Let Over Lambda

Let Over Lambda -- 50 Years of Lisp

by Doug Hoyte

Closures

Closure-Oriented Programming

One of the conclusions that we reached was that the "object" need not be a primitive notion in a programming language; one can build objects and their behaviour from little more than assignable value cells and good old lambda expressions. —Guy Steele on the design of Scheme

Sometimes it's called a *closure*, other times a saved lexical environment. Or, as some of us like to say, *let over lambda*. Whatever terminology you use, mastering this concept of a closure is the first step to becoming a professional lisp programmer. In fact, this skill is vital for the proper use of many modern programming languages, even ones that don't explicitly contain let or lambda, such as Perl or Javascript.

Closures are one of those few curious concepts that are paradoxically difficult because they are so simple. Once a programmer becomes used to a complex solution to a problem, simple solutions to the same problem feel incomplete and uncomfortable. But, as we will soon see, closures can be a simpler, more direct solution to the problem of how to organise data and code than objects. Even more important than their simplicity, closures represent a better abstraction to use when constructing macros—the topic of this book.

The fact that we can build objects and classes with our closure primitives doesn't mean that object systems are useless to lisp programmers. Far from it. In fact, COMMON LISP includes one of the most powerful object systems ever devised: *CLOS*, the COMMON LISP Object System. Although I am very impressed with the flexibility and features of CLOS, I seldom find a need to use its more advanced features[1] *CLOS is so central to COMMON LISP that it is literally impossible to program in COMMON LISP without it.*

, thanks to assignable value cells and good old lambda expressions.

While much of this book assumes a reasonable level of lisp skill, this chapter attempts to teach the theory and use of closures from the very basics as well as to provide a common terminology for closures that will be used throughout the rest of this book. This chapter also examines the efficiency implications of closures and considers how well modern compilers optimise them.

Environments and Extent

What Steele means by assignable value cells is an environment for storing pointers to data where the environment is subject to something called *indefinite extent*. This is a fancy way of saying that we can continue to refer to such an environment at any time in the future. Once we allocate this environment, it and its references are there to stay as long as we need them. Consider this C function:

```
#include <stdlib.h>

int *environment_with_indefinite_extent(int input) {
  int *a = malloc(sizeof(int));
  *a = input;
  return a;
}
```

After we call this function and receive the pointer it returns, we can continue to refer to the allocated memory indefinitely. In C, new environments are created when invoking a function, but C programmers know to **malloc()** the required memory when returning it for use outside the function.

By contrast, the example below is flawed. C programmers consider **a** to be automatically collected when the function returns because the environment is allocated on the *stack*. In other words, according to lisp programmers, **a** is allocated with *temporary extent*.

```
int *environment_with_temporary_extent(int input) {
  int a = input;
  return &a;
}
```

The difference between C environments and lisp environments is that unless you explicitly tell lisp otherwise it always assumes you mean to use indefinite extent. In other words, lisp always assumes you mean to call **malloc()** as above. It can be argued that this is inherently less efficient than using temporary extent, but the benefits almost always exceed the marginal performance costs. What's more, lisp can often determine when data can safely be allocated on the stack and will do so automatically. You can even use *declarations* to tell lisp to do this explicitly. We will discuss declarations in more detail in chapter 7, Macro Efficiency Topics.

But because of lisp's dynamic nature, it doesn't have explicit pointer values or types like C. This can be confusing if you, as a C programmer, are used to casting pointers and values to indicate types. Lisp thinks about all this slightly differently. In lisp, a handy mantra is the following:

Variables don't have types. Only values have types.

Still, we have to return something to hold pointers. In lisp there are many data structures that can store pointers. One of the most favoured by lisp programmers is a simple structure: the *cons cell*. Each cons cell holds exactly two pointers, affectionately called car and cdr. When **environment-with-indefinite-extent** is invoked, a cons cell will be returned with the car pointing to whatever was passed as **input** and the cdr pointing to **nil**. And, most importantly, this cons cell (and with it the pointer to **input**) has indefinite extent so we can continue to refer to it as long as we need to:

```
(defun environment-with-indefinite-extent (input)
  (cons input nil))
```

The efficiency disadvantages of indefinite extent are approaching irrelevance as the state of the art in lisp compilation technology improves. Environments and extent are closely related to closures and more will be said about them throughout this chapter.

Lexical and Dynamic Scope

The technical term for where to consider a variable reference valid is *scope*. The most common type of scope in modern languages is called *lexical* scope. When a fragment of code is surrounded by the lexical binding of a variable, that variable is said to be in the lexical scope of the binding. The **let** form, which is one of the most common ways to create bindings, can introduce these lexically scoped variables:

```
* (let ((x 2))
    x)

2
```

The **x** inside the body of the **let** form was accessed through lexical scope. Similarly, arguments to functions defined by **lambda** or **defun** are also lexically bound variables inside the text of the function definition. Lexical variables are variables that can only be accessed by code appearing inside the context of, for instance, the above **let** form. Because lexical scoping is such an intuitive way to limit the scope of access to a variable, it can appear to be the only way. Are there any other possibilities for scoping?

As useful as the combination of indefinite extent and lexical scoping turns out to be, it has until recently not been used to its fullest extent in mainstream programming languages. The first implementation was by Steve Russell for Lisp 1.5[HISTORY-OF-LISP] and was subsequently designed directly into languages like Algol-60, Scheme, and COMMON LISP. Despite this long and fruitful history, the numerous advantages of lexical scoping are only slowly being taken up by many Blubs.

Although the scoping methods provided by C-like languages are limited, C programmers need to program across different environments too. To do so, they often use an imprecisely defined scoping known as *pointer scope*. Pointer scope is famous for its difficulty to debug, numerous security risks, and, somewhat artificially, its efficiency. The idea behind pointer scoping is to define a domain specific language for controlling the registers and memory of a Von Neumman machine similar to most modern CPUs[PAIP-PIX], then to use this language to access and manipulate data-structures with fairly direct commands to the CPU running the program. Pointer scoping was necessary for performance reasons before decent lisp compilers were invented but is now regarded as a problem with, rather than a feature of, modern programming languages.

Even though lisp programmers seldom think in terms of pointers, the understanding of pointer scoping is very valuable in the construction of efficient lisp code. In section 7.4, Pointer Scope we will investigate implementing pointer scoping for the rare cases where we need to instruct the compiler on specific code creation. But for now we only need discuss its mechanics. In C, we sometimes would like to access a variable defined outside the function we are writing:

```
#include <stdio.h>

void pointer_scope_test() {
  int a;
  scanf("%d", &a);
}
```

In the above function we use the C **&** operator to give the actual address in memory of our local variable **a** to the **scanf** function so it knows where to write the data it scans. Lexical scoping in lisp forbids us from implementing this directly. In lisp, we would likely pass an anonymous function to a hypothetical lisp **scanf** function, allowing it to set our lexical variable **a** even though **scanf** is defined outside our lexical scope:

```
(let (a)
  (scanf "%d" (lambda (v) (setf a v))))
```

Lexical scope is the enabling feature for closures. In fact, closures are so related to this concept of lexical scope that they are often referred to more specifically as *lexical closures* to distinguish them from other types of closures. Unless otherwise noted, all closures in this book are lexical.

In addition to lexical scope, COMMON LISP provides *dynamic scope*. This is lisp *slang* for the combination of temporary extent and global scope. Dynamic scoping is a type of scoping that is unique to lisp in that it offers a very different behaviour but shares an identical syntax with lexical scope. In COMMON LISP we deliberately choose to call attention to variables accessed with dynamic scope by calling them *special variables*. These special variables can be defined with **defvar**. Some programmers follow a convention of prefixing and postfixing special variable names with asterisks, like ***temp-special***. This is called the *earmuff* convention. For reasons explained in section 3.7, Duality of Syntax, this book does not use earmuffs so our special variable declarations look like this:

```
(defvar temp-special)
```

When defined like this, **temp-special** will be designated special[2] *We can also indicate the specialness of variables by usin* but will not be initialised with a value. In this state, a special variable is said to be *unbound*. Only special variables can be unbound—lexical variables are always bound and thus always have values. Another way of thinking of this is that by default all symbols represent lexically unbound variables. Just as with lexical variables, we can assign a value to special variables with **setq** or **setf**. Some lisps, like Scheme, do not have dynamic scope. Others, like EuLisp[SMALL-PIECES-P46], use different syntax for accessing lexical versus special variables. But in COMMON LISP the syntax is shared. Many lispers consider this a feature. Here we assign a value to our special variable **temp-special**:

```
(setq temp-special 1)
```

So far, this special variable doesn't seem that special. It seems to be just another variable, bound in some sort of global namespace. This is because we have only bound it once—its default special global binding. Special variables are most interesting when they are re-bound, or *shadowed*, by new environments. If we define a function that simply evaluates and returns **temp-special**:

```
(defun temp-special-returner ()
  temp-special)
```

This function can be used to examine the value that lisp evaluates **temp-special** to be at the moment in time when it was called:

```
* (temp-special-returner)

1
```

This is sometimes referred to as evaluating the form in a *null lexical environment*. The null lexical environment obviously doesn't contain any lexical bindings. Here the value of **temp-special** returned is that of its global special value, 1. But if we evaluate it in a non-null lexical environment—one that contains a binding for our special variable—the specialness of **temp-special** reveals itself[3] *Because we create a dynamic binding we are not actually creating a lexical environment. It just looks that way.*
:

```
* (let ((temp-special 2))
    (temp-special-returner))

2
```

Notice that the value 2 was returned, meaning that the **temp-special** value was taken from our **let** environment, not its global special value. If this still does not seem interesting, see how this cannot be done in most other conventional programming languages as exemplified by this piece of Blub pseudo-code:

```
int global_var = 0;

function whatever() {
  int global_var = 1;
  do_stuff_that_uses_global_var();
}

function do_stuff_that_uses_global_var() {
  // global_var is 0
}
```

While the memory locations or register assignments for lexical bindings are known at compile-time[4] *Sometimes lexical scoping is called "static scoping" for this reason.*
, special variable bindings are determined at run-time—in a sense. Thanks to a clever trick, special variables aren't as inefficient as they seem. A special variable actually always does refer to the same location in memory. When you use **let** to bind a special variable, you are actually compiling in code that will store a copy of the variable, over-write the memory location with a new value, evaluate the forms in the let body, and, finally, restore the original value from the copy.

Special variables are perpetually associated with the symbol used to name them. The location in memory referred to by a special variable is called the **symbol-value** cell of a symbol. This is in direct contrast to lexical variables. Lexical variables are only indicated with symbols at compile-time. Because lexical variables can only be accessed from inside the lexical scope of their bindings, the compiler has no reason to even remember the symbols that were used to reference lexical variables so it will remove them from compiled code. We will stretch the truth of this statement in section 6.7, Pandoric Macros.

Although COMMON LISP does offer the invaluable feature of dynamic scope, lexical variables are the most common. Dynamic scoping used to be a defining feature of lisp but has, since COMMON LISP, been almost completely replaced by lexical scope. Since lexical scoping enables things like lexical closures (which we examine shortly), as well as more effective compiler optimisations, the superseding of dynamic scope is mostly seen as a good thing. However, the designers of COMMON LISP have left us a very transparent window into the world of dynamic scoping, now acknowledged for what it really is: special.

Let It Be Lambda

Let is a lisp special form for creating an environment with names (bindings) initialised to the results of evaluating corresponding forms. These names are available to the code inside the **let** body while its forms are evaluated consecutively, returning the result of the final form. Although what **let** does is unambiguous, how it does it is deliberately left unspecified. What **let** does is separated from how it does it. Somehow, **let** needs to provide a data structure for storing pointers to values.

Cons cells are undeniably useful for holding pointers, as we saw above, but there are numerous structures that can be used. One of the best ways to store pointers in lisp is to let lisp take care of it for you with the **let** form. With **let** you only have to name (bind) these pointers and lisp will figure out how best to store them for you. Sometimes we can help the compiler make this more efficient by giving it extra bits of information in the form of declarations:

```
(defun register-allocated-fixnum ()
  (declare (optimize (speed 3) (safety 0)))
  (let ((acc 0))
    (loop for i from 1 to 100 do
      (incf (the fixnum acc)
            (the fixnum i)))
    acc))
```

For example, in **register-allocated-fixnum** we provide some hints to the compiler that allow it to sum the integers from 1 to 100 very efficiently. When compiled, this function will allocate the data in registers, eliminating the need for pointers altogether. Even though it seems we've asked lisp to create an indefinite extent environment to hold **acc** and **i**, a lisp compiler will be able to optimise this function by storing the values solely in CPU registers. The result might be this machine code:

```
; 090CEB52:       31C9             XOR ECX, ECX
;       54:       B804000000       MOV EAX, 4
;       59:       EB05             JMP L1
;       5B: L0:   01C1             ADD ECX, EAX
;       5D:       83C004           ADD EAX, 4
;       60: L1:   3D90010000       CMP EAX, 400
;       65:       7EF4             JLE L0
```

Notice that 4 represents 1 and 400 represents 100 because fixnums are shifted by two bits in compiled code. This has to do with *tagging*, a way to pretend that something is a pointer but actually store data inside it. Our lisp compiler's tagging scheme has the nice benefit that no shifting needs to occur to index word aligned memory[DESIGN-OF-CMUCL]. We'll get to know our lisp compiler better in chapter 7, Macro Efficiency Topics.

But if lisp determines that you might want to refer to this environment later on it will have to use something less transient than a register. A common structure for storing pointers in environments is an array. If each environment has an array and all the variable references enclosed in that environment are just references into this array, we have an efficient environment with potentially indefinite extent.

As mentioned above, **let** will return the evaluation of the last form in its body. This is common for many lisp special forms and macros, so common that this pattern is often referred to as an *implicit progn* due to the **progn** special form designed to do nothing but this[5] *__Progn__ is actually also useful for clustering forms to give them all top-level behaviour.*

. Sometimes the most valuable thing to have a let form return is an anonymous function which takes advantage of the lexical environment supplied by the let form. To create these functions in lisp we use *lambda*.

Lambda is a simple concept that can be intimidating because of its flexibility and importance. The lambda from lisp and scheme owes its roots to Alonzo Church's logic system but has evolved and adapted into its altogether own lisp specification. Lambda is a concise way to repeatably assign temporary names (bindings) to values for a specific lexical context and underlies lisp's concept of a function. A lisp function is very different from the mathematical function description that Church had in mind. This is because lambda has evolved as a powerful, practical tool at the hands of generations of lispers, stretching and extending it much further than early logicians could have foreseen.

Despite the reverence lisp programmers have for lambda, there is nothing inherently special about the notation. As we will see, lambda is just one of many possible ways to express this sort of variable naming. In particular, we will see that macros allow us to customise the renaming of variables in ways that are effectively impossible in other programming languages. But after exploring this, we will return to lambda and discover that it is very close to the optimal notation for expressing such naming. This is no accident. Church, as dated and irrelevant as he might seem to our modern programming environment, really was on to something. His mathematical notation, along with its numerous enhancements in the hands of generations of lisp professionals, has evolved into a flexible, general tool[6] *The classic example of a macro is an implementation of __let__ as a lambda form. I will not bore you with that in this book.*

.

Lambda is so useful that, like many of lisp's features, most modern languages are beginning to import the idea from lisp into their own systems. Some language designers feel that lambda is too lengthy, instead using **fn** or some other abbreviation. On the other hand, some regard lambda as a concept so fundamental that obscuring it with a lesser name is next to heresy. In this book, although we will describe and explore many variations on lambda, we happily call it lambda, just as generations of lisp programmers before us.

But what is lisp's lambda? First off, as with all names in lisp, lambda is a *symbol*. We can quote it, compare it, and store it in lists. Lambda only has a special meaning when it appears as the first element of a list. When it appears there, the list is referred to as a *lambda form* or as a *function designator*. But this form is not a function. This form is a list data structure that can be converted into a function using the **function** special form:

```
* (function '(lambda (x) (+ 1 x)))
```

```
#<Interpreted Function>
```

COMMON LISP provides us a convenience shortcut for this with the #' (sharp-quote) read macro. Instead of writing **function** as above, for the same effect we can take advantage of this shortcut:

```
* #'(lambda (x) (+ 1 x))

#<Interpreted Function>
```

As a further convenience feature, lambda is also defined as a macro that expands into a call to the function special form above. The COMMON LISP ANSI standard requires[ANSI-CL-ISO-COMPATIBILITY] a **lambda** macro defined like so:

```
(defmacro lambda (&whole form &rest body)
  (declare (ignore body))
  `#',form)
```

Ignore the ignore declaration for now[7] *A U-Language declaration.* . This macro is just a simple way to automatically apply the **function** special form to your function designators. This macro allows us to evaluate function designators to create functions because they are expanded into sharp-quoted forms:

```
* (lambda (x) (+ 1 x))

#<Interpreted Function>
```

There are few good reasons to prefix your lambda forms with #' thanks to the **lambda** macro. Because this book makes no effort to support pre-ANSI COMMON LISP environments, backwards compatibility reasons are easily rejected. But what about stylistic objections? Paul Graham, in *ANSI COMMON LISP*[GRAHAM-ANSI-CL], considers this macro, along with its brevity benefits, a "specious sort of elegance at best". Graham's objection seems to be that since you still need to sharp-quote functions referenced by symbols, the system seems asymmetric. However, I believe that not sharp-quoting lambda forms is actually a stylistic improvement because it highlights the asymmetry that exists in the second namespace specification. Using sharp-quote for symbols is for referring to the second namespace, whereas functions created by lambda forms are, of course, nameless.

Without even invoking the **lambda** macro, we can use lambda forms as the first argument in a function call. Just like when a symbol is found in this position and lisp assumes we are referencing the **symbol-function** cell of the symbol, if a lambda form is found, it is assumed to represent an anonymous function:

```
* ((lambda (x) (+ 1 x)) 2)

3
```

But note that just as you can't call a function to dynamically return the symbol to be used in a regular function call, you can't call a function to return a lambda form in the function position. For both of these tasks, use either **funcall** or **apply**.

A benefit of lambda expressions that is largely foreign to functions in C and other languages is that lisp compilers can often optimise them out of existence completely. For example, although **compiler-test** looks like it applies an increment function to the number 2 and returns the result, a decent compiler will be smart enough to know that this function always returns the value 3 and will simply return that number directly, invoking no functions in the process. This is called *lambda folding*:

```
(defun compiler-test ()
  (funcall
    (lambda (x) (+ 1 x))
    2))
```

An important efficiency observation is that a compiled lambda form is a constant form. This means that after your program is compiled, all references to that function are simply pointers to a chunk of machine code. This pointer can be returned from functions and embedded in new environments, all with no function creation overhead. The overhead was absorbed when the program was compiled. In other words, a function that returns another function will simply be a constant time pointer return function:

```
(defun lambda-returner ()
  (lambda (x) (+ 1 x)))
```

This is in direct contrast to the **let** form, which is designed to create a new environment at run-time and as such is usually not a constant operation because of the garbage collection overhead implied by lexical closures, which are of indefinite extent.

```
(defun let-over-lambda-returner ()
  (let ((y 1))
    (lambda (x)
      (incf y x))))
```

Every time **let-over-lambda-returner** is invoked, it must create a new environment, embed the constant pointer to the code represented by the lambda form into this new environment, then return the resulting *closure*. We can use **time** to see just how small this environment is:

```
* (progn
    (compile 'let-over-lambda-returner)
    (time (let-over-lambda-returner)))

; Evaluation took:
;   ...
;   24 bytes consed.
;
```

```
#<Closure Over Function>
```

If you try to call compile on a closure, you will get an error saying you can't compile functions defined in non-null lexical environments[CLTL2-P677]. You can't compile closures, only the functions that create closures. When you compile a function that creates closures, the closures it creates will also be compiled[ON-LISP-P25].

The use of a let enclosing a lambda above is so important that we will spend the remainder of this chapter discussing the pattern and variations on it.

Let Over Lambda

Let over lambda is a nickname given to a lexical closure. Let over lambda more closely mirrors the lisp code used to create closures than does most terminology. In a let over lambda scenario, the last form returned by a **let** statement is a **lambda** expression. It literally looks like **let** is sitting on top of **lambda**:

```
* (let ((x 0))
    (lambda () x))

#<Interpreted Function>
```

Recall that the **let** form returns the result of evaluating the last form inside its body, which is why evaluating this let over lambda form produced a function. However, there is something special about the last form in the **let**. It is a **lambda** form with **x** as a *free variable*. Lisp was smart enough to determine what **x** should refer to for this function: the **x** from the surrounding lexical environment created by the **let** form. And, because in lisp everything is of indefinite extent by default, the environment will be available for this function to use as long as it needs it.

So lexical scope is a tool for specifying exactly where references to a variable are valid, and exactly what the references refer to. A simple example of a closure is a *counter*, a closure that stores an integer in an environment and increments and returns this value upon every invocation. Here is how it is typically implemented, with a let over lambda:

```
(let ((counter 0))
  (lambda () (incf counter)))
```

This closure will return 1 the first time it is called, 2 the subsequent time, and so on. One way of thinking about closures is that they are functions with *state*. These functions are not mathematical functions, but rather procedures, each with a little memory of its own. Sometimes data structures that bundle together code and data are called *objects*. An object is a collection of procedures and some associated state. Since objects are so closely related to closures, they can often be thought of as one and the same. A closure is like an object that has exactly one method: **funcall**. An object is like a closure that you can **funcall** in multiple ways.

Although closures are always a single function and its enclosing environment, the multiple methods, inner classes, and static variables of object systems all have their closure counterparts. One possible way to simulate multiple methods is to simply return multiple **lambda**s from inside the same lexical scope:

```
(let ((counter 0))
  (values
    (lambda () (incf counter))
    (lambda () (decf counter))))
```

This *let over two lambdas* pattern will return two functions, both of which access the same enclosing counter variable. The first increments it and the second decrements it. There are many other ways to accomplish this. One of which, **dlambda**, is discussed in section 5.7, Dlambda. For reasons that will be explained as we go along, the code in this book will structure all data using closures instead of objects. Hint: It has to do with macros.

Lambda Over Let Over Lambda

In some object systems there is a sharp distinction between objects, collections of procedures with associated state, and classes, the data structures used to create objects. This distinction doesn't exist with closures. We saw examples of forms you can evaluate to create closures, most of them following the pattern let over lambda, but how can our program create these objects as needed?

The answer is profoundly simple. If we can evaluate them in the REPL, we can evaluate them inside a function too. What if we create a function whose sole purpose is to evaluate a let over lambda and return the result? Because we use **lambda** to represent functions, it would look something like this:

```
(lambda ()
  (let ((counter 0))
    (lambda () (incf counter))))
```

When the *lambda over let over lambda* is invoked, a new closure containing a counter binding will be created and returned. Remember that **lambda** expressions are constants: mere pointers to machine code. This expression is a simple bit of code that creates new environments to close over the inner **lambda** expression (which is itself a constant, compiled form), just as we were doing at the REPL.

With object systems, a piece of code that creates objects is called a class. But lambda over let over lambda is subtly different than the classes of many languages. While most languages require classes to be named, this pattern avoids naming altogether. Lambda over let over lambda forms can be called *anonymous classes*.

Although anonymous classes are often useful, we usually do name classes. The easiest way to give them names is to recognise that such classes are regular functions. How do we normally name functions? With the **defun** form, of course. After naming, the above anonymous class becomes:

```
(defun counter-class ()
  (let ((counter 0))
    (lambda () (incf counter))))
```

Where did the first **lambda** go? **Defun** supplies an *implicit lambda* around the forms in its body. When you write regular functions with **defun** they are still lambda forms underneath but this fact is hidden beneath the surface of the **defun** syntax.

Unfortunately, most lisp programming books don't provide realistic examples of closure usage, leaving readers with the inaccurate impression that closures are only good for toy examples like counters. Nothing could be further from the truth. Closures are the building blocks of lisp. Environments, the functions defined inside those environments, and macros like **defun** that make using them convenient, are all that are needed for modelling any problem. This book aims to stop beginning lisp programmers used to object-based languages from acting upon their gut instinct of reaching for systems like CLOS. While CLOS does have certain things to offer the professional lisp programmer, do not use it when a lambda will suffice.

BLOCK-SCANNER

```
(defun block-scanner (trigger-string)
  (let* ((trig (coerce trigger-string 'list))
         (curr trig))
    (lambda (data-string)
      (let ((data (coerce data-string 'list)))
        (dolist (c data)
          (if curr
            (setq curr
                  (if (char= (car curr) c)
                    (cdr curr) ; next char
                    trig))))   ; start over
        (not curr))))) ; return t if found
```

In order to motivate the use of closures, a realistic example is presented: **block-scanner**. The problem **block-scanner** solves is that for some forms of data transfer the data is delivered in groups (blocks) of uncertain sizes. These sizes are generally convenient for the underlying system but not for the application programmer, often being determined by things like operating system buffers, hard drive blocks, or network packets. Scanning a stream of data for a specific sequence requires more than just scanning each block as it comes in with a regular, stateless procedure. We need to keep state between the scanning of each block because it is possible that the sequence we are scanning for will be split between two (or more) blocks.

The most straightforward, natural way to implement this stored state in modern languages is with a closure. An initial sketch of a closure-based block scanner is given as **block-scanner**. Like all lisp development, creating closures is an iterative process. We might start off with code given in **block-scanner** and decide to improve its efficiency by avoiding coercion of strings to lists, or possibly improve the information gathered by counting the number of occurrences of the sequence.

Although **block-scanner** is an initial implementation waiting to be improved, it is still a good demonstration of the use of lambda over let over lambda. Here is a demonstration of its use, pretending to be some sort of communications tap watching out for a specific black-listed word, *jihad*:

```
* (defvar scanner
    (block-scanner "jihad"))

SCANNER
* (funcall scanner "We will start ")

NIL
# (funcall scanner "the ji")

NIL
* (funcall scanner "had tomorrow.")

T
```

Let Over Lambda Over Let Over Lambda

Users of object systems store values they want shared between all objects of a certain class into so-called *class variables* or *static variables*[8] *The term static is one of the most overloaded programming language terms. Variables shared by all objects of a class are call*

. In lisp, this concept of sharing state between closures is handled by environments in the same way that closures themselves store state. Since an environment is accessible indefinitely, as long as it is still possible to reference it, we are guaranteed that it will be available as long as is needed.

If we want to maintain a global direction for all counters, **up** to increment each closure's counter and **down** to decrement, then we might want to use a let over lambda over let over lambda pattern:

```
(let ((direction 'up))
  (defun toggle-counter-direction ()
    (setq direction
          (if (eq direction 'up)
            'down
            'up)))

  (defun counter-class ()
    (let ((counter 0))
      (lambda ()
        (if (eq direction 'up)
          (incf counter)
          (decf counter)))))
```

In the above example, we have extended **counter-class** from the previous section. Now calling closures created with **counter-class** will either increment its counter binding or decrement it, depending on the value of the direction binding which is shared between all counters. Notice that we also take advantage of another **lambda** inside the direction environment by creating a function called **toggle-counter-direction** which changes the current direction for all counters.

While this combination of **let** and **lambda** is so useful that other languages have adopted it in the form of class or static variables, there exist other combinations of **let** and **lambda** that allow you to structure code and state in ways that don't have direct analogs in object systems[9] *But these analogs can sometimes be built on top of object systems.* . Object systems are a formalisation of a subset of let and lambda combinations, sometimes with gimmicks like *inheritance* bolted on[10] *Having macros is immeasurably more important than having inheritance.* . Because of this, lisp programmers often don't think in terms of classes and objects. Let and lambda are fundamental; objects and classes are derivatives. As Steele says, the "object" need not be a primitive notion in programming languages. Once assignable value cells and good old lambda expressions are available, object systems are, at best, occasionally useful abstractions and, at worst, special-case and redundant.

All material is (C) Doug Hoyte unless otherwise noted or implied. All rights reserved.