CHAPTER SEVEN

Lists

7.1 The Definition of a List

In Chapter 6, we learned how to construct abstract data types where any element of the type had the same number of components as any other. Sometimes, however, the individual data will have varying sizes. One example is lists. In this chapter, we first give a formal definition of lists. Then we show how this definition is used to construct lists, and we explore common list-processing idioms.

Exactly what is a list? We could start thinking of some concrete examples of lists by having everyone in our class make a list of the classes he or she attended yesterday. Some people may have relatively long lists, and others may have been to just one or two classes. Some people may have not have attended any classes at all. (If today is Monday, perhaps everyone is in this situation.) Lists can be written in a variety of ways as well. Some people might write lists in a column, others might use rows, and still others may come up with more creative ways of doing it. No matter how the lists are written, those lists that have at least one element have some inherent order (i.e., each list can be written to start with the first class of the day and work its way forward from there). Thus we could define a list as being a collection of 0 or more elements, written in some particular order.

Now imagine that we need to write a computer program to deal with lists of some sort of elements, say, integers for simplicity. Our first job is to decide what procedures we need to define the abstract data type of integer lists. The definition we gave in the foregoing is not much of a guide. Clearly, we are going to be implementing compound data, because some of our lists may have quite a few components. But how many components are we going to need? With the abstract data types that we considered in Chapter 6, each individual piece of data had exactly the same number of components as any other piece. Different lists, on the other hand, could have wildly different sizes. So what do we do?

Let's think about this question a little more. In Chapter 6 we learned how to construct abstract data types where all values belonging to a particular type had the same number of components as each other. Now we want to have values of varying sizes belong to a single type. This parallels what happened moving from Chapter 1 to Chapter 2. In Chapter 1 we wrote procedures where all processes generated by a particular procedure were of the same size as each other. In Chapter 2 we asked how to have a single procedure generate processes of varying size. There the answer was recursion.

We also learned in Chapter 6 that the best way to define an abstract data type is to first concentrate on how the type is used. We can illustrate how lists are used by looking at the example of grocery lists. One of the authors usually constructs grocery lists so that the items are ordered by their positions in the grocery store. The list is then used by finding the first thing on it, putting that item into the cart, and crossing it off the list. This gives us a new grocery list (with one less item) that is used to continue the grocery shopping. Eventually, there is only one item left on the list (the chocolate candy bar located just before the checkout counter). After putting this item into the cart and crossing it off the list, the grocery list is empty. This means that it is time to check out and pay for the groceries.

This grocery list example has a strong recursive flavor to it, and so we can use it as a model for a recursive definition of lists. We will need a "base case" that defines the smallest possible list, and we will need some way of defining larger lists in terms of smaller lists. The base case is easy—there is a special list, called the *empty list*, which has no elements in it. The general case is hinted at in the grocery list example above: a nonempty list has two parts, one of which is an element (the first item in the list), and the other is a list, namely, the list of all the other items in the whole list. We can put this more succinctly:

> **The two-part list viewpoint:** A list is either empty or it consists of two parts: the first item in the list and the list of its remaining items.

The first element of a nonempty list is often called the *head* of the list, whereas the list of remaining elements is called the *tail*. The two-part list viewpoint is one of the things that distinguish a computer scientist from a normal person. Normal people think lists can have any number of components (the items on the list), whereas computer scientists think that all nonempty lists have two components (the head and the tail). The tail isn't one item that's on the list; it's a whole list of items itself.

How can we implement lists in Scheme? Given that most lists have two parts, it would be natural to use pairs: Let the car of the pair be the first element of the list, and let the cdr of the pair be the tail. However, because a list may be empty and would therefore not have two parts, we need to account for empty lists as well. Scheme does so by having a special type of value called the empty list, which we explain in the next section, and a predicate `null?` that tests whether a given list is

empty. Using `null?` in conjunction with the pair operators `cons`, `car`, and `cdr`, we can implement the list ADT in Scheme by adopting the following conventions:

`(cons` *elt lst*`)` Given an element *elt* and a list *lst*, `cons` returns the list whose head is *elt* and whose tail is *lst*.

`(car` *lst*`)` If *lst* is a nonempty list, `car` returns its head.

`(cdr` *lst*`)` If *lst* is a nonempty list, `cdr` returns its tail.

`(null?` *lst*`)` `Null?` returns true if and only if *lst* is the empty list.

Of course, a better data-abstraction practice would be to define a separate set of procedures, perhaps called `make-list`, `head`, `tail`, and `empty-list?`, that would keep the separation between the list abstraction and the particular representation using pairs. However, because this particular representation of lists is such a long-established tradition, Scheme programmers normally just use the pair operations as though they were also list operations. For example, the `car` and `cdr` selectors of the pair data type are traditionally used as though they were also the head and tail selectors of the list data type. You can always define `head` and `tail` and use them in your programming, if you'd rather, but you'll be in a small minority.

Note that these procedures do what is described above only when all parameters that need to be lists have themselves been constructed using these conventions. Furthermore, one common mistake to avoid is applying `car` or `cdr` to the empty list.

A number of the procedures we will write are actually built into Scheme. We're going to write them anyway because they provide excellent examples of list processing techniques. Furthermore, by writing them, you should gain a better understanding of what the built-in procedures do.

7.2 Constructing Lists

How do we make lists in Scheme? Fundamentally, all nonempty lists are made by using the pair-constructor `cons`. However, rather than using `cons` directly, we can also use some other procedure that itself uses `cons`. For example, Scheme has a built-in procedure, `list`, that you can use to build a list if you know exactly what elements you want to be in your list. You pass the list elements into `list` as its arguments:

```
(list 1 2 3)
(1 2 3)
```

Note that we let Scheme display the resulting list value by itself and that the displayed value consisted of the elements of the list in order, separated by spaces and

surrounded by a pair of parentheses. The fact that we allowed Scheme to display the list is one exception to the general rule stated in the preceding chapter, where we said that the way Scheme displays pairs is confusing, and therefore you should write special purpose display procedures, such as `display-game-state`. In the case where pairs are used to represent lists, the way Scheme displays them is simple and natural, so there is no need to write something like `display-list` ourselves.

Another thing to note is that the stuff that gets printed out when a list is displayed is not an expression that will evaluate to the list. Just as you can't make a game state by evaluating the output from `display-game-state`, namely, something like "Pile 1: 5 Pile 2: 8," so too you can't make a list by evaluating `(1 2 3)`. If you were to evaluate that, it would try to apply 1 as a procedure to 2 and 3, which fails because 1 isn't a procedure. This particular list was displayed in a way that looked like an erroneous Scheme expression. Other lists are displayed in ways that look like valid Scheme expressions; for example, if the first element of the list were the symbol `+` rather than the number 1, the list would display as `(+ 2 3)`. Even evaluating this expression won't produce the list `(+ 2 3)`. (From this past sentence onward, we will take a shortcut and say things like "the list `(+ 2 3)`" when what we really mean is "the list that, when displayed, looks like `(+ 2 3)`.")

Exercise 7.1

You also can't get the list `(+ 2 3)` by evaluating `(list + 2 3)`.

a. What *do* you get if you evaluate that expression? Explain why.

b. What expression can you evaluate that will produce the list `(+ 2 3)`, which starts with the symbol `+`?

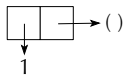
We've seen that when you want the list `(+ 2 3)`, you can't just type in `(+ 2 3)`. For example, you couldn't find the cdr of this list by evaluating `(cdr (+ 2 3))`, because that would try to find the cdr of 5. One option would be to get some procedure to build the list you want for you, instead of typing it in. For example, you could use `list`. There is one other option, however, that lets you type in a list as itself. You can use the same quoting mechanism that you use to type in symbols. For example,

```
(cdr '(1 2 3))
(2 3)
```

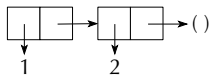
Lists and symbols both need quoting for the same reason: They look like expressions, but we want the name itself or the list itself, not the result of evaluating the name or list. Quote also gives us a way to get the empty list, namely, as the value of the expression `'()`.

What if we want to construct a list that is too long to type? For example, suppose we needed the list of integers from 1 to 2000. Rather than tediously typing in the whole list, or typing in all 2000 arguments to the `list` procedure, we should automate the process by writing a procedure to do it for us. As in the factorial example in the second chapter, it makes sense to write a general purpose procedure that produces the list of integers from *low* to *high* and then use it on `1` and `2000`. To see how to write such a procedure, let's first construct some fairly short lists, using `cons`. We'll also draw a box and pointer diagram of each of these lists, using the technique from Section 6.4, to illustrate the structure of lists:

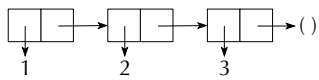
```
(cons 1 '())
(1)
```




```
(cons 1 (cons 2 '()))
(1 2)
```




```
(cons 1 (cons 2 (cons 3 '())))
(1 2 3)
```



Note that to get the list of integers from `1` to `3`, we consed `1` to the list of integers from `2` to `3`, which shows us how to write the general procedure:

```
(define integers-from-to
  (lambda (low high)
    (if (> low high)
        '()
        (cons low
              (integers-from-to (+ 1 low) high)))))
```

Such lists can then be created by making calls like the following one:

```
(integers-from-to 1 7)
(1 2 3 4 5 6 7)
```

This technique of using `cons` to recursively construct a list is often called *consing up a list.*

Exercise 7.2

What do you get if you evaluate `(integers-from-to 7 1)`? Exactly which integers will be included in the list that is the value of `(integers-from-to` *low high*`)`? More precisely, describe exactly when a given integer *k* will be included in the list

that is the value of `(integers-from-to` *low high*`)`. (You can do so by describing how *k*, *low*, and *high* are related to each other.) Do you think the result of `integers-from-to` needs to be more carefully specified (for example, in a comment) than the implicit specification via the procedure's name? Or do you think the behavior of the procedure should be changed? Discuss.

Exercise 7.3

Write a procedure that will generate the list of even integers from *a* to *b*.

Exercise 7.4

We could rewrite `integers-from-to` so that it generates an iterative process. Consider the following attempt at this:

```
(define integers-from-to  ; faulty version
  (lambda (low high)
    (define iter
      (lambda (low lst)
        (if (> low high)
            lst
            (iter (+ 1 low)
                  (cons low lst)))))
    (iter low '())))
```

What happens when we evaluate `(integers-from-to 2 7)`? Why? Rewrite this procedure so that it generates the correct list.

7.3 Basic List Processing Techniques

Suppose we need to write a procedure that counts the number of elements in a list. We can use the recursive definition of lists to help us define exactly what we mean by the number of elements in a list. Recall that a list is either empty or it has two parts, a first element and the list of its remaining elements. When a list is empty, the number of elements in it is zero. When it isn't empty, the number of elements is one more than the number of elements in its tail. We can write this in Scheme as

```
(define length
  (lambda (lst)
    (if (null? lst)
        0
        (+ 1 (length (cdr lst))))))
```

Similarly, to write a procedure that finds the sum of a list of integers, we would do the following:

```
(define sum
  (lambda (lst)
    (if (null? lst)
        0
        (+ (car lst) (sum (cdr lst))))))
```

Notice how similar these procedures are to recursive procedures with integer parameters, such as `factorial`. The base case in `length` and `sum` occurs when the list is empty, just as the base case in `factorial` is when the integer is 0. In `factorial` we reduced our integer by subtracting 1, and in `length` and `sum`, we reduce our list by taking its cdr. Procedures that traverse a list by working on the cdr in this manner are said to *cdr down a list*.

Exercise 7.5

Generalize `sum` to a higher-order procedure that can accumulate together the elements of a list in an arbitrary fashion by using a combining procedure (such as `+`) specified by a procedural parameter. When the list is empty, `sum` returned 0, but this result isn't appropriate for other combining procedures. For example, if the combining procedure is `*`, 1 would be the appropriate value for an empty list. (Why?) Following are two possible approaches to this problem:

a. Write the higher-order procedure so that it only works for nonempty lists. That way, the base case can be for one-element lists, in which case the one element can be returned.

b. Write the higher-order procedure so that it takes an additional argument, beyond the list and the combining procedure, that specifies the value to return for an empty list.

Exercise 7.6

a. Write a procedure that will count the number of times a particular element occurs in a given list.

b. Generalize this procedure to one that will count the number of elements in a given list that satisfy a given predicate.

Exercise 7.7

In addition to the procedure `length`, Scheme has a built-in procedure `list-ref` that returns a specified element of a list. More precisely, a call of the form `(list-ref lst n)` will return the $(n + 1)$st element of *lst*, because by convention $n = 0$ returns the first element, $n = 1$ returns the second, etc. Try this procedure for various parameter values. Write this procedure yourself.

Exercise 7.8

Here are some more exercises in cdring down a list:

a. Write a predicate that will determine whether or not a particular element is in a list.

b. Generalize this to a predicate that will determine whether any element of a list satisfies a given predicate.

c. Write a procedure that will find and return the first element of a list that satisfies a given predicate.

d. Write a procedure that will determine whether all elements of a list satisfy a given predicate.

e. Write a procedure that will find the position of a particular element in a list. For example,

```
(position 50 '(10 20 30 40 50 3 2 1))
4
```

Notice that we are using the same convention for position as is used in `list-ref`, namely, the first position is 0, etc. What should be returned if the element is not in the list? What should be returned if the element appears more than once in the list?

f. Write a procedure that will find the largest element in a nonempty list.

g. Write a procedure that will find the position of the largest element in a nonempty list. Specify how you are breaking ties.

Exercise 7.9

This exercise involves cdring down two lists.

a. Write a procedure that gets two lists of integers of the same size and returns true when each element in the first list is less than the corresponding element in the second list. For example,

```
(list-< '(1 2 3 4) '(2 3 4 5))
#t
```

What should happen if the lists are not the same size?

b. Generalize this procedure to one called `lists-compare?`. This procedure should get three arguments; the first is a predicate that takes two arguments (such as `<`) and the other two are lists. It returns true if and only if the predicate always returns true on corresponding elements of the lists. We could redefine `list-<` in the following manner:

```
(define list-<
  (lambda (l1 l2)
    (lists-compare? < l1 l2)))
```

We frequently will use lists to build other lists, in which case we cdr down one list while consing up the other. To illustrate what we mean, here is a simple procedure that selects those elements of a given list that satisfy a given predicate:

```
(define filter
  (lambda (ok? lst)
    (cond ((null? lst)
           '())
          ((ok? (car lst))
           (cons (car lst) (filter ok? (cdr lst))))
          (else
           (filter ok? (cdr lst))))))

(filter odd? (integers-from-to 1 15))
(1 3 5 7 9 11 13 15)
```

At this point, we've seen enough isolated examples of list processing procedures. Let's embark on a larger-scale project that will naturally involve list-processing. Consider the following remarkable fact: After doing a certain small number of "perfect shuffles" on a 52 card deck, the deck always returns to its original order. (By a perfect shuffle, we mean that the deck is divided into two equal parts, which are then combined in a strictly alternating fashion starting with the first card in the first half.) How many perfect shuffles are required to return a 52 card deck to its original order?

We can represent our original deck as a list of the numbers 1 to 52 using the procedure `integers-from-to`. How do we divide the deck into two equal halves? We can write two general purpose procedures, one to get the first however many elements of a list and the other to get the remaining elements.

Let's first write the procedure that will construct a list of the first *n* elements of a given list. This procedure is very similar to the procedure `list-ref` in Exercise 7.7:

```
(define first-elements-of
  (lambda (n list)
    (if (= n 0)
        '()
        (cons (car list)
              (first-elements-of (- n 1)
                                 (cdr list))))))
```

Exercise 7.10

Write the procedure `list-tail`, that gets a list and an integer *n* and returns the list of all but the first *n* elements in the original list. (`List-tail` is actually already built into Scheme.)

For any given value of *n* in the range $0 \leq n \leq$ `(length lst)`, the procedures `first-elements-of` and `list-tail` can be used to split `lst` into two parts.

Once we've cut our deck into two halves, we still need to combine those halves into the shuffled deck. We combine them by using the procedure `interleave`, which takes two lists and combines them into a single list in an alternating manner:

```
(define interleave    ; interleaves lst1 and lst2, starting with
  (lambda (lst1 lst2) ; the first element of lst1 (if any)
    (if (null? lst1)
        lst2
        (cons (car lst1)
              (interleave lst2 (cdr lst1))))))
```

To see why `interleave` works correctly, focus on the comment, which says that the first element in the result is going to be the first element from `lst1` (i.e., the first element of the first argument). What does the rest of the result look like, after that first element? If you interleave a stack of red cards with a stack of black cards, so that the top card is red, what does the rest of it look like? Well, the rest (under that top red card) will start with a black card and then will alternate colors. It will include all of the black cards and all the rest of the red cards. In other words, it is the result of interleaving the black cards with the rest of the red cards. This explains why in the recursive call to `interleave`, we pass in `lst2` as the first argument (so that the first element from `lst2` winds up right after the first element of `lst1` in the result) and then the cdr of `lst1`.

Combining `interleave`, `list-tail`, and `first-elements-of`, we can now define the procedure `shuffle`, which takes as arguments the deck as well as its size:

```
(define shuffle
  (lambda (deck size)
    (let ((half (quotient (+ size 1) 2)))
      (interleave (first-elements-of half deck)
                  (list-tail deck half)))))
```

The purpose of the parameter `size` is efficiency—otherwise we would need to use the procedure `length`. We write `(quotient (+ size 1) 2)` instead of the more natural `(quotient size 2)` in order to ensure that when *size* is odd, the first half of the deck has the extra card. Notice that when *size* = 52, `(quotient (+ size 1) 2)` = 26.

In order to find out how many shuffles are needed, we write the following procedure, which automates multiple shuffles:

```
(define multiple-shuffle
  (lambda (deck size times)
    (if (= times 0)
        deck
        (multiple-shuffle (shuffle deck size)
                          size (- times 1)))))
```

We can then find out how many shuffles are needed by making calls as follows:

```
(multiple-shuffle (integers-from-to 1 52) 52 1)
(1 27 2 28 3 29 4 30 5 31 6 32 7 33 8 34 9 35 10 36 11 37 12
38 13 39 14 40 15 41 16 42 17 43 18 44 19 45 20 46 21 47 22 48
23 49 24 50 25 51 26 52)

(multiple-shuffle (integers-from-to 1 52) 52 2)
(1 14 27 40 2 15 28 41 3 16 29 42 4 17 30 43 5 18 31 44 6 19
32 45 7 20 33 46 8 21 34 47 9 22 35 48 10 23 36 49 11 24 37 50
12 25 38 51 13 26 39 52)

                                  ⋮

(multiple-shuffle (integers-from-to 1 52) 52 8)
(1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44
45 46 47 48 49 50 51 52)
```

Thus, eight perfect shuffles return the deck to its original order.

Exercise 7.11

We have written `shuffle` so that it can operate on decks of any size. In fact, decks of all sizes have the property that after a certain number of perfect shuffles, the deck is returned to its original order. In this exercise you will write procedures that will automate the process of finding the number of shuffles, which we call the *shuffle-number*, required for a deck of a given size.

a. Given that you start with an ordered deck, the first thing you will need is a predicate, called `in-order?`, that determines whether a list of integers is in increasing order. Write this procedure.

b. Using `in-order?`, write a procedure `shuffle-number` that, when passed a positive integer n, returns the shuffle-number for n. You should start off with an ordered deck of size n and repeatedly shuffle until the deck is in order.

Exercise 7.12

Throughout this section on perfect shuffles, we've been passing in the size of the deck as well as the list representing the deck itself in order to avoid computing the length of the list when we already knew it. Another approach would be to create a new compound data type for decks, with two selectors: one to get the list of elements and the other to get the length. That way we could pass in just a single thing, the deck, but could still find the length without counting. Flesh out the remaining details of this idea, implement it, and try it out.

We end this section with another example of using higher-order programming with lists. Suppose you wanted to find the shuffle-number for decks of size 1, 2, 3, . . . , 100 so that you could look at them all and see if there seemed to be any pattern. Rather than manually applying your `shuffle-number` procedure to each of the integers from 1 to 100, you could get a list of those integers, using `integers-from-to`, and then use some general purpose higher-order procedure to *map* each element of that list into its `shuffle-number`. A procedure called `map` that is built into Scheme does this mapping. Its first argument is the procedure to use for the mapping, and its second argument is the list of values that should be mapped. So in order to get the shuffle numbers for decks ranging in size from 1 to 100, we could do the following:

```
(map shuffle-number (integers-from-to 1 100))
```

Or, we could find the squares of 5, 12, and 13 by evaluating

```
(map (lambda (x) (* x x)) '(5 12 13))
(25 144 169)
```

Exercise 7.13

The procedure `map` is extraordinarily handy for creating lists of all sorts. Each of the following problems can be solved by using `map`.

a. Write a procedure that, when given a positive integer *n*, returns a list of the first *n* perfect squares.

b. Write a procedure that, when given a positive integer *n*, returns a list of the first *n* even integers.

c. Write a procedure called `sevens` that, when given a positive integer *n*, returns a list of *n* sevens. For example:

```
(sevens 5)
(7 7 7 7 7)
```

d. Write a procedure that, when given a list of positive integers, returns a list of lists of integers. Each of these lists should be the positive integers from 1 to whatever was in the original list. For example,

```
(list-of-lists '(1 5 3))
((1) (1 2 3 4 5) (1 2 3))
```

Exercise 7.14

Even though `map` is built into Scheme, it is a good exercise to write it yourself. Do so.

7.4 List Processing and Iteration

A palindrome is a word, such as *madam*, that stays unchanged when you write the letters in reverse order. Sometimes, entire sentences are palindromes, if you ignore spaces and punctuation; one of the classic examples is "Madam, I'm Adam." In this section, we'll test lists of symbols to see whether they are palindromes when viewed symbol by symbol rather than letter by letter. What we mean by this is that reversing the order of the elements of the list leaves it unchanged. For example, the list `(m a d a m)` is a palindrome and so is `(record my record)`.

We can determine whether or not a list of symbols is a palindrome by reversing it and seeing if the result is equal to the original list. We can do the equality testing using `equal?` but need to figure out how to reverse the list. Actually, as so often,

there is a procedure built into Scheme called `reverse` that does just what we need. But in the best of no-pain/no-gain tradition, we'll write it ourselves.

Reversing an empty list is quite easy. To reverse a nonempty list, one approach is to first reverse the cdr of the list and then to stick the car onto the end of this reversed cdr. The major obstacle is that we don't have any procedure currently in our toolbox for tacking an element onto the end of a list. What we want is a procedure `add-to-end` such that `(add-to-end '(1 2 3) 4)` would evaluate to the list `(1 2 3 4)`. We can write this procedure in our usual recursive way, cdring down the list and consing up the result:

```
(define add-to-end
  (lambda (lst elt)
    (if (null? lst)    ; adding to an empty list
        (cons elt '()) ;  makes a one-element list
        (cons (car lst)
              (add-to-end (cdr lst)
                          elt)))))
```

Given this, we can write `reverse` as follows:

```
(define reverse
  (lambda (lst)
    (if (null? lst)
        '()
        (add-to-end (reverse (cdr lst))
                    (car lst)))))
```

This way of reversing a list is very time consuming because of the call to `add-to-end`. A good way to measure how much time it takes is to count up the number of times `cons` is called. Adding to the end of a k-element list will make $k + 1$ calls to `cons`. Suppose we use $R(n)$ to denote the number of `cons`es that `reverse` does (indirectly, by way of `add-to-end`) when reversing a list of size n. Then we know that $R(0) = 0$ because `reverse` simply returns the empty list when its argument is empty. When the argument to `reverse` is a nonempty list, the number of calls to `cons` will be however many are done by reversing the cdr of the list plus however many are done by adding to the end of this reversed cdr. Thus,

$$R(n) = R(n - 1) + ((n - 1) + 1) = R(n - 1) + n$$

But by the same argument

$$R(n - 1) = R(n - 2) + ((n - 2) + 1) = R(n - 2) + (n - 1)$$

so we get

$$\begin{aligned} R(n) &= R(n-1) + n \\ &= R(n-2) + (n-1) + n \\ &\vdots \\ &= R(0) + 1 + 2 + 3 + \cdots + (n-2) + (n-1) + n \\ &= 0 + 1 + 2 + 3 + \cdots + (n-2) + (n-1) + n \\ &= \frac{n(n+1)}{2} \end{aligned}$$

Therefore the number of `cons`es done by this version of `reverse` is $\Theta(n^2)$. (The last equation, expressing the sum as $n(n+1)/2$, is from the solution to Exercise 4.1 on page 81. Even without it you could figure out that the sum was $\Theta(n^2)$ using the reasoning given in Section 4.1.) We can expect the time taken to similarly be $\Theta(n^2)$, which implies that as lists get longer, reversing them will slow down more than proportionately quickly. A 200-element list is likely to take 4 times as long to reverse as a 100-element list, rather than only twice as long.

There must be a better way of reversing a list. In fact, if you remember Exercise 7.4, our initial attempt to write an iterative procedure that generates the list of integers from *a* to *b* produced a list with the right numbers but in reverse order. Although that was a mistake there, it suggests an iterative strategy for reversing a list.

Before trying to write an iterative `reverse`, a concrete example might be helpful. Put a stack of cards on the table face up in front of you and reverse the order of them, leaving them face up. Chances are you did it by taking the first card off of top of the stack and setting it down elsewhere on the table, then moving the next card from the top of the original stack to the top of the new stack, etc., until all the cards had been moved. If you interrupt this process somewhere in the middle and turn the rest of the job over to someone else, you might tell them to "reverse the rest of these cards onto this other stack."

In terms of our Scheme procedure, we can reduce the problem of reversing `(1 2 3 4)` to the smaller problem of putting the elements of `(2 3 4)` in reverse order onto the front of `(1)`, which in turn reduces to putting the elements of `(3 4)` in reverse order onto the front of `(2 1)`, etc.:

```
(define reverse
  (lambda (lst)
    (define reverse-onto   ; return a list of the elements of lst1
      (lambda (lst1 lst2)  ;  in reverse order followed by the
                           ;  elements of lst2
```

```
        (if (null? lst1)
            lst2
            (reverse-onto (cdr lst1)
                          (cons (car lst1)
                                lst2)))))
    (reverse-onto lst '())))
```

The internally defined procedure `reverse-onto` does one `cons` for each element in the list, as it moves it from the front of *lst*1 to the front of *lst*2. Notice that the total number of `cons`es is equal to the total number of `cdr`s; if we use n to denote the size of the list *lst*, the number of `cons`es is n. Thus we have reduced a $\Theta(n^2)$ process to a $\Theta(n)$ one.

Now we have all the procedures that we need to determine whether or not a list is actually a palindrome:

```
(define palindrome?
  (lambda (lst)
    (equal? lst (reverse lst))))


(palindrome? '(m a d a m i m a d a m))
#t
```

7.5 Tree Recursion and Lists

In this section, we will look at two examples of using tree recursion with lists. The first example is a merge sort procedure roughly paralleling what you did by hand in Chapter 4. The basic approach to merge sorting a list is to separate the list into two smaller lists, merge sort each of them, and then merge the two sorted lists together. We can only separate a list into two shorter lists if it has at least two elements, but luckily all empty and one-element lists are already sorted. Thus our merge sort procedure would look something like the following:

```
(define merge-sort
  (lambda (lst)
    (cond ((null? lst)
           '())
          ((null? (cdr lst))
           lst)
          (else
           (merge (merge-sort (one-part lst))
                  (merge-sort (the-other-part lst)))))))
```

We still have to do most of the programming, namely, writing `merge` and figuring out some way to break the list into two parts, which for efficiency should be of equal size or at least as close to equal size as possible.

We start with the procedure `merge`. Here we have two lists of numbers, where each list is in order from the smallest to the largest. We want to produce a third list, that consists of all of the elements of both of the original lists and that is still in order. Notice that we have essentially two base cases, which occur when either one of the original lists is empty. In each case, the result that we want to return is the other (possibly nonempty) list. We additionally have three recursive cases, depending on how the first elements of the two lists compare to each other. We always want to cons the smaller number onto the result of merging the cdr of the list this number came from with the other list. What happens if both lists have the same number as their first element? The answer depends on why we're merging the two lists. There are some applications where we want to keep only one of the duplicated number, and some applications where we want to keep both duplicates. Here, we've arbitrarily decided to keep only one of the duplicated element. This will result in a `merge-sort` that eliminates duplicates as it sorts:

```
(define merge
  (lambda (lst1 lst2)
    (cond ((null? lst1) lst2)
          ((null? lst2) lst1)
          ((< (car lst1) (car lst2))
           (cons (car lst1) (merge (cdr lst1) lst2)))
          ((= (car lst1) (car lst2))
           (cons (car lst1) (merge (cdr lst1) (cdr lst2))))
          (else
           (cons (car lst2) (merge  lst1 (cdr lst2)))))))
```

What about breaking the list into two halves? One way of doing so would be to use the procedures `first-elements-of` and `list-tail` as we did in the perfect shuffle problem. The problem with this approach is that we would need to know how long the list is that we're trying to sort. True, we can use `length` to determine this. But instead we'll show off a different way of separating the list into parts, which is roughly the opposite of interleaving. In other words, if we think of our list as a deck of cards, we could separate it into halves by dealing the cards to two people. One person would get the first, third, fifth, ... cards, and the other would get the second, fourth, sixth, ... cards. We'll call the resulting two hands of cards the odd part and the even part.

To write the procedures `odd-part` and `even-part`, think about how you deal a deck of cards to two people, let us say Alice and Bob. You start out facing Alice and are going to give her the odd part and Bob the even part. In other words, you

are going to give the odd-numbered cards to the person you are facing. You start by dealing Alice the first card. Now you turn and face Bob, holding the rest of the cards in your hand. At this moment, the situation is exactly as it was at the beginning, except that you are facing Bob and have one fewer card. You are about to give Bob the first, third, etc., of the remaining cards, and Alice the even-numbered ones. So, all in all, Alice's hand of cards (the odd part of the deck) consists of the first card and then the even part of the remaining cards. Meanwhile, Bob's hand of cards (the even part of the deck) consists of the odd part of what's left of the deck after dealing out the first card. The `odd-part` and `even-part` procedures therefore provide an interesting example of *mutual recursion*, because we can most easily define them in terms of each other:

```
(define odd-part
  (lambda (lst)
    (if (null? lst)
        '()
        (cons (car lst) (even-part (cdr lst))))))

(define even-part
  (lambda (lst)
    (if (null? lst)
        '()
        (odd-part (cdr lst)))))
```

Now all we need to do to make `merge-sort` work is

```
(define one-part odd-part)
(define the-other-part even-part)
```

Our second example of using tree recursion comes from a family outing. One day, we took our sons (aged 3 and 4) to the local video arcade to play a game called Whacky Gator. Each child won several tickets that could be exchanged for "prizes" at the main counter. Each kind of prize has a price attached. Some prizes are worth ten tickets, some are worth nine tickets, and so on down to the plastic bugs, which are only worth one ticket. The older child had won ten tickets and wanted to know what prizes he could get. Obviously, he could get only one of the ten ticket prizes. Alternatively, he could get one nine-ticket prize and one one-ticket prize or one eight-ticket prize and two one-ticket prizes or one eight-ticket prize and one two-ticket prize or .... As the child's mother started enumerating the different combinations that he could get, we whipped out our pocket Scheme systems and discovered that there are 1778 possible combinations of prizes that the child had to choose from, given the number of different prizes there were. (See Table 7.1.)

TABLE 7.1 The low-value prizes at our local video arcade

Value in tickets	Number of distinct prizes
10	9
9	3
8	2
7	4
6	3
5	4
4	3
3	3
2	4
1	2

How did we do that? First, we decided to write a general procedure for figuring out how many prize combinations there were. To see how we developed that procedure, let's consider a much smaller problem. Suppose that there are only two kinds of one-ticket prizes, plastic spiders and plastic beetles. Then there are plastic worms that are worth three tickets and little magnifying glasses that are worth five tickets. If the child has five tickets, he can either get a magnifying glass or not. If he doesn't get a magnifying glass, he needs to get some combination of the rest of the prizes that adds up to 5. If he does get the magnifying glass, he's used up all his tickets and there's only one combination of additional prizes worth the remaining 0 tickets, (i.e., the empty combination).

This approach gives us a way of reducing our problem to smaller problems. Suppose the prizes are represented by a list of their values. In our small example above, we would use the list `(5 3 1 1)` to represent the magnifying glasses, the worms, the beetles, and the spiders. In this case, if a child has a certain amount of tickets, she can get a combination of prizes that includes the first item in the list or one that doesn't include the first item. If she chooses the first item, we need to count the number of combinations that she can get for the amount of tickets minus the value of the first item. If she doesn't choose the first item, we need to count how many combinations of items she can get for the amount of tickets using only the rest of the list. Thus, our recursive call would look like

```
(define count-combos
  (lambda (prize-list amount)
    .
    .
    .
    (+ (count-combos prize-list (- amount (car prize-list)))
       (count-combos (cdr prize-list) amount))))
```

What are the base cases? To figure these out, note that our problem gets smaller in one of two ways: either the prize list gets smaller or the amount of tickets decreases, because all of the prizes have positive prices. Thus the process should halt when the amount is 0, when it is less than 0, or when the prize list is null.

Exercise 7.15

What values should be returned in each of these cases? Using your answer, finish writing the procedure `count-combos`.

To check to see if there really were 1778 possible combinations worth 10 tickets, we need to enter a list of 37 numbers. Alternatively, we could write a procedure that will generate that list for us, given the data. What is the best way to give the data? One way would be to give it as a list of pairs, where the first number in each pair is the value and the second number is the number of distinct prizes worth that value. Another way would be to give the data by giving the value of the most expensive prize and then giving the list of numbers of different prizes, with the first number representing the number of different prizes for the most expensive prize, the second number representing the number of distinct prizes worth one ticket less than the most expensive prize, and so on.

Exercise 7.16

Which representation is best? Why? Can you think of any other, better way of representing the data? Think about what the corresponding procedures would look like as well as entering the data.

Exercise 7.17

Write the procedure that would generate the list needed for `count-combos` given the data in Table 7.1. Check to see that there really are 1778 possible combinations of prizes that are worth 10 tickets.

Exercise 7.18

One of our children has learned that he doesn't need to spend all of his tickets because he can save them up for his next trip. Thus, instead of finding the number of combinations that he can get with one particular amount he would like to know the number of combinations that he can get for any amount that is less than or equal to the number of tickets he has. Write a procedure that is given a prize list and a

maximum amount and returns the number of combinations of prizes that you can buy using no more than than the maximum amount of tickets.

Exercise 7.19

When our children started bringing home dozens of cheap plastic spiders, we asked them to restrict themselves to getting only one of each kind of prize. Write a procedure that is given the prize list and amount and computes the number of prize combinations that you can buy using exactly that amount and assuming that you can't get more than one of any particular prize.

Exercise 7.20

Write another procedure that will determine the number of combinations you can buy using no more than a maximum amount of tickets while still insisting that you can have at most one of each kind of prize.

Exercise 7.21

A similar problem to this is to imagine that you have an unlimited amount of quarters, dimes, nickels, and pennies and that you need to come up with a combination of these coins to make a certain amount. How many different ways can you do this? Write a procedure that will count the number of ways to make change for a given amount using only quarters, dimes, nickels, and pennies.

7.6 An Application: A Movie Query System

Have you ever gone to a video store to rent a movie, only to be confronted with so many movies that you couldn't find one you wanted to see? Perhaps you were interested in seeing a movie by a given director, but you didn't know which ones they were, and the movies weren't organized by director anyway. Or perhaps you wanted to know which movies were directed by the person, whose name you forgot, who directed some favorite movie? What if you didn't know the name of the movie, but you knew that it was made in the mid to late 1980s and Dennis Quaid starred in it?

Being able to answer such questions would go a long ways toward finding a movie. One possibility would be to ask the store personnel, but perhaps they are busy or unfriendly or only like slasher movies. Another possibility would be to take some movie expert, say Roger Ebert, along with you to the store, but that is probably unrealistic. Wouldn't it be nice if the video store provided a computer that had a

program you could ask such questions? Perhaps it could even tell you which movies were currently available at the store.

Let's imagine how this program might be structured. First, it must have access to the database of movies owned by the store. Second, it should have the ability to search through the database in various ways. Finally, the user should be able to use these search procedures flexibly and intuitively. Perhaps the user could carry on a dialog with the computer that looks much like an ordinary English conversation. Such a feature is called a *natural language query system*.

We will write a small version of such a program in this section. The database will simply be a list of movie records. Next we will write procedures to search through this list in various ways. Finally, we provide a query system for the program by using pattern-matching on the user's queries.

So let's first create the database. For the individual movies, we need to define a compound ADT with four components: the title of the movie, the name of its director, the year the movie was made, and a list of the actors in it. For simplicity's sake, we assume that the year is a number and that names (of movies and of people) are lists of symbols. We could construct movie records in a manner similar to how we constructed three-pile game states in Chapter 6; alternatively, we could simply put everything into a list. Because this alternative is easily done using the built-in procedure `list`, we'll represent movie records as lists:

```
(define make-movie
  (lambda (title director year-made actors)
    (list title director year-made actors)))

(define movie-title car)
(define movie-director cadr)
(define movie-year-made caddr)
(define movie-actors cadddr)
```

(These definitions take advantage of the fact that `cadr`, `caddr`, and `cadddr` are built into Scheme as procedures for selecting the second, third, and fourth element of a list. The names stand for "the car of the cdr," etc.) We can then define `our-movie-database` to be a list of such records as follows:

```
(define our-movie-database
  (list (make-movie '(amarcord)
                    '(federico fellini)
                    1974
                    '((magali noel) (bruno zanin)
                                    (pupella maggio)
                                    (armando drancia)))
```

```
(make-movie '(the big easy)
            '(jim mcbride)
            1987
            '((dennis quaid) (ellen barkin)
                             (ned beatty)
                             (lisa jane persky)
                             (john goodman)
                             (charles ludlam)))
(make-movie '(the godfather)
            '(francis ford coppola)
            1972
            '((marlon brando) (al pacino)
                              (james caan)
                              (robert duvall)
                              (diane keaton)))
(make-movie '(boyz n the hood)
            '(john singleton)
            1991
            '((cuba gooding jr.) (ice cube)
                                 (larry fishburne)
                                 (tyra ferrell)
                                 (morris chestnut)))))
```

This example is of course a very small database. In the software on the web site for this book, we include a more extensive database, also called `our-movie-database`, that you can use for experimentation.

What types of database search procedures will we want to implement? We might want to find all the movies by a given director or all of the movies that were made in a given year or all the movies that have a particular actor in them.

Exercise 7.22

We can use the procedure `filter` defined in Section 7.3 to do any one of these searches. For example, to find all the movies that were made in 1974, we would evaluate

```
(filter (lambda (movie) (= (movie-year-made movie) 1974))
        our-movie-database)
```

a. Write a procedure called `movies-made-in-year` that takes two parameters, the list of movies and a year, and finds all the movies that were made in that year.

b. Use the procedure `filter` to find all the movies that were directed by John Singleton.

c. Write a procedure called `movies-directed-by` that takes two parameters, the list of movies and the name of a director, and finds all of the movies that were directed by that director.

d. Write a procedure called `movies-with-actor` that takes two parameters, a list of the movies and the name of an actor, and finds all the movies that have that actor in them. You could use the Scheme predicate `member`, which tests to see whether its first argument is equal to any element of its second argument (which must be a list).

Exercise 7.23

The biggest problem with the previous procedures is that they return a list of the actual movie records, when we often would prefer just a list of the titles of the movies. Write a procedure called `titles-of-movies-satisfying` that takes two arguments, a list of movies and a predicate, and returns a list of the titles of the movies satisfying the predicate argument. For example, evaluating the expression

```
(titles-of-movies-satisfying our-movie-database
                             (lambda (movie)
                               (= (movie-year-made movie)
                                  1974)))
```

would give the list of titles of movies made in 1974. *Hint:* Use the procedure `map` described in Section 7.3.

Exercise 7.24

Sometimes we want some attribute other than the title when we're searching for the movies that satisfy a given property. Generalize `titles-of-movies-satisfying` to a procedure `movies-satisfying` that takes three arguments: a list of movie records, a predicate, and a selector. Evaluating the expression

```
(movies-satisfying our-movie-database
                   (lambda (movie)
                     (= (movie-year-made movie) 1974))
                   movie-title)
```

should have the same result as the previous exercise.

Now that we have procedures to search through our database, we need to consider the people who use the system. Typical video store patrons are not going to be willing to deal with the kind of Scheme expressions that we've been evaluating to find answers to their questions, and so we need to build a query system (i.e., an interface that allows them to ask questions more easily).

The goal of the query system is to handle the user's questions, causing the appropriate database searching procedures to be called and the results reported to the user. We will construct a query system that at least superficially simulates an English dialog between the program and the user; such an interface is often called a *natural language interface*. (*Natural* in this context refers to a naturally occurring human language like English as opposed to a computer language such as Scheme; *interface* refers to the fact that it is the point of contact between the user and the internals of the program.) Thus, the task of this query system will be to read the user's questions, interpret them as requesting specific actions, and perform and report the results of those actions.

Let's be as concrete as possible. Suppose that we have a procedure called `query-loop` that repeatedly reads and responds to the user's questions. We would like to have an interaction something like the following; the first line is a Scheme expression, which is evaluated to start the loop, and the remaining lines are the interaction with the loop:

```
(query-loop)

(who was the director of amarcord)
(federico fellini)

(who were the actors in the big easy)
((dennis quaid)
 (ellen barkin)
 (ned beatty)
 (lisa jane persky)
 (john goodman)
 (charles ludlam))

(what movies were made in 1991)
((boyz n the hood) (dead again))

(what movies were made between 1985 and 1990)
((the big easy))

(what movies were made in 1921)
(i do not know)

(why is the sky blue)
(i do not understand)
```

```
(so long)
(see you later)
```

Some observations are

- The user's questions are English sentences without punctuation and enclosed in parenthesis; our program views them as lists of symbols.
- The program displays the list of answers to the user's query. If no answer is found, it responds with `(i do not know)`; if it can't interpret the question, it responds with `(i do not understand)`.
- The movie database we used for this sample interaction is slightly larger than the one given above.

How can we program such an interaction? Remember the query loop repeatedly reads a question, interprets that question as a request for some specific action, performs that action, and reports the result. Not all questions can be interpreted. However, those that can be interpreted typically match one of a small set of patterns. For example, the questions `(who was the director of amarcord)` and `(who was the director of the big easy)` both have the form `(who was the director of ...)`. The key idea to programming the query loop is to use an abstract data type called *pattern/action pairs*. Roughly speaking, a pattern specifies one possible form that questions can have, whereas the corresponding action is the procedure for answering questions of that form. The procedure `query-loop` will use a list of these pattern/action pairs to respond to the user and will terminate if the user's question appears in a list of the ways of quitting the program. (This test for quitting is done in the `exit?` procedure using the predefined procedure `member`, which tests whether its first argument is equal to any element of its second argument. We introduced `member` in Exercise 7.22d.)

```
(define query-loop
  (lambda ()
    (newline)
    (newline)
    (let ((query (read)))
      (cond ((exit? query) (display '(see you later)))
            ;; movie-p/a-list is the list of the
            ;;  pattern/action pairs
            (else (answer-by-pattern query movie-p/a-list)
                  (query-loop))))))

(define exit?
  (lambda (query)
    (member query '((bye)
```

```
(quit)
(exit)
(so long)
(farewell)))))
```

All of the real work in `query-loop` gets done by the procedure `answer-by-pattern`. How does this procedure work? The idea is that each of the questions that the program can answer must match one of the patterns in the pattern/action list. The action corresponding to the matching pattern is an actual Scheme procedure that does what needs to be done in order to answer the given question. In order to apply this action, we figure out what particular information must be substituted into the blanks in the general pattern to make it match the specific question. These substitutions are the arguments to the action procedure. Our `answer-by-pattern` procedure will use two (as yet unwritten) procedures: `matches?`, which determines if a question matches a given pattern, and `substitutions-in-to-match`, which determines the necessary substitutions.

What should `answer-by-pattern` display? By looking at `query-loop`, we know that the output should be a list that answers the user's question. Thus, if the question matched none of the patterns, the answer should be the list `(i do not understand)`. On the other hand, if the user's question did match one of the patterns, `answer-by-pattern` applies an action procedure that causes our database to be searched. The value of this expression will be a list of answers. If that list of answers is nonempty, we can simply display it; if it is empty, no answers were found. In this case we should display the list `(i do not know)`, indicating that we could not find the movie or movies the user was looking for.

```
(define answer-by-pattern
  (lambda (query p/a-list)
    (cond ((null? p/a-list)
           (display '(i do not understand)))
          ((matches? (pattern (car p/a-list)) query)
           (let ((subs (substitutions-in-to-match
                        (pattern (car p/a-list))
                        query)))
             (let ((result (apply (action (car p/a-list))
                                  subs)))
               (if (null? result)
                   (display '(i do not know))
                   (display result)))))
          (else
           (answer-by-pattern query
                              (cdr p/a-list))))))
```

This procedure cdrs down the list of pattern/action pairs until it finds a pattern matching the query. It then uses `substitutions-in-to-match` to get the substitutions from the query and applies the appropriate action to those substitutions. We are using the built-in Scheme procedure `apply` to apply the action procedure to its arguments. To illustrate how `apply` works, consider the following interaction:

```
(apply + '(1 4))
5

(apply * '(2 3))
6

(apply (lambda (x) (* x x)) '(3))
9

(apply movies-satisfying
       (list our-movie-database
             (lambda (movie) (= (movie-year-made movie) 1974))
             movie-title))
((amarcord))
```

Notice that the first argument to `apply` is a procedure and the second argument is the list of arguments to which the procedure is applied. Therefore, when we make the following call in `answer-by-pattern`,

```
(apply (action (car p/a-list))
       subs)
```

we are applying the action procedure in the first pattern/action pair to the list consisting of the substitutions we got from the pattern match.

To get our query system working, we need to do three things. We need to construct an ADT for pattern/action pairs, we need to start building the list of these pairs, and we need to write the procedures `matches?` and `substitutions-in-to-match`. Doing these things depends on understanding what patterns are. Consider the following possible questions:

```
(who is the director of amarcord)
(who is the director of the big easy)
(who is the director of boyz n the hood)
```

The common pattern of these three questions is clear. If we use ellipsis points (. . .) to represent the title, we can write this pattern as

```
(who is the director of ...)
```

The ellipsis points are sometimes called a *wild card*, because they can stand in for any title.

What action should correspond to this pattern? As we said before, an action is simply a procedure. After determining that `(who is the director of the big easy)` matches this pattern, we will need to apply our action to the title `(the big easy)`, because we would substitute the title for the `...` wild card to make the question match the pattern. Thus, this particular action should be a procedure that takes a title, finds the movie in our movie database that has this title, and returns the director of that movie:

```
(lambda (title)
  (movies-satisfying our-movie-database
                     (lambda (movie)
                       (equal? (movie-title movie) title))
                     movie-director))
```

Constructing the pattern/action ADT and building a list of pattern/action pairs is straightforward. We define the pattern/action ADT much as we defined game states in Chapter 6:

```
(define make-pattern/action
  (lambda (pattern action)
    (cons pattern action)))

(define pattern car)
(define action cdr)
```

We start building our list of pattern/action pairs by constructing a list with just one pair:

```
(define movie-p/a-list
  (list (make-pattern/action
         '(who is the director of ...)
         (lambda (title)
           (movies-satisfying
            our-movie-database
            (lambda (movie) (equal? (movie-title movie) title))
            movie-director)))))
```

We will be extending this list throughout the rest of the section.

Exercise 7.25

Add a pattern of the form `(who acted in ...)` to your program by adding the appropriate pattern/action pair to `movie-p/a-list` and reevaluating this definition. What other patterns can you add?

To make `answer-by-pattern` work, we still need to write the procedures `matches?` and `substitutions-in-to-match`. To begin with, suppose that all of our patterns have the same form as `(who is the director of ...)` or `(who acted in ...)`. Therefore, a pattern would be a list of symbols; the last symbol (and only the last) could be the `...` wild card. For a pattern to match a question, all non-wild card symbols in the pattern must be equal to the corresponding symbols in the question. Beyond these equal symbols, if the pattern ends with the `...` wild card, there must be one or more additional symbols at the end of the question. Thus, to write `matches?`, we need to cdr down both the pattern and the question, checking that the cars are equal. We should stop if either of the two lists is empty, if the . . . wild card is the car of the pattern list, or if the two cars are not equal:

```
(define matches?
  (lambda (pattern question)
    (cond ((null? pattern)  (null? question))
          ((null? question) #f)
          ((equal? (car pattern) '...) #t)
          ((equal? (car pattern) (car question))
           (matches? (cdr pattern)
                     (cdr question)))
          (else #f))))
```

The procedure `substitutions-in-to-match` will be very similar to `matches?` in that it will get two arguments, a pattern and a question. However, `substitutions-in-to-match` will be called only when these two lists match. It needs to return a *list* of the substitutions for the wild cards in the pattern that will make it match the question. Currently, we are assuming that there will only be one substitution, which is the list of symbols that are matched by the `...` wild card. However, we will soon extend the definitions of both `matches?` and `substitutions-in-to-match` so that patterns can contain more than one wild card. Thus, your `substitutions-in-to-match` should return a one-element list, where the one element is the list of symbols that matches the . . . wild card, as follows:

```
(substitutions-in-to-match '(foo ...)
                           '(foo bar baz))
((bar baz))
```

Exercise 7.26

Write the procedure `substitutions-in-to-match`. Be sure to return a list containing the list of symbols that are matched by the `...` symbol. *Note:* You needn't use the whole query system to test whether `substitutions-in-to-match` works. Instead, you could check whether you have interactions like the preceding one. This note applies as well to later exercises that ask you to extend `matches?` and `substitutions-in-to-match`.

Exercise 7.27

Test the whole query system by evaluating the expression `(query-loop)`.

At this point, a typical question/answer might look like

```
(who is the director of amarcord)
((federico fellini))
```

Note that the answer is a list containing the director's name, which is itself a list. This is because `movies-satisfying` is finding the director of each of the movies called `amarcord`, even though there's only one. Asking for the actors in a particular movie is even uglier; you get a list containing the list of the actors' names, which are themselves lists.

We can get better looking output by writing a procedure called `the-only-element-in` and changing the action for finding the director of a movie to

```
(lambda (title)
  (the-only-element-in
   (movies-satisfying
    our-movie-database
    (lambda (movie) (equal? (movie-title movie) title))
    movie-director)))
```

The procedure `the-only-element-in` has a single parameter, which should be a list. If this list has only one element in it, `the-only-element-in` returns that element.

Exercise 7.28

What should it return if there are no elements in the list? What if there are two or more? Write this procedure, and use it to modify the action for finding the actors of a particular movie.

Now that our program can recognize very simple patterns, we can start adding more complicated ones. The next pattern we add to our list is typified by the following sentences:

```
(what movies were made in 1955)
(what movie was made in 1964)
```

What is the common pattern for these two queries? By using an extended pattern language, we could write it as follows:

```
(what (movie movies) (was were) made in _)
```

We have extended our pattern language in two ways:

1. The symbol `_` stands for a single-word wild card (as opposed to the multiword wild card `...`). Note that we can have as many `_`'s as we want; each one matches a single word. Unlike with the `...` wild card, the `_` wild cards need not appear at the end of the pattern.
2. List wild cards such as `(movie movies)` and `(was were)` in the pattern are more restricted versions of the `_` wild card. A wild card of either type must be matched by a single word. However, the `_` wild card can be matched by any word at all, whereas a list wild card can be matched only by one of its elements. Thus, the wild card `(movie movies)` can only match `movie` or `movies`.

Here's how we can extend `matches?` to account for the second extension to our pattern language. We use the Scheme predicate `list?`, which returns true if the argument is a list, and also once again use `member` to test for list membership:

```
(define matches?
  (lambda (pattern question)
    (cond ((null? pattern)  (null? question))
          ((null? question) #f)
          ((list? (car pattern))
           (if (member (car question) (car pattern))
               (matches? (cdr pattern)
                         (cdr question))
               #f))
          ((equal? (car pattern) '...) #t)
          ((equal? (car pattern) (car question))
           (matches? (cdr pattern)
                     (cdr question)))
          (else #f))))
```

Exercise 7.29

Extend `matches?` so that it also checks for the `_` wild card. Remember that this is a wild card for a single word. Also, remember that there can be more than one `_` in a single pattern and that they need not be at the end of the pattern.

Exercise 7.30

Extend `substitutions-in-to-match` to account for both of these extensions. It should return the list of substitutions, one for each wild card.

Using these extended versions, we can redefine `movie-p/a-list` as follows:

```
(define movie-p/a-list
  (list (make-pattern/action
         '(who is the director of ...)
         (lambda (title)
           (the-only-element-in
            (movies-satisfying
             our-movie-database
             (lambda (movie) (equal? (movie-title movie) title))
             movie-director))))
        (make-pattern/action
         '(what (movie movies) (was were) made in _)
         (lambda (noun verb year)
           (movies-satisfying
            our-movie-database
            (lambda (movie) (= (movie-year-made movie) year))
            movie-title)))))
```

Note that the action for this new pattern totally ignores the first two substitutions. The substitutions for the first two wild cards in `(what (movie movies) (was were) made in _)` are often called *noise words* because they are not used in the corresponding action. Not all list wild cards are used for noise words, however. Suppose that instead of the pattern `(what (movie movies) (was were) made in _)`, we used `(what (movie movies) (was were) made (in before after since) _)`. The corresponding action would still ignore those first two noise words, but it would need to know the substitution for the third wild card in order to know which comparison operator to use.

Exercise 7.31

What would the action corresponding to `(what (movie movies) (was were) made (in before after since) _)` be? Remember, because the pattern contains four wild cards, the action procedure should get four arguments. It ignores the first two of these and uses the third and fourth to construct a predicate. Note that the third argument is a symbol; you will need to use that symbol to decide which comparison to do.

Exercise 7.32

Add a pattern/action for the pattern

```
(what (movie movies) (was were) made between _ and _)
```

Exercise 7.33

What if the user asks for the director of "Godfather," which is listed in our database as "The Godfather"? As it stands, the program will respond that it doesn't know, even though it really does have the movie in its database. The point is that the symbol `the` rarely contributes significant information as to the movie's title. Similarly the symbols `a` and `an` add little significant information.

Write a predicate that compares two titles but ignores any articles in either title. Where would you use this predicate in the interface?

Exercise 7.34

It would be nice if we could add patterns of the form

```
(when was the godfather made)
(when was amarcord made)
```

The pattern could be `(when was ... made)`, but unfortunately, `matches?` and `substitutions-in-to-match` require that the `...` wild card occur at the end of the pattern, because as written, the `...` absorbs the remainder of the sentence.

Extend `matches?` and `substitutions-in-to-match` to allow for patterns having only one occurrence of the `...` wild card but where that occurrence need not be at the end of the pattern. *Hint:* If the pattern starts with `...` and is of length n, and the sentence is of length m, and there can be no additional `...` wild cards in the rest of the pattern, then how many words must the `...` match up with?

Using these extended versions, add a pattern/action for the pattern

```
(when was ... made)
```

Exercise 7.35

If you allow more than one `...` in the same pattern, there can be more than one set of substitutions that makes the pattern match a sentence. For example, if the pattern is `(do you have ... in ...)`, and the sentence is `(do you have boyz in the hood in the store)`, the pattern will match not only if you substitute "boyz in the hood" for the first wild card and "the store" for the second but also if you substitute "boyz" for the first wild card and "the hood in the store" for the second wild card.

Redesign `substitutions-in-to-match` so it returns a list of all the possible sets of substitutions that make the pattern match the sentence, rather than just one set. Allow multiple instances of `...` in the same pattern. You'll need to make other changes in the way the program uses `substitutions-in-to-match`. You can also redesign the program to eliminate `matches?`, because a pattern matches if there are one or more sets of substitutions that make it match.

Exercise 7.36

Add pattern/action pairs that allow the user to ask other questions of your own choosing. Try to make the patterns as general as possible, for example, by allowing singular and plural as well as past and present tenses. Also allow for the various ways the user might pose the query.

Exercise 7.37

Earlier, we said that the query system reads the user's questions as lists of symbols. We were stretching the truth, as illustrated by the query:

```
(what movie was made in 1951)
```

The last element of that question is not a symbol; it's the number `1951`. This raises an interesting point. Because it is a number, it can be compared to other numbers using the `=` operator. However, consider what would happen if we had the question

```
(what movie was made in Barcelona)
```

In this case, the action procedure attempts to compare the symbol `Barcelona` with the year each movie was made using the `=` operator, and because `Barcelona` isn't a

number, an error is signaled. The whole problem here is that the `_` is too general; it will match anything at all and not just a number. Change the pattern language (and the query system) to allow wild cards that are predicates such as `number?`.

Exercise 7.38

The pattern `(what (movie movies) (was were) made in _)` would match questions such as `(what movie were made in 1967)`. To enforce grammatical correctness, we would need to change our pattern language so that it would allow wild cards that provide a choice among alternative lists of words rather than simply among single words. One example of this would be the pattern

```
(what ((movie was) (movies were)) made in _)
```

Make the appropriate changes in the query system to allow for patterns of this type.

Is There More to Intelligence Than the Appearance of Intelligence?

The natural language interface presented in this section is quite clearly unintelligent. It has no real understanding of the sentences it accepts as input or produces as output—it is just mechanically matching patterns and spitting out canned responses. Yet if you ignore little things like punctuation and capitalization (which are easy, but uninteresting, to fix), the dialog between the system and the user could easily be mistaken for one between two humans.

Of course, the illusion only holds up as long as the input sentences are well suited to the patterns and actions that are available. However, as we add more patterns, the range of coverage gets larger. What if we also added progressively more and more sophisticated kinds of pattern-matching, and stored data about more and more topics? Presumably it would get harder and harder to distinguish the system from an intelligent being—yet it would still be every bit as much a mechanical "symbol pusher" as the current system. What if this progression were taken to such extremes that no one could tell the difference between the system's behavior and that of a human? Even if no one ever achieves this feat of programming, the hypothetical question has already provoked much philosophical debate.

Some people, including most mainstream computer scientists, apply the operational stance to intelligence: If it acts intelligent, it *is* intelligent. The idea that an intelligent entity could be (at least hypothetically) the endpoint in a

(Continued)

Is There More to Intelligence Than the Appearance of Intelligence?

progression of progressively fancier mechanistic symbol pushers isn't bothersome to these people. There is no inherent contradiction between "mechanistic" and "intelligent" because one refers to how the behavior is produced, and the other refers to the nature of the behavior. Perhaps no mechanistic symbol-pushing system could ever produce behavior that matched that of humans—after all, not all kinds of mechanisms can be used to produce all kinds of results. But, these people argue, if such a system ever *did* produce behavior like that of humans, we would have no choice but to accept it as intelligent—after all, what else could "intelligent" mean other than "behaving intelligently"? This operational stance regarding intelligence was subscribed to by Alan Turing, among others, as described in the biographical sketch of him in Chapter 5, and is frequently referred to as the *Turing test* definition of intelligence.

On the other hand, some people (of whom the most well known is the philosopher John Searle) say that if every program in our progression is a mechanical symbol pusher, every one of them is operating on tokens that to us may bring real things like people and movies to mind, but to the program are completely content-free groupings of letters. The only sense in which the sentences can be said to be "about movies" is that we humans can successfully associate them with movies. To the computer, there is nothing but the words themselves. This is every bit as true for the hypothetical endpoint of our evolution, which is indistinguishable in its behavior from a human as it is for the crude version presented in this section. Even flawless conversation "about movies" is only truly "about movies" for us humans, this group of philosophers would argue—to the computer, even the hypothetical flawless conversation is just a string of words. Because Searle made this point using a story about being locked in a room following precise rules for processing things that were to him just squiggles, but to certain outsiders made sense as Chinese text, this argument against the Turing test definition of intelligence is frequently called the *Chinese room argument*.

A related point, which has also been persuasively argued by Searle, is that we humans can do things in our minds, such as liking a movie, whether or not we choose to utter the string of words that conventionally expresses this state, whereas there is no particular reason to assume that some mechanical system that utters "I like *Boyz N the Hood*" really does like the movie or even is the kind of thing that is capable of liking. When other people state their likes, we may or may not trust them to have honestly done so, but at least we accept that they can have likes because we have likes and the other people are similar to us. For a dissimilar thing, such as a computer program, we don't have any reason to believe there are truly any likes inside the shell of statements about likes. We have no reason to

(Continued)

Is there more to intelligence? (Continued)

suppose the program is the kind that could lie about its likes, any more than that it could tell the truth, because there is nothing we can presume the existence of that would form the standard against which a purported like would be judged. In a fellow human, on the other hand, we presume that there are "real likes" and other *mental states* inside because we feel them in ourselves.

In short, Turing prefers to define *intelligence* in terms of behavior we can observe, whereas Searle would prefer to define it in terms of internal mental states. Searle presumes that a mute, paralyzed human being has such states and reserves judgment on flawlessly communicating computer programs.

Review Problems

Exercise 7.39

Prove using induction on n that the following procedure produces a list of length n.

```
(define sevens
  (lambda (n)
    (if (= n 0)
        '()
        (cons 7
              (sevens (- n 1))))))
```

Exercise 7.40

Suppose that $f_1, f_2, \ldots, f_n$ are all functions from real numbers to real numbers. The functional sum of $f_1, f_2, \ldots, f_n$ is the function that, when given a number x, returns the value $f_1(x) + f_2(x) + \cdots + f_n(x)$. Write a procedure `function-sum` that takes a list of functions and returns the functional sum of those functions. For example

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


(define cube
  (lambda (x) (* x (* x x))))


((function-sum (list square cube)) 2)
12
```

Exercise 7.41

Write an iterative version of the following procedure:

```
(define square-sum
  (lambda (lst)
    (if (null? lst)
        0
        (+ (square (car lst))
           (square-sum (cdr lst))))))
```

Exercise 7.42

Write a procedure called `apply-all` that, when given a list of functions and a number, will produce the list of the values of the functions when applied to the number. For example,

```
(apply-all (list sqrt square cube) 4)
(2 16 64)
```

Exercise 7.43

Prove by induction on n that the following procedure produces a list of $2n$ seventeens:

```
(define seventeens
  (lambda (n)
    (if (= n 0)
        '()
        (cons 17 (cons 17 (seventeens (- n 1)))))))
```

Exercise 7.44

Consider the following two procedures. The procedure `last` selects the last element from a list, which must be nonempty. It uses `length` to find the length of the list.

```
(define last
  (lambda (lst)
    (if (= (length lst) 1)
        (car lst)
        (last (cdr lst)))))
```

```
(define length
  (lambda (lst)
    (if (null? lst)
        0
        (+ 1 (length (cdr lst))))))
```

a. How many `cdr`s does `(length` *lst*`)` do when *lst* has *n* elements?

b. How many calls to `length` does `(last` *lst*`)` make when *lst* has *n* elements?

c. Express in Θ notation the total number of `cdr`s done by `(last` *lst*`)`, including `cdr`s done by `length`, again assuming that *lst* has *n* elements.

d. Give an exact formula for the total number of `cdr`s done by `(last` *lst*`)`, including `cdr`s done by `length`, again assuming that *lst* has *n* elements.

Exercise 7.45

Lists are collections of data accessible by *position*. That is, we can ask for the first element in a list, the second, . . . , the last. Sometimes, however, we'd prefer to have a collection of data accessible by *size*. In other words, we'd like to be able to ask for the largest element, the second largest, . . . , the smallest.

In this problem, we'll simplify this goal by restricting ourselves to collections containing exactly two real numbers. Thus the two selectors will select the `smaller` and `larger` of the two numbers. Here are some examples of this data abstraction in use; the constructor is called `make-couple`. Note that the order in which the argument values are given to the constructor is irrelevant, because selection is based on their relative size.

```
(define x (make-couple 2 7))
(define y (make-couple 5 3))
(define z (make-couple 4 4))

(smaller x)              (larger x)
2                        7

(smaller y)              (larger y)
3                        5

(smaller z)              (larger z)
4                        4
```

Write two versions of `make-couple`, `smaller`, and `larger`. One version should have `make-couple` compare the two numbers, and the other version should leave that to `smaller` and `larger`.

Exercise 7.46

Write a higher-order procedure `make-list-scaler` that takes a single number *scale* and returns a procedure that, when applied to a list *lst* of numbers, will return the list obtained by multiplying each element of *lst* by *scale*. Thus, you might have the following interaction:

```
(define scale-by-5 (make-list-scaler 5))

(scale-by-5 '(1 2 3 4))
(5 10 15 20)
```

Exercise 7.47

Write a procedure `map-2` that takes a procedure and two lists as arguments and returns the list obtained by mapping the procedure over the two lists, drawing the two arguments from the two lists. For example, it would yield the following results:

```
(map-2 + '(1 2 3) '(2 0 -5))
(3 2 -2)

(map-2 * '(1 2 3) '(2 0 -5))
(2 0 -15)
```

Write this procedure `map-2`. You may assume that the lists have the same length.

Exercise 7.48

Given the following procedure:

```
(define sub1-each
  (lambda (nums)
    (define help
      (lambda (nums results)
        (if (null? nums)
            (reverse results)
            (help (cdr nums)
                  (cons (- (car nums) 1) results)))))
    (help nums '())))
```

Evaluate the expression `(sub1-each '(5 4 3))` using the substitution model of evaluation. Assume `reverse` operates in a single "black-box" step, but otherwise

show each step in the evolution of the process. What kind of process is generated by this procedure?

Exercise 7.49

Given a predicate that tests a single item, such as `positive?`, we can construct an "all are" version of it for testing a list; an example is a predicate that tests whether all elements of a list are positive. Define a procedure `all-are` that does this; that is, it should be possible to use it in ways like the following:

```
((all-are positive?) '(1 2 3 4))
#t

((all-are even?) '(2 4 5 6 8))
#f
```

Exercise 7.50

Consider the following procedure (together with two sample calls):

```
(define repeat
  (lambda (num times)
    (if (= times 0)
        '()
        (cons num (repeat num (- times 1))))))

(repeat 3 2)
(3 3)

(repeat 17 5)
(17 17 17 17 17)
```

a. Explain why `repeat` generates a recursive process.

b. Write an iterative version of `repeat`.

Exercise 7.51

If a list contains multiple copies of the same element in succession, the list can be stored more compactly using *run length encoding*, in which the repeated element is given just once, preceded by the number of times it is repeated. The `expand` procedure given here is designed to decompress a run-length-encoded list; for example, it

could be used as follows to expand to full size some 1950s lyrics we got from Abelson and Sussman's text:

```
(expand '(get a job sha 8 na get a job sha 8 na wah 8 yip sha
             boom))
(get a job sha na na na na na na na get a job sha na na na na
na na na na wah yip yip yip yip yip yip yip yip yip sha boom)
```

a. Given the following definition of the `expand` procedure, show the key steps in evaluating `(expand '(3 ho merry-xmas))` using the substitution model.

b. Does this procedure generate an iterative or recursive process? Justify your answer.

```
(define expand
  (lambda (lst)
    (cond ((null? lst) lst)
          ((number? (car lst))
           (cons (cadr lst)
                 (expand (if (= (car lst) 1)
                             (cddr lst)
                             (cons (- (car lst) 1)
                                   (cdr lst))))))
          (else
           (cons (car lst)
                 (expand (cdr lst)))))))
```

Exercise 7.52

Suppose you have a two-argument procedure, such as `+` or `*`, and you want to apply it elementwise to two lists. For example, the procedures `list+` and `list*` would apply `+` and `*`, respectively, to the corresponding elements of two lists as follows:

```
(list+ '(1 2 3) '(2 4 6))
(3 6 9)
(list* '(1 2 3) '(2 4 6))
(2 8 18)
```

Because the two procedures `list+` and `list*` are so similar in form, it makes sense to write the higher-order procedure "factory" `make-list-combiner` that generates the two procedures `list+` and `list*` as follows:

```
(define list+
  (make-list-combiner +))

(define list*
  (make-list-combiner *))
```

Write the procedure `make-list-combiner`. You may assume that the two list arguments have the same length.

Chapter Inventory

Vocabulary

empty list
head
tail
cons up
cdr down
mutual recursion
database
natural language query system
natural language interface
wild card
noise word
Chinese room argument
mental state
run length encoding

Slogans

The two-part list viewpoint

Abstract Data Types

lists
movies
pattern/action pairs
couples

New Predefined Scheme Names

`null?`
`list`
`length`
`list-ref`
`list-tail`
`map`
`reverse`
`cadr`
`caddr`
`cadddr`
`member`
`apply`
`list?`
`number?`

Scheme Names Defined in This Chapter

`integers-from-to`
`length`
`sum`
`position`
`list-<`
`lists-compare?`

```
filter
first-elements-of
list-tail
interleave
shuffle
multiple-shuffle
in-order?
shuffle-number
sevens
list-of-lists
add-to-end
reverse
palindrome?
merge-sort
merge
odd-part
even-part
count-combos
make-movie
movie-title
movie-director
movie-year-made
movie-actors
our-movie-database
movies-made-in-year
movies-directed-by
movies-with-actor
titles-of-movies-satisfying
movies-satisfying
query-loop
exit?
answer-by-pattern
matches?
substitutions-in-to-match
make-pattern/action
pattern
action
movie-p/a-list
the-only-element-in
sevens
function-sum
square-sum
apply-all
last
make-couple
smaller
larger
make-list-scaler
map-2
sub1-each
all-are
repeat
expand
make-list-combiner
```

Sidebars

Is there more to intelligence than the appearance of intelligence?

Notes

For more information about perfect shuffles, including their applications in magic and computing, see Morris [38].

Turing's operational definition of intelligence is given in [52], and Searle's Chinese-room argument, in [47]. Searle has more recently (and more persuasively) made the case for mental states in his book *The Rediscovery of the Mind* [48].