17

Read-Macros

The three big moments in a Lisp expression's life are read-time, compile-time, and runtime. Functions are in control at runtime. Macros give us a chance to perform transformations on programs at compile-time. This chapter discusses read-macros, which do their work at read-time.

17.1 Macro Characters

In keeping with the general philosophy of Lisp, you have a great deal of control over the reader. Its behavior is controlled by properties and variables that can all be changed on the fly. The reader can be programmed at several levels. The easiest way to change its behavior is by defining new macro characters.

A *macro character* is a character which exacts special treatment from the Lisp reader. A lower-case `a`, for example, is ordinarily handled just like a lower-case `b`, but a left parenthesis is something different: it tells Lisp to begin reading a list. Each such character has a function associated with it that tells the Lisp reader what to do when the character is encountered. You can change the function associated with an existing macro character, or define new macro characters of your own.

The built-in function `set-macro-character` provides one way to define read-macros. It takes a character and a function, and thereafter when `read` encounters the character, it returns the result of calling the function.

One of the oldest read-macros in Lisp is `'`, the quote. You could do without `'` by always writing `(quote a)` instead of `'a`, but this would be tiresome and would make your code harder to read. The quote read-macro makes it possible to use `'a` as an abbreviation for `(quote a)`. We could define it as in Figure 17.1.

```
(set-macro-character #\'
  #'(lambda (stream char)
      (list 'quote (read stream t nil t))))
```

Figure 17.1: Possible definition of '.

When `read` encounters an instance of `'` in a normal context (e.g. not in `"a'b"` or `|a'b|`), it will return the result of calling this function on the current stream and character. (The function ignores this second parameter, which will always be the quote character.) So when `read` sees `'a`, it will return `(quote a)`.

The last three arguments to `read` control respectively whether encountering an end-of-file should cause an error, what value to return otherwise, and whether the call to `read` occurs within a call to `read`. In nearly all read-macros, the second and fourth arguments should be `t`, and the third argument is therefore irrelevant.

Read-macros and ordinary macros are both functions underneath. And like the functions that generate macro expansions, the functions associated with macro characters shouldn't have side-effects, except on the stream from which they read. Common Lisp explicitly makes no guarantees about when, or how often, the function associated with a read-macro will be called. (See CLTL2, p. 543.)

Macros and read-macros see your program at different stages. Macros get hold of the program when it has already been parsed into Lisp objects by the reader, and read-macros operate on a program while it is still text. However, by invoking `read` on this text, a read-macro can, if it chooses, get parsed Lisp objects as well. Thus read-macros are at least as powerful as ordinary macros.

Indeed, read-macros are more powerful in at least two ways. A read-macro affects everything read by Lisp, while a macro will only be expanded in code. And since read-macros generally call `read` recursively, an expression like

```
''a
```

becomes

```
(quote (quote a))
```

whereas if we had tried to define an abbreviation for `quote` using a normal macro,

```
(defmacro q (obj)
  `(quote ,obj))
```

```
(set-dispatch-macro-character #\# #\?
  #'(lambda (stream char1 char2)
      `#'(lambda (&rest ,(gensym))
           ,(read stream t nil t))))
```

Figure 17.2: A read-macro for constant functions.

it would work in isolation,

```
> (eq 'a (q a))
T
```

but not when nested. For example,

```
(q (q a))
```

would expand into

```
(quote (q a))
```

17.2 Dispatching Macro Characters

The sharp-quote, like other read-macros beginning with `#`, is an example of a subspecies called *dispatching* read-macros. These appear as two characters, the first of which is called the dispatching character. The purpose of such read-macros is simply to make the most of the ASCII character set; one can only have so many one-character read-macros.

You can (with `make-dispatch-macro-character`) define your own dispatching macro characters, but since `#` is already defined as one, you may as well use it. Some combinations beginning with `#` are explicitly reserved for your use; others are available in that they do not yet have a predefined meaning in Common Lisp. The complete list appears in CLTL2, p. 531.

New dispatching macro character combinations can be defined by calling the function `set-dispatch-macro-character`, like `set-macro-character` except that it takes two character arguments. One of the combinations reserved to the programmer is `#?`. Figure 17.2 shows how to define this combination as a read-macro for constant functions. Now `#?2` will be read as a function which takes any number of arguments and returns 2. For example:

```
> (mapcar #?2 '(a b c))
(2 2 2)
```

```
(set-macro-character #\] (get-macro-character #\)))

(set-dispatch-macro-character #\# #\[
  #'(lambda (stream char1 char2)
      (let ((accum nil)
            (pair (read-delimited-list #\] stream t)))
        (do ((i (ceiling (car pair)) (1+ i)))
            ((> i (floor (cadr pair)))
             (list 'quote (nreverse accum)))
          (push i accum)))))
```

Figure 17.3: A read-macro defining delimiters.

This example makes the new operator look rather pointless, but in programs that use a lot of functional arguments, constant functions are often needed. In fact, some dialects provide a built-in function called `always` for defining them.

Note that it is perfectly ok to use macro characters in the definition of this macro character: as with any Lisp expression, they disappear when the definition is read. It is also fine to use macro-characters after the `#?`. The definition of `#?` calls `read`, so macro-characters like `'` and `#'` behave as usual:

```
> (eq (funcall #?'a) 'a)
T
> (eq (funcall #?#'oddp) (symbol-function 'oddp))
T
```

17.3 Delimiters

After simple macro characters, the most commonly defined macro characters are list delimiters. Another character combination reserved for the user is `#[`. Figure 17.3 gives an example of how this character might be defined as a more elaborate kind of left parenthesis. It defines an expression of the form `#[x y]` to read as a list of all the integers between `x` and `y`, inclusive:

```
> #[2 7]
(2 3 4 5 6 7)
```

The only new thing about this read-macro is the call to `read-delimited-list`, a built-in function provided just for such cases. Its first argument is the character to treat as the end of the list. For `]` to be recognized as a delimiter, it must first be given this role, hence the preliminary call to `set-macro-character`.

```
(defmacro defdelim (left right parms &body body)
  `(ddfn ,left ,right #'(lambda ,parms ,@body)))

(let ((rpar (get-macro-character #\) )))
  (defun ddfn (left right fn)
    (set-macro-character right rpar)
    (set-dispatch-macro-character #\# left
      #'(lambda (stream char1 char2)
          (apply fn
                 (read-delimited-list right stream t))))))
```

Figure 17.4: A macro for defining delimiter read-macros.

Most potential delimiter read-macro definitions will duplicate a lot of the code in Figure 17.3. A macro could put a more abstract interface on all this machinery. Figure 17.4 shows how we might define a utility for defining delimiter read-macros. The `defdelim` macro takes two characters, a parameter list, and a body of code. The parameter list and the body of code implicitly define a function. A call to `defdelim` defines the first character as a dispatching read-macro which reads up to the second, then returns the result of applying this function to what it read.

Incidentally, the body of the function in Figure 17.3 also cries out for a utility—for one we have already defined, in fact: `mapa-b`, from page 54. Using `defdelim` and `mapa-b`, the read-macro defined in Figure 17.3 could now be written:

```
(defdelim #\[ #\] (x y)
  (list 'quote (mapa-b #'identity (ceiling x) (floor y))))
```

Another useful delimiter read-macro would be one for functional composition. Section 5.4 defined an operator for functional composition:

```
> (let ((f1 (compose #'list #'1+))
        (f2 #'(lambda (x) (list (1+ x)))))
    (equal (funcall f1 7) (funcall f2 7)))
T
```

When we are composing built-in functions like `list` and `1+`, there is no reason to wait until runtime to evaluate the call to `compose`. Section 5.7 suggested an alternative; by prefixing the sharp-dot read-macro to a `compose` expression,

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

```
(defdelim #\{ #\} (&rest args)
  `(fn (compose ,@args)))
```

Figure 17.5: A read-macro for functional composition.

we could cause it to be evaluated at read-time.

Here we show a similar but cleaner solution. The read-macro in Figure 17.5 defines an expression of the form `#{`$f_1\ f_2 \ldots f_n$`}` to read as the composition of $f_1, f_2, \ldots, f_n$. Thus:

```
> (funcall #{list 1+} 7)
(8)
```

It works by generating a call to `fn` (page 202), which will create the function at compile-time.

17.4 When What Happens

Finally, it might be useful to clear up a possibly confusing issue. If read-macros are invoked before ordinary macros, how is it that macros can expand into expressions which contain read-macros? For example, the macro:

```
(defmacro quotable ()
  '(list 'able))
```

generates an expansion with a quote in it. Or does it? In fact, what happens is that both quotes in the definition of this macro are expanded when the `defmacro` expression is read, yielding

```
(defmacro quotable ()
  (quote (list (quote able))))
```

Usually, there is no harm in acting as if macroexpansions could contain read-macros, because the definition of a read-macro will not (or should not) change between read-time and compile-time.