15

Macros Returning Functions

Chapter 5 showed how to write functions which return other functions. Macros make the task of combining operators much easier. This chapter will show how to use macros to build abstractions which are equivalent to those defined in Chapter 5, but cleaner and more efficient.

15.1 Building Functions

If f and g are functions, then $f \circ g(x) = f(g(x))$. Section 5.4 showed how to implement the $\circ$ operator as a Lisp function called `compose`:

```
> (funcall (compose #'list #'1+) 2)
(3)
```

In this section, we consider ways to define better function builders with macros. Figure 15.1 contains a general function-builder called `fn`, which builds compound functions from their descriptions. Its argument should be an expression of the form (*operator* . *arguments*). The *operator* can be the name of a function or macro—or `compose`, which is treated specially. The *arguments* can be names of functions or macros of one argument, or expressions that could be arguments to `fn`. For example,

```
(fn (and integerp oddp))
```

yields a function equivalent to

```
#'(lambda (x) (and (integerp x) (oddp x)))
```

```
(defmacro fn (expr) `#',(rbuild expr))

(defun rbuild (expr)
  (if (or (atom expr) (eq (car expr) 'lambda))
      expr
      (if (eq (car expr) 'compose)
          (build-compose (cdr expr))
          (build-call (car expr) (cdr expr)))))

(defun build-call (op fns)
  (let ((g (gensym)))
    `(lambda (,g)
       (,op ,@(mapcar #'(lambda (f)
                          `(,(rbuild f) ,g))
                      fns)))))

(defun build-compose (fns)
  (let ((g (gensym)))
    `(lambda (,g)
       ,(labels ((rec (fns)
                   (if fns
                       `(,(rbuild (car fns))
                         ,(rec (cdr fns)))
                       g)))
          (rec fns)))))
```

Figure 15.1: General function-building macro.

If we use `compose` as the operator, we get a function representing the composition of the arguments, but without the explicit `funcall`s that were needed when compose was defined as a function. For example,

```
(fn (compose list 1+ truncate))
```

expands into:

```
#'(lambda (#:g1) (list (1+ (truncate #:g1))))
```

which enables inline compilation of simple functions like `list` and `1+`. The `fn` macro takes names of operators in the general sense; lambda-expressions are allowed too, as in

```
(fn (compose (lambda (x) (+ x 3)) truncate))
```

which expands into

```
#'(lambda (#:g2) ((lambda (x) (+ x 3)) (truncate #:g2)))
```

Here the function expressed as a lambda-expression will certainly be compiled inline, whereas a sharp-quoted lambda-expression given as an argument to the function `compose` would have to be funcalled.

Section 5.4 showed how to define three more function builders: `fif`, `fint`, and `fun`. These are now subsumed in the general `fn` macro. Using `and` as the operator yields the intersection of the operators given as arguments:

```
> (mapcar (fn (and integerp oddp))
          '(c 3 p 0))
(NIL T NIL NIL)
```

while `or` yields the union:

```
> (mapcar (fn (or integerp symbolp))
          '(c 3 p 0.2))
(T T T NIL)
```

and `if` yields a function whose body is a conditional:

```
> (map1-n (fn (if oddp 1+ identity)) 6)
(2 2 4 4 6 6)
```

However, we can use other Lisp functions besides these three:

```
> (mapcar (fn (list 1- identity 1+))
          '(1 2 3))
((0 1 2) (1 2 3) (2 3 4))
```

and the arguments in the `fn` expression may themselves be expressions:

```
> (remove-if (fn (or (and integerp oddp)
                         (and consp cdr)))
             '(1 (a b) c (d) 2 3.4 (e f g)))
(C (D) 2 3.4)
```

Making `fn` treat `compose` as a special case does not make it any more powerful. If you nest the arguments to `fn`, you get functional composition. For example,

```
(fn (list (1+ truncate)))
```

expands into:

```
#'(lambda (#:g1)
    (list ((lambda (#:g2) (1+ (truncate #:g2))) #:g1)))
```

which behaves like

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

The `fn` macro treats `compose` as a special case only to make such calls easier to read.

15.2 Recursion on Cdrs

Sections 5.5 and 5.6 showed how to write functions that build recursive functions. The following two sections show how anaphoric macros can provide a cleaner interface to the functions we defined there.

Section 5.5 showed how to define a flat list recurser builder called `lrec`. With `lrec` we can express a call to:

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

for e.g. `oddp` as:

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

Here macros could make life easier. How much do we really have to say to express recursive functions? If we can refer anaphorically to the current car of the list (as `it`) and the recursive call (as `rec`), we should be able to make do with something like:

```
(alrec (and (oddp it) rec) t)
```

Figure 15.2 contains the definition of the macro which will allow us to say this.

```
> (funcall (alrec (and (oddp it) rec) t)
           '(1 3 5))
T
```

```
(defmacro alrec (rec &optional base)
  "cltl2 version"
  (let ((gfn (gensym)))
    `(lrec #'(lambda (it ,gfn)
               (symbol-macrolet ((rec (funcall ,gfn)))
                 ,rec))
           ,base)))

(defmacro alrec (rec &optional base)
  "cltl1 version"
  (let ((gfn (gensym)))
    `(lrec #'(lambda (it ,gfn)
               (labels ((rec () (funcall ,gfn)))
                 ,rec))
           ,base)))

(defmacro on-cdrs (rec base &rest lsts)
  `(funcall (alrec ,rec #'(lambda () ,base)) ,@lsts))
```

Figure 15.2: Macros for list recursion.

The new macro works by transforming the expression given as the second argument into a function to be passed to `lrec`. Since the second argument may refer anaphorically to `it` or `rec`, in the macro expansion the body of the function must appear within the scope of bindings established for these symbols.

Figure 15.2 actually has two different versions of `alrec`. The version used in the preceding examples requires symbol macros (Section 7.11). Only recent versions of Common Lisp have symbol macros, so Figure 15.2 also contains a slightly less convenient version of `alrec` in which `rec` is defined as a local function. The price is that, as a function, `rec` would have to be enclosed within parentheses:

```
(alrec (and (oddp it) (rec)) t)
```

The original version is preferable in Common Lisp implementations which provide `symbol-macrolet`.

Common Lisp, with its separate name-space for functions, makes it awkward to use these recursion builders to define named functions:

```
(setf (symbol-function 'our-length)
      (alrec (1+ rec) 0))
```

```
(defun our-copy-list (lst)
  (on-cdrs (cons it rec) nil lst))

(defun our-remove-duplicates (lst)
  (on-cdrs (adjoin it rec) nil lst))

(defun our-find-if (fn lst)
  (on-cdrs (if (funcall fn it) it rec) nil lst))

(defun our-some (fn lst)
  (on-cdrs (or (funcall fn it) rec) nil lst))
```

Figure 15.3: Common Lisp functions defined with `on-cdrs`.

The final macro in Figure 15.2 is intended to make this more abstract. Using `on-cdrs` we could say instead:

```
(defun our-length (lst)
  (on-cdrs (1+ rec) 0 lst))

(defun our-every (fn lst)
  (on-cdrs (and (funcall fn it) rec) t lst))
```

Figure 15.3 shows some existing Common Lisp functions defined with the new macro. Expressed with `on-cdrs`, these functions are reduced to their most basic form, and we notice similarities between them which might not otherwise have been apparent.

Figure 15.4 contains some new utilities which can easily be defined with `on-cdrs`. The first three, `unions`, `intersections`, and `differences` implement set union, intersection, and complement, respectively. Common Lisp has built-in functions for these operations, but they can only take two lists at a time. Thus if we want to find the union of three lists we have to say:

```
> (union '(a b) (union '(b c) '(c d)))
(A B C D)
```

The new `unions` behaves like `union`, but takes an arbitrary number of arguments, so that we could say:

```
> (unions '(a b) '(b c) '(c d))
(D C A B)
```

```
(defun unions (&rest sets)
  (on-cdrs (union it rec) (car sets) (cdr sets)))

(defun intersections (&rest sets)
  (unless (some #'null sets)
    (on-cdrs (intersection it rec) (car sets) (cdr sets))))

(defun differences (set &rest outs)
  (on-cdrs (set-difference rec it) set outs))

(defun maxmin (args)
  (when args
    (on-cdrs (multiple-value-bind (mx mn) rec
               (values (max mx it) (min mn it)))
             (values (car args) (car args))
             (cdr args))))
```

Figure 15.4: New utilities defined with `on-cdrs`.

Like `union`, `unions` does not preserve the order of the elements in the initial lists.

The same relation holds between the Common Lisp `intersection` and the more general `intersections`. In the definition of this function, the initial test for null arguments was added for efficiency; it short-circuits the computation if one of the sets is empty.

Common Lisp also has a function called `set-difference`, which takes two lists and returns the elements of the first which are not in the second:

```
> (set-difference '(a b c d) '(a c))
(D B)
```

Our new version handles multiple arguments much as `-` does. For example, `(differences x y z)` is equivalent to `(set-difference x (unions y z))`, though without the consing that the latter would entail.

```
> (differences '(a b c d e) '(a f) '(d))
(B C E)
```

These set operators are intended only as examples. There is no real need for them, because they represent a degenerate case of list recursion already handled by the built-in `reduce`. For example, instead of

```
(unions ...)
```

you might as well say just

```
((lambda (&rest args) (reduce #'union args)) ...)
```

In the general case, `on-cdrs` is more powerful than `reduce`, however.

Because `rec` refers to a call instead of a value, we can use `on-cdrs` to create functions which return multiple values. The final function in Figure 15.4, `maxmin`, takes advantage of this possibility to find both the maximum and minimum elements in a single traversal of a list:

```
> (maxmin '(3 4 2 8 5 1 6 7))
8
1
```

It would also have been possible to use `on-cdrs` in some of the code which appears in later chapters. For example, `compile-cmds` (page 310)

```
(defun compile-cmds (cmds)
  (if (null cmds)
      'regs
      `(,@(car cmds) ,(compile-cmds (cdr cmds)))))
```

could have been defined as simply:

```
(defun compile-cmds (cmds)
  (on-cdrs `(,@it ,rec) 'regs cmds))
```

15.3 Recursion on Subtrees

What macros did for recursion on lists, they can also do for recursion on trees. In this section, we use macros to define cleaner interfaces to the tree recursers defined in Section 5.6.

In Section 5.6 we defined two tree recursion builders, `ttrav`, which always traverses the whole tree, and `trec` which is more complex, but allows you to control when recursion stops. Using these functions we could express `our-copy-tree`

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

as

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

and a call to `rfind-if`

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

for e.g. `oddp` as:

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

Anaphoric macros can make a better interface to `trec`, as they did for `lrec` in the previous section. A macro sufficient for the general case will have to be able to refer anaphorically to three things: the current tree, which we'll call `it`, the recursion down the left subtree, which we'll call `left`, and the recursion down the right subtree, which we'll call `right`. With these conventions established, we should be able to express the preceding functions in terms of a new macro thus:

```
(atrec (cons left right))

(atrec (or left right) (and (oddp it) it))
```

Figure 15.5 contains the definition of this macro.

In versions of Lisp which don't have `symbol-macrolet`, we can define `atrec` using the second definition in Figure 15.5. This version defines `left` and `right` as local functions, so `our-copy-tree` would have to be expressed as:

```
(atrec (cons (left) (right)))
```

For convenience, we also define a macro `on-trees`, which is analogous to `on-cdrs` from the previous section. Figure 15.6 shows the four functions from Section 5.6 defined with `on-trees`.

As noted in Chapter 5, functions built by the recurser generators defined in that chapter will not be tail-recursive. Using `on-cdrs` or `on-trees` to define a function will not necessarily yield the most efficient implementation. Like the underlying `trec` and `lrec`, these macros are mainly for use in prototypes and in parts of a program where efficiency is not paramount. However, the underlying idea of this chapter and Chapter 5 is that one can write function generators and put a clean macro interface on them. This same technique could equally well be used to build function generators which yielded particularly efficient code.

```
(defmacro atrec (rec &optional (base 'it))
  "cltl2 version"
  (let ((lfn (gensym)) (rfn (gensym)))
    `(trec #'(lambda (it ,lfn ,rfn)
               (symbol-macrolet ((left (funcall ,lfn))
                                 (right (funcall ,rfn)))
                 ,rec))
           #'(lambda (it) ,base))))

(defmacro atrec (rec &optional (base 'it))
  "cltl1 version"
  (let ((lfn (gensym)) (rfn (gensym)))
    `(trec #'(lambda (it ,lfn ,rfn)
               (labels ((left () (funcall ,lfn))
                        (right () (funcall ,rfn)))
                 ,rec))
           #'(lambda (it) ,base))))

(defmacro on-trees (rec base &rest trees)
  `(funcall (atrec ,rec ,base) ,@trees))
```

Figure 15.5: Macros for recursion on trees.

```
(defun our-copy-tree (tree)
  (on-trees (cons left right) it tree))

(defun count-leaves (tree)
  (on-trees (+ left (or right 1)) 1 tree))

(defun flatten (tree)
  (on-trees (nconc left right) (mklist it) tree))

(defun rfind-if (fn tree)
  (on-trees (or left right)
            (and (funcall fn it) it)
            tree))
```

Figure 15.6: Functions defined using `on-trees`.

```
(defconstant unforced (gensym))

(defstruct delay  forced closure)

(defmacro delay (expr)
  (let ((self (gensym)))
    `(let ((,self (make-delay :forced unforced)))
       (setf (delay-closure ,self)
             #'(lambda ()
                 (setf (delay-forced ,self) ,expr)))
       ,self)))

(defun force (x)
  (if (delay-p x)
      (if (eq (delay-forced x) unforced)
          (funcall (delay-closure x))
          (delay-forced x))
      x))
```

Figure 15.7: Implementation of `force` and `delay`.

15.4 Lazy Evaluation

Lazy evaluation means only evaluating an expression when you need its value. One way to use lazy evaluation is to build an object known as a *delay*. A delay is a placeholder for the value of some expression. It represents a promise to deliver the value of the expression if it is needed at some later time. Meanwhile, since the promise is a Lisp object, it can serve many of the purposes of the value it represents. And when the value of the expression is needed, the delay can return it.

Scheme has built-in support for delays. The Scheme operators `force` and `delay` can be implemented in Common Lisp as in Figure 15.7. A delay is represented as a two-part structure. The first field indicates whether the delay has been evaluated yet, and if it has, contains the value. The second field contains a closure which can be called to find the value that the delay represents. The macro `delay` takes an expression, and returns a delay representing its value:

```
> (let ((x 2))
    (setq d (delay (1+ x))))
#S(DELAY ...)
```

To call the closure within a delay is to *force* the delay. The function `force` takes any object: for ordinary objects it is the identity function, but for delays it is a demand for the value that the delay represents.

```
> (force 'a)
A
> (force d)
3
```

We use `force` whenever we are dealing with objects that might be delays. For example, if we are sorting a list which might contain delays, we would say:

```
(sort lst #'(lambda (x y) (> (force x) (force y))))
```

It's slightly inconvenient to use delays in this naked form. In a real application, they might be hidden beneath another layer of abstraction.