Let Over Lambda

Let Over Lambda -- 50 Years of Lisp

by Doug Hoyte

Anaphoric Macros

More Phors?

Some of the most interesting macros from Paul Graham's *On Lisp* are *anaphoric macros*. An anaphoric macro is one that deliberately captures a variable from forms supplied to the macro. Thanks to their *transparent specifications*, these deliberately captured variables allow us windows of control over the macro expansion. Through these windows we can manipulate the expansion through *combinations*.

The classic anaphora like those in *On Lisp* are named after the literal words[1] *A U-language quotation.* *anaphor* and its plural, *anaphora*. An anaphor is a means of capturing a *free U-language word* for use in subsequent U-language. In programming terms, implementing a classic anaphor means finding places in your code—or in code you would like to write—where expressions can benefit from being able to refer to the results of previous, related expressions. Graham's anaphora and associated code are worth careful study. Especially see the **defanaph** macro[ON-LISP-P223] which enables some interesting types of *automatic anaphor* programming.

After some period of use, **alambda** has been found to be the most useful of the anaphoric macros in *On Lisp*. It is also one of the most simple and elegant demonstrations of an anaphoric macro and its intentional variable capture.

ALAMBDA

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

With **alambda** we capture the name **self** so that we can use it to refer to the very anonymous function we are building. In other words, recursing is as simple as a call to **self**. For example, the following function returns a

list[2] *If the condition is false, an absent tertiary clause on an if form returns* ***nil****, which is a list.*
of the numbers from **n** down to 1:

```
(alambda (n)
  (if (> n 0)
    (cons
      n
      (self (- n 1)))))
```

Alambda lets our code be intuitive and easy to read, and allows us to change our mind about whether an anonymous function should be able to call itself as easily as adding a single letter[3] *This is another reason to not sharp-quote lambda forms. Changing a sharp quoted lambda form to an a*
. Because of **alambda**'s transparent specification for the **self** binding—and the fact that the only reason to ever use **alambda** is to make use of this binding—unwanted variable capture is never a problem.

AIF

```
;; Graham's aif
(defmacro aif (test then &optional else)
  `(let ((it ,test))
     (if it ,then ,else)))
```

Another handy anaphoric macro from *On Lisp* is **aif**, a macro that binds the result of the test clause to **it** for the true (secondary or consequent) clause to make use of[4] *Exercise: Why would the false (tertiary or alternant) clause never make use of this anaphor?*
. **Aif** makes use of a valuable COMMON LISP feature: *generalised booleans*. In COMMON LISP, any non-nil value is a true boolean value so COMMON LISP programmers typically embed interesting information in truth values. Languages that have reserved true and false values—notably Scheme—employ *explicit booleans*, which sometimes force you to throw out extra information to satisfy redundant type constraints. Scheme has actually added a *kludge* to allow **if**, **cond**, **and**, **or**, and **do** to accept non-boolean[5] *According to the Scheme* ***boolean?*** *predicate.*
values[R5RS-P25]. COMMON LISP, of course, is designed *right*—everything is a boolean.

It must also be pointed out that **aif** and **alambda**, like all anaphoric macros, violate *lexical transparency*. A fashionable way of saying this is currently to say that they are *unhygienic* macros. That is, like a good number of macros in this book, they invisibly introduce lexical bindings and thus cannot be created with macro systems that strictly enforce hygiene. Even the vast majority of Scheme systems, the platform that has experimented the most with hygiene, provide unhygienic defmacro-style macros—presumably because not even Scheme implementors take hygiene very seriously. Like training wheels on a bicycle, hygiene systems are for the most part toys that should be discarded after even a modest level of skill has been acquired.

Yes, there are many interesting things we can do with deliberate variable capture. There are a lot more phors. This book and Graham's *On Lisp* describe a only a

tiny fraction of the potential inherent to this technique. Many more incredible inventions will come out of the intelligent application of anaphoric macros.

After a brief interlude into anaphora introduced by read macros, the remainder of this chapter describes a modest, specific application of anaphora related to one of the central themes of this book: the lexical closure—*let over lambda*. Most of this chapter describes interesting anaphoric macros for customising, adapting, and extending closures. Although the topics are very practical for use in real code, their main purpose is as a platform for discussing the properties and variations of anaphoric macros. Using macros to extend the concept of a closure is currently a hot research topic[FIRST-CLASS-EXTENTS][OPENING-CLOSURES].

Sharp-Backquote

Although most anaphora are introduced by regular macros, read macros also have the potential to introduce code that invisibly creates bindings for us. When read macros do so, they are called *read anaphora*. This section presents one such read macro that, while itself very modest, surprised even myself by turning out to be one of the most consistently useful throughout this book. I have tried to introduce this macro as soon as possible so that it can be used for the remainder of the code. Already, several macros shown should have used it.

SHARP-BACKQUOTE

```
(defun |#`-reader| (stream sub-char numarg)
  (declare (ignore sub-char))
  (unless numarg (setq numarg 1))
  `(lambda ,(loop for i from 1 to numarg
                  collect (symb 'a i))
     ,(funcall
        (get-macro-character #\`) stream nil)))

(set-dispatch-macro-character
  #\# #\` #'|#`-reader|)
```

Sharp-backquote is a read macro that reads in as a lambda form. By default, this lambda form will take exactly one argument: **a1**. The read macro then recursively invokes the **read** function with the supplied stream. Here is an example with the evaluation stopped (by **quote**) so we can observe the transparent introduction of the read anaphor[6] *The prefix of the captured symbol, "a", of course stands for anaphor.* :

```
* '#`((,a1))

(LAMBDA (A1)
  `((,A1)))
```

This read macro abstracts out a common macro pattern. For example, if we have a list of variables and would like to make a list of let bindings that bind

each variable to a symbol, say, **empty**, we can use **mapcar** like so:

```
* (mapcar (lambda (a)
            (list a ''empty))
    '(var-a var-b var-c))

((VAR-A 'EMPTY)
 (VAR-B 'EMPTY)
 (VAR-C 'EMPTY))
```

But especially for complicated list structure, this can get messy, so lisp programmers like to use backquote to knock it up one level of quotation:

```
* (mapcar (lambda (a)
            `(,a 'empty))
    '(var-a var-b var-c))

((VAR-A 'EMPTY)
 (VAR-B 'EMPTY)
 (VAR-C 'EMPTY))
```

Our new anaphor-introducing read macro hides the lambda form:

```
* (mapcar #`(,a1 'empty)
    '(var-a var-b var-c))

((VAR-A 'EMPTY)
 (VAR-B 'EMPTY)
 (VAR-C 'EMPTY))
```

The reason for the **1** character in the symbol **a1** above is that users of the read macro can introduce a variable number of anaphora depending on the number provided to the **numarg** parameter of the read macro:

```
* '#2`(,a1 ,a2)

(LAMBDA (A1 A2)
  `(,A1 ,A2))
```

So we can **mapcar** sharp-backquote expressions across more than one list at a time:

```
* (let ((vars '(var-a var-b var-c)))
    (mapcar #2`(,a1 ',a2)
      vars
      (loop for v in vars
            collect (gensym
                      (symbol-name v)))))

((VAR-A '#:VAR-A1731)
```

```
 (VAR-B '#:VAR-B1732)
 (VAR-C '#:VAR-C1733))
```

Another way to think about sharp-backquote is that it is to list interpolation as the **format** function is to string interpolation. Just as format lets us use a template with slots that are to be filled with the values of separate arguments, sharp-backquote lets us separate the structure of the list interpolation from the values we want to splice in. Because of the earlier described *duality of syntax* between lambda forms in the function position of a list and the lambda forms that use the **lambda** macro to expand into a function, we can also use sharp-backquote as the first element in a function call:

```
* (#3`(((,a1)) ,@a2 (,a3))
      (gensym)
      '(a b c)
      'hello)

(((#:G1734)) A B C (HELLO))
```

Unlike **format**, sharp-backquote doesn't use sequential positioning. Instead it uses the number on our anaphoric bindings. As a consequence, the order can be mixed up and we can even splice in bindings more than once:

```
* (#3`(((,@a2)) ,a3 (,a1 ,a1))
      (gensym)
      '(a b c)
      'hello)

(((A B C)) HELLO (#:G1735 #:G1735))
```

Exercise: The references to the gensym **#:G1735** look like they point to the same symbol but, of course, you never really can tell with gensyms by looking at their print names. Are these symbols **eq**? Why or why not?

Alet and Finite State Machines

With *lambda* and *if* there is only one useful anaphoric configuration. But the most interesting types of anaphoric macros make use of expansions in unforeseen ways. This section—even most of this chapter—is based around one such macro: **alet**. What extra bindings could be useful to forms inside the body of a let form? The very purpose of **let** is to create such bindings so capturing the variable introductions given to a let form is already done. However, a macro enhancement to **let** can have complete access to all the forms given to it, even the body of expressions intended to be evaluated with the new bindings. So what is the most useful part of the body? In most cases it is the last form in the body since the results from that form will be returned from the let statement itself[7] *Because **let** provides an implicit progn.* . We've seen that when we return a lambda expression that references these bindings

created by **let**, the result is a lexical closure—an object frequently stored and used to later access the variables in the let statement. So, extending our closure-object analogy, the **alet%** macro acts exactly like the **let** special form except that it captures the symbol **this** from the body and binds it to the last expression in the form's body—the one that will be returned as the closure[8] .

[8] ***Setq*** *is used so that the form bound to* ***this*** *is defined in the lexical scope of the other arguments given by* ***letargs****.*

ALET-1

```
(defmacro alet% (letargs &rest body)
  `(let ((this) ,@letargs)
     (setq this ,@(last body))
     ,@(butlast body)
     this))
```

Alet% can be useful when we have initialisation code in the lambda form that we don't want to duplicate. Because **this** is bound to the lambda form that we are returning, we can execute it before we return it from the enclosing let. The following is a closure whose construction shows a simple example use of **alet%** that avoids duplicating its reset and initialisation code:

```
* (alet% ((sum) (mul) (expt))
    (funcall this :reset)
    (dlambda
      (:reset ()
        (psetq sum 0
               mul 1
               expt 2))
      (t (n)
        (psetq sum (+ sum n)
               mul (* mul n)
               expt (expt expt n))
        (list sum mul expt))))

#<Interpreted Function>
```

Which we can call successively to change the values of **sum**, **mul**, and **expt**:

```
* (loop for i from 1 to 5 collect (funcall * 2))

((2 2 4)
 (4 4 16)
 (6 8 256)
 (8 16 65536)
 (10 32 4294967296))
```

We can now reset the closure by invoking its **:reset** method. Notice that we only had to write the reset base cases (0 for **sum**, 1 for **mul**, and 2 for **expt**) in

one location thanks to **alet%**:

```
* (funcall ** :reset)

NIL
```

Now that the closure's variables are reset, we can see a new sequence from the start:

```
* (loop for i from 1 to 5 collect (funcall *** 0.5))

((0.5 0.5 1.4142135)
 (1.0 0.25 1.1892071)
 (1.5 0.125 1.0905077)
 (2.0 0.0625 1.0442737)
 (2.5 0.03125 1.0218971))
```

Note that **alet%** changes the evaluation order of the forms in the let body. If you look at the expansion you will notice that the last form in the body is actually evaluated first, and its results are then bound to the lexical binding **this** before the preceding forms are evaluated. As long as the last argument is a constant, however, this re-ordering makes no difference. Remember that a lambda expression[9] ***Dlambda*** *expands into lambda forms.* is a constant value and is thus perfectly suitable for use in **alet%**.

As with many macro enhancements, because of the many degrees of freedom available, improvements to this macro are counterintuitive. Although there are many possibilities, this section considers one such specific improvement. **Alet%** can be made to not return the last form in its body—which we anticipate to be a lambda form—but instead a function that looks up another function inside the let form's lexical scope, then calls that function instead. This is sometimes called *indirection* because instead of returning a function to do something, we return a function that looks up a function using a pointer dereference, then uses that function instead. Indirection is a concept ubiquitous throughout programming languages for good reason. It lets us change things at run-time that, without indirection, are fixed at compile-time. Lisp lets us use indirection in a more succinct and efficient manner than many other programming languages. **Alet**, a version of **alet%** with indirection added, allows the function we returned as the closure to now be accessed or replaced by code inside the alet body, or, if we use **dlambda**, as will be shown soon, even outside the alet body.

ALET

```
(defmacro alet (letargs &rest body)
  `(let ((this) ,@letargs)
     (setq this ,@(last body))
     ,@(butlast body)
     (lambda (&rest params)
       (apply this params))))
```

Now that we can change the function that gets executed when invoking a closure with our **alet** macro we can create a pair of mutually referential functions using a pattern called *alet over alambda*. As long as all the states go back to the original state—instead of going to each other—alet over alambda is a convenient way to specify nameless state machines.

The following is a typical counter closure that takes an argument **n** and can have its direction toggled between increment and decrement by **n** when we pass the symbol **invert** as the argument instead of a number:

```
* (alet ((acc 0))
    (alambda (n)
      (if (eq n 'invert)
        (setq this
              (lambda (n)
                (if (eq n 'invert)
                  (setq this #'self)
                  (decf acc n))))
        (incf acc n))))

#<Interpreted Function>
```

Let's store this closure so we can use it as often as we want:

```
* (setf (symbol-function 'alet-test) *)

#<Interpreted Function>
```

When we start, we are going up:

```
* (alet-test 10)

10
```

But we can change the actual function to be called to the internal lambda expression in our definition by passing the symbol **invert** to the closure:

```
* (alet-test 'invert)

#<Interpreted Function>
```

And now we're going down:

```
* (alet-test 3)

7
```

Finally, thanks to the **self** binding provided by **alambda**, we can again change the function to be called by passing the symbol **invert**:

```
* (alet-test 'invert)
```

```
#<Interpreted Function>
```

Back where we started, going up:

```
* (alet-test 5)

12
```

This closure has been bound in the function namespace to the symbol **alet-test**. But this closure is slightly different than a regular closure. While both this closure and regular closures are pointers to a single environment, one that can have any number of references to it, this closure uses indirection to change which piece of code gets run when it is invoked. Although any piece of code can be installed, only ones in the lexical scope of the **alet**, the one with the **this** anaphor available, can access its lexical bindings. But there is still nothing to prevent us from installing a new closure, with its own lexical bindings and perhaps with changed behaviour in the *indirection environment* installed by **alet**. Much of the remainder of this chapter is about useful things we can do with indirection environments created by **alet**.

A common macro technique is informally known as *turning a macro inside out*. When you turn a macro inside out you pick a typical form that uses a macro similar to the macro you would like to create, and expand it. You then use that expansion as a template for your desired macro. For example, we would like a more general method of creating closures with multiple states than the alet over alambda counter presented earlier. Here is the above inside out expansion of the invertible counter alambda use case:

```
* (macroexpand
   '(alambda (n)
      (if (eq n 'invert)
        (setq this
              (lambda (n)
                (if (eq n 'invert)
                  (setq this #'self)
                  (decf acc n))))
        (incf acc n))))

(LABELS ((SELF (N)
           (IF (EQ N 'INVERT)
             (SETQ THIS
                   (LAMBDA (N)
                     (IF (EQ N 'INVERT)
                       (SETQ THIS #'SELF)
                       (DECF ACC N))))
             (INCF ACC N))))
  #'SELF)
```

If we re-factor the above expansion slightly to take advantage of the fact that

labels allows us to create multiple function bindings[10] *Hence the plurality of* ***labels****.* , we arrive at the following:

```
(alet ((acc 0))
  (labels ((going-up (n)
             (if (eq n 'invert)
               (setq this #'going-down)
               (incf acc n)))
           (going-down (n)
             (if (eq n 'invert)
               (setq this #'going-up)
               (incf acc (- n)))))
    #'going-up))
```

From this, we notice that **alambda** can use the **labels** special form to make all of its bindings available to all of the bodies of its functions. What's more, we now have a fairly complete template for our eventual macro.

ALET-FSM

```
(defmacro alet-fsm (&rest states)
  `(macrolet ((state (s)
                `(setq this #',s)))
     (labels (,@states) #',(caar states))))
```

Alet-fsm gives us a convenient syntax for expressing multiple possible *states* for our closure to exist in. It is a very light sugar coating of macros over top of **labels**, combined with a *code-walking* **macrolet** transformation that allows us to pretend as though we have a function, **state**, that can change the closure's current state, accessed through the **this** anaphor provided by **alet**. As an example, here is a cleaner version of our invertible counter:

```
(alet ((acc 0))
  (alet-fsm
    (going-up (n)
      (if (eq n 'invert)
        (state going-down)
        (incf acc n)))
    (going-down (n)
      (if (eq n 'invert)
        (state going-up)
        (decf acc n)))))
```

Alet-fsm is an instance of a technique we haven't seen before: *anaphor injection*. The use of this anaphor violates lexical transparency in so many ways that it is actually, somehow, *lexically invisible*. Not only does **alet** bind **this** for us invisibly, but the use of **this** by our **alet-fsm** macro is similarly invisible. **Alet-fsm** injects a free variable into our lexical context without us being able to see it at all in the lexical context.

The stylistic issues of this are uncertain[11] *As are, by nature, all stylistic issues. Once something is perfectly understood, style l*
, but macro programming is, of course, not about style. It is about power. Sometimes free variable injection can create a symbiosis between two macros—one that can better programmatically construct expansions than can two isolated expansions. Because this type of macro programming is sophisticated, parallels can again be drawn to the C pointer analogy. Just as learning C pointers breeds dubious stylistic advice, so does free variable injection.

The most plausible hypothesis for the source of difficulty in understanding free variable injection is its *fail-safe* behaviour[12] *Safe in the sense that, contrary to the real world, failing as quickly and as loudly*
. With an anaphor, if the supplied user code doesn't make use of the binding the code will probably continue to function, whether you intended it to or not. It has, possibly, failed silently and thus un-safely. However, when you inject a free variable and there is no environment there to capture it, your entire expression has become free. When this happens, you need to decide what to do before you can evaluate that expression. It has failed safe.

Style aside, free variable injection is sometimes just what we need when we want two related macros to communicate back and forth. Injection is really the same operation as that performed by anaphora, just in the opposite direction. Because you are opening up a new channel of communication between your macros, the complexity issues scale even more quickly. Consider sitting in a house full of fragile glass. You can safely throw objects to people outside the house, even if they don't bother catching them, but you had better make sure you catch any objects thrown back at you.

Indirection Chains

There are many ways we can take advantage of the **this** anaphor provided by **alet**. Since the environment is accessed through a dummy closure that forwards all invocations to the real closure pointed to by **this**, we can pass the dummy closure reference around, copying it as often as needed. *Indirection* like this is useful because we can change what happens when this dummy closure is invoked without having to change references to the dummy closure.

ICHAIN-BEFORE

```
(defmacro! ichain-before (&rest body)
  `(let ((,g!indir-env this))
     (setq this
       (lambda (&rest ,g!temp-args)
         ,@body
         (apply ,g!indir-env
                ,g!temp-args)))))
```

Ichain-before is intended to be expanded in an **alet** form. It adds a new body of code to be executed before invoking the main closure. Going back to our counter example, **ichain-before** lets us add a new closure that prints out the

previous value of the closed-over **acc** variable before it goes ahead and increments it:

```
* (alet ((acc 0))
    (ichain-before
      (format t "Changing from ~a~%" acc))
    (lambda (n)
      (incf acc n)))

#<Interpreted Function>
```

Which works as expected:

```
* (funcall * 2)
Changing from 0
2
* (funcall ** 2)
Changing from 2
4
```

There is a reason we put chain in the name of **ichain-before**, though. We can put as many of these closures on to be executed as we please:

```
* (alet ((acc 0))
    (ichain-before
      (format t "A~%"))
    (ichain-before
      (format t "B~%"))
    (ichain-before
      (format t "C~%"))
    (lambda (n)
      (incf acc n)))

#<Interpreted Function>
```

Each addition of a new link in the chain adds the link to the very beginning of the chain, resulting in the links being visited in the reverse order from which they were added:

```
* (funcall * 2)
C
B
A
2
```

Statically adding indirection chains is sometimes useful when changing macros to avoid re-structuring macros by adding new surrounding code. But the most interesting possibilities for indirection chains pop up when we add them dynamically. Because we can create new closures at run-time and because we can access the internals of a closure through an anaphor, we can re-write how

functions work at run-time. Here is a simple example in which every invocation of the closure adds another bit of code that prints "Hello world" when run:

```
* (alet ((acc 0))
    (lambda (n)
      (ichain-before
        (format t "Hello world~%"))
      (incf acc n)))

#<Interpreted Function>
```

Every invocation adds a new closure to the indirection chain:

```
* (loop for i from 1 to 4
    do
      (format t "~:r invocation:~%" i)
      (funcall * i))
first invocation:
second invocation:
Hello world
third invocation:
Hello world
Hello world
fourth invocation:
Hello world
Hello world
Hello world
```

The **ichain-after** macro is similar to the **ichain-before** macro except it adds the closures to the other end of the execution chain: after the main closure has been invoked. **Ichain-after** uses **prog1**, which executes its provided forms consecutively and then returns the result of evaluating the first form.

ICHAIN-AFTER

```
(defmacro! ichain-after (&rest body)
  `(let ((,g!indir-env this))
     (setq this
       (lambda (&rest ,g!temp-args)
         (prog1
           (apply ,g!indir-env
                  ,g!temp-args)
           ,@body)))))
```

Ichain-before and **ichain-after** can be combined so that the before forms are executed before the evaluation of the main closure and the after forms after:

```
* (alet ((acc 0))
    (ichain-before
      (format t "Changing from ~a~%" acc))
```

```
    (ichain-after
      (format t "Changed to ~a~%" acc))
    (lambda (n)
      (incf acc n)))

#<Interpreted Function>
* (funcall * 7)
Changing from 0
Changed to 7
7
```

Ichain-before and **ichain-after** are macros that inject free variables into their expansion. They inject the symbol **this** which we rely on being captured by the expansion of an **alet** macro. This sort of injection of symbols might seem to be bad style or error-prone, but it is actually a common macro technique. In fact, almost all macros inject symbols into the expansion. For instance, along with **this**, the macro **ichain-before** also injects symbols like **let**, **setq**, and **lambda** to be spliced into wherever the macro is expanded. The difference between symbols like **this** and pre-defined symbols like **setq** is that while **lambda** always refers to a single well-understood ANSI macro, symbols like **this** can refer to different things depending on the environments in which they are expanded.

Ichain-before and **ichain-after** are useful for tagging on code for a closure to run before or after the execution of the original closed-over expression but are by no means the only things we can do with the **this** anaphor. Another common task is checking for the validity of closure data after an invocation of the closure.

ICHAIN-INTERCEPT-1

```
(defmacro! ichain-intercept% (&rest body)
  `(let ((,g!indir-env this))
     (setq this
       (lambda (&rest ,g!temp-args)
         (block intercept
           (prog1
             (apply ,g!indir-env
                    ,g!temp-args)
             ,@body))))))
```

Ichain-intercept% is another macro designed to be used inside an **alet** form. The idea is that we would like to be able to intercept invocations of the closure and verify that the actions they performed didn't cause some sort of inconsistent state in the closure.

So we can add an intercept to our usual counter closure like so:

```
* (alet ((acc 0))
    (ichain-intercept%
      (when (< acc 0)
```

```
        (format t "Acc went negative~%")
        (setq acc 0)
        (return-from intercept acc)))
    (lambda (n)
      (incf acc n)))

#<Interpreted Function>
```

When the counter falls below 0, the code installed by **ichain-intercept%** will warn us:

```
* (funcall * -8)
Acc went negative
0
```

The counter was reset back to 0:

```
* (funcall ** 3)

3
```

The most interesting thing about **ichain-intercept%** is that it introduces a *block anaphor* named **intercept**. To use this anaphor we use **return-from**. The block will return this value from the closure invocation, intercepting the original value.

ICHAIN-INTERCEPT

```
(defmacro! ichain-intercept (&rest body)
  `(let ((,g!indir-env this))
     (setq this
       (lambda (&rest ,g!temp-args)
         (block ,g!intercept
           (macrolet ((intercept (v)
                        `(return-from
                           ,',g!intercept
                           ,v)))
             (prog1
               (apply ,g!indir-env
                      ,g!temp-args)
               ,@body)))))))
```

Instead of capturing the block anaphor **intercept**, **ichain-intercept** creates a local macro that allows the code inside **ichain-intercept** to use **intercept** to expand into a **return-from** where the block is specified by a gensym.

```
* (alet ((acc 0))
    (ichain-intercept
      (when (< acc 0)
        (format t "Acc went negative~%")
```

```
        (setq acc 0)
        (intercept acc)))
    (lambda (n)
      (incf acc n)))

#<Interpreted Function>
```

This works the same as with **ichain-intercept%**:

```
* (funcall * -8)
Acc went negative
0
* (funcall ** 3)

3
```

Of course, introducing all these closures transparently into operations can affect run-time performance. Luckily, modern lisp compilers are very good at optimising closures. If your application can stand a few pointer dereferences—and often it can—indirection chains might just be the best way to structure it. See section 7.4, Pointer Scope for another interesting way to think about indirection chains. Also see CLOS's before, after, and around functionalities.

Hotpatching Closures

There are three purposes for this important section. First, another interesting use of the **this** anaphor from **alet** is described. Second, the pattern *alet over dlambda* is discussed. Finally, a useful macro technique called *anaphor closing* is introduced.

ALET-HOTPATCH-1

```
(defmacro alet-hotpatch% (letargs &rest body)
  `(let ((this) ,@letargs)
     (setq this ,@(last body))
     ,@(butlast body)
     (lambda (&rest args)
       (if (eq (car args) ':hotpatch)
         (setq this (cadr args))
         (apply this args)))))
```

In order to clearly illustrate anaphor closing, we will not work with the **alet** macro but instead an inside out expansion. **Alet-hotpatch%** is an expansion of **alet** with a special lambda form provided. This lambda form checks the first argument[13] *With a pointer comparison.* to see if it is the keyword symbol **:hotpatch** and, if so, replaces the indirected closure with another provided argument.

Being able to change the closure used in another forwarding closure at run-time is known as *hotpatching*. For instance, here we create a hotpatchable closure and

store it in the symbol-function cell of the symbol **hotpatch-test** for later use:

```
* (setf (symbol-function 'hotpatch-test)
    (alet-hotpatch% ((acc 0))
      (lambda (n)
        (incf acc n))))

#<Interpreted Function>
```

It can now be used like so:

```
* (hotpatch-test 3)

3
* (hotpatch-test 4)

7
```

We can replace the lambda form—along with its associated environment—by calling this closure with the symbol **:hotpatch** and a replacement function or closure:

```
* (hotpatch-test
    :hotpatch
    (let ((acc 0))
      (lambda (n)
        (incf acc (* 2 n)))))

#<Interpreted Function>
```

Now the closure will have the new, hotpatched behaviour:

```
* (hotpatch-test 2)

4
* (hotpatch-test 5)

14
```

Notice how the counter value reset to 0 since we also hotpatched the closure's environment with a new value for the counter's accumulator, **acc**.

ALET-HOTPATCH

```
(defmacro alet-hotpatch (letargs &rest body)
  `(let ((this) ,@letargs)
     (setq this ,@(last body))
     ,@(butlast body)
     (dlambda
       (:hotpatch (closure)
         (setq this closure))
```

```
      (t (&rest args)
        (apply this args)))))
```

Haven't we seen this sort of *run-time destructuring* on keyword symbols before? Yes, in fact we wrote a macro for doing exactly this in section 5.7, Dlambda. **Alet-hotpatch** is a version of **alet-hotpatch%** that takes advantage of **dlambda**. Sometimes without even realising it, by using macros we wrote previously in the definition of new macros, we are performing *macro combination*. With well designed macros the expansion can be fully understood and, although it might violate lexical transparency in many ways, no combination problems emerge because all components fit together meaningfully.

Alet-hotpatch creates a hotpatchable closure but there is one slight conceptual flaw. Because the only real reason for using **alet-hotpatch** is to create this sort of hotpatchable closure, we might forget that it also introduces the anaphor **this** into the scope of the forms provided. When we forget about anaphora we've created, we risk unwanted variable capture problems. To avoid these problems, we might choose to employ a technique known as *anaphor closing*. When we close an anaphor, we don't need to change the way our anaphoric macros function, just restrict them in the ways they can be combined.

Because we have turned the **alet** expansion inside out, we can lexically see the creation of the **this** anaphor in the definition of **alet-hotpatch**. And because **alet-hotpatch** also contains the code using the **this** anaphor to implement hotpatching, we can close the anaphor so that the symbol **this** is no longer captured by the macro. How do we normally avoid introducing unwanted bindings? We name the bindings using gensyms of course.

LET-HOTPATCH

```
(defmacro! let-hotpatch (letargs &rest body)
  `(let ((,g!this) ,@letargs)
     (setq ,g!this ,@(last body))
     ,@(butlast body)
     (dlambda
       (:hotpatch (closure)
         (setq ,g!this closure))
       (t (&rest args)
         (apply ,g!this args)))))
```

Let-hotpatch is an example of closing the **this** anaphor into a more contained version—a safer version for when hotpatching is all that is required. The leading **a** was removed from the name to suggest that this new macro no longer introduces an anaphor into the supplied body of code. Of course if we wanted to refer to **this** for some reason other than hotpatching, we should have left the anaphor open.

This technique of opening and closing anaphora becomes second nature after you have written enough of such macros. Just like we can write macros that inject

free variables into an expansion without thinking about how we will capture them until we write the lexical context in which they will be expanded, we sometimes choose to leave an anaphor open while developing macros to experiment with combinations of anaphoric macros and free variable injection macros. Once the most useful combinations are found, we can merge the macros together and replace all anaphora used during development with with gensyms. Like **let-hotpatch** does, this technique can use **defmacro!** to move the anaphor's scope from the macro expansion to the macro definition. Instead of lexically introducing an anaphor, we introduced another type of anaphor—one that doesn't take effect in the full lexical scope of the expansion but only in another, more limited scope. This scope is described further in the following section.

Sub-Lexical Scope

Our **defmacro!** macro-defining macros that we defined in section 3.5, Unwanted Capture look for the presence of automatic gensyms in the provided code using Graham's **flatten** utility. Now is the time to confess a small lie we have been telling through this book. Before now, because we hadn't explained free variable injection and anaphora, we pretended that the G-bang symbol names in **defmacro!** definitions are applicable in the lexical scope of the macro definition. This is actually not true—**defmacro!** provides these bindings under a slightly different type of scope called *sub-lexical scope*.

Remember that scope means where references to a variable are valid and lexical scope means that the name is applicable to code in the textual body of a binding construct such as **let**. The important distinction between lexical scope and sub-lexical scope is that for lexical scope this includes all macroexpansions of code in the **let** body. So describing lexical scope as creating variables only accessible to code in the textual body of a binding construct is actually a lie too—macros can *inject* variable references. Such variables are injected from outside the textual body of the binding construct.

Implementing genuine textual scoping by limiting the possible ways to access lexical variables results in sub-lexical scope. References to a sub-lexically scoped variable are only valid if the symbols representing them occur in the raw lists that were passed to lisp before macro-expansion.

Because **defmacro!** pre-processes the code it was given and creates the list of all the G-bang symbols before the code gets expanded, the G-bang symbols are sub-lexically bound. We can't write macros that inject G-bang symbols into **defmacro!** because lexical bindings for the G-bang symbols were never created. Here is a typical use of a sub-lexical G-bang symbol:

```
* (defmacro! junk ()
    `(let ((,g!var))
       ,g!var))

JUNK
```

Both G-bang symbols were found in the sub-lexical scope of **defmacro!** so expansion is as we would expect:

```
* (macroexpand '(junk))

(LET ()
  (LET ((#:VAR1663))
    #:VAR1663))
T
```

However, in order to explore the concept of sub-lexical scope, we will define a macro that injects a G-bang symbol:

```
* (defmacro injector-for-g!var ()
    ''g!var)

INJECTOR-FOR-G!VAR
```

Now we can write **junk2**. **Junk2** is identical to **junk** except that we have replaced our G-bang symbols with a macro that expands into a G-bang symbol:

```
* (defmacro! junk2 ()
    `(let ((,(injector-for-g!var)))
       ,(injector-for-g!var)))

JUNK2
```

But since the G-bang symbols are sub-lexically bound—and thus don't look into the macro expansions of forms—**defmacro!** doesn't convert the symbols into automatic gensyms:

```
* (macroexpand '(junk2))

(LET ()
  (LET ((G!VAR))
    G!VAR))
T
```

Although the above code will still function, sub-lexically scoped variable references can break expressions when some references that refer to a variable exist in sub-lexical scope and others don't:

```
* (defmacro! junk3 ()
   `(let ((,g!var))
      ,(injector-for-g!var)))

JUNK3
* (macroexpand '(junk3))

(LET ()
```

```
  (LET ((#:VAR1672))
    G!VAR))
T
```

Sub-lexical scoping turns up surprisingly often in complex macros. As well as **defmacro!**, we've seen it in at least one other example: the **with-all-cxrs** macro from section 5.6, Recursive Solutions sub-lexically binds list accessor functions. The consequence of sub-lexical binding is that we can't refer to such bindings from macro expansions. Sometimes this access limitation is useful, sometimes not. In **with-all-cxrs**, sub-lexicality could be considered undesirable. When our accessor is in **with-all-cxrs**'s sub-lexical scope, there is no problem:

```
* (with-all-cxrs
    (cadadadr nil))

NIL
```

And we can even write macros that expand into these accessors, as long as the macro definitions are in the sub-lexical scope of **with-all-cxrs**:

```
* (with-all-cxrs
    (macrolet ((accessor (l)
                 `(cadadadr ,l)))
      (accessor nil)))

NIL
```

But notice that **with-all-cxrs** binds the accessor function sub-lexically so we can't define a macro to inject the accessor:

```
* (macrolet ((accessor (l)
               `(cadadadr ,l)))
    (with-all-cxrs
      (accessor nil)))

This function is undefined: CADADADR
```

Now that we are familiar with *anaphora* and have seen numerous examples of complex macros—including some that utilise sub-lexical scope—we can discuss an interesting theoretical macro: **sublet**. This macro is designed to create sub-lexical bindings for code using a syntax similar to the usual let form syntax. The discussion of **sublet**, as with many lisp macros, begins with a utility.

LET-BINDING-TRANSFORM

```
(defun let-binding-transform (bs)
  (if bs
    (cons
      (cond ((symbolp (car bs))
             (list (car bs)))
```

```
            ((consp (car bs))
              (car bs))
            (t
              (error "Bad let bindings")))
      (let-binding-transform (cdr bs)))))
```

Let-binding-transform is a simple utility that handles the case of a let form binding being a single symbol. In the following, **a** is normalised to **(a)**:

```
* (let-binding-transform
    '(a (b) (c nil)))

((A) (B) (C NIL))
```

Sublet also uses the **tree-leaves** utility we defined in section 5.3, Implicit Contexts. Recall that the **tree-leaves** macro takes three arguments: an arbitrary list structure, an expression that can make use of an **x** variable to determine whether a leaf should be changed, and another expression that can make use of a different **x** to determine what valid leaves should be changed to.

The choice to implicitise the bindings of **x** with the same name turns out to be a useful *duality of syntax*. When we can't factor common code in an expression the usual way, sometimes we can gain this brevity advantage by using syntactic duals in other ways. The definition of **sublet** uses the self-referential read macros described in section 4.5, Cyclic Expressions. Especially for things like accessors that can change many times throughout the writing of a program, read macros allow us to have one and only one form representing the accessor. Thanks to our use of *implicitisation* with the **tree-leaves** macro, it is easy to find and understand the code duplication because the code is close together.

SUBLET

```
(defmacro sublet (bindings% &rest body)
  (let ((bindings (let-binding-transform
                    bindings%)))
    (setq bindings
      (mapcar
        (lambda (x)
          (cons (gensym (symbol-name (car x))) x))
        bindings))
    `(let (,@(mapcar #'list
                     (mapcar #'car bindings)
                     (mapcar #'caddr bindings)))
       ,@(tree-leaves
           body
           #1=(member x bindings :key #'cadr)
           (caar #1#)))))
```

Sublet takes the form representing the let bindings and applies our **let-**

binding-transform utility, generating new list structure in the process. It then prepends[14] a gensym to each binding with a print name corresponding to the binding name. **Sublet** expands into a let form which binds these gensym symbols to the values passed into the binding form, then uses **tree-leaves** to replace all occurrences of the binding name symbols in the provided code with their corresponding gensyms. **Sublet** does not expand any macros or parse any special forms in the body to look for occurrences of these binding name symbols because **sublet** creates sub-lexical bindings. For example, if all references to **a** are sub-lexical, it will replace them with gensyms:

[14] *Prepends instead of appends so we can still support bindings without default values, such as* ***(a)****.*

```
* (macroexpand
   '(sublet ((a 0))
      (list a)))

(LET ((#:A1657 0))
  (LIST #:A1657))
T
```

However, because sub-lexical scope doesn't involve expanding macros, and thus necessarily doesn't involve interpreting special forms like **quote**, instances of the symbol **a** that aren't supposed to be variable references are also changed:

```
* (macroexpand
   '(sublet ((a 0))
      (list 'a)))

(LET ((#:A1658 0))
  (LIST '#:A1658))
T
```

Sub-lexical scope takes effect before list structure is interpreted as lisp code by your system's code-walker. That is an important observation—one with ramifications still not completely explored. **Sublet** interprets your code differently than does the code-walker provided with COMMON LISP.

Here we are standing on one of the many edges of macro understanding. What sort of interesting types of scoping lie between unexpanded sub-lexical scope and fully expanded lexical scope? For lack of a better name, we will call this infinitely large category of scopes *super sub-lexical scopes*[15] .

[15] *I am giving it this silly name because I expect better names to become obvious when the concept is better understood.*

SUBLET*

```
(defmacro sublet* (bindings &rest body)
  `(sublet ,bindings
     ,@(mapcar #'macroexpand-1 body)))
```

A fairly obvious super sub-lexical scope uses **sublet***. This macro uses **sublet**

underneath but changes each form in the body by macro expanding them with the **macroexpand-1** function. Now, instead of appearing in the raw list structure, references to symbols must occur after the first step of macro expansion. This type of super sub-lexical scope allows a macro in each of the let form's body to inject or remove references from the scope. If the macros don't do either of these things—or if the forms aren't macros at all—this type of super sub-lexical scope acts just like sub-lexical scope:

```
* (macroexpand
   '(sublet* ((a 0))
      (list a)))

(LET ((#:A1659 0))
  (LIST #:A1659))
T
```

But we can define another injector macro to test this super sub-lexical scope:

```
* (defmacro injector-for-a ()
    'a)

INJECTOR-FOR-A
```

Sublet* will expand this injector macro:

```
* (macroexpand-1
   '(sublet* ((a 0))
      (injector-for-a)))

(SUBLET ((A 0))
  A)
T
```

Which will then be interpreted sub-lexically by **sublet**, meaning that the injected variable **a** exists within the type of super sub-lexical scope provided by **sublet***:

```
* (macroexpand-1 *)

(LET ((#:A1663 0))
  #:A1663)
```

But nested macros in the expression are not expanded by **macroexpand-1** so **sublet*** doesn't put them into sub-lexical scope for **sublet** to see:

```
* (macroexpand-1
   '(sublet* ((a 0))
      (list (injector-for-a))))

(SUBLET ((A 0))
  (LIST (INJECTOR-FOR-A)))
```

```
T
```

So **a** is not captured sub-lexically[16] ***Walker:macroexpand-all*** *is a CMUCL component of a complete code-walker.* :

```
* (walker:macroexpand-all *)

(LET ((#:A1666 0))
  (LIST A))
```

With **sublet** and **sublet*** we can control at what level of macro expansion the **a** variable is considered valid by using sub-lexical or super sub-lexical scopes. As mentioned above, super sub-lexical scope is actually an infinite class of scopes, one that is almost completely unexplored intellectually. As many ways as there are to walk code (a lot) there are super sub-lexical scopes. This class of scoping leads into another category of mostly unexplored macros: macros that change how lisp macros work, when they are expanded, where references are valid, how special forms are interpreted, etc. Eventually, a macro-programmable macro expander.

Pandoric Macros

Pandora's box is a Greek myth about the world's first woman: Pandora. Pandora, the U-language symbol, translates from Greek into all-gifted. Pandora, the woman, was tempted by curiosity to open a small box which irreparably unleashed all of humanity's evil and sins upon the world. While the macros described in this section are very powerful and might teach you a way of programming you never forget, rest assured that our outcome will be far better than poor Pandora's. Open the box.

First we take a slight detour through another famous lisp book: *Lisp in Small Pieces*[SMALL-PIECES] by Christian Queinnec. Queinnec is a widely respected lisp expert and has contributed much to our lisp knowledge. Queinnec's book is about implementing compilers and interpreters of varying sophistication in and for the Scheme programming language[17] *Though it sometimes describes other lisps and their features.* .

In *Lisp in Small Pieces* there is a short but interesting discussion on macros. Much of it relates to describing the different macro system variations possible thanks to the ambiguity of the Scheme macro specification[18] *Thanks, but no thanks.* , but there are also a few interesting notes on why we might want to use macros and how to go about using them. If you have read and understood chapter 3, Macro Basics, most of the macros presented in the chapter of *Lisp in Small Pieces* about macros will, to you, belong in the trivial category, save one enticing macro that we will now discuss.

Like many programming books, *Lisp in Small Pieces* takes us to and leaves us at an implementation of a system for *object-oriented* programming. Usually these implementations serve to outline a subset of *CLOS*, the COMMON LISP

Object System. Queinnec calls his subset *MEROONET*. Queinnec remarks that when defining a method for a MEROONET class it would be nice to be able to directly refer to the fields of the object being defined instead of using accessors. In Queinnec's (translated) words[SMALL-PIECES-P340-341]:

Let's take the case, for example, of the macro ***with-slots*** *from CLOS; we'll adapt it to a MEROONET context. The fields of an object—let's say the fields of an instance of* ***Point****—are handled by read and write functions like* ***Point-x*** *or* ***set-Point-y!****. It would be simpler to handle them directly by the name of their fields,* ***x*** *or* ***y****, for example, in the context of defining a method.*

Here is Queinnec's desired interface (which he has called **define-handy-method**) defining a new method, **double**:

```
(define-handy-method (double (o Point))
  (set! x (* 2 x))
  (set! y (* 2 y))
  o )
```

Which is more pleasing for programmers than the otherwise necessary MEROONET syntax:

```
(define-method (double (o Point))
  (set-Point-x! o (* 2 (Point-x o)))
  (set-Point-y! o (* 2 (Point-y o)))
  o )
```

In other words, it would be nice if we could use macros to access foreign bindings—in this case object slots—as if they were lexical bindings. Although this is undeniably useful for abbreviation purposes, its most important implication is its ability to give *dualities of syntax* to our existing and future macros.

As Queinnec notes, COMMON LISP implements this functionality for CLOS with a macro called **with-slots**. This is an example of COMMON LISP doing what it was designed to do: allowing abstractions based on a refined, standardised macro system. While most languages are designed to be easy to implement, COMMON LISP is designed to be powerful to program. Queinnec's conclusion was that language limitations make this mostly impossible in Scheme, especially where portability is required:

[L]acking reflective information about the language and its implementations, we cannot write a portable code-walker in Scheme, so we have to give up writing ***define-handy-method****.*

Although COMMON LISP still allows a large number of legal ways to implement macro systems, it is designed to provide general meta-programming tools that come together in standard and portable ways. The two advanced COMMON LISP macro features that allow us to implement things like CLOS's **with-slots** are *generalised variables* and *symbol macros*. This section takes advantage of an opportunity to show off this wonderful confluence of COMMON LISP features

as well as to bring together everything we have seen so far regarding anaphoric macros, in the process discovering an interesting class of macros called *pandoric macros*.

PANDORICLET

```
(defmacro pandoriclet (letargs &rest body)
  (let ((letargs (cons
                  '(this)
                  (let-binding-transform
                    letargs))))
    `(let (,@letargs)
       (setq this ,@(last body))
       ,@(butlast body)
       (dlambda
         (:pandoric-get (sym)
           ,(pandoriclet-get letargs))
         (:pandoric-set (sym val)
           ,(pandoriclet-set letargs))
         (t (&rest args)
           (apply this args))))))
```

The idea behind **pandoriclet** is to *open closures*, allowing their otherwise closed-over lexical variables to be accessed externally. As with some of our previous macros like **alet-hotpatch**, **pandoriclet** compiles an indirection environment that chooses different run-time behaviours depending on the arguments passed.

We again started with an *inside out* expansion of **alet**, keeping in mind the introduction of an anaphor called **this**. **Pandoriclet** is similar to other macros we've seen. As with all of our anaphoric **let** variants, we assume the final form in the **pandoriclet** body will be a lambda form. Like **alet-hotpatch**, **pandoriclet** uses the **dlambda** macro to dispatch between different possible pieces of code to execute when the closure returned from **pandoriclet** is invoked. **Pandoriclet** uses the **let-binding-transform** utility function introduced in the previous section to deal with null bindings created—like **(let (a) ...)**. This utility function is necessary to **pandoriclet** for the same reason that it was necessary for **sublet**: these macros code-walk the bindings provided to **let** where our previous macros blindly spliced the bindings into another **let**.

We have put in two calls to list-creating utility functions yet to be defined: **pandoriclet-get** and **pandoriclet-set**, each of which accept a list of let bindings. Notice we can reference functions that don't yet exist as long as we define them before the macro is expanded which obviously can't occur before we use the macro. Using auxiliary functions to help with defining macros is a good habit to pick up. Not only can it make your definitions more readable, but also can help when testing components of the macro and can prove useful in future macros. The best part about this sort of abstraction is that, as when combining macros, we keep our lexical context available for utilities to make use of.

PANDORICLET-ACCESSORS

```
(defun pandoriclet-get (letargs)
  `(case sym
     ,@(mapcar #`((,(car a1)) ,(car a1))
               letargs)
     (t (error
          "Unknown pandoric get: ~a"
          sym))))

(defun pandoriclet-set (letargs)
  `(case sym
     ,@(mapcar #`((,(car a1))
                  (setq ,(car a1) val))
               letargs)
     (t (error
          "Unknown pandoric set: ~a"
          sym val))))
```

So, remembering the lexical context, we write **pandoriclet-get** and **pandoriclet-set**. For **pandoriclet-get**, we remember that **dlambda** has bound a variable **sym** around where our list will be spliced in. We use **sym** in a **case** form that compares it to the symbols that were passed to **pandoriclet**[19] *Recall that* ***case*** *with symbols compiles to a single pointer comparison per case.* . If we find a symbol, the current value of the binding it refers to is returned. If not, an error is thrown. **Pandoriclet-set** is nearly identical, except that **dlambda** bound one extra symbol for it to use: **val**. **Pandoriclet-set** uses **setq** to change the binding referred to by **sym** to **val**.

Pandoriclet provides the same interface as all our anaphoric let variants so we can use it to make our usual counter closure:

```
* (setf (symbol-function 'pantest)
    (pandoriclet ((acc 0))
      (lambda (n) (incf acc n))))

#<Interpreted Function>
```

Which works as expected:

```
* (pantest 3)

3
* (pantest 5)

8
```

However, now we have direct access to the binding that was called **acc** when the closure was created:

```
* (pantest :pandoric-get 'acc)

8
```

And we can similarly change the value of this binding:

```
* (pantest :pandoric-set 'acc 100)

100
* (pantest 3)

103
```

Even the value of our **this** anaphor is accessible, since we deliberately left the anaphor open and added the symbol **this** to the **letargs** binding list when the macro was expanded:

```
* (pantest :pandoric-get 'this)

#<Interpreted Function>
```

So this closure we've created with **pandoriclet** is actually no longer closed. The environment used by this closure—even when all lexical variable symbols have been removed by the compiler—is still accessible through our anonymous function returned from **pandoriclet**. How does this work? With pandoric macros, additional code is compiled in to provide a way to access the closure from outside. But the power of pandoric macros can't be seen by looking at this low-level view of what is happening. What we have done is created an *inter-closure protocol*, or message passing system, for communicating between closures.

Before we continue with pandoric macros, first we need to point out one of the most important examples of *duality of syntax* in COMMON LISP: *generalised variables*. The details of this are complicated, and I won't describe all of them here. For that I recommend Graham's *On Lisp* which has the best treatment I am aware of. The finer points are subtle the idea is simple: accessing a generalised variable is syntactically dual to setting it. You have only one setter form, **setf**, which is capable of setting all types of variables by using the same syntax you would use to access them.

For example, with a regular variable you usually access its value through its symbol, say, **x**. To set it, you can use **(setf x 5)**. Similarly, to access the car slot of a cons called, say, **x**, you use **(car x)**. To set it, you can use the form **(setf (car x) 5)**. This hides the fact that the actual way to set a cons is to use the function **rplaca**. By implementing this duality of syntax we cut in half the number of accessors/setters we need to memorise and, most importantly, enable new ways for us to use macros.

GET-PANDORIC

```
(declaim (inline get-pandoric))

(defun get-pandoric (box sym)
  (funcall box :pandoric-get sym))

(defsetf get-pandoric (box sym) (val)
  `(progn
     (funcall ,box :pandoric-set ,sym ,val)
     ,val))
```

The function **get-pandoric** is a wrapper around the inter-closure protocol getter syntax. It is declaimed inline to eliminate any performance impact caused by this wrapping.

Defsetf is an interesting COMMON LISP macro not entirely unlike our **defmacro!** extension to **defmacro** in that it implicitly binds gensyms around provided forms. **Defsetf** works great for defining the setter side of a generalised variable duality as long as the getter can be expressed as a function or macro that evaluates all of its arguments exactly once. Note that while **get-pandoric** could have been defined as a macro, the only reason to do that would be for inlining purposes. Macros are not for inlining, compilers are for inlining.

So going back to our pandoric counter stored in the symbol-function of **pantest**, we can use this new getter function to retrieve the current value of **pantest**'s **acc** binding:

```
* (get-pandoric #'pantest 'acc)

103
```

And now, thanks to generalised variables and **defsetf**, we can use a syntactic dual to set it:

```
* (setf (get-pandoric #'pantest 'acc) -10)

-10
* (pantest 3)

-7
```

The environments that close over functions—what we've been calling the let in *let over lambda*—are starting to look like regularly accessible generalised variables, just like a cons cell or a hash-table entry. Closures are now even more *first-class* data structures than they used to be. Bindings that were previously closed to outside code are now wide open for us to tinker with, even if those bindings were compiled to something efficient and have long since had their accessor symbols forgotten.

But any discussion of generalised variables is incomplete without a mention of its close relative: the *symbol macro*. **Symbol-macrolet**, like its name implies,

allows us to expand symbols into general lisp forms. Since it is intuitive and more flexible to use forms that look like function calls to represent macro transformations[20] *Symbol macros take no arguments so a symbol macro definition always expands the same.* , there isn't much use for **symbol-macrolet** save one important application for which it is vital: symbol macros let us hide generalised variables such that users of our macro think they are accessing regular lexical variables.

The introduction of symbol macros has resulted in one of the strangest *kludges* to the COMMON LISP language: normally when setting a variable accessed through a regular symbol, like **(setf x t)**, **setf** will expand into a **setq** form because this is what **setq** was originally designed to do: set lexical and dynamic variables (which are always referred to by symbols). But the **setq** special form cannot set generalised variables, so when symbol macros were introduced and it became possible for symbols to represent not just a lexical/dynamic binding but instead any generalised variable, it was necessary to mandate that **setq** forms setting symbols with symbol macro definitions are to be converted back into **setf** forms. Strangely, this really is the *right* thing to do because it allows macros that completely hide the presence of generalised variables from the macro's users, even if they choose to use **setq**. The *really right* solution would be to remove the redundant **setq** from the language in favour of the more general **setf**, but this will not happen for obvious compatibility reasons and because during macro creation **setq** can also be a useful safety shortcut—**setf** plus a check that a symbol has been spliced in instead of a list form. When using **setq** for this remember that it only helps for its splicing safety property; as we've seen, a symbol can reference any generalised variable thanks to **symbol-macrolet**.

WITH-PANDORIC

```
(defmacro! with-pandoric (syms o!box &rest body)
  `(symbol-macrolet
     (,@(mapcar #`(,a1 (get-pandoric ,g!box ',a1))
                syms))
     ,@body))
```

The **with-pandoric** macro expands into a **symbol-macrolet** which defines a symbol macro for each symbol provided in **syms**. Each symbol macro will expand references to its symbol in the lexical scope of the **symbol-macrolet** into generalised variable references using our **get-pandoric** accessor/setter to access the result of evaluating the second argument to the macro: **o!box** (stored in **g!box**).

So **with-pandoric** lets us peek into a closure's closed over variable bindings:

```
* (with-pandoric (acc) #'pantest
    (format t "Value of acc: ~a~%" acc))
Value of acc: -7
NIL
```

As per our design of using generalised variables to form a syntactic dual for the

setting and getting of this variable, we can even pretend it is a regular lexical variable and set it with **setq**:

```
* (with-pandoric (acc) #'pantest
    (setq acc 5))

5
* (pantest 1)

6
```

We have now looked at most of the pieces that make up pandoric macros. First, a macro for creating closures: **pandoriclet**, which captures an anaphor, **this**, referring to the actual function used when invoking the closure. This macro also compiles in some special code that intercepts certain invocations of this closure and instead accesses or modifies its closed-over lexical variables. Second, a single syntax for both accessing and setting an accessor is is implemented with **get-pandoric** and **defsetf**. Finally, the macro **with-pandoric** uses **symbol-macrolet** to install these generalised variables as seemingly new lexical variables with the same names as the closed-over variables. These variables refer to the original environment created with **pandoriclet**, but from separate lexical contexts.

As an example, we relate this ability to open up closures by comparing it to the *hotpatching* macros from section 6.5, Hotpatching Closures. Recall that **alet-hotpatch** and its closed anaphor cousin, **let-hotpatch**, create closures with an indirection environment so that the function that is called when the closure is invoked can be changed on the fly. The biggest limitation with these macros is that they force you to throw out all the lexical bindings that closed over the previous anonymous function when you hotpatched it. This was unavoidable because when we wrote those macros, closures were closed to us.

With **alet-hotpatch** and **let-hotpatch**, we had to compile special purpose code into each closure that was capable of setting the **this** anaphoric lexical binding to its new value. But since we can now open up a closure defined with **pandoriclet** and run this setter code externally, we can define a hotpatching function **pandoric-hotpatch** that will work with any pandoric closure.

PANDORIC-HOTPATCH

```
(defun pandoric-hotpatch (box new)
  (with-pandoric (this) box
    (setq this new)))
```

Sometimes abstractions just feel right and it is hard to exactly say why. Probably because most programming is the disharmonious combining of disjoint parts, it is surprising and pleasant when you discover abstractions that—seemingly by chance—fit together perfectly. **Pandoric-hotpatch** just seems to read exactly like how it works: it opens a pandoric interface, takes the variable **this** from the

lexical scope of the closure **box**, and then uses **setq** to set **this** to the closure to be hotpatched in, **new**.

We can even use **pandoric-hotpatch** on a pandoric closure we created before we knew we wanted it to be hotpatchable. Remember the counter closure we have been playing with throughout this section? It should still be bound to the **symbol-function** of the symbol **pantest**. We were at 6 and counting up when we left off:

```
* (pantest 0)

6
```

Let's install a new closure—one that has a new binding for **acc** starting at 100, and is counting down:

```
* (pandoric-hotpatch #'pantest
    (let ((acc 100))
      (lambda (n) (decf acc n))))

#<Interpreted Function>
```

Sure enough, the hotpatch went through:

```
* (pantest 3)

97
```

So now our counter closure has a new value bound to **this** that it uses to perform the counting. However, did this hotpatch change the pandoric value of **acc** binding?

```
* (with-pandoric (acc) #'pantest
    acc)

6
```

No. **Acc** is still the previous value, 6, because the only binding we changed in the pandoric environment was **this**, and we changed that to a new closure with its own binding for **acc**.

PANDORIC-RECODE

```
(defmacro pandoric-recode (vars box new)
  `(with-pandoric (this ,@vars) ,box
     (setq this ,new)))
```

The macro **pandoric-recode** takes a slightly different approach to hotpatching. It conserves the original lexical environment of the code while still managing to change the function to be executed when the closure is invoked to something coded and compiled externally. Sound too good to be true? Remembering that the current value for **acc** is 6 in the original pandoric environment, we can use

pandoric-recode to install a new function that makes use of this original value and, oh, let's say, decrements the counter by half the value of the provided **n**:

```
* (pandoric-recode (acc) #'pantest
    (lambda (n)
      (decf acc (/ n 2))))

#<Interpreted Function>
```

Sure enough, we have the new behaviour, which decrements **acc** by **(* 1/2 2)**, from 6 to 5:

```
* (pantest 2)

5
```

And is it associated with the original pandoric binding?

```
* (with-pandoric (acc) #'pantest
    acc)

5
```

Yes. How does **pandoric-recode** work? It closes over the provided lambda form with the pandorically opened bindings of the original closure.

PLAMBDA

```
(defmacro plambda (largs pargs &rest body)
  (let ((pargs (mapcar #'list pargs)))
    `(let (this self)
       (setq
         this (lambda ,largs ,@body)
         self (dlambda
                (:pandoric-get (sym)
                  ,(pandoriclet-get pargs))
                (:pandoric-set (sym val)
                  ,(pandoriclet-set pargs))
                (t (&rest args)
                  (apply this args)))))))
```

The macro we have used so far to create pandoric closures is **pandoriclet**. **Plambda** is an inside out re-write of **pandoriclet** that adds a few important features. First and foremost, **plambda** no longer creates the let environment to be used through our pandoric accessors. Instead, **plambda** takes a list of symbols that refer to variables that are expected to be in the caller's lexical environment. **Plambda** can *export* any variables in your lexical environment, transparently making them available for other lexical scopes to access—even ones written and compiled before or after the **plambda** form is.

This is an incremental improvement to our *let over lambda* closure system designed to maximise dual syntax. Thanks to pandoric macros, the most important of which are **plambda** and **with-pandoric**, we can easily and efficiently transcend the boundaries of lexical scope when we need to. Closures are no longer closed; we can open closures as easily as re-writing our lambda forms to be plambda forms. We use **plambda** to export lexical variables and **with-pandoric** to import them as completely equivalent lexical variables. In fact these new variables are so equivalent that they aren't even really new variables at all. A better way of thinking about pandoric variables are that they are simply an extension of the original lexical scope. As a simple example use of **plambda**, here is a pandoric counter that exports variables from two potentially different lexical environments:

```
* (setf (symbol-function 'pantest)
    (let ((a 0))
      (let ((b 1))
        (plambda (n) (a b)
          (incf a n)
          (setq b (* b n))))))

#<Interpreted Function>
```

Notice how easy it was to export these lexical references. Making a closure pandoric is as easy as adding a **p** character before the **lambda** and adding a list of variables to export after the lambda arguments. We can the open this closure—and any pandoric closure that exports the symbols **a** and **b**—by using **with-pandoric** like so:

```
* (defun pantest-peek ()
    (with-pandoric (a b) #'pantest
      (format t "a=~a, b=~a~%" a b)))

PANTEST-PEEK
* (pantest-peek)

a=0, b=1
NIL
```

Plambda is an example of how factoring out general components of macro expansions can be helpful. Remember when we wrote **pandoriclet** and decided to move the creation of **case** statements for the getter code to the function **pandoriclet-get** and the setter code to **pandoriclet-get**? **Plambda** makes use of these same functions. Even though these macros splice the results from these functions into fairly different lexical contexts, since both macros have been written to use the same variable naming convention and inter-closure protocol, the code is re-usable.

So pandoric macros break down lexical boundaries. They allow you to open up

closures whenever needed and represent a beautiful confluence of a variety of COMMON LISP language features: anaphoric macros, generalised variables, and symbol macros. But what good are they, really?

MAKE-STATS-COUNTER

```
(defun make-stats-counter
       (&key (count 0)
             (sum 0)
             (sum-of-squares 0))
  (plambda (n) (sum count sum-of-squares)
    (incf sum-of-squares (expt n 2))
    (incf sum n)
    (incf count)))
```

Pandoric macros are important because they give us the main advantages of object systems like CLOS without requiring us to depart from the more natural let-lambda combination programming style. In particular, we can add functionality, or *methods*, for closures to use without having to re-instantiate instances of already created objects.

Make-stats-counter is a lambda over let over plambda we have created to create counters, except that it maintains three pieces of information. In addition to the sum, the sum of the squares, and the number of items so far processed are also kept. If we had used **lambda** instead of **plambda** in the definition of **make-stats-counter**, most of this information would be inaccessible to us. We would be locked out because these variables would be closed to us.

How do we write pandoric methods? We can simply access the variables using **with-pandoric** as we have demonstrated above, or, since this is lisp, design a more specific interface.

DEFPAN

```
(defmacro defpan (name args &rest body)
  `(defun ,name (self)
     ,(if args
        `(with-pandoric ,args self
           ,@body)
        `(progn ,@body))))
```

Defpan is a *combination* of the **defun** and **with-pandoric** macros. **Defpan**'s main purpose is to enable a *duality of syntax* between function writing using **defun** and foreign lexical scope access using **with-pandoric**. Although we provide arguments to **defpan** using the same syntax as in lambda forms—a list of symbols—the arguments to **defpan** mean something different. Instead of creating a new lexical environment, these *pandoric functions* extend the lexical environment of the pandoric closures they are applied to. With **defun** and regular lambda forms, the name (symbol) you give a variable is unimportant. With pandoric functions, it is everything. Furthermore, with pandoric functions

the order of the arguments doesn't matter and you can elect to use as few or as many of the exported lexical variables as you please.

Defpan also provides an anaphor called **self** that allows us to perform a useful technique called *anaphor chaining*. By invisibly passing the value of **self** between pandoric functions, we can maintain the value of this anaphor throughout a chain of function calls. As with all chaining constructs, be sure you don't end up in an infinite loop.

STATS-COUNTER-METHODS

```
(defpan stats-counter-mean (sum count)
  (/ sum count))

(defpan stats-counter-variance
        (sum-of-squares sum count)
  (if (< count 2)
    0
    (/ (- sum-of-squares
          (* sum
             (stats-counter-mean self)))
       (- count 1))))

(defpan stats-counter-stddev ()
  (sqrt (stats-counter-variance self)))
```

Three methods are presented that can be used on our closures created with **make-stats-counter** or any other pandoric closure that exports the necessary variable names. **Stats-counter-mean** simply returns the averaged value of all the values that have been passed to the closure. **Stats-counter-variance** computes the variance of these values by following a link in the chain and **stats-counter-stddev** follows yet another to compute the standard deviation. Notice that each link in the chain only needs to pass on the anaphor **self** to refer to the full lexical context of the closure. We see that the individual pandoric functions only need to reference the variables they actually use and that these variables can be referred to in any order we wish.

So **plambda** creates another anaphor—**self**. While the anaphor **this** refers to the actual closure that is to be invoked, **self** refers to the indirection environment that calls this closure. Although it sounds a bit peculiar, code inside our **plambda** can use **self** to pandorically access its own lexical environment instead of directly accessing it. This so far only seems useful for **defpan** methods that have been written to work inside our lexical scope.

MAKE-NOISY-STATS-COUNTER

```
(defun make-noisy-stats-counter
       (&key (count 0)
             (sum 0)
```

```
            (sum-of-squares 0))
  (plambda (n) (sum count sum-of-squares)
    (incf sum-of-squares (expt n 2))
    (incf sum n)
    (incf count)

    (format t
      "~&MEAN=~a~%VAR=~a~%STDDEV=~a~%"
           (stats-counter-mean self)
           (stats-counter-variance self)
           (stats-counter-stddev self))))
```

Make-noisy-stats-counter is identical to **make-stats-counter** except that it uses the **self** anaphor to invoke our **defpan** functions **stats-counter-mean**, **stats-counter-variance**, and **stats-counter-stddev**.

Plambda and **with-pandoric** can re-write lexical scope in any way we please. We conclude this chapter with such an example. A limitation of lexical scope sometimes lamented upon is the fact that the COMMON LISP function **eval** will throw out your current lexical environment when it evaluates the form passed to it. In other words, **eval** evaluates the form in a *null lexical environment*. In COMMON LISP it couldn't be any other way: **eval** is a function. Here is the problem:

```
* (let ((x 1))
    (eval
      '(+ x 1)))

Error: The variable X is unbound.
```

Sometimes it would apparently be desirable to extend your lexical environment to **eval**. But be careful. Often it is said that if you are using **eval** you are probably doing something wrong. Misuse of **eval** can result in slower programs because **eval** can be a very expensive operation—mostly because it needs to expand macros present in the form passed to it. Should you suddenly find a need for **eval** when programming, ask yourself why you didn't do whatever it is you want to do a lot earlier. If the answer is that you couldn't have, say because you just read the form in, then congratulations, you have found one of the rare legitimate uses of **eval**. Any other answers will lead straight back to the way you probably should have done it in the first place: with a macro.

PANDORIC-EVAL

```
(defvar pandoric-eval-tunnel)

(defmacro pandoric-eval (vars expr)
  `(let ((pandoric-eval-tunnel
           (plambda () ,vars t)))
     (eval `(with-pandoric
```

```
              ,',vars pandoric-eval-tunnel
              ,,expr))))
```

But let's say that you really do want to **eval** something, if only you could carry along that pesky lexical context. The **pandoric-eval** macro is a fun example use **plambda** and **with-pandoric**. **Pandoric-eval** uses a special variable that we have named **pandoric-eval-tunnel** to make a pandoric closure available to the **eval** function through the dynamic environment. We choose exactly which lexical variables to *tunnel* through the dynamic environment by providing a list of all their symbols as the first argument to **pandoric-eval**. Here it is applied to our earlier example:

```
* (let ((x 1))
    (pandoric-eval (x)
      '(+ 1 x)))

2
```

And the expression evaluated by **pandoric-eval** can modify the original lexical environment; **pandoric-eval** is a two way tunnel:

```
* (let ((x 1))
    (pandoric-eval (x)
      '(incf x))
    x)

2
```

This section, although very lengthy, has still only scratched the surface of what is possible with pandoric macros and their many possible variations. I am looking forward to the many interesting future developments that will come out of them.

Exercise: Can **pandoric-eval** calls nest? That is, can you use **pandoric-eval** to evaluate a form that evaluates **pandoric-eval**? Why or why not?

Exercise: Although the implementation of pandoric macros here is fairly efficient, it could be improved. Try replacing **pandoriclet-get** and **pandoriclet-set** to generate code that uses a hash-table instead of **case** and benchmark these two implementations for small and large numbers of pandoric variables. Investigate your favourite CLOS implementation, imitate how dispatching is done there, re-benchmark.