7. Macros: Standard Control Constructs

While many of the ideas that originated in Lisp, from the conditional expression to garbage collection, have been incorporated into other languages, the one language feature that continues to set Common Lisp apart is its macro system. Unfortunately, the word *macro* describes a lot of things in computing to which Common Lisp's macros bear only a vague and metaphorical similarity. This causes no end of misunderstanding when Lispers try to explain to non-Lispers what a great feature macros are.[1] To understand Lisp's macros, you really need to come at them fresh, without preconceptions based on other things that also happen to be called macros. So let's start our discussion of Lisp's macros by taking a step back and looking at various ways languages support extensibility.

All programmers should be used to the idea that the definition of a language can include a standard library of functionality that's implemented in terms of the "core" language--functionality that could have been implemented by any programmer on top of the language if it hadn't been defined as part of the standard library. C's standard library, for instance, can be implemented almost entirely in portable C. Similarly, most of the ever-growing set of classes and interfaces that ship with Java's standard Java Development Kit (JDK) are written in "pure" Java.

One advantage of defining languages in terms of a core plus a standard library is it makes them easier to understand and implement. But the real benefit is in terms of expressiveness--since much of what you think of as "the language" is really just a library--the language is easy to extend. If C doesn't have a function to do some thing or another that you need, you can write that function, and now you have a slightly richer version of C. Similarly, in a language such as Java or Smalltalk where almost all the interesting parts of the "language" are defined in terms of classes, by defining new classes you extend the language, making it more suited for writing programs to do whatever it is you're trying to do.

While Common Lisp supports both these methods of extending the language, macros give Common Lisp yet another way. As I discussed briefly in Chapter 4, each macro defines its own syntax, determining how the s-expressions it's passed are turned into Lisp forms. With macros as part of the core language it's possible to build new syntax--control constructs such as **`WHEN`**, **`DOLIST`**, and **`LOOP`** as well as definitional forms such as **`DEFUN`** and **`DEFPARAMETER`**--as part of the "standard library" rather than having to hardwire them into the core. This has implications for how the language itself is implemented, but as a Lisp programmer you'll care more that it gives you another way to extend the language, making it a better language for expressing solutions to your particular programming problems.

Now, it may seem that the benefits of having another way to extend the language would be easy to recognize. But for some reason a lot of folks who haven't actually used Lisp macros--folks who think nothing of spending their days creating new functional abstractions or defining hierarchies of classes to solve their programming problems--get spooked by the idea of being able to define new syntactic abstractions. The most common cause of macrophobia seems to be bad experiences with other "macro" systems. Simple fear of the unknown no doubt plays a role, too. To avoid triggering any macrophobic reactions, I'll ease into the subject by discussing several of the standard control-construct macros defined by Common Lisp. These are some of the things that, if Lisp didn't have macros, would have to be built into the language core. When you use them, you don't have to care that they're implemented as macros, but they provide a good example of some of the things you can do with macros.[2] In the next chapter, I'll show you how you can define your own macros.

WHEN and UNLESS

As you've already seen, the most basic form of conditional execution--if *x*, do *y*; otherwise do *z*--is provided by the **`IF`** special operator, which has this basic form:

```
(if condition then-form [else-form])
```

The *condition* is evaluated and, if its value is non-**`NIL`**, the *then-form* is evaluated and the resulting value returned. Otherwise, the *else-form*, if any, is evaluated and its value returned. If *condition* is **`NIL`** and there's no *else-form*, then the **`IF`** returns **`NIL`**.

```
(if (> 2 3) "Yup" "Nope") ==> "Nope"
(if (> 2 3) "Yup")        ==> NIL
(if (> 3 2) "Yup" "Nope") ==> "Yup"
```

However, **`IF`** isn't actually such a great syntactic construct because the *then-form* and *else-form* are each restricted to being a single Lisp form. This means if you want to perform a sequence of actions in either clause, you need to wrap them in some other syntax. For instance, suppose in the middle of a spam-filtering program you wanted to both file a message as spam and update the spam database when a message is spam. You can't write this:

```
(if (spam-p current-message)
    (file-in-spam-folder current-message)
    (update-spam-database current-message))
```

because the call to `update-spam-database` will be treated as the else clause, not as part of the then clause. Another special operator, **`PROGN`**, executes any number of forms in order and returns the value of the last form. So you could get the desired behavior by writing the following:

```
(if (spam-p current-message)
    (progn
      (file-in-spam-folder current-message)
      (update-spam-database current-message)))
```

That's not too horrible. But given the number of times you'll likely have to use this idiom, it's not hard to imagine that you'd get tired of it after a while. "Why," you might ask yourself, "doesn't Lisp provide a way to say what I really want, namely, 'When *x* is true, do this, that, and the other thing'?" In other words, after a while you'd notice the pattern of an **IF** plus a **PROGN** and wish for a way to abstract away the details rather than writing them out every time.

This is exactly what macros provide. In this case, Common Lisp comes with a standard macro, **WHEN**, which lets you write this:

```
(when (spam-p current-message)
  (file-in-spam-folder current-message)
  (update-spam-database current-message))
```

But if it wasn't built into the standard library, you could define **WHEN** yourself with a macro such as this, using the backquote notation I discussed in Chapter 3:[3]

```
(defmacro when (condition &rest body)
  `(if ,condition (progn ,@body)))
```

A counterpart to the **WHEN** macro is **UNLESS**, which reverses the condition, evaluating its body forms only if the condition is false. In other words:

```
(defmacro unless (condition &rest body)
  `(if (not ,condition) (progn ,@body)))
```

Admittedly, these are pretty trivial macros. There's no deep black magic here; they just abstract away a few language-level bookkeeping details, allowing you to express your true intent a bit more clearly. But their very triviality makes an important point: because the macro system is built right into the language, you can write trivial macros like **WHEN** and **UNLESS** that give you small but real gains in clarity that are then multiplied by the thousands of times you use them. In Chapters 24, 26, and 31 you'll see how macros can also be used on a larger scale, creating whole domain-specific embedded languages. But first let's finish our discussion of the standard control-construct macros.

COND

Another time raw **IF** expressions can get ugly is when you have a multibranch conditional: if *a* do *x*, else if *b* do *y*; else do *z*. There's no logical problem writing such a chain of conditional expressions with just **IF**, but it's not pretty.

```
(if a
    (do-x)
    (if b
       (do-y)
       (do-z)))
```

And it would be even worse if you needed to include multiple forms in the then clauses, requiring **PROGN**s. So, not surprisingly, Common Lisp provides a macro for expressing multibranch conditionals: **COND**. This is the basic skeleton:

```
(cond
  (test-1 form*)
      .
      .
      .
  (test-N form*))
```

Each element of the body represents one branch of the conditional and consists of a list containing a condition form and zero or more forms to be evaluated if that branch is chosen. The conditions are evaluated in the order the branches appear in the body until one of them evaluates to true. At that point, the remaining forms in that branch are evaluated, and the value of the last form in the branch is returned as the value of the **COND** as a whole. If the branch contains no forms after the condition, the value of the condition is returned instead. By convention, the branch representing the final else clause in an if/else-if chain is written with a condition of **T**. Any non-**NIL** value will work, but a **T** serves as a useful landmark when reading the code. Thus, you can write the previous nested **IF** expression using **COND** like this:

```
(cond (a (do-x))
      (b (do-y))
      (t (do-z)))
```

AND, OR, and NOT

When writing the conditions in **IF**, **WHEN**, **UNLESS**, and **COND** forms, three operators that will come in handy are the boolean logic operators, **AND**, **OR**, and **NOT**.

NOT is a function so strictly speaking doesn't belong in this chapter, but it's closely tied to **AND** and **OR**. It takes a single argument and inverts its truth value, returning **T** if the argument is **NIL** and **NIL** otherwise.

AND and **OR**, however, are macros. They implement logical conjunction and disjunction of any number of subforms and are defined as macros so they can *short-circuit*. That is, they evaluate only as many of their subforms--in left-to-right order--as necessary to determine the overall truth value. Thus, **AND** stops and returns **NIL** as soon as one of its subforms evaluates to **NIL**. If all the subforms evaluate to non-**NIL**, it returns the value of the last subform. **OR**, on the other hand, stops as soon as one of its subforms evaluates to non-**NIL** and returns the resulting value. If none of the subforms evaluate to true, **OR** returns **NIL**. Here are some examples:

```
(not nil)             ==> T
(not (= 1 1))         ==> NIL
(and (= 1 2) (= 3 3)) ==> NIL
(or (= 1 2) (= 3 3))  ==> T
```

Looping

Control constructs are the other main kind of looping constructs. Common Lisp's looping facilities are--in addition to being quite powerful and flexible--an interesting lesson in the have-your-cake-and-eat-it-too style of programming that macros provide.

As it turns out, none of Lisp's 25 special operators directly support structured looping. All of Lisp's looping control constructs are macros built on top of a pair of special operators that provide a primitive goto facility.[4] Like many good abstractions, syntactic or otherwise, Lisp's looping macros are built as a set of layered abstractions starting from the base provided by those two special operators.

At the bottom (leaving aside the special operators) is a very general looping construct, **DO**. While very powerful, **DO** suffers, as do many general-purpose abstractions, from being overkill for simple situations. So Lisp also provides two other macros, **DOLIST** and **DOTIMES**, that are less flexible than **DO** but provide convenient support for the common cases of looping over the elements of a list and counting loops. While an implementation can implement these macros however it wants, they're typically implemented as macros that expand into an equivalent **DO** loop. Thus, **DO** provides a basic structured looping construct on top of the underlying primitives provided by Common Lisp's special operators, and **DOLIST** and **DOTIMES** provide two easier-to-use, if less general, constructs. And, as you'll see in the next chapter, you can build your own looping constructs on top of **DO** for situations where **DOLIST** and **DOTIMES** don't meet your needs.

Finally, the **LOOP** macro provides a full-blown mini-language for expressing looping constructs in a non-Lispy, English-like (or at least Algol-like) language. Some Lisp hackers love **LOOP**; others hate it. **LOOP**'s fans like it because it provides a concise way to express certain commonly needed looping constructs. Its detractors dislike it because it's not Lispy enough. But whichever side one comes down on, it's a remarkable example of the power of macros to add new constructs to the language.

DOLIST and DOTIMES

I'll start with the easy-to-use **DOLIST** and **DOTIMES** macros.

DOLIST loops across the items of a list, executing the loop body with a variable holding the successive items of the list.[5] This is the basic skeleton (leaving out some of the more esoteric options):

```
(dolist (var list-form)
  body-form*)
```

When the loop starts, the *list-form* is evaluated once to produce a list. Then the body of the loop is evaluated once for each item in the list with the variable *var* holding the value of the item. For instance:

```
CL-USER> (dolist (x '(1 2 3)) (print x))
1
2
3
NIL
```

Used this way, the **DOLIST** form as a whole evaluates to **NIL**.

If you want to break out of a **DOLIST** loop before the end of the list, you can use **RETURN**.

```
CL-USER> (dolist (x '(1 2 3)) (print x) (if (evenp x) (return)))
1
2
NIL
```

DOTIMES is the high-level looping construct for counting loops. The basic template is much the same as **DOLIST**'s.

```
(dotimes (var count-form)
  body-form*)
```

The *count-form* must evaluate to an integer. Each time through the loop *var* holds successive integers from 0 to one less than that number. For instance:

```
CL-USER> (dotimes (i 4) (print i))
0
1
2
3
NIL
```

As with **DOLIST**, you can use **RETURN** to break out of the loop early.

Because the body of both **DOLIST** and **DOTIMES** loops can contain any kind of expressions, you can also nest loops. For example, to print out the times tables from `1 × 1 = 1` to `20 × 20 = 400`, you can write this pair of nested **DOTIMES** loops:

```
(dotimes (x 20)
  (dotimes (y 20)
    (format t "~3d " (* (1+ x) (1+ y))))
  (format t "~%"))
```

DO

While **DOLIST** and **DOTIMES** are convenient and easy to use, they aren't flexible enough to use for all loops. For instance, what if you want to step multiple variables in parallel? Or use an arbitrary expression to test for the end of the loop? If neither **DOLIST** nor **DOTIMES** meet your needs, you still have access to the more general **DO** loop.

Where **DOLIST** and **DOTIMES** provide only one loop variable, **DO** lets you bind any number of variables and gives you complete control over how they change on each step through the loop. You also get to define the test that determines when to end the loop and can provide a form to evaluate at the end of the loop to generate a return value for the **DO** expression as a whole. The basic template looks like this:

```
(do (variable-definition*)
    (end-test-form result-form*)
  statement*)
```

Each *variable-definition* introduces a variable that will be in scope in the body of the loop. The full form of a single variable definition is a list containing three elements.

```
(var init-form step-form)
```

The *init-form* will be evaluated at the beginning of the loop and the resulting values bound to the variable *var*. Before each subsequent iteration of the loop, the *step-form* will be evaluated and the new value assigned to *var*. The *step-form* is optional; if it's left out, the variable will keep its value from iteration to iteration unless you explicitly assign it a new value in the loop body. As with the variable definitions in a **LET**, if the *init-form* is left out, the variable is bound to **NIL**. Also as with **LET**, you can use a plain variable name as shorthand for a list containing just the name.

At the beginning of each iteration, after all the loop variables have been given their new values, the *end-test-form* is evaluated. As long as it evaluates to **NIL**, the iteration proceeds, evaluating the *statements* in order.

When the *end-test-form* evaluates to true, the *result-forms* are evaluated, and the value of the last result form is returned as the value of the **DO** expression.

At each step of the iteration the step forms for all the variables are evaluated before assigning any of the values to the variables. This means you can refer to any of the other loop variables in the step forms.[6] That is, in a loop like this:

```
(do ((n 0 (1+ n))
     (cur 0 next)
     (next 1 (+ cur next)))
    ((= 10 n) cur))
```

the step forms `(1+ n)`, `next`, and `(+ cur next)` are all evaluated using the old values of `n`, `cur`, and `next`. Only after all the step forms have been evaluated are the variables given their new values. (Mathematically inclined readers may notice that this is a particularly efficient way of computing the eleventh Fibonacci number.)

This example also illustrates another characteristic of **DO**--because you can step multiple variables, you often don't need a body at all. Other times, you may leave out the result form, particularly if you're just using the loop as a control construct. This flexibility, however, is the reason that **DO** expressions can be a bit cryptic. Where exactly do all the parentheses go? The best way to understand a **DO** expression is to keep in mind the basic template.

```
(do (variable-definition*)
    (end-test-form result-form*)
  statement*)
```

The six parentheses in that template are the only ones required by the **DO** itself. You need one pair to enclose the variable declarations, one pair to enclose the end test and result forms, and one pair to enclose the whole expression. Other forms within the **DO** may require their own

parentheses--variable definitions are usually lists, for instance. And the test form is often a function call. But the skeleton of a **DO** loop will always be the same. Here are some example **DO** loops with the skeleton in bold:

```
(do ((i 0 (1+ i)))
    ((>= i 4))
  (print i))
```

Notice that the result form has been omitted. This is, however, not a particularly idiomatic use of **DO**, as this loop is much more simply written using **DOTIMES**.[7]

```
(dotimes (i 4) (print i))
```

As another example, here's the bodiless Fibonacci-computing loop:

```
(do ((n 0 (1+ n))
     (cur 0 next)
     (next 1 (+ cur next)))
    ((= 10 n) cur))
```

Finally, the next loop demonstrates a **DO** loop that binds no variables. It loops while the current time is less than the value of a global variable, printing "Waiting" once a minute. Note that even with no loop variables, you still need the empty variables list.

```
(do ()
    ((> (get-universal-time) *some-future-date*))
  (format t "Waiting~%")
  (sleep 60))
```

The Mighty LOOP

For the simple cases you have **DOLIST** and **DOTIMES**. And if they don't suit your needs, you can fall back on the completely general **DO**. What more could you want?

Well, it turns out a handful of looping idioms come up over and over again, such as looping over various data structures: lists, vectors, hash tables, and packages. Or accumulating values in various ways while looping: collecting, counting, summing, minimizing, or maximizing. If you need a loop to do one of these things (or several at the same time), the **LOOP** macro may give you an easier way to express it.

The **LOOP** macro actually comes in two flavors--*simple* and *extended*. The simple version is as simple as can be--an infinite loop that doesn't bind any variables. The skeleton looks like this:

```
(loop
  body-form*)
```

The forms in body are evaluated each time through the loop, which will iterate forever unless you use **RETURN** to break out. For example, you could write the previous **DO** loop with a simple **LOOP**.

```
(loop
  (when (> (get-universal-time) *some-future-date*)
    (return))
  (format t "Waiting~%")
  (sleep 60))
```

The extended **LOOP** is quite a different beast. It's distinguished by the use of certain *loop keywords* that implement a special-purpose language for expressing looping idioms. It's worth noting that not all Lispers love the extended **LOOP** language. At least one of Common Lisp's original designers hated it. **LOOP**'s detractors complain that its syntax is totally un-Lispy (in other words, not enough parentheses). **LOOP**'s fans counter that that's the point: complicated looping constructs are hard enough to understand without wrapping them up in **DO**'s cryptic syntax. It's better, they say, to have a slightly more verbose syntax that gives you some clues what the heck is going on.

For instance, here's an idiomatic **DO** loop that collects the numbers from 1 to 10 into a list:

```
(do ((nums nil) (i 1 (1+ i)))
    ((> i 10) (nreverse nums))
  (push i nums)) ==> (1 2 3 4 5 6 7 8 9 10)
```

A seasoned Lisper won't have any trouble understanding that code--it's just a matter of understanding the basic form of a **DO** loop and recognizing the **PUSH**/**NREVERSE** idiom for building up a list. But it's not exactly transparent. The **LOOP** version, on the other hand, is almost understandable as an English sentence.

```
(loop for i from 1 to 10 collecting i) ==> (1 2 3 4 5 6 7 8 9 10)
```

The following are some more examples of simple uses of **LOOP**. This sums the first ten squares:

```
(loop for x from 1 to 10 summing (expt x 2)) ==> 385
```

This counts the number of vowels in a string:

```
(loop for x across "the quick brown fox jumps over the lazy dog"
      counting (find x "aeiou")) ==> 11
```

This computes the eleventh Fibonacci number, similar to the **DO** loop used earlier:

```
(loop for i below 10
      and a = 0 then b
      and b = 1 then (+ b a)
      finally (return  a))
```

The symbols `across`, `and`, `below`, `collecting`, `counting`, `finally`, `for`, `from`, `summing`, `then`, and `to` are some of the loop keywords whose presence identifies these as instances of the extended **LOOP**. [8]

I'll save the details of **LOOP** for Chapter 22, but it's worth noting here as another example of the way macros can be used to extend the base language. While **LOOP** provides its own language for

expressing looping constructs, it doesn't cut you off from the rest of Lisp. The loop keywords are parsed according to loop's grammar, but the rest of the code in a **LOOP** is regular Lisp code.

And it's worth pointing out one more time that while the **LOOP** macro is quite a bit more complicated than macros such as **WHEN** or **UNLESS**, it *is* just another macro. If it hadn't been included in the standard library, you could implement it yourself or get a third-party library that does.

With that I'll conclude our tour of the basic control-construct macros. Now you're ready to take a closer look at how to define your own macros.

---

[1]To see what this misunderstanding looks like, find any longish Usenet thread cross-posted between comp.lang.lisp and any other comp.lang.* group with *macro* in the subject. A rough paraphrase goes like this:

Lispnik: "Lisp is the best because of its macros!";

Othernik: "You think Lisp is good *because of* macros?! But macros are horrible and evil; Lisp must be horrible and evil."

[2]Another important class of language constructs that are defined using macros are all the definitional constructs such as **DEFUN**, **DEFPARAMETER**, **DEFVAR**, and others. In Chapter 24 you'll define your own definitional macros that will allow you to concisely write code for reading and writing binary data.

[3]You can't actually feed this definition to Lisp because it's illegal to redefine names in the `COMMON-LISP` package where **WHEN** comes from. If you really want to try writing such a macro, you'd need to change the name to something else, such as `my-when`.

[4]The special operators, if you must know, are **TAGBODY** and **GO**. There's no need to discuss them now, but I'll cover them in Chapter 20.

[5]**DOLIST** is similar to Perl's `foreach` or Python's `for`. Java added a similar kind of loop construct with the "enhanced" `for` loop in Java 1.5, as part of JSR-201. Notice what a difference macros make. A Lisp programmer who notices a common pattern in their code can write a macro to give themselves a source-level abstraction of that pattern. A Java programmer who notices the same pattern has to convince Sun that this particular abstraction is worth adding to the language. Then Sun has to publish a JSR and convene an industry-wide "expert group" to hash everything out. That process--according to Sun--takes an average of 18 months. After that, the compiler writers all have to go upgrade their compilers to support the new feature. And even once the Java programmer's favorite compiler supports the new version of Java, they probably *still* can't use the new feature until they're allowed to break source compatibility with older versions of Java. So an annoyance that Common Lisp programmers can resolve for themselves within five minutes plagues Java programmers for years.

[6]A variant of **DO**, **DO***, assigns each variable its value before evaluating the step form for subsequent variables. For more details, consult your favorite Common Lisp reference.

[7]The **DOTIMES** is also preferred because the macro expansion will likely include declarations that allow the compiler to generate more efficient code.

[8]*Loop keywords* is a bit of a misnomer since they aren't keyword symbols. In fact, **LOOP** doesn't care what package the symbols are from. When the **LOOP** macro parses its body, it considers any appropriately named symbols equivalent. You could even use true keywords if you wanted--`:for`, `:across`, and so on--because they also have the correct name. But most folks just use plain symbols. Because the loop keywords are used only as syntactic markers, it doesn't matter if they're used for other purposes--as function or variable names.