7

Macros

Lisp's macro facility allows you to define operators that are implemented by transformation. The definition of a macro is essentially a function that generates Lisp code—a program that writes programs. From these small beginnings arise great possibilities, and also unexpected hazards. Chapters 7–10 form a tutorial on macros. This chapter explains how macros work, gives techniques for writing and testing them, and looks at the issue of macro style.

7.1 How Macros Work

Since macros can be called and return values, they tend to be associated with functions. Macro definitions sometimes resemble function definitions, and speaking informally, people call `do`, which is actually a macro, a "built-in function." But pushing the analogy too far can be a source of confusion. Macros work differently from normal functions, and knowing how and why macros are different is the key to using them correctly. A function produces results, but a macro produces *expressions*—which, when evaluated, produce results.

The best way to begin is to move straight into an example. Suppose we want to write a macro `nil!`, which sets its argument to `nil`. We want `(nil! x)` to have the same effect as `(setq x nil)`. We do it by defining `nil!` as a macro which turns instances of the first form into instances of the second.

```
> (defmacro nil! (var)
    (list 'setq var nil))
NIL!
```

Paraphrased in English, this definition tells Lisp: "Whenever you see an expression of the form `(nil! `*var*`)`, turn it into one of the form `(setq `*var*` nil)` before evaluating it."

The expression generated by the macro will be evaluated in place of the original macro call. A macro call is a list whose first element is the name of a macro. What happens when we type the macro call `(nil! x)` into the toplevel? Lisp notices that `nil!` is the name of a macro, and

1. builds the expression specified by the definition above, then

2. evaluates that expression in place of the original macro call.

The step of building the new expression is called *macroexpansion.* Lisp looks up the definition of `nil!`, which shows how to construct a replacement for the macro call. The definition of `nil!` is applied like a function to the *expressions* given as arguments in the macro call. It returns a list of three elements: `setq`, the expression given as the argument to the macro, and `nil`. In this case, the argument to `nil!` is `x`, and the macroexpansion is `(setq x nil)`.

After macroexpansion comes a second step, *evaluation.* Lisp evaluates the macroexpansion `(setq x nil)` as if you had typed that in the first place. Evaluation does not always come immediately after expansion, as it does at the toplevel. A macro call occurring in the definition of a function will be expanded when the function is compiled, but the expansion—or the object code which results from it—won't be evaluated until the function is called.

Many of the difficulties you might encounter with macros can be avoided by maintaining a sharp distinction between macroexpansion and evaluation. When writing macros, know which computations are performed during macroexpansion, and which during evaluation, for the two steps generally operate on objects of two different sorts. The macroexpansion step deals with expressions, and the evaluation step deals with their values.

Sometimes macroexpansion can be more complicated than it was in the case of `nil!`. The expansion of `nil!` was a call to a built-in special form, but sometimes the expansion of a macro will be yet another macro call, like a Russian doll which contains another doll inside it. In such cases, macroexpansion simply continues until it arrives at an expression which is no longer a macro call. The process can take arbitrarily many steps, so long as it terminates eventually.

Many languages offer some form of macro, but Lisp macros are singularly powerful. When a file of Lisp is compiled, a parser reads the source code and sends its output to the compiler. Here's the stroke of genius: the output of the parser consists of *lists of Lisp objects.* With macros, we can manipulate the program while it's in this intermediate form between parser and compiler. If necessary, these manipulations can be very extensive. A macro generating its expansion has at its disposition the full power of Lisp. Indeed, a macro is really a Lisp function—one which happens to return expressions. The definition of `nil!` contains a single call to `list`, but another macro might invoke a whole subprogram to generate its expansion.

Being able to change what the compiler sees is almost like being able to rewrite it. We can add any construct to the language that we can define by transformation into existing constructs.

7.2 Backquote

Backquote is a special version of quote which can be used to create templates for Lisp expressions. One of the most common uses of backquote is in macro definitions.

The backquote character, `‘`, is so named because it resembles a regular quote, `’`, reversed. When backquote alone is affixed to an expression, it behaves just like quote:

`‘(a b c)` is equal to `’(a b c)`.

Backquote becomes useful only when it appears in combination with comma, `,`, and comma-at, `,@`. If backquote makes a template, comma makes a slot within a template. A backquoted list is equivalent to a call to `list` with the elements quoted. That is,

`‘(a b c)` is equal to `(list ’a ’b ’c)`.

Within the scope of a backquote, a comma tells Lisp: “turn off the quoting.” When a comma appears before one of the elements of the list, it has the effect of cancelling out the quote that would have been put there. So

`‘(a ,b c ,d)` is equal to `(list ’a b ’c d)`.

Instead of the symbol `b`, its value is inserted into the resulting list. Commas work no matter how deeply they appear within a nested list,

```
> (setq a 1 b 2 c 3)
3
> ‘(a ,b c)
(A 2 C)
> ‘(a (,b c))
(A (2 C))
```

and they may even appear within quotes, or within quoted sublists:

```
> `(a b ,c (',(+ a b c)) (+ a b) 'c '((,a ,b)))
(A B 3 ('6) (+ A B) 'C '((1 2)))
```

One comma counteracts the effect of one backquote, so commas must match backquotes. Say that a comma is *surrounded* by a particular operator if the operator is prepended to the comma, or prepended to an expression which contains it. In `` `(,a ,(b `,c)))``, for example, the last comma is surrounded by one comma and two backquotes. The general rule is: a comma surrounded by *n* commas must be surrounded by at least *n*+1 backquotes. An obvious corollary is that commas may not appear outside of a backquoted expression. Backquotes and commas can be nested, so long as they obey the rule above. Any of the following expressions would generate an error if typed into the toplevel:

```
        ,x    `(a ,,b c)    `(a ,(b ,c) d)    `(,,`a)
```

Nested backquotes are only likely to be needed in macro-defining macros. Both topics are discussed in Chapter 16.

Backquote is usually used for making lists.[1] Any list generated by backquote can also be generated by using `list` and regular quotes. The advantage of backquote is just that it makes expressions easier to read, because a backquoted expression resembles the expression it will produce. In the previous section we defined `nil!` as:

```
(defmacro nil! (var)
  (list 'setq var nil))
```

With backquote the same macro can be defined as:

```
(defmacro nil! (var)
  `(setq ,var nil))
```

which in this case is not all that different. The longer the macro definition, however, the more important it is to use backquote. Figure 7.1 contains two possible definitions of `nif`, a macro which does a three-way numeric `if`.[2]

The first argument should evaluate to a number. Then the second, third, or fourth argument is evaluated, depending on whether the first was positive, zero, or negative:

```
> (mapcar #'(lambda (x)
              (nif x 'p 'z 'n))
          '(0 2.5 -8))
(Z P N)
```

With backquote:

```
(defmacro nif (expr pos zero neg)
  `(case (truncate (signum ,expr))
     (1 ,pos)
     (0 ,zero)
     (-1 ,neg)))
```

Without backquote:

```
(defmacro nif (expr pos zero neg)
  (list 'case
        (list 'truncate (list 'signum expr))
        (list 1 pos)
        (list 0 zero)
        (list -1 neg)))
```

Figure 7.1: A macro defined with and without backquote.

The two definitions in Figure 7.1 define the same macro, but the first uses backquote, while the second builds its expansion by explicit calls to `list`. From the first definition it's easy to see that `(nif x 'p 'z 'n)`, for example, expands into

```
(case (truncate (signum x))
  (1 'p)
  (0 'z)
  (-1 'n))
```

because the body of the macro definition looks just like the expansion it generates. To understand the second version, without backquote, you have to trace in your head the building of the expansion.

Comma-at, `,@`, is a variant of comma. It behaves like comma, with one difference: instead of merely inserting the value of the expression to which it is affixed, as comma does, comma-at *splices* it. Splicing can be thought of as inserting while removing the outermost level of parentheses:

```
> (setq b '(1 2 3))
(1 2 3)
```

[1]Backquote can also be used to create vectors, but this is rarely done in macro definitions.

[2]This macro is defined a little oddly to avoid using gensyms. A better definition is given on page 150.

```
> `(a ,b c)
(A (1 2 3) C)
> `(a ,@b c)
(A 1 2 3 C)
```

The comma causes the list `(1 2 3)` to be inserted in place of `b`, while the comma-at causes the elements of the list to be inserted there. There are some additional restrictions on the use of comma-at:

1. In order for its argument to be spliced, comma-at must occur within a sequence. It's an error to say something like `` `,@b `` because there is nowhere to splice the value of `b`.

2. The object to be spliced must be a list, unless it occurs last. The expression `` `(a ,@1) `` will evaluate to `(a . 1)`, but attempting to splice an atom into the middle of a list, as in `` `(a ,@1 b) ``, will cause an error.

Comma-at tends to be used in macros which take an indeterminate number of arguments and pass them on to functions or macros which also take an indeterminate number of arguments. This situation commonly arises when implementing implicit blocks. Common Lisp has several operators for grouping code into blocks, including `block`, `tagbody`, and `progn`. These operators rarely appear directly in source code; they are more often *implicit*—that is, hidden by macros.

An implicit block occurs in any built-in macro which can have a *body* of expressions. Both `let` and `cond` provide implicit `progn`, for example. The simplest built-in macro to do so is probably `when`:

```
(when (eligible obj)
  (do-this)
  (do-that)
  obj)
```

If `(eligible obj)` returns true, the remaining expressions will be evaluated, and the `when` expression as a whole will return the value of the last. As an example of the use of comma-at, here is one possible definition for `when`:

```
(defmacro our-when (test &body body)
  `(if ,test
       (progn
         ,@body)))
```

This definition uses an `&body` parameter (identical to `&rest` except for its effect on pretty-printing) to take in an arbitrary number of arguments, and a comma-at to splice them into a `progn` expression. In the macroexpansion of the call above, the three expressions in the body will appear within a single `progn`:

```
(if (eligible obj)
    (progn (do-this)
           (do-that)
           obj))
```

Most macros for iteration splice their arguments in a similar way.

The effect of comma-at can be achieved without using backquote. The expression `‘(a ,@b c)` is equal to `(cons ’a (append b (list ’c)))`, for example. Comma-at exists only to make such expression-generating expressions more readable.

Macro definitions (usually) generate lists. Although macro expansions could be built with the function `list`, backquote list-templates make the task much easier. A macro defined with `defmacro` and backquote will superficially resemble a function defined with `defun`. So long as you are not misled by the similarity, backquote makes macro definitions both easier to write and easier to read.

Backquote is so often used in macro definitions that people sometimes think of backquote as part of `defmacro`. The last thing to remember about backquote is that it has a life of its own, separate from its role in macros. You can use backquote anywhere sequences need to be built:

```
(defun greet (name)
  ‘(hello ,name))
```

7.3 Defining Simple Macros

In programming, the best way to learn is often to begin experimenting as soon as possible. A full theoretical understanding can come later. Accordingly, this section presents a way to start writing macros immediately. It works only for a narrow range of cases, but where applicable it can be applied quite mechanically. (If you’ve written macros before, you may want to skip this section.)

As an example, we consider how to write a variant of the the built-in Common Lisp function `member`. By default `member` uses `eql` to test for equality. If you want to test for membership using `eq`, you have to say so explicitly:

```
(member x choices :test #’eq)
```

If we did this a lot, we might want to write a variant of `member` which always used `eq`. Some earlier dialects of Lisp had such a function, called `memq`:

```
(memq x choices)
```

Ordinarily one would define `memq` as an inline function, but for the sake of example we will reincarnate it as a macro.

```
call:             (memq x choices)

expansion:  (member x choices :test #'eq)
```

Figure 7.2: Diagram used in writing `memq`.

The method: Begin with a typical call to the macro you want to define. Write it down on a piece of paper, and below it write down the expression into which it ought to expand. Figure 7.2 shows two such expressions. From the macro call, construct the parameter list for your macro, making up some parameter name for each of the arguments. In this case there are two arguments, so we'll have two parameters, and call them `obj` and `lst`:

```
(defmacro memq (obj lst)
```

Now go back to the two expressions you wrote down. For each argument in the macro call, draw a line connecting it with the place it appears in the expansion below. In Figure 7.2 there are two parallel lines. To write the body of the macro, turn your attention to the expansion. Start the body with a backquote. Now, begin reading the expansion expression by expression. Wherever you find a parenthesis that isn't part of an argument in the macro call, put one in the macro definition. So following the backquote will be a left parenthesis. For each expression in the expansion

1. If there is no line connecting it with the macro call, then write down the expression itself.
2. If there is a connection to one of the arguments in the macro call, write down the symbol which occurs in the corresponding position in the macro parameter list, preceded by a comma.

There is no connection to the first element, `member`, so we use `member` itself:

```
(defmacro memq (obj lst)
  `(member
```

However, `x` has a line leading to the first argument in the source expression, so we use in the macro body the first parameter, with a comma:

```
(defmacro memq (obj lst)
  `(member ,obj
```

Continuing in this way, the completed macro definition is:

```
(while hungry
  (stare-intently)
  (meow)
  (rub-against-legs))

(do ()
    ((not hungry))
  (stare-intently)
  (meow)
  (rub-against-legs))
```

Figure 7.3: Diagram used in writing `while`.

```
(defmacro memq (obj lst)
  `(member ,obj ,lst :test #'eq))
```

So far, we can only write macros which take a fixed number of arguments. Now suppose we want to write a macro `while`, which will take a test expression and some body of code, and loop through the code as long as the test expression returns true. Figure 7.3 contains an example of a `while` loop describing the behavior of a cat.

To write such a macro, we have to modify our technique slightly. As before, begin by writing down a sample macro call. From that, build the parameter list of the macro, but where you want to take an indefinite number of arguments, conclude with an `&rest` or `&body` parameter:

```
(defmacro while (test &body body)
```

Now write the desired expansion below the macro call, and as before draw lines connecting the arguments in the macro call to their position in the expansion. However, when you have a sequence of arguments which are going to be sucked into a single `&rest` or `&body` parameter, treat them as a group, drawing a single line for the whole sequence. Figure 7.3 shows the resulting diagram.

To write the body of the macro definition, proceed as before along the expansion. As well as the two previous rules, we need one more:

3. If there is a connection from a series of expressions in the expansion to a series of the arguments in the macro call, write down the corresponding `&rest` or `&body` parameter, preceded by a comma-at.

So the resulting macro definition will be:

```
(defmacro while (test &body body)
  `(do ()
       ((not ,test))
     ,@body))
```

To build a macro which can have a *body* of expressions, some parameter has to act as a funnel. Here multiple arguments in the macro call are joined together into `body`, and then broken up again when `body` is spliced into the expansion.

The approach described in this section enables us to write the simplest macros—those which merely shuffle their parameters. Macros can do a lot more than that. Section 7.7 will present examples where expansions can't be represented as simple backquoted lists, and to generate them, macros become programs in their own right.

7.4 Testing Macroexpansion

Having written a macro, how do we test it? A macro like `memq` is simple enough that one can tell just by looking at it what it will do. When writing more complicated macros, we have to be able to check that they are being expanded correctly.

Figure 7.4 shows a macro definition and two ways of looking at its expansion. The built-in function `macroexpand` takes an expression and returns its macroexpansion. Sending a macro call to `macroexpand` shows how the macro call will finally be expanded before being evaluated, but a complete expansion is not always what you want in order to test a macro. When the macro in question relies on other macros, they too will be expanded, so a complete macroexpansion can sometimes be difficult to read.

From the first expression shown in Figure 7.4, it's hard to tell whether or not `while` is expanding as intended, because the built-in `do` macro gets expanded, as well as the `prog` macro into which *it* expands. What we need is a way of seeing the result after only one step of expansion. This is the purpose of the built-in function `macroexpand-1`, shown in the second example; `macroexpand-1` stops after just one step, even if the expansion is still a macro call.

When we want to look at the expansion of a macro call, it will be a nuisance always to have to type

```
(pprint (macroexpand-1 '(or x y)))
```

Figure 7.5 defines a new macro which allows us to say instead:

```
(mac (or x y))
```

Typically you debug functions by calling them, and macros by expanding them. But since a macro call involves two layers of computation, there are two

```
> (defmacro while (test &body body)
    `(do ()
         ((not ,test))
       ,@body))
WHILE

> (pprint (macroexpand '(while (able) (laugh))))

(BLOCK NIL
  (LET NIL
    (TAGBODY
      #:G61
      (IF (NOT (ABLE)) (RETURN NIL))
      (LAUGH)
      (GO #:G61))))
T
> (pprint (macroexpand-1 '(while (able) (laugh))))

(DO NIL
    ((NOT (ABLE)))
  (LAUGH))
T
```

Figure 7.4: A macro and two depths of expansion.

```
(defmacro mac (expr)
  `(pprint (macroexpand-1 ',expr)))
```

Figure 7.5: A macro for testing macroexpansion.

points where things can go wrong. If a macro is misbehaving, most of the time you will be able to tell what's wrong just by looking at the expansion. Sometimes, though, the expansion will look fine and you'll want to evaluate it to see where the problems arise. If the expansion contains free variables, you may want to set some variables first. In some systems, you will be able to copy the expansion and paste it into the toplevel, or select it and choose `eval` from a menu. In the worst case you can set a variable to the list returned by `macroexpand-1`, then call `eval` on it:

```
> (setq exp (macroexpand-1 '(memq 'a '(a b c))))
(MEMBER (QUOTE A) (QUOTE (A B C)) :TEST (FUNCTION EQ))
> (eval exp)
(A B C)
```

Finally, macroexpansion is more than an aid in debugging, it's also a way of learning how to write macros. Common Lisp has over a hundred macros built-in, some of them quite complex. By looking at the expansions of these macros you will often be able to see how they were written.

7.5 Destructuring in Parameter Lists

Destructuring is a generalization of the sort of assignment[3] done by function calls. If you define a function of several arguments

```
(defun foo (x y z)
  (+ x y z))
```

then when the function is called

```
(foo 1 2 3)
```

the parameters of the function are assigned arguments in the call according to their position: `x` to `1`, `y` to `2`, and `z` to `3`. *Destructuring* describes the situation where this sort of positional assignment is done for arbitrary list structures, as well as flat lists like `(x y z)`.

The Common Lisp `destructuring-bind` macro (new in CLTL2) takes a pattern, an argument evaluating to a list, and a body of expressions, and evaluates the expressions with the parameters in the pattern bound to the corresponding elements of the list:

```
> (destructuring-bind (x (y) . z) '(a (b) c d)
    (list x y z))
(A B (C D))
```

This new operator and others like it form the subject of Chapter 18.

Destructuring is also possible in macro parameter lists. The Common Lisp `defmacro` allows parameter lists to be arbitrary list structures. When a macro call is expanded, components of the call will be assigned to the parameters as if by `destructuring-bind`. The built-in `dolist` macro takes advantage of such parameter list destructuring. In a call like:

[3]Destructuring is usually seen in operators which create bindings, rather than do assignments. However, conceptually destructuring is a way of assigning values, and would work just as well for existing variables as for new ones. That is, there is nothing to stop you from writing a destructuring `setq`.

```
(dolist (x '(a b c))
  (print x))
```

the expansion function must pluck `x` and `'(a b c)` from within the list given as the first argument. That can be done implicitly by giving `dolist` the appropriate parameter list:[4]

```
(defmacro our-dolist ((var list &optional result) &body body)
  `(progn
     (mapc #'(lambda (,var) ,@body)
           ,list)
     (let ((,var nil))
       ,result)))
```

In Common Lisp, macros like `dolist` usually enclose within a list the arguments not part of the body. Because it takes an optional `result` argument, `dolist` must enclose its first arguments in a distinct list anyway. But even if the extra list structure were not necessary, it would make calls to `dolist` easier to read. Suppose we want to define a macro `when-bind`, like `when` except that it binds some variable to the value returned by the test expression. This macro may be best implemented with a nested parameter list:

```
(defmacro when-bind ((var expr) &body body)
  `(let ((,var ,expr))
     (when ,var
       ,@body)))
```

and called as follows:

```
(when-bind (input (get-user-input))
  (process input))
```

instead of:

```
(let ((input (get-user-input)))
  (when input
    (process input)))
```

Used sparingly, parameter list destructuring can result in clearer code. At a minimum, it can be used in macros like `when-bind` and `dolist`, which take two or more arguments followed by a body of expressions.

[4]This version is written in this strange way to avoid using gensyms, which are not introduced till later.

```
(defmacro our-expander (name) `(get ,name 'expander))

(defmacro our-defmacro (name parms &body body)
  (let ((g (gensym)))
    `(progn
       (setf (our-expander ',name)
             #'(lambda (,g)
                 (block ,name
                   (destructuring-bind ,parms (cdr ,g)
                     ,@body))))
       ',name)))

(defun our-macroexpand-1 (expr)
  (if (and (consp expr) (our-expander (car expr)))
      (funcall (our-expander (car expr)) expr)
      expr))
```

Figure 7.6: A sketch of `defmacro`.

7.6 A Model of Macros

A formal description of what macros do would be long and confusing. Experienced programmers do not carry such a description in their heads anyway. It's more convenient to remember what `defmacro` does by imagining how it would be defined.

There is a long tradition of such explanations in Lisp. The *Lisp 1.5 Programmer's Manual*, first published in 1962, gives for reference a definition of ○ `eval` written in Lisp. Since `defmacro` is itself a macro, we can give it the same treatment, as in Figure 7.6. This definition uses several techniques which haven't been covered yet, so some readers may want to refer to it later.

The definition in Figure 7.6 gives a fairly accurate impression of what macros do, but like any sketch it is incomplete. It wouldn't handle the `&whole` keyword properly. And what `defmacro` really stores as the `macro-function` of its first argument is a function of *two* arguments: the macro call, and the lexical environment in which it occurs. However, these features are used only by the most esoteric macros. If you worked on the assumption that macros were implemented as in Figure 7.6, you would hardly ever go wrong. Every macro defined in this book would work, for example.

The definition in Figure 7.6 yields an expansion function which is a sharp-quoted lambda-expression. That should make it a closure: any free symbols in the macro definition should refer to variables in the environment where the `defmacro` occurred. So it should be possible to say this:

```
(let ((op 'setq))
  (defmacro our-setq (var val)
    (list op var val)))
```

As of CLTL2, it is. But in CLTL1, macro expanders were defined in the null lexical environment,[5] so in some old implementations this definition of `our-setq` will not work.

7.7 Macros as Programs

A macro definition need not be just a backquoted list. A macro is a function which transforms one sort of expression into another. This function can call `list` to generate its result, but can just as well invoke a whole subprogram consisting of hundreds of lines of code.

Section 7.3 gave an easy way of writing macros. Using this technique we can write macros whose expansions contain the same subexpressions as appear in the macro call. Unfortunately, only the simplest macros meet this condition. As a more complicated example, consider the built-in macro `do`. It isn't possible to write `do` as a macro which simply shuffles its parameters. The expansion has to build complex expressions which never appear in the macro call.

The more general approach to writing macros is to think about the sort of expression you want to be able to use, what you want it to expand into, and then write the *program* that will transform the first form into the second. Try expanding an example by hand, then look at what happens when one form is transformed into another. By working from examples you can get an idea of what will be required of your proposed macro.

Figure 7.7 shows an instance of `do`, and the expression into which it should expand. Doing expansions by hand is a good way to clarify your ideas about how a macro should work. For example, it may not be obvious until one tries writing the expansion that the local variables will have to be updated using `psetq`.

The built-in macro `psetq` (named for "parallel `setq`") behaves like `setq`, except that all its (even-numbered) arguments will be evaluated before any of the assignments are made. If an ordinary `setq` has more than two arguments, then the new value of the first argument is visible during the evaluation of the fourth:

[5]For an example of macro where this distinction matters, see the note on page 393.

```
(do ((w 3)
     (x 1 (1+ x))
     (y 2 (1+ y))
     (z))
    ((> x 10) (princ z) y)
  (princ x)
  (princ y))
```

should expand into something like

```
(prog ((w 3) (x 1) (y 2) (z nil))
   foo
    (if (> x 10)
        (return (progn (princ z) y)))
    (princ x)
    (princ y)
    (psetq x (1+ x) y (1+ y))
   (go foo))
```

Figure 7.7: Desired expansion of `do`.

```
> (let ((a 1))
    (setq a 2 b a)
    (list a b))
(2 2)
```

Here, because `a` is set first, `b` gets its new value, 2. A `psetq` is supposed to behave as if its arguments were assigned in parallel:

```
> (let ((a 1))
    (psetq a 2 b a)
    (list a b))
(2 1)
```

So here `b` gets the old value of `a`. The `psetq` macro is provided especially to support macros like `do`, which need to evaluate some of their arguments in parallel. (Had we used `setq`, we would have been defining `do*` instead.)

On looking at the expansion, it is also clear that we can't really use `foo` as the loop label. What if `foo` is also used as a loop label within the body of the `do`? Chapter 9 will deal with this problem in detail; for now, suffice it to say that instead of using `foo`, the macroexpansion must use a special anonymous symbol returned by the function `gensym`.

```
(defmacro our-do (bindforms (test &rest result) &body body)
  (let ((label (gensym)))
    `(prog ,(make-initforms bindforms)
       ,label
       (if ,test
           (return (progn ,@result)))
       ,@body
       (psetq ,@(make-stepforms bindforms))
       (go ,label))))

(defun make-initforms (bindforms)
  (mapcar #'(lambda (b)
              (if (consp b)
                  (list (car b) (cadr b))
                  (list b nil)))
          bindforms))

(defun make-stepforms (bindforms)
  (mapcan #'(lambda (b)
              (if (and (consp b) (third b))
                  (list (car b) (third b))
                 nil))
          bindforms))
```

Figure 7.8: Implementing `do`.

In order to write `do`, we consider what it would take to transform the first expression in Figure 7.7 into the second. To perform such a transformation, we need to do more than get the macro parameters into the right positions in some backquoted list. The initial `prog` has to be followed by a list of symbols and their initial bindings, which must be extracted from the second argument passed to the `do`. The function `make-initforms` in Figure 7.8 will return such a list. We also have to build a list of arguments for the `psetq`, but this case is more complicated because not all the symbols should be updated. In Figure 7.8, `make-stepforms` returns arguments for the `psetq`. With these two functions, the rest of the definition becomes fairly straightforward.

The code in Figure 7.8 isn't exactly the way `do` would be written in a real implementation. To emphasize the computation done during expansion, `make-initforms` and `make-stepforms` have been broken out as separate functions. In the future, such code will usually be left within the `defmacro` expression.

With the definition of this macro, we begin to see what macros can do. A macro has full access to Lisp to build an expansion. The code used to generate the expansion may be a program in its own right.

7.8 Macro Style

Good style means something different for macros. Style matters when code is either read by people or evaluated by Lisp. With macros, both of these activities take place under slightly unusual circumstances.

There are two different kinds of code associated with a macro definition: *expander code*, the code used by the macro to generate its expansion, and *expansion code*, which appears in the expansion itself. The principles of style are different for each. For programs in general, to have good style is to be clear and efficient. These principles are bent in opposite directions by the two types of macro code: expander code can favor clarity over efficiency, and expansion code can favor efficiency over clarity.

It's in compiled code that efficiency counts most, and in compiled code the macro calls have already been expanded. If the expander code was efficient, it made compilation go slightly faster, but it won't make any difference in how well the program runs. Since the expansion of macro calls tends to be only a small part of the work done by a compiler, macros which expand efficiently can't usually make much of a difference even in the compilation speed. So most of the time you can safely write expander code the way you would write a quick, first version of a program. If the expander code does unnecessary work or conses a lot, so what? Your time is better spent improving other parts of the program. Certainly if there's a choice between clarity and speed in expander code, clarity should prevail. Macro definitions are generally harder to read than function definitions, because they contain a mix of expressions evaluated at two different times. If this confusion can be reduced at the expense of efficiency in the expander code, it's a bargain.

For example, suppose that we wanted to define a version of `and` as a macro. Since `(and a b c)` is equivalent to `(if a (if b c))`, we can write `and` in terms of `if` as in the first definition in Figure 7.9. According to the standards by which we judge ordinary code, `our-and` is badly written. The expander code is recursive, and on each recursion finds the length of successive cdrs of the same list. If this code were going to be evaluated at runtime, it would be better to define this macro as in `our-andb`, which generates the same expansion with no wasted effort. However, as a macro definition `our-and` is just as good, if not better. It may be inefficient in calling `length` on each recursion, but its organization shows more clearly the way in which the expansion depends on the number of conjuncts.

```
(defmacro our-and (&rest args)
  (case (length args)
    (0 t)
    (1 (car args))
    (t `(if ,(car args)
            (our-and ,@(cdr args))))))

(defmacro our-andb (&rest args)
  (if (null args)
      t
      (labels ((expander (rest)
                 (if (cdr rest)
                     `(if ,(car rest)
                          ,(expander (cdr rest)))
                     (car rest))))
        (expander args))))
```

Figure 7.9: Two macros equivalent to `and`.

As always, there are exceptions. In Lisp, the distinction between compile-time and runtime is an artificial one, so any rule which depends upon it is likewise artificial. In some programs, compile-time is runtime. If you're writing a program whose main purpose is transformation and which uses macros to do it, then everything changes: the expander code becomes your program, and the expansion its output. Of course under such circumstances expander code should be written with efficiency in mind. However, it's safe to say that most expander code (a) only affects the speed of compilation, and (b) doesn't affect it very much—meaning that clarity should nearly always come first.

With expansion code, it's just the opposite. Clarity matters less for macro expansions because they are rarely looked at, especially by other people. The forbidden goto is not entirely forbidden in expansions, and the disparaged `setq` not quite so disparaged.

Proponents of structured programming disliked goto for what it did to *source* code. It was not machine language jump instructions that they considered harmful—so long as they were hidden by more abstract constructs in source code. Gotos are condemned in Lisp precisely because it's so easy to hide them: you can use `do` instead, and if you didn't have `do`, you could write it. Of course, if we're going to build new abstractions on top of goto, the goto is going to have to exist somewhere. Thus it is not necessarily bad style to use `go` in the definition of a new macro, if it can't be written in terms of some existing macro.

Similarly, `setq` is frowned upon because it makes it hard to see where a given variable gets its value. However, a macroexpansion is not going to be read by many people, so there is usually little harm in using `setq` on variables created within the macroexpansion. If you look at expansions of some of the built-in macros, you'll see quite a lot of `setq`s.

Several circumstances can make clarity more important in expansion code. If you're writing a complicated macro, you may end up reading the expansions after all, at least while you're debugging it. Also, in simple macros, only a backquote separates expander code from expansion code, so if such macros generate ugly expansions, the ugliness will be all too visible in your source code. However, even when the clarity of expansion code becomes an issue, efficiency should still predominate. Efficiency is important in most runtime code. Two things make it especially so for macro expansions: their ubiquity and their invisibility.

Macros are often used to implement general-purpose utilities, which are then called everywhere in a program. Something used so often can't afford to be inefficient. What looks like a harmless little macro could, after the expansion of all the calls to it, amount to a significant proportion of your program. Such a macro should receive more attention than its length would seem to demand. Avoid consing especially. A utility which conses unnecessarily can ruin the performance of an otherwise efficient program.

The other reason to look to the efficiency of expansion code is its very invisibility. If a function is badly implemented, it will proclaim this fact to you every time you look at its definition. Not so with macros. From a macro definition, inefficiency in the expansion code may not be evident, which is all the more reason to go looking for it.

7.9 Dependence on Macros

If you redefine a function, other functions which call it will automatically get the new version.[6] The same doesn't always hold for macros. A macro call which occurs in a function definition gets replaced by its expansion when the function is compiled. What if we redefine the macro after the calling function has been compiled? Since no trace of the original macro call remains, the expansion within the function can't be updated. The behavior of the function will continue to reflect the *old* macro definition:

```
> (defmacro mac (x) `(1+ ,x))
MAC
```

[6]Except functions compiled inline, which impose the same restrictions on redefinition as macros.

```
> (setq fn (compile nil '(lambda (y) (mac y))))
#<Compiled-Function BF7E7E>
> (defmacro mac (x) `(+ ,x 100))
MAC
> (funcall fn 1)
2
```

Similar problems occur if code which calls some macro is compiled before the macro itself is defined. CLTL2 says that "a macro definition must be seen by the compiler before the first use of the macro." Implementations vary in how they respond to violations of this rule. Fortunately it's easy to avoid both types of problem. If you adhere to the following two principles, you need never worry about stale or nonexistent macro definitions:

1. Define macros before functions (or macros) which call them.
2. When a macro is redefined, also recompile all the functions (or macros) which call it—directly or via other macros.

It has been suggested that all the macros in a program be put in a separate file, to make it easier to ensure that macro definitions are compiled first. That's taking things too far. It would be reasonable to put general-purpose macros like `while` into a separate file, but general-purpose utilities ought to be separated from the rest of a program anyway, whether they're functions or macros.

Some macros are written just for use in one specific part of a program, and these should be defined with the code which uses them. So long as the definition of each macro appears before any calls to it, your programs will compile fine. Collecting together all your macros, simply because they're macros, would do nothing but make your code harder to read.

7.10 Macros from Functions

This section describes how to transform functions into macros. The first step in translating a function into a macro is to ask yourself if you really need to do it. Couldn't you just as well declare the function `inline` (p. 26)?

There are some legitimate reasons to consider how to translate functions into macros, though. When you begin writing macros, it sometimes helps to think as if you were writing a function—an approach that usually yields macros which aren't quite right, but which at least give you something to work from. Another reason to look at the relationship between macros and functions is to see how they *differ*. Finally, Lisp programmers sometimes actually want to convert functions into macros.

The difficulty of translating a function into a macro depends on a number of properties of the function. The easiest class to translate are the functions which

1. Have a body consisting of a single expression.
2. Have a parameter list consisting only of parameter names.
3. Create no new variables (except the parameters).
4. Are not recursive (nor part of a mutually recursive group).
5. Have no parameter which occurs more than once in the body.
6. Have no parameter whose value is used before that of another parameter occurring before it in the parameter list.
7. Contain no free variables.

One function which meets these criteria is the built-in Common Lisp function `second`, which returns the second element of a list. It could be defined:

```
(defun second (x) (cadr x))
```

Where a function definition meets all the conditions above, you can easily transform it into an equivalent macro definition. Simply put a backquote in front of the body and a comma in front of each symbol which occurs in the parameter list:

```
(defmacro second (x) `(cadr ,x))
```

Of course, the macro can't be called under all the same conditions. It can't be given as the first argument to `apply` or `funcall`, and it should not be called in environments where the functions it calls have new local bindings. For ordinary in-line calls, though, the macro `second` should do the same thing as the function `second`.

The technique changes slightly when the body has more than one expression, because a macro must expand into a single expression. So if condition 1 doesn't hold, you have to add a `progn`. The function `noisy-second`:

```
(defun noisy-second (x)
  (princ "Someone is taking a cadr!")
  (cadr x))
```

could be duplicated by the following macro:

```
(defmacro noisy-second (x)
  `(progn
     (princ "Someone is taking a cadr!")
     (cadr ,x)))
```

When the function doesn't meet condition 2 because it has an `&rest` or `&body` parameter, the rules are the same, except that the parameter, instead of simply having a comma before it, must be spliced into a call to `list`. Thus

```
(defun sum (&rest args)
  (apply #'+ args))
```

becomes

```
(defmacro sum (&rest args)
  `(apply #'+ (list ,@args)))
```

which in this case would be better rewritten:

```
(defmacro sum (&rest args)
  `(+ ,@args))
```

When condition 3 doesn't hold—when new variables are created within the function body—the rule about the insertion of commas must be modified. Instead of putting commas before all symbols in the parameter list, we only put them before those which will refer to the parameters. For example, in:

```
(defun foo (x y z)
  (list x (let ((x y))
            (list x z))))
```

neither of the last two instances of `x` will refer to the parameter `x`. The second instance is not evaluated at all, and the third instance refers to a new variable established by the `let`. So only the first instance will get a comma:

```
(defmacro foo (x y z)
  `(list ,x (let ((x ,y))
              (list x ,z))))
```

Functions can sometimes be transformed into macros when conditions 4, 5 and 6 don't hold. However, these topics are treated separately in later chapters. The issue of recursion in macros is covered in Section 10.4, and the dangers of multiple and misordered evaluation in Sections 10.1 and 10.2, respectively.

As for condition 7, it is possible to simulate closures with macros, using a technique similar to the error described on page 37. But seeing as this is a low hack, not consonant with the genteel tone of this book, we shall not go into details.

7.11 Symbol Macros

CLTL2 introduced a new kind of macro into Common Lisp, the symbol-macro. While a normal macro call looks like a function call, a symbol-macro "call" looks like a symbol.

Symbol-macros can only be locally defined. The `symbol-macrolet` special form can, within its body, cause a lone symbol to behave like an expression:

```
> (symbol-macrolet ((hi (progn (print "Howdy")
                               1)))
    (+ hi 2))
"Howdy"
3
```

The body of the `symbol-macrolet` will be evaluated as if every `hi` in argument position had been replaced with `(progn (print "Howdy") 1)`.

Conceptually, symbol-macros are like macros that don't take any arguments. With no arguments, macros become simply textual abbreviations. This is not to say that symbol-macros are useless, however. They are used in Chapter 15 (page 205) and Chapter 18 (page 237), and in the latter instance they are indispensable.