14

Anaphoric Macros

Chapter 9 treated variable capture exclusively as a problem—as something which happens inadvertently, and which can only affect programs for the worse. This chapter will show that variable capture can also be used constructively. There are some useful macros which couldn't be written without it.

It's not uncommon in a Lisp program to want to test whether an expression returns a non-nil value, and if so, to do something with the value. If the expression is costly to evaluate, then one must normally do something like this:

```
(let ((result (big-long-calculation)))
  (if result
      (foo result)))
```

Wouldn't it be easier if we could just say, as we would in English:

```
(if (big-long-calculation)
    (foo it))
```

By taking advantage of variable capture, we can write a version of `if` which works just this way.

14.1 Anaphoric Variants

In natural language, an *anaphor* is an expression which refers back in the conversation. The most common anaphor in English is probably "it," as in "Get the wrench and put it on the table." Anaphora are a great convenience in everyday language—imagine trying to get along without them—but they don't appear much in programming languages. For the most part, this is good. Anaphoric expressions are often genuinely ambiguous, and present-day programming languages are not designed to handle ambiguity.

However, it is possible to introduce a very limited form of anaphora into Lisp programs without causing ambiguity. An anaphor, it turns out, is a lot like a captured symbol. We can use anaphora in programs by designating certain symbols to serve as pronouns, and then writing macros intentionally to capture these symbols.

In the new version of `if`, the symbol `it` is the one we want to capture. The anaphoric if, called `aif` for short, is defined as follows:

```
(defmacro aif (test-form then-form &optional else-form)
  `(let ((it ,test-form))
     (if it ,then-form ,else-form)))
```

and used as in the previous example:

```
(aif (big-long-calculation)
     (foo it))
```

When you use an `aif`, the symbol `it` is left bound to the result returned by the test clause. In the macro call, `it` seems to be free, but in fact the expression `(foo it)` will be inserted by expansion of the `aif` into a context in which the symbol `it` is bound:

```
(let ((it (big-long-calculation)))
  (if it (foo it) nil))
```

So a symbol which looks free in the source code is left bound by the macroexpansion. All the anaphoric macros in this chapter use some variation of the same technique.

Figure 14.1 contains anaphoric variants of several Common Lisp operators. After `aif` comes `awhen`, the obvious anaphoric variant of `when`:

```
(awhen (big-long-calculation)
  (foo it)
  (bar it))
```

Both `aif` and `awhen` are frequently useful, but `awhile` is probably unique among the anaphoric macros in being more often needed than its regular cousin, `while` (defined on page 91). Macros like `while` and `awhile` are typically used in situations where a program needs to poll some outside source. And when you are polling a source, unless you are simply waiting for it to change state, you will usually want to do something with the object you find there:

```
(defmacro aif (test-form then-form &optional else-form)
  `(let ((it ,test-form))
     (if it ,then-form ,else-form)))

(defmacro awhen (test-form &body body)
  `(aif ,test-form
        (progn ,@body)))

(defmacro awhile (expr &body body)
  `(do ((it ,expr ,expr))
       ((not it))
     ,@body))

(defmacro aand (&rest args)
  (cond ((null args) t)
        ((null (cdr args)) (car args))
        (t `(aif ,(car args) (aand ,@(cdr args))))))

(defmacro acond (&rest clauses)
  (if (null clauses)
      nil
      (let ((cl1 (car clauses))
            (sym (gensym)))
        `(let ((,sym ,(car cl1)))
           (if ,sym
               (let ((it ,sym)) ,@(cdr cl1))
               (acond ,@(cdr clauses)))))))
```

Figure 14.1: Anaphoric variants of Common Lisp operators.

```
(awhile (poll *fridge*)
  (eat it))
```

The definition of `aand` is a bit more complicated than the preceding ones. It provides an anaphoric version of `and`; during the evaluation of each of its arguments, `it` will be bound to the value returned by the previous argument.[1] In practice, `aand` tends to be used in programs which make conditional queries, as in:

[1]Although one tends to think of `and` and `or` together, there would be no point in writing an anaphoric version of `or`. An argument in an `or` expression is evaluated only if the previous argument evaluated to `nil`, so there would be nothing useful for an anaphor to refer to in an `aor`.

```
(aand (owner x) (address it) (town it))
```

which returns the town (if there is one) of the address (if there is one) of the owner (if there is one) of `x`. Without `aand`, this expression would have to be written

```
(let ((own (owner x)))
  (if own
      (let ((adr (address own)))
        (if adr (town adr)))))
```

The definition of `aand` shows that the expansion will vary depending on the number of arguments in the macro call. If there are no arguments, then `aand`, like the regular `and`, should simply return `t`. Otherwise the expansion is generated recursively, each step yielding one layer in a chain of nested `aif`s:

```
(aif ⟨first argument⟩
     ⟨expansion for rest of arguments⟩)
```

The expansion of an `aand` must terminate when there is one argument left, instead of working its way down to `nil` like most recursive functions. If the recursion continued until no conjuncts remained, the expansion would always be of the form:

```
(aif ⟨c1⟩
     ⋮
     (aif ⟨cn⟩
          t)...)
```

Such an expression would always return `t` or `nil`, and the example above wouldn't work as intended.

Section 10.4 warned that if a macro always yielded an expansion containing a call to itself, the expansion would never terminate. Though recursive, `aand` is safe because in the base case its expansion doesn't refer to `aand`.

The last example, `acond`, is meant for those cases where the remainder of a `cond` clause wants to use the value returned by the test expression. (This situation arises so often that some Scheme implementations provide a way to use the value returned by the test expression in a `cond` clause.)

In the expansion of an `acond` clause, the result of the test expression will initially be kept in a gensymed variable, in order that the symbol `it` may be bound only within the remainder of the clause. When macros create bindings, they should always do so over the narrowest possible scope. Here, if we dispensed with the

```
(defmacro alambda (parms &body body)
  `(labels ((self ,parms ,@body))
     #'self))

(defmacro ablock (tag &rest args)
  `(block ,tag
     ,(funcall (alambda (args)
                 (case (length args)
                   (0 nil)
                   (1 (car args))
                   (t `(let ((it ,(car args)))
                         ,(self (cdr args))))))
               args)))
```

Figure 14.2: More anaphoric variants.

gensym and instead bound `it` immediately to the result of the test expression, as in:

```
(defmacro acond (&rest clauses)                            ; wrong
  (if (null clauses)
      nil
      (let ((cl1 (car clauses)))
        `(let ((it ,(car cl1)))
           (if it
               (progn ,@(cdr cl1))
               (acond ,@(cdr clauses)))))))
```

then that binding of `it` would also have within its scope the *following* test expression.

Figure 14.2 contains some more complicated anaphoric variants. The macro `alambda` is for referring literally to recursive functions. When does one want to refer literally to a recursive function? We can refer literally to a function by using a sharp-quoted lambda-expression:

```
#'(lambda (x) (* x 2))
```

But as Chapter 2 explained, you can't express a recursive function with a simple lambda-expression. Instead you have to define a local function with `labels`. The following function (reproduced from page 22)

```
(defun count-instances (obj lists)
  (labels ((instances-in (list)
             (if list
                 (+ (if (eq (car list) obj) 1 0)
                    (instances-in (cdr list)))
                 0)))
    (mapcar #'instances-in lists)))
```

takes an object and a list, and returns a list of the number of occurrences of the object in each element:

```
> (count-instances 'a '((a b c) (d a r p a) (d a r) (a a)))
(1 2 1 2)
```

With anaphora we can make what amounts to a literal recursive function. The `alambda` macro uses `labels` to create one, and thus can be used to express, for example, the factorial function:

```
(alambda (x) (if (= x 0) 1 (* x (self (1- x)))))
```

Using `alambda` we can define an equivalent version of `count-instances` as follows:

```
(defun count-instances (obj lists)
  (mapcar (alambda (list)
            (if list
                (+ (if (eq (car list) obj) 1 0)
                   (self (cdr list)))
                0))
          lists))
```

Unlike the other macros in Figures 14.1 and 14.2, which all capture `it`, `alambda` captures `self`. An instance of `alambda` expands into a `labels` expression in which `self` is bound to the function being defined. As well as being smaller, `alambda` expressions look like familiar `lambda` expressions, making code which uses them easier to read.

The new macro is used in the definition of `ablock`, an anaphoric version of the built-in `block` special form. In a `block`, the arguments are evaluated left-to-right. The same happens in an `ablock`, but within each the variable `it` will be bound to the value of the previous expression.

This macro should be used with discretion. Though convenient at times, `ablock` would tend to beat what could be nice functional programs into imperative form. The following is, unfortunately, a characteristically ugly example:

```
> (ablock north-pole
    (princ "ho ")
    (princ it)
    (princ it)
    (return-from north-pole))
ho ho ho
NIL
```

Whenever a macro which does intentional variable capture is exported to another package, it is necessary also to export the symbol being captured. For example, wherever `aif` is exported, `it` should be as well. Otherwise the `it` which appears in the macro definition would be a different symbol from an `it` used in a macro call.

14.2 Failure

In Common Lisp the symbol `nil` has at least three different jobs. It is first of all the empty list, so that

```
> (cdr '(a))
NIL
```

As well as the empty list, `nil` is used to represent falsity, as in

```
> (= 1 0)
NIL
```

And finally, functions return `nil` to indicate failure. For example, the job of the built-in `find-if` is to return the first element of a list which satisfies some test. If no such element is found, `find-if` returns `nil`:

```
> (find-if #'oddp '(2 4 6))
NIL
```

Unfortunately, we can't tell this case from the one in which `find-if` succeeds, but succeeds in finding `nil`:

```
> (find-if #'null '(2 nil 6))
NIL
```

In practice, it doesn't cause too much trouble to use `nil` to represent both falsity and the empty list. In fact, it can be rather convenient. However, it is a pain to have `nil` represent failure as well, because it means that the result returned by a function like `find-if` can be ambiguous.

The problem of distinguishing between failure and a `nil` return value arises with any function which looks things up. Common Lisp offers no less than three solutions to the problem. The most common approach, before multiple return values, was to return gratuitous list structure. There is no trouble distinguishing failure with `assoc`, for example; when successful it returns the whole pair in question:

```
> (setq synonyms '((yes . t) (no . nil)))
((YES . T) (NO))
> (assoc 'no synonyms)
(NO)
```

Following this approach, if we were worried about ambiguity with `find-if`, we would use `member-if`, which instead of just returning the element satisfying the test, returns the whole cdr which begins with it:

```
> (member-if #'null '(2 nil 6))
(NIL 6)
```

Since the advent of multiple return values, there has been another solution to this problem: use one value for data and a second to indicate success or failure. The built-in `gethash` works this way. It always returns two values, the second indicating whether anything was found:

```
> (setf edible                      (make-hash-table)
        (gethash 'olive-oil edible) t
        (gethash 'motor-oil edible) nil)
NIL
> (gethash 'motor-oil edible)
NIL
T
```

So if you want to detect all three possible cases, you can use an idiom like the following:

```
(defun edible? (x)
  (multiple-value-bind (val found?) (gethash x edible)
    (if found?
        (if val 'yes 'no)
        'maybe)))
```

thereby distinguishing falsity from failure:

```
> (mapcar #'edible? '(motor-oil olive-oil iguana))
(NO YES MAYBE)
```

Common Lisp supports yet a third way of indicating failure: to have the access function take as an argument a special object, presumably a gensym, to be returned in case of failure. This approach is used with `get`, which takes an optional argument saying what to return if the specified property isn't found:

```
> (get 'life 'meaning (gensym))
#:G618
```

Where multiple return values are possible, the approach used by `gethash` is the cleanest. We don't want to have to pass additional arguments to every access function, as we do with `get`. And between the other two alternatives, using multiple values is the more general; `find-if` could be written to return two values, but `gethash` could not, without consing, be written to return disambiguating list structure. Thus in writing new functions for lookup, or for other tasks where failure is possible, it will usually be better to follow the model of `gethash`.

The idiom found in `edible?` is just the sort of bookkeeping which is well hidden by a macro. For access functions like `gethash` we will want a new version of `aif` which, instead of binding and testing the same value, binds the first but also tests the second. The new version of `aif`, called `aif2`, is shown in Figure 14.3. Using it we could write `edible?` as:

```
(defun edible? (x)
  (aif2 (gethash x edible)
        (if it 'yes 'no)
        'maybe))
```

Figure 14.3 also contains similarly altered versions of `awhen`, `awhile`, and `acond`. For an example of the use of `acond2`, see the definition of `match` on page 239. By using this macro we are able to express in the form of a `cond` a function that would otherwise be much longer and less symmetrical.

The built-in `read` indicates failure in the same way as `get`. It takes optional arguments saying whether or not to generate an error in case of eof, and if not, what value to return. Figure 14.4 contains an alternative version of `read` which uses a second return value to indicate failure: `read2` returns two values, the input expression and a flag which is `nil` upon eof. It calls `read` with a gensym to be returned in case of eof, but to save the trouble of building the gensym each time `read2` is called, the function is defined as a closure with a private copy of a gensym made at compile time.

Figure 14.4 also contains a convenient macro to iterate over the expressions in a file, written using `awhile2` and `read2`. Using `do-file` we could, for example, write a version of `load` as:

```
(defun our-load (filename)
  (do-file filename (eval it)))
```

```
(defmacro aif2 (test &optional then else)
  (let ((win (gensym)))
    `(multiple-value-bind (it ,win) ,test
       (if (or it ,win) ,then ,else))))

(defmacro awhen2 (test &body body)
  `(aif2 ,test
         (progn ,@body)))

(defmacro awhile2 (test &body body)
  (let ((flag (gensym)))
    `(let ((,flag t))
       (while ,flag
         (aif2 ,test
               (progn ,@body)
               (setq ,flag nil))))))

(defmacro acond2 (&rest clauses)
  (if (null clauses)
      nil
      (let ((cl1 (car clauses))
            (val (gensym))
            (win (gensym)))
        `(multiple-value-bind (,val ,win) ,(car cl1)
           (if (or ,val ,win)
               (let ((it ,val)) ,@(cdr cl1))
               (acond2 ,@(cdr clauses)))))))
```

Figure 14.3: Multiple-value anaphoric macros.

14.3 Referential Transparency

Anaphoric macros are sometimes said to violate referential transparency, which Gelernter and Jagannathan define as follows:

> A language is *referentially transparent* if (a) every subexpression can be replaced by any other that's equal to it in value and (b) all occurrences of an expression within a given context yield the same value.

○

Note that this standard applies to languages, not to programs. No language with assignment is referentially transparent. The first and the last `x` in this expression

```
(let ((g (gensym)))
  (defun read2 (&optional (str *standard-input*))
    (let ((val (read str nil g)))
      (unless (equal val g) (values val t)))))

(defmacro do-file (filename &body body)
  (let ((str (gensym)))
    `(with-open-file (,str ,filename)
       (awhile2 (read2 ,str)
         ,@body))))
```

Figure 14.4: File utilities.

```
(list x
      (setq x (not x))
      x)
```

yield different values, because a `setq` intervenes. Admittedly, this is ugly code. The fact that it is even possible means that Lisp is not referentially transparent.

Norvig mentions that it would be convenient to redefine `if` as: ○

```
(defmacro if (test then &optional else)
  `(let ((that ,test))
     (if that ,then ,else)))
```

but rejects this macro on the grounds that it violates referential transparency.

However, the problem here comes from redefining built-in operators, not from using anaphora. Clause (b) of the definition above requires that an expression always return the same value "within a given context." It is no problem if, within this `let` expression,

```
(let ((that 'which))
  ...)
```

the symbol `that` denotes a new variable, because `let` is advertised to create a new context.

The trouble with the macro above is that it redefines `if`, which is *not* supposed to create a new context. This problem goes away if we give anaphoric macros distinct names. (As of CLTL2, it is illegal to redefine `if` anyway.) As long as it is part of the definition of `aif` that it establishes a new context in which `it` is a new variable, such a macro does not violate referential transparency.

Now, `aif` does violate another convention, which has nothing to do with referential transparency: that newly established variables somehow be indicated in the source code. The `let` expression above clearly indicates that `that` will refer to a new variable. It could be argued that the binding of `it` within an `aif` is not so clear. However, this is not a very strong argument: `aif` only creates one variable, and the creation of that variable is the only reason to use it.

Common Lisp itself does not treat this convention as inviolable. The binding of the CLOS function `call-next-method` depends on the context in just the same way that the binding of the symbol `it` does within the body of an `aif`. (For a suggestion of how `call-next-method` would be implemented, see the macro `defmeth` on page 358.) In any case, such conventions are only supposed to be a means to an end: programs which are easy to read. And anaphora do make programs easier to read, just as they make English easier to read.