8

When to Use Macros

How do we know whether a given function should really be a function, rather than a macro? Most of the time there is a clear distinction between the cases which call for macros and those which don't. By default we should use functions: it is inelegant to use a macro where a function would do. We should use macros only where they bring us some specific advantage.

When do macros bring advantages? That is the subject of this chapter. Usually the question is not one of advantage, but necessity. Most of the things we do with macros, we could not do with functions. Section 8.1 lists the kinds of operators ○ which can only be implemented as macros. However, there is also a small (but interesting) class of borderline cases, in which an operator might justifiably be written as a function or a macro. For these situations, Section 8.2 gives the arguments for and against macros. Finally, having considered what macros are capable of doing, we turn in Section 8.3 to a related question: what kinds of things do people do with them?

8.1 When Nothing Else Will Do

It's a general principle of good design that if you find similar code appearing at several points in a program, you should write a subroutine and replace the similar sequences of code with calls to the subroutine. When we apply this principle to Lisp programs, we have to decide whether the "subroutine" should be a function or a macro.

In some cases it's easy to decide to write a macro instead of a function, because only a macro can do what's needed. A function like `1+` could conceivably

be written as either a function or a macro:

```
(defun 1+ (x) (+ 1 x))

(defmacro 1+ (x) `(+ 1 ,x))
```

But `while`, from Section 7.3, could only be defined as a macro:

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

There is no way to duplicate the behavior of this macro with a function. The definition of `while` splices the expressions passed as `body` into the body of a `do`, where they will be evaluated only if the `test` expression returns `nil`. No function could do that; in a function call, all the arguments are evaluated before the function is even invoked.

When you do need a macro, what do you need from it? Macros can do two things that functions can't: they can control (or prevent) the evaluation of their arguments, and they are expanded right into the calling context. Any application which requires macros requires, in the end, one or both of these properties.

The informal explanation that "macros don't evaluate their arguments" is slightly wrong. It would be more precise to say that macros *control* the evaluation of the arguments in the macro call. Depending on where the argument is placed in the macro's expansion, it could be evaluated once, many times, or not at all. Macros use this control in four major ways:

1. *Transformation.* The Common Lisp `setf` macro is one of a class of macros which pick apart their arguments before evaluation. A built-in access function will often have a converse whose purpose is to set what the access function retrieves. The converse of `car` is `rplaca`, of `cdr`, `rplacd`, and so on. With `setf` we can use calls to such access functions as if they were variables to be set, as in `(setf (car x) 'a)`, which could expand into `(progn (rplaca x 'a) 'a)`.

   To perform this trick, `setf` has to look inside its first argument. To know that the case above requires `rplaca`, `setf` must be able to see that the first argument is an expression beginning with `car`. Thus `setf`, and any other operator which transforms its arguments, must be written as a macro.

2. *Binding.* Lexical variables must appear directly in the source code. The first argument to `setq` is not evaluated, for example, so anything built on `setq` must be a macro which expands into a `setq`, rather than a function which

calls it. Likewise for operators like `let`, whose arguments are to appear as parameters in a `lambda` expression, for macros like `do` which expand into `let`s, and so on. Any new operator which is to alter the lexical bindings of its arguments must be written as a macro.

3. *Conditional evaluation.* All the arguments to a function are evaluated. In constructs like `when`, we want some arguments to be evaluated only under certain conditions. Such flexibility is only possible with macros.

4. *Multiple evaluation.* Not only are the arguments to a function all evaluated, they are all evaluated exactly once. We need a macro to define a construct like `do`, where certain arguments are to be evaluated repeatedly.

There are also several ways to take advantage of the inline expansion of macros. It's important to emphasize that the expansions thus appear in the lexical context of the macro call, since two of the three uses for macros depend on that fact. They are:

5. *Using the calling environment.* A macro can generate an expansion containing a variable whose binding comes from the context of the macro call. The behavior of the following macro:

```
(defmacro foo (x)
  `(+ ,x y))
```

   depends on the binding of `y` where `foo` is called.

   This kind of lexical intercourse is usually viewed more as a source of contagion than a source of pleasure. Usually it would be bad style to write such a macro. The ideal of functional programming applies as well to macros: the preferred way to communicate with a macro is through its parameters. Indeed, it is so rarely necessary to use the calling environment that most of the time it happens, it happens by mistake. (See Chapter 9.) Of all the macros in this book, only the continuation-passing macros (Chapter 20) and some parts of the ATN compiler (Chapter 23) use the calling environment in this way.

6. *Wrapping a new environment.* A macro can also cause its arguments to be evaluated in a new lexical environment. The classic example is `let`, which could be implemented as a macro on `lambda` (page 144). Within the body of an expression like `(let ((y 2)) (+ x y))`, `y` will refer to a new variable.

7. *Saving function calls.* The third consequence of the inline insertion of macroexpansions is that in compiled code there is no overhead associated with a macro call. By runtime, the macro call has been replaced by its expansion. (The same is true in principle of functions declared `inline`.)

Significantly, cases 5 and 6, when unintentional, constitute the problem of variable capture, which is probably the worst thing a macro writer has to fear. Variable capture is discussed in Chapter 9.

Instead of seven ways of using macros, it might be better to say that there are six and a half. In an ideal world, all Common Lisp compilers would obey `inline` declarations, and saving function calls would be a task for inline functions, not macros. An ideal world is left as an exercise to the reader.

8.2 Macro or Function?

The previous section dealt with the easy cases. Any operator that needs access to its parameters before they are evaluated should be written as a macro, because there is no other choice. What about those operators which could be written either way? Consider for example the operator `avg`, which returns the average of its arguments. It could be defined as a function

```
(defun avg (&rest args)
  (/ (apply #'+ args) (length args)))
```

but there is a good case for defining it as a macro,

```
(defmacro avg (&rest args)
  `(/ (+ ,@args) ,(length args)))
```

because the function version would entail an unnecessary call to `length` each time `avg` was called. At compile-time we may not know the values of the arguments, but we do know how many there are, so the call to `length` could just as well be made then. Here are several points to consider when we face such choices:

THE PROS

1. *Computation at compile-time.* A macro call involves computation at two times: when the macro is expanded, and when the expansion is evaluated. All the macroexpansion in a Lisp program is done when the program is compiled, and every bit of computation which can be done at compile-time is one bit that won't slow the program down when it's running. If an operator could be written to do some of its work in the macroexpansion stage, it will be more efficient to make it a macro, because whatever work a

smart compiler can't do itself, a function has to do at runtime. Chapter 13 describes macros like `avg` which do some of their work during the expansion phase.

2. *Integration with Lisp.* Sometimes, using macros instead of functions will make a program more closely integrated with Lisp. Instead of writing a program to solve a certain problem, you may be able to use macros to transform the problem into one that Lisp already knows how to solve. This approach, when possible, will usually make programs both smaller and more efficient: smaller because Lisp is doing some of your work for you, and more efficient because production Lisp systems generally have had more of the fat sweated out of them than user programs. This advantage appears mostly in embedded languages, which are described starting in Chapter 19.

3. *Saving function calls.* A macro call is expanded right into the code where it appears. So if you write some frequently used piece of code as a macro, you can save a function call every time it's used. In earlier dialects of Lisp, programmers took advantage of this property of macros to save function calls at runtime. In Common Lisp, this job is supposed to be taken over by functions declared `inline`.

   By declaring a function to be `inline`, you ask for it to be compiled right into the calling code, just like a macro. However, there is a gap between theory and practice here; CLTL2 (p. 229) says that "a compiler is free to ignore this declaration," and some Common Lisp compilers do. It may still be reasonable to use macros to save function calls, if you are compelled to use such a compiler.

In some cases, the combined advantages of efficiency and close integration with Lisp can create a strong argument for the use of macros. In the query compiler of Chapter 19, the amount of computation which can be shifted forward to compile-time is so great that it justifies turning the whole program into a single giant macro. Though done for speed, this shift also brings the program closer to Lisp: in the new version, it's easier to use Lisp expressions—arithmetic expressions, for example—within queries.

THE CONS

4. *Functions are data,* while macros are more like instructions to the compiler. Functions can be passed as arguments (e.g. to `apply`), returned by functions, or stored in data structures. None of these things are possible with macros.

   In some cases, you can get what you want by enclosing the macro call within a lambda-expression. This works, for example, if you want to `apply` or `funcall` certain macros:

```
> (funcall #'(lambda (x y) (avg x y)) 1 3)
2
```

However, this is an inconvenience. It doesn't always work, either: even if, like `avg`, the macro has an `&rest` parameter, there is no way to pass it a varying number of arguments.

5. *Clarity of source code.* Macro definitions can be harder to read than the equivalent function definitions. So if writing something as a macro would only make a program marginally better, it might be better to use a function instead.

6. *Clarity at runtime.* Macros are sometimes harder to debug than functions. If you get a runtime error in code which contains a lot of macro calls, the code you see in the backtrace could consist of the expansions of all those macro calls, and may bear little resemblance to the code you originally wrote.

   And because macros disappear when expanded, they are not accountable at runtime. You can't usually use `trace` to see how a macro is being called. If it worked at all, `trace` would show you the call to the macro's expander function, not the macro call itself.

7. *Recursion.* Using recursion in macros is not so simple as it is in functions. Although the expansion function of a macro may be recursive, the expansion itself may not be. Section 10.4 deals with the subject of recursion in macros.

All these considerations have to be balanced against one another in deciding when to use macros. Only experience can tell which will predominate. However, the examples of macros which appear in later chapters cover most of the situations in which macros are useful. If a potential macro is analogous to one given here, then it is probably safe to write it as such.

Finally, it should be noted that clarity at runtime (point 6) rarely becomes an issue. Debugging code which uses a lot of macros will not be as difficult as you might expect. If macro definitions were several hundred lines long, it might be unpleasant to debug their expansions at runtime. But utilities, at least, tend to be written in small, trusted layers. Generally their definitions are less than 15 lines long. So even if you are reduced to poring over backtraces, such macros will not cloud your view very much.

8.3 Applications for Macros

Having considered what can be done with macros, the next question to ask is: in what sorts of applications can we use them? The closest thing to a general

description of macro use would be to say that they are used mainly for syntactic transformations. This is not to suggest that the scope for macros is restricted. Since Lisp programs are made from[1] lists, which are Lisp data structures, "syntactic transformation" can go a long way indeed. Chapters 19–24 present whole programs whose purpose could be described as syntactic transformation, and which are, in effect, all macro.

Macro applications form a continuum between small general-purpose macros like `while`, and the large, special-purpose macros defined in the later chapters. On one end are the *utilities*, the macros resembling those that every Lisp has built-in. They are usually small, general, and written in isolation. However, you can write utilities for specific classes of programs too, and when you have a collection of macros for use in, say, graphics programs, they begin to look like a programming language for graphics. At the far end of the continuum, macros allow you to write whole programs in a language distinctly different from Lisp. Macros used in this way are said to implement *embedded languages*.

Utilities are the first offspring of the bottom-up style. Even when a program is too small to be built in layers, it may still benefit from additions to the lowest layer, Lisp itself. The utility `nil!`, which sets its argument to `nil`, could not be defined except as a macro:

```
(defmacro nil! (x)
  `(setf ,x nil))
```

Looking at `nil!`, one is tempted to say that it doesn't *do* anything, that it merely saves typing. True, but all any macro does is save typing. If one wants to think of it in these terms, the job of a compiler is to save typing in machine language. The value of utilities should not be underestimated, because their effect is cumulative: several layers of simple macros can make the difference between an elegant program and an incomprehensible one.

Most utilities are patterns embodied. When you notice a pattern in your code, consider turning it into a utility. Patterns are just the sort of thing computers are good at. Why should you bother following them when you could have a program do it for you? Suppose that in writing some program you find yourself using in many different places `do` loops of the same general form:

```
(do ()
    ((not ⟨condition⟩))
  .  ⟨body of code⟩)
```

[1] *Made from*, in the sense that lists are the input to the compiler. Functions are no longer *made of* lists, as they used to be in some earlier dialects.

When you find a pattern repeated through your code, that pattern often has a name. The name of this pattern is *while*. If we want to provide it in a new utility, we will have to use a macro, because we need conditional and repeated evaluation. If we define `while` using this definition from page 91:

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

then we can replace the instances of the pattern with

```
(while ⟨condition⟩
  .  ⟨body of code⟩)
```

Doing so will make the code shorter and also make it declare in a clearer voice what it's doing.

The ability to transform their arguments makes macros useful in writing interfaces. The appropriate macro will make it possible to type a shorter, simpler expression where a long or complex one would have been required. Although graphic interfaces decrease the need to write such macros for end users, programmers use this type of macro as much as ever. The most common example is `defun`, which makes the binding of functions resemble, on the surface, a function definition in a language like Pascal or C. Chapter 2 mentioned that the following two expressions have approximately the same effect:

```
(defun foo (x) (* x 2))

(setf (symbol-function 'foo)
      #'(lambda (x) (* x 2)))
```

Thus `defun` can be implemented as a macro which turns the former into the latter. We could imagine it written as follows:[2]

```
(defmacro our-defun (name parms &body body)
  `(progn
     (setf (symbol-function ',name)
           #'(lambda ,parms (block ,name ,@body)))
     ',name))
```

Macros like `while` and `nil!` could be described as general-purpose utilities. Any Lisp program might use them. But particular domains can have their utilities

[2] For clarity, this version ignores all the bookkeeping that `defun` must perform.

```
(defun move-objs (objs dx dy)
  (multiple-value-bind (x0 y0 x1 y1) (bounds objs)
    (dolist (o objs)
      (incf (obj-x o) dx)
      (incf (obj-y o) dy))
    (multiple-value-bind (xa ya xb yb) (bounds objs)
      (redraw (min x0 xa) (min y0 ya)
              (max x1 xb) (max y1 yb)))))

(defun scale-objs (objs factor)
  (multiple-value-bind (x0 y0 x1 y1) (bounds objs)
    (dolist (o objs)
      (setf (obj-dx o) (* (obj-dx o) factor)
            (obj-dy o) (* (obj-dy o) factor)))
    (multiple-value-bind (xa ya xb yb) (bounds objs)
      (redraw (min x0 xa) (min y0 ya)
              (max x1 xb) (max y1 yb)))))
```

Figure 8.1: Original move and scale.

as well. There is no reason to suppose that base Lisp is the only level at which you have a programming language to extend. If you're writing a CAD program, for example, the best results will sometimes come from writing it in two layers: a language (or if you prefer a more modest term, a toolkit) for CAD programs, and in the layer above, your particular application.

Lisp blurs many distinctions which other languages take for granted. In other languages, there really are conceptual distinctions between compile-time and runtime, program and data, language and program. In Lisp, these distinctions exist only as conversational conventions. There is no line dividing, for example, language and program. You can draw the line wherever suits the problem at hand. So it really is no more than a question of terminology whether to call an underlying layer of code a toolkit or a language. One advantage of considering it as a language is that it suggests you can extend this language, as you do Lisp, with utilities.

Suppose we are writing an interactive 2D drawing program. For simplicity, we will assume that the only objects handled by the program are line segments, represented as an origin $\langle x,y \rangle$ and a vector $\langle dx,dy \rangle$. One of the things such a program will have to do is slide groups of objects. This is the purpose of the function `move-objs` in Figure 8.1. For efficiency, we don't want to redraw the whole screen after each operation—only the parts which have changed. Hence

```
(defmacro with-redraw ((var objs) &body body)
  (let ((gob (gensym))
        (x0 (gensym)) (y0 (gensym))
        (x1 (gensym)) (y1 (gensym)))
    `(let ((,gob ,objs))
       (multiple-value-bind (,x0 ,y0 ,x1 ,y1) (bounds ,gob)
         (dolist (,var ,gob) ,@body)
         (multiple-value-bind (xa ya xb yb) (bounds ,gob)
           (redraw (min ,x0 xa) (min ,y0 ya)
                   (max ,x1 xb) (max ,y1 yb)))))))

(defun move-objs (objs dx dy)
  (with-redraw (o objs)
    (incf (obj-x o) dx)
    (incf (obj-y o) dy)))

(defun scale-objs (objs factor)
  (with-redraw (o objs)
    (setf (obj-dx o) (* (obj-dx o) factor)
          (obj-dy o) (* (obj-dy o) factor))))
```

Figure 8.2: Move and scale filleted.

the two calls to the function `bounds`, which returns four coordinates (min x, min y, max x, max y) representing the bounding rectangle of a group of objects. The operative part of `move-objs` is sandwiched between two calls to `bounds` which find the bounding rectangle before and then after the movement, and then redraw the entire affected region.

The function `scale-objs` is for changing the size of a group of objects. Since the bounding region could grow or shrink depending on the scale factor, this function too must do its work between two calls to `bounds`. As we wrote more of the program, we would see more of this pattern: in functions to rotate, flip, transpose, and so on.

With a macro we can abstract out the code that these functions would all have in common. The macro `with-redraw` in Figure 8.2 provides the skeleton that the functions in Figure 8.1 share.[3] As a result, they can now be defined in four lines each, as at the end of Figure 8.2. With these two functions the new macro has already paid for itself in brevity. And how much clearer the two functions ○

[3]The definition of this macro anticipates the next chapter by using gensyms. Their purpose will be explained shortly.

become once the details of screen redrawing are abstracted away.

One way to view `with-redraw` is as a construct in a language for writing interactive drawing programs. As we develop more such macros, they will come to resemble a programming language in fact as well as in name, and our application itself will begin to show the elegance one would expect in a program written in a language defined for its specific needs.

The other major use of macros is to implement embedded languages. Lisp is an exceptionally good language in which to write programming languages, because Lisp programs can be expressed as lists, and Lisp has a built-in parser (`read`) and compiler (`compile`) for programs so expressed. Most of the time you don't even have to call `compile`; you can have your embedded language compiled implicitly, by compiling the code which does the transformations (page 25).

An embedded language is one which is not written on top of Lisp so much as commingled with it, so that the syntax is a mixture of Lisp and constructs specific to the new language. The naive way to implement an embedded language is to write an interpreter for it in Lisp. A better approach, when possible, is to implement the language by transformation: transform each expression into the Lisp code that the interpreter would have run in order to evaluate it. That's where macros come in. The job of macros is precisely to transform one type of expression into another, so they're the natural choice when writing embedded languages.

In general, the more an embedded language can be implemented by transformation, the better. For one, it's less work. If the new language has arithmetic, for example, you needn't face all the complexities of representing and manipulating numeric quantities. If Lisp's arithmetic capabilities are sufficient for your purposes, then you can simply transform your arithmetic expressions into the equivalent Lisp ones, and leave the rest to the Lisp.

Using transformation will ordinarily make your embedded languages faster as well. Interpreters have inherent disadvantages with respect to speed. When code occurs within a loop, for example, an interpreter will often have to do work on each iteration which in compiled code could have been done just once. An embedded language which has its own interpreter will therefore be slow, even if the interpreter itself is compiled. But if the expressions in the new language are transformed into Lisp, the resulting code can then be compiled by the Lisp compiler. A language so implemented need suffer none of the overheads of interpretation at runtime. Short of writing a true compiler for your language, macros will yield the best performance. In fact, the macros which transform the new language can be seen as a compiler for it—just one which relies on the existing Lisp compiler to do most of the work.

We won't consider any examples of embedded languages here, since Chapters 19–25 are all devoted to the topic. Chapter 19 deals specifically with the dif-

ference between interpreting and transforming embedded languages, and shows the same language implemented by each of the two methods.

One book on Common Lisp asserts that the scope for macros is limited, citing as evidence the fact that, of the operators defined in CLTL1, less than 10% were macros. This is like saying that since our house is made of bricks, our furniture will be too. The proportion of macros in a Common Lisp program will depend entirely on what it's supposed to do. Some programs will contain no macros. Some programs could be all macros.