16

Macro-Defining Macros

Patterns in code often signal the need for new abstractions. This rule holds just as much for the code in macros themselves. When several macros have definitions of a similar form, we may be able to write a macro-defining macro to produce them. This chapter presents three examples of macro-defining macros: one to define abbreviations, one to define access macros, and a third to define anaphoric macros of the type described in Section 14.1.

16.1 Abbreviations

The simplest use of macros is as abbreviations. Some Common Lisp operators have rather long names. Ranking high among them (though by no means the longest) is `destructuring-bind`, which has 18 characters. A corollary of ○ Steele's principle (page 43) is that commonly used operators ought to have short names. ("We think of addition as cheap partly because we can notate it with a single character: '+'.") The built-in `destructuring-bind` macro introduces a new layer of abstraction, but the actual gain in brevity is masked by its long name:

```
(let ((a (car x)) (b (cdr x))) ...)
```

```
(destructuring-bind (a . b) x ...)
```

A program, like printed text, is easiest to read when it contains no more than about 70 characters per line. We begin at a disadvantage when the lengths of individual names are a quarter of that.

```
(defmacro abbrev (short long)
  `(defmacro ,short (&rest args)
     `(,',long ,@args)))

(defmacro abbrevs (&rest names)
  `(progn
     ,@(mapcar #'(lambda (pair)
                   `(abbrev ,@pair))
               (group names 2))))
```

Figure 16.1: Automatic definition of abbreviations.

Fortunately, in a language like Lisp you don't have to live with all the decisions of the designers. Having defined

```
(defmacro dbind (&rest args)
  `(destructuring-bind ,@args))
```

you need never use the long name again. Likewise for `multiple-value-bind`, which is longer and more frequently used.

```
(defmacro mvbind (&rest args)
  `(multiple-value-bind ,@args))
```

Notice how nearly identical are the definitions of `dbind` and `mvbind`. Indeed, this formula of `&rest` and comma-at will suffice to define an abbreviation for any function,[1] macro, or special form. Why crank out more definitions on the model of `mvbind` when we could have a macro do it for us?

To define a macro-defining macro we will often need nested backquotes. Nested backquotes are notoriously hard to understand. Eventually common cases will become familiar, but one should not expect to be able to look at an arbitrary backquoted expression and say what it will yield. It is not a fault in Lisp that this is so, any more than it is a fault of the notation that one can't just look at a complicated integral and know what its value will be. The difficulty is in the problem, not the notation.

However, as we do when finding integrals, we can break up the analysis of backquotes into small steps, each of which can easily be followed. Suppose we want to write a macro `abbrev`, which will allow us to define `mvbind` just by saying

[1] Though the abbreviation can't be passed to `apply` or `funcall`.

```
(abbrev mvbind multiple-value-bind)
```

Figure 16.1 contains a definition of this macro. Where did it come from? The definition of such a macro can be derived from a sample expansion. One expansion is:

```
(defmacro mvbind (&rest args)
  `(multiple-value-bind ,@args))
```

The derivation will be easier if we pull `multiple-value-bind` from within the backquote, because we know it will be an argument to the eventual macro. This yields the equivalent definition

```
(defmacro mvbind (&rest args)
  (let ((name 'multiple-value-bind))
    `(,name ,@args)))
```

Now we take this expression and turn it into a template. We affix a backquote, and replace the expressions which will vary, with variables.

```
`(defmacro ,short (&rest args)
   (let ((name ',long))
     `(,name ,@args)))
```

The final step is to simplify this expression by substituting `',long` for `name` within the inner backquote:

```
`(defmacro ,short (&rest args)
   `(,',long ,@args))
```

which yields the body of the macro defined in Figure 16.1.

Figure 16.1 also contains `abbrevs`, for cases where we want to define several abbreviations in one shot.

```
(abbrevs dbind  destructuring-bind
         mvbind multiple-value-bind
         mvsetq multiple-value-setq)
```

The user of `abbrevs` doesn't have to insert additional parentheses because `abbrevs` calls `group` (page 47) to group its arguments by twos. It's generally a good thing for macros to save users from typing logically unnecessary parentheses, and `group` will be useful to most such macros.

```
(defmacro propmacro (propname)
  `(defmacro ,propname (obj)
     `(get ,obj ',',propname)))

(defmacro propmacros (&rest props)
  `(progn
     ,@(mapcar #'(lambda (p) `(propmacro ,p))
               props)))
```

Figure 16.2: Automatic definition of access macros.

16.2 Properties

Lisp offers many ways to associate properties with objects. If the object in question can be represented as a symbol, one of the most convenient (though least efficient) ways is to use the symbol's property list. To describe the fact that an object *o* has a property *p*, the value of which is *v*, we modify the property list of *o*:

```
(setf (get o p) v)
```

So to say that `ball1` has `color` `red`, we say:

```
(setf (get 'ball1 'color) 'red)
```

If we're going to refer often to some property of objects, we can define a macro to retrieve it:

```
(defmacro color (obj)
  `(get ,obj 'color))
```

and thereafter use `color` in place of `get`:

```
> (color 'ball1)
RED
```

Since macro calls are transparent to `setf` (see Chapter 12) we can also say:

```
> (setf (color 'ball1) 'green)
GREEN
```

Such macros have the advantage of hiding the particular way in which the program represents the color of an object. Property lists are slow; a later version of the program might, for the sake of speed, represent color as a field in a structure, or an entry in a hash-table. When data is reached through a facade of macros like `color`, it becomes easy, even in a comparatively mature program, to make pervasive changes to the lowest-level code. If a program switches from using property lists to structures, nothing above the facade of access macros will have to be changed; none of the code which looks upon the facade need even be aware of the rebuilding going on behind it.

For the weight property, we can define a macro similar to the one written for color:

```
(defmacro weight (obj)
  `(get ,obj 'weight))
```

Like the abbreviations in the previous section, the definitions of of `color` and `weight` are nearly identical. Here `propmacro` (Figure 16.2) can play the same role as `abbrev` did.

A macro-defining macro can be designed by the same process as any other macro: look at the macro call, then its intended expansion, then figure out how to transform the former into the latter. We want

```
(propmacro color)
```

to expand into

```
(defmacro color (obj)
  `(get ,obj 'color))
```

Though this expression is itself a `defmacro`, we can still make a template of it, by backquoting it and putting comma'd parameter names in place of instances of `color`. As in the previous section, we begin by transforming it so that no instances of `color` are within existing backquotes:

```
(defmacro color (obj)
  (let ((p 'color))
    `(get ,obj ',p)))
```

Then we go ahead and make the template,

```
`(defmacro ,propname (obj)
   (let ((p ',propname))
     `(get ,obj ',p)))
```

which simplifies to ◦

```
`(defmacro ,propname (obj)
   `(get ,obj ',',propname))
```

For cases where a group of property-names all have to be defined as macros, there is `propmacros` (Figure 16.2), which expands into a series of individual calls to `propmacro`. Like `abbrevs`, this modest piece of code is actually a macro-defining-macro-defining macro.

Though this section dealt with property lists, the technique described here is a general one. We could use it to define access macros on data stored in any form.

16.3 Anaphoric Macros

Section 14.1 gave definitions of several anaphoric macros. When you use a macro like `aif` or `aand`, during the evaluation of some arguments the symbol `it` will be bound to the values returned by other ones. So instead of

```
(let ((res (complicated-query)))
  (if res
      (foo res)))
```

you can use just

```
(aif (complicated-query)
     (foo it))
```

and instead of

```
(let ((o (owner x)))
  (and o (let ((a (address o)))
           (and a (city a)))))
```

simply

```
(aand (owner x) (address it) (city it))
```

Section 14.1 presented seven anaphoric macros: `aif`, `awhen`, `awhile`, `acond`, `alambda`, `ablock`, and `aand`. These seven are by no means the only useful anaphoric macros of their type. In fact, we can define an anaphoric variant of just about any Common Lisp function or macro. Many of these macros will be like `mapcon`: rarely used, but indispensable when they are needed.

For example, we can define `a+` so that, as with `aand`, `it` is always bound to the value returned by the previous argument. The following function calculates the cost of dining out in Massachusetts:

```
(defmacro a+ (&rest args)
  (a+expand args nil))

(defun a+expand (args syms)
  (if args
      (let ((sym (gensym)))
        `(let* ((,sym ,(car args))
                (it ,sym))
           ,(a+expand (cdr args)
                      (append syms (list sym)))))
      `(+ ,@syms)))

(defmacro alist (&rest args)
  (alist-expand args nil))

(defun alist-expand (args syms)
  (if args
      (let ((sym (gensym)))
        `(let* ((,sym ,(car args))
                (it ,sym))
           ,(alist-expand (cdr args)
                          (append syms (list sym)))))
      `(list ,@syms)))
```

Figure 16.3: Definitions of `a+` and `alist`.

```
(defun mass-cost (menu-price)
  (a+ menu-price (* it .05) (* it 3)))
```

The Massachusetts Meals Tax is 5%, and residents often calculate the tip by tripling the tax. By this formula, the total cost of the broiled scrod at Dolphin Seafood is therefore:

```
> (mass-cost 7.95)
9.54
```

but this includes salad and a baked potato.

The macro `a+`, defined in Figure 16.3, relies on a recursive function, `a+expand`, to generate its expansion. The general strategy of `a+expand` is to cdr down the list of arguments in the macro call, generating a series of nested `let` expressions; each `let` leaves `it` bound to a different argument, but also binds a distinct gensym

```
(defmacro defanaph (name &optional calls)
   (let ((calls (or calls (pop-symbol name))))
    `(defmacro ,name (&rest args)
       (anaphex args (list ',calls)))))

(defun anaphex (args expr)
  (if args
      (let ((sym (gensym)))
        `(let* ((,sym ,(car args))
                (it ,sym))
           ,(anaphex (cdr args)
                     (append expr (list sym)))))
      expr))

(defun pop-symbol (sym)
  (intern (subseq (symbol-name sym) 1)))
```

Figure 16.4: Automatic definition of anaphoric macros.

to each argument. The expansion function accumulates a list of these gensyms, and when it reaches the end of the list of arguments it returns a + expression with the gensyms as the arguments. So the expression

```
(a+ menu-price (* it .05) (* it 3))
```

yields the macroexpansion:

```
(let* ((#:g2 menu-price) (it #:g2))
  (let* ((#:g3 (* it 0.05)) (it #:g3))
    (let* ((#:g4 (* it 3)) (it #:g4))
      (+ #:g2 #:g3 #:g4))))
```

Figure 16.3 also contains the definition of the analogous `alist`:

```
> (alist 1 (+ 2 it) (+ 2 it))
(1 3 5)
```

Once again, the definitions of `a+` and `alist` are almost identical. If we want to define more macros like them, these too will be mostly duplicate code. Why not have a program produce it for us? The macro `defanaph` in Figure 16.4 will do so. With `defanaph`, defining `a+` and `alist` is as simple as

```
(defanaph a+)
(defanaph alist)
```

The expansions of `a+` and `alist` so defined will be identical to the expansions made by the code in Figure 16.3. The macro-defining macro `defanaph` will create an anaphoric variant of anything whose arguments are evaluated according to the normal evaluation rule for functions. That is, `defanaph` will work for anything whose arguments are all evaluated, and evaluated left-to-right. So you couldn't use this version of `defanaph` to define `aif` or `awhile`, but you can use it to define an anaphoric variant of any function.

As `a+` called `a+expand` to generate its expansion, `defanaph` defines a macro which will call `anaphex` to do so. The generic expander `anaphex` differs from `a+expand` only in taking as an argument the function name to appear finally in the expansion. In fact, `a+` could now be defined:

```
(defmacro a+ (&rest args)
  (anaphex args '(+)))
```

Neither `anaphex` nor `a+expand` need have been defined as distinct functions: `anaphex` could have been defined with `labels` or `alambda` within `defanaph`. The expansion generators are here broken out as separate functions only for the sake of clarity.

By default, `defanaph` determines what to call in the expansion by pulling the first letter (presumably an `a`) from the front of its argument. (This operation is performed by `pop-symbol`.) If the user prefers to specify an alternate name, it can be given as an optional argument. Although `defanaph` can build anaphoric variants of all functions and some macros, it imposes some irksome restrictions:

1. It only works for operators whose arguments are all evaluated.
2. In the macroexpansion, `it` is always bound to successive arguments. In some cases—`awhen`, for example—we want `it` to stay bound to the value of the *first* argument.
3. It won't work for a macro like `setf`, which expects a generalized variable as its first argument.

Let's consider how to remove some of these restrictions. Part of the first problem can be solved by solving the second. To generate expansions for a macro like `aif`, we need a modified version of `anaphex` which only replaces the first argument in the macro call:

```
(defun anaphex2 (op args)
  `(let ((it ,(car args)))
     (,op it ,@(cdr args))))
```

This nonrecursive version of anaphex doesn't need to ensure that the macroexpansion will bind it to successive arguments, so it can generate an expansion which won't necessarily evaluate all the arguments in the macro call. Only the first argument must be evaluated, in order to bind it to its value. So aif could be defined as:

```
(defmacro aif (&rest args)
  (anaphex2 'if args))
```

This definition would differ from the original on page 191 only in the point where it would complain if aif were given the wrong number of arguments; for correct macro calls, the two generate identical expansions.

The third problem, that defanaph won't work with generalized variables, can be solved by using _f (page 173) in the expansion. Operators like setf can be handled by a variant of anaphex2 defined as follows:

```
(defun anaphex3 (op args)
  `(_f (lambda (it) (,op it ,@(cdr args))) ,(car args)))
```

This expander assumes that the macro call will have one or more arguments, the first of which will be a generalized variable. Using it we could define asetf thus:

```
(defmacro asetf (&rest args)
  (anaphex3 'setf args))
```

Figure 16.5 shows all three expander functions yoked together under the control of a single macro, the new defanaph. The user signals the type of macro expansion desired with the optional rule keyword parameter, which specifies the evaluation rule to be used for the arguments in the macro call. If this parameter is:

:all (the default) the macroexpansion will be on the model of alist. All the arguments in the macro call will be evaluated, with it always bound to the value of the previous argument.

:first the macroexpansion will be on the model of aif. Only the first argument will necessarily be evaluated, and it will be bound to its value.

:place the macroexpansion will be on the model of asetf. The first argument will be treated as a generalized variable, and it will be bound to its initial value.

Using the new defanaph, some of the previous examples would be defined as follows:

```
(defmacro defanaph (name &optional &key calls (rule :all))
  (let* ((opname (or calls (pop-symbol name)))
         (body (case rule
                 (:all   `(anaphex1 args '(,opname)))
                 (:first `(anaphex2 ',opname args))
                 (:place `(anaphex3 ',opname args)))))
    `(defmacro ,name (&rest args)
       ,body)))

(defun anaphex1 (args call)
  (if args
      (let ((sym (gensym)))
        `(let* ((,sym ,(car args))
                (it ,sym))
           ,(anaphex1 (cdr args)
                      (append call (list sym)))))
      call))

(defun anaphex2 (op args)
  `(let ((it ,(car args))) (,op it ,@(cdr args))))

(defun anaphex3 (op args)
  `(_f (lambda (it) (,op it ,@(cdr args))) ,(car args)))
```

Figure 16.5: More general `defanaph`.

```
(defanaph alist)
(defanaph aif :rule :first)
(defanaph asetf :rule :place)
```

One of the advantages of `asetf` is that it makes it possible to define a large class of macros on generalized variables without worrying about multiple evaluation. For example, we could define `incf` as:

```
(defmacro incf (place &optional (val 1))
  `(asetf ,place (+ it ,val)))
```

and, say, `pull` (page 173) as:

```
(defmacro pull (obj place &rest args)
  `(asetf ,place (delete ,obj it ,@args)))
```