10

Other Macro Pitfalls

Writing macros requires an extra degree of caution. A function is isolated in its own lexical world, but a macro, because it is expanded into the calling code, can give the user an unpleasant surprise unless it is carefully written. Chapter 9 explained variable capture, the biggest such surprise. This chapter discusses four more problems to avoid when defining macros.

10.1 Number of Evaluations

Several incorrect versions of `for` appeared in the previous chapter. Figure 10.1 shows two more, accompanied by a correct version for comparison.

Though not vulnerable to capture, the second `for` contains a bug. It will generate an expansion in which the form passed as `stop` will be evaluated on each iteration. In the best case, this kind of macro is inefficient, repeatedly doing what it could have done just once. If `stop` has side-effects, the macro could actually produce incorrect results. For example, this loop will never terminate, because the goal recedes on each iteration:

```
> (let ((x 2))
    (for (i 1 (incf x))
      (princ i)))
12345678910111213...
```

In writing macros like `for`, one must remember that the arguments to a macro are forms, not values. Depending on where they appear in the expansion, they

A correct version:

```
(defmacro for ((var start stop) &body body)
  (let ((gstop (gensym)))
    `(do ((,var ,start (1+ ,var))
          (,gstop ,stop))
         ((> ,var ,gstop))
       ,@body)))
```

Subject to multiple evaluations:

```
(defmacro for ((var start stop) &body body)
  `(do ((,var ,start (1+ ,var)))
       ((> ,var ,stop))
     ,@body))
```

Incorrect order of evaluation:

```
(defmacro for ((var start stop) &body body)
  (let ((gstop (gensym)))
    `(do ((,gstop ,stop)
          (,var ,start (1+ ,var)))
         ((> ,var ,gstop))
       ,@body)))
```

Figure 10.1: Controlling argument evaluation.

could be evaluated more than once. In this case, the solution is to bind a variable to the value returned by the `stop` form, and refer to the variable during the loop.

Unless they are clearly intended for iteration, macros should ensure that expressions are evaluated exactly as many times as they appear in the macro call. There are obvious cases in which this rule does not apply: the Common Lisp `or` would be much less useful (it would become a Pascal `or`) if all its arguments were always evaluated. But in such cases the user knows how many evaluations to expect. This isn't so with the second version of `for`: the user has no reason to suppose that the `stop` form is evaluated more than once, and in fact there is no reason that it should be. A macro written like the second version of `for` is most likely written that way by mistake.

Unintended multiple evaluation is a particularly difficult problem for macros built on `setf`. Common Lisp provides several utilities to make writing such macros easier. The problem, and the solution, are discussed in Chapter 12.

10.2 Order of Evaluation

The order in which expressions are evaluated, though not as important as the number of times they are evaluated, can sometimes become an issue. In Common Lisp function calls, arguments are evaluated left-to-right:

```
> (setq x 10)
10
> (+ (setq x 3) x)
6
```

and it is good practice for macros to do the same. Macros should usually ensure that expressions are evaluated in the same order that they appear in the macro call.

In Figure 10.1, the third version of `for` also contains a subtle bug. The parameter `stop` will be evaluated before `start`, even though they appear in the opposite order in the macro call:

```
> (let ((x 1))
    (for (i x (setq x 13))
     (princ i)))
13
NIL
```

This macro gives a disconcerting impression of going back in time. The evaluation of the `stop` form influences the value returned by the `start` form, even though the `start` form appears first textually.

The correct version of `for` ensures that its arguments will be evaluated in the order in which they appear:

```
> (let ((x 1))
    (for (i x (setq x 13))
     (princ i)))
12345678910111213
NIL
```

Now setting `x` in the `stop` form has no effect on the value returned by the previous argument.

Although the preceding example is a contrived one, there are cases in which this sort of problem might really happen, and such a bug would be extremely difficult to find. Perhaps few people would write code in which the evaluation of one argument to a macro influenced the value returned by another, but people may do by accident things that they would never do on purpose. As well as having to work right when used as intended, a utility must not mask bugs. If anyone wrote code like the foregoing examples, it would probably be by mistake, but the correct version of `for` will make the mistake easier to detect.

10.3 Non-functional Expanders

Lisp expects code which generates macro expansions to be purely functional, in the sense described in Chapter 3. Expander code should depend on nothing but the forms passed to it as arguments, and should not try to have an effect on the world except by returning values.

As of CLTL2 (p. 685), it is safe to assume that macro calls in compiled code will not be re-expanded at runtime. Otherwise, Common Lisp makes no guarantees about when, or how often, a macro call will be expanded. It is considered an error for the expansion of a macro to vary depending on either. For example, suppose we wanted to count the number of times some macro is used. We can't simply do a search through the source files, because the macro might be called in code which is generated by the program. We might therefore want to define the macro as follows:

```
(defmacro nil! (x)                                    ; wrong
  (incf *nil!s*)
  `(setf ,x nil))
```

With this definition, the global `*nil!s*` will be incremented each time a call to `nil!` is expanded. However, we are mistaken if we expect the value of this variable to tell us how often `nil!` was called. A given call can be, and often is, expanded more than once. For example, a preprocessor which performed transformations on your source code might have to expand the macro calls in an expression before it could decide whether or not to transform it.

As a general rule, expander code shouldn't depend on anything except its arguments. So any macro which builds its expansion out of strings, for example, should be careful not to assume anything about what the package will be at the time of expansion. This concise but rather pathological example,

```
(defmacro string-call (opstring &rest args)           ; wrong
  `(,(intern opstring) ,@args))
```

defines a macro which takes the print name of an operator and expands into a call to it:

```
> (defun our+ (x y) (+ x y))
OUR+
> (string-call "OUR+" 2 3)
5
```

The call to `intern` takes a string and returns the corresponding symbol. However, if we omit the optional package argument, it does so in the current package. The

expansion will thus depend on the package at the time the expansion is generated, and unless `our+` is visible in that package, the expansion will be a call to an unknown function.

Miller and Benson's *Lisp Style and Design* mentions one particularly ugly ○ example of problems arising from side-effects in expander code. In Common Lisp, as of CLTL2 (p. 78), the lists bound to `&rest` parameters are not guaranteed to be freshly made. They may share structure with lists elsewhere in the program. In consequence, you shouldn't destructively modify `&rest` parameters, because you don't know what else you'll be modifying.

This possibility affects both functions and macros. With functions, problems would arise when using `apply`. In a valid implementation of Common Lisp the following could happen. Suppose we define a function `et-al`, which returns a list of its arguments with `et al` added to the end:

```
(defun et-al (&rest args)
  (nconc args (list 'et 'al)))
```

If we called this function normally, it would seem to work fine:

```
> (et-al 'smith 'jones)
(SMITH JONES ET AL)
```

However, if we called it via `apply`, it could alter existing data structures:

```
> (setq greats '(leonardo michelangelo))
(LEONARDO MICHELANGELO)
> (apply #'et-al greats)
(LEONARDO MICHELANGELO ET AL)
> greats
(LEONARDO MICHELANGELO ET AL)
```

At least, a valid implementation of Common Lisp could do this, though so far none seems to.

For macros, the danger is greater. A macro which altered an `&rest` parameter could thereby alter the macro call. That is, you could end up with inadvertently self-rewriting programs. The danger is also more real—it actually happens under existing implementations. If we define a macro which nconcs something onto its `&rest` argument[1]

```
(defmacro echo (&rest args)
  `',(nconc args (list 'amen)))
```

[1] `` `',(foo) `` is equivalent to `` `(quote ,(foo)) ``.

and then define a function that calls it:

```
(defun foo () (echo x))
```

in one widely used Common Lisp, the following will happen:

```
> (foo)
(X AMEN AMEN)
> (foo)
(X AMEN AMEN AMEN)
```

Not only does `foo` return the wrong result, it returns a different result each time, because each macroexpansion alters the definition of `foo`.

This example also illustrates the point made earlier about multiple expansions of a given macro call. In this particular implementation, the first call to `foo` returns a lists with *two* `amen`s. For some reason this implementation expanded the macro call once when `foo` was defined, as well as once in each of the succeeding calls.

It would be safer to have defined `echo` as:

```
(defmacro echo (&rest args)
  ‘’(,@args amen))
```

because a comma-at is equivalent to an `append` rather than an `nconc`. After redefining this macro, `foo` will have to be redefined as well, even if it wasn't compiled, because the previous version of `echo` caused it to be rewritten.

In macros, it's not only `&rest` parameters which are subject to this danger. Any macro argument which is a list should be left alone. If we define a macro which modifies one of its arguments, and a function which calls it,

```
(defmacro crazy (expr) (nconc expr (list t)))

(defun foo () (crazy (list)))
```

then the source code of the calling function could get modified, as happens in one implementation the first time we call it:

```
> (foo)
(T T)
```

This happens in compiled as well as interpreted code.

The upshot is, don't try to avoid consing by destructively modifying parameter list structure. The resulting programs won't be portable, if they run at all. If you want to avoid consing in a function which takes a variable number of arguments,

one solution is to use a macro, and thereby shift the consing forward to compile-time. For this application of macros, see Chapter 13.

One should also avoid performing destructive operations on the expressions returned by macro expanders, if these expressions incorporate quoted lists. This is not a restriction on macros *per se*, but an instance of the principle outlined in Section 3.3.

10.4 Recursion

Sometimes it's natural to define a function recursively. There's something inherently recursive about a function like this:

```
(defun our-length (x)
  (if (null x)
      0
      (1+ (our-length (cdr x)))))
```

This definition somehow seems more natural (though probably slower) than the iterative equivalent:

```
(defun our-length (x)
  (do ((len 0 (1+ len))
       (y x (cdr y)))
      ((null  y) len)))
```

A function which is neither recursive, nor part of some mutually recursive set of functions, can be transformed into a macro by the simple technique described in Section 7.10. However, just inserting backquotes and commas won't work with a recursive function. Let's take the built-in `nth` as an example. (For simplicity, our versions of `nth` will do no error-checking.) Figure 10.2 shows a mistaken attempt to define `nth` as a macro. Superficially, `nthb` appears to be equivalent to `ntha`, but a program containing a call to `nthb` would not compile, because the expansion of the call would never terminate.

In general, it's fine for macros to contain references to other macros, so long as expansion terminates somewhere. The trouble with `nthb` is that every expansion contains a reference to `nthb` itself. The function version, `ntha`, terminates because it recurses on the *value* of `n`, which is decremented on each recursion. But macroexpansion only has access to forms, not to their values. When the compiler tries to macroexpand, say, `(nthb x y)`, the first expansion will yield

```
(if (= x 0)
    (car y)
    (nthb (- x 1) (cdr y)))
```

This will work:

```
(defun ntha (n lst)
  (if (= n 0)
      (car lst)
      (ntha (- n 1) (cdr lst))))
```

This won't compile:

```
(defmacro nthb (n lst)
  `(if (= ,n 0)
       (car ,lst)
       (nthb (- ,n 1) (cdr ,lst))))
```

Figure 10.2: Mistaken analogy to a recursive function.

which will in turn expand into:

```
(if (= x 0)
    (car y)
    (if (= (- x 1) 0)
        (car (cdr y))
        (nthb (- (- x 1) 1) (cdr (cdr y)))))
```

and so on into an infinite loop. It's fine for a macro to expand into a call to itself, just so long as it doesn't always do so.

The dangerous thing about recursive macros like `nthb` is that they usually work fine under the interpreter. Then when you finally have your program working and you try to compile it, it won't even compile. Not only that, but there will usually be no indication that the problem is due to a recursive macro; the compiler will simply go into an infinite loop and leave you to figure out what went wrong.

In this case, `ntha` is tail-recursive. A tail-recursive function can easily be transformed into an iterative equivalent, and *then* used as a model for a macro. A macro like `nthb` could be written

```
(defmacro nthc (n lst)
  `(do ((n2 ,n (1- n2))
        (lst2 ,lst (cdr lst2)))
       ((= n2 0) (car lst2))))
```

so it is not impossible in principle to duplicate a recursive function with a macro. However, transforming more complicated recursive functions could be difficult, or even impossible.

```
(defmacro nthd (n lst)
  `(nth-fn ,n ,lst))

(defun nth-fn (n lst)
  (if (= n 0)
      (car lst)
      (nth-fn (- n 1) (cdr lst))))

(defmacro nthe (n lst)
  `(labels ((nth-fn (n lst)
              (if (= n 0)
                  (car lst)
                  (nth-fn (- n 1) (cdr lst)))))
     (nth-fn ,n ,lst)))
```

Figure 10.3: Two ways to fix the problem.

Depending on what you need a macro for, you may find it sufficient to use instead a combination of macro and function. Figure 10.3 shows two ways to make what appears to be a recursive macro. The first strategy, embodied by `nthd`, is simply to make the macro expand into a call to a recursive function. If, for example, you need a macro only to save users the trouble of quoting arguments, then this approach should suffice.

If you need a macro because you want its whole expansion to be inserted into the lexical environment of the macro call, then you would more likely want to follow the example of `nthe`. The built-in `labels` special form (Section 2.7) creates a local function definition. While each expansion of `nthc` will call the globally defined function `nth-fn`, each expansion of `nthe` will have its own version of such a function within it.

Although you can't translate a recursive function directly into a macro, you can write a macro whose expansion is recursively generated. The expansion function of a macro is a regular Lisp function, and can of course be recursive. For example, if we were to define a version of the built-in `or`, we would want to use a recursive expansion function.

Figure 10.4 shows two ways of defining recursive expansion functions for `or`. The macro `ora` calls the recursive function `or-expand` to generate its expansion. This macro will work, and so will the equivalent `orb`. Although `orb` recurses, it recurses on the arguments to the macro (which are available at macroexpansion time), not upon their values (which aren't). It might seem as if the expansion would contain a reference to `orb` itself, but the call to `orb` generated by one

```
(defmacro ora (&rest args)
  (or-expand args))

(defun or-expand (args)
  (if (null args)
       nil
      (let ((sym (gensym)))
        `(let ((,sym ,(car args)))
           (if ,sym
               ,sym
               ,(or-expand (cdr args)))))))

(defmacro orb (&rest args)
  (if (null args)
      nil
      (let ((sym (gensym)))
        `(let ((,sym ,(car args)))
           (if ,sym
               ,sym
               (orb ,@(cdr args)))))))
```

Figure 10.4: Recursive expansion functions.

macroexpansion step will be replaced by a `let` in the next one, yielding in the final expansion nothing more than a nested stack of `lets`; `(orb x y)` expands into code equivalent to:

```
(let ((g2 x))
  (if g2
      g2
      (let ((g3 y))
        (if g3 g3 nil))))
```

In fact, `ora` and `orb` are equivalent, and which style to use is just a matter of personal preference.