7 Abstracting Procedures

7.1 Overview

In this chapter, we first see how procedures can be passed as arguments to other procedures and how procedures may be the values of other procedures. We illustrate these ideas with a development of the Ackermann procedure. We then show how a procedure of two arguments may be rewritten as a procedure of one argument whose value is a procedure of one argument. This process is called currying. We next look at several programs that are similar in structure and we abstract these common features in a program that can be used easily to generate any other program with these features. This process is called *procedural abstraction.* Flat recursion on lists is often encountered in programming, so we have selected it as the first candidate for abstraction. That is followed by an abstraction of deep recursion.

7.2 Procedures as Arguments and Values

In this section, we shall study the use of procedures as arguments to other procedures and as values of procedures. In Chapter 1, we included procedures as a type of datum and have on occasion used procedures as arguments to other procedures. For example, in the definition of `max` in terms of `extreme-value` in Chapter 3, we passed the procedure `>` as an argument to the procedure `extreme-value`. In Scheme, all procedures may be used as arguments to other procedures and as values of procedures. This idea is illustrated by many examples in this section.

Suppose we have a list of numbers, such as `(1 3 5 7 9)`, and we want to

Program 7.1 `map`

```
(define map
  (lambda (proc ls)
    (if (null? ls)
        '()
        (cons (proc (car ls)) (map proc (cdr ls))))))
```

produce a new list that is obtained from the old by adding 1 to each item in the list, so that in our example, we would get `(2 4 6 8 10)`. We can define a procedure `add1-to-each-item` that takes a list `ls` and returns the new list with each number augmented by 1.

```
(define add1-to-each-item
  (lambda (ls)
    (if (null? ls)
        '()
        (cons (+ 1 (car ls)) (add1-to-each-item (cdr ls))))))
```

Now if we want to add 2 to each element, we have to write the definition again but with `(+ 1 (car ls))` replaced by `(+ 2 (car ls))`. Since we may want to perform many different operations on the elements of the list, it would be more efficient if we had a procedure that takes as arguments both the procedure we wish to apply to each element and the list. There is a Scheme procedure `map` that has the parameters `proc` and `ls` and returns a list that contains those elements that are obtained when the procedure `proc` of one argument is applied to each element of `ls`. Thus

```
(map add1 '(1 3 5 7 9)) ⟹ (2 4 6 8 10)
```

A definition of `map` is given in Program 7.1. To add 2 to each element in the list, we pass the procedure of one argument, `(lambda (num) (+ num 2))`, as the first argument to `map`. Thus we have

```
(map (lambda (num) (+ num 2)) '(1 3 5 7 9)) ⟹ (3 5 7 9 11)
```

We can also apply `map` with a procedure that operates on lists as its first argument. For example:

```
(let ((proc (lambda (ls) (cons 'a ls))))
  (map proc '((b c) (d e) (f g h)))) ⟹ ((a b c) (a d e) (a f g h))
```

Program 7.2 `for-each`

```
(define for-each
  (lambda (proc ls)
    (if (not (null? ls))
        (begin
          (proc (car ls))
          (for-each proc (cdr ls))))))
```

```
(let ((x 'a))
  (let ((proc (lambda (ls) (member? x ls))))
    (map proc '((a b c) (b c d) (c d a)))) ⟹ (#t #f #t)
```

Observe that the elements of the list making up the second argument to **map** must be of the correct type for the procedure that is applied to them. In the first of these two examples, **proc** is a procedure that takes a list as its argument and conses the symbol a onto the list. Thus each element of the second argument to **map** is a list, and the list that is returned consists of sublists, each of which begins with the a that was consed onto it.

There are procedures, such as **display**, that produce side effects of interest to us rather than their returned values. If we apply such a procedure to each item in a list, the list that is returned is not what interests us but only the side effects produced by the procedure. In such cases, we use the Scheme procedure **for-each** instead of **map** to apply the side-effecting procedure to the elements of the list. When **for-each** is applied with a side-effecting procedure as its first argument and a list as its second argument, the procedure is applied to each item in the list, the desired side effects are produced, and the value that is returned is unspecified, that is, it depends upon the implementation of Scheme being used. A definition of **for-each** is given in Program 7.2. An example using **for-each** is:

```
[1] (for-each display '("Hello." " " "How are you?"))
Hello. How are you?
```

We shall see several more examples of the use of **for-each** below. But first we introduce the form of **lambda** that is used to define a procedure that takes an arbitrary number of arguments. We use this *unrestricted lambda* to define the procedures **writeln** and **error**, which we have been using.

In a lambda expression, the keyword **lambda** is followed by a list of parameters. Its syntax is

(lambda ($parameter_1$ ...) $expr_1$ $expr_2$ ...)

where zero or more parameters are in the list of parameters and where the number of arguments passed to the procedure, which is the value of this lambda expression, must match the number of parameters. The body of the lambda expression consists of one or more expressions, which are evaluated in order and the value of the last one is returned. Suppose we want to define a procedure **add** that can be applied to arbitrarily many numbers and returns their sum. For example, we would like to have

```
(add 1 3 5 7 9) ⟹ 25
(add 1 3 5 7 9 11) ⟹ 36
(add 1 3 5 7 9 11 13) ⟹ 49
```

It is possible to define a procedure that can be applied to any number of arguments using the unrestricted **lambda**, whose syntax is

(lambda *var* $expr_1$ $expr_2$ ...)

and it may be applied to any number of operands by invoking

((lambda *var* $expr_1$ $expr_2$ ...) $operand_1$ ...)

If the operands $operand_1$... have the values arg_1..., then the variable *var* is bound to the *list* of arguments (arg_1 ...). The expressions $expr_1$ $expr_2$... in the body are evaluated with this binding in effect.

Program 7.3 add

```
(define add
  (letrec ((list-add
            (lambda (ls)
              (if (null? ls)
                  0
                  (+ (car ls) (list-add (cdr ls)))))))
    (lambda args
      (list-add args))))
```

As an example, Program 7.3 shows the definition of a procedure **add** that produces the sum of its arguments. For example, (**add 1 2 3 4 5**) ⟹ **15**.

Program 7.4 list

```
(define list (lambda args args))
```

Program 7.5 writeln

```
(define writeln
  (lambda args
    (for-each display args)
    (newline)))
```

Program 7.6 error

```
(define error
  (lambda args
    (display "Error:")
    (for-each (lambda (value) (display " ") (display value)) args)
    (newline)
    (reset)))
```

The general strategy for using this form of **lambda** is to remember that **args** is a list, so we define a local procedure **list-add** that takes a list as its argument and let it do what we want **add** to do. Then we call **list-add** with the list **args** as its argument.

Similarly, the procedure **list** is defined in Program 7.4 so that

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

Two procedures, **writeln** and **error**, can also be defined using the unrestricted lambda. These are shown in Programs 7.5 and 7.6. The procedure of no arguments **reset** in Program 7.6 returns the user to the prompt. Many implementations of Scheme provide the procedure **reset**. A discussion of the concepts used to define **reset** is given in Chapter 16. (See Exercise 16.6.)

Suppose we now want to find the maximum of two numbers in a list **ls**. We cannot invoke **(max ls)**, since the list **ls** is not the correct data type for an argument to **max**, which expects each of its arguments to be a number. If **ls** were, for example, **(2 4)**, we would be looking at the expression **(max '(2 4))**, which has the wrong type of argument for **max**. We could write a recursive program that would compute the maximum of the values in **ls**.

Program 7.7 add

```
(define add
  (lambda args
    (if (null? args)
        0
        (+ (car args) (apply add (cdr args))))))
```

However, there is the Scheme procedure, **apply**, that allows us to apply a procedure of k arguments to a list of k items, and the results are the same as if the items in the list were passed as the k arguments. The procedure **apply** has the call structure

(apply *proc list-of-items*)

where the procedure *proc* takes the same number of arguments as the number of items in the list *list-of-items*. It returns the value obtained when we invoke *proc* with the items in *list-of-items* as its arguments. For example, we can call

```
(apply max '(2 4)) ⟹ 4
(apply + '(4 11)) ⟹ 15
```

The use of **apply** gives us another way to define procedures using the unrestricted **lambda**. Program 7.7 illustrates it by redefining the procedure **add** given in Program 7.3, this time using **apply** in the recursive invocation of **add** on the list **(cdr args)**. There **add** is defined to apply to an arbitrary number of numbers, so it cannot be applied directly to **(cdr args)**, which is a list of numbers. Thus we use **apply** to invoke **add** on the items in the list **(cdr args)**.

The Scheme procedures + and * are also defined to take an arbitrary number of arguments. Thus we have:

```
(+ 1 3 5 7 9) ⟹ 25
(+ 5) ⟹ 5
(+) ⟹ 0
(* 2 4 6) ⟹ 48
(* 5) ⟹ 5
(*) ⟹ 1
```

Similarly, the Scheme procedures `max` and `min` are defined to take one or more arguments. Thus we have

```
(max 5 -10 15 -20) ⟹ 15
(min 5 -10 15 -20) ⟹ -20
```

An object in Scheme is said to be a *first-class object* if it can be passed as an argument to procedures, can be returned by procedures, and variables may be bound to it. We have been using data objects such as numbers, symbols, or lists of numbers or symbols as arguments to procedures and as values of procedures, and we have bound them to variables using `define`, `lambda`, `let` and `letrec`. Procedures are also treated as first-class objects in Scheme. This is not the case in many other programming languages. We now explore further the implications of procedures as first-class objects.

To discuss the composition of two procedures, we first look at the composition of two functions from a mathematical point of view. Assume that f and g are functions that take one argument and that each value of the function g is a valid argument of the function f. We can then speak of the composition h of the two functions f and g to be the function of one argument defined by $h(x) = f(g(x))$; that is, to get the value of h at x, we first evaluate g at x, and then invoke f on the value $g(x)$. This idea can be interpreted for the procedures we use in our programs. We now define a procedure `compose` that takes two procedures `f` and `g` as parameters and returns another procedure that is the composition of `f` and `g`.

Program 7.8 `compose`

```
(define compose
  (lambda (f g)
    (lambda (x)
      (f (g x)))))
```

The body of the first lambda expression constructs the procedure

```
(lambda (x)
  (f (g x)))
```

with one parameter `x`. Thus `(compose f g)` is a procedure of one argument, and we invoke this new procedure on `8` by writing `((compose f g) 8)`. As

an example, let us take `add1` for `g` and `sqrt` for `f`. Then we can define the composition `h` by writing

```
(define h (compose sqrt add1))
```

The new procedure `h` is the procedure that adds 1 to `x` and then takes the square root of the result; expressed mathematically, $h(x) = \sqrt{x+1}$. If we invoke `h` with argument 8, we get `(h 8)` $\Longrightarrow$ 3. Observe that we have passed the procedures `sqrt` and `add1` as arguments to the procedure `compose`. Furthermore, the value of the procedure `compose` is itself a procedure of one argument. This illustrates both the fact that we can pass procedures, such as `sqrt` and `add1`, as arguments to a procedure and we can have the value of a procedure be a procedure.

If we reverse the order of the two procedures `add1` and `sqrt` as arguments of `compose` in our previous example, we get the procedure

```
(define k (compose add1 sqrt))
```

The procedure `k` so defined first takes the square root of its argument and then adds one to the result; that is, $k(x) = \sqrt{x}+1$. Thus k is quite a different function from h.

Exercise

Exercise 7.1
What operand do we pass to `k` to get the same value as `(h 8)`?

We next develop several basic arithmetic procedures that lead to an interesting example that illustrates the use of procedures as values. The procedure `plus` may be defined in terms of `add1` and `sub1` by making use of the fact that to add two nonnegative integers `x` and `y`, we can add 1 to `x` repeatedly `y` times. This leads to Program 7.9. Similarly, using the fact that multiplication of positive integers can be considered as repeated addition, `times` can be defined in terms of `plus` and `sub1` as shown in Program 7.10. This says that multiplication of positive integers `x` and `y` is the same as adding `x` to itself `y` times. In the same way, we can consider raising `x` to the exponent `y` as multiplying `x` by itself `y` times, so we can write the procedure `exponent` as shown in Program 7.11.

Program 7.9 plus

```
(define plus
  (lambda (x y)
    (if (zero? y)
        x
        (add1 (plus x (sub1 y))))))
```

Program 7.10 times

```
(define times
  (lambda (x y)
    (if (zero? y)
        0
        (plus x (times x (sub1 y))))))
```

Program 7.11 exponent

```
(define exponent
  (lambda (x y)
    (if (zero? y)
        1
        (times x (exponent x (sub1 y))))))
```

Program 7.12 super

```
(define super
  (lambda (x y)
    (if (zero? y)
        1
        (exponent x (super x (sub1 y))))))
```

The three procedures we have defined follow a simple pattern. Using this pattern, we can define another procedure, which we call **super**, that uses **exponent** and **sub1**, as shown in Program 7.12. What does **super** do? Let us evaluate **(super 2 3)**.

Program 7.13 superduper

```
(define superduper
  (lambda (x y)
    (if (zero? y)
        1
        (super x (superduper x (sub1 y))))))
```

Program 7.14 super-order

```
(define super-order
  (lambda (n)
    (cond
      ((= n 1) plus)
      ((= n 2) times)
      (else (lambda (x y)
              (cond
                ((zero? y) 1)
                (else ((super-order (sub1 n))
                       x
                       ((super-order n) x (sub1 y))))))))))
```

```
(super 2 3) ⟹ (exponent 2 (super 2 2))
            ⟹ (exponent 2 (exponent 2 (super 2 1)))
            ⟹ (exponent 2 (exponent 2 (exponent 2 (super 2 0))))
            ⟹ (exponent 2 (exponent 2 (exponent 2 1)))
            ⟹ (exponent 2 (exponent 2 2))
            ⟹ (exponent 2 4)
            ⟹ 16
```

Thus (super 2 3) is 2^{2^2}. In the same way we get that (super 2 4) is $2^{2^{2^2}}$ (a tower of 4 twos), which is 65,536. We see that super yields large numbers even with relatively small arguments like 2 and 4.

We now go to the next step and define superduper using super and sub1, as shown in Program 7.13. Then (superduper 2 3) is (super 2 4) or 65,536, and (superduper 2 4) is (super 2 65536), which is a tower of 65,536 twos. This is a very large number.

We can continue defining successive procedures by this process, but we must make up a new name for each one. It would be better to define a procedure

super-order that depends upon a number n, so that (super-order 1) is the same procedure as plus, (super-order 2) is the same procedure as times, (super-order 3) is the same procedure as exponent, and so forth. The definition of super-order is given in Program 7.14. If n is 1, super-order is the same as plus, and if n is 2, then super-order is the same as times. For each value of n, (super-order n) is a procedure of two arguments; for example, ((super-order 4) 2 3) is the same as (super 2 3) or simply 16. We can now write any procedure in the sequence by selecting the appropriate value for the parameter n in (super-order n). For example, the procedure that comes after superduper is (super-order 6).

If all three of the arguments, n, x, and y, in super-order are the same, it is called the Ackermann procedure. Specifically, we can define

Program 7.15 ackermann

```
(define ackermann
  (lambda (n)
    ((super-order n) n n)))
```

Then

```
(ackermann 1) is the same as (plus 1 1)      which is 2.
(ackermann 2) is the same as (times 2 2)     which is 4.
(ackermann 3) is the same as (exponent 3 3) which is 27.
(ackermann 4) is the same as (super 4 4)     which is 4^(4^4)
```

To get an estimate of how large (ackermann 4) is, we first note that 4^4 is 256. To estimate $4^{4^4} = 4^{256}$, we set $x = 4^{256}$ and take the logarithm to get $\log_{10} x = 256 \log_{10} 4 = 154.13$. Thus we get $4^{256} \approx 10^{154}$ as our estimate for 4^{4^4}. Finally we estimate

$$4^{4^{4^4}}$$

similarly. If we set $y =$ (ackermann 4), then $\log_{10} y \approx 10^{154} \log_{10} 4 \approx 10^{154} 0.602$. Then $y \approx 10^{10^{153}}$, which means that (ackermann 4) has approximately 10^{153} digits. Can you estimate the magnitude of (ackermann 5)? The Ackermann procedure played an important role as an example in the general theory of recursive functions. (See, for example, Minsky, 1967.) It certainly does grow fast as n increases.

We see in the definition of `super-order` that we have a procedure with parameter `n` whose value is itself a procedure with parameters `x` and `y`, illustrating again how procedures are first-class objects in Scheme. We shall explore these ideas further in the next section, which deals with procedural abstraction.

Exercises

Exercise 7.2: `compose3`
Use the procedure `compose` to define a procedure `compose3` that takes as arguments three procedures, f, g, and h, and returns the composition k such that for each argument x, $k(x) = f(g(h(x)))$.

Exercise 7.3: `compose-many`
Use the unrestricted `lambda` to define a composition procedure `compose-many` that forms the composition of arbitrarily many procedures of one argument. Test your procedure on

```
((compose-many add1 add1 add1 add1) 3) ⟹ 7
((compose-many sqrt abs sub1 (lambda (n) (* n n))) 0.6) ⟹ 0.8
(let ((f (lambda (n) (if (even? n) (/ n 2) (add1 n)))))
  ((compose-many f f f f f f) 21))                    ⟹ 4
```

Exercise 7.4: `subtract`
Based on the technique used in this chapter to define `plus`, `times`, etc., define the procedure `subtract` that has as parameters two nonnegative integers `x` and `y`, with `x` $\geq$ `y`, and returns the difference between `x` and `y`.

Exercise 7.5
In the following experiment, fill the blanks with the values of the expressions.

```
[1] (let ((h (lambda (x) (cons x x))))
      (map h '((1 2) (3 4) (5 6))))
?__________
[2] (map (lambda (x) (cons x x)) '((1 2) (3 4) (5 6)))
?__________
[3] (map (lambda (x) (+ 5 x)) '(1 2 3 4))
?__________
[4] (let ((n 5))
      (let ((proc (lambda (x) (+ n x))))
        (map proc '(1 2 3 4))))
?__________
```

```
[5] (define iota
      (lambda (n)
        (letrec ((iota-helper
                   (lambda (k acc)
                     (cond
                       ((zero? k) (cons 0 acc))
                       (else (iota-helper (sub1 k) (cons k acc)))))))
          (iota-helper (sub1 n) '()))))
[6] (letrec ((fact
               (lambda (n)
                 (if (zero? n) 1 (* n (fact (sub1 n)))))))
      (map fact (iota 6)))
?__________
[7] (map (lambda (x) (+ x (add1 x))) (iota 5))
?__________
[8] (define mystery
      (lambda (len base)
        (letrec
          ((mystery-help
             (lambda (n s)
               (if (zero? n)
                   (list s)
                   (let ((h (lambda (x)
                              (mystery-help (sub1 n) (cons x s)))))
                     (apply append (map h (iota base))))))))
          (mystery-help len '()))))
[9] (mystery 4 3)
?__________
```

Exercise 7.6: `map-first-two`

Define a procedure, `map-first-two`, that works exactly like `map` except that the procedure argument is always a procedure of two arguments instead of just one argument. Use the first and second elements of the list as the first pair of arguments to the procedure, then the second and third elements, then the third and fourth elements, and so on, until the end of the list is reached. If there are fewer than two elements in the list, the empty list is the value. Test your procedure on:

```
(map-first-two + '(2 3 4 5 7)) ⟹ (5 7 9 12)
(map-first-two max '(2 4 3 5 4 1)) ⟹ (4 4 5 5 4)
```

Exercise 7.7: `reduce`

Define a procedure, `reduce`, that has two parameters, `proc` and `ls`. The procedure `proc` takes two arguments. The procedure `reduce` reduces the list

ls by successively applying this operation: it builds a new list with the first two elements of the preceding list replaced by the value obtained when proc is applied to them. When the list is reduced to containing only two elements, the value returned is the value of proc applied to these two elements. If the original list ls contains fewer than two elements, an error is reported. Here is how the successive stages in the reduction look when proc is + and ls is (3 5 7 9):

```
(3 5 7 9) → (8 7 9) → (15 9) → 24
```

Test your procedure on:

```
(reduce + '(1 3 5 7 9)) ⟹ 25
(reduce max '(2 -4 6 8 3 1)) ⟹ 8
(reduce (lambda (x y) (and x y)) '(#t #t #t #t)) ⟹ #t
```

The last example is not written as (reduce and '(#t #t #t #t)) because and is a keyword of a special form and not a procedure. Keywords only appear in the first position of a list.

Exercise 7.8: andmap

Define a predicate andmap that takes two arguments, a one-argument predicate pred and a list ls. The value returned by andmap is true when pred applied to each of the elements of ls is true. If pred applied to any one of the elements of ls is false, andmap returns false. The solution

```
(define andmap
  (lambda (pred ls)
    (reduce (lambda (x y) (and x y)) (map pred ls))))
```

is unacceptable because of the extra recursion. Test your predicate on:

```
(andmap positive? '(3 4 6 9)) ⟹ #t
(andmap positive? '(3 -1 4 8)) ⟹ #f
(let ((not-null? (compose not null?)))
  (andmap not-null? '((a b) (c) (c d e)))) ⟹ #t
```

Exercise 7.9: map2

Define map2, which is exactly like map except that its procedure argument is always a procedure that takes two arguments, and it takes an additional argument that is a list the same length as its second argument. The additional list is where it gets its second argument. Test your procedure on:

```
(map2 + '(1 2 3 4) '(5 7 9 11)) ⟹ (6 9 12 15)
```

```
(map2 (let ((n 5))
        (lambda (x y)
          (and (< x n) (< n y))))
      '(1 3 2 1 7)
      '(9 11 4 7 8)) ⟹ (#t #t #f #t #f)
```

Exercise 7.10: `map`, `ormap`

We now present a definition of `map` that accepts any number of arguments.

(`map` *proc* ls_1 ls_2 ... ls_n)

where *proc* is a procedure that takes n arguments and each of the n lists has the same length. This generalizes the procedures `map` and `map2` given above.

```
(define map
  (lambda args
    (let ((proc (car args)))
      (letrec ((map-helper
                 (lambda (a*)
                   (if (any-null? a*)
                       '()
                       (cons
                         (apply proc (map car a*))
                         (map-helper (map cdr a*)))))))
        (map-helper (cdr args))))))
```

This program, as written, is incorrect because the two invocations of *map* within the definition refer to the simple `map` we defined earlier in the chapter. Add a definition of the simple `map` to the `letrec` (in the same way that `even?` and `odd?` are in the same `letrec`) so that no names will be changed in the definition of `map-helper`, and write `any-null?` using the definition of `ormap` given below.

```
(define ormap
  (lambda (pred ls)
    (if (null? ls)
        #f
        (or (pred (car ls)) (ormap pred (cdr ls))))))
```

What does this version of `map` return when the n lists are not of equal length?

Exercise 7.11

To test your understanding of scope, determine the value of the expression

```
(letrec ((a (let ((a (lambda (b c)
                       (if (zero? b) c (a (sub1 b))))))
              (lambda (x) (a x x)))))
  (a 3))
```

7.3 Currying

The procedure + takes two numbers as arguments and returns their sum. The procedure **add1** adds 1 to its argument. We can also define a procedure **add5** that adds 5 to its argument by writing

```
(define add5
  (lambda (n)
    (+ 5 n)))
```

This can clearly be done for any number in place of 5. Another way of approaching this problem makes use of the fact that a procedure may return another procedure as its value. We can define a procedure **curried+** that has only one parameter, **m**, and returns a procedure having one parameter **n**, that adds **m** and **n**:

```
(define curried+
  (lambda (m)
    (lambda (n)
      (+ m n))))
```

Thus `(curried+ 5)` returns a procedure defined by

```
(lambda (n) (+ m n))
```

where **m** is bound to 5. To add 5 and 7, we would then invoke

```
((curried+ 5) 7) ⟹ 12
```

We can now define **add5** by writing

```
(define add5 (curried+ 5))
```

Moreover, we can define **add8** by writing

```
(define add8 (curried+ 8))
```

and we clearly can do the same for any other number in place of **8**. What underlies this method is the fact that we can take any procedure that has two parameters, say **x** and **y**, and rewrite it as a procedure with one parameter **x** that returns a procedure with one parameter **y**. The process of writing a procedure of two parameters as a procedure of one parameter that returns a procedure of one parameter is called *currying* the procedure.[1] It is often advantageous to use a curried procedure when you want to keep one argument fixed while the other varies, so in essence, you are using a procedure of one argument.

We next use currying to rewrite the definitions of four procedures in a way that demonstrates certain common structural features that they possess. In the next section, we shall abstract these common features and write a single procedure from which the original four and many others can be obtained. The four procedures are **member?**, **map**, **sum**, and **product**.

The procedure **member?** can be defined as follows:

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

It tests whether the object **item** is a top-level object in the list **ls**. We are going to apply the procedure **member?** with the same object **item** but different lists **ls**, so we define the curried procedure **member?-c**, which is a procedure with parameter **item** and returns a procedure that has the parameter **ls** and tests whether **item** is a top-level member of **ls**. We do that in Program 7.16. Observe the following points in the definition of **member?-c**:

1. **member?-c** is a procedure with one parameter **item**.
2. The procedure **member?-c** returns a procedure **helper** that has one parameter **ls**.

---

[1] Conceived by Moses Schönfinkel in 1924 (See Schönfinkel, 1924) and named after the logician Haskell B. Curry.

Program 7.16 member?-c

```
(define member?-c
  (lambda (item)
    (letrec
      ((helper
         (lambda (ls)
           (if (null? ls)
               #f
               (or (equal? (car ls) item) (helper (cdr ls)))))))
       helper)))
```

3. We introduced the letrec expression to avoid having to pass the argument `item` each time we make a recursive procedure call, since `item` does not change throughout the program.

We can now define the original procedure `member?` in terms of `member?-c` by writing

```
(define member?
  (lambda (a ls)
    ((member?-c a) ls)))
```

As another example of currying, we look at the definition of the procedure `map`, presented in Program 7.1, which has two parameters, a procedure `proc`, and a list `ls`. It applies the procedure `proc` elementwise to `ls` and returns a list of the results. For example,

```
(map add1 '(1 2 3 4)) ⟹ (2 3 4 5)
```

Its definition is:

```
(define map
  (lambda (proc ls)
    (if (null? ls)
        '()
        (cons (proc (car ls)) (map proc (cdr ls))))))
```

This can be written in curried form by using the procedure `apply-to-all`, which takes one argument `proc` and is itself a procedure of the argument `ls`. We give its definition in Program 7.17. We can write `map` in terms of `apply-to-all` by defining

Program 7.17 `apply-to-all`

```
(define apply-to-all
  (lambda (proc)
    (letrec
      ((helper
         (lambda (ls)
           (if (null? ls)
               '()
               (cons (proc (car ls)) (helper (cdr ls)))))))
       helper)))
```

Program 7.18 `sum`

```
(define sum
  (letrec
    ((helper
       (lambda (ls)
         (if (null? ls)
             0
             (+ (car ls) (helper (cdr ls)))))))
     helper))
```

Program 7.19 `product`

```
(define product
  (letrec
    ((helper
       (lambda (ls)
         (if (null? ls)
             1
             (* (car ls) (helper (cdr ls)))))))
     helper))
```

```
(define map
  (lambda (proc ls)
    ((apply-to-all proc) ls)))
```

We next look at two more procedures that take lists as arguments. The first,

Program 7.20 `swapper-m`

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

sum, assumes that the objects in the list are numbers and returns the sum of the numbers in the list, and the second, **product**, assumes that the objects in the list are numbers and returns the product of the numbers in the list. We write their definitions in Programs 7.18 and 7.19 in such a way that they demonstrate the same structure as the preceding definitions of **member?-c** and **apply-to-all**. We could have written the procedures **sum** and **product** without the letrec expressions, but we have chosen to do it this way to be able to compare the structure of these two procedures with the structure of **member?-c** and **apply-to-all** when we abstract this structure in the next section.

We close this section with an example that is similar to currying, this time modifying a procedure with three parameters to get a procedure with two parameters that returns a procedure with one parameter. We look at the procedure **swapper** introduced in Program 2.8. Its definition is:

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

We modify it to get a procedure **swapper-m** (we use **-m** for "modified") that has the two parameters **x** and **y** and that returns a procedure of one parameter **ls**. Its definition is given in Program 7.20. To swap the numbers 0 and 1 in the list `(0 1 2 0 1 2)`, we would invoke

```
((swapper-m 0 1) '(0 1 2 0 1 2)) ⟹ (1 0 2 1 0 2)
```

This example illustrates that a generalization of currying can be used to redefine a procedure with $n = m + k$ parameters to become a procedure with m parameters that returns a procedure with k parameters. The term currying refers to redefining a procedure with n parameters to be expressed as n procedures, each having only one parameter.

In this section, we have introduced the concept of currying a procedure of two arguments to get a procedure of one argument that returns a procedure of one argument. This technique is useful when we want to consider the behavior of the procedure as the second argument varies while the first argument is fixed. More generally, a procedure of $n = m + k$ arguments may be modified to get a procedure of m arguments that returns a procedure of k arguments.

Exercises

Exercise 7.12: **curried***

Curry the procedure * to get a procedure **curried*** and use it to define the procedure **times10** that multiplies its argument by 10. Test your procedures on:

```
((curried* 25) 5) ⟹ 125
(times10 125) ⟹ 1250
```

Exercise 7.13: **swapper-c**

Curry the procedure **swapper-m** so that the curried procedure **swapper-c** has one parameter **x**. It returns a procedure with one parameter **y**, which in turn returns a procedure with one parameter **ls**. That procedure swaps **x** and **y** in **ls**.

Exercise 7.14: **round-n-places**

In Program 6.5, the procedure **round-n-places** was defined to take two parameters, **n** and **dec-num**, and returned the number **dec-num** rounded off to **n** decimal places. Rewrite the definition of **round-n-places** so that it takes one parameter, **n**, and returns a procedure with one parameter, **dec-num**, that rounds the number **dec-num** off to **n** decimal places. We can then write

```
(define round-5-places (round-n-places 5))
```

to get the procedure that rounds a given number off to five decimal places.

Exercise 7.15: `subst-all-m`

Modify the deeply recursive procedure `subst-all`, which has the parameters `new`, `old`, and `ls`, to get a procedure `subst-all-m` with the two parameters `new` and `old`, which returns a procedure with the parameter `ls`, which replaces each occurrence of `old` in `ls` by `new`. Test your procedure on:

```
((subst-all-m 1 0) '(0 1 2 0 1 2)) ⟹ (1 1 2 1 1 2)
((subst-all-m 1 0) '(0 1 2 ((0 1 2)))) ⟹ (1 1 2 ((1 1 2)))
```

Exercise 7.16: `extreme-value-c`

In Program 3.19, the procedure `extreme-value` was defined and then it was used to define the procedures `rmax` and `rmin` by passing it the appropriate predicate. Write the definition of the procedure `extreme-value-c`, which takes the predicate `pred` and returns a procedure that finds the maximum of its two arguments or the minimum of its two arguments, depending upon `pred`. Then express `rmax` and `rmin` in terms of `extreme-value-c`.

Exercise 7.17: `extreme-value-c`

In the previous exercise, the procedure `(extreme-value-c pred)` expects only two arguments. Rewrite the definition of `extreme-value-c` using the unrestricted lambda so that `(extreme-value-c pred)` is a procedure that takes arbitrarily many numbers as arguments and returns the extreme value (maximum or minimum) depending upon the predicate `pred`.

Exercise 7.18: `between?`, `between?-c`

Define a predicate `between?` that has three numbers `x`, `y`, and `z`, as parameters and returns true when `y` is strictly between `x` and `z`, that is, when `x < y < z`. Then define `between?-c`, a curried version of `between?`, where each of the procedures has only one parameter. That is, `between?-c` has the parameter `x` and returns a procedure that has the parameter `y`, which in turn returns a procedure with the parameter `z`, that tests whether `y` is strictly between `x` and `z`. Test your procedure on:

```
(((between?-c 5) 6) 7) ⟹ #t
(((between?-c 5) 5) 7) ⟹ #f
(((between?-c 5) 4) 7) ⟹ #f
```

Exercise 7.19: `andmap-c`, `ormap-c`

Consider this definition of `andmap-c`:

```
(define andmap-c
  (lambda (pred)
    (letrec
      ((and-help
         (lambda (ls)
           (cond
             ((null? ls) #t)
             (else (and (pred (car ls)) (and-help (cdr ls))))))))
       and-help)))
```

Fill in the blanks below.

```
[1] (define all-positive? (andmap-c positive?))
[2] (all-positive? '(3 4 8 9))
?__________
[3] (all-positive? '(3 -1 4 8))
?__________
[4] ((andmap-c (compose not null?)) '((a b) (c) (c d e)))
?__________
```

Now define the procedure `ormap-c`, which takes a predicate as an argument and returns a predicate that accepts a list as a value. We can define `ormap` (see Exercise 7.10) using `ormap-c` as follows:

```
(define ormap
  (lambda (pred ls)
    ((ormap-c pred) ls)))
```

Test `ormap-c` by filling in the blanks below.

```
[5] (define some-positive? (ormap-c positive?))
[6] (some-positive? '(3 4 8 9))
?__________
[7] (some-positive? '(3 -1 4 8))
?__________
[8] ((ormap-c (compose not null?)) '(() () (a b) (c) (c d e)))
?__________
```

Exercise 7.20: `is-divisible-by?`, `prime?`
Consider the definition

```
(define is-divisible-by?
  (lambda (n)
    (lambda (k)
      (zero? (remainder n k)))))
```

A *prime number* is a positive integer greater than 1 that is not divisible by any positive number other than 1 and itself. Using `is-divisible-by?`, write a definition of the procedure `prime?` that tests whether a positive integer `n` > 2 is prime by first testing whether it is odd and greater than 1 and then testing whether it is not divisible by any of the odd integers from 3 to the largest odd integer less than or equal to the square root of `n`. Why is it necessary only to try integers less than the square root of `n`?

Exercise 7.21
Justify the statement "If we restrict ourselves to using only lambda expressions having only one parameter in its list of parameters, we can still define any procedure, regardless of how many parameters it has." Note that the currying examples in this section show how to define procedures having two and three parameters using only lambda expressions with one parameter.

7.4 Procedural Abstraction of Flat Recursion

In this section, we show how to abstract the structure of flatly recursive procedures to obtain a general procedure in terms of which the various special cases can be defined. We illustrate this idea by looking for common structural features in the four procedures `member?-c`, `apply-to-all`, `sum`, and `product` defined in Section 7.3. A comparison of the code for these four procedures yields the fact that the four lines

```
(letrec
  ((helper
     (lambda (ls)
       (if (null? ls)
```

and the last line

```
helper
```

are identical in all four programs. Furthermore, in all four, we do something to `(car ls)` and make the recursive call to `helper` on `(cdr ls)`. We want to define a procedure `flat-recur` that abstracts the structure of these four programs; that is, it embodies the common features of these programs, and they can all be derived from it by using suitable parameters. Let us see how much of `flat-recur` we can write based on the above observations.

```
(define flat-recur
  (lambda ( _____________ )
    (letrec
      ((helper
         (lambda (ls)
           (if (null? ls)
               _______
               _____________________ ))))
       helper)))
```

How do we fill in the blanks? Let us first look at the blank that is the consequent of the if expression. It is the action taken when `ls` is empty. We call this consequent of the test `(null? ls)` the *seed* and denote it by the variable `seed`. This will be the first parameter in the outer lambda expression. Table 7.21 shows the seed for each of the four cases.

Procedure	seed
member?-c	#f
apply-to-all	()
sum	0
product	1

Table 7.21 Seeds for the four procedures

The other blank in the if expression is in the action taken on `(car ls)` and `(helper (cdr ls))` when `(null? ls)` is false. The action taken on `(car ls)` and `(helper (cdr ls))` is a procedure that takes `(car ls)` and `(helper (cdr ls))` as arguments, and we call this procedure `list-proc`. We write `list-proc` as a procedure with the parameters `x` and `y`. When `list-proc` is invoked, `x` will be bound to `(car ls)` and `y` will be bound to `(helper (cdr ls))` to give us the alternative action when `(null? ls)` is false. For example, the alternative action in the case of `apply-to-all` is

```
(cons (proc (car ls)) (helper (cdr ls)))
```

If `list-proc` is the value of

```
(lambda (x y) (cons (proc x) y))
```

then

```
(list-proc (car ls) (helper (cdr ls)))
```

is the desired alternative action. We pass `list-proc` as the second parameter in the outer lambda expression. Table 7.22 gives `list-proc` for each of the four programs.[2]

Procedure	list-proc
member?-c	(lambda (x y) (or (equal? x item) y))
apply-to-all	(lambda (x y) (cons (proc x) y))
sum	+
product	*

Table 7.22 The four list procedures

We are now ready to define the procedure `flat-recur`, which takes `seed` and `list-proc` as arguments and produces precisely the procedure with parameter `ls` that abstracts the structure of the four procedures. (See Program 7.23.) We can then write each of the four procedures in terms of this new procedure. Furthermore, we can use it to write any procedure using recursion on a list of top-level items.

We can now write the four procedures using `flat-recur` as follows:

```
(define member?-c
  (lambda (item)
    (flat-recur #f (lambda (x y) (or (equal? x item) y)))))

(define apply-to-all
  (lambda (proc)
    (flat-recur '() (lambda (x y) (cons (proc x) y)))))

(define sum (flat-recur 0 +))

(define product (flat-recur 1 *))
```

[2] The procedure that we selected for `list-proc` in the case of `member?-c` does more processing than is necessary, for it loses the benefit of the behavior of `or`. Generally when the first argument to `or` is true, the value `#t` is returned without evaluating the second argument. However, when `list-proc` is called, both arguments are first evaluated, and then the `or` expression is evaluated, so the argument to which `y` is bound is always evaluated. Even though the resulting version of `member?-c` is less efficient, it illustrates the principle of procedural abstraction and a feature that one should be aware of when applying it.

Program 7.23 `flat-recur`

```
(define flat-recur
  (lambda (seed list-proc)
    (letrec
      ((helper
         (lambda (ls)
           (if (null? ls)
               seed
               (list-proc (car ls) (helper (cdr ls)))))))
       helper)))
```

You may be concerned that the procedure **list-proc** in these last two examples has a different structure from those in the first two examples. This is really not the case, since we could also have used **(lambda (x y) (+ x y))** in place of the variable **+**, and we could have used **(lambda (x y) (* x y))** in place of the variable *****.

The process we have used here looks for common features in several programs and then produces a procedure that embodies the code that is similar in all of these programs. It enables us to express each of the original procedures more simply. This process is called *procedural abstraction*. This is a very powerful programming tool that should be exploited when it is applicable.

The procedure **flat-recur** can be used whenever a program does recursion on the top-level objects in a list. We now see an example of how we can use it to define the procedure **filter-in-c**. Let **ls** be a list and suppose that we have a predicate **pred** that we want to apply to each top-level object in the list. If the result of applying **pred** to an object in the list is false, then the object is to be dropped from the list. Thus the procedure **filter-in-c** returns a list consisting of those objects from **ls** that "pass" the test. This program involves recursion on the top-level objects in the list **ls**, and if **ls** is empty, **filter-in-c** returns the empty list, so **seed** is **()**. To get **list-proc** we shall again use **x** for the **(car ls)** and **y** for **(helper (cdr ls))**. Then if **pred** applied to **x** is true, we cons **x** to **y**; otherwise we just return **y**. Thus the **list-proc** of **flat-recur** can be written as

```
(lambda (x y)
  (if (pred x)
      (cons x y)
      y))
```

Program 7.24 `filter-in-c`

```
(define filter-in-c
  (lambda (pred)
    (flat-recur
      '()
      (lambda (x y)
        (if (pred x)
            (cons x y)
            y)))))
```

and we can define `filter-in-c` as shown in Program 7.24. If we do not want to use `filter-in-c` in curried form, we can define the procedure `filter-in` as:

```
(define filter-in
  (lambda (pred ls)
    ((filter-in-c pred) ls)))
```

Here are some examples using `filter-in`:

```
(filter-in odd? '(1 2 3 4 5 6 7 8 9)) ⟹ (1 3 5 7 9)
(filter-in positive? '(1 0 2 0 3 0 4)) ⟹ (1 2 3 4)
(filter-in (lambda (x) (< x 5)) '(1 2 3 4 5 6 7 8 9)) ⟹ (1 2 3 4)
```

In this section, we have illustrated the process of procedural abstraction of flat recursion. We defined a procedure `flat-recur` from which procedures that use flat recursion can be derived by passing `flat-recur` the appropriate arguments. This is a powerful tool that can often be used to make programs easier to write and to understand.

Exercises

Exercise 7.22: `mult-by-scalar`
In Exercise 3.1, we called a list of numbers an n-tuple. Using `flat-recur`, define a procedure `mult-by-scalar` that takes as its argument a number `c` and returns a procedure that takes as its argument an n-tuple `ntpl` and multiplies each component of `ntpl` by the number `c`. Test your procedure on:

```
((mult-by-scalar 3) '(1 -2 3 -4)) ⟹ (3 -6 9 -12)
((mult-by-scalar 5) '()) ⟹ ()
```

Exercise 7.23: `filter-out`

The procedure `filter-out` takes two arguments, a predicate `pred` and a list `ls`. It removes from the list `ls` all of its top-level elements that "pass" the test, that is, it removes those top-level objects `item` for which `(pred item)` is true. Write the definition of `filter-out` using a local procedure `filter-out-c` that is defined using `flat-recur`.

Exercise 7.24: `insert-left`

Starting with the procedure `insert-left` described in Exercise 4.1 and using `flat-recur`, define the modified version `insert-left-m` that takes as parameters the new and old values and returns a procedure with the list as its parameter. Then define `insert-left` using `insert-left-m`.

Exercise 7.25: `partial`

Let `proc` be a procedure of one numerical argument with numerical values.

a. Define a procedure `partial-sum` that computes the sum of the numbers `(proc` i`)` for i ranging from k to n, where $k \leq n$. For example,

```
(partial-sum (lambda (m) (* m m)) 3 7) ⟹ 135
```

b. Define a procedure `partial-product` that computes the product of the numbers `(proc` i`)` for i ranging from k to n, where $k \leq n$. For example

```
(partial-product (lambda (m) (* m m)) 3 7) ⟹ 6350400
```

c. Define an abstraction of `partial-sum` and `partial-product` named `partial` so that `partial-sum` and `partial-product` can be defined as

```
(define partial-sum (partial 0 +))
(define partial-product (partial 1 *))
```

7.5 Procedural Abstraction of Deep Recursion

The deeply recursive procedures defined in Chapter 4 use recursion on nested sublists rather than being limited to top-level objects of lists. They also display a common structure that can be abstracted in a procedure `deep-recur`. We now look at some deeply recursive procedures to find their common structure and then formulate the definition of `deep-recur`.

We start with `filter-in-all-c`, which takes a `pred` as its argument and returns a procedure that has a list `ls` as its parameter and, when applied to

`ls`, drops from the list those items that do not "pass" the test. For example, if `pred` is `odd?` and `ls` is `((4 5) 2 (3 5 (8 7)))` we have

```
((filter-in-all-c odd?) ls) ⟹ ((5) (3 5 (7)))
```

The code for `filter-in-all-c` is given in Program 7.25. We define `filter-in-all` as the procedure of two arguments, `pred` and `ls`, in terms of `filter-in-all-c` as shown in Program 7.26.

In the same way, we define `sum-all` as a procedure of one argument `ls`, which is a list of numbers, such that `(sum-all ls)` is the sum of all of the numbers in `ls`. For example

```
(sum-all '(3 (1 4) (2 (-3 5)))) ⟹ 12
```

The code for `sum-all` is presented in Program 7.27. Both of these procedures, `sum-all` and `filter-in-all-c`, share the following lines:

```
(letrec
  ((helper
     (lambda (ls)
       (if (null? ls)
.....................
(let ((a (car ls)))
  (if (or (pair? a) (null? a))
..............................
helper
```

We are going to define a procedure `deep-recur` to abstract the structure of these two procedures. Let us see how much of the code we can fill in from the above observations.

```
(define deep-recur
  (lambda ( ________________ )
    (letrec
      ((helper
         (lambda (ls)
           (if (null? ls)
               ________
               (let ((a (car ls)))
                 (if (or (pair? a) (null? a))
                     ______________________
                     ______________________ ))))))
       helper)))
```

Program 7.25 `filter-in-all-c`

```
(define filter-in-all-c
  (lambda (pred)
    (letrec
      ((helper
         (lambda (ls)
           (if (null? ls)
               '()
               (let ((a (car ls)))
                 (if (or (pair? a) (null? a))
                     (cons (helper a) (helper (cdr ls)))
                     (if (pred a)
                         (cons a (helper (cdr ls)))
                         (helper (cdr ls)))))))))
       helper)))
```

Program 7.26 `filter-in-all`

```
(define filter-in-all
  (lambda (pred ls)
    ((filter-in-all-c pred) ls)))
```

Program 7.27 `sum-all`

```
(define sum-all
  (letrec ((helper
            (lambda (ls)
              (if (null? ls)
                  0
                  (let ((a (car ls)))
                    (if (or (pair? a) (null? a))
                        (+ (helper a) (helper (cdr ls)))
                        (+ a (helper (cdr ls)))))))))
    helper))
```

Once again, we use the variable **seed** to denote the consequent of the first if expression with test **(null? ls)**. In the case of **sum-all**, **seed** is **0**, and in the case of **filter-in-all-c**, **seed** is **()**. We take **seed** to be the first parameter for the outer lambda expression.

In the consequent of the second if expression with test **(or (pair? a) (null? a))**, the local procedure **helper** for **filter-in-all-c** invokes

```
(cons (helper a) (helper (cdr ls)))
```

and the local procedure **helper** for **sum-all** invokes

```
(+ (helper a) (helper (cdr ls)))
```

We refer to the procedure that is applied to **(helper a)** and **(helper (cdr ls))** as **list-proc**. We fill the blank with the application

```
(list-proc (helper a) (helper (cdr ls)))
```

and to generate the expression needed for **filter-in-all-c**, we bind **list-proc** to **cons**, and to generate the expression needed for **sum-all**, we bind **list-proc** to **+**. We take **list-proc** as the third parameter to the outer lambda expression. We next consider what to use as the second parameter.

In both of our examples, the alternative of the second if expression with test **(or (pair? a) (null? a))** is a procedure invocation that involves **a** and **(helper (cdr ls))**. For **filter-in-all-c**, we want to generate the expression

```
(if (pred a) (cons a (helper (cdr ls))) (helper (cdr ls)))
```

while for **sum-all**, we need

```
(+ a (helper (cdr ls)))
```

We can generate both of these using a procedure **item-proc** that has two parameters, **x** and **y**. If we fill the blank with

```
(item-proc a (helper (cdr ls)))
```

then to get what we need for **sum-all**, we bind **item-proc** to **+**. To get what we need for **filter-in-all-c**, we bind **item-proc** to

Program 7.28 `deep-recur`

```
(define deep-recur
  (lambda (seed item-proc list-proc)
    (letrec
      ((helper
         (lambda (ls)
           (if (null? ls)
               seed
               (let ((a (car ls)))
                 (if (or (pair? a) (null? a))
                     (list-proc (helper a) (helper (cdr ls)))
                     (item-proc a (helper (cdr ls)))))))))
       helper)))
```

```
(lambda (x y)
  (if (pred x)
      (cons x y)
      y))
```

We are now in a position to write a procedure `deep-recur` that abstracts the structure of these procedures (and those in the exercises at the end of this section). The procedure `deep-recur` has three parameters: `seed`, `item-proc`, and `list-proc`. It returns `helper`, which is a procedure with only one parameter `ls`. Combining the observations made above, we get the definition presented in Program 7.28.

In particular, we can now write

```
(define sum-all (deep-recur 0 + +))
```

and

```
(define filter-in-all-c
  (lambda (pred)
    (deep-recur
      '()
      (lambda (x y)
        (if (pred x)
            (cons x y)
            y))
      cons)))
```

In this chapter, we looked at the definitions of the four procedures **sum**, **product**, **member?-c**, and **filter-in-c**, all of which performed flat recursion, and abstracted from them their common structural features. We then defined the procedure **flat-recur**, which incorporated those common features and took as arguments the things that produced the features of the four procedures that were not common to them all. This enabled us to recover the original four procedures and others that do flat recursion from **flat-recur** by passing to **flat-recur** the appropriate arguments. We then did a similar thing with procedures that performed deep recursions. We abstracted from the two procedures **filter-in-all-c** and **sum-all** their common features and defined the procedure **deep-recur**. We were able to recover the original two procedures by passing to **deep-recur** the appropriate arguments. This process of defining a procedure incorporating the common structural features of a class of procedures, and then obtaining the procedures in that class by passing the abstraction the appropriate arguments, is what we called procedural abstraction.

Exercises

Exercise 7.26: **remove-all-c, product-all**
Write the definitions of **remove-all-c** and **product-all** for arbitrary lists. The procedure **remove-all-c** takes an object **item** as its argument and returns a procedure of the list **ls**, which removes all occurrences of **item** in **ls**. The call (**product-all ls**) returns the product of all of the numbers in the list of numbers **ls**. In both procedures, preserve the structure displayed in the above definitions of **sum-all** and **filter-in-all-c** using **letrec**.

Exercise 7.27: **remove-all-c, product-all** *(continued)*
Define the two procedures **product-all** and **remove-all-c** described in the previous exercise using **deep-recur**.

Exercise 7.28: **filter-out-all**
In a manner analogous to that used in Exercise 7.23, use **deep-recur** to define the deeply recursive procedure **filter-out-all-c**, and then use it to define **filter-out-all**.

Exercise 7.29: **subst-all-m**
The procedure **subst-all-m** was described in Exercise 7.15. Define it using **deep-recur**.

Exercise 7.30: `reverse-all`
The procedure `reverse-all` was defined in Program 4.10. Define it using `deep-recur`.

Exercise 7.31: `flat-recur`
Define `flat-recur` using `deep-recur`.

Exercise 7.32: `deep-recur`
Define `deep-recur` using `flat-recur`. Hint: Use `letrec`.
