3

Functional Programming

The previous chapter explained how Lisp and Lisp programs are both built out of a single raw material: the function. Like any building material, its qualities influence both the kinds of things we build, and the way we build them.

This chapter describes the kind of construction methods which prevail in the Lisp world. The sophistication of these methods allows us to attempt more ambitious kinds of programs. The next chapter will describe one particularly important class of programs which become possible in Lisp: programs which evolve instead of being developed by the old plan-and-implement method.

3.1 Functional Design

The character of an object is influenced by the elements from which it is made. A wooden building looks different from a stone one, for example. Even when you are too far away to see wood or stone, you can tell from the overall shape of the building what it's made of. The character of Lisp functions has a similar influence on the structure of Lisp programs.

Functional programming means writing programs which work by returning values instead of by performing side-effects. Side-effects include destructive changes to objects (e.g. by `rplaca`) and assignments to variables (e.g. by `setq`). If side-effects are few and localized, programs become easier to read, test, and debug. Lisp programs have not always been written in this style, but over time Lisp and functional programming have gradually become inseparable.

An example will show how functional programming differs from what you might do in another language. Suppose for some reason we want the elements of

```
(defun bad-reverse (lst)
  (let* ((len (length lst))
         (ilimit (truncate (/ len 2))))
    (do ((i 0 (1+ i))
         (j (1- len) (1- j)))
        ((>= i ilimit))
      (rotatef (nth i lst) (nth j lst)))))
```

Figure 3.1: A function to reverse lists.

a list in the reverse order. Instead of writing a function to reverse lists, we write a function which takes a list, and returns a list with the same elements in the reverse order.

Figure 3.1 contains a function to reverse lists. It treats the list as an array, reversing it in place; its return value is irrelevant:

```
> (setq lst '(a b c))
(A B C)
> (bad-reverse lst)
NIL
> lst
(C B A)
```

As its name suggests, `bad-reverse` is far from good Lisp style. Moreover, its ugliness is contagious: because it works by side-effects, it will also draw its callers away from the functional ideal.

Though cast in the role of the villain, `bad-reverse` does have one merit: it shows the Common Lisp idiom for swapping two values. The `rotatef` macro rotates the values of any number of generalized variables—that is, expressions you could give as the first argument to `setf`. When applied to just two arguments, the effect is to swap them.

In contrast, Figure 3.2 shows a function which returns reversed lists. With `good-reverse`, we get the reversed list as the return value; the original list is not touched.

```
> (setq lst '(a b c))
(A B C)
> (good-reverse lst)
(C B A)
> lst
(A B C)
```

```
(defun good-reverse (lst)
  (labels ((rev (lst acc)
             (if (null lst)
                 acc
                 (rev (cdr lst) (cons (car lst) acc)))))
    (rev lst nil)))
```

Figure 3.2: A function to return reversed lists.

It used to be thought that you could judge someone's character by looking at the shape of his head. Whether or not this is true of people, it is generally true of Lisp programs. Functional programs have a different shape from imperative ones. The structure in a functional program comes entirely from the composition of arguments within expressions, and since arguments are indented, functional code will show more variation in indentation. Functional code looks fluid [1] on the page; imperative code looks solid and blockish, like Basic.

Even from a distance, the shapes of `bad-` and `good-reverse` suggest which is the better program. And despite being shorter, `good-reverse` is also more efficient: $O(n)$ instead of $O(n^2)$.

We are spared the trouble of writing `reverse` because Common Lisp has it built-in. It is worth looking briefly at this function, because it is one that often brings to the surface misconceptions about functional programming. Like `good-reverse`, the built-in `reverse` works by returning a value—it doesn't touch its arguments. But people learning Lisp may assume that, like `bad-reverse`, it works by side-effects. If in some part of a program they want a list `lst` to be reversed, they may write

```
(reverse lst)
```

and wonder why the call seems to have no effect. In fact, if we want *effects* from such a function, we have to see to it ourselves in the calling code. That is, we need to write

```
(setq lst (reverse lst))
```

instead. Operators like `reverse` are intended to be called for return values, not side-effects. It is worth writing your own programs in this style too—not only for its inherent benefits, but because, if you don't, you will be working against the language.

[1] For a characteristic example, see page 242.

One of the points we ignored in the comparison of `bad-` and `good-reverse` is that `bad-reverse` doesn't cons. Instead of building new list structure, it operates on the original list. This can be dangerous—the list could be needed elsewhere in the program—but for efficiency it is sometimes necessary. For such cases, Common Lisp provides an *O(n)* destructive reversing function called `nreverse`. ○

A destructive function is one that can alter the arguments passed to it. However, even destructive functions usually work by returning values: you have to assume that `nreverse` will recycle lists you give to it as arguments, but you still can't assume that it will reverse them. As before, the reversed list has to be found in the return value. You still can't write

```
(nreverse lst)
```

in the middle of a function and assume that afterwards `lst` will be reversed. This is what happens in most implementations:

```
> (setq lst '(a b c))
(A B C)
> (nreverse lst)
(C B A)
> lst
(A)
```

To reverse `lst`, you have would have to set `lst` to the return value, as with plain `reverse`.

If a function is advertised as destructive, that doesn't mean that it's meant to be called for side-effects. The danger is, some destructive functions give the impression that they are. For example,

```
(nconc x y)
```

almost always has the same effect as

```
(setq x (nconc x y))
```

If you wrote code which relied on the former idiom, it might seem to work for some time. However, it wouldn't do what you expected when `x` was `nil`.

Only a few Lisp operators are intended to be called for side-effects. In general, the built-in operators are meant to be called for their return values. Don't be misled by names like `sort`, `remove`, or `substitute`. If you want side-effects, use `setq` on the return value.

This very rule suggests that some side-effects are inevitable. Having functional programming as an ideal doesn't imply that programs should never have side-effects. It just means that they should have no more than necessary.

It may take time to develop this habit. One way to start is to treat the following operators as if there were a tax on their use:

```
set setq setf psetf psetq incf decf push pop pushnew
rplaca rplacd  rotatef shiftf  remf remprop  remhash
```

and also `let*`, in which imperative programs often lie concealed. Treating these operators as taxable is only proposed as a help toward, not a criterion for, good Lisp style. However, this alone can get you surprisingly far.

In other languages, one of the most common causes of side-effects is the need for a function to return multiple values. If functions can only return one value, they have to "return" the rest by altering their parameters. Fortunately, this isn't necessary in Common Lisp, because any function can return multiple values.

The built-in function `truncate` returns two values, for example—the truncated integer, and what was cut off in order to create it. A typical implementation will print both when `truncate` is called at the toplevel:

```
> (truncate 26.21875)
26
0.21875
```

When the calling code only wants one value, the first one is used:

```
> (= (truncate 26.21875) 26)
T
```

The calling code can catch both return values by using a `multiple-value-bind`. This operator takes a list of variables, a call, and a body of code. The body is evaluated with the variables bound to the respective return values from the call:

```
> (multiple-value-bind (int frac) (truncate 26.21875)
    (list int frac))
(26 0.21875)
```

Finally, to return multiple values, we use the `values` operator:

```
> (defun powers (x)
    (values x (sqrt x) (expt x 2)))
POWERS
> (multiple-value-bind (base root square) (powers 4)
    (list base root square))
(4 2.0 16)
```

Functional programming is a good idea in general. It is a particularly good idea in Lisp, because Lisp has evolved to support it. Built-in operators like `reverse` and `nreverse` are meant to be used in this way. Other operators, like `values` and `multiple-value-bind`, have been provided specifically to make functional programming easier.

3.2 Imperative Outside-In

The aims of functional programming may show more clearly when contrasted with those of the more common approach, imperative programming. A functional program tells you what it wants; an imperative program tells you what to do. A functional program says "Return a list of `a` and the square of the first element of `x`:"

```
(defun fun (x)
  (list 'a (expt (car x) 2)))
```

An imperative programs says "Get the first element of `x`, then square it, then return a list of `a` and the square:"

```
(defun imp (x)
  (let (y sqr)
    (setq y (car x))
    (setq sqr (expt y 2))
    (list 'a sqr)))
```

Lisp users are fortunate in being able to write this program both ways. Some languages are only suited to imperative programming—notably Basic, along with most machine languages. In fact, the definition of `imp` is similar in form to the machine language code that most Lisp compilers would generate for `fun`.

Why write such code when the compiler could do it for you? For many programmers, this question does not even arise. A language stamps its pattern on our thoughts: someone used to programming in an imperative language may have begun to conceive of programs in imperative terms, and may actually find it easier to write imperative programs than functional ones. This habit of mind is worth overcoming if you have a language that will let you.

For alumni of other languages, beginning to use Lisp may be like stepping onto a skating rink for the first time. It's actually much easier to get around on ice than it is on dry land—*if* you use skates. Till then you will be left wondering what people see in this sport.

What skates are to ice, functional programming is to Lisp. Together the two allow you to travel more gracefully, with less effort. But if you are accustomed

to another mode of travel, this may not be your experience at first. One of the obstacles to learning Lisp as a second language is learning to program in a functional style.

Fortunately there is a trick for transforming imperative programs into functional ones. You can begin by applying this trick to finished code. Soon you will begin to anticipate yourself, and transform your code as you write it. Soon after that, you will begin to conceive of programs in functional terms from the start.

The trick is to realize that an imperative program is a functional program turned inside-out. To find the functional program implicit in our imperative one, we just turn it outside-in. Let's try this technique on `imp`.

The first thing we notice is the creation of `y` and `sqr` in the initial `let`. This is a sign that bad things are to follow. Like `eval` at runtime, uninitialized variables are so rarely needed that they should generally be treated as a symptom of some illness in the program. Such variables are often used like pins which hold the program down and keep it from coiling into its natural shape.

However, we ignore them for the time being, and go straight to the end of the function. What occurs last in an imperative program occurs outermost in a functional one. So our first step is to grab the final call to `list` and begin stuffing the rest of the program inside it—just like turning a shirt inside-out. We continue by applying the same transformation repeatedly, just as we would with the sleeves of the shirt, and in turn with their cuffs.

Starting at the end, we replace `sqr` with `(expt y 2)`, yielding:

```
(list 'a (expt y 2)))
```

Then we replace `y` by `(car x)`:

```
(list 'a (expt (car x) 2))
```

Now we can throw away the rest of the code, having stuffed it all into the last expression. In the process we removed the need for the variables `y` and `sqr`, so we can discard the `let` as well.

The final result is shorter than what we began with, and easier to understand. In the original code, we're faced with the final expression `(list 'a sqr)`, and it's not immediately clear where the value of `sqr` comes from. Now the source of the return value is laid out for us like a road map.

The example in this section was a short one, but the technique scales up. Indeed, it becomes more valuable as it is applied to larger functions. Even functions which perform side-effects can be cleaned up in the portions which don't.

3.3 Functional Interfaces

Some side-effects are worse than others. For example, though this function calls `nconc`

```
(defun qualify (expr)
  (nconc (copy-list expr) (list 'maybe)))
```

it preserves referential transparency.[2] If you call it with a given argument, it will always return the same (`equal`) value. From the caller's point of view, `qualify` might as well be purely functional code. We can't say the same for `bad-reverse` (page 29), which actually modifies its argument.

Instead of treating all side-effects as equally bad, it would be helpful if we had some way of distinguishing between such cases. Informally, we could say that it's harmless for a function to modify something that no one else owns. For example, the `nconc` in `qualify` is harmless because the list given as the first argument is freshly consed. No one else could own it.

In the general case, we have to talk about ownership not by functions, but by invocations of functions. Though no one else owns the variable `x` here,

```
(let ((x 0))
  (defun total (y)
    (incf x y)))
```

the effects of one call will be visible in succeeding ones. So the rule should be: a given invocation can safely modify what it uniquely owns.

Who owns arguments and return values? The convention in Lisp seems to be that an invocation owns objects it receives as return values, but not objects passed to it as arguments. Functions that modify their arguments are distinguished by the label "destructive," but there is no special name for functions that modify objects returned to them.

This function adheres to the convention, for example:

```
(defun ok (x)
  (nconc (list 'a x) (list 'c)))
```

It calls `nconc`, which doesn't, but since the list spliced by `nconc` will always be freshly made rather than, say, a list passed to `ok` as an argument, `ok` itself is ok.

If it were written slightly differently, however,

```
(defun not-ok (x)
  (nconc (list 'a) x (list 'c)))
```

[2] A definition of referential transparency appears on page 198.

then the call to nconc would be modifying an argument passed to not-ok.

Many Lisp programs violate this convention, at least locally. However, as we saw with ok, local violations need not disqualify the calling function. And functions which do meet the preceding conditions will retain many of the advantages of purely functional code.

To write programs that are really indistinguishable from functional code, we have to add one more condition. Functions can't share objects with other code that doesn't follow the rules. For example, though this function doesn't have side-effects,

```
(defun anything (x)
  (+ x *anything*))
```

its return value depends on the global variable *anything*. So if any other function can alter the value of this variable, anything could return anything.

Code written so that each invocation only modifies what it owns is almost as good as purely functional code. A function that meets all the preceding conditions at least presents a functional interface to the world: if you call it twice with the same arguments, you should get the same results. And this, as the next section will show, is a crucial ingredient in bottom-up programming.

One problem with destructive operations is that, like global variables, they can destroy the locality of a program. When you're writing functional code, you can narrow your focus: you only need consider the functions that call, or are called by, the one you're writing. This benefit disappears when you want to modify something destructively. It could be used anywhere.

The conditions above do not guarantee the perfect locality you get with purely functional code, though they do improve things somewhat. For example, suppose that f calls g as below:

```
(defun f (x)
  (let ((val (g x)))
    ; safe to modify val here?
    ))
```

Is it safe for f to nconc something onto val? Not if g is identity: then we would be modifying something originally passed as an argument to f itself.

So even in programs which do follow the convention, we may have to look beyond f if we want to modify something there. However, we don't have to look as far: instead of worrying about the whole program, we now only have to consider the subtree beginning with f.

A corollary of the convention above is that functions shouldn't return anything that isn't safe to modify. Thus one should avoid writing functions whose return

values incorporate quoted objects. If we define `exclaim` so that its return value incorporates a quoted list,

```
(defun exclaim (expression)
  (append expression '(oh my)))
```

Then any later destructive modification of the return value

```
> (exclaim '(lions and tigers and bears))
(LIONS AND TIGERS AND BEARS OH MY)
> (nconc * '(goodness))
(LIONS AND TIGERS AND BEARS OH MY GOODNESS)
```

could alter the list within the function:

```
> (exclaim '(fixnums and bignums and floats))
(FIXNUMS AND BIGNUMS AND FLOATS OH MY GOODNESS)
```

To make `exclaim` proof against such problems, it should be written:

```
(defun exclaim (expression)
  (append expression (list 'oh 'my)))
```

There is one major exception to the rule that functions shouldn't return quoted lists: the functions which generate macro expansions. Macro expanders can safely incorporate quoted lists in the expansions they generate, if the expansions are going straight to the compiler.

Otherwise, one might as well be a suspicious of quoted lists generally. Many other uses of them are likely to be something which ought to be done with a macro like `in` (page 152).

3.4 Interactive Programming

The previous sections presented the functional style as a good way of organizing programs. But it is more than this. Lisp programmers did not adopt the functional style purely for aesthetic reasons. They use it because it makes their work easier. In Lisp's dynamic environment, functional programs can be written with unusual speed, and at the same time, can be unusually reliable.

In Lisp it is comparatively easy to debug programs. A lot of information is available at runtime, which helps in tracing the causes of errors. But even more important is the ease with which you can *test* programs. You don't have to compile a program and test the whole thing at once. You can test functions individually by calling them from the toplevel loop.

Incremental testing is so valuable that Lisp style has evolved to take advantage of it. Programs written in the functional style can be understood one function at a time, and from the point of view of the reader this is its main advantage. However, the functional style is also perfectly adapted to incremental testing: programs written in this style can also be *tested* one function at a time. When a function neither examines nor alters external state, any bugs will appear immediately. Such a function can affect the outside world only through its return values. Insofar as these are what you expected, you can trust the code which produced them.

Experienced Lisp programmers actually design their programs to be easy to test:

1. They try to segregate side-effects in a few functions, allowing the greater part of the program to be written in a purely functional style.

2. If a function must perform side-effects, they try at least to give it a functional interface.

3. They give each function a single, well-defined purpose.

When a function is written, they can test it on a selection of representative cases, then move on to the next one. If each brick does what it's supposed to do, the wall will stand.

In Lisp, the wall can be better-designed as well. Imagine the kind of conversation you would have with someone so far away that there was a transmission delay of one minute. Now imagine speaking to someone in the next room. You wouldn't just have the same conversation faster, you would have a different kind of conversation. In Lisp, developing software is like speaking face-to-face. You can test code as you're writing it. And instant turnaround has just as dramatic an effect on development as it does on conversation. You don't just write the same program faster; you write a different kind of program.

How so? When testing is quicker you can do it more often. In Lisp, as in any language, development is a cycle of writing and testing. But in Lisp the cycle is very short: single functions, or even parts of functions. And if you test everything as you write it, you will know where to look when errors occur: in what you wrote last. Simple as it sounds, this principle is to a large extent what makes bottom-up programming feasible. It brings an extra degree of confidence which enables Lisp programmers to break free, at least part of the time, from the old plan-and-implement style of software development.

Section 1.1 stressed that bottom-up design is an evolutionary process. You build up a language as you write a program in it. This approach can work only if you *trust* the lower levels of code. If you really want to use this layer as a language, you have to be able to assume, as you would with any language, that any bugs you encounter are bugs in your application and not in the language itself.

So your new abstractions are supposed to bear this heavy burden of responsibility, and yet you're supposed to just spin them off as the need arises? Just so; in Lisp you can have both. When you write programs in a functional style and test them incrementally, you can have the flexibility of doing things on the spur of the moment, plus the kind of reliability one usually associates with careful planning.