5

Returning Functions

The previous chapter showed how the ability to pass functions as arguments leads to greater possibilities for abstraction. The more we can do to functions, the more we can take advantage of these possibilities. By defining functions to build and return new functions, we can magnify the effect of utilities which take functions as arguments.

The utilities in this chapter operate on functions. It would be more natural, at least in Common Lisp, to write many of them to operate on expressions—that is, as macros. A layer of macros will be superimposed on some of these operators in Chapter 15. However, it is important to know what part of the task can be done with functions, even if we will eventually call these functions only through macros.

5.1 Common Lisp Evolves

Common Lisp originally provided several pairs of complementary functions. The functions `remove-if` and `remove-if-not` make one such pair. If `pred` is a predicate of one argument, then

```
(remove-if-not #'pred lst)
```

is equivalent to

```
(remove-if #'(lambda (x) (not (pred x))) lst)
```

By varying the function given as an argument to one, we can duplicate the effect of the other. In that case, why have both? CLTL2 includes a new function intended for cases like this: `complement` takes a predicate *p* and returns a function which always returns the opposite value. When *p* returns true, the complement returns false, and vice versa. Now we can replace

```
(remove-if-not #'pred lst)
```

with the equivalent

```
(remove-if (complement #'pred) lst)
```

With `complement`, there is little justification for continuing to use the `-if-not` functions.[1] Indeed, CLTL2 (p. 391) says that their use is now deprecated. If they remain in Common Lisp, it will only be for the sake of compatibility.

The new `complement` operator is the tip of an important iceberg: functions which return functions. This has long been an important part of the idiom of Scheme. Scheme was the first Lisp to make functions lexical closures, and it is this which makes it interesting to have functions as return values.

It's not that we couldn't return functions in a dynamically scoped Lisp. The following function would work the same under dynamic or lexical scope:

```
(defun joiner (obj)
  (typecase obj
    (cons   #'append)
    (number #'+)))
```

It takes an object and, depending on its type, returns a function to add such objects together. We could use it to define a polymorphic `join` function that worked for numbers or lists:

```
(defun join (&rest args)
  (apply (joiner (car args)) args))
```

However, returning constant functions is the limit of what we can do with dynamic scope. What we can't do (well) is build functions at runtime; `joiner` can return one of two functions, but the two choices are fixed.

On page 18 we saw another function for returning functions, which relied on lexical scope:

```
(defun make-adder (n)
  #'(lambda (x) (+ x n)))
```

[1] Except perhaps `remove-if-not`, which is used more often than `remove-if`.

Calling `make-adder` will yield a closure whose behavior depends on the value originally given as an argument:

```
> (setq add3 (make-adder 3))
#<Interpreted-Function BF1356>
> (funcall add3 2)
5
```

Under lexical scope, instead of merely choosing among a group of constant functions, we can build new closures at runtime. With dynamic scope this technique is impossible.[2] If we consider how `complement` would be written, we see that it too must return a closure:

```
(defun complement (fn)
  #'(lambda (&rest args) (not (apply fn args))))
```

The function returned by `complement` uses the value of the parameter `fn` when `complement` was called. So instead of just choosing from a group of constant functions, `complement` can custom-build the inverse of any function:

```
> (remove-if (complement #'oddp) '(1 2 3 4 5 6))
(1 3 5)
```

Being able to pass functions as arguments is a powerful tool for abstraction. The ability to write functions which return functions allows us to make the most of it. The remaining sections present several examples of utilities which return functions.

5.2 Orthogonality

An *orthogonal* language is one in which you can express a lot by combining a small number of operators in a lot of different ways. Toy blocks are very orthogonal; a plastic model kit is hardly orthogonal at all. The main advantage of `complement` is that it makes a language more orthogonal. Before `complement`, Common Lisp had pairs of functions like `remove-if` and `remove-if-not`, `subst-if` and `subst-if-not`, and so on. With `complement` we can do without half of them.

The `setf` macro also improves Lisp's orthogonality. Earlier dialects of Lisp would often have pairs of functions for reading and writing data. With property-lists, for example, there would be one function to establish properties and another function to ask about them. In Common Lisp, we have only the latter, `get`. To

[2] Under dynamic scope, we could write something like `make-adder`, but it would hardly ever work. The binding of `n` would be determined by the environment in which the returned function was eventually called, and we might not have any control over that.

```
(defvar *!equivs* (make-hash-table))

(defun ! (fn)
  (or (gethash fn *!equivs*) fn))

(defun def! (fn fn!)
  (setf (gethash fn *!equivs*) fn!))
```

Figure 5.1: Returning destructive equivalents.

establish a property, we use `get` in combination with `setf`:

```
(setf (get 'ball 'color) 'red)
```

We may not be able to make Common Lisp smaller, but we can do something almost as good: use a smaller subset of it. Can we define any new operators which would, like `complement` and `setf`, help us toward this goal? There is at least one other way in which functions are grouped in pairs. Many functions also come in a destructive version: `remove-if` and `delete-if`, `reverse` and `nreverse`, `append` and `nconc`. By defining an operator to return the destructive counterpart of a function, we would not have to refer to the destructive functions directly.

Figure 5.1 contains code to support the notion of destructive counterparts. The global hash-table `*!equivs*` maps functions to their destructive equivalents; `!` returns destructive equivalents; and `def!` sets them. The name of the `!` (bang) operator comes from the Scheme convention of appending `!` to the names of functions with side-effects. Now once we have defined

```
(def! #'remove-if #'delete-if)
```

then instead of

```
(delete-if #'oddp lst)
```

we would say

```
(funcall (! #'remove-if) #'oddp lst)
```

Here the awkwardness of Common Lisp masks the basic elegance of the idea, which would be more visible in Scheme:

```
((! remove-if) oddp lst)
```

```
(defun memoize (fn)
  (let ((cache (make-hash-table :test #'equal)))
    #'(lambda (&rest args)
        (multiple-value-bind (val win) (gethash args cache)
          (if win
              val
              (setf (gethash args cache)
                    (apply fn args)))))))
```

Figure 5.2: Memoizing utility.

As well as greater orthogonality, the `!` operator brings a couple of other benefits. It makes programs clearer, because we can see immediately that `(! #'foo)` is the destructive equivalent of `foo`. Also, it gives destructive operations a distinct, recognizable form in source code, which is good because they should receive special attention when we are searching for a bug.

Since the relation between a function and its destructive counterpart will usually be known before runtime, it would be most efficient to define `!` as a macro, or even provide a read macro for it.

5.3 Memoizing

If some function is expensive to compute, and we expect sometimes to make the same call more than once, then it pays to *memoize:* to cache the return values of all the previous calls, and each time the function is about to be called, to look first in the cache to see if the value is already known.

Figure 5.2 contains a generalized memoizing utility. We give a function to `memoize`, and it returns an equivalent memoized version—a closure containing a hash-table in which to store the results of previous calls.

```
> (setq slowid (memoize #'(lambda (x) (sleep 5) x)))
#<Interpreted-Function C38346>
> (time (funcall slowid 1))
Elapsed Time = 5.15 seconds
1
> (time (funcall slowid 1))
Elapsed Time = 0.00 seconds
1
```

With a memoized function, repeated calls are just hash-table lookups. There is of course the additional expense of a lookup on each initial call, but since we

```
(defun compose (&rest fns)
  (if fns
      (let ((fn1 (car (last fns)))
            (fns (butlast fns)))
        #'(lambda (&rest args)
            (reduce #'funcall fns
                    :from-end t
                    :initial-value (apply fn1 args))))
      #'identity))
```

Figure 5.3: An operator for functional composition.

would only memoize a function that was sufficiently expensive to compute, it's reasonable to assume that this cost is insignificant in comparison.

Though adequate for most uses, this implementation of `memoize` has several limitations. It treats calls as identical if they have `equal` argument lists; this could be too strict if the function had keyword parameters. Also, it is intended only for single-valued functions, and cannot store or return multiple values.

5.4 Composing Functions

The complement of a function f is denoted $\sim f$. Section 5.1 showed that closures make it possible to define $\sim$ as a Lisp function. Another common operation on functions is composition, denoted by the operator $\circ$. If f and g are functions, then $f \circ g$ is also a function, and $f \circ g(x) = f(g(x))$. Closures also make it possible to define $\circ$ as a Lisp function.

Figure 5.3 defines a `compose` function which takes any number of functions and returns their composition. For example

```
(compose #'list #'1+)
```

returns a function equivalent to

```
#'(lambda (x) (list (1+ x)))
```

All the functions given as arguments to `compose` must be functions of one argument, except the last. On the last function there are no restrictions, and whatever arguments it takes, so will the function returned by compose:

```
> (funcall (compose #'1+ #'find-if) #'oddp '(2 3 4))
4
```

```
(defun fif (if then &optional else)
  #'(lambda (x)
      (if (funcall if x)
          (funcall then x)
          (if else (funcall else x)))))

(defun fint (fn &rest fns)
  (if (null fns)
      fn
      (let ((chain (apply #'fint fns)))
        #'(lambda (x)
            (and (funcall fn x) (funcall chain x))))))

(defun fun (fn &rest fns)
  (if (null fns)
      fn
      (let ((chain (apply #'fun fns)))
        #'(lambda (x)
            (or (funcall fn x) (funcall chain x))))))
```

Figure 5.4: More function builders.

Since `not` is a Lisp function, `complement` is a special case of `compose`. It could be defined as:

```
(defun complement (pred)
  (compose #'not pred))
```

We can combine functions in other ways than by composing them. For example, we often see expressions like

```
(mapcar #'(lambda (x)
            (if (slave x)
                (owner x)
                (employer x)))
        people)
```

We could define an operator to build functions like this one automatically. Using `fif` from Figure 5.4, we could get the same effect with:

```
(mapcar (fif #'slave #'owner #'employer)
        people)
```

Figure 5.4 contains several other constructors for commonly occurring types of functions. The second, `fint`, is for cases like this:

```
(find-if #'(lambda (x)
             (and (signed x) (sealed x) (delivered x)))
         docs)
```

The predicate given as the second argument to `find-if` defines the intersection of the three predicates called within it. With `fint`, whose name stands for "function intersection," we can say:

```
(find-if (fint #'signed #'sealed #'delivered) docs)
```

We can define a similar operator to return the union of a set of predicates. The function `fun` is like `fint` but uses `or` instead of `and`.

5.5 Recursion on Cdrs

Recursive functions are so important in Lisp programs that it would be worth having utilities to build them. This section and the next describe functions which build the two most common types. In Common Lisp, these functions are a little awkward to use. Once we get into the subject of macros, we will see how to put a more elegant facade on this machinery. Macros for building recursers are discussed in Sections 15.2 and 15.3.

Repeated patterns in a program are a sign that it could have been written at a higher level of abstraction. What pattern is more commonly seen in Lisp programs than a function like this:

```
(defun our-length (lst)
  (if (null lst)
      0
      (1+ (our-length (cdr lst)))))
```

or this:

```
(defun our-every (fn lst)
  (if (null lst)
      t
      (and (funcall fn (car lst))
           (our-every fn (cdr lst)))))
```

Structurally these two functions have a lot in common. They both operate recursively on successive cdrs of a list, evaluating the same expression on each step,

```
(defun lrec (rec &optional base)
  (labels ((self (lst)
             (if (null lst)
                 (if (functionp base)
                     (funcall base)
                     base)
                 (funcall rec (car lst)
                              #'(lambda ()
                                  (self (cdr lst)))))))
    #'self))
```

Figure 5.5: Function to define flat list recursers.

except in the base case, where they return a distinct value. This pattern appears so frequently in Lisp programs that experienced programmers can read and reproduce it without stopping to think. Indeed, the lesson is so quickly learned, that the question of how to package the pattern in a new abstraction does not arise.

However, a pattern it is, all the same. Instead of writing these functions out by hand, we should be able to write a function which will generate them for us. Figure 5.5 contains a function-builder called `lrec` ("list recurser") which should be able to generate most functions that recurse on successive cdrs of a list.

The first argument to `lrec` must be a function of two arguments: the current car of the list, and a function which can be called to continue the recursion. Using `lrec` we could express `our-length` as:

```
(lrec #'(lambda (x f) (1+ (funcall f))) 0)
```

To find the length of the list, we don't need to look at the elements, or stop partway, so the object `x` is always ignored, and the function `f` always called. However, we need to take advantage of both possibilities to express `our-every`, for e.g. `oddp`:[3]

```
(lrec #'(lambda (x f) (and (oddp x) (funcall f))) t)
```

The definition of `lrec` uses `labels` to build a local recursive function called `self`. In the recursive case the function `rec` is passed two arguments, the current car of the list, and a function embodying the recursive call. In functions like `our-every`, where the recursive case is an `and`, if the first argument returns false we want to stop right there. Which means that the argument passed in the recursive

[3] In one widely used Common Lisp, `functionp` erroneously returns true for `t` and `nil`. In that implementation it won't work to give either as the second argument to `lrec`.

```
; copy-list
(lrec #'(lambda (x f) (cons x (funcall f))))

; remove-duplicates
(lrec #'(lambda (x f) (adjoin x (funcall f))))

; find-if, for some function fn
(lrec #'(lambda (x f) (if (fn x) x (funcall f))))

; some, for some function fn
(lrec #'(lambda (x f) (or (fn x) (funcall f))))
```

Figure 5.6: Functions expressed with `lrec`.

case must not be a value but a function, which we can call (if we want) in order to get a value.

Figure 5.6 shows some existing Common Lisp functions defined with `lrec`.[4] Calling `lrec` will not always yield the most efficient implementation of a given function. Indeed, `lrec` and the other recurser generators to be defined in this chapter tend to lead one away from tail-recursive solutions. For this reason they are best suited for use in initial versions of a program, or in parts where speed is not critical.

5.6 Recursion on Subtrees

There is another recursive pattern commonly found in Lisp programs: recursion on subtrees. This pattern is seen in cases where you begin with a possibly nested list, and want to recurse down both its car and its cdr.

The Lisp list is a versatile structure. Lists can represent, among other things, sequences, sets, mappings, arrays, and trees. There are several different ways to interpret a list as a tree. The most common is to regard the list as a binary tree whose left branch is the car and whose right branch is the cdr. (In fact, this is usually the internal representation of lists.) Figure 5.7 shows three examples of lists and the trees they represent. Each internal node in such a tree corresponds to a dot in the dotted-pair representation of the list, so the tree structure may be

[4] In some implementations, you may have to set `*print-circle*` to `t` before these functions can be displayed.

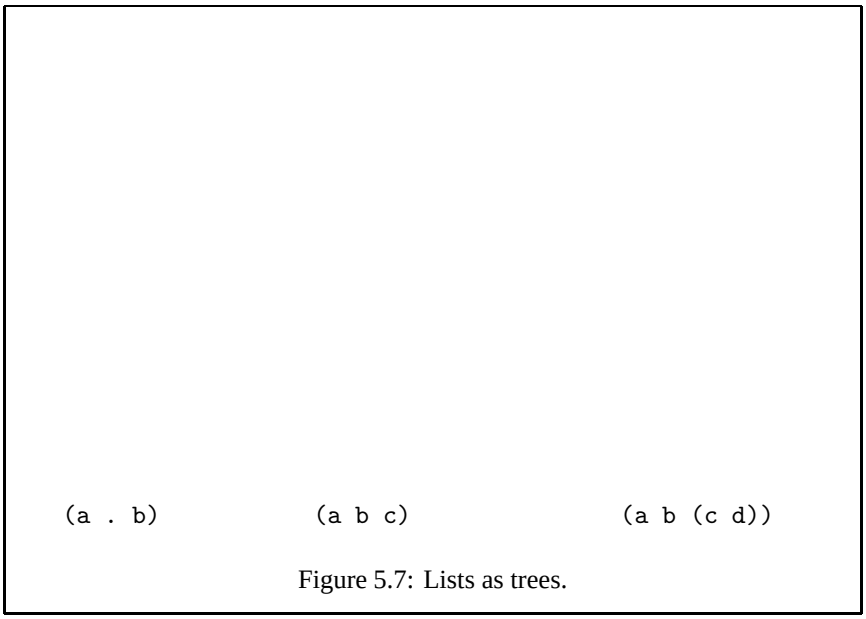

Figure 5.7: Lists as trees.

easier to interpret if the lists are considered in that form:

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

Any list can be interpreted as a binary tree. Hence the distinction between pairs of Common Lisp functions like `copy-list` and `copy-tree`. The former copies a list as a sequence—if the list contains sublists, the sublists, being mere elements in the sequence, are not copied:

```
> (setq x     '(a b)
        listx (list x 1))
((A B) 1)
> (eq x (car (copy-list listx)))
T
```

In contrast, `copy-tree` copies a list as a tree—sublists are subtrees, and so must also be copied:

```
> (eq x (car (copy-tree listx)))
NIL
```

We could define a version of `copy-tree` as follows:

```
(defun our-copy-tree (tree)
  (if (atom tree)
      tree
      (cons (our-copy-tree (car tree))
            (if (cdr tree) (our-copy-tree (cdr tree))))))
```

This definition turns out to be one instance of a common pattern. (Some of the following functions are written a little oddly in order to make the pattern obvious.) Consider for example a utility to count the number of leaves in a tree:

```
(defun count-leaves (tree)
  (if (atom tree)
      1
      (+ (count-leaves (car tree))
         (or (if (cdr tree) (count-leaves (cdr tree)))
             1))))
```

A tree has more leaves than the atoms you can see when it is represented as a list:

```
> (count-leaves '((a b (c d)) (e) f))
10
```

The leaves of a tree are all the atoms you can see when you look at the tree in its *dotted-pair* representation. In dotted-pair notation, `((a b (c d)) (e) f)` would have four `nil`s that aren't visible in the list representation (one for each pair of parentheses) so `count-leaves` returns `10`.

In the last chapter we defined several utilities which operate on trees. For example, `flatten` (page 47) takes a tree and returns a list of all the atoms in it. That is, if you give `flatten` a nested list, you'll get back a list that looks the same except that it's missing all but the outermost pair of parentheses:

```
> (flatten '((a b (c d)) (e) f ()))
(A B C D E F)
```

This function could also be defined (somewhat inefficiently) as follows:

```
(defun flatten (tree)
  (if (atom tree)
      (mklist tree)
      (nconc (flatten (car tree))
             (if (cdr tree) (flatten (cdr tree))))))
```

Finally, consider `rfind-if`, a recursive version of `find-if` which works on trees as well as flat lists:

```
(defun rfind-if (fn tree)
  (if (atom tree)
      (and (funcall fn tree) tree)
      (or (rfind-if fn (car tree))
          (if (cdr tree) (rfind-if fn (cdr tree))))))
```

To generalize `find-if` for trees, we have to decide whether we want to search for just leaves, or for whole subtrees. Our `rfind-if` takes the former approach, so the caller can assume that the function given as the first argument will only be called on atoms: ○

```
> (rfind-if (fint #'numberp #'oddp) '(2 (3 4) 5))
3
```

How similar in form are these four functions, `copy-tree`, `count-leaves`, `flatten`, and `rfind-if`. Indeed, they're all instances of an archetypal function for recursion on subtrees. As with recursion on cdrs, we need not leave this archetype to float vaguely in the background—we can write a function to generate instances of it.

To get at the archetype itself, let's look at these functions and see what's not pattern. Essentially `our-copy-tree` is two facts:

1. In the base case it returns its argument.
2. In the recursive case, it applies `cons` to the recursions down the left (car) and right (cdr) subtrees.

We should thus be able to express it as a call to a builder with two arguments:

```
(ttrav #'cons #'identity)
```

A definition of `ttrav` ("tree traverser") is shown in Figure 5.8. Instead of passing one value in the recursive case, we pass two, one for the left subtree and one for the right. If the `base` argument is a function it will be called on the current leaf. In flat list recursion, the base case is always `nil`, but in tree recursion the base case could be an interesting value, and we might want to use it.

With `ttrav` we could express all the preceding functions except `rfind-if`. (They are shown in Figure 5.9.) To define `rfind-if` we need a more general tree recursion builder which gives us control over when, and if, the recursive calls are made. As the first argument to `ttrav` we gave a function which took the *results* of the recursive calls. For the general case, we want to use instead a function which takes two closures representing the calls themselves. Then we can write recursers which only traverse as much of the tree as they want to.

```
(defun ttrav (rec &optional (base #'identity))
  (labels ((self (tree)
             (if (atom tree)
                 (if (functionp base)
                     (funcall base tree)
                     base)
                 (funcall rec (self (car tree))
                              (if (cdr tree)
                                  (self (cdr tree)))))))
    #'self))
```

Figure 5.8: Function for recursion on trees.

```
; our-copy-tree
(ttrav #'cons)

; count-leaves
(ttrav #'(lambda (l r) (+ l (or r 1)))  1)

; flatten
(ttrav #'nconc #'mklist)
```

Figure 5.9: Functions expressed with `ttrav`.

Functions built by `ttrav` always traverse a whole tree. That's fine for functions like `count-leaves` or `flatten`, which have to traverse the whole tree anyway. But we want `rfind-if` to stop searching as soon as it finds what it's looking for. It must be built by the more general `trec`, shown in Figure 5.10. The second arg to `trec` should be a function of three arguments: the current object and the two recursers. The latter two will be closures representing the recursions down the left and right subtrees. With `trec` we would define `flatten` as:

```
(trec #'(lambda (o l r) (nconc (funcall l) (funcall r)))
      #'mklist)
```

Now we can also express `rfind-if` for e.g. `oddp` as:

```
(trec #'(lambda (o l r) (or (funcall l) (funcall r)))
      #'(lambda (tree) (and (oddp tree) tree)))
```

```
(defun trec (rec &optional (base #'identity))
  (labels
    ((self (tree)
       (if (atom tree)
           (if (functionp base)
               (funcall base tree)
               base)
           (funcall rec tree
                       #'(lambda ()
                           (self (car tree)))
                       #'(lambda ()
                           (if (cdr tree)
                               (self (cdr tree))))))))
    #'self))
```

Figure 5.10: Function for recursion on trees.

5.7 When to Build Functions

Expressing functions by calls to constructors instead of sharp-quoted lambda-expressions could, unfortunately, entail unnecessary work at runtime. A sharp-quoted lambda-expression is a constant, but a call to a constructor function will be evaluated at runtime. If we really have to make this call at runtime, it might not be worth using constructor functions. However, at least some of the time we can call the constructor beforehand. By using `#.`, the sharp-dot read macro, we can have the new functions built at read-time. So long as `compose` and its arguments are defined when this expression is read, we could say, for example,

```
(find-if #.(compose #'oddp #'truncate) lst)
```

Then the call to `compose` would be evaluated by the reader, and the resulting function inserted as a constant into our code. Since both `oddp` and `truncate` are built-in, it would safe to assume that we can evaluate the `compose` at read-time, so long as `compose` itself were already loaded.

In general, composing and combining functions is more easily and efficiently done with macros. This is particularly true in Common Lisp, with its separate name-space for functions. After introducing macros, we will in Chapter 15 cover much of the ground we covered here, but in a more luxurious vehicle.