CS 61A Lecture Notes Week 8

Topic: Local state variables, environments

Reading: Abelson & Sussman, Section 3.1, 3.2; OOP below the line

We said the three big ideas in the OOP interface are message passing, local state, and inheritance. You know from section 2.4 how message passing is implemented below the line in Scheme, i.e., with a dispatch function that takes a message as argument and returns a method. This week we're talking about how local state works.

A *local* variable is one that's only available within a particular part of the program; in Scheme this generally means within a particular procedure. We've used local variables before; `let` makes them. A *state* variable is one that remembers its value from one invocation to the next; that's the new part.

First of all let's look at *global* state—that is, let's try to remember some information about a computation but not worry about having separate versions for each object.

```
;;;;;                          In file cs61a/lectures/3.1/count1.scm
(define counter 0)

(define (count)
  (set! counter (+ counter 1))
  counter)

> (count)
1
> (count)
2
```

What's new here is the special form `set!` that allows us to change the value of a variable. This is not like `let`, which creates a temporary, local binding; this makes a permanent change in some variable that must have already existed. The syntax is just like `define` (but not the abbreviation for defining a function): it takes an unevaluated name and an expression whose value provides the new value.

A crucial thing to note about `set!` is that the substitution model no longer works. We can't substitute the value of `counter` wherever we see the name `counter`, or we'll end up with

```
(set! 0 (+ 0 1))
0
```

which doesn't make any sense. From now on we use a model of variables that's more like what you learned in 7th grade, in which a variable is a shoebox in which you can store some value. The difference from the 7th grade version is that we can have several shoeboxes with the same name (the instance variables in the different objects, for example) and we have to worry about how to keep track of that. Section 3.2 of A&S explains the *environment* model that keeps track for us.

Another new thing is that a procedure body can include more than one expression. In functional programming, the expressions don't *do* anything except compute a value, and a function can only return one value, so it doesn't make sense to have more than one expression in it. But when we invoke `set!` there is an *effect* that lasts beyond the computation of that expression, so now it makes sense to have that expression and then another expression that does something else. When a body has more than one expression, the expressions are evaluated from left to right (or top to bottom) and the value returned by the procedure is the value computed by the last expression. All but the last are just *for effect*.

We've seen how to have a global state variable. We'd like to try for *local* state variables. Here's an attempt that doesn't work:

```
;;;;;                          In file cs61a/lectures/3.1/count.lose
(define (count)
  (let ((counter 0))                    > (count)
    (set! counter (+ counter 1))        1
    counter))                           > (count)
                                        1
                                        > (count)
                                        1
```

It was a good idea to use `let`, because that's a way we know to create local variables. But `let` creates a *new* local variable each time we invoke it. Each call to `count` creates a new `counter` variable whose value is 0.

The secret is to find a way to call `let` only once, when we *create* the `count` function, instead of calling `let` every time we *invoke* `count`. Here's how:

```
;;;;;                          In file cs61a/lectures/3.1/count2.scm
(define count
  (let ((result 0))
    (lambda ()
      (set! result (+ result 1))
      result)))
```

Notice that there are no parentheses around the word `count` on the first line! Instead of

```
(define count (lambda () (let ...)))
```

(which is what the earlier version means) we have essentially interchanged the `lambda` and the `let` so that the former is inside the latter:

```
(define count (let ... (lambda () ...)))
```

We'll have to examine the environment model in detail before you can really understand why this works. A handwavy explanation is that the `let` creates a variable that's available to things in the body of the `let`; the `lambda` is in the body of the `let`; and so the variable is available to the function that the `lambda` creates.

The reason we wanted local state variables was so that we could have more than one of them. Let's take that step now. Instead of having a single procedure called `count` that has a single local state variable, we'll write a procedure `make-count` that, each time you call it, makes a new counter.

```
;;;;;                          In file cs61a/lectures/3.1/count3.scm

(define (make-count)                    > (define dracula (make-count))
  (let ((result 0))                     > (dracula)
    (lambda ()                          1
      (set! result (+ result 1))        > (dracula)
      result)))                         2
                                        > (define monte-cristo (make-count))
                                        > (monte-cristo)
                                        1
                                        > (dracula)
                                        3
```

Each of `dracula` and `monte-cristo` is the result of evaluating the expression `(lambda () ...)` to produce a procedure. Each of those procedures has access to its own local state variable called `result`. `Result` is temporary with respect to `make-count` but permanent with respect to `dracula` or `monte-cristo`, because the `let` is inside the `lambda` for the former but outside the `lambda` for the latter.

• Environment model of evaluation.

For now we're just going to introduce the central issues about environments, leaving out a lot of details. You'll get those next time.

The question is, what happens when you invoke a procedure? For example, suppose we've said

```
(define (square x) (* x x))
```

and now we say `(square 7)`; what happens? The substitution model says

1. Substitute the actual argument value(s) for the formal parameter(s) in the body of the function;

2. Evaluate the resulting expression.

In this example, the substitution of 7 for `x` in `(* x x)` gives `(* 7 7)`. In step 2 we evaluate that expression to get the result 49.

We now forget about the substitution model and replace it with the environment model:

1. Create a *frame* with the formal parameter(s) *bound to* the actual argument values;

2. Use this frame to extend the lexical environment;

3. Evaluate the body (without substitution!) in the resulting environment.

A frame is a collection of name-value associations or *bindings*. In our example, the frame has one binding, from `x` to `7`.

Skip step 2 for a moment and think about step 3. The idea is that we are going to evaluate the expression `(* x x)` but we are refining our notion of what it means to evaluate an expression. Expressions are no longer evaluated in a vacuum, but instead, every evaluation must be done with respect to some environment—that is, some collection of bindings between names and values. When we are evaluating `(* x x)` and we see the symbol `x`, we want to be able to look up `x` in our collection of bindings and find the value `7`.

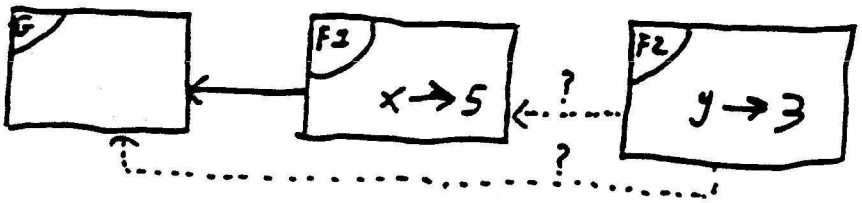
Looking up the value bound to a symbol is something we've done before with global variables. What's new is that instead of one central collection of bindings we now have the possibility of *local* environments. The symbol `x` isn't always `7`, only during this one invocation of `square`. So, step 3 means to evaluate the expression in the way that we've always understood, but looking up names in a particular place.

What's step 2 about? The point is that we can't evaluate `(* x x)` in an environment with nothing but the `x`/`7` binding, because we also have to look up a value for the symbol `*` (namely, the multiplication function). So, we create a new frame in step 1, but that frame isn't an environment by itself. Instead we use the new frame to *extend* an environment that already existed. That's what step 2 says.

Which old environment do we extend? In the `square` example there is only one candidate, the *global* environment. But in more complicated situations there may be several environments available. For example:

```
(define (f x)
  (define (g y)
    (+ x y))
  (g 3))

> (f 5)
```



When we invoke `f`, we create a frame (call it F1) in which `x` is bound to `5`. We use that frame to extend the global environment (call it G), creating a new environment E1. Now we evaluate the body of `f`, which contains the internal definition for `g` and the expression `(g 3)`. To invoke `g`, we create a frame in which `y` is bound to 3. (Call this frame F2.) We are going to use F2 to extend some old environment, but which? G or E1? The body of `g` is the expression `(+ x y)`. To evaluate that, we need an envoironment in which we can

look up all of `+` (in G), `x` (in F1), and `y` (in F2). So we'd better make our new environment by extending E1, not by extending G.

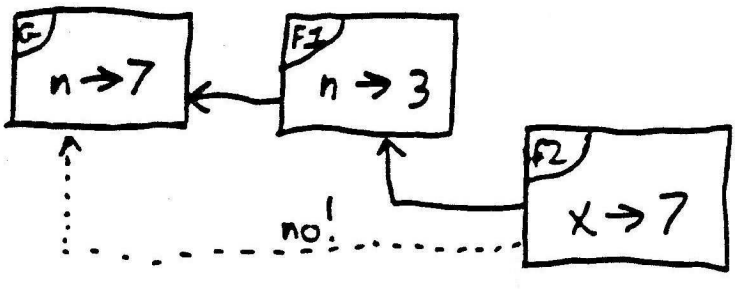
The example with `f` and `g` shows, in a very simple way, why the question of multiple environments comes up. But it still doesn't show us the full range of possible rules for choosing an environment. In the `f` and `g` example, the environment where `g` is defined is the same as the environment from which it's invoked. But that doesn't always have to be true:

```
(define (make-adder n)
  (lambda (x) (+ x n)))

(define 3+ (make-adder 3))

(define n 7)

> (3+ n)
```



When we invoke `make-adder`, we create the environment E1 in which `n` is bound to `3`. In the global environment G, we bind `n` to `7`. When we evaluate the expression `(3+ n)`, what environment are we in? What value does `n` have in this expression? Surely it should have the value `7`, the global value. So we evaluate expressions that you type in G. When we invoke `3+` we create the frame F2 in which `x` is bound to `7`. (Remember, `3+` is the function that was created by the `lambda` inside `make-adder`.

We are going to use F2 to extend some environment, and in the resulting environment we'll evaluate the body of `3+`, namely `(+ x n)`. What value should `n` have in this expression? It had better have the value `3` or we've defeated the purpose of `make-adder`. Therefore, the rule is that we do *not* extend the *current* environment at the time the function is invoked, which would be G in this case. Rather, we extend the environment in which the function was *created*, i.e., the environment in which we evaluated the `lambda` expression that created it. In this case that's E1, the environment that was created for the invocation of `make-adder`.

Scheme's rule, in which the procedure's defining environment is extended, is called *lexical* scope. The other rule, in which the current environment is extended, is called *dynamic* scope. We'll see in project 4 that a language with dynamic scope is possible, but it would have different features from Scheme.

Remember why we needed the environment model: We want to understand local state variables. The mechanism we used to create those variables was

```
(define some-procedure
  (let ((state-var initial-value))
    (lambda (...) ...)))
```

Roughly speaking, the `let` creates a frame with a binding for `state-var`. Within that environment, we evaluate the `lambda`. This creates a procedure within the scope of that binding. Every time that procedure is invoked, the environment where it was created—that is, the environment with `state-var` bound—is extended to form the new environment in which the body is evaluated. These new environments come and go, but the state variable isn't part of the new frames; it's part of the frame in which the procedure was defined. That's why it sticks around.

• Here are the complete rules for the environment model:

Every expression is either an atom or a list.

At any time there is a *current frame*, initially the global frame.

I. Atomic expressions.

A. Numbers, strings, `#T`, and `#F` are self-evaluating.

B. If the expression is a symbol, find the *first available* binding. (That is, look in the current frame; if not found there, look in the frame "behind" the current frame; and so on until the global frame is reached.)

II. Compound expressions (lists).

If the car of the expression is a symbol that names a special form, then follow its rules (II.B below). Otherwise the expression is a procedure invocation.

A. Procedure invocation.

Step 1: Evaluate all the subexpressions (using these same rules).

Step 2: Apply the procedure (the value of the first subexpression) to the arguments (the values of the other subexpressions).

(a) If the procedure is compound (user-defined):

a1: Create a frame with the formal parameters of the procedure bound to the actual argument values.

a2: Extend the procedure's defining environment with this new frame.

a3: Evaluate the procedure body, using the new frame as the current frame.

*** ONLY COMPOUND PROCEDURE INVOCATION CREATES A FRAME ***

(b) If the procedure is primitive:

Apply it by magic.

B. Special forms.

1. `Lambda` creates a procedure. The left circle points to the text of the `lambda` expression; the right circle points to the defining environment, i.e., to the current environment at the time the `lambda` is seen.

*** ONLY `LAMBDA` CREATES A PROCEDURE ***

2. `Define` adds a *new* binding to the *current frame*.

3. `Set!` changes the *first available* binding (see I.B for the definition of "first available").

4. `Let` = `lambda` (II.B.1) + invocation (II.A)

5. `(define (...) ...)` = `lambda` (II.B.1) + `define` (II.B.2)

6. Other special forms follow their own rules (`cond`, `if`).

• Environments and OOP.

Class and instance variables are both local state variables, but in different environments:

```
;;;;;                          In file cs61a/lectures/3.2/count4.scm
(define make-count
  (let ((glob 0))
    (lambda ()
      (let ((loc 0))
        (lambda ()
          (set! loc (+ loc 1))
          (set! glob (+ glob 1))
          (list loc glob))))))
```

The class variable `glob` is created in an environment that surrounds the creation of the outer `lambda`, which represents the entire class. The instance variable `loc` is created in an environment that's inside the class `lambda`, but outside the second `lambda` that represents an instance of the class.

The example above shows how environments support state variables in OOP, but it's simplified in that the instance is not a message-passing dispatch procedure. Here's a slightly more realistic version:

```
;;;;;                          In file cs61a/lectures/3.2/count5.scm
(define make-count
  (let ((glob 0))
    (lambda ()
      (let ((loc 0))
        (lambda (msg)
          (cond ((eq? msg 'local)
                 (lambda ()
                   (set! loc (+ loc 1))
                   loc))
                ((eq? msg 'global)
                 (lambda ()
                   (set! glob (+ glob 1))
                   glob))
                (else (error "No such method" msg)) ))))))
```

The structure of alternating `let`s and `lambda`s is the same, but the inner `lambda` now generates a dispatch procedure. Here's how we say the same thing in OOP notation:

```
;;;;;                          In file cs61a/lectures/3.2/count6.scm
(define-class (count)
  (class-vars (glob 0))
  (instance-vars (loc 0))
  (method (local)
    (set! loc (+ loc 1))
    loc)
  (method (global)
    (set! glob (+ glob 1))
    glob))
```