1 Data and Operators

...it is not the thing done or made which is beautiful, but the doing. If we appreciate the thing, it is because we relive the heady freedom of making it. Beauty is the by-product of interest and pleasure in the choice of action.

Jacob Bronowski,
The Visionary Eye

Computing is an art form. Some programs are elegant, some are exquisite, some are sparkling. My claim is that it is possible to write grand programs, noble programs, truly magnificent programs.

Donald E. Knuth,
from an article by William Marling
in *Case Alumnus*

1.1 Introduction

Computer programming is many faceted.

It is engineering. Computer programs must be carefully designed. They should be reliable and inexpensive to maintain. Like any other engineering discipline, computer programming has special challenges. The foremost challenge is managing complexity. As programs grow larger, the number of possible interactions between their pieces tends to grow much faster than the volume of code. Abstraction is the primary technique for managing complexity. An abstraction hides unnecessary detail and allows recurring patterns to be expressed concisely. In this book we emphasize several powerful techniques for building abstractions.

It is a craft. A program made with craftsmanship is both more serviceable and more satisfying. Programming requires proficiency born of practice (hence the many exercises in this book!). It requires great dexterity, though of a mental rather than a manual sort. As woodworkers enjoy working with their hands and fine tools, so programmers enjoy exercising their minds and working with a fine programming language.

It is an art. Fine programs are the result of more than routine engineering. They require a refined intuition, based on a sense of style and aesthetics that is both personal and practical. As an artistic medium, programming is highly

plastic, unconstrained by physical reality. In programming, perhaps more than in other arts, less is more. Simplicity is nowhere more practical than in programming, where the bane is complexity. When just the right abstraction for a problem has been found, it may be a thing of beauty. We hope you take pleasure in the programs of this book.

It is not a science, but it is based on one: computer science. Though our primary concern in this book is with the techniques of programming, we will have occasion to introduce a number of important scientific results. We hope you find the language and style of this book to be vehicles for deeper understanding and appreciation throughout your study of computer science.

It is a literary endeavor. Of course, programs must be understood by computers, which requires mastery of a programming language. But that is not enough! Programs must be analyzed to understand their behavior, and most programs must be modified periodically to accommodate changing needs. Thus it is essential that programs be intelligible by humans as well as by computers. The challenge is to convey the necessary details without losing sight of the overall structure. This in turn requires creative use of abstractions and a good sense of style—habits we attempt to instill by example in this book.

But this book is ultimately about more than the craft of engineering artistic and literate programs. Programming teaches an algorithmic (step by well-specified step) approach to problem solving, which in turn encourages an algorithmic approach to gaining knowledge. This view of the world is providing numerous fresh insights in fields as diverse as mathematics, biology, and sociology, as well as providing tools that assist and extend our minds in almost every field of study. Thus programming ability, like mathematical and writing ability, is an asset of universal value.

Programming ability and literary ability have another thing in common. An essay or short story can be correct grammatically and can convey the information that the author intended and still not be a literary work of art. Computer programming has its own aesthetic, and good programmers strive to produce programs that evoke appreciative responses in their readers. Writing such programs requires both inspiration and the application of craftsmanship that employs a thorough command of the programming language and the metaphors it can support.

There are many languages from which to choose when designing a course to teach the principles of programming. Scheme was selected because it is an expressive language that provides powerful abstraction mechanisms for expressing the solutions to computational problems. This facilitates the writing of clear and satisfying programs. It is especially good as a vehicle for teaching

programming because the student is not required to learn unnecessary rules and prohibitions before being able to write meaningful programs.

The programming language LISP (which stands for List Processing) was developed around 1960 by John McCarthy. (See McCarthy, 1960.) Scheme was derived from LISP by Gerald Jay Sussman and Guy Lewis Steele Jr. around 1975. (See Sussman and Steele, 1975.) A number of people have been involved in the evolution of Scheme since its inception and these developers of the language have published a series of reports describing the current state of the language. For the first such report, see Steele and Sussman, 1978. The third revised report appeared in 1986. (See Rees and Clinger, 1986.) The fourth revision is expected in 1989. There is also a working group preparing for an IEEE Standard for Scheme. A number of books about Scheme have appeared since then, including:

- *Structure and Interpretation of Computer Programs* by Abelson and Sussman with Sussman, MIT Press and McGraw-Hill Book Company, 1985.
- *The Little LISPer* by Friedman and Felleisen, MIT Press, 1987 and SRA Pergamon, 1989.
- *The Scheme Programming Language*, by Dybvig, Prentice-Hall, 1987.
- *Programming in Scheme* by Eisenberg, Scientific Press, 1988.
- *An Introduction to Scheme* by Smith, Prentice-Hall, 1988.

The following two publications are manuals for Scheme that accompany the implementations of Scheme on microcomputers:

- *MacScheme+Toolsmith*™, Semantic Microsystems, 1987.
- *PC Scheme*, by Texas Instruments, Scientific Press, 1988.

We encourage you to read them because each presents its own programming philosophy. We are all using the same language, but we have somewhat different stories to tell.

As you read these pages, remember that you should care how elegant your programs are. The task that confronts you is not only to learn a programming language but to learn to think as a computer scientist and develop an aesthetic about computer programs. Enjoy this as an opportunity to understand the creative process better. Solve problems not only for their solutions but also for an understanding of how the solutions were obtained.

1.2 The Computer

We begin by briefly describing the components of a computer. At this stage, it suffices to think of the computer as being composed of four components:

1. The *input device*, in this case the keyboard with the standard typewriter keys and some additional ones. Each key can perform several functions. On both the typewriter and the computer keyboard, we choose between lower and upper case by depressing the Shift key. On the computer, we can also hold down the Control (CTRL) key while pressing another key to get another behavior, and on some computers, we can similarly hold down the Alternate (ALT) key while pressing another key to get yet another behavior. Finally, pressing and releasing the Escape (ESC) key before pressing another key gives still another behavior. When a key is pressed, the result is usually shown on the screen.
2. The *processor*, in which the computing is done. This contains the internal memory of the computer, the arithmetic logic unit, and the registers where the computations take place.
3. The *output devices*: the video monitor on which the interactive computing is viewed, which we refer to as the screen, and the printer where printed copy of the output is produced.
4. The *external storage device*. In microcomputers, this often consists of two floppy disk drives. The user places diskettes into these drives and either reads files from a diskette into the computer's internal memory or writes from the internal memory to a file on a diskette. Many microcomputers and all larger computers have an internal disk on which files can be stored and accessed.

Implementations of Scheme are available on a wide variety of computers ranging from larger mainframe computers that support many users to individual workstations or personal computers.

1.3 Numbers and Symbols

In order to make a computer do something for us, we must communicate with the computer in a language that it "understands." The English language, which we are using for our communication in this paragraph, makes use of words and certain grammatical rules that enable us to combine words into

sentences. The words themselves consist of certain strings of characters, that is, characters written one after the other with no blank spaces between them. The computer languages also have their analogs of words, which we call *symbols*. The characters used to make up the symbols are the same characters on a standard typewriter keyboard, with a few additions and deletions. We shall generally use the letters of the alphabet, the digits from 0 through 9, and some of the other characters on the keyboard. A few of the other characters on the keyboard have special meaning, just as certain characters like the period and comma have special meaning in English. In Scheme, the characters

```
( ) [ ] { } ; , " ' ` # \
```

have special meaning and cannot appear in symbols. Similarly, the characters

```
+ - .
```

are used in numbers and may occur anywhere in a symbol except as the first character. The following list contains examples of symbols in Scheme:

```
abcd r cdr p2q4 bugs? one-two *now&
```

Numbers are not considered to be symbols in Scheme; they form a separate category. Thus, as you would expect, `10`, `-753`, and `31.5` are Scheme numbers. In the English language, not every combination of letters gives us a meaningful word. We keep words that are meaningful in our minds or in a dictionary, and when we see or hear a word, we retrieve its meaning in order to use it. In much the same way, symbols may be assigned some meaning in Scheme. A symbol used to represent some value is called a *variable*. The computer must determine the meaning of each variable or number it is given. It recognizes that the numbers have their usual meaning. Scheme also keeps the meaning of certain variables that have been assigned values, and when it is given a symbol, it checks to see if it is one of those that has been kept. If so, it can use that meaning. Otherwise it tells us that the symbol has not yet been given a meaning.

To carry the analogy with the English language a step further, words are put together in sentences to express the thoughts you want to convey. The Scheme analog of a sentence is an *expression*. An expression may consist of a single symbol or number (or certain other items to be defined later), or a *list*, which is defined to consist of a left parenthesis, followed by expressions separated by blank spaces, and ending with a right parenthesis. We first

discuss the use of expressions involving symbols or numbers, and return to discussing other types of data in Section 1.4.

When you turn on the computer and call up Scheme, you usually get a message telling what implementation of Scheme you are using. Then a *prompt* appears on the screen, prompting you to enter something. The nature of the prompt depends on the implementation you are using. The prompt we use in this book to simulate the output on the screen is a pair of square brackets surrounding a number. Thus the first prompt will be

```
[1]
```

If you type a number after the prompt and then press the <RETURN> key (sometimes referred to as the <ENTER> key),

```
[1] 7 <RETURN>
```

Scheme recognizes that the meaning of the character that you typed is the number 7. We say that the *value* of the character you typed is the number 7 or that what you type has been *evaluated* to give the number 7. Scheme then writes the value of what you type at the beginning of the next line and moves down one more line and prints the next prompt:

```
[1] 7 <RETURN>
7
[2]
```

Let us review what we have just seen. At the first prompt, you enter 7 and press <RETURN>. In general, an expression (or a collection of such expressions) you enter in response to the prompt and before pressing <RETURN> is called a *program*. In this example, Scheme reads your program, evaluates it to the number 7, prints the value 7 on the screen at the beginning of the next line, and then prints the next prompt one line lower. Thus Scheme does three things in succession: it reads, it evaluates, and it prints. We refer to this sequence of events performed by Scheme as its *read-eval-print loop*. After printing the prompt, Scheme waits for you to type the next program. In the example, when you press <RETURN>, Scheme completes one cycle of the loop and begins another.

What happens when a symbol is typed after the prompt? Suppose first that you type the symbol `ten` and press <RETURN>. If Scheme has not previously been given a meaning for the symbol `ten`, we say that `ten` has not been *bound* to a value. In the evaluate phase of the read-eval-print loop, no value is found

for `ten`, and a message is printed informing you that an error was made and describing the nature of the error. For example,

```
[2] ten <RETURN>
Error: variable ten not bound.
```

(The actual message printed depends on the implementation of Scheme you are using.) How then do we assign a meaning or value to a symbol? Suppose we want to assign the value `10` to the symbol `ten`. For this purpose we use a *define expression*. (A define expression is an example of a *special form*: a form of expression identified by a special symbol called a *keyword*, which in this case is `define`.) The define expression is entered after the next prompt as follows:

```
[3] (define ten 10) <RETURN>
```

In this example, Scheme evaluates the third subexpression, which has the value `10`, assigns that value to the symbol `ten`, and finally, in our implementation of Scheme, prints the next prompt. Since the value returned by a define expression is never used, that value is not prescribed in the specification of the language. For convenience in writing this book, we opt to suppress the value returned by a define expression.

Now let's see what happens when we enter the symbol `ten`:

```
[4] ten <RETURN>
10
```

This time, Scheme successfully evaluates the variable `ten`, so it prints the value `10`.

We have seen that a variable is a symbol to which a meaning (i.e., a value) can be given. When a value is given to a variable, we say that the variable is *bound* to that value. In our previous example, the symbol `ten` is a variable bound to the value `10`. In general, if *var* represents a variable and *expr* represents an expression whose value we would like to *bind* or assign to *var*, we accomplish the assignment by writing

(define *var expr*)

The define expression is made up of a keyword, a variable name *var*, and an expression *expr*.

Now let's suppose that `ten` is bound to `10` and we want Scheme to print not the value `10` but instead to print the symbol `ten`. We want to have some way

of telling Scheme not to evaluate **ten** but to print its *literal value* **ten**. The mechanism that Scheme provides for doing this is called *quoting* the symbol. We quote a symbol by enclosing in parentheses the word **quote** followed by the symbol:

(quote *symbol*)

For example, you quote the symbol **ten** by writing **(quote ten)**. If you type **(quote ten)** and then **<RETURN>** in response to a Scheme prompt, you see

```
[5] (quote ten) <RETURN>
ten
```

From now on, we shall omit the **<RETURN>** notation. It is understood that each line that we type must be followed by **<RETURN>**. We use the word *enter* when we want to indicate that something is to be typed in response to the Scheme prompt. The value that Scheme prints in response to what we enter is said to be the value that the expression "evaluates to" or that is "returned." For example, we could have said, "If the symbol **ten** is bound to **10**, and you enter **(quote ten)**, then Scheme evaluates it to **ten**, while if you enter **ten**, Scheme evaluates it to **10**."

In all cases, whether a symbol is bound to some value or not, when a quoted symbol is entered, the literal symbol is returned. Thus if we enter **(quote abc3)**, Scheme returns **abc3**. It is not necessary to quote numbers, for the value of a number as an expression is the number itself.

```
[6] (quote abc3)
abc3
[7] (quote 12)
12
```

An object whose value is the same as the object itself is called a *constant*. At this point, the only constants we have seen are numbers.

It is somewhat inconvenient to have to type so much each time we want to quote a symbol, so an abbreviation for the quoting process is also available in Scheme. In order to quote a symbol, we need only place an apostrophe immediately before the symbol. Thus to quote the symbol **ten**, we simply write **'ten**. The apostrophe is referred to as "quote," and the expression **'ten** is verbalized as "quote ten." Thus the responses to the prompts **[6]** and **[7]** can also be made as follows:

```
[6] 'abc3
abc3
[7] '12
12
```

We can also assign to a variable a value that is the literal value of a symbol. For example, if we enter the following:[1]

```
[8] (define Robert 'Bob)
```

we bind the variable `Robert` to the symbol `Bob`. When we next enter `Robert`, we get

```
[9] Robert
Bob
```

so that Scheme has evaluated `Robert` and returned the value `Bob`.

We have two types of data so far, numbers and symbols. How are they used? The use of numbers should be no surprise, since we usually think of doing arithmetic operations on numbers to get answers to problems. We shall take a brief look at how we do arithmetic in Scheme in this section and then return for a more complete look at using numbers in Chapter 3. To perform the arithmetic operations on numbers, Scheme uses *prefix notation*; the arithmetic operator is placed to the left of the numbers it operates on. The numbers on which it operates are called the *operands* of the operator. Furthermore, the operator and its operands are enclosed in a pair of parentheses. Thus to add the two numbers 3 and 4, we enter `(+ 3 4)` and Scheme evaluates it and returns the answer 7. On the computer screen it looks like this:

```
[10] (+ 3 4)
7
```

---

[1] We are mixing lower and uppercase letters in our symbols and showing that Scheme returns the same mix of lower and uppercase letters as their literal values. Thus, if we enter `'Bob`, Scheme returns `Bob`. An implementation of Scheme that preserves the case of letters is called *case preserving*, and in this book, we are assuming that the implementation is case preserving. There are some implementations that are not case preserving, which means that the case is changed to either all lowercase or all uppercase letters. Thus, in some implementations, all letters are returned in lowercase, and when we enter `'Bob`, Scheme evaluates it to `bob`. Other implementations that are not case preserving return all uppercase letters, so that if we enter `'Bob`, Scheme evaluates it to `BOB`.

Multiplication is performed with the operator *, subtraction with −, and division with /. How do we compute the arithmetical expression $3 \times (12 - 5)$? In prefix notation, we place the multiplication operator * first followed by the first number 3. The second operand to the operator * is the difference between 12 and 5, which itself is written as `(- 12 5)`. Thus the whole arithmetic expression is entered as

```
[11] (* 3 (- 12 5))
21
[12] (+ 2 (/ 30 15))
4
```

In general, Scheme uses this prefix notation whenever it applies any kind of operator to its operands. We shall return to a more complete discussion of numerical computations in Chapter 3. A number of experiments with numerical operations are included in the exercises at the end of this section.

In summary, a symbol can be bound to a value using a special form that begins with the keyword `define`. When a variable that has been bound to a value is entered in response to a Scheme prompt, its value is returned. If we want Scheme to return the literal value of the symbol instead of the value to which it is bound, we quote the variable. The value of a quoted symbol is just the literal value of the symbol.

It is possible to keep a record of the session you have in Scheme. The particular mechanism for doing so depends on the implementation of Scheme you are using. If you are using a version of Scheme that uses the windowing capability of the computer, you may be able to send what is in the window to a file. In some implementations, it is possible to run Scheme in an editor and use the saving capability of the editor to preserve what you want from the session in a file. Some versions offer a transcript facility that you turn on at the beginning of the session and give it a filename, and then turn off at the end of the session. The session is then preserved in the named file. The manual for the Scheme you are using should identify the facility you have available to save your Scheme sessions.

We strongly recommend that you try each of the things discussed in this book at the computer to see how they work. Feel free to experiment with variations on these ideas or anything else that occurs to you. You get a much better feeling for computers and for Scheme if you "play around" at the keyboard.

Exercises

Exercise 1.1

Find out what method your implementation of Scheme has for recording your Scheme session in a file. Bring up Scheme on the computer and record this session in a file called "session1.rec." Enter each of the following to successive prompts: `15`, `-200`, `12345678901234`, `(quote alphabet-soup)`, `'alphabet-soup`, `''alphabet-soup`. (Note: Experiment with entering even larger positive and negative whole numbers and decimals and see what is returned.)

Exercise 1.2

Assume that the following definitions have been made in the given order:

```
(define big-number 10500900)
(define small-number 0.00000025)
(define cheshire 'cat)
(define number1 big-number)
(define number2 'big-number)
```

What values are returned when the following are entered in response to the prompt?

a. `big-number`
b. `small-number`
c. `'big-number`
d. `cheshire`
e. `'cheshire`
f. `number1`
g. `number2`
h. `'number1`

Conduct the experiment on the computer in order to verify your answers.

Exercise 1.3

What is the result of entering each of the following expressions in response to the Scheme prompt? Verify your answer by performing these experiments on the computer.

a. `(- 10 (- 8 (- 6 4)))`

b. `(/ 40 (* 5 20))`

c. `(/ 2 3)`

d. `(+ (* 0.1 20) (/ 4 -3))`

Exercise 1.4

Write the Scheme expressions that denote the same calculation as the following arithmetic expressions. Verify your answers by conducting the appropriate experiment on the computer.

a. $(4 \times 7) - (13 + 5)$

b. $(3 \times (4 + (-5 - -3)))$

c. $(2.5 \div (5 \times (1 \div 10)))$

d. $5 \times ((537 \times (98.3 + (375 - (2.5 \times 153)))) + 255)$

Exercise 1.5
If α, β, and γ are any three numbers, translate each of the following Scheme expressions into the usual arithmetical expressions. For example:

(+ α (+ β γ)) translates into $\alpha + (\beta + \gamma)$

a. (+ α (- (+ β γ) α))

b. (+ (* α β) (* γ β))

c. (/ (- α β) (- α γ))

1.4 Constructing Lists

So far, we have seen two data types, symbols and numbers. Another important data type in Scheme is *lists*. We all use lists in our daily lives—shopping lists, laundry lists, address lists, menus, schedules, and so forth. In computing, it is also convenient to keep information in lists and to be able to manipulate that information. This section shows how to build lists and how to perform simple operations on lists. In Scheme, a list is denoted by a collection of items enclosed by parentheses. For example, `(1 2 3 4)` is a list containing the four numbers `1`, `2`, `3`, and `4`. A special list that we make frequent use of is the empty list, which contains no items. We denote the empty list by `()`.

Scheme provides a procedure to build lists one element at a time. This procedure is called **cons**, a shortening of "constructor." We refer to **cons** as a *constructor* of lists. We now look at how **cons** works. We shall first perform a number of experiments and then describe its general behavior. Suppose we want to build a list that contains only the number `1`. We enter the following:

```
[1] (cons 1 '())
(1)
```

We see from this example that we enclosed three things in parentheses: the variable **cons**, the number `1`, and the *empty list* `'()`. The first entry tells us

the name of the procedure we are applying, and the remaining two entries tell what the procedure **cons** is operating on. The entries following the name of the procedure are called the *operands* of the procedure. The values of the operands are called the *arguments* of the procedure. In our case, the first argument is the first item in the list we are constructing, and the second argument is a list that contains the rest of the items in the list we are building. Scheme first reads what we enter. In its evaluation phase, the operands are evaluated, and the desired list is built. It then prints the list `(1)`. (Note the parallel between the application of **cons** and the application of the arithmetic operations such as `(+ 3 4)`. We again see that the operator is placed to the left of the operands, using prefix notation.) Let us bind the variable `ls1` to the list containing the number `1` by writing

```
[2] (define ls1 (cons 1 '()))
[3] ls1
(1)
```

The **define** expression we entered at the prompt `[2]` binds the variable `ls1` to the value obtained by evaluating the subexpression `(cons 1 '())`. That subexpression evaluates to the list `(1)`. Thus `ls1` is bound to the list `(1)`. Thus when the variable `ls1` is entered at the prompt `[3]`, its value `(1)` is returned.

We now create a list with `2` as its first element and the elements of `ls1` as the rest of its elements. To accomplish this we write

```
[4] (cons 2 ls1)
(2 1)
[5] ls1
(1)
```

Once again, the two operands are evaluated—`2` evaluates to itself and `ls1` to the list `(1)`. Then a new list is formed having `2` as its first item and the items of `ls1` as the rest of its items, giving us `(2 1)`. This is the value that is returned. At prompt `[5]`, we verify that `ls1` is unchanged. Let us next bind the variable `ls2` to a list like the one in `[4]`.

```
[6] (define ls2 (cons 2 ls1))
[7] ls2
(2 1)
[8] ls1
(1)
```

The expression entered at the prompt [9] binds the variable c to the literal value of the symbol three. We can now create a list containing three as its first element and the elements of ls2 as the rest of its elements by writing

```
[9] (define c 'three)
[10] (cons c ls2)
(three 2 1)
```

When we apply cons to its two operands, the operands are both evaluated. The first operand, c, evaluates to three, and the second operand, ls2, evaluates to the list (2 1). Then a new list is built with three as its first item and the elements of the list (2 1) as the rest of its elements. The value (three 2 1) is returned.

Continuing our experiment, we bind the variable ls3 to the value of (cons c ls2) using a define expression:

```
[11] (define ls3 (cons c ls2))
```

We now perform another experiment with cons. Let us build a list that has as its first item the list ls2 and as the rest of its items the same items as those in ls3. This is done by making ls2 the first operand and ls3 the second operand of cons:

```
[12] (cons ls2 ls3)
((2 1) three 2 1)
[13] ls3
(three 2 1)
[14] (define ls4 (cons ls2 ls3))
```

The first operand of cons evaluates to the list (2 1), and the second operand of cons evaluates to (three 2 1). Thus the procedure cons produces a new list that has as its first item the list (2 1), followed by the elements in ls3. This gives us the value that was returned by Scheme: ((2 1) three 2 1). Notice that when ls3 was entered in response to the prompt [13], (three 2 1), the original value of ls3, was returned, so cons did build a new list and did not affect the list ls3.

We are now in a position to summarize the facts that we observed in the experiments. The procedure cons takes two operands. We apply the procedure cons to these operands by enclosing the procedure name cons followed by its two operands in parentheses. In general, a procedure name followed by its operands, all enclosed in a pair of parentheses, is called an *application*, and

we say that the operator is *applied* to its operands. When an application is evaluated by Scheme, all of the expressions in the list are evaluated in some unspecified order. The value of the first expression (the operator) informs Scheme of the kind of computation that is to be made (in our case, **cons** informs Scheme that a list is to be constructed). Then the computation defined by the procedure (the value of the operator) is performed on the arguments, which are the values of the operands. We assume for now that the second operand of **cons** evaluates to a list (which may be the empty list). Then a new list is created containing the value of the first operand as its first item followed by all of the items in the list to which the second operand evaluated. It is this new list that is returned as the value of the application. Since **cons** first evaluates its operands, the lists contain only values. So far, these values may be numbers, the literal value of symbols, and lists of these items. As we progress through the chapters of this book, we shall encounter other data types, all of which can be included in lists.

We have assumed in the discussion that the second operand of the cons application evaluates to a list. This is the usual situation that we shall encounter, but it is also possible for the second argument to **cons** not to be a list. We shall discuss this case in the next section. Furthermore, we see in **ls4** that a list may contain in it other lists. We say that the inner list is *nested* within the outer list. The nesting may be several levels deep, for a nested list may itself contain nested lists. Suppose we have a given list. Items that are not nested within lists contained in the given list are called the *top-level* items of the given list. Thus, if the given list is `((a b (c d)) e (f g) h)`, the top-level items are the list `(a b (c d))`, the symbol `e`, the list `(f g)`, and the symbol `h`.

We can also build the list `(2 1)` in one step by applying **cons** twice as the next experiment illustrates:

```
[15] (cons 2 (cons 1 '()))
(2 1)
```

To construct the list `((2 1) three 2 1)`, we could write

```
[16] (cons (cons 2 (cons 1 '())) (cons 'three (cons 2 (cons 1 '()))))
((2 1) three 2 1)
```

The second and third **cons**'s build the list `(2 1)`, and the fourth, fifth, and sixth build the list `(three 2 1)`. Then the first **cons** constructs the desired list.

We have used parentheses in writing several types of expressions—in the application of a procedure to its arguments, in the special form with keyword **define**, and in a list of values. When Scheme sees an expression enclosed in parentheses, it assumes that the first item following the left parenthesis evaluates to a procedure such as **cons** or is a keyword such as **define**.[2] It then evaluates the expression according to what the first item tells it to do. What happens when we enter an expression such as `(2 1)` in response to a Scheme prompt?

```
[17] (2 1)
Error: bad procedure 2
```

This experiment shows that Scheme expected to see an application or special form, and when the first item in the list is not an operator or a keyword, it returned a message saying it detected an error. In this case it tried to treat the list as an application but discovered as its first item the number 2, which is not a procedure.

Is there some way to enter a list of items that is to be taken literally? The answer is yes. Suppose we want to enter a list containing the following items: **three**, 2, 1. We use the quote symbol (apostrophe) and place it in front of the left parenthesis. This indicates that each of the items included in the parentheses is to be taken with its literal value. Thus to get a list containing the desired three items, we would enter `'(three 2 1)`. The symbol **three** should not be quoted within the parentheses since the outer quote already indicates that it should be taken with its literal value. Let's look at some more examples:

```
[18] '((2 1) three 2 1)
((2 1) three 2 1)
[19] '(a b (c (d e)))
(a b (c (d e)))
[20] (cons '(a b) '(c (d e)))
((a b) c (d e))
```

We now have a way of indicating whether a list we enter consists of literal values. If the expression beginning with a parenthesis is not quoted, Scheme

---

[2] We shall use several special forms in the coming chapters and then study their properties more fully in Chapter 14. They are called *special* because their operands are not evaluated as in procedure applications. If `(define ten 10)` were evaluated as a procedure application, the operands `ten` and `10` would first be evaluated, but since `ten` has not yet been bound, an error would result. In this special form, the symbol `ten` is not evaluated.

assumes that the expression is not a quoted list, and the first item in the list is examined to determine the nature of the expression and the computation that should follow. If the expression in parentheses is quoted, Scheme assumes that each item is to be taken literally.

We have now seen several procedures: the arithmetic operators `+`, `*`, `-`, and `/`, and the list-manipulating operator **cons**. Procedures form another type of data in Scheme. We have now encountered four types of data: numbers, symbols, lists, and procedures.

Exercises

Exercise 1.6

Using the symbols **one** and **two** and the procedure **cons**, we can construct the list `(one two)` by typing `(cons 'one (cons 'two '()))`. Using the symbols **one**, **two**, **three**, and **four** and the procedure **cons**, construct the following lists without using quoted lists (you may use quoted symbols and the empty list):

a. `(one two three four)`

b. `(one (two three four))`

c. `(one (two three) four)`

d. `((one two) (three four))`

e. `(((one)))`

Exercise 1.7

Consider a list **ls** containing n values. If α evaluates to any value, how many values does the list obtained by evaluating `(cons` α **ls**`)` contain?

Exercise 1.8

What is the result of evaluating `'(a 'b)`? (Try it!) Explain this result.

1.5 Taking Lists Apart

We have seen how to build lists using the constructor **cons**. We now consider how to take a list apart so that we can manipulate the pieces separately and build new lists from old. We accomplish this decomposition of lists using two

selector procedures, `car` and `cdr`.[3] If *ls* represents a nonempty list of items, `car` applied to *ls* gives the first item in *ls*, while `cdr` applied to *ls* gives the list consisting of all items in *ls* with the exception of its first item. Both `car` and `cdr` take one operand that must evaluate to a nonempty list. Both `car` and `cdr` are not defined on an empty list, and applying them to an empty list produces an error.

Let's look at the behavior of the selector `car`. When its argument is a nonempty list, it returns the first top-level item in the list. Thus we have

```
[1] (car '(1 2 3 4))
1
```

It is rather space consuming to indicate what a procedure returns by reproducing what is seen on the computer screen. We shall adopt a more efficient notation in which we express the above by

```
(car '(1 2 3 4)) ⟹ 1
```

The double arrow "⟹" is read as "evaluates to" or "returns." Here are some other examples of applying the procedure `car` (As in the previous section, `ls4` is bound to the list `((2 1) three 2 1)`.)

```
(car '(a b c d)) ⟹ a
(car ls4) ⟹ (2 1)
(car '((1) (2) (3) (4))) ⟹ (1)
(car '(ab (cd ef) gh)) ⟹ ab
(car '(((hen cow pig)))) ⟹ ((hen cow pig))
(car '(())) ⟹ ()
```

When the selector `cdr` is applied to an argument that is a nonempty list, the list returned is obtained when the first item (the `car`) of the argument list is removed. Thus

```
(cdr '(1 2 3 4)) ⟹ (2 3 4)
(cdr ls4) ⟹ (three 2 1)
```

---

[3] The symbol `cdr` is pronounced "could-er." The names *car* and *cdr* had their origin in the way the list-processing language LISP was originally implemented on the IBM 704, where one could reference the "address" and "decrement" parts of a memory location. Thus *car* is an acronym for "contents of address register," and *cdr* is an acronym for "contents of decrement register."

```
(cdr '(a (b c) (d e f))) ⟹ ((b c) (d e f))
(cdr '((ant hill) (bee hive) (wasp nest)))
                          ⟹ ((bee hive) (wasp nest))
(cdr '(1)) ⟹ ()
(cdr '((1 2))) ⟹ ()
(cdr '(())) ⟹ ()
```

We now have three list-manipulating procedures: the constructor **cons** and the two selectors **car** and **cdr**. By applying these in succession, we can do almost anything we want with lists. For example, if we want to get the second item in the list **(a b c d)**, we first apply **cdr** to get **(b c d)** and then apply **car** to the result to get **b**. We combine these applications of **cdr** and **car** into one expression by writing

```
(car (cdr '(a b c d))) ⟹ b
```

For the next example, let **list-of-names** be bound to the list **((Jane Doe) (John Jones))**. We look at how we retrieve Jane Doe's last name from this list. If we first apply **car** to **list-of-names**, we get the list **(Jane Doe)**. We now get the list **(Doe)** by applying **cdr**, and finally, we get **Doe** by applying **car**. We want to emphasize the distinction between the list **(Doe)** containing one item and the item **Doe** itself. All of these steps are combined in the following expression:

```
(car (cdr (car list-of-names))) ⟹ Doe
```

In this example, we see that the procedures **car** and **cdr** are applied in succession a number of times. The successive applications of **car**'s and **cdr**'s is facilitated by the use of the procedures **caar**, **cadr**, **caddr**, ..., **cddddr**. The number of **a**'s and **d**'s between the **c** and **r** tells us how many times we apply **car** or **cdr**, respectively, in order from right to left. For example, **(cadr '(a b c))** is equivalent to **(car (cdr '(a b c)))** and is **b**. Similarly, **(caddr '(a b c))** is equivalent to **(car (cdr (cdr '(a b c))))** and is **c**. We can put up to four letters (**a**'s or **d**'s) between the **c** and **r**. We make use of these procedures in the next example.

Consider the following situation. We ask our helper to prepare a menu that has on it the two items: chicken soup and ice cream. He prepares the menu by using a define expression to bind the variable **menu** to the list **(chicken soup ice cream)**:

```
(define menu '(chicken soup ice cream))
```

We find this unsatisfactory and want to use the items in the list **menu** to produce the list **((chicken soup) (ice cream))**, which groups together the related items. We build the new list one step at a time:

```
(car menu) ⟹ chicken
(cadr menu) ⟹ soup
(cons (cadr menu) '()) ⟹ (soup)
(cons (car menu) (cons (cadr menu) '())) ⟹ (chicken soup)
(cddr menu) ⟹ (ice cream)
```

We now have the two items that will make up our final list. We use **cons** to build the final answer. We first use **cons** to build a list around the list **(ice cream)** to get **((ice cream))** and then use **cons** again to build a list that has **(chicken soup)** as its first item and **(ice cream)** as its second item.

```
(cons (cddr menu) '()) ⟹ ((ice cream))
(cons (cons (car menu) (cons (cadr menu) '())) (cons (cddr menu) '()))
                    ⟹ ((chicken soup) (ice cream))
```

The process shown here can be used to build and manipulate lists in just about any way we want. As we learn more about Scheme, we shall discover shortcuts that facilitate the manipulation of lists.

Up to now, we have assumed that the second argument to **cons** is a list. If it is not a list, we can still apply **cons**; the result, however, is not a list but rather a *dotted pair*. A dotted pair is written as a pair of objects, separated by a dot (or period) and enclosed by parentheses. The first object in the dotted pair is the car of the dotted pair, and the second object in the dotted pair is the cdr of the dotted pair. Thus **(cons 'a 'b)** ⟹ **(a . b)**, and **(car '(a . b))** ⟹ **a**, while **(cdr '(a . b))** ⟹ **b**. Much of the work in this book involves lists, which are built out of dotted pairs. For example, **'(a . ())** ⟹ **(a)**, and **'(a . (b c))** ⟹ **(a b c)**. Thus any item built with the constructor **cons** is referred to as a *pair*.

Exercise

Exercise 1.9

If α and β evaluate to any values, what is

a. **(car (cons** α β**))**

b. **(cdr (cons** α β**))**

The procedures **cons**, **car**, and **cdr** do not alter their operands. Let us demonstrate this with an experiment.

```
[1] (define a 10)
[2] (define ls-b '(20 30 40))
[3] (car ls-b)
20
[4] (cdr ls-b)
(30 40)
[5] (cons a ls-b)
(10 20 30 40)
[6] a
10
[7] ls-b
(20 30 40)
```

After all of these operations involving **car**, **cdr**, and **cons**, the values of the operands **a** and **ls-b** stayed the same when they were entered in **[6]** and **[7]** as they were when they were defined in the beginning.

So far, we have encountered three procedures—**car**, **cdr**, and **cons**—that help us manipulate lists and four procedures—**+**, *, **-**, and /—that allow us to operate on numbers. Another group of procedures, called *predicates*, applies a test to their arguments and returns true or false depending on whether the test is passed. Scheme uses **#t** to denote true and **#f** to denote false. The value of **#t** is **#t** and the value of **#f** is **#f**, so both of these are constants.[4] **#t** and **#f**, representing true and false, are known as *boolean* (or *logical*) values. They form a separate type of data to give us five distinct types: numbers, symbols, booleans, pairs (including lists), and procedures. More data types will be introduced in later chapters. We now look at several predicates that apply to these five data types.

The first predicate tests whether its argument is a number, and its name is **number?**. Like most other predicates, the name ends with a question mark, signaling that the procedure is a predicate. Thus if we apply the predicate **number?** to some object, **#t** is returned if the object is a number, and otherwise **#f** is returned. If we make the following definitions,

```
(define num 35.4)
(define twelve 'dozen)
```

[4] In some implementations of Scheme, the empty list () is returned instead of **#f** to indicate false.

we get the following results:

```
(number? -45.67) ⟹ #t
(number? '3) ⟹ #t
(number? num) ⟹ #t
(number? twelve) ⟹ #f
(number? 'twelve) ⟹ #f
(number? (+ 2 3)) ⟹ #t
(number? #t) ⟹ #f
(number? (car '(15.3 -31.7))) ⟹ #t
(number? (cdr '(15.3 -31.7))) ⟹ #f
```

In the last example, the operand evaluates to `(-31.7)`, which is a list, not a number.

The predicate **symbol?** tests whether its argument is a symbol. With the definitions of **num** and **twelve** given above, we get the following results:

```
(symbol? 15) ⟹ #f
(symbol? num) ⟹ #f
(symbol? 'num) ⟹ #t
(symbol? twelve) ⟹ #t
(symbol? 'twelve) ⟹ #t
(symbol? #f) ⟹ #f
(symbol? (car '(banana cream))) ⟹ #t
(symbol? (cdr '(banana cream))) ⟹ #f
```

In the last example, `(cdr '(banana cream))` evaluates to a list, not a symbol.

There is also a predicate **boolean?** to test whether its argument is one of the boolean values **#t** or **#f**.

```
(boolean? #t) ⟹ #t
(boolean? (number? 'a)) ⟹ #t
(boolean? (cons 'a '())) ⟹ #f
```

A *pair* is an object built by the constructor **cons**, and the predicate **pair?** tests whether its argument is a pair. For example, nonempty lists are constructed by **cons**, so they are pairs. We have

```
(pair? '(Ann Ben Carl)) ⟹ #t
(pair? '(1)) ⟹ #t
(pair? '()) ⟹ #f
(pair? '(())) ⟹ #t
```

```
(pair? '(a (b c) d)) ⟹ #t
(pair? (cons 'a '())) ⟹ #t
(pair? (cons 3 4)) ⟹ #t
(pair? 'pair) ⟹ #f
```

There is also a predicate **null?** which tests whether its argument is the empty list.

```
(null? '()) ⟹ #t
(null? (cdr '(cat))) ⟹ #t
(null? (car '((a b)))) ⟹ #f
```

Exercises

Exercise 1.10
If the operands α and β evaluate to any values, what is

a. `(symbol? (cons` α β`))`

b. `(pair? (cons` α β`))`

c. `(null? (cons` α β`))`

d. `(null? (cdr (cons` α `'())))`

Exercise 1.11
If a list `ls` contains only one item, what is `(null? (cdr ls))`?

We have given tests to determine whether an object is a number, a symbol, a boolean, or a list, but we have not given a test to determine whether it is a procedure. There is also a predicate **procedure?** which tests whether its argument is a procedure.

```
(procedure? cons) ⟹ #t
(procedure? +) ⟹ #t
(procedure? 'cons) ⟹ #f
(procedure? 100) ⟹ #f
```

At this point, we have introduced five data types: numbers, symbols, booleans, pairs, and procedures. As we progress through the book, we shall meet other data types, such as strings, characters, vectors, and streams. A question that we often ask is whether two objects are the same. Scheme offers

several different predicates to test for the sameness of its arguments. Which predicate you use depends upon the information you seek and the data type of the objects. We list a number of these sameness predicates below and introduce others as the need arises. When both objects are numbers, we use the predicate = to test whether its arguments represent the same number. The predicate = is used only to test the sameness of numbers. It is safe to use it only on integers, since the representation of nonintegers in the computer can lead to undesirable results.

```
(= 3 (/ 6 2)) ⟹ #t
(= (/ 12 2) (* 2 3)) ⟹ #t
(= (car '(-1 ten 543)) (/ -20 (* 4 5))) ⟹ #t
(= (* 2 100) 20) ⟹ #f
```

There is also a predicate **eq?** to test the sameness of symbols. If its operands evaluate to the same symbol, **#t** is returned. For this example, assume that **Garfield** has been bound to **'cat**.

```
(eq? 'cat 'cat) ⟹ #t
(eq? Garfield 'cat) ⟹ #t
(eq? Garfield Garfield) ⟹ #t
(eq? 'Garfield 'cat) ⟹ #f
(eq? (car '(Garfield cat)) 'cat) ⟹ #f
(eq? (car '(Garfield cat)) 'Garfield) ⟹ #t
```

The predicate **eq?** returns **#t** if its two arguments are identical in all respects; otherwise it returns **#f**. Symbols have the property that they are identical if they are written with the same characters in the same order. Thus we use **eq?** to test for the sameness of symbols. On the other hand, each application of **cons** constructs a new and distinct pair. Two pairs constructed with separate applications of **cons** will always test **#f** using **eq?** even if the pairs they produce look alike. For example, let us make the following definitions:

```
[1] (define ls-a (cons 1 '(2 3)))
[2] (define ls-b (cons 1 '(2 3)))
[3] (define ls-c ls-a)
```

Then we have

```
[4] (eq? (cons 1 '(2 3)) (cons 1 '(2 3)))
#f
[5] (eq? ls-a '(cons 1 '(2 3)))
#f
[6] (eq? ls-a ls-b)
#f
[7] (eq? ls-a ls-c)
#t
```

In **[4]**, **cons** is applied twice to build two distinct pairs, so **#f** is returned even though both of the pairs look alike as lists **(1 2 3)**. In **[5]**, the variable **ls-a** refers to the pair defined in **[1]**, which is distinct from the pair defined by the **cons** in **[5]**, so **#f** is returned. In **[6]**, **ls-b** refers to the pair built by the **cons** in **[2]**, which is distinct from that built in **[1]**, so **eq?** again evaluates to **#f**. Finally, **ls-c** is defined to be the value of **ls-a**, which is the pair built by the **cons** in **[1]**, so both **ls-a** and **ls-c** refer to the same pair, and **eq?** evaluates to **#t**.

When we want to include numbers, symbols, and booleans in the types of objects the predicate tests for sameness, we use the predicate **eqv?**. We shall later see that **eqv?** also tests vectors, strings, and characters for sameness.

```
(eqv? (+ 2 3) (- 10 5)) ⟹ #t
(eqv? 5 6) ⟹ #f
(eqv? 5 'five) ⟹ #f
(eqv? 'cat 'cat) ⟹ #t
(eqv? 'cat 'kitten) ⟹ #f
(eqv? (car '(a a a)) (car (cdr '(a a a)))) ⟹ #t
```

We have not included lists among the data types we can test for sameness using the predicates discussed. If we want a universal sameness predicate that can be applied to test numbers, symbols, booleans, procedures, and lists (and strings, characters, and vectors), we use the predicate **equal?**. In the case of pairs constructed using separate applications of **cons**, **equal?** tests the corresponding entries, and if they are the same, **#t** is returned. Thus **equal?** tells us that the two lists **(cons 'a '(b c d))** and **(cons 'a '(b c d))** are the same, whereas **eq?** and **eqv?** claim that they are different.

```
(equal? 3 (/ 6 2)) ⟹ #t
(equal? 'cat 'cat) ⟹ #t
(equal? '(a b c) (cons 'a '(b c))) ⟹ #t
(equal? (cons 1 '(2 3)) (cons 1 '(2 3))) ⟹ #t
(equal? '(a (b c) d) '(a (b c) d)) ⟹ #t
```

```
(equal? '(a (b c)) '(a (c b))) ⟹ #f
(equal? (cdr '(a c d)) (cdr '(b c d))) ⟹ #t
```

Now for the obvious question: How do we know which one to use? When a predicate must first test to determine the type of its arguments, it is less efficient than one designed specifically for the type of its arguments. Thus for numbers, `=` is the most efficient sameness predicate. Similarly, for symbols, **eq?** is the most efficient predicate. For testing only numbers or symbols, **eqv?** is more efficient than **equal?**. When we know that we shall be using numbers or symbols, then **eqv?** is the sameness predicate we use. When the discussion is limited to numbers, we use `=`.

When we respond to a prompt with a number or a quoted symbol, we have seen that the number or symbol is returned. If we enter a symbol that has been bound to a value, that value is returned. If we apply a procedure such as **car** to a list `(1 2 3)` by entering `(car '(1 2 3))`, the expression is evaluated and the value `1` is returned and printed on the screen. On the other hand, not every Scheme object is printable. If we enter only the name of a procedure, such as **car**, the procedure, which is the value of **car**, is returned, but not printed; instead a message is displayed, which indicates a procedure. In this book, we indicate a procedure by printing angle brackets surrounding the name of the procedure in italics. Thus, when we enter **car**, *<car>* is displayed. In general, when we use *<some-symbol>*, it denotes a procedure.

We now summarize our discussion of **cons**, **car**, **cdr**, and predicates by writing some facts that apply to their use. The list is certainly not all inclusive, and we recommend that you add your own entries to it to reinforce your understanding of the use of predicates. Let α and β be operands such that α evaluates to any value and β evaluates to any nonempty list. We then have:

- The number of items in (**cons** α β) is one greater than the number of items in β.
- (**eq?** α β) ⟹ **#t**
  implies
  (**eqv?** α β) ⟹ **#t**
  implies
  (**equal?** α β) ⟹ **#t**
- (**eq?** (**cons** α β) (**cons** α β)) ⟹ **#f**
- (**eqv?** (**cons** α β) (**cons** α β)) ⟹ **#f**
- (**equal?** (**cons** α β) (**cons** α β)) ⟹ **#t**
- (**boolean?** (**eqv?** α β)) ⟹ **#t**

- (null? (cdr (cons α '()))) $\Longrightarrow$ #t
- (equal? (cons (car β) (cdr β)) β) $\Longrightarrow$ #t
- (equal? (car (cons α β)) α) $\Longrightarrow$ #t
- (equal? (cdr (cons α β)) β) $\Longrightarrow$ #t
- (null? β) $\Longrightarrow$ #f
- (pair? β) $\Longrightarrow$ #t
- (pair? (cons α β)) $\Longrightarrow$ #t
- (pair? (cons α_1 α_2)) $\Longrightarrow$ #t

We have been introduced to five basic data types (numbers, symbols, booleans, pairs, and procedures), and we have seen a number of procedures to manipulate and test the data. In Chapter 2 we shall develop the tools to compute with lists, and in Chapter 3 we shall do the same for numbers.

Exercises

Exercise 1.12
Evaluate each of the following.

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

b. `(car (cdr (cdr '(a (b c) (d e)))))`

c. `(car (cdr '((1 2) (3 4) (5 6))))`

d. `(cdr (car '((1 2) (3 4) (5 6))))`

e. `(car (cdr (car '((cat dog hen)))))`

f. `(cadr '(a b c d))`

g. `(cadar '((a b) (c d) (e f)))`

Exercise 1.13
We can extract the symbol `a` from the list `(b (a c) d)` using `car` and `cdr` by going through the following steps:

```
(cdr '(b (a c) d)) ⟹ ((a c) d)
(car (cdr '(b (a c) d))) ⟹ (a c)
(car (car (cdr '(b (a c) d)))) ⟹ a
```

For each of the following lists, write the expression using `car` and `cdr` that extracts the symbol `a`:

a. `(b c a d)`

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

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

d. `(((a)))`

Exercise 1.14

Decide whether the following expressions are true or false:

a. `(symbol? (car '(cat mouse)))`

b. `(symbol? (cdr '((cat mouse))))`

c. `(symbol? (cdr '(cat mouse)))`

d. `(pair? (cons 'hound '(dog)))`

e. `(pair? (car '(cheshire cat)))`

f. `(pair? (cons '() '()))`

Exercise 1.15

Decide whether the following expressions are true or false:

a. `(eqv? (car '(a b)) (car (cdr '(b a))))`

b. `(eqv? 'flea (car (cdr '(dog flea))))`

c. `(eq? (cons 'a '(b c)) (cons 'a '(b c)))`

d. `(eqv? (cons 'a '(b c)) (cons 'a '(b c)))`

e. `(equal? (cons 'a '(b c)) (cons 'a '(b c)))`

f. `(null? (cdr (cdr '((a b c) d))))`

g. `(null? (car '(())))`

h. `(null? (car '((()))))`

2 Procedures and Recursion

2.1 Overview

In Chapter 1 we used several Scheme procedures such as those bound to the numerical operators `+`, `*`, `-`, and `/`, the list-manipulating procedures bound to `cons`, `car`, and `cdr`, and the predicates that test their arguments and return `#t` or `#f`. One of the advantages of using the programming language Scheme is that the number of procedures provided by the language is relatively small, so we do not have to learn to use many procedures in order to write Scheme programs. Instead, Scheme makes it easy for us to define our own procedures as we need them. In this chapter, we discuss how to define procedures to manipulate lists. In Chapter 3, we shall see how to define procedures to do numerical computations. In this chapter, we also discuss how a procedure can call itself within its definition, a process called recursion. Finally, we introduce an elementary tracing tool to help us in debugging programs.

2.2 Procedures

The notation $f(x, y)$ is used in mathematics to denote a function; it has the name f and has two variables, x and y. We call the values that are given to the variables the *arguments* of the function. To each pair of arguments, the function associates a corresponding value. In computing, we are concerned with how that value is produced, and we speak about the sequence of computational steps that we perform to get the value returned by the function as an *algorithm* for computing the function's value. The way we implement the algorithm on the computer to get the desired value is called a *procedure* for

computing the desired value. If `f` is the name of the procedure with variables `x` and `y`, we use a list version, `(f x y)`, of the *prefix* notation $f(x, y)$ used in mathematics. In general, prefix notation places the procedure or function name in front of the variables. In the list version of prefix notation, the whole expression is surrounded by parentheses, and within the parentheses, the name of the procedure comes first, followed by the variables separated by spaces. Although we used a procedure taking two arguments in this illustration, the number of arguments depends on the procedure being used. For example, we have already seen the procedure `cons` takes two arguments, and the procedure `car` takes one.

Procedures such as those bound to the values of `+`, `cons`, `car`, `cdr`, `null?`, `eqv?`, and `symbol?` are provided by the system as standard routines. It is impossible for the system to provide all procedures needed. Therefore, it is important to be able to define procedures as they are needed. Scheme provides an elegant way of defining procedures based upon the lambda calculus introduced by the logician Alonzo Church. (See Church, 1941.) We illustrate this method with an example.

When we write `(cons 19 '())`, we get a list with one number in it, `(19)`. If we write `(cons 'bit '())`, we get a list with one symbol in it, namely `(bit)`. Now let's write a procedure of one variable that returns a list containing the value given to that variable as its only element. We do it with a *lambda* expression,

```
(lambda (item) (cons item '()))
```

A lambda expression is an example of a *special form*: a form of expression identified by a special symbol called a *keyword*, in this case `lambda`.[1]

If the procedure defined by this lambda expression is applied to `19`, the *parameter* `item`, which is in the list following the keyword `lambda`, is assigned (bound to) the value `19`. Then the following subexpression (known as the *body* of the lambda expression) is evaluated with the parameter `item` bound to `19`. The value of the body so obtained is returned as the value of the application. In this case, it returns the value of `(cons item '())`, which is `(19)`. In summary, when a procedure that is the value of a lambda expression is applied to some value, the parameter is bound to that value, and the body

[1] Special forms look like applications but are not, and in order to recognize them, we have to memorize the keywords, such as `lambda` and `define`. We shall see other keywords later, but the list of keywords we have to memorize is small.

of the lambda expression is evaluated with this parameter binding. The value of the body is returned as the value of the application of the procedure.

The lambda expression has the syntax

(**lambda** (*parameter ...*) *body*)

The keyword **lambda** is followed by a list that contains the parameters. The *ellipsis* (three dots) following *parameter* indicates that the list contains zero or more parameters. The next subexpression is the *body* of the lambda expression. The value of a lambda expression is the procedure, which can be applied to values appropriate for the evaluation of the body. These values must agree in number with the number of parameters in the lambda expression's parameter list. When the procedure is applied, the parameters are bound to the corresponding values, and the body is evaluated. The value of the body is then the value of the application.

In general, when a procedure is applied, the syntax is

(*operator operand ...*)

where *operator* is a subexpression that evaluates to the procedure being applied, and the *operands* are subexpressions that evaluate to the *arguments* to which the procedure is applied. We stress that the arguments are the values of the operands. For example, in the application `(* (+ 2 3) (- 7 1))`, the operator `*` evaluates to the multiplication procedure, the two operands are `(+ 2 3)` and `(- 7 1)`, and the two arguments are `5` and `6`. The value of the application is then `30`, the product of `5` and `6`.

Thus to apply the procedure we defined above to build a list containing the symbol **bit**, we enter

```
((lambda (item) (cons item '())) 'bit)
```

and we get as the result `(bit)`. Similarly,

```
((lambda (item) (cons item '())) (* 5 6)) ⟹ (30)
```

It is awkward to write the whole expression

```
(lambda (item) (cons item '()))
```

each time we want to apply the procedure. We can avoid this by giving the procedure a name and using that name in the procedure applications. This

is done by choosing a name, say **make-list-of-one**, for this procedure and then defining **make-list-of-one** to have the desired procedure as its value. We write

```
(define make-list-of-one (lambda (item) (cons item '())))
```

This is easier to read if we display the parts more clearly on separate lines as follows:

```
(define make-list-of-one
  (lambda (item)
    (cons item '())))
```

Scheme ignores any spaces in excess of the one space needed to separate expressions. Scheme also treats **<RETURN>**'s as spaces until the one following the last right parenthesis that is entered to close the first left parenthesis in the expression. Thus Scheme reads the two ways of writing this definition of **make-list-of-one** as the same Scheme expression. The indentation sets off subexpressions, making the structure of the program easier to understand at a glance.[2] To apply the procedure **make-list-of-one**, we enter the application

```
(make-list-of-one 'bit)
```

and **(bit)** is returned.

We have now written a program that builds a list containing one item. *Computer programs to perform various tasks are written by defining the appropriate procedure to accomplish the desired tasks.* As the tasks become more complicated, there are usually different ways of defining the procedures to achieve the desired results. It is the aim of this book to lead you through a series of learning experiences that will prepare you not only to be able to write such programs but to do so in a way that is efficient, elegant, and clear to read.

A word is in order about the choice of names for procedures and parameters. Since a symbol can have as many characters in it as we wish, programs will be easier to read if we choose names that describe the procedure or parameter.

---

[2] To make entering expressions easier, some implementations of Scheme provide automatic indenting and parenthesis matching. The automatic indenting places the cursor in the proper position for the start of the next line, and the parenthesis matching indicates the left parenthesis that a right parenthesis is closing.

Thus we used the name **make-list-of-one** for the procedure that converted a value into a list containing the value. In the lambda expression in the definition of the procedure **make-list-of-one**, we selected the name **item** for the parameter to indicate that it is expecting to be bound to the item that is to be included in the list.

Now let's write a procedure called **make-list-of-two** that takes two arguments and returns a list whose elements are those two arguments. The definition is:

```
(define make-list-of-two        ; This procedure creates
  (lambda (item1 item2)         ; a list of two items.
    (cons item1 (make-list-of-one item2))))
```

The parameter list following the keyword **lambda** consists of two parameters, **item1** and **item2**. You may be wondering about the semicolons in the first and second lines of the program and the statements following them. When Scheme reads an expression, it ignores all semicolons and whatever follows them on a line. This allows us to make remarks about the program so that the reader looking at it will know the intent of the programmer. Such remarks are called *documentation* and can make understanding programs easier. By choosing the names of variables carefully, you can reduce the amount of documentation necessary to understand a program. The documentation can also precede or follow the program if each line is preceded by a semicolon. In the programs in this book, we try to select variable names that make such documentation unnecessary. When we wish to make points of clarification, we shall state them in the accompanying discussion.

We apply the procedure **make-list-of-two** to the two symbols **one** and **two** by writing

```
(make-list-of-two 'one 'two) ⟹ (one two)
```

When we defined the procedure **make-list-of-two**, we used the parameters **item1** and **item2**. When we applied the procedure **make-list-of-two**, its two arguments were the values of the operands **'one** and **'two**.

In Section 1.5, we saw how to take a list containing four items (**menu** was bound to the list **(chicken soup ice cream)**) and build a new list containing the same items but grouped into two lists, each containing two items. We can use the procedure **make-list-of-two** to give us another way of doing that grouping. We define a procedure called **regroup** that has as its parameter **list-of-4**, which will be bound to a list of four items. It returns a list with the items in **list-of-4** regrouped into two lists of two items each. In the

course of writing the definition of **regroup**, we shall find it clearer to make use of certain other procedures, which express what we want to appear in the list of the two items we create. We use these procedures in the definition of **regroup** and then define them afterward. The order in which the definitions are written does not matter, and it is often more convenient to use a procedure in a definition where it is needed, and then to define it later. In the definition that follows, we make use of two such *helping procedures*, **first-group** and **second-group**.

```
(define regroup
  (lambda (list-of-4)
    (make-list-of-two
      (first-group list-of-4)
      (second-group list-of-4))))
```

The procedure **make-list-of-two** is used to create a list of two items, the first item being a list consisting of the first two items in **list-of-4** and the second consisting of the last two items in **list-of-4**. To construct the first grouping, we use a helping procedure **first-group** that we define as:

```
(define first-group
  (lambda (ls)
    (make-list-of-two (car ls) (cadr ls))))
```

We define the helping procedure **second-group** as:

```
(define second-group
  (lambda (ls)
    (cddr ls)))
```

When **first-group** is applied to **list-of-4**, the parameter **ls** is bound to the list of four items and the helping procedure **make-list-of-two** is applied to build the desired list consisting of the first two items in the list of four items. Similarly, the helping procedure **second-group** produces the rest of the list of four items following the first two, that is, the list consisting of the last two items.

Now to get the new menu, we simply apply the procedure **regroup** to **menu**, and we get the desired list:

```
(regroup menu) ⟹ ((chicken soup) (ice cream))
```

What is gained by using these procedures over the method used in Chapter 1 in which everything was expressed in terms of **cons**, **car**, **cdr**, and so forth? The version in Chapter 1 is hard to understand when it is scanned, for we have to pause to work out what the constructors and selectors are doing. In the new version, you can look at the code for **regroup** and see immediately that it is making a list of two items; the first group is again a list of two items, the first two items in the list of four items, and the second group is a list consisting of the remaining two items in the list of four items. By carefully choosing the names of the procedures and parameters, we can make the programs easy to read and understand. In our case, the use of the three helping procedures, **make-list-of-two**, **first-group**, and **second-group**, make the program easier to understand. Often the helping procedures can be used in many programs. In reality, helping procedures are ordinary procedures that we happen to want to make use of in writing some program. Any procedure can be used as a helping procedure.

We have defined procedures to build lists containing one item and two items. Scheme provides a procedure **list**, which takes any number of arguments and constructs a list containing those arguments. For example,

```
(list 'a 'b 'c 'd) ⟹ (a b c d)
(list '(1 2) '(3 4)) ⟹ ((1 2) (3 4))
(list) ⟹ ()
```

We shall see how **list** is defined in Chapter 7.

There are two styles of writing programs, *top-down* and *bottom-up* programming. In both, we are looking for the solution of some problem and want to write a procedure that returns the desired solution as its value. For now, we refer to this as the main procedure. In top-down style, we first write the definition of the main procedure. The main procedure often uses certain helping procedures, so we write the definitions of the helping procedures next. These in turn may require other helping procedures, so we write those, and so on. In bottom-up style, we first write the definitions of the helping procedures that we anticipate using, and at the end, we write the main procedure. We shall use both styles of programming in this book.

We summarize this discussion by observing that the value of a lambda expression with the syntax

(lambda (*parameter ...*) *body*)

is a procedure. The *ellipsis* after *parameter* means that this is a list of zero or more parameters. When the procedure is applied, the parameters are bound to the arguments (i.e., the values of the operands), and the body is evaluated.

We can give the procedure a name by using a define expression with the structure

(**define** *procedure-name lambda-expression*)

where *procedure-name* is the variable used as the name of the procedure.[3] We *apply* (*call* or *invoke*) such a named procedure by writing the application

(*procedure-name operand ...*)

where the number of operands matches the number of parameters in the definition of the procedure. In general, when an application of the form

(*operator operand ...*)

is evaluated, the *operands* and the *operator* are all evaluated in some unspecified order. The *operator* must evaluate to a procedure. The values of the *operands* are the arguments. The procedure binds the parameters to the arguments and evaluates the body, the value of which is the value of the application. Because the operands are first evaluated and it is their values, the arguments, that the procedure receives, we say the operands are *passed by value* to the procedure.

We have also encountered two expressions that are called *special forms*: those with the keywords **define** and **lambda**. These expressions are not applications because not all the items in the expressions are evaluated initially. For example, in a lambda expression, the parameter list is never evaluated and its body is not evaluated initially. Most computer languages have some keywords that have special meaning and cannot be used for other purposes. In Scheme the number of such keywords for special forms is relatively small. In Chapter 14, we shall see how we can add to Scheme our own special forms.

---

[3] Scheme also supports

(**define** (*procedure-name parameter ...*) *body*)

as a syntax for a define expression.

Exercises

When doing these exercises, you may find it convenient to save the definitions of the procedures in a file. These procedures can then be used again. They can be entered into Scheme from a file in which they were saved either by using a transfer mechanism or by invoking a loading procedure. In some implementations of Scheme, this is done with `(load "`*filename*`")`.

Exercise 2.1: `second`
Define a procedure called `second` that takes as its argument a list and that returns the second item in the list. Assume that the list contains at least two items.

Exercise 2.2: `third`
Define a procedure called `third` that takes as its argument a list and that returns the third item in the list. Assume that the list contains at least three items.

Exercise 2.3: `firsts-of-both`
The procedure `firsts-of-both` is defined as follows:

```
(define firsts-of-both
  (lambda (list-1 list-2)
    (make-list-of-two (car list-1) (car list-2))))
```

Determine the value of the following expressions:

a. `(firsts-of-both '(1 3 5 7) '(2 4 6))`

b. `(firsts-of-both '((a b) (c d)) '((e f) (g h)))`

Exercise 2.4: `juggle`
Define a procedure `juggle` that rotates a three-element list. The procedure `juggle` returns a list that is a rearrangement of the input list so that the first element of this list becomes the second, the second element becomes the third, and the third element becomes the first. Test your procedure on:

```
(juggle '(jump quick spot)) ⟹ (spot jump quick)
(juggle '(dog bites man)) ⟹ (man dog bites)
```

Exercise 2.5: **switch**

Define a procedure **switch** that interchanges the first and third elements of a three-element list. Test your procedure on the examples given in the previous exercise.

2.3 Conditional Expressions

Suppose we want to define a predicate that tests whether a value is a number, a symbol, an empty list, or a pair, and returns a symbol indicating its type. The structure of the test can be written in natural language as:

> If the value is a pair, return the symbol **pair**.
> If the value is an empty list, return the symbol **empty-list**.
> If the value is a number, return the symbol **number**.
> If the value is a symbol, return the symbol **symbol**.
> Otherwise, return the symbol **some-other-type**.

This description of the procedure using English gives a sequence of steps that we follow to carry out the computation. Such a sequence of steps describing a computation is called an *algorithm*. We implement the kind of "case analysis" given in tl 's algorithm using a *cond expression* (the *special form with keyword* **cond**). The keyword **cond** is derived from the word *conditional*. Using **cond**, we write a procedure called **type-of** that tests its argument and returns the type of the item as described above:

```
(define type-of
  (lambda (item)
    (cond
      ((pair? item) 'pair)
      ((null? item) 'empty-list)
      ((number? item) 'number)
      ((symbol? item) 'symbol)
      (else 'some-other-type))))
```

Let us analyze the cond expression. In this case, the cond expression has five *clauses*, each represented by two expressions enclosed in parentheses. The first clause, **((pair? item) 'pair)**, has as its first expression **(pair? item)**, which is a boolean or logical expression with the value **#t** or **#f** depending on whether the value bound to **item** is or is not a pair. We shall also refer to the boolean expression as the *condition*. If the condition evaluates to true, then the second expression in the clause (the *consequent*), **'pair**, is evaluated and **pair** is returned. If the condition in the first clause evaluates to false, the

condition in the second clause `((null? item) 'empty-list)` is evaluated. If one of the subsequent conditions is true, then its consequent is evaluated and that value is returned. The last clause has the keyword `else` as its first expression, and if all of the preceding conditions are false, the expression following `else` is evaluated, and its value is returned. The expression following `else` is referred to as the *alternative*.

In general, the syntax of a cond expression is

```
(cond
  (condition_1 consequent_1)
  (condition_2 consequent_2)
        .            .
        .            .
        .            .
  (condition_n consequent_n)
  (else alternative))
```

where for each $k = 1, \ldots, n$, the expressions (*condition*$_k$ *consequent*$_k$) and (`else` *alternative*) are called *clauses*. The *condition*$_k$ and *consequent*$_k$, for $k = 1, \ldots, n$, and the *alternative* are expressions, and `else` is a keyword. Each of the conditional parts of the clauses is evaluated in succession until one is true, in which case the corresponding consequent is evaluated, and the value of the cond expression is the same as the value of the consequent corresponding to the true condition. If none of the conditions is true, the cond expression has the same value as the alternative, which is in the last cond clause, known as the else clause.[4]

Scheme has another way of handling conditional expressions that have only two cases. We can also use the special form with keyword `if`. Suppose we want to write a procedure `car-if-pair` that does the following:

> If its argument is a pair, return the car of the pair.
> Otherwise, return the argument.

Here is the procedure `car-if-pair` using `cond`:

```
(define car-if-pair
  (lambda (item)
    (cond
      ((pair? item) (car item))
      (else item))))
```

[4] The else clause is optional. If it is omitted and all of the conditions are false, then Scheme does not specify the value that is returned as the value of the cond expression. We shall avoid using cond expressions that return unspecified values.

or using an if expression, it can be written as:

```
(define car-if-pair
  (lambda (item)
    (if (pair? item)
        (car item)
        item)))
```

In general, the syntax of an if expression is

(if *condition consequent alternative*)

or

(if *condition consequent*)

In the first case, if *condition* is true, the value of *consequent* is returned as the value of the if expression; if *condition* is false, the value of *alternative* is returned as the value of the if expression. In the second case, the alternative is not present. In this "one-armed if," if *condition* is true, the value of *consequent* is returned as the value of the if expression. If it is false, an unspecified value is returned.

If expressions can be nested, enabling us to write the procedure **type-of** given above as follows:

```
(define type-of
  (lambda (item)
    (if (pair? item)
        'pair
        (if (null? item)
            'empty-list
            (if (number? item)
                'number
                (if (symbol? item)
                    'symbol
                    'some-other-type))))))
```

Any cond expression can be written as nested if expressions, but as the number of cases increases, the nesting of the if expressions gets deeper, and the meaning of the whole conditional expression is obscured. Thus, using a cond expression is often advantageous when there are several cases.

The use of conditional expressions with either **if** or **cond** depends upon first evaluating a condition. The condition may be simple, such as (**null? ls**), or it may involve something like testing whether **ls** is a pair *and* whether its **car** is some symbol such as **cat**. A condition that involves a combination of two or more simple conditions is called a *compound* condition. We build compound conditions by combining simple conditions with the logical composition operators **and**, **or**, and **not**. The compound condition mentioned above can be written using **and** as follows:

```
(and (pair? ls) (eq? (car ls) 'cat))
```

The syntax of each of these logical operators is given below:

(**and** $expr_1$ $expr_2$ ... $expr_n$)
(**or** $expr_1$ $expr_2$ ... $expr_n$)
(**not** $expr$)

The and expression evaluates each of the subexpressions $expr_1$, $expr_2$, ..., $expr_n$ in succession. If any one of them is false, it stops evaluating the rest of the subexpressions, and the value of the and expression is **#f**. If all of the subexpressions have true values, the value of the last subexpression is returned as the value of the and expression.[5]

The or expression evaluates each of the subexpressions $expr_1$, $expr_2$, ..., $expr_n$ in succession. If any one of them is true, it stops evaluating the rest of the subexpressions, and the value of the or expression is the value of that first true subexpression. If all of the subexpressions are false, the value of the or expression is **#f**.

The value of the not expression is **#f** when $expr$ has a true value, and it is **#t** when $expr$ is false.

We illustrate the use of **and** and **or** in the following examples:

```
(define s-and-n-list?
  (lambda (ls)
    (and (pair? ls)
         (symbol? (car ls))
         (pair? (cdr ls))
         (number? (cadr ls)))))
```

[5] Scheme has a convention of treating any value that is not false as true. Thus (**if 'cat 'kitten 'puppy**) ⟹ **kitten**, since the condition **'cat** evaluates to **cat**, which is not false. It is good programming style, however, for the conditions to be boolean expressions that evaluate to either **#t** or **#f**.

The predicate `s-and-n-list?` takes a list as its argument. The value of the expression `(s-and-n-list? some-list)` is `#t` if:

> `some-list` is a pair,
> and the first item in `some-list` is a symbol,
> and the cdr of `some-list` is a pair,
> and the second item in `some-list` is a number.

Otherwise, the value of `(s-and-n-list? some-list)` is `#f`. For example,

```
(s-and-n-list? '(a 1 b)) ⟹ #t
```

while

```
(s-and-n-list? '(a b 1)) ⟹ #f
```

The test to determine whether the list is a pair is necessary since we can only take the `car` of a pair. If the list is empty, the evaluation of the `car` of the list never takes place. The evaluation terminates on the first false value.

```
(define s-or-n-list?
  (lambda (ls)
    (and (pair? ls)
         (or (symbol? (car ls))
             (number? (car ls))))))
```

The predicate `s-or-n-list?` takes a list as its argument. The expression `(s-or-n-list? some-list)` ⟹ `#t` if:

> `some-list` is a pair,
> and either the first item in `some-list` is a symbol or it is a number.

Otherwise `(s-or-n-list? some-list)` ⟹ `#f`.

There are occasions when we want to test whether a list contains precisely one item, that is, whether the list is a *singleton* list. It is easy to define a predicate `singleton-list?` that tests whether its argument is a pair and whether it contains just one element. To test whether a pair contains just one element, it is enough to test whether its `cdr` is empty. Thus we can write

Program 2.1 `singleton-list?`

```
(define singleton-list?
  (lambda (ls)
    (and (pair? ls) (null? (cdr ls)))))
```

This definition makes use of the fact that the empty list is not a pair. Thus the nonempty list whose `cdr` is empty must contain just one item and is thus a singleton list.

Exercises

Exercise 2.6

Assume that `a`, `b`, and `c` are expressions that evaluate to `#t` and that `e` and `f` are expressions that evaluate to `#f`. Decide whether the following expressions are true or false.

a. `(and a (or b e))`

b. `(or e (and (not f) a c))`

c. `(not (or (not a) (not b)))`

d. `(and (or a f) (not (or b e)))`

Exercise 2.7

Decide whether the following expressions are true or false if `expr` is some boolean expression.

a. `(or (symbol? expr) (not (symbol? expr)))`

b. `(and (null? expr) (not (null? expr)))`

c. `(not (and (or expr #f) (not expr)))`

d. `(not (or expr #t))`

Exercise 2.8

Decide whether the following expressions are true or false using `s-and-n-list?` as defined in this section.

a. `(s-and-n-list? '(2 pair 12 dozen))`

b. `(s-and-n-list? '(b 4 u c a j))`

c. `(s-and-n-list? '(a ten))`

d. `(s-and-n-list? '(a))`

Exercise 2.9

Decide whether the following expressions are true or false using `s-or-n-list?` as defined in this section.

a. `(s-or-n-list? '(b))`

b. `(s-or-n-list? '(c 2 m))`

c. `(s-or-n-list? '(10 10 10 10))`

d. `(s-or-n-list? '())`

2.4 Recursion

We saw in Section 2.2 that certain procedures use other procedures as helping procedures. In this section, we define procedures that use themselves as helping procedures. When a procedure calls itself within the body of the lambda expression defining it, we say that the procedure is *recursive*. To introduce the idea of a recursive procedure, we set as our goal the definition of a procedure **last-item**, that, when applied to a nonempty list, returns the last top-level item in the list. Here are some examples of applications of **last-item**:

```
(last-item '(1 2 3 4 5)) ⟹ 5
(last-item '(a b (c d))) ⟹ (c d)
(last-item '(cat)) ⟹ cat
(last-item '((cat))) ⟹ (cat)
```

It is a good idea to begin with the simplest cases of the arguments to which the procedure is applied. In this case, the simplest nonempty list is a list containing only one item. For example, if the list is `(a)`, then the last item is also the first item, and applying **car** to this list produces the last item. This would work with any list containing only one top-level item, for the car of the list is both its first and its last top-level item. Let us use the variable **ls** as the parameter in the definition of **last-item**. How can we test whether **ls** contains only one top-level item? When **ls** contains only one top-level item, its cdr is the empty list. Thus the boolean expression `(null? (cdr ls))` returns **#t** when—and indeed only when—the nonempty list **ls** contains only one top-level item. Thus, we may use a cond expression to test whether we have the case of a one-item list and return the car of the list if that is the case. We can then begin our program as follows:

```
(define last-item
  (lambda (ls)
    (cond
      ((null? (cdr ls)) (car ls))
      ... )))
```

If we now consider a list `ls` containing more than one top-level item, the cdr of that list contains one fewer top-level items, but still includes the last item of the original list. Each successive application of `cdr` reduces the number of top-level items by one, until we finally have a list containing only one top-level item, for which we have a solution. In this sense, application of `cdr` to the list reduces the problem to a simpler case. This leads us to consider the list obtained by evaluating `(cdr ls)`,[6] which contains all of the items of `ls` except its first item. The last item in `(cdr ls)` is the same as the last item in `ls`. For example, the list `(a b c)` and the list `(b c)`, which is its cdr, have the same last item, `c`. Thus if we call the procedure `last-item` as a helping procedure to be applied to `(cdr ls)`, we get the desired last item of the original list, and that solves our problem. Thus to complete the definition of `last-item`, we add the else clause to handle the case where the list contains more than one item:

Program 2.2 `last-item`

```
(define last-item
  (lambda (ls)
    (cond
      ((null? (cdr ls)) (car ls))
      (else (last-item (cdr ls))))))
```

To see that this does define the procedure `last-item` so that it returns the correct result for any nonempty list `ls`, we consider first a list `(a)` containing only one item. Then the condition in the first cond clause is true, and `(car ls)` does give us the last (which is also the first) item, `a`, in the list. Thus `last-item` works on any list containing only one item. Now let's consider the case in which `ls` is a list `(a b)` containing two items. Then its cdr, `(b)`, contains one item, so the procedure `last-item` does work on `(cdr ls)`, allowing us to use it as a helping procedure in the else clause to get the correct result. Thus `last-item` solves the problem for any list of two items. Now we use the fact that `last-item` works on the cdr of any three-item list to conclude that it

[6] It is common practice, when the context is clear, not to include the phrase *obtained by evaluating*. We say, "the list `(cdr ls)`" instead of "the list obtained by evaluating `(cdr ls)`" whenever the context makes it clear that we want the value of `(cdr ls)` rather than the literal list whose first item is `cdr` and whose second item is `ls`. When we want the literal list, and the context is not clear, we indicate so by quoting it.

works on the three-item list itself. We can continue this process of increasing by one the number of items in the list indefinitely, showing that `last-item` solves the problem for any list.

Since the procedure `last-item` called itself as a helping procedure, `last-item` is a recursive procedure. Our strategy in general in designing a recursive procedure on a list is first to identify the "simplest case" and write the expression that solves the problem for that case as the consequent in the first cond clause. We call this simplest case the *base case* or *terminating condition*. We then identify a simplifying operation, which on repeated application to the list produces the base case. Then in each of the other cases, we solve the problem with some expression that calls the recursive procedure as a helping procedure applied to the simplified list. In our example, the base case is the list consisting of only one item. The simplifying operation is `cdr`, and in the other cases, we see that the expression that solves the problem applies `last-item` to the simplified list `(cdr ls)`.

To give us a better intuition about how `last-item` works, we shall apply `last-item` to the list `(a b c)`. What is `(last-item '(a b c))`? We shall walk through the evaluation of this expression. The parameter `ls` is bound to the argument `(a b c)`, and the cond expression is evaluated. In this case, `(cdr ls)` is not empty, so the alternative in the else clause is evaluated. This tells us to apply `last-item` to `(cdr ls)`. Since `(cdr ls)` is `(b c)`, we must evaluate `(last-item '(b c))`. We thus bind the parameter `ls` to the argument `(b c)` and enter the cond expression. Once again, `(cdr ls)` is not empty, so we evaluate the alternative in the else clause. This tells us to apply `last-item` to `(cdr ls)`, which now is `(c)`. Thus we must evaluate `(last-item '(c))`. We now bind the parameter `ls` to the argument `(c)` and enter the cond expression. This time `(cdr '(c))` is the empty list. Thus the consequent is evaluated to give `(car '(c))`—`c` as the value of the expression.

The recursion in the illustration stops when the list is simplified to the base case. In that case, the condition in the first cond clause is true. We call the condition used to stop the recursion the *terminating condition*. In our example, the terminating condition is `(null? (cdr ls))`. Generally, whenever a recursive procedure is defined, a terminating condition must be included so that the recursion will eventually stop. (In Chapter 15 on streams, we shall see examples in which a terminating condition is not needed.) We usually begin the definition of a recursive procedure by writing the terminating condition as the first cond clause. We then proceed with the rest of the definition.

In the preceding discussion we introduced the *substitution model*. Using the substitution model, we can determine the value of an expression by substitut-

ing values for parameters. Through the first eight chapters, the substitution model suffices. From Chapter 9 on, however, there will be times when the substitution model does not work. From time to time, we use it to clarify a computation; most of the time, however, we use the general approach: the *environment model*. In that approach we just remember the bindings of variables and avoid any substitutions.

Let us next define a procedure **member?** that decides for us whether its first argument is **equal?** to one of the top-level items in the list that is its second argument. For example,

1. `(member? 'cat '(dog hen cat pig))` ⟹ `#t`
2. `(member? 'fox '(dog hen cat pig))` ⟹ `#f`
3. `(member? 2 '(1 (2 3) 4))` ⟹ `#f`
4. `(member? '(2 3) '(1 (2 3) 4))` ⟹ `#t`
5. `(member? 'cat '())` ⟹ `#f`

In Example 3, `2` is not a top-level item in the list `(1 (2 3) 4)`, so `#f` is returned. We begin the definition of **member?** by determining the base case. Regardless of what **item** is, if **ls** is the empty list, `#f` is returned. This is the simplest case and will be taken as our base case. To test for the base case, we use the predicate **null?** so the terminating condition is `(null? ls)`. The consequent for the terminating condition is `#f`. We can therefore begin the definition of **member?** as a procedure having two parameters, **item** and **ls**:

```
(define member?
  (lambda (item ls)
    (cond
      ((null? ls) #f)
      ... )))
```

Now given any list, what is the simplifying operation that simplifies **ls** to the empty list? It is again the procedure **cdr**. Assume that **ls** is not empty. If we know the value of `(member? item (cdr ls))`, how do we get the value for `(member? item ls)`? Well, when is the latter statement true? It is true if either the first item in **ls** is the same as **item** or if **item** is a member of the rest of the list following the first item. This can be written as the or expression:

```
(or (equal? (car ls) item) (member? item (cdr ls)))
```

Thus in the case when **ls** is not empty, the above expression is true exactly

when the expression (member? item ls) is true. We then complete the definition of member? with

Program 2.3 member?

```
(define member?
  (lambda (item ls)
    (cond
      ((null? ls) #f)
      (else (or (equal? (car ls) item)
                (member? item (cdr ls)))))))
```

The procedure member? is recursive since it calls itself. Let us review the reasoning used in the program for member?. If the terminating condition (null? ls) is true, then item is not in ls, and the consequent is false. Otherwise we look at the alternative, which is true if either item is the first item in ls or if item is in (cdr ls) and is otherwise false.

When member? calls itself with argument (cdr ls), its parameter is bound to the value of (cdr ls), which is a shorter list than the parameter's previous binding to ls. In each successive recursive procedure call, the list is shorter, and the process is guaranteed to stop because of the terminating condition (null? ls).

In order to use a list as the first argument to member? (as in Example 4), we used the predicate equal? to make the sameness test in the else clause. If we know that the items to which item is bound will always be symbols, we can use eq? in place of equal?. The procedure so defined using eq? is named memq? to distinguish it from member?, which is defined using equal? for the sameness test. Similarly, if we know that the items to which item is bound will always be either symbols or numbers, we can use eqv? for the sameness test and call the procedure so defined memv?.[7]

We have now defined the procedure last-item, which picks the last top-level item out of a list, and the procedure member?, which tests whether an item is a top-level element in a given list. We continue illustrating how to define recursive procedures with the definition of another useful procedure

[7] Scheme provides the three procedures member, memq, and memv, written without the question mark. These behave somewhat differently from the ones we defined with the question mark in that if item is not found, false is returned, but if item is found in ls, the sublist whose car is item is returned. For example, (memq 'b '(a b c)) ⟹ (b c).

for manipulating lists. The procedure **remove-1st** removes the first top-level occurrence of a given item from a list of items. For example,

1. `(remove-1st 'fox '(hen fox chick cock))`
   `⟹ (hen chick cock)`
2. `(remove-1st 'fox '(hen fox chick fox cock))`
   `⟹ (hen chick fox cock)`
3. `(remove-1st 'fox '(hen (fox chick) cock))`
   `⟹ (hen (fox chick) cock)`
4. `(remove-1st 'fox '()) ⟹ ()`
5. `(remove-1st '(1 2) '(1 2 (1 2) ((1 2))))`
   `⟹ (1 2 ((1 2)))`

In general, the procedure **remove-1st** takes two arguments, an element **item** and a list **ls**. It builds a new list from **ls** with the first top-level occurrence of **item** removed from it. We again begin looking at the simplest case, in which **ls** is the empty list. Since **item** does not occur at all in the empty list, the list we build is still the empty list. The test for the base case is then **(null? ls)**, and the value returned in its consequent is **()**. Thus the definition of the procedure **remove-1st** begins with

```
(define remove-1st
  (lambda (item ls)
    (cond
      ((null? ls) '())
      ... )))
```

If **ls** is not empty, the procedure that simplifies it to the base case is again **cdr**. If we already know **(remove-1st item (cdr ls))**, that is, if we have a list consisting of the first top-level occurrence of **item** removed from **(cdr ls)**, how do we build up a list that is obtained by removing the first top-level occurrence of **item** in **ls**? There are two cases to consider. Let's first consider the example in which we remove the first occurrence of **a** from the list **(a b c d)**. Since **a** is the first item in the list, we get the desired result by merely taking the **cdr** of the original list. This is the first case we consider. If the first top-level item in **ls** is the same as **item**, then we get the desired list by simply using **(cdr ls)**. This case can be added to the definition of **remove-1st** by writing

```
(define remove-1st
  (lambda (item ls)
    (cond
      ((null? ls) '())
      ((equal? (car ls) item) (cdr ls))
      ... )))
```

The only case left to be considered is when `ls` is not empty, and its first top-level item is not the same as `item`. Consider the example in which we apply `remove-1st` to remove the letter `c` from the list `(a b c d)`. The list is not empty and its first item is not `c`. Thus the list we build begins with `a` and continues with the items in `(b d)`. But `(b d)` is just the list obtained by removing `c` from `(b c d)`. The final result is then `(a b d)`, which was obtained by building the list

```
(cons (car '(a b c d)) (remove-1st 'c (cdr '(a b c d))))
```

In general, the list we are building now begins with the first element of `ls` and has in it the elements of `(cdr ls)` with the first top-level occurrence of `item` removed. But this is obtained when we cons[8] `(car ls)` onto `(remove-1st item (cdr ls))`, so the final case is disposed of by adding the else clause to the definition, which is given in Program 2.4.

Program 2.4 `remove-1st`

```
(define remove-1st
  (lambda (item ls)
    (cond
      ((null? ls) '())
      ((equal? (car ls) item) (cdr ls))
      (else (cons (car ls) (remove-1st item (cdr ls)))))))
```

To get a better understanding of how recursion works, let's walk through the evaluation of an application of the procedure `remove-1st`; for example

```
(remove-1st 'c '(a b c d))
```

[8] Scheme programmers use the verb *cons*, which has an infinitive "to cons", tenses "cons, cons'd, has cons'd", participle "consing", and conjugation "I cons, he conses, etc." We shall make frequent use of these words.

Since the list `(a b c d)` is not empty and the first entry is not `c`, the alternative in the else clause is evaluated. This gives us

```
(cons 'a (remove-1st 'c '(b c d)))
```

To get the value of this expression, we must evaluate the `remove-1st` subexpression. Once again, the list `(b c d)` is not empty, and the first item in the list is not the same as `c`. Thus the alternative in the else clause is evaluated. This gives us as the value of the whole expression above:

```
(cons 'a (cons 'b (remove-1st 'c '(c d))))
```

Once again, to get the value of this expression, we must evaluate the `remove-1st` subexpression. Now the list `(c d)` is not empty, but its first item *is* the same as `c`. Thus the condition in the second cond clause in the definition of `remove-1st` is true and the value of its consequent is `(d)`. Thus the above expression has the value

```
(cons 'a (cons 'b '(d)))
```

which can be simplified to give the value

```
(a b d)
```

This is the value returned by the procedure call. In the next section, we shall see how the computer can help us walk through a procedure application.

In order to be able to remove a sublist from a given list, as in Example 5, the predicate `equal?` was used to test for sameness in the second cond clause. If we know that all of the arguments to which `item` will be bound are symbols, we can use `eq?` to test for sameness. The procedure defined using `eq?` instead of `equal?` is named `remq-1st`. Similarly, if we restrict the arguments to which `item` will be bound to symbols or numbers, we can use `eqv?` to test for sameness in the second cond clause, and we name the procedure so defined `remv-1st`.

Exercises

Exercise 2.10

Rewrite the definitions of the three procedures `last-item`, `member?` and `remove-1st` with the cond expression replaced by if expressions.

Exercise 2.11

The definition of **member?** given in this section uses an or expression in the else clause. Rewrite the definition of **member?** so that each of the two subexpressions of the or expression is handled in a separate cond clause. Compare the resulting definition with the definition of **remove-1st**.

Exercise 2.12

The following procedure, named **mystery**, takes as its argument a list that contains at least two top-level items.

```
(define mystery
  (lambda (ls)
    (if (null? (cddr ls))
        (cons (car ls) '())
        (cons (car ls) (mystery (cdr ls))))))
```

What is the value of **(mystery '(1 2 3 4 5))**? Describe the general behavior of **mystery**. Suggest a good name for the procedure **mystery**.

Exercise 2.13: **subst-1st**

Define a procedure **subst-1st** that takes three parameters: an item **new**, an item **old**, and a list of items **ls**. The procedure **subst-1st** looks for the first top-level occurrence of the item **old** in **ls** and replaces it with the item **new**. Test your procedure on:

```
(subst-1st 'dog 'cat '(my cat is clever))
                          ⟹ (my dog is clever)
(subst-1st 'b 'a '(c a b a c))
                          ⟹ (c b b a c)

(subst-1st '(0) '(*) '((*) (1) (*) (2)))
                          ⟹ ((0) (1) (*) (2))
(subst-1st 'two 'one '()) ⟹  ()
```

In order to be able to include lists as possible arguments to which the parameters **new** and **old** are bound, use **equal?** to test for sameness. Also define procedures **substq-1st** and **substv-1st** that use **eq?** and **eqv?** respectively, instead of **equal?** to test for sameness.

Exercise 2.14: `insert-right-1st`

The procedure `insert-right-1st` is like `remove-1st` except that instead of removing the item that it is searching for, it inserts a new item to its right. For example,

```
(insert-right-1st 'not 'does '(my dog does have fleas))
                          ⟹ (my dog does not have fleas)
```

The definition of `insert-right-1st` is

```
(define insert-right-1st
   (lambda (new old ls)
      (cond
        ((null? ls) '())
        ((equal? (car ls) old)
         (cons old (cons new (cdr ls))))
        (else (cons (car ls)
                    (insert-right-1st new old (cdr ls)))))))
```

Define a procedure `insert-left-1st` that is like `insert-right-1st` except that instead of inserting a new item to the right of the item it is searching for, it inserts it to its left. Test your procedure on

```
(insert-left-1st 'hot 'dogs '(I eat dogs))
                              ⟹ (I eat hot dogs)
(insert-left-1st 'fun 'games '(some fun))
                              ⟹ (some fun)
(insert-left-1st 'a 'b '(a b c a b c))
                              ⟹ (a a b c a b c)
(insert-left-1st 'a 'b '()) ⟹ ()
```

Exercise 2.15: `list-of-first-items`

Define a procedure `list-of-first-items` that takes as its argument a list composed of nonempty lists of items. Its value is a list composed of the first top-level item in each of the sublists. Test your procedure on:

```
(list-of-first-items '((a) (b c d) (e f))) ⟹ (a b e)
(list-of-first-items '((1 2 3) (4 5 6))) ⟹ (1 4)
(list-of-first-items '((one))) ⟹ (one)
(list-of-first-items '()) ⟹ ()
```

Exercise 2.16: `replace`

Define a procedure `replace` that replaces each top-level item in a list of items `ls` by a given item `new-item`. Test your procedure on:

```
(replace 'no '(will you do me a favor))
                    ⟹ (no no no no no no)
(replace 'yes '(do you like ice cream))
                    ⟹ (yes yes yes yes yes)
(replace 'why '(not)) ⟹ (why)
(replace 'maybe '())  ⟹ ()
```

Exercise 2.17: `remove-2nd`

Define a procedure `remove-2nd` that removes the second occurrence of a given item `a` from a list of items `ls`. You may use the procedure `remove-1st` in defining `remove-2nd`. Test your procedure on:

```
(remove-2nd 'cat '(my cat loves cat food))
                    ⟹ (my cat loves food)
(remove-2nd 'cat '(my cat loves food))
                    ⟹ (my cat loves food)
(remove-2nd 'cat '(my cat and your cat love cat food))
                    ⟹ (my cat and your love cat food)
(remove-2nd 'cat '()) ⟹ ()
```

Exercise 2.18: `remove-last`

Define a procedure `remove-last` that removes the last top-level occurrence of a given element `item` in a list `ls`. Test your procedure on:

```
(remove-last 'a '(b a n a n a s)) ⟹ (b a n a n s)
(remove-last 'a '(b a n a n a)) ⟹ (b a n a n)
(remove-last 'a '()) ⟹ ()
```

Exercise 2.19: `sandwich-1st`

Define a procedure `sandwich-1st` that takes two items, `a` and `b`, and a list `ls` as its arguments. It replaces the first occurrence of two successive `b`'s in `ls` with `b a b`. Test your procedure on:

```
(sandwich-1st 'meat 'bread '(bread cheese bread bread))
                              ⟹ (bread cheese bread meat bread)
(sandwich-1st 'meat 'bread '(bread jam bread cheese bread))
                              ⟹ (bread jam bread cheese bread)
(sandwich-1st 'meat 'bread '()) ⟹ ()
```

Exercise 2.20: `list-of-symbols?`

Define a procedure `list-of-symbols?` that tests whether the top-level items in a given list `ls` are symbols. Write your definitions in three ways, first using `cond`, then `if`, and finally `and` and `or`. Test your procedures with:

```
(list-of-symbols? '(one two three four five)) ⟹ #t
(list-of-symbols? '(cat dog (hen pig) cow)) ⟹ #f
(list-of-symbols? '(a b 3 4 d)) ⟹ #f
(list-of-symbols? '()) ⟹ #t
```

Exercise 2.21: `all-same?`

Define a procedure `all-same?` that takes a list `ls` as its argument and tests whether all top-level elements of `ls` are the same. Test your procedure with:

```
(all-same? '(a a a a a)) ⟹ #t
(all-same? '(a b a b a b)) ⟹ #f
(all-same? '((a b) (a b) (a b))) ⟹ #t
(all-same? '(a)) ⟹ #t
(all-same? '()) ⟹ #t
```

2.5 Tracing and Debugging

We have now walked through several programs to understand their behavior. We had to evaluate expressions ourselves and make decisions as to which branches of conditional expressions to follow. The computer is able to do both of these, so we can take advantage of its power to relieve us of this kind of work. The tool we develop here enables us to walk through or, as it is technically known, *trace* our programs. We can also use this tool to find and correct errors in our programs, a process called *debugging*.

The computer can help us walk through or trace our programs if we make use of a procedure `writeln` (read as "write-line") that prints its arguments directly to the computer screen. Some Scheme implementations provide the procedure `writeln`, and if the one you are using does not make it available, you can enter its simple definition.[9] The procedure `writeln` takes any number of arguments. When we evaluate

[9] A more complete discussion of `writeln` and related procedures that write to the screen is presented in Chapter 7. You may enter the definition of `writeln` given in Program 7.5 if your implementation of Scheme does not provide it.

(`writeln` $expr_1$ $expr_2$ ... $expr_n$)

the expressions $expr_1$ $expr_2$ ... $expr_n$ are all evaluated; then their values are printed on the screen in order from left to right with no blank spaces between them. When the last value is printed, the cursor moves to the beginning of the next line. Like every other procedure, `writeln` must return a value, but we are not concerned with this value. In fact, different implementations of Scheme may return different values. Since it is *unspecified* in Scheme what value `writeln` returns, we shall assume in our implementation that the value returned is not printed on the screen.

For example, if the variable `Jack` is bound to the value `Jill` and the variable `Punch` is bound to the value `Judy`, the evaluation of `(writeln Punch Jack)` will print

```
JudyJill
```

on the screen with no space between the words. If we evaluate the expression `(writeln 'Punch 'Jack)`, then the screen shows

```
PunchJack
```

We can control the spacing and print sentences on the screen if we use another type of data called *strings*. A string is any sequence of keyboard characters. In Scheme, a string is written as a sequence of characters enclosed with double quotes: ". Thus `"This is a string."` is an example of a string. If we want to include a double quote or a backslash in a string, we must precede it by a backslash.[10] Thus, we can write the string `"He said \"Hello\"."`, which has `"Hello"` within double quotes. If we evaluate the expression

```
(writeln "This is a string.")
```

then

```
This is a string.
```

appears on the screen. Note that the double quotes are not printed with the string. Thus the evaluation of the expression

[10] A character, such as a backslash, which is used to change the normal meaning of what follows it is referred to as an *escape character*.

```
(writeln "He said \"Hello\".")
```

prints

```
He said "Hello".
```

A string is another example of a constant in Scheme. Thus if we enter a string in response to a prompt, the string is returned, including the double quotes.

```
[1] "This is a string."
"This is a string."
```

If we evaluate

```
(writeln "My friends Jack and " Jack ".")
```

we see on the screen:

```
My friends Jack and Jill.
```

The first occurrence of **Jack** is in the string, so it is printed literally as **Jack**. The second occurrence of **Jack** is not in a string, so it is evaluated, and its value **Jill** is printed. This time we have a space between the words **and** and **Jill**, since the blank space is included after the word **and** in the string. The last string in the writeln expression contains only the period.

The procedure **writeln** is usually evaluated as one of a sequence of expressions that are evaluated consecutively. This is accomplished by using the special form with keyword **begin**. A begin expression has any number of subexpressions following the keyword **begin**. Each of these subexpressions is evaluated consecutively in the order that it appears and the value of the last subexpression is returned as the value of the begin expression. For example,

```
[2] (begin
      (writeln "The remove-1st expression")
      (writeln "is applied to the list (1 2 3 4)")
      (writeln "to build a new list without the number 2.")
      (remove-1st 2 '(1 2 3 4)))
The remove-1st expression
is applied to the list (1 2 3 4)
to build a new list without the number 2.
(1 3 4)
```

When the preceding begin expression is evaluated, the four subexpressions are evaluated consecutively. The first three are writeln expressions, which print their arguments on the screen, with a new line starting after each writeln expression is evaluated. The values returned by the writeln expressions are ignored. The value of the last expression is the only value returned—that is the `(1 3 4)` that appears on the last line.

We want to stress that what is printed on the screen is not the value of the writeln expressions. Instead, what is printed on the screen is done as a *side effect*. A side effect causes some change to take place (in this case, the change was printing on the screen), but it is not a value that is returned. When using a begin expression, all of the subexpressions before the last one are included for their side effects and not for the values that they return. The value of the last subexpression is the only one returned. Here is another example to illustrate that only the value of the last subexpression is returned.

```
[3] (begin
      (+ 3 4)
      (- 5 11)
      (* 10 10))
100
```

The values of the first two subexpressions are ignored. In this case, the first two subexpressions did not produce any side effects, so although they were evaluated, we do not see any evidence of it and there really was no point in putting them there!

The syntax of the begin expression is

(**begin** $expr_1$ $expr_2$ ... $expr_n$)

where the expressions $expr_1$, $expr_2$, ... $expr_n$ are evaluated in their given order, and the value of the last one, $expr_n$, is returned.

We now have all the tools we need to use `writeln` to help us walk through an application of `remove-1st` to remove the letter `c` from the list `(a b c d)`. We "wrap" a helping procedure `entering` around the condition of each cond clause as we enter it and wrap a helping procedure `leaving` around the consequent (or alternative) as we leave the cond clause. The definitions of these helping procedures are given after the main program. The procedure `entering` takes three arguments: the value of the condition, the value of `ls`, and the identifying number of the cond clause: 1 for the first, 2 for the second, and 3 for the last. It tells us, using a writeln statement, which cond clause we are entering and the value of `ls`. The procedure `leaving` takes two arguments:

the value of the consequent (or alternative) and the identifying number of the cond clause. It tells us which cond clause we are leaving and the value of the consequent. When we run the program, we thus get a written record each time we enter or leave a cond clause. Inserting such writeln expressions into the definition of a procedure to study the evaluation of the procedure is one way of *tracing* the procedure. Program 2.5 contains the code for the procedure that traces `remove-1st`. The definition of the helping procedure `entering` is in Program 2.6, and of the helping procedure `leaving` is in Program 2.7.

When we enter a cond clause, the condition is the entering expression whose parameter `test` is bound to the value of the original condition of `remove-1st`. If `test` is true, it writes the fact that we are entering the cond clause with the appropriate identifying number and the current value of the variable `ls`. In any event, `test` is returned as the value of the condition. If `test` is false, the next cond clause is entered. If `test` is true, the consequent of that cond clause is evaluated. If the else clause is entered, we use the quoted symbol `else` as the first argument of `entering`. Scheme treats the symbol `else` as true (since it is not false) so the alternative is evaluated.

The consequent (or alternative) in each cond clause of `remove-1st-trace` is a leaving expression. It has the value of the original consequent (or alternative) of the cond clause of `remove-1st` as the binding of its first parameter, `result`. When the leaving expression is evaluated, it tells us the identifying number of the cond clause and the value to which `result` is bound. It then returns `result`.

Now let's apply `remove-1st-trace` to see how this tracing information helps us see what is happening during the evaluation.

```
[1] (remove-1st-trace 'c '(a b c d))
   Entering cond-clause-3 with ls = (a b c d)
   Entering cond-clause-3 with ls = (b c d)
   Entering cond-clause-2 with ls = (c d)
Leaving cond-clause-2 with result = (d)
Leaving cond-clause-3 with result = (b d)
Leaving cond-clause-3 with result = (a b d)
(a b d)
```

This output tells us that we first entered the third cond clause with `ls` bound to `(a b c d)`. With this binding, the leaving expression in the alternative is evaluated, so that its first operand

```
(cons 'a (remove-1st-trace 'c '(b c d)))          (1)
```

Program 2.5 `remove-1st-trace`

```
(define remove-1st-trace
  (lambda (item ls)
    (cond
      ((entering (null? ls) ls 1)
       (leaving '() 1))
      ((entering (equal? (car ls) item) ls 2)
       (leaving (cdr ls) 2))
      ((entering 'else ls 3)
       (leaving
         (cons (car ls) (remove-1st-trace item (cdr ls)))
         3)))))
```

Program 2.6 `entering`

```
(define entering
  (lambda (test input cond-clause-number)
    (begin
      (if test (writeln "   Entering cond-clause-"
                 cond-clause-number " with ls = " input))
      test)))
```

Program 2.7 `leaving`

```
(define leaving
  (lambda (result cond-clause-number)
    (begin
      (writeln "Leaving cond-clause-"
        cond-clause-number " with result = " result)
      result)))
```

is evaluated. Thus `remove-1st-trace` is called again, and `a` is waiting to be consed onto the value obtained before we can leave cond clause 3. The next message on the screen tells us that we are entering the third cond expression again with argument `(b c d)`. This time, the alternative

```
(cons 'b (remove-1st-trace 'c '(c d)))          (2)
```

is evaluated, and `b` is waiting to be consed onto its value before we can leave cond clause 3. As before, `remove-1st-trace` is called again before the leaving writeln expression is evaluated. This time, the first item in `(c d)` is the same as `c`, and we are told that we entered the second cond clause with `ls` bound to `(c d)`. When we enter the consequent, the first operand in the leaving expression evaluates to `(d)`. Then the writeln expression prints on the screen that we are leaving the second cond clause with the `result` bound to `(d)`, and the value `(d)` is returned.

Cons expression (2) is waiting for the value of the `remove-1st-trace` call, and now that value is `(d)`. With this value, the cons expression in (2) evaluates to `(b d)`. We can now complete the evaluation of the leaving expression, which tells us that we are leaving cond clause 3 with `result` bound to `(b d)`. But this is just the value that cons expression (1) is waiting for as the value of its `remove-1st-trace` invocation. Using the value `(b d)` as its last argument, cons expression (1) evaluates to `(a b d)`. It was the first operand in the application of `leaving` in the third cond clause. Now that it has been evaluated, the writeln expression writes its message, which says that we are leaving cond clause 3 with `result` bound to `(a b d)`. The `leaving` invocation now returns the value to which `result` is bound, `(a b d)`, and that becomes the value of the original procedure call. The trace we made here illustrates well the order in which we enter and leave the cond clauses. We see that we do not leave the cond clause until a value is found for the recursive invocation of `remove-1st-trace`, and the evaluation of the cons expression can be completed.

In the previous example, we entered only the second and third cond clauses. If we invoke `remove-1st-trace` to remove an item from a list that does not contain it, we enter only the first and third cond clauses, as the following trace illustrates:

```
[2] (remove-1st-trace 'e '(a b c d))
   Entering cond-clause-3 with ls = (a b c d)
   Entering cond-clause-3 with ls = (b c d)
   Entering cond-clause-3 with ls = (c d)
   Entering cond-clause-3 with ls = (d)
   Entering cond-clause-1 with ls = ()
Leaving cond-clause-1 with result = ()
Leaving cond-clause-3 with result = (d)
Leaving cond-clause-3 with result = (c d)
Leaving cond-clause-3 with result = (b c d)
Leaving cond-clause-3 with result = (a b c d)
(a b c d)
```

Analyze the trace to be sure you can explain it in a manner similar to that used in the previous example.

We have used writeln expressions to trace a program by printing certain information about places in the program where the evaluation is being made and the values of certain variables at that place. This helps us understand how programs work. It is also an excellent tool for finding errors in programs. If a program is not doing what you expect it to do, you can put a writeln expression at certain places in the program where you think the error may be and look at the values of variables to compare them with what you expect at that place. By studying these values, you can frequently pinpoint the source of the error and make the appropriate changes to cause the program to work correctly. When the program is corrected and runs as you want, the writeln expressions used to locate the errors should be removed. Tracing a program with the writeln expressions placed at strategic points is a helpful and often used debugging tool.

Exercise

Exercise 2.22

In the first trace, the second and third cond clauses were entered. In the second trace, the first and third cond clauses were entered. Can you give a **remove-1st-trace** invocation that enters only the first and second cond clauses? Explain.

The last example of recursion in this chapter is the procedure **swapper**, which takes three arguments: an item **x**, an item **y**, and a list **ls**. It builds a new list in which each top-level occurrence of **x** in **ls** is replaced by **y**, and each top-level occurrence of **y** in **ls** is replaced by **x**. We are "swapping" **x** and **y** in **ls**. For example,

```
(swapper 'cat 'dog '(my cat eats dog food))
                                    ⟹ (my dog eats cat food)
(swapper 'john 'mary '(john loves mary)) ⟹ (mary loves john)
(swapper 'a 'n '(b n a n a n)) ⟹ (b a n a n a)
(swapper 'a 'b '(c (a b) d)) ⟹ (c (a b) d)
(swapper 'a 'b '()) ⟹ ()
```

In the fourth example, the **a** and **b** in the list are not at top level, so they are not swapped.

In order to define **swapper**, we begin with an analysis of the base case. What is the simplest case for this problem? If **ls** is empty, there is nothing to swap and the empty list is returned. Thus we take as the base case for **ls** the empty list, and we begin the definition as follows:

```
(define swapper
  (lambda (x y ls)
    (cond
      ((null? ls) '())
      ... )))
```

A nonempty list is simplified to the base case using the simplifying operation **cdr**. What is returned if we invoke **(swapper x y (cdr ls))**? The result will be **(cdr ls)** with the items **x** and **y** interchanged. But this differs from **(swapper x y ls)** only in that the first item in **(swapper x y ls)** is missing. We will get **(swapper x y ls)** from **(swapper x y (cdr ls))** by consing the correct first item onto **(swapper x y (cdr ls))**. There are three possibilities for this first item: it can be **x**, **y**, or neither. First, if **(car ls)** is **x**, we should cons **y** onto **(swapper x y (cdr ls))**, so the next cond clause in our definition can be added:

```
(define swapper
  (lambda (x y ls)
    (cond
      ((null? ls) '())
      ((equal? (car ls) x)
       (cons y (swapper x y (cdr ls))))
      ... )))
```

Program 2.8 `swapper`

```
(define swapper
  (lambda (x y ls)
    (cond
      ((null? ls) '())
      ((equal? (car ls) x)
       (cons y (swapper x y (cdr ls))))
      ((equal? (car ls) y)
       (cons x (swapper x y (cdr ls))))
      (else
       (cons (car ls) (swapper x y (cdr ls)))))))
```

Second, if `(car ls)` is `y`, we should cons `x` onto `(swapper x y (cdr ls))`, so the next cond clause can be added:

```
(define swapper
  (lambda (x y ls)
    (cond
      ((null? ls) '())
      ((equal? (car ls) x)
       (cons y (swapper x y (cdr ls))))
      ((equal? (car ls) y)
       (cons x (swapper x y (cdr ls))))
      ... )))
```

Finally, if `(car ls)` is neither `x` nor `y`, then we just cons `(car ls)` itself onto `(swapper x y (cdr ls))`, giving us the else clause and completing the definition given in Program 2.8.

If we invoke the procedure `swapper` with the arguments `'b`, `'d`, and `'(a b c d b)`, it should return the list `(a d c b d)` in which `b` and `d` have been interchanged. Let's walk through the program to see how it constructs this answer. In the first procedure call, `ls` is bound to `(a b c d b)`. This list is not empty, and its car is neither `b` nor `d`, so the else clause is evaluated and gives as the answer the cons expression:

```
(cons 'a (swapper 'b 'd '(b c d b)))
```

Let's refer to the value of this cons expression as *answer-1*, and that is the value that we are looking for to solve the problem. At this point, however,

we have not yet evaluated the recursive invocation of `swapper`, so let's give its value the name *answer-2*. We can now rewrite *answer-1* as

answer-1 is: `(cons 'a` *answer-2*`)`
answer-2 is: `(swapper 'b 'd '(b c d b))`

We see that *answer-1* is waiting for the value of *answer-2*, so we move on to evaluating *answer-2* and we shall return to get the value of *answer-1* when *answer-2* is known.

To evaluate *answer-2*, we observe that the list `(b c d b)` begins with `b`, so the second cond clause is the one with the true condition, and evaluating its consequent gives us

answer-1 is: `(cons 'a` *answer-2*`)`
answer-2 is: `(cons 'd` *answer-3*`)`
answer-3 is: `(swapper 'b 'd '(c d b))`

We still do not have a value for *answer-3*, so we once again set aside *answer-2* until we have a value for *answer-3*. Note that we are making a table of these various answers, with each successive entry placed below the preceding one. We shall often refer to this table, so we give it the name *return table*.

To evaluate *answer-3*, we see that `(c d b)` is not empty, and does not begin with `b` or `d`, so the alternative in the else clause is evaluated. We get for *answer-3*

```
(cons 'c (swapper 'b 'd '(d b)))
```

and we give the invocation of `swapper` within *answer-3* the name *answer-4*. This gives us the return table:

answer-1 is: `(cons 'a` *answer-2*`)`
answer-2 is: `(cons 'd` *answer-3*`)`
answer-3 is: `(cons 'c` *answer-4*`)`
answer-4 is: `(swapper 'b 'd '(d b))`

We have added *answer-3* to our return table to wait until we have the value of *answer-4*.

For the invocation of `swapper` in *answer-4*, the condition in the third cond clause is true, so our return table now becomes

answer-1 is: `(cons 'a` *answer-2*`)`
answer-2 is: `(cons 'd` *answer-3*`)`
answer-3 is: `(cons 'c` *answer-4*`)`
answer-4 is: `(cons 'b` *answer-5*`)`
answer-5 is: `(swapper 'b 'c '(b))`

We have added *answer-4* to our return table to wait for a value for *answer-5*.

For the invocation of **swapper** in *answer-5*, the condition in the second cond clause is true, so the return table now becomes

answer-1 is:	`(cons 'a answer-2)`
answer-2 is:	`(cons 'd answer-3)`
answer-3 is:	`(cons 'c answer-4)`
answer-4 is:	`(cons 'b answer-5)`
answer-5 is:	`(cons 'd answer-6)`
answer-6 is:	`(swapper 'b 'd '())`

Once again we have added *answer-5* to the return table to wait until we have a value for *answer-6*. In the invocation of **swapper** in *answer-6*, the terminating condition in the first cond clause is true, and the value () is returned for *answer-6*.

What effect does this termination have on the return table? Although we have a value for *answer-6*, the computation does not stop, for we have to get the values of each of the waiting variables in our return table. Until now, on each recursive invocation of **swapper**, a new row was added to the return table waiting for a value. This time we got a value for *answer-6*, so we do not have to add a row to the return table. Instead we replace the **swapper** expression in the last row by its value `()`. We can now work our way back up the table one row at a time, replacing each variable on the right side by the value it has on the next row below. We shall write these replacements in a new table, starting with the value for *answer-6*.

answer-6 is:	`()`
answer-5 is:	`(d)`
answer-4 is:	`(b d)`
answer-3 is:	`(c b d)`
answer-2 is:	`(d c b d)`
answer-1 is:	`(a d c b d)`

The last row gives us the anticipated value for our invocation of **swapper**.

Let's take another look at the definition of the procedure **swapper**. In the last three cond clauses, something is consed onto

```
(swapper x y (cdr ls))
```

What that something should be is determined by testing the value of `(car ls)`. We can write a helping procedure **swap-tester** that makes the test and returns the correct value to be consed onto

```
(swapper x y (cdr ls))
```

Assuming that we have such a test procedure, we can rewrite the definition of **swapper** as follows:

```
(define swapper
  (lambda (x y ls)
    (cond
      ((null? ls) '())
      (else (cons (swap-tester x y (car ls))
                  (swapper x y (cdr ls)))))))
```

We now define the helping procedure **swap-tester** to distinguish the three cases for us:

```
(define swap-tester
  (lambda (x y a)
    (cond
      ((equal? a x) y)
      ((equal? a y) x)
      (else a))))
```

When **swap-tester** is called within **swapper**, the arguments **x**, **y**, and **(car ls)** are substituted for the parameters **x**, **y** and **a**, respectively, and **swap-tester** returns the correct value to be consed onto

```
(swapper x y (cdr ls))
```

The use of such helping procedures often simplifies the writing and reading of programs. We shall make frequent use of this technique.

We could also have achieved the same effect without using the helping procedure **swap-tester** by using in **swapper** the cond expression of **swap-tester** in place of calling **swap-tester**. This leads to another version of **swapper**:

```
(define swapper
  (lambda (x y ls)
    (cond
      ((null? ls) '())
      (else (cons (cond
                    ((equal? (car ls) x) y)
                    ((equal? (car ls) y) x)
                    (else (car ls)))
                  (swapper x y (cdr ls)))))))
```

In this section, we have seen how to use writeln expressions to trace or debug a program. We have also seen how a return table is created when a recursive procedure is evaluated.

Exercises

Exercise 2.23
Identify what is printed on the screen and what is returned in each of the following:

a.
```
(begin
  (writeln "(* 3 4) = " (* 3 4))
  (= (* 3 4) 12))
```

b.
```
(begin
  (writeln "(cons 'a '(b c)) has the value " (cons 'a '(b c)))
  (writeln "(cons 'a '(b c)) has the value " '(a b c))
  (writeln "(cons 'a '(b c)) has the value (a b c)")
  (cons 'a '(b c)))
```

c.
```
(begin
  (writeln "Hello, how are you?")
  (writeln "Fine, thank you.  How are you? " 'Jack)
  (writeln "Just great!  It is good to see you again, " 'Jill)
  "Good-bye.  Have a nice day.")
```

Exercise 2.24: **describe**
With **describe** defined as

```
(define describe
  (lambda (s)
    (cond
      ((null? s) (quote '()))
      ((number? s) s)
      ((symbol? s) (list 'quote s))
      ((pair? s) (list 'cons (describe (car s)) (describe (cdr s))))
      (else s))))
```

evaluate each of the following expressions:

a. **(describe 347)**

b. **(describe 'hello)**

c. `(describe '(1 2 button my shoe))`

d. `(describe '(a (b c (d e) f g) h))`

Describe what `describe` does in general.

Exercise 2.25

Write a trace similar to the one used in `remove-1st-trace` to trace the procedure `swapper`, showing the binding of the parameter `ls` each time the cond expression is entered and whenever a cond clause is exited. Invoke the traced procedure `swapper-trace` on the arguments `b`, `d`, and `(a b c d b)` used in the example in this section.

Exercise 2.26

In the return table built for the invocation of `swapper` in this section, the computation did not stop when the terminating condition was true and the first cond clause returned `()`. Instead, the variables in the table were evaluated one by one until the value of the first was obtained to provide the value of the original invocation. This program behaved in this way because after each invocation of `swapper`, a `cons` still had to be completed. There was still an operation to perform after `swapper` was invoked. Do a similar analysis, building the return tables, on the two procedures `last-item` in Program 2.2 and `member?` in Program 2.3. In the first case, consider `(last-item '(a b c))`, and in the second case, consider `(member? 'c '(a b c d))`. In these two examples, there is no procedure waiting to be done after the recursive invocations of the procedure. Such programs are called *iterative*. We shall discuss the behavior of iterative programs more thoroughly in the chapter on numerical recursion.

Exercise 2.27

Does the answer change if cond clause 2 and cond clause 3 are interchanged in the definition of `swapper`? Does the same thing hold if cond clauses 1 and 2 are interchanged in `swap-tester`?

Exercise 2.28: `tracing`, `test-tracing`

A more generally applicable tracing tool than the procedure `leaving` given in Program 2.7 is the procedure `tracing` defined by

```
(define tracing
  (lambda (message result)
    (begin
      (writeln message result)
      result)))
```

Similarly, the procedure `test-tracing` defined by

```
(define test-tracing
  (lambda (test message input)
    (begin
      (if test (tracing message input))
      test)))
```

is useful for tracing the test part of a conditional expression. Rewrite the definition of `remove-1st-trace` using `test-tracing` and `tracing` instead of `entering` and `leaving` in such a way as to produce *exactly* the same output as that generated using `entering` and `leaving`.
