-ref string-ref)
        (char=? char=?) (list? list?) (number? number?) (empty? empty?)
        (word? word?) (string? string?))
    (define (symbol-in-list? symbol string lst)
      (cond ((null? lst) #f)
            ((and (symbol? (car lst))
                  (eq? symbol (car lst))))
            ((string? (car lst))
             (cond ((not string)
                    (symbol-in-list? symbol (symbol->string symbol) lst))
                   ((string=? string (car lst)) #t)
                   (else (symbol-in-list? symbol string (cdr lst)))))
            (else (symbol-in-list? symbol string (cdr lst)))))
    (define (word-in-list? wd lst)
      (cond ((null? lst) #f)
            ((equal? wd (car lst)) #t)
            (else (word-in-list? wd (cdr lst)))))
    (define (word-in-word? small big)
      (let ((one-letter-str (word->string small)))
        (if (> (string-length one-letter-str) 1)
            (whoops "Invalid arguments to MEMBER?: " small big)
            (let ((big-str (word->string big)))
              (char-in-string? (string-ref one-letter-str 0)
                               big-str
                               (- (string-length big-str) 1))))))
    (define (char-in-string? char string i)
      (cond ((< i 0) #f)
            ((char=? char (string-ref string i)) #t)
            (else (char-in-string? char string (- i 1)))))
    (lambda (x stuff)
      (cond ((empty? stuff) #f)
            ((word? stuff) (word-in-word? x stuff))
            ((not (list? stuff))
             (whoops "Invalid second argument to MEMBER?: " stuff))
            ((symbol? x) (symbol-in-list? x #f stuff))
            ((or (number? x) (string? x))
             (word-in-list? x stuff))
            (else (whoops "Invalid first argument to MEMBER?: " x))))))
```

```
(define before?
  (let ((not not) (word? word?) (whoops whoops) (string<? string<?)
        (word->string word->string))
    (lambda (wd1 wd2)
      (cond ((not (word? wd1))
             (whoops "Invalid first argument to BEFORE? (not a word): " wd1))
            ((not (word? wd2))
             (whoops "Invalid second argument to BEFORE? (not a word): " wd2))
            (else (string<? (word->string wd1) (word->string wd2)))))))


;;; Higher Order Functions

(define filter
  (let ((null? null?) (car car) (cons cons) (cdr cdr) (not not)
        (procedure? procedure?) (whoops whoops) (list? list?))
    (lambda (pred l)
      ;; Helper function so recursive calls don't show up in TRACE
      (define (real-filter l)
        (cond ((null? l) '())
              ((pred (car l))
               (cons (car l) (real-filter (cdr l))))
              (else (real-filter (cdr l)))))
      (cond ((not (procedure? pred))
             (whoops "Invalid first argument to FILTER (not a procedure): "
                     pred))
            ((not (list? l))
             (whoops "Invalid second argument to FILTER (not a list): " l))
            (else (real-filter l))))))
```

```
(define keep
  (let ((+ +) (= =) (pair? pair?) (substring substring)
        (char->word char->word) (string-ref string-ref)
        (string-set! string-set!) (word->string word->string)
        (string-length string-length) (string->word string->word)
        (make-string make-string) (procedure? procedure?)
        (whoops whoops) (word? word?) (null? null?))
    (lambda (pred w-or-s)
      (define (keep-string in i out out-len len)
        (cond ((= i len) (substring out 0 out-len))
              ((pred (char->word (string-ref in i)))
               (string-set! out out-len (string-ref in i))
               (keep-string in (+ i 1) out (+ out-len 1) len))
              (else (keep-string in (+ i 1) out out-len len))))
      (define (keep-word wd)
        (let* ((string (word->string wd))
               (len (string-length string)))
          (string->word
           (keep-string string 0 (make-string len) 0 len))))
      (cond ((not (procedure? pred))
             (whoops "Invalid first argument to KEEP (not a procedure): "
                     pred))
            ((pair? w-or-s) (filter pred w-or-s))
            ((word? w-or-s) (keep-word w-or-s))
            ((null? w-or-s) '())
            (else
             (whoops "Bad second argument to KEEP (not a word or sentence): "
                     w-or-s))))))

(define appearances
  (let ((count count) (keep keep) (equal? equal?))
    (lambda (item aggregate)
      (count (keep (lambda (element) (equal? item element)) aggregate)))))
```

```
(define every
  (let ((= =) (+ +) (se se) (char->word char->word) (string-ref string-ref)
        (empty? empty?) (first first) (bf bf) (not not) (procedure? procedure?)
        (whoops whoops) (word? word?) (word->string word->string)
        (string-length string-length))
    (lambda (fn stuff)
      (define (string-every string i length)
        (if (= i length)
            '()
            (se (fn (char->word (string-ref string i)))
                (string-every string (+ i 1) length))))
      (define (sent-every sent)
        ;; This proc. can't be optimized or else it will break the
        ;; exercise where we ask them to reimplement sentences as
        ;; vectors and then see if every still works.
        (if (empty? sent)
            sent                ; Can't be '() or exercise breaks.
            (se (fn (first sent))
                (sent-every (bf sent)))))
      (cond ((not (procedure? fn))
             (whoops "Invalid first argument to EVERY (not a procedure):"
                     fn))
            ((word? stuff)
             (let ((string (word->string stuff)))
               (string-every string 0 (string-length string))))
            (else (sent-every stuff))))))

(define accumulate
  (let ((not not) (empty? empty?) (bf bf) (first first) (procedure? procedure?)
        (whoops whoops) (member member) (list list))
    (lambda (combiner stuff)
      (define (real-accumulate stuff)
        (if (empty? (bf stuff))
            (first stuff)
            (combiner (first stuff) (real-accumulate (bf stuff)))))
      (cond ((not (procedure? combiner))
             (whoops "Invalid first argument to ACCUMULATE (not a procedure):"
                     combiner))
            ((not (empty? stuff)) (real-accumulate stuff))
            ((member combiner (list + * word se)) (combiner))
            (else
             (whoops "Can't accumulate empty input with that combiner"))))))
```

```
(define reduce
  (let ((null? null?) (cdr cdr) (car car) (not not) (procedure? procedure?)
        (whoops whoops) (member member) (list list))
    (lambda (combiner stuff)
      (define (real-reduce stuff)
        (if (null? (cdr stuff))
            (car stuff)
            (combiner (car stuff) (real-reduce (cdr stuff)))))
      (cond ((not (procedure? combiner))
             (whoops "Invalid first argument to REDUCE (not a procedure):"
                     combiner))
            ((not (null? stuff)) (real-reduce stuff))
            ((member combiner (list + * word se append)) (combiner))
            (else (whoops "Can't reduce empty input with that combiner"))))))

(define repeated
  (let ((= =) (- -))
    (lambda (fn number)
      (if (= number 0)
          (lambda (x) x)
          (lambda (x)
            ((repeated fn (- number 1)) (fn x)))))))


;; Tree stuff
(define make-node cons)
(define datum car)
(define children cdr)


;; I/O

(define show
  (let ((= =) (length length) (display display) (car car) (newline newline)
        (not not) (output-port? output-port?) (apply apply) (whoops whoops))
    (lambda args
      (cond
       ((= (length args) 1)
        (display (car args))
        (newline))
       ((= (length args) 2)
        (if (not (output-port? (car (cdr args))))
            (whoops "Invalid second argument to SHOW (not an output port): "
                    (car (cdr args))))
        (apply display args)
        (newline (car (cdr args))))
       (else (whoops "Incorrect number of arguments to procedure SHOW"))))))
```

```
(define show-line
  (let ((>= >=) (length length) (whoops whoops) (null? null?)
        (current-output-port current-output-port) (car car) (not not)
        (list? list?) (display display) (for-each for-each) (cdr cdr)
        (newline newline))
    (lambda (line . args)
      (if (>= (length args) 2)
          (whoops "Too many arguments to show-line")
          (let ((port (if (null? args) (current-output-port) (car args))))
            (cond ((not (list? line))
                   (whoops "Invalid argument to SHOW-LINE (not a list):" line))
                  ((null? line) #f)
                  (else
                   (display (car line) port)
                   (for-each (lambda (wd) (display " " port) (display wd port))
                             (cdr line))))
            (newline port))))))
```

```
(define read-string
  (let ((read-char read-char) (eqv? eqv?) (apply apply)
        (string-append string-append) (substring substring) (reverse reverse)
        (cons cons) (>= >=) (+ +) (string-set! string-set!) (length length)
        (whoops whoops) (null? null?) (current-input-port current-input-port)
        (car car) (cdr cdr) (eof-object? eof-object?) (list list)
        (make-string make-string) (peek-char peek-char))
    (define (read-string-helper chars all-length chunk-length port)
      (let ((char (read-char port))
            (string (car chars)))
        (cond ((or (eof-object? char) (eqv? char #\newline))
               (apply string-append
                      (reverse
                       (cons
                        (substring (car chars) 0 chunk-length)
                        (cdr chars)))))
              ((>= chunk-length 80)
               (let ((newstring (make-string 80)))
                 (string-set! newstring 0 char)
                 (read-string-helper (cons newstring chars)
                                     (+ all-length 1)
                                     1
                                     port)))
              (else
               (string-set! string chunk-length char)
               (read-string-helper chars
                                   (+ all-length 1)
                                   (+ chunk-length 1)
                                   port)))))
    (lambda args
      (if (>= (length args) 2)
          (whoops "Too many arguments to read-string")
          (let ((port (if (null? args) (current-input-port) (car args))))
            (if (eof-object? (peek-char port))
                (read-char port)
                (read-string-helper (list (make-string 80)) 0 0 port)))))))
```

```
(define read-line
  (let ((= =) (list list) (string->word string->word) (substring substring)
        (char-whitespace? char-whitespace?) (string-ref string-ref)
        (+ +) (string-length string-length) (apply apply)
        (read-string read-string))
    (lambda args
      (define (tokenize string)
        (define (helper i start len)
          (cond ((= i len)
                 (if (= i start)
                     '()
                     (list (string->word (substring string start i)))))
                ((char-whitespace? (string-ref string i))
                 (if (= i start)
                     (helper (+ i 1) (+ i 1) len)
                     (cons (string->word (substring string start i))
                           (helper (+ i 1) (+ i 1) len))))
                (else (helper (+ i 1) start len))))
        (if (eof-object? string)
            string
            (helper 0 0 (string-length string))))
      (tokenize (apply read-string args)))))

(define *the-open-inports* '())
(define *the-open-outports* '())
```

```
(define align
  (let ((< <) (abs abs) (* *) (expt expt) (>= >=) (- -) (+ +) (= =)
        (null? null?) (car car) (round round) (number->string number->string)
        (string-length string-length) (string-append string-append)
        (make-string make-string) (substring substring)
        (string-set! string-set!) (number? number?)
        (word->string word->string))
    (lambda (obj width . rest)
      (define (align-number obj width rest)
        (let* ((sign (< obj 0))
               (num (abs obj))
               (prec (if (null? rest) 0 (car rest)))
               (big (round (* num (expt 10 prec))))
               (cvt0 (number->string big))
               (cvt (if (< num 1) (string-append "0" cvt0) cvt0))
               (pos-str (if (>= (string-length cvt0) prec)
                            cvt
                            (string-append
                             (make-string (- prec (string-length cvt0)) #\0)
                             cvt)))
               (string (if sign (string-append "-" pos-str) pos-str))
               (length (+ (string-length string)
                          (if (= prec 0) 0 1)))
               (left (- length (+ 1 prec)))
               (result (if (= prec 0)
                           string
                           (string-append
                            (substring string 0 left)
                            "."
                            (substring string left (- length 1))))))
          (cond ((= length width) result)
                ((< length width)
                 (string-append (make-string (- width length) #\space) result))
                (else (let ((new (substring result 0 width)))
                        (string-set! new (- width 1) #\+)
                        new)))))
      (define (align-word string)
        (let ((length (string-length string)))
          (cond ((= length width) string)
                ((< length width)
                 (string-append string (make-string (- width length) #\space)))
                (else (let ((new (substring string 0 width)))
                        (string-set! new (- width 1) #\+)
                        new)))))
      (if (number? obj)
          (align-number obj width rest)
          (align-word (word->string obj))))))
```

```
(define open-output-file
  (let ((oof open-output-file) (cons cons))
    (lambda (filename)
      (let ((port (oof filename)))
        (set! *the-open-outports* (cons port *the-open-outports*))
        port))))

(define open-input-file
  (let ((oif open-input-file) (cons cons))
    (lambda (filename)
      (let ((port (oif filename)))
        (set! *the-open-inports* (cons port *the-open-inports*))
        port))))

(define remove!
  (let ((null? null?) (cdr cdr) (eq? eq?) (set-cdr! set-cdr!) (car car))
    (lambda (thing lst)
      (define (r! prev)
        (cond ((null? (cdr prev)) lst)
              ((eq? thing (car (cdr prev)))
               (set-cdr! prev (cdr (cdr prev)))
               lst)
              (else (r! (cdr prev)))))
      (cond ((null? lst) lst)
            ((eq? thing (car lst)) (cdr lst))
            (else (r! lst))))))

(define close-input-port
  (let ((cip close-input-port) (remove! remove!))
    (lambda (port)
      (set! *the-open-inports* (remove! port *the-open-inports*))
      (cip port))))

(define close-output-port
  (let ((cop close-output-port) (remove! remove!))
    (lambda (port)
      (set! *the-open-outports* (remove! port *the-open-outports*))
      (cop port))))

(define close-all-ports
  (let ((for-each for-each)
        (close-input-port close-input-port)
        (close-output-port close-output-port))
    (lambda ()
      (for-each close-input-port *the-open-inports*)
      (for-each close-output-port *the-open-outports*)
      'closed)))
```

```
;; Make arithmetic work on numbers in string form:
(define maybe-num
  (let ((string? string?) (string->number string->number))
    (lambda (arg)
      (if (string? arg)
          (let ((num (string->number arg)))
            (if num num arg))
          arg))))

(define logoize
  (let ((apply apply) (map map) (maybe-num maybe-num))
    (lambda (fn)
      (lambda args
        (apply fn (map maybe-num args))))))

;; special case versions of logoize, since (lambda args ...) is expensive
(define logoize-1
  (let ((maybe-num maybe-num))
    (lambda (fn)
      (lambda (x) (fn (maybe-num x))))))

(define logoize-2
  (let ((maybe-num maybe-num))
    (lambda (fn)
      (lambda (x y) (fn (maybe-num x) (maybe-num y))))))

(define strings-are-numbers
  (let ((are-they? #f)
        (real-* *) (real-+ +) (real-- -) (real-/ /) (real-< <)
        (real-<= <=) (real-= =) (real-> >) (real->= >=) (real-abs abs)
        (real-acos acos) (real-asin asin) (real-atan atan)
        (real-ceiling ceiling) (real-cos cos) (real-even? even?)
        (real-exp exp) (real-expt expt) (real-floor floor) (real-align align)
        (real-gcd gcd) (real-integer? integer?) (real-item item)
        (real-lcm lcm) (real-list-ref list-ref) (real-log log)
        (real-make-vector make-vector) (real-max max) (real-min min)
        (real-modulo modulo) (real-negative? negative?)
        (real-number? number?) (real-odd? odd?) (real-positive? positive?)
        (real-quotient quotient) (real-random random) (real-remainder remainder)
        (real-repeated repeated) (real-round round) (real-sin sin)
        (real-sqrt sqrt) (real-tan tan) (real-truncate truncate)
        (real-vector-ref vector-ref) (real-vector-set! vector-set!)
        (real-zero? zero?) (maybe-num maybe-num) (number->string number->string)
        (cons cons) (car car) (cdr cdr) (eq? eq?) (show show) (logoize logoize)
        (logoize-1 logoize-1) (logoize-2 logoize-2) (not not) (whoops whoops))
```

```
(lambda (yesno)
  (cond ((and are-they? (eq? yesno #t))
         (show "Strings are already numbers"))
        ((eq? yesno #t)
         (set! are-they? #t)
         (set! * (logoize real-*))
         (set! + (logoize real-+))
         (set! - (logoize real--))
         (set! / (logoize real-/))
         (set! < (logoize real-<))
         (set! <= (logoize real-<=))
         (set! = (logoize real-=))
         (set! > (logoize real->))
         (set! >= (logoize real->=))
         (set! abs (logoize-1 real-abs))
         (set! acos (logoize-1 real-acos))
         (set! asin (logoize-1 real-asin))
         (set! atan (logoize real-atan))
         (set! ceiling (logoize-1 real-ceiling))
         (set! cos (logoize-1 real-cos))
         (set! even? (logoize-1 real-even?))
         (set! exp (logoize-1 real-exp))
         (set! expt (logoize-2 real-expt))
         (set! floor (logoize-1 real-floor))
         (set! align (logoize align))
         (set! gcd (logoize real-gcd))
         (set! integer? (logoize-1 real-integer?))
         (set! item (lambda (n stuff)
                      (real-item (maybe-num n) stuff)))
         (set! lcm (logoize real-lcm))
         (set! list-ref (lambda (lst k)
                          (real-list-ref lst (maybe-num k))))
         (set! log (logoize-1 real-log))
         (set! max (logoize real-max))
         (set! min (logoize real-min))
         (set! modulo (logoize-2 real-modulo))
         (set! negative? (logoize-1 real-negative?))
         (set! number? (logoize-1 real-number?))
         (set! odd? (logoize-1 real-odd?))
         (set! positive? (logoize-1 real-positive?))
         (set! quotient (logoize-2 real-quotient))
         (set! random (logoize real-random))
         (set! remainder (logoize-2 real-remainder))
         (set! round (logoize-1 real-round))
         (set! sin (logoize-1 real-sin))
         (set! sqrt (logoize-1 real-sqrt))
```

```
        (set! tan (logoize-1 real-tan))
        (set! truncate (logoize-1 real-truncate))
        (set! zero? (logoize-1 real-zero?))
        (set! vector-ref
              (lambda (vec i) (real-vector-ref vec (maybe-num i))))
        (set! vector-set!
              (lambda (vec i val)
                (real-vector-set! vec (maybe-num i) val)))
        (set! make-vector
              (lambda (num . args)
                (apply real-make-vector (cons (maybe-num num)
                                              args))))
        (set! list-ref
              (lambda (lst i) (real-list-ref lst (maybe-num i))))
        (set! repeated
              (lambda (fn n) (real-repeated fn (maybe-num n)))))
       ((and (not are-they?) (not yesno))
        (show "Strings are already not numbers"))
       ((not yesno)
        (set! are-they? #f) (set! * real-*) (set! + real-+)
        (set! - real--) (set! / real-/) (set! < real-<)
        (set! <= real-<=) (set! = real-=) (set! > real->)
        (set! >= real->=) (set! abs real-abs) (set! acos real-acos)
        (set! asin real-asin) (set! atan real-atan)
        (set! ceiling real-ceiling) (set! cos real-cos)
        (set! even? real-even?)
        (set! exp real-exp) (set! expt real-expt)
        (set! floor real-floor) (set! align real-align)
        (set! gcd real-gcd) (set! integer? real-integer?)
        (set! item real-item)
        (set! lcm real-lcm) (set! list-ref real-list-ref)
        (set! log real-log) (set! max real-max) (set! min real-min)
        (set! modulo real-modulo) (set! odd? real-odd?)
        (set! quotient real-quotient) (set! random real-random)
        (set! remainder real-remainder) (set! round real-round)
        (set! sin real-sin) (set! sqrt real-sqrt) (set! tan real-tan)
        (set! truncate real-truncate) (set! zero? real-zero?)
        (set! positive? real-positive?) (set! negative? real-negative?)
        (set! number? real-number?) (set! vector-ref real-vector-ref)
        (set! vector-set! real-vector-set!)
        (set! make-vector real-make-vector)
        (set! list-ref real-list-ref) (set! item real-item)
        (set! repeated real-repeated))
       (else (whoops "Strings-are-numbers: give a #t or a #f")))
       are-they?)))


;; By default, strings are numbers:
(strings-are-numbers #t)
```

D GNU General Public License

The following software license, written by the Free Software Foundation, applies to the Scheme programs in this book. We chose to use this license in order to encourage the free sharing of software—our own and, we hope, yours.

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software—to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

GNU GENERAL PUBLIC LICENSE

TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

a) You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

b) You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

c) If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

a) Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

b) Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

c) Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted

by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

```
<one line to give the program's name
  and a brief idea of what it does.>
Copyright (C) 19yy  <name of author>


This program is free software; you can
redistribute it and/or modify it under the terms
of the GNU General Public License as published
by the Free Software Foundation; either version
2 of the License, or (at your option) any later
version.


This program is distributed in the hope that it
will be useful, but WITHOUT ANY WARRANTY;
without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR
PURPOSE.  See the GNU General Public License for
more details.


You should have received a copy of the GNU
General Public License along with this program;
if not, write to the Free Software Foundation,
Inc., 675 Mass Ave, Cambridge, MA 02139, USA.
```

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

```
Gnomovision version 69,
Copyright (C) 19yy name of author
Gnomovision comes with ABSOLUTELY NO WARRANTY; for
details type 'show w'.  This is free software, and
you are welcome to redistribute it under certain
conditions; type 'show c' for details.
```

The hypothetical commands 'show w' and 'show c' should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than 'show w' and 'show c'; they could even be mouse-clicks or menu items—whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the program, if necessary. Here is a sample; alter the names:

```
Yoyodyne, Inc., hereby disclaims all copyright
interest in the program 'Gnomovision' (which makes
passes at compilers) written by James Hacker.


<signature of Ty Coon>, 1 April 1989
Ty Coon, President of Vice
```

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Library General Public License instead of this License.

Credits

Many of the examples in this book revolve around titles of songs, especially Beatles songs. Since titles aren't covered by copyright, we didn't need official permission for this, but nevertheless we want to acknowledge the debt we owe to the great musicians who've added so much pleasure to our lives. The use of their names here does not mean that they've endorsed our work; rather, we mean to show our respect for them.

The cover and the illustrations on pages 4, 10, 34 (top), 102, 90, 91, 216, and 280 were drawn by Polly Jordan.

The illustration on page 16 was drawn using Mathematica^{TM}.

The illustrations on pages 28, 70, 342, 386, and 438 appear courtesy of the Bettmann Archive.

The illustration on page 40 appears courtesy of the Computer Museum, Boston. Photograph by Ben G.

The quotations on pages 44 and 70 are from *Through the Looking-Glass, and What Alice Found There,* by Lewis Carroll (Macmillan, 1871).

The Far Side cartoons on page 56 are reprinted by permission of Chronicle Features, San Francisco, CA. All rights reserved.

The photograph on page 126 appears courtesy of UCLA, Department of Philosophy.

The photograph on page 146 appears courtesy of the Computer Museum, Boston.

The photograph on page 88 appears courtesy of UPI/Bettmann.

The illustration on page 172 is *Printgallery,* by M. C. Escher. © 1956 M. C. Escher Foundation–Baarn–Holland.

The quotation on page 173 (top) is from "I Know an Old Lady," a traditional song of which a recent version is copyright © 1952, 1954 by Rose Bonne (words) and Alan Mills (music).

The illustration on page 188 is *Drawing Hands,* by M. C. Escher. © 1948 M. C. Escher Foundation–Baarn–Holland.

The illustration on page 234 is reprinted with permission from Michael Barnsley, *Fractals Everywhere,* Academic Press, 1988, page 319.

The illustration on page 248 is reprinted from page 234 of M. Bongard, *Pattern Recognition,* Spartan Books, 1970.

The illustration on page 304 is *Flowering Apple Tree,* by Piet Mondrian, 1912. Courtesy of Collection Haags Gemeentemuseum, The Hague.

The illustration on page 326 is a photograph by Ben G.

The illustration on page 366 is from *The Wizard of Oz,* © 1939. Courtesy of Turner Home Entertainment.

Alphabetical Table of Scheme Primitives

This table does not represent the complete Scheme language. It includes the nonstandard Scheme primitives that we use in this book, and it omits many standard ones that are not needed here.

`'`	Abbreviation for `(quote ...)`.
`*`	Multiply numbers.
`+`	Add numbers.
`-`	Subtract numbers.
`/`	Divide numbers.
`<`	Is the first argument less than the second?
`<=`	Is the first argument less than or equal to the second?
`=`	Are two numbers equal? (Like `equal?` but works only for numbers).
`>`	Is the first argument greater than the second?
`>=`	Is the first argument greater than or equal to the second?
`abs`	Return the absolute value of the argument.
`accumulate`	Apply a combining function to all elements (see p. 108).
`align`	Return a string spaced to a given width (see p. 358).
`and`	(Special form) Are all of the arguments true values (i.e., not `#f`)?
`appearances`	Return the number of times the first argument is in the second.
`append`	Return a list containing the elements of the argument lists.
`apply`	Apply a function to the arguments in a list.
`assoc`	Return association list entry matching key.
`before?`	Does the first argument come alphabetically before the second?
`begin`	(Special form) Carry out a sequence of instructions (see p. 348).
`bf`	Abbreviation for `butfirst`.
`bl`	Abbreviation for `butlast`.
`boolean?`	Return true if the argument is `#t` or `#f`.
`butfirst`	Return all but the first letter of a word, or word of a sentence.
`butlast`	Return all but the last letter of a word, or word of a sentence.
`c...r`	Combinations of `car` and `cdr` (see p. 286).
`car`	Return the first element of a list.

`cdr`	Return all but the first element of a list.
`ceiling`	Round a number up to the nearest integer.
`children`	Return a list of the children of a tree node.
`close-all-ports`	Close all open input and output ports.
`close-input-port`	Close an input port.
`close-output-port`	Close an output port.
`cond`	(Special form) Choose among several alternatives (see p. 78).
`cons`	Prepend an element to a list.
`cos`	Return the cosine of a number (from trigonometry).
`count`	Return the number of letters in a word or number of words in a sentence.
`datum`	Return the datum of a tree node.
`define`	(Special form) Create a global name (for a procedure or other value).
`display`	Print the argument without starting a new line.
`empty?`	Is the argument empty, i.e., the empty word `""` or the empty sentence `()`?
`eof-object?`	Is the argument an end-of-file object?
`equal?`	Are the two arguments the same thing?
`error`	Print an error message and return to the Scheme prompt.
`even?`	Is the argument an even integer?
`every`	Apply a function to each element of a word or sentence (see p. 104).
`expt`	Raise the first argument to the power of the second.
`filter`	Select a subset of a list (see p. 289).
`first`	Return first letter of a word, or first word of a sentence.
`floor`	Round a number down to the nearest integer.
`for-each`	Perform a computation for each element of a list.
`if`	(Special form) Choose between two alternatives (see p. 71).
`integer?`	Is the argument an integer?
`item`	Return the *n*th letter of a word, or *n*th word of a sentence.
`keep`	Select a subset of a word or sentence (see p. 107).
`lambda`	(Special form) Create a new procedure (see Chapter 9).
`last`	Return last letter of a word, or last word of a sentence.
`length`	Return the number of elements in a list.
`let`	(Special form) Give temporary names to values (see p. 95).
`list`	Return a list containing the arguments.
`list->vector`	Return a vector with the same elements as the list.
`list-ref`	Select an element from a list (counting from zero).
`list?`	Is the argument a list?
`load`	Read a program file into Scheme.
`log`	Return the logarithm of a number.
`make-node`	Create a new node of a tree.
`make-vector`	Create a new vector of the given length.
`map`	Apply a function to each element of a list (see p. 289).
`max`	Return the largest of the arguments.
`member`	Return subset of a list starting with selected element, or `#f`.
`member?`	Is the first argument an element of the second? (see p. 73).
`min`	Return the smallest of the arguments.

`newline`	Go to a new line of printing.
`not`	Return `#t` if argument is `#f`; return `#f` otherwise.
`null?`	Is the argument the empty list?
`number?`	Is the argument a number?
`odd?`	Is the argument an odd integer?
`open-input-file`	Open a file for reading, return a port.
`open-output-file`	Open a file for writing, return a port.
`or`	(Special form) Are any of the arguments true values (i.e., not `#f`)?
`procedure?`	Is the argument a procedure?
`quote`	(Special form) Return the argument, unevaluated (see p. 57).
`quotient`	Divide numbers, but round down to integer.
`random`	Return a random number ≥ 0 and smaller than the argument.
`read`	Read an expression from the keyboard (or a file).
`read-line`	Read a line from the keyboard (or a file), returning a sentence.
`read-string`	Read a line from the keyboard (or a file), returning a string.
`reduce`	Apply a combining function to all elements of list (see p. 290).
`remainder`	Return the remainder from dividing the first number by the second.
`repeated`	Return the function described by $f(f(\cdots(f(x))))$ (see p. 113).
`round`	Round a number to the nearest integer.
`se`	Abbreviation for `sentence`.
`sentence`	Join the arguments together into a big sentence.
`sentence?`	Is the argument a sentence?
`show`	Print the argument and start a new line.
`show-line`	Show the argument sentence without surrounding parentheses.
`sin`	Return the sine of a number (from trigonometry).
`sqrt`	Return the square root of a number.
`square`	Not a primitive! `(define (square x) (* x x))`
`trace`	Report on all future invocations of a procedure.
`untrace`	Undo the effect of `trace`.
`vector`	Create a vector with the arguments as elements.
`vector->list`	Return a list with the same elements as the vector.
`vector-length`	Return the number of elements in a vector.
`vector-ref`	Return an element of a vector (counting from zero).
`vector-set!`	Replace an element in a vector.
`vector?`	Is the argument a vector?
`vowel?`	Not a primitive! `(define (vowel? x) (member? x '(a e i o u)))`
`word`	Joins words into one big word.
`word?`	Is the argument a word? (Note: numbers are words.)
`write`	Print the argument in machine-readable form (see p. 400).

Glossary

ADT: See *abstract data type.*

a-list: Synonym for *association list.*

abstract data type: A *type* that isn't provided automatically by Scheme, but that the programmer invents. In order to create an abstract data type, a programmer must define *selectors* and *constructors* for that type, and possibly also *mutators.*

abstraction: An approach to complex problems in which the solution is built in layers. The structures needed to solve the problem (algorithms and data structures) are implemented using lower-level capabilities and given names that can then be used as if they were primitive facilities.

actual argument expression: An expression that produces an actual argument value. In `(+ 2 (* 3 5))`, the subexpression `(* 3 5)` is an actual argument expression, since it provides an argument for the invocation of `+`.

actual argument value: A value used as an argument to a procedure. For example, in the expression `(+ 2 (* 3 5))`, the number `15` is an actual argument value.

aggregate: An object that consists of a number of other objects. For example, a sentence is an aggregate whose elements are words. Lists and vectors are also aggregates. A word can be thought of, for some purposes, as an aggregate whose elements are one-letter words.

algorithm: A method for solving a problem. A computer program is the expression of an algorithm in a particular programming language; the same algorithm might also be expressed in a different language.

apply: To *invoke* a procedure with arguments. For example, "Apply the procedure `+` to the arguments `3` and `4`."

argument: A datum provided to a procedure. For example, in `(square 13)`, `13` is the argument to `square`.

association list: A list in which each element contains a *name* and a corresponding *value.* The list is used to look up a value, given a name.

atomic expression: An expression that isn't composed of smaller pieces.

backtracking: A programming technique in which the program tries one possible solution to a problem, but tries a different solution if the first isn't successful.

base case: In a recursive procedure, the part that solves the smallest possible version of the problem without needing a recursive invocation.

body: An expression, part of the definition of a procedure, that is evaluated when that procedure is invoked. For example, in

```
(define (square x)
  (* x x))
```

the expression `(* x x)` is the body of the `square` procedure.

Boolean: The value `#t`, meaning "true," or `#f`, meaning "false."

branch node: A *tree node* with *children.* The opposite of a *leaf node.*

bug: An error in a program. This word did *not* originate with Grace Hopper finding an actual insect inside a malfunctioning computer; she may have done so, but the terminology predates computers by centuries.

call: Synonym for *invoke.*

cell: One location in a *spreadsheet.*

children: The *nodes* directly under this one, in a *tree.* (See also *siblings* and *parent.*)

composition of functions: Using the value returned by a function as an argument to another. In the expression `(+ 2 (* 3 5))`, the value returned by the `*` function is used as an argument to the `+` function.

compound expression: An expression that contains subexpressions. Opposite of *atomic expression.*

compound procedure: A procedure that a programmer defines. This is the opposite of a *primitive* procedure.

constructor: A procedure that returns a new object of a certain type. For example, the `word` procedure is a constructor that takes words as arguments and returns a new word. See also *selector, mutator,* and *abstract data type.*

data abstraction: The invention of *abstract data types.*

data structure: A mechanism through which several pieces of information are combined into one larger unit. The most appropriate mechanism will depend on the ways in which the small pieces are used in the program, for example, sequentially or in arbitrary order.

database program: A program that maintains an organized collection of data, with facilities to modify or delete old entries, add new entries, and select certain entries for display.

datum: The piece of information stored in each node of a tree.

debugging: The process by which a programmer finds and corrects mistakes in a program. No interesting program works the first time; debugging is a skill to develop, not something to be ashamed of.

destructive: A destructive procedure is one that modifies its arguments. Since the only data type in this book that can be modified is the vector, all destructive procedures call `vector-set!`.

domain: The set of all legal arguments to a function. For example, the domain of the `count` function is the set of all sentences and all words.

effect: Something a procedure does other than return a value. For example, a procedure might create a file on disk, or print something to the screen, or change the contents of a vector.

empty sentence: The sentence `()`, which has no words in it.

empty word: The word `""`, which has no letters in it.

end-of-file object: What the file-reading procedures return if asked to read a file with no more unread data.

expression: The representation in Scheme notation of a request to perform a computation. An expression is either an *atomic expression,* such as `345` or `x`, or a *compound expression* consisting of one or more subexpressions enclosed in parentheses, such as `(+ 3 4)`.

field: A single component of a database *record.* For example, "title" is a field in our example database of albums.

first-class data: Data with the following four properties:

- It can be the argument to a procedure.
- It can be the return value from a procedure.
- It can be given a name.
- It can be part of a data aggregate.

In Scheme, words, lists, sentences, trees, vectors, ports, end-of-file objects, Booleans, and procedures are all first-class.

forest: A list of *trees.*

formal parameter: In a procedure definition, the name given to refer to an argument. In

```
(define (square x)
  (* x x))
```

`x` is the formal parameter. (Note that this is not the same thing as an actual argument! When we invoke `square` later, the argument will be a number, such as 5. The parameter is the *name* for that number, not the number itself.)

function: A transformation of information that associates a *return value* with some number of *argument values*. There may be many different *algorithms* that compute the same function; the function itself is the relationship between argument values and return value, no matter how it may be implemented.

functional programming: A style of programming in which programs are expressed as compositions of functions, emphasizing their arguments and return values. Compare to *sequential programming*.

global variable: A variable created with `define`, which has meaning everywhere in the program. The opposite of a *local variable*.

helper procedure: A procedure that exists to help another procedure do its work. Normally, a user does not invoke a helper procedure directly. Instead, the user invokes a top-level procedure, which invokes the helper procedure to assist it in coming up with the answer.

higher-order procedure: A procedure whose domain or range includes other procedures.

index: A number used to select one of the elements of a vector.

initialization procedure: A procedure that doesn't do any work except to invoke a *helper procedure* with appropriate argument values.

interactive: An interactive program or programming language does its work in response to messages typed by the user at a keyboard (or perhaps indicated with a pointing device like a mouse). Each message from the user causes the program to respond in some way. By contrast, a non-interactive program works with input data that have been prepared in advance.

invoke: To ask a procedure to do its work and come up with a return value. For example, "Invoke the `+` procedure," or "Invoke the `+` procedure with the arguments `3` and `4`."

keyword: The name of a *special form*.

kludge: A method that gets the job done but isn't very elegant. Usually the result is a program that can't be extended the next time a new feature is needed.

leaf node: A *tree node* with no *children.* The opposite of a *branch node.*

leap of faith: A method for understanding recursion in which you say to yourself, "I'm going to assume that the recursive call always returns the right answer," and then use the answer from the recursive call to produce the answer to the entire problem.

list: A data aggregate containing elements that may be of any type.

local variable: A variable that associates a formal parameter name with an actual argument value. It's "local" because the variable exists only within one procedure invocation. (This includes variables created by `let`.) This is the opposite of a *global variable.*

mutable: A data structure is mutable if its contents can change.

mutator: A procedure that changes the value of a data object. In this book, the only mutable data objects we use are vectors, so every mutator is implemented using `vector-set!`. See also *selector, constructor,* and *abstract data type.*

mutual recursion: The program structure in which one procedure invokes another, and the second invokes the first.

node: An element of a *tree.* A node has a *datum* and zero or more *children.*

parent: The node above this one, in a *tree.* (See also *children* and *siblings.*)

pattern matcher: A program that takes a pattern and a piece of data as inputs and says whether or not that piece of data is one that the pattern describes. We present a pattern matcher in Chapter 16.

plumbing diagram: A pictorial representation of the composition of functions, with the return value from one procedure connected to an argument intake of another.

port: An object that Scheme uses to keep track of a file that is currently open for reading or writing.

portable: A portable program is one that can be run in more than one version of Scheme or on more than one computer.

potsticker: A Chinese dumpling stuffed with meat and vegetables, first steamed and then pan-fried, or sometimes first pan-fried and then simmered in water added to the pan.

predicate: A procedure that always returns a *Boolean* value. By convention, Scheme predicates have names like "`equal?`" that end in a question mark.

primitive procedure: A procedure that is already defined when a Scheme session begins. By contrast, a *compound* procedure is one that the programmer defines in Scheme.

procedure: The expression of an algorithm in Scheme notation.

prompt: A character or characters that an interactive program prints to tell the user that it's ready for the user to type something. In many versions of Scheme, the prompt is a `>` character.

random access: A data structure allows random access if the time required to locate an element of the structure is independent of its position within the structure.

range: The set of all possible return values from a function. For example, the range of the `count` function is the set of non-negative integers.

read-eval-print loop: The overall structure of a Scheme interpreter. It *reads* an expression from the keyboard, *evaluates* the expression by invoking procedures, etc., and *prints* the resulting value. The same process repeats forever.

record: One complete entry in a database. For example, one album in our database of albums. A record contains several *fields.*

recursion: Solving a big problem by reducing it to smaller problems of the same kind. If something is defined recursively, then it's defined in terms of itself. See *recursion.*

recursive case: In a recursive procedure, the part that requires a recursive invocation. The opposite of the *base case.*

rest parameter: A parameter that represents a variable number of arguments. In the formal parameter list `(a b . x)`, `x` is a rest parameter.

result replacement: A technique people can use to figure out the value of a complicated Scheme expression by rewriting the expression repeatedly, each time replacing some small subexpression with a simpler expression that has the same value, until all that's left is a single quoted or self-evaluating value.

robust: Able to function despite user errors. Robust programs check for likely errors and recover from them gracefully.

root node: The *node* at the very top of a *tree.*

selector: A procedure that takes an object as its argument and returns some part of that object. For example, the selector `first` takes a word or sentence as argument and returns the first letter of the word or first word of the sentence. See also *constructor, mutator,* and *abstract data type.*

self-evaluating: An expression is self-evaluating if, when evaluated, it has as its value the expression itself. Numbers, Booleans, and strings are the only self-evaluating objects we use in this book.

semipredicate: A procedure that answers a yes-no question by returning `#f` for "no," but instead of returning `#t` for "yes," it returns some additional piece of information. The primitive `member` procedure is a good example of a semipredicate. ("Semipredicate" isn't a common term; we made it up for this book.)

sequencing: Evaluating two or more expressions one after the other, for the sake of their *effects.*

sequential programming: A style of programming in which programs say, "First do this, then do that, then do that other thing." (Compare to *functional programming.*)

siblings: Two *nodes* of a *tree* that are the children of the same node. (See also *children* and *parent.*)

side effect: See *effect.*

special form: A Scheme expression that begins with a *keyword* and is evaluated using a special rule. In particular, some of the subexpressions might not be evaluated. The keywords used in this book are `and`, `begin`, `cond`, `define`, `if`, `lambda`, `let`, `or`, and `quote`. (The keyword itself is also sometimes called a special form.)

spreadsheet program: A program that maintains a two-dimensional display of data can compute some elements automatically, based on the values of other elements.

state: A program's memory of what has happened in the past.

string: A *word* delimited by double-quote marks, such as `"A Hard Day's Night"` or `"000123"`.

structured list: A list with *sublists.*

subexpression: An element of a *compound expression.* For example, the expression `(+ (* 2 3) 4)` has three subexpressions: `+`, `(* 2 3)`, and `4`.

sublist: An element of a list that is itself a smaller list. For example, `(c d)` is a sublist of the list `(a b (c d) e)`.

substitution model: The way we've explained how Scheme evaluates function invocations. According to the substitution model, when a compound procedure is invoked, Scheme goes through the body of that procedure and replaces every copy of a formal parameter with the corresponding actual argument value. Then Scheme evaluates the resulting expression.

subtree: A tree that is part of a larger tree.

symbol: A word that isn't a number or a string.

symbolic computing: Computing that is about words, sentences, and ideas instead of just numbers.

tree: A two-dimensional data structure used to represent hierarchical information.

tree recursion: A form of recursion in which a procedure calls itself recursively more than one time in each level of the recursion.

type: A category of data. For example, words, sentences, Booleans, and procedures are types. Some types overlap: All numbers are also words, for example.

variable: A connection between a name and a value. Variables can be *global* or *local.*

vector: A primitive data structure that is mutable and allows random access.

word: A sequence of characters, including letters, digits, or punctuation. Numbers are a special case of words.

Index of Defined Procedures

This index contains example procedures whose definitions are in the text and procedures that you are asked to write as exercises. (The exercises are marked as such in the index.) Other sources of information are the general index, which contains technical terms and primitive procedures (for which there is no Scheme definition); the glossary, which defines many technical terms; and the Alphabetical Table of Scheme Primitives on page 553.

A

`abs` 74
`accumulate` 332
`acronym` 8, 9, 110, 223
`add` 269
`addup` 221
`add-field` (exercise) 490
`add-move` 353
`add-numbers` 109, 222
`add-three` 127
`add-three-to-each` 127
`all-evaluated?` 455
`already-won?` 353
`already-won?` (exercise) 169
`always-one` 109
`amazify` (exercise) 122
`american-time` (exercise) 84
`any-numbers?` 115
`aplize` (exercise) 140
`apl-sqrt` (exercise) 140
`appearances` (exercise) 139
`arabic` (exercise) 205
`area` 328
`arg-count` 370
`ask-for-name` 362
`ask-question` 364
`ask-user` 354
`ask` (exercise) 484
`average` 46

B

`backwards` 135
`base-grade` (exercise) 124, 204
`beatle-number` 117
`before-in-list?` (exercise) 301
`best-free-square` 166
`best-move` 164
`best-square` 164
`best-square-helper` 165
`bill` (exercise) 420
`bottles` 344
`bound-check` 449

branch (exercise) 302
bridge-val (exercise) 144
butfirst 288
butlast 288
buzz 74

C

card-list 411
card-val (exercise) 142
cell-children 461
cell-expr 460
cell-parents 461
cell-structure 460
cell-value 460
char-count 395
children 315
choices 12
choose-beatles (exercise) 122
choose-win 158
circle-area 134, 327
circumference 134
cities 309
clear-current-db! (exercise) 486
combinations 13
command-loop 441
common-words (exercise) 138
compose (exercise) 139
compute 323
concatenate (exercise) 401
converse (exercise) 364
copies 228
copies (exercise) 187
count 110
countdown (exercise) 187
count-adjacent-duplicates
(exercise) 230
count-db (exercise) 485
count-leaves 310
count-leaves-in-forest 310
count-nodes (exercise) 324
count-suit (exercise) 143
count-ums (exercise) 124, 185
current-db (exercise) 483
current-fields (exercise) 483

D

datum 315
db-fields (exercise) 482
db-filename (exercise) 482
db-insert (exercise) 484
db-records (exercise) 483
db-set-fields! (exercise) 482
db-set-filename! (exercise) 482
db-set-records! (exercise) 483
deep-appearances 297
deep-map 335
deep-pigl 297, 334
depth (exercise) 324
describe-time (exercise) 87
describe-time (exercise) 205
describe (exercise) 137
differences (exercise) 230
disjoint-pairs 219
divisible? 75
double 105, 114
doubles 220
down 201
downup 174, 178, 195

E

earliest-word 236
echo 354
edit-record (exercise) 485
effect 346
ends-e? 107
ends-vowel? (exercise) 122
ends (exercise) 68
european-time (exercise) 84
evens 195
even-count? (exercise) 122
every 329
every-nth 224
every-nth-helper 224, 228
exaggerate (exercise) 123, 204
execute-command 443
exhibit 443
explode 181
extract-digit 161

extract-ids 454
extract-spaces 395

F

factorial 14, 194
fib 213
figure 455
filemerge 397
filemerge-helper 398
file-map 392
file-map-helper 392
fill-array-with-rows 461
fill-row-with-cells 462
filter 331
find-triples 154
first 288
first-choice 166
first-if-any 160
first-last (exercise) 139
first-letters 104
first-number 225
first-two (exercise) 67
flatten (exercise) 302
flip 130
fourth-power 132
from-binary 237, 238
functions-loop 367

G

generic-before? (exercise) 489
gertrude (exercise) 98
get-arg 377
get-args 368
get-fn 378
get-record-loop (exercise) 484
get-record (exercise) 484
get-song 390
get-value 269
get (exercise) 486
global-array-lookup 460
gpa (exercise) 124, 204
greet 71
greet (exercise) 87

H

hand-dist-points (exercise) 144
hang-letter (exercise) 138
hang (exercise) 138
has-vowel? 222
hexagon-area 328
high-card-points (exercise) 142
hyphenate 108
hypotenuse 50, 91

I

increasing? 292
indef-article (exercise) 85
initialize-lap-vector 408
initials (exercise) 187
init-array 461
insert-and (exercise) 68
insert (exercise) 484
integer-quotient 77
in-domain? 371
in-forest? 313
in-tree? 312, 313
item 115
i-can-advance? 164
i-can-fork? 159
i-can-win? 157

J

join (exercise) 403
justify 395

K

keeper 136
keep-h 136
knight (exercise) 68

L

lap 409
last 288
lastfirst 393

`leader` (exercise) 420
`leaf` 309
`leaf?` 310
`letterwords` (exercise) 138
`letter-count` (exercise) 123, 229
`letter-pairs` 181, 218
`let-it-be` (exercise) 137
`list->vector` 414
`list-db` (exercise) 485
`lm-helper` 268
`load-db` (exercise) 485
`locate` 314
`locate-in-forest` 314
`location` 352
`location` (exercise) 230
`longest-match` 267
`lookup` 269, 286
`lookup` (exercise) 402
`lots-of-effect` 347
`lots-of-value` 347

M

`make-adder` 129
`make-db` (exercise) 482
`make-deck` 412
`make-node` 314
`map` 330
`match` 266
`match-special` 267
`match-using-known-values` 267
`max2` (exercise) 301
`maybe-display` 355
`member-types-ok?` 371
`merge` 239
`mergesort` 238
`merge-copy` 398
`merge-db` (exercise) 495
`merge` (exercise) 231
`middle-names` (exercise) 69
`music-critic` 358
`mystery` (exercise) 300
`my-pair?` 156
`my-single?` 160

N

`named-every` 373
`named-keep` 373
`name-table` 360
`new-db` (exercise) 483
`no-db?` (exercise) 483
`numbers` (exercise) 204
`number-name` (exercise) 233
`number-of-arguments` 375
`num-divisible-by-4?` 76

O

`odds` (exercise) 229
`one-half` 239
`opponent` 156
`opponent-can-win?` 158
`order` 281
`order` (exercise) 420
`other-half` 239

P

`pad` 396
`page` (exercise) 402
`parse` 322
`parse-scheme` (exercise) 325
`phone-spell` (exercise) 244
`phone-unspell` (exercise) 124, 186
`pi` 134
`pigl` 10, 181
`pin-down` 448
`pin-down-cell` 450
`pivots` 160
`play-ttt` 352
`play-ttt-helper` 352
`plural` 75, 105
`plural` (exercise) 87
`poker-value` (exercise) 245
`praise` 285
`prepend-every` 12, 240
`prev-row` 443
`print-file` 390
`print-file-helper` 397

print-position 355
print-row 355
print-screen 458
process-command 442
process-grades 393
progressive-squares? (exercise) 231
prune (exercise) 324
put 445
put-all-cells-in-col 445
put-all-cells-in-row 445
put-all-helper 445
put-expr 453
put-formula-in-cell 447

Q

query (exercise) 69
quoted? 457
quoted-value 457

R

real-accumulate 332
real-word? 9, 108
real-words (exercise) 205
remdup (exercise) 229
remove-adjacent-duplicates (exercise) 230
remove-once 236
remove-once (exercise) 229
remove (exercise) 205
repeated-numbers 161
roman-value 79
roots 96
rotate 11

S

safe-pigl 222
safe-sqrt (exercise) 140
same-arg-twice 129
save-db (exercise) 485
scheme-procedure 370
scrunch-words 221
second 60
second (exercise) 139
select-by (exercise) 493
select-id! 444
sentence-version (exercise) 138
sentence (exercise) 301
sent-before? 227
sent-equal? 252
sent-max 221
sent-of-first-two 106
setvalue 455
set-cell-children! 461
set-cell-expr! 461
set-cell-parents! 461
set-cell-value! 460
set-current-db! (exercise) 483
set-selected-row! 443
show-addition 350
show-and-return 361
show-answer 368
show-list 351
shuffle! 412
sign (exercise) 84
skip-songs 390
skip-value 269
sort 236
sort2 (exercise) 86
sort-digits 162
sort-on (exercise) 490
sort (exercise) 487
spaces 396
spell-number (exercise) 204
spell-digit 116
sphere-area 327
sphere-surface-area 134
sphere-volume 134
spreadsheet 461
square 41, 131
square-area 327
ss-eval 456
stupid-ttt 352
subsets 241
substitute-letter 153
substitute-triple 154
substitute (exercise) 139
substring? (exercise) 243

substrings (exercise) 243
subword 355
subword (exercise) 124
suit-counts (exercise) 143
suit-dist-points (exercise) 143
sum-square (exercise) 99
sum-vector (exercise) 419
superlative (exercise) 98
syllables (exercise) 232

T

teen? (exercise) 85
third-person-singular (exercise) 83
third (exercise) 67
thismany (exercise) 86
three-firsts 103
tie-game? 353
tie-game? (exercise) 169
transform-beatles (exercise) 122
translate 286, 292
truefalse 80
true-for-all? (exercise) 124
true-for-all-pairs? (exercise) 337
true-for-any-pair? (exercise) 337
try-putting 445
ttt 148, 155
ttt-choose 163
two-firsts 103
two-first-sent (exercise) 68
two-first (exercise) 68
two-numbers? 371
type-of (exercise) 85
type-check (exercise) 140
type-predicate 370

U

unabbrev (exercise) 139
unscramble (exercise) 244
up (exercise) 229
utensil (exercise) 84

V

valid-date? (exercise) 86
valid-fn-name? 375
valid-infix? (exercise) 303
value 346
vector-append (exercise) 419
vector-fill! (exercise) 419
vector-map! (exercise) 419
vector-map (exercise) 419
vector-swap! 412
verse 345, 350
vowel? 107

W

who (exercise) 137
words (exercise) 123

General Index

This index contains technical terms and primitive procedures. Other sources of information are the index of defined procedures, which contains procedures whose definitions are in the text and procedures that you are asked to write as exercises; the glossary, which defines many technical terms; and the Alphabetical Table of Scheme Primitives on page 553.

`#f` 71
`#t` 71
`'` 58
`*` 553
`+` 553
`-` 74, 553
`/` 553
`<` 73
`<=` 73
`=` 73
`>` 73
`>=` 73

A

Abelson, Harold xxi, xxxi, 209, 501
`abs` 74
abstract data type 270, 287, 315, 441
abstraction 5, 47, 134, 270, 434, 501
`accumulate` 108, 110, 331
actual argument 45
actual argument expression 45
actual argument value 45
ADT 270, 287, 315, 441
algorithm 13, 238
`align` 358
`and` 75, 76
apostrophe 58
`append` 283
`apply` 293
argument 6, 32
argument, actual 45
arguments, variable number of 292
arithmetic function 18
array 406, 421
artificial intelligence xviii
`assoc` 291
association list 291
atomic expression 29

B

backtracking 256, 270
base case 178
base cases, simplifying 197
Baskerville, John iv

Beatles xxxii, 551
before? 73
begin 348
bf 61
binary number 237
bl 61
body 42
Bonne, Rose 552
Boole, George 21
Boolean 21, 71
boolean? 73
boring exercise xxiii
bottom-up 142
branch node 305
bridge points 141
butfirst 60
butlast 60

C

c 347
c...r 286
cadr 287
car 282
Carroll, Lewis 45, 551
case, base 178
case, recursive 178
cdr 282
ceiling 554
cell 425
chalkboard model 93, 94
child (in spreadsheet program) 451
children 306
children 308
Chinese food xxxii
Clancy, Michael xxxi
clause, cond 79
Clinger, William 394
close-all-ports 401
close-input-port 388
close-output-port 388
comments 32
complexity, control of 5
composition of functions 26, 47
compound expression 29
compound procedure 8
computing, symbolic xviii, 14
cond 78, 157
cond clause 79
condition 79
cons 283
consequent 79
constant, named 89
constructor 61, 282, 283, 307
control of complexity 5
conversational program 343
cos 554
count 109

D

data abstraction violation 316
data file 387
data structure 149
data type, abstract 270, 287, 315, 441
database 265, 477
datum 306
datum 308
Dave Dee, Dozy, Beaky, Mick, and Tich xxxii
debugger 7
debugging xxiii
Dee, Dave xxxii
define 41, 130
definition, global 131
diagram, plumbing 34
Dijkstra, Edsger xvii
display 350, 387
Dodgson, Charles 551
domain 20
double-quote marks 58, 61
Dubinsky, Ed xxxii, xxxi

E

effect, side 345
Ehling, Terry xxxi
else 80
EMACS 429, 436
empty sentence 61, 72

`empty?` 73
end-of-file object 390
engineering, software xx
`eof-object?` 391
`equal?` 72
`error` 554
error messages 7
`eval` 456
evaluation, order of 31
`even?` 554
`every` 104, 110
exclusive, mutually 80
exercise, boring xxiii
exercise, real xxiii
`exit` 7
expression 29
expression, actual argument 45
expression, atomic 29
expression, compound 29
expression, self-evaluating 30, 61, 62
`expt` 554
extensibility 435
extensions to Scheme xxiv, 59, 525

F

factorial 14, 192
false 71
Fibonacci numbers 213
field 477
file, data 387
`file-map` 391
`filter` 289, 331
`first` 60
first-class 63, 113
`floor` 554
food, Chinese xxxii
forest 308
fork 159
form, special 42, 58, 76, 78, 95, 128, 348
formal parameter 45
forms, special 214
formula 426, 431
Fortran 335
`for-each` 351
Free Software Foundation 547
Freud, Sigmund 505
Friedman, Daniel P. xxi, xxxi, 406
Frisell, Bill xxxii
function 17
function as argument 104
function as data 21
function as object 23
function as process 23
function composition 26, 47
function machine 33, 106
function vs. procedure 43, 104
function, arithmetic 18
function, higher-order 23, 106, 289, 327
function, unnamed 133
functional programming 17, 27, 89, 348
`functions` 367

G

generalization 46, 327, 392, 434
global variable 131

H

Harvey, Brian 209
Harvey, Tessa xxxi
helper procedure 224
Hennessy, John L. xxxii
higher-order function 23, 106, 289, 327
Hofstadter, Douglas R. xxxii
Hypercard 435

I

ice cream xxxii
identity element 119, 221, 332
`if` 71, 76
IFSMACSE xxxii
imperative programming 417
indentation 35
indentation in a program 11
Indianapolis 405
infix notation 317
initialization procedure 224

integer? 554
intelligence, artificial xviii
interactive programming 343
interface, user 360
item 114

J

join 402
justify 394

K

Katz, Yehuda xxxi
keep 107, 110
keep pattern 220
keyboard 343
keyword 42
kludge 157, 162, 165, 333
Knuth, Donald iv

L

lambda 127
lambda calculus 128
last 60
Latin, Pig 10, 179
leaf node 305
length 291
Leron, Uri xxxii
let 95
lines of a program 10
Lisp xix
list 281
list 283
list, association 291
list, structured 282, 335
list? 290
list->vector 411
list-ref 291
little people 30, 49, 90, 207
load 554
local variable 93
log 554
Logo xxxi

M

machine, function 33, 106
make-node 307
make-vector 406
Manilow, Barry 57
map 289
mapping 289
Marx, Karl 505
matcher, pattern 249
matrix 421
max 554
McCarthy, John xix
member 290
member? 73, 82
mergesort 238
metaphor 434
Mills, Alan 552
min 554
model, chalkboard 93
moon, phase of the 212
mutator 407
mutual recursion 310
mutually exclusive 80

N

named constant 89
naming a value 89, 130
newline 350, 387
node 305
node, branch 305
node, leaf 305
node, root 305
not 75
null? 283
number, binary 237
number? 73
numbers, Fibonacci 213

O

object, end-of-file 390
odd? 555
open-input-file 388

`open-output-file` 388
`or` 76
order of evaluation 31

P

parameter, formal 45
parameter, rest 293
parent 306
parent (in spreadsheet program) 451
parentheses 36
parentheses, for procedure invocation 6, 32, 119
parentheses, for `cond` clauses 79
parentheses, for `let` variables 96
Pascal xxi
pattern matcher 249
pattern, recursive 217
pattern: `keep` 220
Patterson, David A. xxxii
Payne, Jonathan iv
phase of the moon 212
Pig Latin 10, 179
Pisano, Leonardo 213
pivot 159
placeholder 250
plumbing diagram 34
points, bridge 141
port 387
portable 428
position 148
predicate 72
prefix notation 317
primitive procedure 8
printed twice, return value 211
printing 343, 362
Prior, Robert xxxi
procedure 6
procedure as argument 104
procedure vs. function 43, 104
procedure, compound 8
procedure, helper 224
procedure, higher-order 327
procedure, initialization 224
procedure, primitive 8
`procedure?` 555
program, conversational 343
programming, functional 17, 27, 89, 348
programming, imperative 417
programming, interactive 343
programming, structured xx
programming, symbolic xviii, 14
prompt 6

Q

quadratic formula 94
quotation marks, double 58
`quote` 57
`quotient` 555

R

`random` 410, 555
range 20
`read` 354, 387
read-eval-print loop 29, 343, 367
`read-string` 396
`read-line` 356, 387
real exercise xxiii
record 477
recursion 174, 577
recursion, mutual 310
recursion, tree 312
recursive case 178
recursive pattern 217
`reduce` 290, 332
Rees, Jonathan 394
`remainder` 555
`repeated` 113
replacement, result 33
representation 150
rest parameter 293
result replacement 33
return 17
robust 334
Rogers, Annika xxxi
root node 305
`round` 555

S

Scheme xix, xviii, 394
Scheme, extensions to xxiv, 59, 525
screen 343
`se` 62
selector 59, 282, 308
self-evaluating expression 30, 61, 62
semicolon 32
semipredicate 77, 291
sentence 21
`sentence` 61
sentence, empty 61, 72
`sentence?` 74
sequencing 347
`show` 344, 387, 388
`show-line` 358, 387
shuffle 410
sibling 306
side effect 345
simplifying base cases 197
`sin` 555
software engineering xx
sorting 235, 238
special form 42, 58, 76, 78, 95, 128, 348
special forms 214
spreadsheet 425
Springer, George xxi
`sqrt` 555
state 405
strategy 148
string 36, 58, 61, 154, 350
Structure and Interpretation of Computer Programs xxi, 501
structure, data 149
structured list 282, 335
structured programming xx
subexpression 29
sublist 282
subsets 239
substitution 48, 94
substitution model 94
substitution model and global variables 131
subtree 307
Sussman, Gerald Jay xxi, xxxi, 209, 501
Sussman, Julie xxi, xxxi, 209, 501
symbol 58
symbolic programming xviii, 14

T

template 217
top-down 142
`trace` 210
tree 305
tree recursion 312
triple 150
true 71
type 19
type, abstract 270, 287, 315, 441

U

unnamed function 133
`untrace` 211
user interface 360

V

value, actual argument 45
variable 89
variable number of arguments 292
variable, global 131
variable, local 93
vector 406
`vector` 413
`vector-fill!` 419
`vector-length` 413
`vector?` 413
`vector->list` 411
`vector-ref` 407
`vector-set!` 407

W

Wirth, Niklaus xxi
word 19
`word` 61
word, empty 61

`word?` 74
Wright, Matthew 209
`write` 400

Z

Zabel, David xxxii

Table of Scheme Primitives by Category

Use this table if you've forgotten the name of a primitive. Then look in the index to find more about how to use the primitive.

Words and Sentences

appearances*
before?*
butfirst (bf)*
butlast (bl)*
count*
empty?*
equal?
first*
item*
last*
member?*
quote
sentence (se)*
sentence?*
word*
word?*

Lists

append
assoc
car
cdr
c...r
cons
filter*
for-each
length
list
list?
list-ref
map
member
null?
reduce*

Trees

children*
datum*
make-node*

* Not part of standard Scheme

Arithmetic

+, -, *, /
<, <=, =, >, >=
abs
ceiling
cos
even?
expt
floor
integer?
log
max
min
number?
odd?
quotient
random
remainder
round
sin
sqrt

True and False

and
boolean?
cond
if
not
or

Variables

define
let

Vectors

list->vector
make-vector
vector
vector?
vector-length
vector->list
vector-ref
vector-set!

Procedures

apply
lambda
procedure?

Higher-Order Procedures

accumulate*
every*
filter*
for-each
keep*
map
reduce*
repeated*

Control

begin
error
load
trace
untrace

Input/Output

align*
display
newline
read
read-line*
read-string*
show*
show-line*
write

Files and Ports

close-all-ports*
close-input-port
close-output-port
eof-object?
open-input-file
open-output-file