4 Data Driven Recursion

4.1 Overview

In this chapter, we continue our study of recursion over the top-level items in lists. Then we make the extension to recursion over the items in the nested sublists as well, giving us tree recursion. In certain of our computations, a return table is built while operations that have yet to be performed wait for recursive procedure calls to return values. We discuss another way of doing the computations, called iteration, in which there are no operations waiting for procedure calls to return values, and hence no return table need be constructed. The factorial procedure and Fibonacci sequences are introduced. To compare the efficiency of various methods for computing them, we investigate the growth of execution time as the argument grows, demonstrating linear and exponential growth rates.

4.2 Flat Recursion

We begin with three more examples of recursive procedures, with the recursion being done over the top-level items in lists. In our examples of recursion involving lists, we made the recursive step by applying the procedure to the `cdr` of the list. The `car` of the list was then treated as a unit, which is why the recursion was over the top-level items in the list. We refer to a recursion over the top-level items of a list as a *flat recursion*, and we say that the procedure so defined is *flatly recursive* or simply a *flat procedure*.

The first procedure we define is the two-argument version of the Scheme procedure `append`, which has as parameters two lists, `ls1` and `ls2` and builds

a list that consists of the top-level items in `ls1` followed by the top-level items in `ls2`. We say that we are appending `ls2` to (the end of) `ls1`. For example,

```
(append '(a b c) '(c d)) ⟹ (a b c c d)
(append '() '(a b c)) ⟹ (a b c)
```

We define **append** using recursion on the first list, `ls1`. Cdring on `ls1` ultimately produces the base case in which `ls1` is empty. In the base case, when `ls1` is empty, `ls2` is returned. Thus we can begin the definition with the base case:

```
(define append
  (lambda (ls1 ls2)
    (if (null? ls1)
        ls2
        ... )))
```

To express `(append ls1 ls2)` in terms of `(append (cdr ls1) ls2)`, observe that `(append (cdr ls1) ls2)` differs from `(append ls1 ls2)` only in the absence of the first top-level item in `ls1`. For example, if `ls1` is `(a b c)` and `ls2` is `(d e)`, then `(append (cdr ls1) ls2)` gives us `(b c d e)`, and only `(car ls1)` remains to be included. Thus when `ls1` is not empty, `(append ls1 ls2)` is the same as `(cons (car ls1) (append (cdr ls1) ls2))`. We can therefore complete the definition of **append**.

Program 4.1 append

```
(define append
  (lambda (ls1 ls2)
    (if (null? ls1)
        ls2
        (cons (car ls1) (append (cdr ls1) ls2)))))
```

Another procedure often used is the Scheme procedure **reverse**, which takes a list as its argument and builds a list consisting of the top-level items in its argument list taken in reverse order. For example,

```
(reverse '(1 2 3 4 5)) ⟹ (5 4 3 2 1)
(reverse '((1 2) (3 4) (5 6))) ⟹ ((5 6) (3 4) (1 2))
```

We again use recursion and look at what **reverse** does to the cdr of the list **ls**. In the first example above,

```
(reverse '(2 3 4 5)) ⟹ (5 4 3 2)
```

To get **reverse** of (1 2 3 4 5) from (5 4 3 2), we must put the 1 into the last position in the list. We can do this with the procedure **append** if we make the 1 into a list (1) and then append (1) to the end of (5 4 3 2). This is the key to writing the definition of the procedure **reverse**.

We take the empty list as the base case and note that if we reverse the items in the empty list, we still have the empty list. Thus we can begin the definition with the terminating condition, which says that if the list is empty, the empty list is returned.

```
(define reverse
  (lambda (ls)
    (if (null? ls)
        '()
        ... )))
```

To get (**reverse ls**) from (**reverse (cdr ls)**), we must append the list that is the value of (**reverse (cdr ls)**) to the front of the list that is the value of (**list (car ls)**). We then complete the definition with

Program 4.2 **reverse**

```
(define reverse
  (lambda (ls)
    (if (null? ls)
        '()
        (append (reverse (cdr ls)) (list (car ls))))))
```

A list of numbers (or n-tuple) is said to be sorted in increasing order if each number in the list is less than or equal to the number following it in the list. For example, (2.3 4.7 5 8.1) is sorted in increasing order. If we have two lists, each sorted in increasing order, we can merge them into a single list in increasing order. For example, if the list given above is merged with the list (1.7 4.7), we get the list (1.7 2.3 4.7 4.7 5 8.1).

Let us now write a procedure **merge**, which takes two n-tuples, **sorted-ntpl1** and **sorted-ntpl2**, which have already been sorted in increasing order,

and builds the list obtained by merging them into one sorted n-tuple. If either list is empty, **merge** returns the other list. Otherwise we compare the car of the lists and cons the smaller one onto the list obtained by merging the rest of the two lists. This analysis leads to the following definition:

Program 4.3 `merge`

```
(define merge
  (lambda (sorted-ntpl1 sorted-ntpl2)
    (cond
      ((null? sorted-ntpl1) sorted-ntpl2)
      ((null? sorted-ntpl2) sorted-ntpl1)
      ((< (car sorted-ntpl1) (car sorted-ntpl2))
       (cons (car sorted-ntpl1)
             (merge (cdr sorted-ntpl1) sorted-ntpl2)))
      (else (cons (car sorted-ntpl2)
                  (merge sorted-ntpl1 (cdr sorted-ntpl2)))))))
```

We shall use **merge** in Chapter 10 when we discuss the sorting of lists.

The definition of **reverse** used the procedure **append**, which was defined earlier. It does not matter which was defined first, as long as both are defined when the procedure **reverse** is invoked.

The test of whether a nonnegative integer is even or odd gives us another good example of one procedure using another in its definition. There are many more direct ways of defining the predicates **even?** and **odd?**, but the one we present now was chosen because it illustrates how each of two procedures invokes the other in its definition. We use the fact that an integer is even if its predecessor is odd and odd if its predecessor is even. Starting with any nonnegative integer, reducing it successively by 1 will eventually bring it to 0, which is even. This analysis leads us to the following definitions:

Program 4.4 `even?`

```
(define even?
  (lambda (int)
    (if (zero? int)
        #t
        (odd? (sub1 int)))))
```

and

Program 4.5 odd?

```
(define odd?
  (lambda (int)
    (if (zero? int)
        #f
        (even? (sub1 int)))))
```

In the definition of the procedure **even?**, the procedure **odd?** is called, and in the definition of **odd?**, the procedure **even?** is called. This is an example of *mutual recursion* in which each procedure calls the other. The two procedures are said to be *mutually recursive.*

The procedure **remove-1st** defined in Chapter 2 removed the first top-level occurrence of an item from a list of items. Let us now define a procedure **remove** that removes *all* top-level occurrences of **item** from a list **ls**. As before, the recursion will be flat, but now we continue the recursion until all top-level occurrences of **item** have been removed from **ls**. The base condition is **(null? ls)**, and when it is true, the empty list is returned. Thus we begin our definition with:

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

Next, if **ls** is not empty, **(remove item (cdr ls))** is exactly the same as **(remove item ls)** when the first item in **ls** is **item**, for that item is removed. On the other hand, when the first item in **ls** is not **item**, then we must cons it onto **(remove item (cdr ls))** in order to get **(remove item ls)**. Thus we complete the definition, which is presented in Program 4.6.

The definition of **remove** differs from that of **remove-1st** in the middle clause of the cond expression. In **remove-1st** the recursion stopped when the first occurrence of **item** was found, whereas in **remove** the recursion continues. This difference is typical of what we see if we compare the definitions of procedures that stop after the first occurrence of an item to those that continue to the end of the list. The procedure **remove** uses **equal?** to test for sameness. You could write a version named **remq** that uses **eq?** to test for sameness and

Program 4.6 `remove`

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

a version named **remv** that uses **eqv?** to test for sameness. The exercises contain other procedures involving flat recursion that go to the end of the lists instead of stopping after the first occurrence of a given item.

Exercises

Exercise 4.1: **insert-left**

Define a procedure **insert-left** with parameters **new**, **old**, and **ls** that builds a list obtained by inserting the item **new** to the left of each top-level occurrence of the item **old** in the list **ls**. Test your procedure on:

```
(insert-left 'z 'a '(a b a c a)) ⟹ (z a b z a c z a)
(insert-left 0 1 '(0 1 0 1)) ⟹ (0 0 1 0 0 1)
(insert-left 'dog 'cat '(my dog is fun)) ⟹ (my dog is fun)
(insert-left 'two 'one '()) ⟹ ()
```

Exercise 4.2: **insert-right**

Define a procedure **insert-right** with parameters **new**, **old**, and **ls** that builds a list obtained by inserting the item **new** to the right of each top-level occurrence of the item **old** in the list **ls**. Test your procedure on:

```
(insert-right 'z 'a '(a b a c a)) ⟹ (a z b a z c a z)
(insert-right 0 1 '(0 1 0 1)) ⟹ (0 1 0 0 1 0)
(insert-right 'dog 'cat '(my dog is fun)) ⟹ (my dog is fun)
(insert-right 'two 'one '()) ⟹ ()
```

Exercise 4.3: **subst**

Define a procedure **subst** with parameters **new**, **old**, and **ls** that builds a list obtained by replacing each top-level occurrence of the item **old** in the list **ls** by the item **new**. Test your procedure on:

```
(subst 'z 'a '(a b a c a)) ⟹ (z b z c z)
(subst 0 1 '(0 1 0 1)) ⟹ (0 0 0 0)
(subst 'dog 'cat '(my dog is fun)) ⟹ (my dog is fun)
(subst 'two 'one '()) ⟹ ()
```

Exercise 4.4: `deepen-1`
Define a procedure `deepen-1` with parameter `ls` that wraps a pair of parentheses around each top-level item in `ls`. Test your procedure on:

```
(deepen-1 '(a b c d)) ⟹ ((a) (b) (c) (d))
(deepen-1 '((a b) (c (d e)) f)) ⟹ (((a b)) ((c (d e))) (f))
(deepen-1 '()) ⟹ ()
```

4.3 Deep Recursion

In this section, we consider recursion over all the sublists of a list. We say that the sublist `(b c)` is *nested* in the list `(a (b c))`. It is convenient to have some way of describing how deep the nesting is. If an item is not enclosed by parentheses, that item has *nesting level* 0. For example, the item `bird` has nesting level 0. The elements of a list such as `(a b c)` have nesting level 1. Thus `b` has nesting level 1 while the whole list `(a b c)` has nesting level 0. Then each additional layer of parentheses adds 1 to the nesting level, so that the nesting level of the item `c` in `(a (b (c d)))` is 3. The objects in the list that have nesting level 1 are the *top-level* objects of the list. The top-level objects in the list `(a (b c) (d (e f)))` are `a`, `(b c)`, and `(d (e f))`.

We define a procedure `count-all` with parameter `ls` that counts those items in the list `ls` that are not pairs. For example

1. `(count-all '((a b) c () ((d (e))))) ⟹ 6`
2. `(count-all '(() () ())) ⟹ 3`
3. `(count-all '(((())))) ⟹ 1`
4. `(count-all '()) ⟹ 0`

To simplify our discussion, we use the adjective *atomic* to describe an item that is not a pair. In this case, all of the atomic items in the list were counted, not just the top-level items. Since the empty list is not a pair, the empty lists that are included as items within the lists of Examples 1, 2, and 3 are counted as atomic items in the lists.

The base case for the recursion is the empty list, for in that case, `count-all` returns zero. Thus the definition begins with:

```
(define count-all
  (lambda (ls)
    (cond
      ((null? ls) 0)
      ... )))
```

If `ls` is not empty, we proceed as we did in our previous examples and consider how we can get `(count-all ls)` from `(count-all (cdr ls))`. The two differ by the number of atomic items in `(car ls)`. If `(car ls)` is atomic, then `(count-all ls)` has a value that is just one greater than the value of `(count-all (cdr ls))`. Thus we can continue the definition with:

```
(define count-all
  (lambda (ls)
    (cond
      ((null? ls) 0)
      ((not (pair? (car ls))) (add1 (count-all (cdr ls))))
      ... )))
```

When `(car ls)` is a pair (as is the case in Examples 1 and 3), we must count the atomic items in `(car ls)` and add that amount to the value of `(count-all (cdr ls))` to get the value of `(count-all ls)`. Thus we complete the definition with:

Program 4.7 `count-all`

```
(define count-all
  (lambda (ls)
    (cond
      ((null? ls) 0)
      ((not (pair? (car ls))) (add1 (count-all (cdr ls))))
      (else (+ (count-all (car ls)) (count-all (cdr ls)))))))
```

In fact, we can combine the last two cond clauses if we write the definition as follows:

```
(define count-all
  (lambda (ls)
    (cond
      ((null? ls) 0)
      (else (+ (if (pair? (car ls))
                   (count-all (car ls))
                   1)
               (count-all (cdr ls)))))))
```

The recursion described differs from flat recursion in that when the car of the list is a pair, we apply the procedure being defined both to the car and to the cdr of the list. In flat recursion, the procedure being defined was applied only to the cdr of the list. When the recursion is over all of the atomic items of a list, so that in the recursive step the procedure is applied to the `car` of the list and to the `cdr` of the list, we call it a *deep recursion*. A procedure defined using a deep recursion will be referred to as a *deeply recursive* procedure or simply a *deep procedure* to distinguish it from a flat procedure. Deep recursion is also called *tree recursion*.

Before leaving the definition of `count-all`, we should observe that we could have avoided the use of the `not` in the second cond clause by changing the order in which we considered the last two cases. That would give us the definition:

```
(define count-all
  (lambda (ls)
    (cond
      ((null? ls) 0)
      ((pair? (car ls))
       (+ (count-all (car ls)) (count-all (cdr ls))))
      (else (+ 1 (count-all (cdr ls)))))))
```

Many of the flat procedures defined earlier have analogs that are deep procedures. To illustrate this, we consider the procedure `remove-all`, which is analogous to `remove`. The procedure `remove-all` removes all occurrences of an item `item` from a list `ls`. For example,

```
(remove-all 'a '((a b (c a)) (b (a c) a))) ⟹ ((b (c)) (b (c)))
```

The base case is the empty list, and when `ls` is empty, the empty list is returned. Thus we begin the definition of `remove-all` with:

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

We next express **(remove-all item ls)** in terms of **(remove-all item (cdr ls))**. If **(equal? (car ls) item)** returns true, then **(remove-all item ls)** is the same as **(remove-all item (cdr ls))**, and we have:

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

If **(car ls)** is a pair that is not the same as **item**, then we remove all occurrences of **item** from **(car ls)** and cons the result onto **(remove-all item (cdr ls))**. Thus,

```
(define remove-all
  (lambda (item ls)
    (cond
      ((null? ls) '())
      ((equal? (car ls) item) (remove-all item (cdr ls)))
      ((pair? (car ls))
       (cons (remove-all item (car ls)) (remove-all item (cdr ls))))
      ... )))
```

Finally, if **(car ls)** is atomic and is not the same as **item**, we must cons it back onto **(remove-all item (cdr ls))** in order to get **(remove-all item ls)**. We wrap up the definition in Program 4.8. We can combine the last two cond clauses if we rewrite the definition as follows:

```
(define remove-all
  (lambda (item ls)
    (cond
      ((null? ls) '())
      ((equal? (car ls) item) (remove-all item (cdr ls)))
      (else (cons (if (pair? (car ls))
                      (remove-all item (car ls))
                      (car ls))
                  (remove-all item (cdr ls)))))))
```

Program 4.8 `remove-all`

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

In this example, we again see that when `(car ls)` is a pair not equal to `item`, the procedure `remove-all` is applied recursively to both the `car` and the `cdr` of `ls`. Thus `remove-all` displays this characteristic behavior of deeply recursive procedures.

We used `equal?` to test for sameness in the definition of `remove-all`. If the arguments to which `item` is bound are always symbols, we can use `eq?` to test for sameness. In this case, we know that the item that is the same as the symbol we are removing is never a pair, so it is expedient to test for `pair?` first. We can write the definition of `remq-all` as shown in Program 4.9. We can similarly define `remv-all`, which uses `eqv?` in place of `eq?`.

Program 4.9 `remq-all`

```
(define remq-all
  (lambda (symbl ls)
    (cond
      ((null? ls) '())
      ((pair? (car ls))
       (cons (remq-all symbl (car ls)) (remq-all symbl (cdr ls))))
      ((eq? (car ls) symbl) (remq-all symbl (cdr ls)))
      (else (cons (car ls) (remq-all symbl (cdr ls)))))))
```

When the flat procedure `reverse` is applied to a list, we get a new list with the top-level objects in reverse order. Thus,

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

We can also define a procedure `reverse-all` that not only reverses the order

of the top-level objects in the list but also reverses the order of the objects at each nesting level with the sublists. We would then have:

```
(reverse-all '(a (b c) (d (e f)))) ⟹ (((f e) d) (c b) a)
```

For the base case, the list is empty, and **(reverse-all '())** returns the empty list. Thus the definition begins with:

```
(define reverse-all
  (lambda (ls)
    (cond
      ((null? ls) '())
      ... )))
```

To carry out the recursion, we build **(reverse-all ls)** from **(reverse-all (cdr ls))**. In the latter, all of the elements of **(reverse-all (cdr ls))** are already in the correct order. We have to see how to include the items of **(car ls)**. If **(car ls)** is a pair, we have to reverse its elements and place them at the end of **(reverse-all (cdr ls))** with the procedure **append**. Thus we have:

```
(define reverse-all
  (lambda (ls)
    (cond
      ((null? ls) '())
      ((pair? (car ls))
       (append (reverse-all (cdr ls))
               (list (reverse-all (car ls)))))
      ... )))
```

In the remaining case, **(car ls)** is not a pair, so we merely place it at the end of **(reverse-all (cdr ls))**.

```
(define reverse-all
  (lambda (ls)
    (cond
      ((null? ls) '())
      ((pair? (car ls))
       (append (reverse-all (cdr ls))
               (list (reverse-all (car ls)))))
      (else
       (append (reverse-all (cdr ls))
               (list (car ls)))))))
```

Once again, in this recursion we see the typical form of a deep recursion. We applied `reverse-all` to both the car and the cdr of the list in the second cond clause.

It is instructive to look back at this definition of `reverse-all` and observe the similarity between the two alternatives that begin with `append` in the last two cond clauses. They differ only in the application of `reverse-all` to `(car ls)` in the last line. Because of this similarity, we can combine the two append expressions into one expression by putting the conditional branch after `(reverse-all (cdr ls))`. We get the following version of the definition of `reverse-all`:

Program 4.10 `reverse-all`

```
(define reverse-all
  (lambda (ls)
    (if (null? ls)
        '()
        (append (reverse-all (cdr ls))
                (list (if (pair? (car ls))
                          (reverse-all (car ls))
                          (car ls)))))))
```

In this section, we have seen how to write deeply recursive procedures. These have the characteristic property that a recursive step applies the procedure being defined to both the car and the cdr of the list.

Exercises

Exercise 4.5: `subst-all`, `substq-all`
Define a procedure `subst-all` with call structure `(subst-all new old ls)` that replaces each occurrence of the item `old` in a list `ls` with the item `new`. Test your procedure on:

```
(subst-all 'z 'a '(a (b (a c)) (a (d a))))
                          ⟹ (z (b (z c)) (z (d z)))
(subst-all 0 '(1) '(((1) (0)))) ⟹ ((0 (0)))
(subst-all 'one 'two '()) ⟹ ()
```

Also define a procedure `substq-all` in which the parameters `new` and `old` are only bound to symbols, so that `eq?` can be used for the sameness test.

Exercise 4.6: `insert-left-all`
Define a procedure `insert-left-all` with call structure `(insert-left-all new old ls)` that inserts the item `new` to the left of each occurrence of the item `old` in the list `ls`. Test your procedure on:

```
(insert-left-all 'z 'a '(a ((b a) ((a (c))))))
                                ⟹ (z a ((b z a) ((z a (c)))))
(insert-left-all 'z 'a '(((a)))) ⟹ (((z a)))
(insert-left-all 'z 'a '()) ⟹ ()
```

Exercise 4.7: `sum-all`
Define a procedure `sum-all` that finds the sum of the numbers in a list that may contain nested sublists of numbers. Test your procedure on:

```
(sum-all '((1 3) (5 7) (9 11))) ⟹ 36
(sum-all '(1 (3 (5 (7 (9)))))) ⟹ 25
(sum-all '()) ⟹ 0
```

4.4 Tree Representation of Lists

There is a convenient way of thinking of a list graphically as a *tree* that has its root at the top and grows by branching downward. The original list is a *node* that is located at the *root*. Each top-level object in the list forms a new node connected to the root node by a *branch*. Each sublist itself then becomes the root of a *subtree*, and the tree grows downward. For example, the tree representing the list `(a (b c d) ((e f) g))` is given in Figure 4.11. Each item or sublist of the original list is a node of this tree. Each sublist is itself the root of a subtree of the original tree. Thus `((e f) g)` corresponds to the subtree given in Figure 4.12.

An item at the lower end of a branch that is not the top end of another branch is called a *leaf* of the tree. We can readily see how deeply an item is nested in the list by looking at its nesting level in the tree. For example, in Figure 4.11, the leaf `a` is at nesting level 1 and the leaf `e` at nesting level 3. We say that the *depth* of a list is the maximum of the nesting levels of all of its items. The list `(a (b c d) ((e f) g))` has depth 3. With the tree growing downward, we can say that the depth of a list is the nesting level of its lowest leaves.

To *traverse* a tree, that is, to move down the tree from one node to another, we use the procedures `car` and `cdr`. Taking the `car` of a list corresponds to

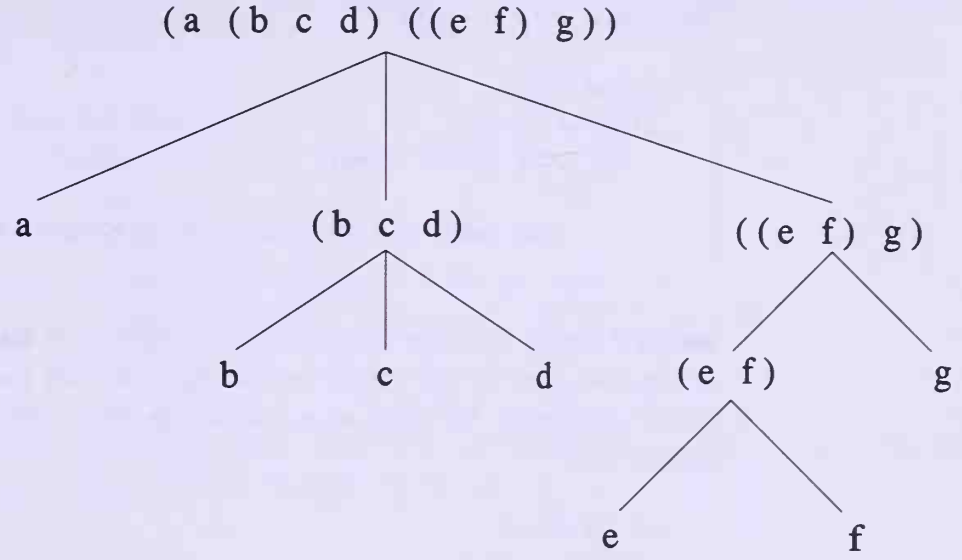

Figure 4.11 Tree representation of the list `(a (b c d) ((e f) g))`

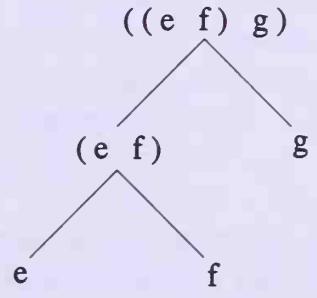

Figure 4.12 The subtree `((e f) g)`

moving down one node on the leftmost branch of the tree. Taking the `cdr` of a list corresponds to considering the tree that is left when the leftmost branch is omitted. Thus when taking the `car`, we move down one level on the tree. When taking the `cdr`, we stay at the same level of the tree. With an appropriate sequence of `car` and `cdr` applications, we can reach any node of a tree. For example, in the tree in Figure 4.11, the node `(e f)` is reached using `caaddr`.

We define a procedure `depth` that takes `item` as its argument and returns its depth. The item may be either atomic or a list. If `item` is atomic, we

Program 4.13 depth

```
(define depth
  (lambda (item)
    (if (not (pair? item))
        0
        (max (add1 (depth (car item))) (depth (cdr item))))))
```

assign it depth 0. Since the empty list is atomic, it also has depth 0. We take as the base case for the recursive definition the test `(not (pair? item))`, for that corresponds to being at a leaf of the tree. We begin the definition of `depth` with:

```
(define depth
  (lambda (item)
    (if (not (pair? item))
        0
        ... )))
```

The depth of the whole tree is the larger of the depth of its leftmost branch and the depth of the rest of its branches. Taking the `car` of the list moves down one node on the leftmost branch, so that the depth of the whole leftmost branch is one greater than the depth of `(car item)`. The depth of the rest of the branches is just the depth of `(cdr item)`. This gives us the definition displayed in Program 4.13.

The procedure `depth` gives us the maximum number of levels in a tree representing its argument. We next define a procedure that gives us a list of the leaves on the tree as a list of atomic items, where each leaf is raised out of its sublist to be at top level. We call this procedure `flatten`. When we apply it to the list `(a (b c d) ((e f) g))`, we get `(a b c d e f g)`. The parameter of the procedure `flatten` will be `ls`. The base case is the empty list, which flattens into itself. Thus we begin the definition of `flatten` with:

```
(define flatten
  (lambda (ls)
    (cond
      ((null? ls) '())
      ... )))
```

When `ls` is not empty, we build `(flatten ls)` from `(flatten (cdr ls))` by first determining whether `(car ls)` is a pair. If it is, we flatten `(car ls)`

and append the already flattened `(flatten (cdr ls))` to it to get `(flatten ls)`. This gives us

```
(define flatten
  (lambda (ls)
    (cond
      ((null? ls) '())
      ((pair? (car ls))
       (append (flatten (car ls)) (flatten (cdr ls))))
      ... )))
```

In the remaining case, `(car ls)` is atomic, so we cons it onto `(flatten (cdr ls))`, and we complete the definition with

Program 4.14 `flatten`

```
(define flatten
  (lambda (ls)
    (cond
      ((null? ls) '())
      ((pair? (car ls))
       (append (flatten (car ls)) (flatten (cdr ls))))
      (else (cons (car ls) (flatten (cdr ls)))))))
```

We have discussed flat and deep recursion. A flat recursion is over the top-level items of a list. This is equivalent to a recursion over the nodes of the corresponding tree, which are one level below the root. A deep recursion is over all of the items in the list. This is equivalent to a recursion over the leaves of the corresponding tree. That is why deep recursion is also referred to as *tree recursion*.

We conclude this section with an example of a procedure that removes an item from a list but only the first (leftmost) occurrence of that item in the list. Let us name the procedure `remove-leftmost` and look at a couple of examples.

1. `(remove-leftmost 'b '(a (b c) (c (b a))))`
   $\Longrightarrow$ `(a (c) (c (b a)))`
2. `(remove-leftmost '(c d) '((a (b c)) ((c d) e)))`
   $\Longrightarrow$ `((a (b c)) (e))`

In Example 1, the first `b` that occurs in `(b c)` is removed, but the second `b` that occurs in `(c (b a))` is not removed. We denote the item to be removed by `item` and the list by `ls`. The base case is again the empty list. When `ls` is empty, the empty list is returned. Thus we begin the definition with the terminating condition:

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

In order to take care of arguments like that in Example 2, we use `equal?` as the sameness predicate. If `(car ls)` is the same as `item`, the answer is `(cdr ls)`, so we continue the definition with:

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

If `(car ls)` is atomic and is not the same as `item`, the answer is obtained by consing `(car ls)` to the list obtained by removing the leftmost item from `(cdr ls)`. Thus we get:

```
(define remove-leftmost
  (lambda (item ls)
    (cond
      ((null? ls) '())
      ((equal? (car ls) item) (cdr ls))
      ((not (pair? ls))
       (cons (car ls) (remove-leftmost item (cdr ls))))
      ... )))
```

We still have the case in which `(car ls)` is a nonempty list not equal to `item`. If we analyze the recursion by looking at

```
(remove-leftmost item (cdr ls))
```

we see that we get a list with the first occurrence of `item` removed; but we do not know whether this was the first occurrence of `item` in `ls`. We want to

Program 4.15 `remove-leftmost`

```
(define remove-leftmost
  (lambda (item ls)
    (cond
      ((null? ls) '())
      ((equal? (car ls) item) (cdr ls))
      ((not (pair? (car ls)))
       (cons (car ls) (remove-leftmost item (cdr ls))))
      ((member-all? item (car ls))
       (cons (remove-leftmost item (car ls)) (cdr ls)))
      (else (cons (car ls) (remove-leftmost item (cdr ls)))))))
```

Program 4.16 `member-all?`

```
(define member-all?
  (lambda (item ls)
    (if (null? ls)
        #f
        (or (equal? (car ls) item)
            (and (not (pair? (car ls)))
                 (member-all? item (cdr ls)))
            (and (pair? (car ls))
                 (or (member-all? item (car ls))
                     (member-all? item (cdr ls))))))))
```

remove only the first occurrence of `item` in `ls`, and its first occurrence may not be in `(cdr ls)`. In order to use this kind of argument, we must first check to see whether the first occurrence of `item` in `ls` is in `(car ls)`. We do that with the helping procedure `member-all?`, a deeply recursive version of `member?`, that we define after this definition. If `item` is in `(car ls)`, we cons `(remove-leftmost item (car ls))` onto `(cdr ls)` to get the answer. Otherwise, we cons `(car ls)` onto `(remove-leftmost item (cdr ls))` to get the answer. Thus we complete the definition as shown in Program 4.15. The definition of `member-all?` is presented in Program 4.16.

A look at the definition of `remove-leftmost` reveals that the consequent in the third cond clause and the alternative in the else clause are the same. We can eliminate the repetition by interchanging the order of the tests we make. The new version is given in Program 4.17.

Program 4.17 `remove-leftmost`

```
(define remove-leftmost
  (lambda (item ls)
    (cond
      ((null? ls) '())
      ((equal? (car ls) item) (cdr ls))
      ((and (pair? (car ls)) (member-all? item (car ls)))
       (cons (remove-leftmost item (car ls)) (cdr ls)))
      (else (cons (car ls) (remove-leftmost item (cdr ls)))))))
```

The recursion in the procedure `remove-leftmost` differs from the list recursions done earlier in that we have to test whether `item` is in the car of the list before proceeding to build the answer. This means cdring through the car of the list twice in some cases. We shall return to the consideration of `remove-leftmost` in Chapter 5, where a definition is presented that avoids this double cdring. We have now seen various examples of both flat and deep (tree) recursions.

Exercises

Exercise 4.8: `count-parens-all`
Write the definition of a procedure `count-parens-all` that takes a list as its argument and counts the number of opening and closing parentheses in the list. Test your procedure on:

```
(count-parens-all '()) ⟹ 2
(count-parens-all '((a b) c)) ⟹ 4
(count-parens-all '(((a () b) c) () ((d) e))) ⟹ 14
```

Exercise 4.9: `count-background-all`
Define a procedure `count-background-all` that takes as its arguments `item` and a list `ls` and returns the number of items in `ls` that are not the same as `item`. Use the appropriate sameness predicate for the data shown in the examples. Test your procedure on:

```
(count-background-all 'a '((a) b (c a) d)) ⟹ 3
(count-background-all 'a '((((b (((a)) c)))))) ⟹ 2
(count-background-all 'b '()) ⟹ 0
```

Program 4.18 `fact`

```
(define fact
  (lambda (n)
    (if (zero? n)
        1
        (* n (fact (sub1 n))))))
```

Exercise 4.10: `leftmost`
Define a procedure `leftmost` that takes a nonempty list as its argument and returns the leftmost atomic item in the list. Test your procedure on:

```
(leftmost '((a b) (c (d e)))) ⟹ a
(leftmost '((((c ((e f) g) h))))) ⟹ c
(leftmost '(() a)) ⟹ ()
```

Exercise 4.11: `rightmost`
Define a procedure `rightmost` that takes a nonempty list as its argument and returns the rightmost atomic item in the list. Test your procedure on:

```
(rightmost '((a b) (d (c d (f (g h) i) m n) u) v)) ⟹ v
(rightmost '(((((((b (c))))))))) ⟹ c
(rightmost '(a ())) ⟹ ()
```

4.5 Numerical Recursion and Iteration

Recursion can also be used in numerical calculations. We consider several examples in this section. We begin with the procedure `fact`, which takes a nonnegative integer `n` as its parameter and returns its factorial—that is, the number multiplied successively by all the positive integers less than that number. For example, `(fact 5)` has the value $5 \times 4 \times 3 \times 2 \times 1 = 120$. We derive this procedure using much the same kind of reasoning as we used with lists, but instead of using `cdr` to reduce the size of the argument, we use `sub1`. Eventually the successive applications of `sub1` to the argument will reduce it to 0. We use the convention that the factorial of 0 is 1, so that `(fact 0)` is 1. The recursive step in this case is done by considering `(fact (sub1 n))`, which gives us the successive products of all of the positive integers less than `n`. To get `(fact n)` from `(fact (sub1 n))`, all we have to do is multiply it by `n`. From this, we get the definition for `fact` in Program 4.18.

When the procedure `fact` is applied to a number, say 3, a return table is built much the same as the one that was built for the procedure `swapper` in Chapter 2. The value of `(fact 3)` is denoted by *answer-1*. It is 3 times `(fact 2)`, so the evaluation of *answer-1* must wait until *answer-2* is evaluated, where *answer-2* is `(fact 2)`. Thus the first two rows of the return table are:

answer-1 is `(* 3` *answer-2*`)`
answer-2 is `(fact 2)`

When we evaluate `(fact 2)`, the return table becomes

answer-1 is `(* 3` *answer-2*`)`
answer-2 is `(* 2` *answer-3*`)`
answer-3 is `(fact 1)`

When we evaluate `(fact 1)`, the return table becomes

answer-1 is `(* 3` *answer-2*`)`
answer-2 is `(* 2` *answer-3*`)`
answer-3 is `(* 1` *answer-4*`)`
answer-4 is `(fact 0)`

where `(fact 0)` is `1`. Now that we have found that *answer-4* is `1`, we work our way up the table, replacing each answer on the right side by the value obtained for it in the row below. This process is known as *backward substitution*. This gives us:

answer-4 is `1`
answer-3 is `1`
answer-2 is `2`
answer-1 is `6`

so `(fact 3)` is `6`. In finding `(fact 3)`, the return table has four rows. In the last row, the value of the variable on the left was obtained directly from the terminating condition of the program. Then each of the other three variables on the right was computed with a multiplication, so there were three multiplications required to complete the computation of `(fact 3)`. The building up of the return table and the subsequent backward substitution may be summarized in the following:

```
(fact 3)
(* 3 (fact 2))
(* 3 (* 2 (fact 1)))
(* 3 (* 2 (* 1 (fact 0))))
(* 3 (* 2 (* 1 1)))
(* 3 (* 2 1))
(* 3 2)
6
```

In general, to find the factorial of the number n, there would be $n + 1$ invocations of procedure `fact`. Thus the return table has $n + 1$ rows. In the last row, the value on the right is found to be `1`—the value returned when the terminating condition is true. In each of the other n rows of the return table, a multiplication is performed to find the value on the right, making a total of n multiplications to complete the computation.

We observed that a return table is constructed when we compute the factorial using the recursive procedure `fact`. When the terminating condition becomes true, the backward substitution must be performed on the return table to get the answer. When the computation requires the construction of a return table and backward substitution to get the answer, we say that the computation is using a *recursive process*. We now look at another way of defining a procedure to compute the factorial of a number that does not build a return table. Instead, at each recursive invocation of the procedure, the computations are performed without having to wait for other needed values, and when the terminating condition is true, the answer is already computed and is returned. In general, when the computer carries out a computation without building a return table, so that backward substitution is not necessary, the computational process is called an *iterative process*.

We have seen that in programs like the one written for `fact`, there is an operation waiting for the value returned by the recursive procedure call. The computational process so defined is not implemented as an iterative process. On the other hand, we saw several iterative procedures, such as `member?`, in which no operations waited for values returned by the recursive procedure calls. In some programming language implementations, when an iterative procedure is executed, it is still possible that a return table is built up and later reduced by backward substitution. However, in Scheme, when a procedure is intended to be iterative, the computation is always implemented in such a way that no return table is needed.

To implement the computation of the factorial procedure as an iterative process, we define a procedure named `fact-it` that has two parameters: `n`,

which is the integer whose factorial we are computing, and `acc`, another integer, called an *accumulator*, which stores the answer at each step. Here is how it works in computing the factorial of 3. Initially, `n` is bound to 3 and `acc` is bound to 1. On each recursive invocation of `fact-it`, `n` is reduced by 1, and `acc` is replaced by its old value multiplied by the previous value of `n`. When the base case `(zero? n)` is true, `acc` is equal to the answer 6. This is illustrated in the following table. The initial values of `n` and `acc` are in the first row. The entries in the first column decrease by 1 while each entry in the second column is computed by multiplying the two entries in the preceding row.

n	acc
3	1
2	3
1	6
0	6

To define `fact-it`, we begin with the base case for which `n` is zero. When `(zero? n)` is true, the accumulator has the answer, so `acc` is returned. Thus we begin the definition with:

```
(define fact-it
  (lambda (n acc)
    (if (zero? n)
        acc
        ... )))
```

If `n` is not zero, we call `fact-it` with `n` reduced by one and the accumulator multiplied by `n`, so the definition is completed with:

Program 4.19 `fact-it`

```
(define fact-it
  (lambda (n acc)
    (if (zero? n)
        acc
        (fact-it (sub1 n) (* acc n)))))
```

Let's walk through an invocation `(fact-it 3 1)`, writing the successive recursive invocations of `fact-it`, and finally writing the value 6 that is returned:

```
(fact-it 3 1)
(fact-it 2 3)
(fact-it 1 6)
(fact-it 0 6)
6
```

In this computation, no return table is built up waiting for uncomputed values to be returned. The accumulator is bound to the answer when the terminating condition is true, and the answer is returned without any backward substitution. The fact that there is no waiting operation on each recursive invocation of `fact-it` is seen when we look at the last line of the definition. After the procedure call, there is no further operation to be done. Compare this last line with the last line,

```
(* n (fact (sub1 n)))
```

in the definition of `fact`. We see that after the procedure `fact` is called, the result must still be multiplied by `n`. When `fact-it` is called, no additional operations are performed on the result. Thus `fact-it` runs as an iterative process, but `fact` does not. When we trace this iterative procedure, we see that the computation does not build up a return table of operations waiting for values to be returned.

If we count the number of times we call the procedure `fact-it` and the number of multiplications, we see that the total number of multiplications is the same for the procedures `fact-it` and `fact`. However, the backward substitution in the return table, which is built up when evaluating `fact`, requires more memory space than is needed when evaluating the iterative `fact-it`, which needs no return table. In the next section, we look at another example, the computation of the Fibonacci numbers, where the difference is more dramatic.

To compute the factorial of 3, we invoke `(fact-it 3 1)`. If we do not like to write the extra argument for the accumulator, we can define an iterative version of `fact` that takes only one argument by writing

```
(define fact
  (lambda (n)
    (fact-it n 1)))
```

Exercises

Exercise 4.12
Enter the procedure `fact` into the computer and compute `(fact n)` for `n = 10, 20, 30, 40, 50` and `100`. You will have an opportunity to observe how the implementation of Scheme you are using displays large numbers.

Exercise 4.13
What happens when you invoke `(fact 3.5)`?

Exercise 4.14: `harmonic-sum-it`
Define an iterative procedure `harmonic-sum-it` that sums the first n terms of the harmonic series

$$1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \frac{1}{5} + \cdots$$

Test your procedure by summing the harmonic series for 10 terms, 100 terms, 1000 terms, and 10,000 terms. It can be shown that

$$\frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} \leq \log n \leq 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n-1}$$

where $\log n$ is the natural logarithm of n. Using the Scheme procedure `log`, verify this inequality for the values of the sums computed above.

4.6 Analyzing the Fibonacci Algorithm

The following problem appeared in a textbook written in 1202 by the Italian mathematician Leonardo of Pisa, who was the son of Bonacci, so his nickname, taken from "filius Bonacci," became Fibonacci. How many pairs of rabbits are born of one pair in a year? It was assumed that every month a pair of rabbits produces another pair and that rabbits begin to bear young two months after their own birth.

The sequence of numbers that give the number of pairs of rabbits each month is 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377. This tells us that at the end of one month, the first pair had a pair of offsprings, so we have two pairs. At the end of two months, only one pair is old enough to have offsprings, so we have three pairs. At the end of three months, the first pair of offsprings is old enough to bear young, so this time we get two new pairs,

and we have five pairs altogether. If we continue in this way, we generate the sequence given above. Observe that each number in the sequence is the sum of the two numbers preceding it. It has become customary to begin the sequence with 0, 1, and use the algorithm that says that the next number is always the sum of the preceding two numbers. The nth number in this sequence is called the nth Fibonacci number.

We now define a procedure `fib` that takes a nonnegative integer `n` as its parameter and returns the Fibonacci number corresponding to `n`. We have `(fib 0)` is 0, `(fib 1)` is 1, `(fib 2)` is 1, `(fib 3)` is 2, and in general, for n > 1, `(fib` n`)` is the sum of `(fib (-` n `1))` and `(fib (-` n `2))`. We now use this last recursive condition to define the procedure `fib` in Program 4.20.

Program 4.20 `fib`

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

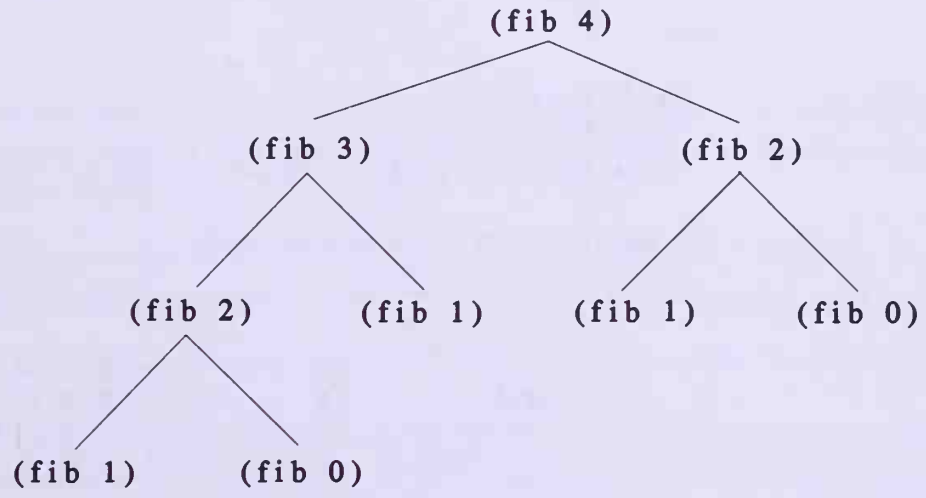

Figure 4.21 Recursion tree for `(fib 4)`

To trace how `(fib 4)` is evaluated, we make a tree (Figure 4.21) in which

the root is labeled (fib 4). This is evaluated by adding (fib 3) and (fib 2), so our tree will have two branches, one going to a node (fib 3) and the other to a node (fib 2). Each of these gives rise to two branches, (fib 3) giving rise to branches to the nodes (fib 2) and (fib 1), and (fib 2) giving rise to branches to the nodes (fib 1) and (fib 0). This continues until all of the leaves are either (fib 1) or (fib 0), which are known to be 1 and 0, respectively. This tree is an example of a *binary tree* because each node that is not a leaf has at most two branches going down from it.

From Figure 4.21, we see that each node corresponds to a procedure call that is made in evaluating (fib 4). In this case, there are nine procedure calls. Each *branch point* (a node from which two branches originate) corresponds to an addition, so there are four additions. In a similar way, we can build a recursion tree for (fib 5), and we will have fifteen nodes and seven branch points, hence fifteen procedure calls and seven additions. We suggest that you draw the recursion trees for (fib 5) and for (fib 6) to see how large they are and count the number of procedure calls and additions. It is not difficult to see from the trees that if (calls-fib n) tells how many procedure calls there are in computing (fib n) and (adds-fib n) tells how many additions there are in computing (fib n), then these procedures satisfy the relations

```
(calls-fib 0)  is 1
(calls-fib 1)  is 1
(calls-fib n)  is (add1 (+ (calls-fib (- n 1)) (calls-fib (- n 2))))
```

and

```
(adds-fib 0)  is 0
(adds-fib 1)  is 0
(adds-fib n)  is (add1 (+ (adds-fib (- n 1)) (adds-fib (- n 2))))
```

We get Table 4.22 for these quantities.

n	0	1	2	3	4	5	6	7	8	9	10
(fib n)	0	1	1	2	3	5	8	13	21	34	55
(calls-fib n)	1	1	3	5	9	15	25	41	67	109	177
(adds-fib n)	0	0	1	2	4	7	12	20	33	54	88

Table 4.22 Count of procedure calls and additions

The number of procedure calls and the number of additions increase so rapidly because in each procedure call, fib calls itself twice. This leads to

acc1	acc2
0	1
1	1
1	2
2	3
3	5
5	8
8	13

Table 4.23 Accumulator values for the iterative Fibonacci procedure

inefficiency since the same `fib` is called with the same arguments a number of times, so that the different recursive calls repeat each other's work. In the tree shown in Figure 4.21, `(fib 2)` is invoked twice and `(fib 1)` is invoked three times. We next look at an iterative method for computing the Fibonacci numbers.

A clue to how to set up an iterative process for computing the Fibonacci numbers is found by observing that it takes the previous two numbers to compute the next number in the sequence. Thus we have to store two numbers at each step. We begin by storing the first two Fibonacci numbers, `0` and `1` in accumulators, which we call `acc1` and `acc2`. Thus at the start,

acc1	acc2
0	1

At each step, `acc1` holds the current Fibonacci number and `acc2` holds the next one. Thus we can describe the algorithm that takes us from one step to the next as follows:

1. The new value of `acc1` is the same as the previous value of `acc2`.
2. The new value for `acc2` is the sum of previous values of `acc1` and `acc2`.

We apply these rules to extend the table to show the next six steps, as displayed in Table 4.23.

We are now ready to define a procedure `fib-it` that takes three arguments, a nonnegative integer `n`, and the two accumulators, `acc1` and `acc2`, and returns the Fibonacci number corresponding to `n`. There are two ways that we can use the algorithm given to write the code. In the first method, we can use the value stored in `acc1` (initially `0`) to give us the answer. In that case, one iteration of the algorithm gives us `(fib 1)`, two iterations give us `(fib 2)`, and in general n iterations give us `(fib` n`)` for any positive n. In the

Program 4.24 `fib-it`

```
(define fib-it
  (lambda (n acc1 acc2)
    (if (= n 1)
        acc2
        (fib-it (sub1 n) acc2 (+ acc1 acc2)))))
```

second method, we can use the value stored in `acc2` (initially `1`) to give us the answer. In this case, one iteration of the algorithm gives us `(fib 2)`, two iterations give us `(fib 3)`, and in general, $(n-1)$ iterations give us `(fib` n`)`. The second method is more efficient for getting the value of `(fib` n`)`. We opt to implement the second method.

Our iterative procedure `fib-it` takes three parameters: the positive integer `n` and the two accumulators `acc1` and `acc2`. To implement the algorithm stated above, we successively replace `acc2` by the sum of `acc1` and `acc2`, and replace `acc1` by the previous value of `acc2`. Then to compute the nth Fibonacci number, we must repeat the process $(n-1)$ times. We use the variable `n` as a counter and reduce it by one on each pass. When `n` reaches `1`, the accumulator `acc2` contains the answer. This leads to the definition given in Program 4.24.

Let's walk through `(fib-it 6 0 1)` to see how this works. On successive passes through the program, the following procedure calls are made:

```
(fib-it 6 0 1)
(fib-it 5 1 1)
(fib-it 4 1 2)
(fib-it 3 2 3)
(fib-it 2 3 5)
(fib-it 1 5 8)
8
```

and the answer is the final value of `acc2`, which is `8`. To compute the sixth Fibonacci number, we only make six procedure calls and 5 additions. In general, to compute the nth Fibonacci number, we make n procedure calls and do $n-1$ additions. This is a noticeable improvement over the number of procedure calls and additions when `fib` is invoked. The iterative version, `fib-it`, is certainly more efficient and saves a considerable amount of time in computing the Fibonacci numbers. The ordinary recursive version, `fib`, is less efficient but it does have the advantage of being easier to define directly in terms of the rule that defines the Fibonacci numbers.

Again, if we do not want to include the initial values of the accumulators in each procedure call, we can define the iterative version of `fib` as

```
(define fib
  (lambda (n)
    (if (zero? n)
        0
        (fib-it n 0 1))))
```

We have seen that some methods of evaluating a given expression may take more resources than other methods. The study of the efficiency of various algorithms is called the *analysis of algorithms*. Let us denote the total resources used in computing an expression that depends on an argument n to be `(res` n`)`. In our discussion, `fib` depended on the argument n, and we can define as the resources used the sum of `(calls-fib` n`)` and `(adds-fib` n`)`. Inspection of the table for `(calls-fib` n`)` shows that the following relation exists between `(calls-fib` n`)` and `(fib` n`)`:

$$\texttt{(calls-fib } n\texttt{)} = \texttt{(add1 (* 2 (sub1 (fib (add1 } n\texttt{)))))}$$

Similarly, `(adds-fib` n`)` and `(fib` n`)` are related by

$$\texttt{(adds-fib } n\texttt{)} = \texttt{(sub1 (fib (add1 } n\texttt{)))}$$

so that

$$\texttt{(res } n\texttt{)} = \texttt{(add1 (* 3 (sub1 (fib (add1 } n\texttt{)))))}$$

We now derive an estimate for `(fib` n`)`. If you prefer, you can skip to the formula for `(fib` n`)` given at the end of the derivation. We use the fact that if a procedure satisfies the Fibonacci *recurrence relation* $F(n) = F(n-1) + F(n-2)$ and the *initial conditions* $F(0) = 0$ and $F(1) = 1$, then $F(n) =$ `(fib` n`)` for all n. We begin by making a rather arbitrary assumption: that $F(n)$ gets large like some number a raised to the nth power. We then look for restrictions that can be placed on the number a in order for the function a^n to satisfy the Fibonacci recurrence relation. If we are lucky enough to find such conditions that determine a, we have solved the problem of finding a formula for $F(n)$. Substitution of a^n into the recurrence relation gives us

$$a^n = a^{n-1} + a^{n-2}$$

and dividing through by a^{n-2} gives us the simple relation

$$a^2 = a + 1$$

This quadratic equation has the positive root

$$a = \frac{(1+\sqrt{5})}{2}$$

and the negative root

$$b = \frac{(1-\sqrt{5})}{2}$$

which are approximately 1.618 and -0.618, respectively.

It is easily verified that since both a^n and b^n satisfy the Fibonacci recurrence relation, then for any pair of numbers A and B, the sum $F(n) = Aa^n + Bb^n$ also satisfies the same recurrence relation. We thus try to find values of A and B so that $F(0) = 0$ and $F(1) = 1$. The constants A and B will now be evaluated from the fact that

$$F(0) = 0 = A + B$$
$$F(1) = 1 = Aa + Bb$$

We find that $A = -B = 1/\sqrt{5}$ and that with these values of A and B, $F(n)$ and (`fib` n) are the same for $n = 0$ and $n = 1$, and that they both satisfy the Fibonacci recurrence relation for all n. This means that they are the same for all n, and we have

$$(\texttt{fib}\ n) = F(n) = \frac{1}{\sqrt{5}}(a^n - b^n) = \frac{1}{\sqrt{5}}\left[\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right]$$

Thus (`fib` n) is somewhat less than 1.7^n, and (`res` n) is somewhat less than $3(1.7^n)$.

In general, we say that the procedure (`res` n) is of order $O(f(n))$ for some function f of n if there is a constant K such that (`res` n) $\leq Kf(n)$ when n is sufficiently large. In our case, we can say (`res` n) $= O(1.7^n)$ and since it grows like the nth power of a number greater than 1, we say that (`res` n) has *exponential order* when computing (`fib` n).

On the other hand, the operation count (`res` n) for computing (`fib-it` n `0 1`) is $2n - 1$, which is simply $O(n)$. Here the n does not appear in an exponent, but rather (`res` n) is simply a constant times n. We say that in this case, (`res` n) has *linear order*. Thus the time required to compute (`fib` n) grows exponentially with n, while the time required to compute (`fib-it` n

Program 4.25 `reverse-it`

```
(define reverse-it
  (lambda (ls acc)
    (if (null? ls)
        acc
        (reverse-it (cdr ls) (cons (car ls) acc)))))
```

0 1) grows linearly with n. We have seen what a dramatic difference this makes.

In our two examples of iterative programs, we used procedures defined on numbers. It is also possible to use similar methods to write iterative versions of some of the list-processing procedures we considered earlier. For example, consider the procedure `reverse`, which takes a list of items `ls` and returns a list with the items in reverse order. We can write an iterative version `reverse-it` that takes two arguments, a list of items `ls` and an accumulator `acc`, which is initialized to be the empty list. The code for `reverse-it` is given in Program 4.25. We now can obtain the procedure `reverse` by writing

```
(define reverse
  (lambda (ls)
    (reverse-it ls '())))
```

We leave it as an exercise to compare this iterative version with the earlier recursive version of `reverse`. If we actually walk through each version with a simple example, we see that the accumulator already is the answer when `ls` is empty, whereas in the recursive version, we still have to use backward substitution in a return table to get the answer. Furthermore the iterative version does not use the helping procedure `append`. Generally, iterative versions tend to require more arguments.

Exercises

Exercise 4.15

Rewrite the recursive version of the procedure `fib` with the line

```
(writeln "n = " n)
```

inserted just below the line `(lambda (n)`. Then compute `(fib 4)` and compare the results with the tree in Figure 4.21. Also compute `(fib 5)` and `(fib 6)` and observe how the number of recursive calls to `fib` increases.

Exercise 4.16
Rewrite the iterative version of the procedure `fib-it` with the line

```
(writeln "n = " n ", acc1 = " acc1 ", acc2 = " acc2)
```

inserted just below the line

```
(lambda (n acc1 acc2)
```

Compute `(fib-it 4 0 1)` and compare the output with the output for `(fib 4)` in the preceding exercise. Do the same for `(fib-it 5 0 1)` and `(fib-it 6 0 1)`.

Exercise 4.17: `calls-fib`, `adds-fib`
Write the definitions of the procedures `calls-fib` and `adds-fib` discussed in this section. Test your procedures on the values given in Table 4.22. Also evaluate each of these procedures for larger values of n to get an idea of their rates of growth.

Exercise 4.18: `length-it`
Write an iterative version `length-it` of the procedure `length` that computes the length of a list.

Exercise 4.19: `mk-asc-list-of-ints`, `mk-desc-list-of-ints`
Write an iterative procedure `mk-asc-list-of-ints` that, for any integer `n`, produces a list of the integers from `1` to `n` in ascending order. Then write an iterative procedure `mk-desc-list-of-ints` that, for any integer `n`, produces a list of integers from `n` to `1` in descending order.

Exercise 4.20: `occurs`, `occurs-it`
Define both recursive and iterative versions of a procedure `occurs` that counts the number of times an item occurs at the top level in a list. Call the iterative version `occurs-it`. Test your procedures by counting how many times the item `a` occurs at top level in each of the following lists:

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