ENVIRONMENTS AND LOCAL STATE 14

GEORGE WANG
gswang.cs61a@gmail.com
Department of Electrical Engineering and Computer Sciences
University of California, Berkeley

July 15, 2010

1 Review

1.1 The Golden Rules

1. *Calling a lambda procedure makes frames*, pointing to the right bubble of the `lambda`.
2. *Making a lambda procedure makes 2 bubbles*, the left holding information, and the right pointing to the current frame.

2 The Environment Model

2.1 Exercise:

Draw the Environment!

```
(define (f x)
  (define (g y)
    (+ x y))
    (g 3))
> (f 5)
```

2.2 Exercise: `make-adder`

Draw the Environment!

```
(define (make-adder n)
  (lambda (x) (+ x n)))
(define 3+ (make-adder 3))
```

```
(define n 7)
> (3+ n)
```

Scheme's rule, in which the procedure's defining environment (the right bubble) is extended, is called LEXICAL SCOPE. The other rule, in which the current environment is extended, is called DYNAMIC SCOPE. 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 variable. 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 variable 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.

3 Converting Above-the-Line OOP

3.1 Exercise: `make-count`

Draw the Environment Diagram:

```
;;;;; 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 variable `glob` here is a class variable if we think of this procedure as something that creates `count` objects. In that case, the variable loc is created in an environment that is *inside* the class `lambda`, but *outside* the `lambda` that represents an **instance**.

This example should show how environments give us the tools we need in order to create the critical section of local state in our three pillars for object oriented programming. Now, you know how to implement each of the three sections: Message Passing, Inheritance, and Local State. Now remember, the difference between this and what we know to be OOP is the idea of all the messages to be passed. But it turns out this is a easy fix:

```
;;;;; 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)) ))))))
```

Note, that the `let`s and the `lambda`s are precisely the same, except that in the innermost `lambda`, we instead have a dispatch procedure. Let's look at the same thing in OOP:

```
;;;;; 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))
```

4 Exercises[2]

Draw the environment diagrams!

4.1 De-Sugaring Lets:

```
(let ((a 3)) (+ 5 a))
(let ((a 3)) (lambda (x) (+ x a)))
((let ((a 3)) (lambda (x) (+ x a))) 5)
```

4.2 Swapping Values:

```
(define (make-prev last-value)
  (lambda (new)
    (define temp last-value)
    (set! last-value new) temp))
(define prev (make-prev '*first-call*))
(prev 'a)
```

4.3 Setter

```
(define x 'x)
(define (changer x y)
  (y)
  (y)
  x)
(changer 16
         (lambda () (set! x (* x x))))
```

4.4 List Recursion with Polymorphism:

Fill in the blanks to code map without using `if`!

[2] Courtesy of Carolen

```
(define-class (pair a b)
  (method (map fn) ____________________)
(define-class (empty-list)
  (parent pair '() '())
  (method (map fn) _____________________)
```

4.5 I'm Confused...

```
(define secret 42)
(define change
  (let ((fn
         (let ((secret 23))
              (lambda (x) (set! secret x))))
        (x 12))
   (lambda (secret) (fn secret))))
(change 92)
```