19 Implementing Higher-Order Functions

This chapter is about writing *higher-order procedures*—that is, procedures that implement higher-order functions. We are going to study the implementation of `every`, `keep`, and so on.

Really there are no new techniques involved. You know how to write recursive procedures that follow the `every` pattern, the `keep` pattern, and so on; it's a small additional step to generalize those patterns. The truly important point made in this chapter is that you aren't limited to a fixed set of higher-order functions. If you feel a need for a new one, you can implement it.

Generalizing Patterns

In Chapter 14, we showed you the procedures `square-sent` and `pigl-sent`, which follow the `every` pattern of recursion. In order to write the general tool, `every` itself, we have to *generalize the pattern* that those two have in common.

Before we get to writing higher-order procedures, let's look at a simpler case of generalizing patterns.

Suppose we want to find out the areas of several different kinds of shapes, given one linear dimension. A straightforward way would be to do it like this:

```
(define pi 3.141592654)

(define (square-area r) (* r r))

(define (circle-area r) (* pi r r))

(define (sphere-area r) (* 4 pi r r))
```

```
(define (hexagon-area r) (* (sqrt 3) 1.5 r r))

> (square-area 6)
36

> (circle-area 5)
78.53981635
```

This works fine, but it's somewhat tedious to define all four of these procedures, given that they're so similar. Each one returns the square of its argument times some constant factor; the only difference is the constant factor.

We want to generalize the pattern that these four procedures exhibit. Each of these procedures has a particular constant factor built in to its definition. What we'd like instead is one single procedure that lets you choose a constant factor when you invoke it. This new procedure will take a second argument besides the linear dimension `r` (the radius or side): a `shape` argument whose value is the desired constant factor.

```
(define (area shape r) (* shape r r))
(define square 1)
(define circle pi)
(define sphere (* 4 pi))
(define hexagon (* (sqrt 3) 1.5))

> (area sphere 7)
615.752160184
```

What's the point? We started with several procedures. Then we found that they had certain points of similarity and certain differences. In order to write a single procedure that generalizes the points of similarity, we had to use an additional argument for each point of difference. (In this example, there was only one point of difference.)

In fact, *every* procedure with arguments is a generalization in the same way. Even `square-area`, which we presented as the special case to be generalized, is more general than these procedures:

```
(define (area-of-square-of-side-5)
  (* 5 5))

(define (area-of-square-of-side-6)
  (* 6 6))
```

These may seem too trivial to be taken seriously. Indeed, nobody would write such procedures. But it's possible to take the area of a particular size square without using a procedure at all, and then later discover that you need to deal with squares of several sizes.

This idea of using a procedure to generalize a pattern is part of the larger idea of abstraction that we've been discussing throughout the book. We notice an algorithm that we need to use repeatedly, and so we separate the algorithm from any particular data values and give it a name.

The idea of generalization may seem obvious in the example about areas of squares. But when we apply the same idea to generalizing over a function, rather than merely generalizing over a number, we gain the enormous expressive power of higher-order functions.

The `Every` Pattern Revisited

Here again is the `every` template:

```
(define (every-something sent)
  (if (empty? sent)
      '()
      (se (__________ (first sent))
          (every-something (bf sent)))))
```

You've been writing `every`-like procedures by filling in the blank with a specific function. To generalize the pattern, we'll use the trick of adding an argument, as we discussed in the last section.

```
(define (every fn sent)
  (if (empty? sent)
      '()
      (se (fn (first sent))
          (every fn (bf sent)))))
```

This is hardly any work at all for something that seemed as mysterious as `every` probably did when you first saw it.

Recall that `every` will also work if you pass it a word as its second argument. The version shown here does indeed work for words, because `first` and `butfirst` work for words. So probably "`stuff`" would be a better formal parameter than "`sent`." (The result from `every` is always a sentence, because `sentence` is used to construct the result.)

The Difference between `Map` and `Every`

Here's the definition of the `map` procedure:

```
(define (map fn lst)
  (if (null? lst)
      '()
      (cons (fn (car lst))
            (map fn (cdr lst)))))
```

The structure here is identical to that of `every`; the only difference is that we use `cons`, `car`, and `cdr` instead of `se`, `first`, and `butfirst`.

One implication of this is that you can't use `map` with a word, since it's an error to take the `car` of a word. When is it advantageous to use `map` instead of `every`? Suppose you're using `map` with a structured list, like this:

```
> (map (lambda (flavor) (se flavor '(is great)))
       '(ginger (ultra chocolate) pumpkin (rum raisin)))
((GINGER IS GREAT) (ULTRA CHOCOLATE IS GREAT)
 (PUMPKIN IS GREAT) (RUM RAISIN IS GREAT))


> (every (lambda (flavor) (se flavor '(is great)))
         '(ginger (ultra chocolate) pumpkin (rum raisin)))
(GINGER IS GREAT ULTRA CHOCOLATE IS GREAT PUMPKIN IS GREAT
 RUM RAISIN IS GREAT)
```

Why does `map` preserve the structure of the sublists while `every` doesn't? `Map` uses `cons` to combine the elements of the result, whereas `every` uses `sentence`:

```
> (cons '(pumpkin is great)
        (cons '(rum raisin is great)
              '()))
((PUMPKIN IS GREAT) (RUM RAISIN IS GREAT))

> (se '(pumpkin is great)
      (se '(rum raisin is great)
          '()))
(PUMPKIN IS GREAT RUM RAISIN IS GREAT)
```

`Filter`

Here's the implementation of `filter`:

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

Like `map`, this uses `cons` as the constructor so that it will work properly on structured lists. We're leaving the definition of `keep`, the version for words and sentences, as an exercise.

(Aside from the difference between lists and sentences, this is just like the `keep` template on page 224.)

`Accumulate` and `Reduce`

Here are the examples of the `accumulate` pattern that we showed you before:

```
(define (addup nums)
  (if (empty? nums)
      0
      (+ (first nums) (addup (bf nums)))))

(define (scrunch-words sent)
  (if (empty? sent)
      ""
      (word (first sent) (scrunch-words (bf sent)))))
```

What are the similarities and differences? There are *two* important differences between these procedures: the combiners (`+` versus `word`) and the values returned in the base cases (zero versus the empty word). According to what we said about generalizing patterns, you might expect that we'd need two extra arguments. You'd invoke `three-arg-accumulate` like this:

```
> (three-arg-accumulate + 0 '(6 7 8))
21

> (three-arg-accumulate word "" '(come together))
COMETOGETHER
```

But we've actually defined accumulate and reduce so that only two arguments are required, the procedure and the sentence or list. We thought it would be too much trouble to have to provide the identity element all the time. How did we manage to avoid it?

The trick is that in our reduce and accumulate the base case is a one-element argument, rather than an empty argument. When we're down to one element in the argument, we just return that element:

```
(define (accumulate combiner stuff)          ;; first version
  (if (empty? (bf stuff))
      (first stuff)
      (combiner (first stuff)
                (accumulate combiner (bf stuff)))))
```

This version is a simplification of the one we actually provide. What happens if stuff is empty? This version blows up, since it tries to take the butfirst of stuff immediately. Our final version has a specific check for empty arguments:

```
(define (accumulate combiner stuff)
  (cond ((not (empty? stuff)) (real-accumulate combiner stuff))
        ((member combiner (list + * word se append))
         (combiner))
        (else (error
               "Can't accumulate empty input with that combiner"))))

(define (real-accumulate combiner stuff)
  (if (empty? (bf stuff))
      (first stuff)
      (combiner (first stuff) (real-accumulate combiner (bf stuff)))))
```

This version works just like the earlier version as long as stuff isn't empty. (Reduce is the same, except that it uses null?, car, and cdr.)

As we mentioned in Chapter 8, many of Scheme's primitive procedures return their identity element when invoked with no arguments. We can take advantage of this; if accumulate is invoked with an empty second argument and one of the procedures +, *, word, sentence, append or list, we invoke the combiner with no arguments to produce the return value.

On the other hand, if accumulate's combiner argument is something like (lambda (x y) (word x '- y)) or max, then there's nothing accumulate can return, so we give an error message. (But it's a more descriptive error message than the first version; what message do you get when you call that first version with an empty second argument?)

It's somewhat of a kludge that we have to include in our procedure a list of the functions that can be called without arguments. What we'd like to do is invoke the combiner and find out if that causes an error, but Scheme doesn't provide a mechanism for causing errors on purpose and recovering from them. (Some dialects of Lisp do have that capability.)

Robustness

Instead of providing a special error message for empty-argument cases that `accumulate` can't handle, we could have just let it blow up:

```
(define (accumulate combiner stuff)          ;; non-robust version
  (if (not (empty? stuff))
      (real-accumulate combiner stuff)
      (combiner)))
```

Some questions about programming have clear right and wrong answers—if your program doesn't work, it's wrong! But the decision about whether to include the extra check for a procedure that's usable with an empty argument is a matter of judgment.

Here is the reasoning in favor of this simpler version: In either version, the user who tries to evaluate an expression like

```
(accumulate max '())
```

is going to get an error message. In the longer version we've spent both our own programming effort and a little of the computer's time on every invocation just to give a *different* error message from the one that Scheme would have given anyway. What's the point?

Here is the reasoning in favor of the longer version: In practice, the empty-argument situation isn't going to arise because someone uses a quoted empty sentence; instead the second argument to `accumulate` will be some expression whose value happens to be empty under certain conditions. The user will then have to debug the program that caused those conditions. Debugging is hard; we should make it easier for the user, if we can, by giving an error message that points clearly to the problem.

A program that behaves politely when given incorrect input is called *robust.* It's not always a matter of better or worse error messages. For example, a program that reads input from a human user might offer the chance to try again if some input value is incorrect. A robust program will also be alert for hardware problems, such as running out of space on a disk, or getting garbled information over a telephone connection to another machine because of noise on the line.

It's possible to pay either too little or too much attention to program robustness. If you're a professional programmer, your employer will expect your programs to survive errors that are likely to happen. On the other hand, your programs will be hard to read and debug if the error checking swamps the real work! As a student, unless you are specifically asked to "bulletproof" your program, don't answer exam questions by writing procedures like this one:

```
(define (even? num)                          ;; silly example
  (cond ((not (number? num)) (error "Not a number."))
        ((not (integer? num)) (error "Not an integer."))
        ((< num 0) (error "Argument must be positive."))
        (else (= (remainder num 2) 0))))
```

In the case of `accumulate`, we decided to be extra robust because we were writing a procedure for use in a beginning programming course. If we were writing this tool just for our own use, we might have chosen the non-robust version. Deciding how robust a program will be is a matter of taste.

Higher-Order Functions for Structured Lists

We've given you a fairly standard set of higher-order functions, but there's no law that says these are the only ones. Any time you notice yourself writing what feels like the same procedure over again, but with different details, consider inventing a higher-order function.

For example, here's a procedure we defined in Chapter 17.

```
(define (deep-pigl structure)
  (cond ((word? structure) (pigl structure))
        ((null? structure) '())
        (else (cons (deep-pigl (car structure))
                    (deep-pigl (cdr structure))))))
```

This procedure converts every word in a structured list to Pig Latin. Suppose we have a structure full of numbers and we want to compute all of their squares. We could write a specific procedure `deep-square`, but instead, we'll write a higher-order procedure:

```
(define (deep-map f structure)
  (cond ((word? structure) (f structure))
        ((null? structure) '())
        (else (cons (deep-map f (car structure))
                    (deep-map f (cdr structure))))))
```

The Zero-Trip Do Loop

The first programming language that provided a level of abstraction over the instructions understood directly by computer hardware was Fortran, a language that is still widely used today despite the advances in programming language design since then. Fortran remains popular because of the enormous number of useful programs that have already been written in it; if an improvement is needed, it's easier to modify the Fortran program than to start again in some more modern language.

Fortran includes a control mechanism called `do`, a sort of higher-order procedure that carries out a computation repeatedly, as `every` does. But instead of carrying out the computation once for each element of a given collection of data (like the sentence argument to `every`), `do` performs a computation once for each integer in a range specified by its endpoints. "For every number between 4 and 16, do such-and-such."

What if you specify endpoints such that the starting value is greater than the ending value? In the first implementation of Fortran, nobody thought very hard about this question, and they happened to implement `do` in such a way that if you specified a backward range, the computation was done once, for the given starting value, before Fortran noticed that it was past the ending value.

Twenty years later, a bunch of computer scientists argued that this behavior was wrong—that a `do` loop with its starting value greater than its ending value should not carry out its computation at all. This proposal for a "zero-trip `do` loop" was strongly opposed by Fortran old-timers, not because of any principle but because of all the thousands of Fortran programs that had been written to rely on the one-trip behavior.

The point of this story is that the Fortran users had to debate the issue so heatedly because they are stuck with only the control mechanisms that are built into the language. Fortran doesn't have the idea of function as data, so Fortran programmers can't write their own higher-order procedures. But you, using the techniques of this chapter, can create precisely the control mechanism that you need for whatever problem you happen to be working on.

Pitfalls

⇒ The most crucial point in inventing a higher-order function is to make sure that the pattern you have in mind really does generalize. For example, if you want to write a higher-order function for structured data, what is the base case? Will you use the tree abstract data type, or will you use `car/cdr` recursion?

⇒ When you generalize a pattern by adding a new argument (typically a procedure), be sure you add it to the recursive invocation(s) as well as to the formal parameter list!

Boring Exercises

19.1 What happens if you say the following?

```
(every cdr '((john lennon) (paul mccartney)
             (george harrison) (ringo starr)))
```

How is this different from using `map`, and why? How about `cadr` instead of `cdr`?

Real Exercises

19.2 Write `keep`. Don't forget that `keep` has to return a sentence if its second argument is a sentence, and a word if its second argument is a word.

(Hint: it might be useful to write a `combine` procedure that uses either `word` or `sentence` depending on the types of its arguments.)

19.3 Write the three-argument version of `accumulate` that we described.

```
> (three-arg-accumulate + 0 '(4 5 6))
15

> (three-arg-accumulate + 0 '())
0
```

```
> (three-arg-accumulate cons '() '(a b c d e))
(A B C D E)
```

19.4 Our `accumulate` combines elements from right to left. That is,

```
(accumulate - '(2 3 4 5))
```

computes $2-(3-(4-5))$. Write `left-accumulate`, which will compute $((2-3)-4)-5$ instead. (The result will be the same for an operation such as `+`, for which grouping order doesn't matter, but will be different for `-`.)

19.5 Rewrite the `true-for-all?` procedure from Exercise 8.10. Do not use `every`, `keep`, or `accumulate`.

19.6 Write a procedure `true-for-any-pair?` that takes a predicate and a sentence as arguments. The predicate must accept two words as its arguments. Your procedure should return `#t` if the argument predicate will return true for any two adjacent words in the sentence:

```
> (true-for-any-pair? equal? '(a b c b a))
#F

> (true-for-any-pair? equal? '(a b c c d))
#T

> (true-for-any-pair? < '(20 16 5 8 6))      ;; 5 is less than 8
#T
```

19.7 Write a procedure `true-for-all-pairs?` that takes a predicate and a sentence as arguments. The predicate must accept two words as its arguments. Your procedure should return `#t` if the argument predicate will return true for *every* two adjacent words in the sentence:

```
> (true-for-all-pairs? equal? '(a b c c d))
#F

> (true-for-all-pairs? equal? '(a a a a a))
#T

> (true-for-all-pairs? < '(20 16 5 8 6))
#F
```

```
> (true-for-all-pairs? < '(3 7 19 22 43))
#T
```

19.8 Rewrite `true-for-all-pairs?` (Exercise 19.7) using `true-for-any-pair?` (Exercise 19.6) as a helper procedure. Don't use recursion in solving this problem (except for the recursion you've already used to write `true-for-any-pair?`). Hint: You'll find the `not` procedure helpful.

19.9 Rewrite either of the sort procedures from Chapter 15 to take two arguments, a list and a predicate. It should sort the elements of that list according to the given predicate:

```
> (sort '(4 23 7 5 16 3) <)
(3 4 5 7 16 23)

> (sort '(4 23 7 5 16 3) >)
(23 16 7 5 4 3)

> (sort '(john paul george ringo) before?)
(GEORGE JOHN PAUL RINGO)
```

19.10 Write `tree-map`, analogous to our `deep-map`, but for trees, using the `datum` and `children` selectors.

19.11 Write `repeated`. (This is a hard exercise!)

19.12 Write `tree-reduce`. You may assume that the combiner argument can be invoked with no arguments.

```
> (tree-reduce
   +
   (make-node 3 (list (make-node 4 '())
                      (make-node 7 '())
                      (make-node 2 (list (make-node 3 '())
                                         (make-node 8 '()))))))
27
```

19.13 Write `deep-reduce`, similar to `tree-reduce`, but for structured lists:

```
> (deep-reduce word '(r ((a (m b) (l)) (e (r)))))
RAMBLER
```