8. Macros: Defining Your Own

Now it's time to start writing your own macros. The standard macros I covered in the previous chapter hint at some of the things you can do with macros, but that's just the beginning. Common Lisp doesn't support macros so every Lisp programmer can create their own variants of standard control constructs any more than C supports functions so every C programmer can write trivial variants of the functions in the C standard library. Macros are part of the language to allow you to create abstractions on top of the core language and standard library that move you closer toward being able to directly express the things you want to express.

Perhaps the biggest barrier to a proper understanding of macros is, ironically, that they're so well integrated into the language. In many ways they seem like just a funny kind of function--they're written in Lisp, they take arguments and return results, and they allow you to abstract away distracting details. Yet despite these many similarities, macros operate at a different level than functions and create a totally different kind of abstraction.

Once you understand the difference between macros and functions, the tight integration of macros in the language will be a huge benefit. But in the meantime, it's a frequent source of confusion for new Lispers. The following story, while not true in a historical or technical sense, tries to alleviate the confusion by giving you a way to think about how macros work.

The Story of Mac: A Just-So Story

Once upon a time, long ago, there was a company of Lisp programmers. It was so long ago, in fact, that Lisp had no macros. Anything that couldn't be defined with a function or done with a special operator had to be written in full every time, which was rather a drag. Unfortunately, the programmers in this company--though brilliant--were also quite lazy. Often in the middle of their programs--when the tedium of writing a bunch of code got to be too much--they would instead write a note describing the code they needed to write at that place in the program. Even more unfortunately, because they were lazy, the programmers also hated to go back and actually write the code described by the notes. Soon the company had a big stack of programs that nobody could run because they were full of notes about code that still needed to be written.

In desperation, the big bosses hired a junior programmer, Mac, whose job was to find the notes, write the required code, and insert it into the program in place of the notes. Mac never ran the programs--they weren't done yet, of course, so he couldn't. But even if they had been completed, Mac wouldn't have known what inputs to feed them. So he just wrote his code based on the contents of the notes and sent it back to the original programmer.

With Mac's help, all the programs were soon completed, and the company made a ton of money selling them--so much money that the company could double the size of its programming staff. But for some reason no one thought to hire anyone to help Mac; soon he was single- handedly assisting several dozen programmers. To avoid spending all his time searching for notes in source code, Mac made a small modification to the compiler the programmers used. Thereafter, whenever the compiler hit a note, it would e-mail him the note and wait for him to e-mail back the replacement code. Unfortunately, even with this change, Mac had a hard time keeping up with the programmers. He worked as carefully as he could, but sometimes-- especially when the notes weren't clear--he would make mistakes.

The programmers noticed, however, that the more precisely they wrote their notes, the more likely it was that Mac would send back correct code. One day, one of the programmers, having a hard time describing in words the code he wanted, included in one of his notes a Lisp program that would generate the code he wanted. That was fine by Mac; he just ran the program and sent the result to the compiler.

The next innovation came when a programmer put a note at the top of one of his programs containing a function definition and a comment that said, "Mac, don't write any code here, but keep this function for later; I'm going to use it in some of my other notes." Other notes in the same program said things such as, "Mac, replace this note with the result of running that other function with the symbols `x` and `y` as arguments."

This technique caught on so quickly that within a few days, most programs contained dozens of notes defining functions that were only used by code in other notes. To make it easy for Mac to pick out the notes containing only definitions that didn't require any immediate response, the programmers tagged them with the standard preface: "Definition for Mac, Read Only." This--as the programmers were still quite lazy--was quickly shortened to "DEF. MAC. R/O" and then "DEFMACRO."

Pretty soon, there was no actual English left in the notes for Mac. All he did all day was read and respond to e-mails from the compiler containing DEFMACRO notes and calls to the functions defined in the DEFMACROs. Since the Lisp programs in the notes did all the real work, keeping up with the e-mails was no problem. Mac suddenly had a lot of time on his hands and would sit in his office daydreaming about white-sand beaches, clear blue ocean water, and drinks with little paper umbrellas in them.

Several months later the programmers realized nobody had seen Mac for quite some time. When they went to his office, they found a thin layer of dust over everything, a desk littered with travel brochures for various tropical locations, and the computer off. But the compiler still worked-- how could it be? It turned out Mac had made one last change to the compiler: instead of emailing notes to Mac, the compiler now saved the functions defined by DEFMACRO notes and ran them when called for by the other notes. The programmers decided there was no reason to tell the big bosses Mac wasn't coming to the office anymore. So to this day, Mac draws a salary and from time to time sends the programmers a postcard from one tropical locale or another.

Macro Expansion Time vs. Runtime

The key to understanding macros is to be quite clear about the distinction between the code that generates code (macros) and the code that eventually makes up the program (everything else). When you write macros, you're writing programs that will be used by the compiler to generate the code that will then be compiled. Only after all the macros have been fully expanded and the resulting code compiled can the program actually be run. The time when macros run is called *macro expansion time*; this is distinct from *runtime*, when regular code, including the code generated by macros, runs.

It's important to keep this distinction firmly in mind because code running at macro expansion time runs in a very different environment than code running at runtime. Namely, at macro expansion time, there's no way to access the data that will exist at runtime. Like Mac, who couldn't run the programs he was working on because he didn't know what the correct inputs were, code running at macro expansion time can deal only with the data that's inherent in the source code. For instance, suppose the following source code appears somewhere in a program:

```
(defun foo (x)
  (when (> x 10) (print 'big)))
```

Normally you'd think of `x` as a variable that will hold the argument passed in a call to `foo`. But at macro expansion time, such as when the compiler is running the **`WHEN`** macro, the only data available is the source code. Since the program isn't running yet, there's no call to `foo` and thus no value associated with `x`. Instead, the values the compiler passes to **`WHEN`** are the Lisp lists representing the source code, namely, `(> x 10)` and `(print 'big)`. Suppose that **`WHEN`** is defined, as you saw in the previous chapter, with something like the following macro:

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

When the code in `foo` is compiled, the **`WHEN`** macro will be run with those two forms as arguments. The parameter `condition` will be bound to the form `(> x 10)`, and the form `(print 'big)` will be collected into a list that will become the value of the **`&rest`** `body` parameter. The backquote expression will then generate this code:

```
(if (> x 10) (progn (print 'big)))
```

by interpolating in the value of `condition` and splicing the value of `body` into the **`PROGN`**.

When Lisp is interpreted, rather than compiled, the distinction between macro expansion time and runtime is less clear because they're temporally intertwined. Also, the language standard doesn't specify exactly how an interpreter must handle macros--it could expand all the macros in the form being interpreted and then interpret the resulting code, or it could start right in on interpreting the form and expand macros when it hits them. In either case, macros are always passed the unevaluated Lisp objects representing the subforms of the macro form, and the job of the macro is still to produce code that will do something rather than to do anything directly.

DEFMACRO

As you saw in Chapter 3, macros really are defined with **`DEFMACRO`** forms, though it stands--of course--for DEFine MACRO, not Definition for Mac. The basic skeleton of a **`DEFMACRO`** is quite similar to the skeleton of a **`DEFUN`**.

```
(defmacro name (parameter*)
  "Optional documentation string."
  body-form*)
```

Like a function, a macro consists of a name, a parameter list, an optional documentation string, and a body of Lisp expressions.[1] However, as I just discussed, the job of a macro isn't to do anything directly--its job is to generate code that will later do what you want.

Macros can use the full power of Lisp to generate their expansion, which means in this chapter I can only scratch the surface of what you can do with macros. I can, however, describe a general process for writing macros that works for all macros from the simplest to the most complex.

The job of a macro is to translate a macro form--in other words, a Lisp form whose first element is the name of the macro--into code that does a particular thing. Sometimes you write a macro starting with the code you'd like to be able to write, that is, with an example macro form. Other times you decide to write a macro after you've written the same pattern of code several times and realize you can make your code clearer by abstracting the pattern.

Regardless of which end you start from, you need to figure out the other end before you can start writing a macro: you need to know both where you're coming from and where you're going before you can hope to write code to do it automatically. Thus, the first step of writing a macro is to write at least one example of a call to the macro and the code into which that call should expand.

Once you have an example call and the desired expansion, you're ready for the second step: writing the actual macro code. For simple macros this will be a trivial matter of writing a backquoted template with the macro parameters plugged into the right places. Complex macros will be significant programs in their own right, complete with helper functions and data structures.

After you've written code to translate the example call to the appropriate expansion, you need to make sure the abstraction the macro provides doesn't "leak" details of its implementation. Leaky macro abstractions will work fine for certain arguments but not others or will interact with code in the calling environment in undesirable ways. As it turns out, macros can leak in a small handful of ways, all of which are easily avoided as long as you know to check for them. I'll discuss how in the section "Plugging the Leaks."

To sum up, the steps to writing a macro are as follows:

1. Write a sample call to the macro and the code it should expand into, or vice versa.
2. Write code that generates the handwritten expansion from the arguments in the sample call.
3. Make sure the macro abstraction doesn't "leak."

A Sample Macro: do-primes

To see how this three-step process works, you'll write a macro `do-primes` that provides a looping construct similar to **`DOTIMES`** and **`DOLIST`** except that instead of iterating over integers or elements of a list, it iterates over successive prime numbers. This isn't meant to be an example of a particularly useful macro--it's just a vehicle for demonstrating the process.

First, you'll need two utility functions, one to test whether a given number is prime and another that returns the next prime number greater or equal to its argument. In both cases you can use a simple, but inefficient, brute-force approach.

```
(defun primep (number)
  (when (> number 1)
    (loop for fac from 2 to (isqrt number) never (zerop (mod number fac)))))

(defun next-prime (number)
  (loop for n from number when (primep n) return n))
```

Now you can write the macro. Following the procedure outlined previously, you need at least one example of a call to the macro and the code into which it should expand. Suppose you start with the idea that you want to be able to write this:

```
(do-primes (p 0 19)
  (format t "~d " p))
```

to express a loop that executes the body once each for each prime number greater or equal to 0 and less than or equal to 19, with the variable `p` holding the prime number. It makes sense to model this macro on the form of the standard **`DOLIST`** and **`DOTIMES`** macros; macros that follow the pattern of existing macros are easier to understand and use than macros that introduce gratuitously novel syntax.

Without the `do-primes` macro, you could write such a loop with **DO** (and the two utility functions defined previously) like this:

```
(do ((p (next-prime 0) (next-prime (1+ p))))
    ((> p 19))
  (format t "~d " p))
```

Now you're ready to start writing the macro code that will translate from the former to the latter.

Macro Parameters

Since the arguments passed to a macro are Lisp objects representing the source code of the macro call, the first step in any macro is to extract whatever parts of those objects are needed to compute the expansion. For macros that simply interpolate their arguments directly into a template, this step is trivial: simply defining the right parameters to hold the different arguments is sufficient.

But this approach, it seems, will not suffice for `do-primes`. The first argument to the `do-primes` call is a list containing the name of the loop variable, `p`; the lower bound, `0`; and the upper bound, `19`. But if you look at the expansion, the list as a whole doesn't appear in the expansion; the three element are split up and put in different places.

You could define `do-primes` with two parameters, one to hold the list and a **`&rest`** parameter to hold the body forms, and then take apart the list by hand, something like this:

```
(defmacro do-primes (var-and-range &rest body)
  (let ((var (first var-and-range))
        (start (second var-and-range))
        (end (third var-and-range)))
    `(do ((,var (next-prime ,start) (next-prime (1+ ,var))))
         ((> ,var ,end))
       ,@body)))
```

In a moment I'll explain how the body generates the correct expansion; for now you can just note that the variables `var`, `start`, and `end` each hold a value, extracted from `var-and-range`, that's then interpolated into the backquote expression that generates `do-primes`'s expansion.

However, you don't need to take apart `var-and-range` "by hand" because macro parameter lists are what are called *destructuring* parameter lists. Destructuring, as the name suggests, involves taking apart a structure--in this case the list structure of the forms passed to a macro.

Within a destructuring parameter list, a simple parameter name can be replaced with a nested parameter list. The parameters in the nested parameter list will take their values from the elements of the expression that would have been bound to the parameter the list replaced. For instance, you can replace `var-and-range` with a list `(var start end)`, and the three elements of the list will automatically be destructured into those three parameters.

Another special feature of macro parameter lists is that you can use **`&body`** as a synonym for **`&rest`**. Semantically **`&body`** and **`&rest`** are equivalent, but many development environments will use the presence of a **`&body`** parameter to modify how they indent uses of the macro--typically **`&body`** parameters are used to hold a list of forms that make up the body of the macro.

So you can streamline the definition of `do-primes` and give a hint to both human readers and your development tools about its intended use by defining it like this:

```
(defmacro do-primes ((var start end) &body body)
  `(do ((,var (next-prime ,start) (next-prime (1+ ,var))))
       ((> ,var ,end))
     ,@body))
```

In addition to being more concise, destructuring parameter lists also give you automatic error checking--with `do-primes` defined this way, Lisp will be able to detect a call whose first argument isn't a three-element list and will give you a meaningful error message just as if you had called a function with too few or too many arguments. Also, in development environments such as SLIME that indicate what arguments are expected as soon as you type the name of a function or macro, if you use a destructuring parameter list, the environment will be able to tell you more specifically the syntax of the macro call. With the original definition, SLIME would tell you `do-primes` is called like this:

```
(do-primes var-and-range &rest body)
```

But with the new definition, it can tell you that a call should look like this:

```
(do-primes (var start end) &body body)
```

Destructuring parameter lists can contain **`&optional`**, **`&key`**, and **`&rest`** parameters and can contain nested destructuring lists. However, you don't need any of those options to write `do-primes`.

Generating the Expansion

Because `do-primes` is a fairly simple macro, after you've destructured the arguments, all that's left is to interpolate them into a template to get the expansion.

For simple macros like `do-primes`, the special backquote syntax is perfect. To review, a backquoted expression is similar to a quoted expression except you can "unquote" particular subexpressions by preceding them with a comma, possibly followed by an at (@) sign. Without an at sign, the comma causes the value of the subexpression to be included as is. With an at sign, the value--which must be a list--is "spliced" into the enclosing list.

Another useful way to think about the backquote syntax is as a particularly concise way of writing code that generates lists. This way of thinking about it has the benefit of being pretty much exactly what's happening under the covers--when the reader reads a backquoted expression, it translates it into code that generates the appropriate list structure. For instance, `` `(,a b) `` might be read as `(list a 'b)`. The language standard doesn't specify exactly what code the reader must produce as long as it generates the right list structure.

Table 8-1 shows some examples of backquoted expressions along with equivalent list-building code and the result you'd get if you evaluated either the backquoted expression or the equivalent code.[2]

Table 8-1. Backquote Examples

Backquote Syntax	Equivalent List-Building Code	Result
`` `(a (+ 1 2) c) ``	`(list 'a '(+ 1 2) 'c)`	`(a (+ 1 2) c)`
`` `(a ,(+ 1 2) c) ``	`(list 'a (+ 1 2) 'c)`	`(a 3 c)`
`` `(a (list 1 2) c) ``	`(list 'a '(list 1 2) 'c)`	`(a (list 1 2) c)`
`` `(a ,(list 1 2) c) ``	`(list 'a (list 1 2) 'c)`	`(a (1 2) c)`
`` `(a ,@(list 1 2) c) ``	`(append (list 'a) (list 1 2) (list 'c))`	`(a 1 2 c)`

It's important to note that backquote is just a convenience. But it's a big convenience. To appreciate how big, compare the backquoted version of `do-primes` to the following version, which uses explicit list-building code:

```
(defmacro do-primes-a ((var start end) &body body)
  (append '(do)
          (list  (list (list var
                             (list 'next-prime start)
                             (list 'next-prime (list '1+ var)))))
          (list (list (list '> var end)))
          body))
```

As you'll see in a moment, the current implementation of `do-primes` doesn't handle certain edge cases correctly. But first you should verify that it at least works for the original example. You can test it in two ways. You can test it indirectly by simply using it--presumably, if the resulting behavior is correct, the expansion is correct. For instance, you can type the original example's use of `do-primes` to the REPL and see that it indeed prints the right series of prime numbers.

```
CL-USER> (do-primes (p 0 19) (format t "~d " p))
2 3 5 7 11 13 17 19
NIL
```

Or you can check the macro directly by looking at the expansion of a particular call. The function **`MACROEXPAND-1`** takes any Lisp expression as an argument and returns the result of doing one level of macro expansion.[3] Because **`MACROEXPAND-1`** is a function, to pass it a literal macro form you must quote it. You can use it to see the expansion of the previous call.[4]

```
CL-USER> (macroexpand-1 '(do-primes (p 0 19) (format t "~d " p)))
(DO ((P (NEXT-PRIME 0) (NEXT-PRIME (1+ P))))
    ((> P 19))
  (FORMAT T "~d " P))
T
```

Or, more conveniently, in SLIME you can check a macro's expansion by placing the cursor on the opening parenthesis of a macro form in your source code and typing `C-c RET` to invoke the Emacs function `slime-macroexpand-1`, which will pass the macro form to **`MACROEXPAND-1`** and "pretty print" the result in a temporary buffer.

However you get to it, you can see that the result of macro expansion is the same as the original handwritten expansion, so it seems that `do-primes` works.

Plugging the Leaks

In his essay "The Law of Leaky Abstractions," Joel Spolsky coined the term *leaky abstraction* to describe an abstraction that "leaks" details it's supposed to be abstracting away. Since writing a macro is a way of creating an abstraction, you need to make sure your macros don't leak needlessly.[5]

As it turns out, a macro can leak details of its inner workings in three ways. Luckily, it's pretty easy to tell whether a given macro suffers from any of those leaks and to fix them.

The current definition suffers from one of the three possible macro leaks: namely, it evaluates the `end` subform too many times. Suppose you were to call `do-primes` with, instead of a literal number such as `19`, an expression such as `(random 100)` in the `end` position.

```
(do-primes (p 0 (random 100))
  (format t "~d " p))
```

Presumably the intent here is to loop over the primes from zero to whatever random number is returned by `(random 100)`. However, this isn't what the current implementation does, as **`MACROEXPAND-1`** shows.

```
CL-USER> (macroexpand-1 '(do-primes (p 0 (random 100)) (format t "~d " p)))
(DO ((P (NEXT-PRIME 0) (NEXT-PRIME (1+ P))))
    ((> P (RANDOM 100)))
  (FORMAT T "~d " P))
T
```

When this expansion code is run, **`RANDOM`** will be called each time the end test for the loop is evaluated. Thus, instead of looping until `p` is greater than an initially chosen random number, this loop will iterate until it happens to draw a random number less than or equal to the current value of `p`. While the total number of iterations will still be random, it will be drawn from a much different distribution than the uniform distribution **`RANDOM`** returns.

This is a leak in the abstraction because, to use the macro correctly, the caller needs to be aware that the `end` form is going to be evaluated more than once. One way to plug this leak would be to simply define this as the behavior of `do-primes`. But that's not very satisfactory--you should try to observe the Principle of Least Astonishment when implementing macros. And programmers will typically expect the forms they pass to macros to be evaluated no more times than absolutely necessary.[6] Furthermore, since `do-primes` is built on the model of the standard macros, **`DOTIMES`** and **`DOLIST`**, neither of which causes any of the forms except those in the body to be evaluated more than once, most programmers will expect `do-primes` to behave similarly.

You can fix the multiple evaluation easily enough; you just need to generate code that evaluates `end` once and saves the value in a variable to be used later. Recall that in a **`DO`** loop, variables defined with an initialization form and no step form don't change from iteration to iteration. So you can fix the multiple evaluation problem with this definition:

```
(defmacro do-primes ((var start end) &body body)
  `(do ((ending-value ,end)
        (,var (next-prime ,start) (next-prime (1+ ,var))))
       ((> ,var ending-value))
     ,@body))
```

Unfortunately, this fix introduces two new leaks to the macro abstraction.

One new leak is similar to the multiple-evaluation leak you just fixed. Because the initialization forms for variables in a **DO** loop are evaluated in the order the variables are defined, when the macro expansion is evaluated, the expression passed as `end` will be evaluated before the expression passed as `start`, opposite to the order they appear in the macro call. This leak doesn't cause any problems when `start` and `end` are literal values like 0 and 19. But when they're forms that can have side effects, evaluating them out of order can once again run afoul of the Principle of Least Astonishment.

This leak is trivially plugged by swapping the order of the two variable definitions.

```
(defmacro do-primes ((var start end) &body body)
  `(do ((,var (next-prime ,start) (next-prime (1+ ,var)))
        (ending-value ,end))
       ((> ,var ending-value))
     ,@body))
```

The last leak you need to plug was created by using the variable name `ending-value`. The problem is that the name, which ought to be a purely internal detail of the macro implementation, can end up interacting with code passed to the macro or in the context where the macro is called. The following seemingly innocent call to `do-primes` doesn't work correctly because of this leak:

```
(do-primes (ending-value 0 10)
  (print ending-value))
```

Neither does this one:

```
(let ((ending-value 0))
  (do-primes (p 0 10)
    (incf ending-value p))
  ending-value)
```

Again, **MACROEXPAND-1** can show you the problem. The first call expands to this:

```
(do ((ending-value (next-prime 0) (next-prime (1+ ending-value)))
     (ending-value 10))
    ((> ending-value ending-value))
  (print ending-value))
```

Some Lisps may reject this code because `ending-value` is used twice as a variable name in the same **DO** loop. If not rejected outright, the code will loop forever since `ending-value` will never be greater than itself.

The second problem call expands to the following:

```
(let ((ending-value 0))
  (do ((p (next-prime 0) (next-prime (1+ p)))
       (ending-value 10))
      ((> p ending-value))
    (incf ending-value p))
  ending-value)
```

In this case the generated code is perfectly legal, but the behavior isn't at all what you want. Because the binding of `ending-value` established by the **`LET`** outside the loop is shadowed by the variable with the same name inside the **`DO`**, the form `(incf ending-value p)` increments the loop variable `ending-value` instead of the outer variable with the same name, creating another infinite loop.[7]

Clearly, what you need to patch this leak is a symbol that will never be used outside the code generated by the macro. You could try using a really unlikely name, but that's no guarantee. You could also protect yourself to some extent by using packages, as described in Chapter 21. But there's a better solution.

The function **`GENSYM`** returns a unique symbol each time it's called. This is a symbol that has never been read by the Lisp reader and never will be because it isn't interned in any package. Thus, instead of using a literal name like `ending-value`, you can generate a new symbol each time `do-primes` is expanded.

```
(defmacro do-primes ((var start end) &body body)
  (let ((ending-value-name (gensym)))
    `(do ((,var (next-prime ,start) (next-prime (1+ ,var)))
          (,ending-value-name ,end))
         ((> ,var ,ending-value-name))
       ,@body)))
```

Note that the code that calls **`GENSYM`** isn't part of the expansion; it runs as part of the macro expander and thus creates a new symbol each time the macro is expanded. This may seem a bit strange at first--`ending-value-name` is a variable whose value is the name of another variable. But really it's no different from the parameter `var` whose value is the name of a variable--the difference is the value of `var` was created by the reader when the macro form was read, and the value of `ending-value-name` is generated programmatically when the macro code runs.

With this definition the two previously problematic forms expand into code that works the way you want. The first form:

```
(do-primes (ending-value 0 10)
  (print ending-value))
```

expands into the following:

```
(do ((ending-value (next-prime 0) (next-prime (1+ ending-value)))
     (#:g2141 10))
    ((> ending-value #:g2141))
  (print ending-value))
```

Now the variable used to hold the ending value is the gensymed symbol, `#:g2141`. The name of the symbol, `G2141`, was generated by **GENSYM** but isn't significant; the thing that matters is the object identity of the symbol. Gensymed symbols are printed in the normal syntax for uninterned symbols, with a leading `#:`.

The other previously problematic form:

```
(let ((ending-value 0))
  (do-primes (p 0 10)
    (incf ending-value p))
  ending-value)
```

looks like this if you replace the `do-primes` form with its expansion:

```
(let ((ending-value 0))
  (do ((p (next-prime 0) (next-prime (1+ p)))
       (#:g2140 10))
      ((> p #:g2140))
    (incf ending-value p))
  ending-value)
```

Again, there's no leak since the `ending-value` variable bound by the **LET** surrounding the `do-primes` loop is no longer shadowed by any variables introduced in the expanded code.

Not all literal names used in a macro expansion will necessarily cause a problem--as you get more experience with the various binding forms, you'll be able to determine whether a given name is being used in a position that could cause a leak in a macro abstraction. But there's no real downside to using a gensymed name just to be safe.

With that fix, you've plugged all the leaks in the implementation of `do-primes`. Once you've gotten a bit of macro-writing experience under your belt, you'll learn to write macros with these kinds of leaks preplugged. It's actually fairly simple if you follow these rules of thumb:

- Unless there's a particular reason to do otherwise, include any subforms in the expansion in positions that will be evaluated in the same order as the subforms appear in the macro call.
- Unless there's a particular reason to do otherwise, make sure subforms are evaluated only once by creating a variable in the expansion to hold the value of evaluating the argument form and then using that variable anywhere else the value is needed in the expansion.
- Use **GENSYM** at macro expansion time to create variable names used in the expansion.

Macro-Writing Macros

Of course, there's no reason you should be able to take advantage of macros only when writing functions. The job of macros is to abstract away common syntactic patterns, and certain patterns come up again and again in writing macros that can also benefit from being abstracted away.

In fact, you've already seen one such pattern--many macros will, like the last version of `do-primes`, start with a **LET** that introduces a few variables holding gensymed symbols to be used in the macro's expansion. Since this is such a common pattern, why not abstract it away with its own macro?

In this section you'll write a macro, `with-gensyms`, that does just that. In other words, you'll write a macro-writing macro: a macro that generates code that generates code. While complex macro-writing macros can be a bit confusing until you get used to keeping the various levels of code clear in your mind, `with-gensyms` is fairly straightforward and will serve as a useful but not too strenuous mental limbering exercise.

You want to be able to write something like this:

```
(defmacro do-primes ((var start end) &body body)
  (with-gensyms (ending-value-name)
    `(do ((,var (next-prime ,start) (next-prime (1+ ,var)))
          (,ending-value-name ,end))
         ((> ,var ,ending-value-name))
       ,@body)))
```

and have it be equivalent to the previous version of `do-primes`. In other words, the `with-gensyms` needs to expand into a **`LET`** that binds each named variable, `ending-value-name` in this case, to a gensymed symbol. That's easy enough to write with a simple backquote template.

```
(defmacro with-gensyms ((&rest names) &body body)
  `(let ,(loop for n in names collect `(,n (gensym)))
     ,@body))
```

Note how you can use a comma to interpolate the value of the **`LOOP`** expression. The loop generates a list of binding forms where each binding form consists of a list containing one of the names given to `with-gensyms` and the literal code `(gensym)`. You can test what code the **`LOOP`** expression would generate at the REPL by replacing `names` with a list of symbols.

```
CL-USER> (loop for n in '(a b c) collect `(,n (gensym)))
((A (GENSYM)) (B (GENSYM)) (C (GENSYM)))
```

After the list of binding forms, the body argument to `with-gensyms` is spliced in as the body of the **`LET`**. Thus, in the code you wrap in a `with-gensyms` you can refer to any of the variables named in the list of variables passed to `with-gensyms`.

If you macro-expand the `with-gensyms` form in the new definition of `do-primes`, you should see something like this:

```
(let ((ending-value-name (gensym)))
  `(do ((,var (next-prime ,start) (next-prime (1+ ,var)))
        (,ending-value-name ,end))
       ((> ,var ,ending-value-name))
     ,@body))
```

Looks good. While this macro is fairly trivial, it's important to keep clear about when the different macros are expanded: when you compile the **`DEFMACRO`** of `do-primes`, the `with-gensyms` form is expanded into the code just shown and compiled. Thus, the compiled version of `do-primes` is just the same as if you had written the outer **`LET`** by hand. When you compile a function that uses `do-primes`, the code *generated* by `with-gensyms` runs generating the `do-primes` expansion, but `with-gensyms` itself isn't needed to compile a `do-primes` form since it has already been expanded, back when `do-primes` was compiled.

Another classic macro-writing MACRO: ONCE-ONLY

Another classic macro-writing macro is `once-only`, which is used to generate code that evaluates certain macro arguments once only and in a particular order. Using `once-only`, you could write `do-primes` almost as simply as the original leaky version, like this:

```
(defmacro do-primes ((var start end) &body body)
  (once-only (start end)
    `(do ((,var (next-prime ,start) (next-prime (1+ ,var))))
         ((> ,var ,end))
       ,@body)))
```

However, the implementation of `once-only` is a bit too involved for a blow-by-blow explanation, as it relies on multiple levels of backquoting and unquoting. If you really want to sharpen your macro chops, you can try to figure out how it works. It looks like this:

```
(defmacro once-only ((&rest names) &body body)
  (let ((gensyms (loop for n in names collect (gensym))))
    `(let (,@(loop for g in gensyms collect `(,g (gensym))))
      `(let (,,@(loop for g in gensyms for n in names collect ``(,,g ,,n)))
        ,(let (,@(loop for n in names for g in gensyms collect `(,n ,g)))
           ,@body)))))
```

Beyond Simple Macros

I could, of course, say a lot more about macros. All the macros you've seen so far have been fairly simple examples that save you a bit of typing but don't provide radical new ways of expressing things. In upcoming chapters you'll see examples of macros that allow you to express things in ways that would be virtually impossible without macros. You'll start in the very next chapter, in which you'll build a simple but effective unit test framework.

---

[1]As with functions, macros can also contain declarations, but you don't need to worry about those for now.

[2]**`APPEND`**, which I haven't discussed yet, is a function that takes any number of list arguments and returns the result of splicing them together into a single list.

[3]Another function, **`MACROEXPAND`**, keeps expanding the result as long as the first element of the resulting expansion is the name of the macro. However, this will often show you a much lower-level view of what the code is doing than you want, since basic control constructs such as **`DO`** are also implemented as macros. In other words, while it can be educational to see what your macro ultimately expands into, it isn't a very useful view into what your own macros are doing.

[4]If the macro expansion is shown all on one line, it's probably because the variable **`*PRINT-PRETTY*`** is **`NIL`**. If it is, evaluating `(setf *print-pretty* t)` should make the macro expansion easier to read.

[5]This is from *Joel on Software* by Joel Spolsky, also available at `http://www.joelonsoftware.com/ articles/LeakyAbstractions.html`. Spolsky's point in the essay is that

all abstractions leak to some extent; that is, there are no perfect abstractions. But that doesn't mean you should tolerate leaks you can easily plug.

[6]Of course, certain forms are supposed to be evaluated more than once, such as the forms in the body of a `do-primes` loop.

[7]It may not be obvious that this loop is necessarily infinite given the nonuniform occurrences of prime numbers. The starting point for a proof that it is in fact infinite is Bertrand's postulate, which says for any $n > 1$, there exists a prime p, $n < p < 2n$. From there you can prove that for any prime number, P less than the sum of the preceding prime numbers, the next prime, P', is also smaller than the original sum plus P.