9

Variable Capture

Macros are vulnerable to a problem called *variable capture.* Variable capture occurs when macroexpansion causes a name clash: when some symbol ends up referring to a variable from another context. Inadvertent variable capture can cause extremely subtle bugs. This chapter is about how to foresee and avoid them. However, intentional variable capture is a useful programming technique, and Chapter 14 is full of macros which rely on it.

9.1 Macro Argument Capture

A macro vulnerable to unintended variable capture is a macro with a bug. To avoid writing such macros, we must know precisely when capture can occur. Instances of variable capture can be traced to one of two situations: macro argument capture and free symbol capture. In argument capture, a symbol passed as an argument in the macro call inadvertently refers to a variable established by the macro expansion itself. Consider the following definition of the macro `for`, which iterates over a body of expressions like a Pascal `for` loop:

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

This macro looks correct at first sight. It even seems to work fine:

```
> (for (x 1 5)
    (princ x))
12345
NIL
```

Indeed, the error is so subtle that we might use this version of the macro hundreds of times and have it always work perfectly. Not if we call it this way, though:

```
(for (limit 1 5)
  (princ limit))
```

We might expect this expression to have the same effect as the one before. But it doesn't print anything; it generates an error. To see why, we look at its expansion:

```
(do ((limit 1 (1+ limit))
     (limit 5))
    ((> limit limit))
  (princ limit))
```

Now it's obvious what goes wrong. There is a name clash between a symbol local to the macro expansion and a symbol passed as an argument to the macro. The macroexpansion *captures* `limit`. It ends up occurring twice in the same `do`, which is illegal.

Errors caused by variable capture are rare, but what they lack in frequency they make up in viciousness. This capture was comparatively mild—here, at least, we got an error. More often than not, a capturing macro would simply yield incorrect results with no indication that anything was wrong. In this case,

```
> (let ((limit 5))
    (for (i 1 10)
      (when (> i limit)
        (princ i))))
NIL
```

the resulting code quietly does nothing.

9.2 Free Symbol Capture

Less frequently, the macro definition itself contains a symbol which inadvertently refers to a binding in the environment where the macro is expanded. Suppose some program, instead of printing warnings to the user as they arise, wants to store the warnings in a list, to be examined later. One person writes a macro `gripe`, which takes a warning message and adds it to a global list, `w`:

```
(defvar w nil)

(defmacro gripe (warning)                              ; wrong
  `(progn (setq w (nconc w (list ,warning)))
          nil))
```

Someone else then wants to write a function `sample-ratio`, to return the ratio of the lengths of two lists. If either of the lists has less than two elements, the function is to return `nil` instead, also issuing a warning that it was called on a statistically insignificant case. (Actual warnings could be more informative, but their content isn't relevant to this example.)

```
(defun sample-ratio (v w)
  (let ((vn (length v)) (wn (length w)))
    (if (or (< vn 2) (< wn 2))
        (gripe "sample < 2")
        (/ vn wn))))
```

If sample-ratio is called with `w = (b)`, then it will want to warn that one of its arguments, with only one element, is statistically insignificant. But when the call to `gripe` is expanded, it will be as if `sample-ratio` had been defined:

```
(defun sample-ratio (v w)
  (let ((vn (length v)) (wn (length w)))
    (if (or (< vn 2) (< wn 2))
        (progn (setq w (nconc w (list "sample < 2")))
               nil)
        (/ vn wn))))
```

The problem here is that `gripe` is used in a context where `w` has its own local binding. The warning, instead of being saved in the global warning list, will be nconced onto the end of one of the parameters of `sample-ratio`. Not only is the warning lost, but the list `(b)`, which is probably used as data elsewhere in the program, will have an extraneous string appended to it:

```
> (let ((lst '(b)))
    (sample-ratio nil lst)
    lst)
(B "sample < 2")
> w
NIL
```

9.3 When Capture Occurs

It's asking a lot of the macro writer to be able to look at a macro definition and foresee all the possible problems arising from these two types of capture. Variable capture is a subtle matter, and it takes some experience to anticipate all the ways a capturable symbol could wreak mischief in a program. Fortunately, you can detect and eliminate capturable symbols in your macro definitions without having to think about *how* their capture could send your program awry. This section provides a straightforward rule for detecting capturable symbols. The remaining sections of this chapter explain techniques for eliminating them.

The rule for defining a capturable variable depends on some subordinate concepts, which must be defined first:

Free: A symbol *s* occurs free in an expression when it is used as a variable in that expression, but the expression does not create a binding for it.

In the following expression,

```
(let ((x y) (z 10))
  (list w x z))
```

`w`, `x` and `z` all occur free within the `list` expression, which establishes no bindings. However, the enclosing `let` expression establishes bindings for `x` and `z`, so within the `let` as a whole, only `y` and `w` occur free. Note that in

```
(let ((x x))
  x)
```

the second instance of `x` is free—it's not within the scope of the new binding being established for `x`.

Skeleton: The skeleton of a macro expansion is the whole expansion, minus anything which was part of an argument in the macro call.

If `foo` is defined:

```
(defmacro foo (x y)
  `(/ (+ ,x 1) ,y))
```

and called thus:

```
(foo (- 5 2) 6)
```

then it yields the macro expansion:

```
(/ (+ (- 5 2) 1) 6)
```

The skeleton of this expansion is the above expression with holes where the parameters `x` and `y` got inserted:

```
(/ (+         1)  )
```

With these two concepts defined, it's possible to state a concise rule for detecting capturable symbols:

Capturable: A symbol is capturable in some macro expansion if (a) it occurs free in the skeleton of the macro expansion, or (b) it is bound by a part of the skeleton in which arguments passed to the macro are either bound or evaluated.

Some examples will show the implications of this rule. In the simplest case:

```
(defmacro cap1 ()
  '(+ x 1))
```

`x` is capturable because it will occur free in the skeleton. That's what caused the bug in `gripe`. In this macro:

```
(defmacro cap2 (var)
  `(let ((x ...)
         (,var ...))
     ...))
```

`x` is capturable because it is bound in an expression where an argument to the macro call will also be bound. (That's what went wrong in `for`.) Likewise for the following two macros

```
(defmacro cap3 (var)
  `(let ((x ...))
     (let ((,var ...))
       ...)))

(defmacro cap4 (var)
  `(let ((,var ...))
     (let ((x ...))
       ...)))
```

in both of which `x` is capturable. However, if there is no context in which the binding of `x` and the variable passed as an argument will both be visible, as in

```
(defmacro safe1 (var)
  `(progn (let ((x 1))
            (print x))
          (let ((,var 1))
            (print ,var))))
```

then x won't be capturable. Not all variables bound by the skeleton are at risk. However, if arguments to the macro call are evaluated within a binding established by the skeleton,

```
(defmacro cap5 (&body body)
  `(let ((x ...))
     ,@body))
```

then variables so bound are at risk of capture: in cap5, x is capturable. In this case, though,

```
(defmacro safe2 (expr)
  `(let ((x ,expr))
     (cons x 1)))
```

x is *not* capturable, because when the argument passed to expr is evaluated, the new binding of x won't be visible. Note also that it's only the binding of skeletal variables we have to worry about. In this macro

```
(defmacro safe3 (var &body body)
  `(let ((,var ...))
     ,@body))
```

no symbol is at risk of inadvertent capture (assuming that the user expects that the first argument will be bound).

Now let's look at the original definition of for in light of the new rule for identifying capturable symbols:

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

It turns out now that this definition of for is vulnerable to capture in *two* ways: limit could be passed as the first argument to for, as in the original example:

```
(for (limit 1 5)
  (princ limit))
```

but it's just as dangerous if `limit` occurs in the body of the loop:

```
(let ((limit 0))
  (for (x 1 10)
    (incf limit x))
  limit)
```

Someone using `for` in this way would be expecting his own binding of `limit` to be the one incremented in the loop, and the expression as a whole to return 55; in fact, only the binding of `limit` generated by the skeleton of the expansion will be incremented:

```
(do ((x 1 (1+ x))
     (limit 10))
    ((> x limit))
  (incf limit x))
```

and since that's the one which controls iteration, the loop won't even terminate.

The rules presented in this section should be used with the reservation that they are intended only as a guide. They are not even formally stated, let alone formally correct. The problem of capture is a vaguely defined one, since it depends on expectations. For example, in an expression like

```
(let ((x 1)) (list x))
```

we don't regard it as an error that when `(list x)` is evaluated, `x` will refer to a new variable. That's what `let` is supposed to do. The rules for detecting capture are also imprecise. You could write macros which passed these tests, and which still would be vulnerable to unintended capture. For example,

```
(defmacro pathological (&body body)                    ; wrong
  (let* ((syms (remove-if (complement #'symbolp)
                          (flatten body)))
         (var (nth (random (length syms))
                   syms)))
    `(let ((,var 99))
       ,@body)))
```

When this macro is called, the expressions in the body will be evaluated as if in a `progn`—but one random variable within the body may have a different value. This is clearly capture, but it passes our tests, because the variable does not occur in the skeleton. In practice, though, the rules will work nearly all the time: one rarely (if ever) wants to write a macro like the example above.

Vulnerable to capture:

```
(defmacro before (x y seq)
  `(let ((seq ,seq))
     (< (position ,x seq)
        (position ,y seq))))
```

A correct version:

```
(defmacro before (x y seq)
  `(let ((xval ,x) (yval ,y) (seq ,seq))
     (< (position xval seq)
        (position yval seq))))
```

Figure 9.1: Avoiding capture with `let`.

9.4 Avoiding Capture with Better Names

The first two sections divided instances of variable capture into two types: argument capture, where a symbol used in an argument is caught by a binding established by the macro skeleton, and free symbol capture, where a free symbol in a macroexpansion is caught by a binding in force where the macro is expanded. The latter cases are usually dealt with simply by giving global variables distinguished names. In Common Lisp, it is traditional to give global variables names which begin and end with asterisks. The variable defining the current package is called `*package*`, for example. (Such a name may be pronounced "star-package-star" to emphasize that it is not an ordinary variable.)

So really it was the responsibility of the author of `gripe` to store warnings in a variable called something like `*warnings*`, rather than just `w`. If the author of `sample-ratio` had used `*warnings*` as a parameter, then he would deserve every bug he got, but he can't be blamed for thinking that it would be safe to call a parameter `w`.

9.5 Avoiding Capture by Prior Evaluation

Sometimes argument capture can be cured simply by evaluating the endangered arguments outside of any bindings created by the macroexpansion. The simplest cases can be handled by beginning the macro with a `let` expression. Figure 9.1 contains two versions of the macro `before`, which takes two objects and a sequence, and returns true iff the first object occurs before the second in the

sequence.[1] The first definition is incorrect. Its initial `let` ensures that the form passed as `seq` is only evaluated once, but it is not sufficient to avoid the following problem:

```
> (before (progn (setq seq '(b a)) 'a)
          'b
          '(a b))
NIL
```

This amounts to asking "Is `a` before `b` in `(a b)`?" If `before` were correct, it would return true. Macroexpansion shows what really happens: the evaluation of the first argument to `<` rearranges the list to be searched in the second.

```
(let ((seq '(a b)))
  (< (position (progn (setq seq '(b a)) 'a)
               seq)
     (position 'b seq)))
```

To avoid this problem, it will suffice to evaluate all the arguments first in one big `let`. The second definition in Figure 9.1 is thus safe from capture.

Unfortunately, the `let` technique works only in a narrow range of cases: macros where

1. all the arguments at risk of capture are evaluated exactly once, and
2. none of the arguments need to be evaluated in the scope of bindings established by the macro skeleton.

This rules out a great many macros. The proposed `for` macro violates both conditions. However, we can use a variation of this scheme to make macros like `for` safe from capture: to wrap its `body` forms within a lambda-expression outside of any locally created bindings.

Some macros, including those for iteration, yield expansions where expressions appearing in the macro call will be evaluated within newly established bindings. In the definition of `for`, for example, the body of the loop must be evaluated within a `do` created by the macro. Variables occurring in the body of the loop are thus vulnerable to capture by bindings established by the `do`. We can protect variables in the body from such capture by wrapping the body in a closure, and, within the loop, instead of inserting the expressions themselves, simply funcalling the closure.

○ Figure 9.2 shows a version of `for` which uses this technique. Since the closure

[1] This macro is used only as an example. Really it should neither be implemented as a macro, nor use the inefficient algorithm that it does. For a proper definition, see page 50.

Vulnerable to capture:

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

A correct version:

```
(defmacro for ((var start stop) &body body)
  `(do ((b #'(lambda (,var) ,@body))
        (count ,start (1+ count))
        (limit ,stop))
       ((> count limit))
     (funcall b count)))
```

Figure 9.2: Avoiding capture with a closure.

is the first thing made by the expansion of a `for`, free symbols occurring in the body will all refer to variables in the environment of the macro call. Now the `do` communicates with its body through the parameters of the closure. All the closure needs to know from the `do` is the number of the current iteration, so it has only one parameter, the symbol specified as the index variable in the macro call.

The technique of wrapping expressions in lambdas is not a universal remedy. You can use it to protect a body of code, but closures won't be any use when, for example, there is a risk of the same variable being bound twice by the same `let` or `do` (as in our original broken `for`). Fortunately, in this case, by rewriting `for` to package its body in a closure, we also eliminated the need for the `do` to establish bindings for the `var` argument. The `var` argument of the old `for` became the parameter of the closure and could be replaced in the `do` by an actual symbol, `count`. So the new definition of `for` is completely immune from capture, as the test in Section 9.3 will show.

The disadvantage of using closures is that they might be less efficient. We could be introducing another function call. Potentially worse, if the compiler doesn't give the closure dynamic extent, space for it will have to be allocated in the heap at runtime.

9.6 Avoiding Capture with Gensyms

There is one certain way to avoid macro argument capture: replacing capturable symbols with gensyms. In the original version of `for`, problems arise when two symbols inadvertently have the same name. If we want to avoid the possibility that a macro skeleton will contain a symbol also used by the calling code, we might hope to get away with using only strangely named symbols in macro definitions:

```
(defmacro for ((var start stop) &body body)           ; wrong
  `(do ((,var ,start (1+ ,var))
        (xsf2jsh ,stop))
       ((> ,var xsf2jsh))
     ,@body))
```

but this is no solution. It doesn't eliminate the bug, just makes it less likely to show. And not so very less likely at that—it's still possible to imagine conflicts arising in nested instances of the same macro.

We need some way to *ensure* that a symbol is unique. The Common Lisp ○ function `gensym` exists just for this purpose. It returns a symbol, called a *gensym*, which is guaranteed not to be `eq` to any symbol either typed in or constructed by a program.

How can Lisp promise this? In Common Lisp, each package keeps a list of all the symbols known in that package. (For an introduction to packages, see page 381.) A symbol which is on the list is said to be *interned* in the package. Each call to `gensym` returns a unique, uninterned symbol. And since every symbol seen by `read` gets interned, no one could type anything identical to a gensym. Thus, if you begin the expression

```
(eq (gensym) ...
```

there is no way to complete it that will cause it to return true.

Asking `gensym` to make you a symbol is like taking the approach of choosing a strangely named symbol one step further—`gensym` will give you a symbol whose name isn't even in the phone book. When Lisp has to display a gensym,

```
> (gensym)
#:G47
```

what it prints is really just Lisp's equivalent of "John Doe," an arbitrary name made up for something whose name is irrelevant. And to be sure that we don't have any illusions about this, gensyms are displayed preceded by a sharp-colon, a special read-macro which exists just to cause an error if we ever try to read the gensym in again.

Vulnerable to capture:

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

A correct version:

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

Figure 9.3: Avoiding capture with `gensym`.

In CLTL2 Common Lisp, the number in a gensym's printed representation comes from `*gensym-counter*`, a global variable always bound to an integer. By resetting this counter we can cause two gensyms to print the same

```
> (setq x (gensym))
#:G48
> (setq *gensym-counter* 48 y (gensym))
#:G48
> (eq x y)
NIL
```

but they won't be identical.

Figure 9.3 contains a correct definition of `for` using gensyms. Now there is no `limit` to clash with symbols in forms passed to the macro. It has been replaced by a symbol gensymed on the spot. In each expansion of the macro, the place of `limit` will be taken by a unique symbol created at expansion-time.

The correct definition of `for` is a complicated one to produce on the first try. Finished code, like a finished theorem, often covers up a lot of trial and error. So don't worry if you have to write several versions of a macro. To begin writing macros like `for`, you may want to write the first version without thinking about variable capture, and then to go back and make gensyms for symbols which could be involved in captures.

9.7 Avoiding Capture with Packages

To some extent, it is possible to avoid capture by defining macros in their own package. If you create a `macros` package and define `for` there, you can even use the definition given first

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

and call it safely from any other package. If you call `for` from another package, say `mycode`, then even if you do use `limit` as the first argument, it will be `mycode::limit`—a distinct symbol from `macros::limit`, which occurs in the macro skeleton.

However, packages do not provide a very general solution to the problem of capture. In the first place, macros are an integral part of some programs, and it would be inconvenient to have to separate them in their own package. Second, this approach offers no protection against capture by other code in the `macros` package.

9.8 Capture in Other Name-Spaces

The previous sections have spoken of capture as if it were a problem which afflicted variables exclusively. Although most capture is variable capture, the problem can arise in Common Lisp's other name-spaces as well.

Functions may also be locally bound, and function bindings are equally liable to inadvertent capture. For example:

```
> (defun fn (x) (+ x 1))
FN
> (defmacro mac (x) `(fn ,x))
MAC
> (mac 10)
11
> (labels ((fn (y) (- y 1)))
    (mac 10))
9
```

As predicted by the capture rule, the `fn` which occurs free in the skeleton of `mac` is at risk of capture. When `fn` is locally rebound, `mac` returns a different value than it does generally.

What to do about this case? When the symbol at risk of capture is the name of a built-in function or macro, then it's reasonable to do nothing. In CLTL2 (p. 260) ○ if the name of anything built-in is given a local function or macro binding, "the consequences are undefined." So it wouldn't matter what your macro did—anyone who rebinds built-in functions is going to have problems with more than just your macros.

Otherwise, you can protect function names against macro argument capture the same way you would protect variable names: by using gensyms as names for any functions given local definitions by the macro skeleton. Avoiding free symbol capture, as in the case above, is a bit more difficult. The way to protect variables against free symbol capture was to give them distinctly global names: e.g. `*warnings*` instead of `w`. This solution is not practical for functions, because there is no convention for distinguishing the names of global functions—most functions are global. If you're concerned about a macro being called in an environment where a function it needs might be locally redefined, the best solution is probably to put your code in a distinct package.

Block-names are also liable to capture, as are the tags used by `go` and `throw`. When your macros need such symbols, you should use gensyms, as in the definition of `our-do` on page 98.

Remember also that operators like `do` are implicitly enclosed in a block named `nil`. Thus a `return` or `return-from nil` within a `do` returns from the `do`, not the containing expression:

```
> (block nil
    (list 'a
          (do ((x 1 (1+ x)))
              (nil)
            (if (> x 5)
                (return-from nil x)
                (princ x)))))
12345
(A 6)
```

If `do` didn't create a block named `nil`, this example would have returned just `6`, rather than `(A 6)`.

The implicit block in `do` is not a problem, because `do` is advertised to behave this way. However, you should realize that if you write macros which expand into `do`s, they will capture the block name `nil`. In a macro like `for`, a `return` or `return-from nil` will return from the `for` expression, not the enclosing block.

9.9 Why Bother?

Some of the preceding examples are pretty pathological. Looking at them, one might be tempted to say "variable capture is so unlikely—why even worry about it?" There are two ways to answer this question. One is with another question: why write programs with small bugs when you could write programs with no bugs?

The longer answer is to point out that in real applications it's dangerous to assume anything about the way your code will be used. Any Lisp program has what is now called an "open architecture." If you're writing code other people will use, they may use it in ways you'd never anticipate. And it's not just people you have to worry about. Programs write programs too. It may be that no human would write code like

```
(before (progn (setq seq '(b a)) 'a)
        'b
        '(a b))
```

but code generated by programs often looks like this. Even if individual macros generate simple and reasonable-looking expansions, once you begin to nest macro calls, the expansions can become large programs which look like nothing any human would write. Under such circumstances, it is worth defending against cases, however contrived, which might make your macros expand incorrectly.

In the end, avoiding variable capture is not very difficult anyway. It soon becomes second-nature. The classic Common Lisp `defmacro` is like a cook's knife: an elegant idea which seems dangerous, but which experts use with confidence.