Lecture 16 & 17: Local State and Environments

Summer 2006
July 24, 2006
Instructor: Ana Ramírez Chang

A little about me
• Ana Ramírez Chang
• 4th year PhD student in Computer Science
  – Work with Professor John Canny
  – Work with speech user interfaces and speech recognition
• B.S. in Computer Science from Carnegie Mellon University
• TAed 61a, 164 and Multimedia Information (in the I-School)
• From Colorado

Contact information
• Email: anar@cs.berkeley.edu
• Office: 360 Hearst Mining Memorial Building
  – Enter HMMB through back door by Cory
  – Go up one flight of stairs
  – 2 sets of numbering systems, old and new, only pay attention to the new one.
• Office Hours
  – Monday 9:45 – 10:45 am
  – Tuesday 3:00 – 4:00

Administrative stuff
• Regrade request deadline
  – hw1a, hw1b, hw2a, hw2b, proj1, mt1
  – Passed today at 11 am
• Midterm 2 on Friday 4 – 6pm, 10 Evans
  – Object Oriented Programming
  – Data Directed Programming, Message Passing, Tagged Data
  – Trees
  – Pairs, Lists, Deep lists
  – Scheme-1

Project 3
• Part I: due Thursday night
• Part II: due next Thursday night (Aug 3rd)
• Part I is easier than part II, so try and finish part I early and get an early start on part II.
• Talk to your TA if you don’t have a partner
• Homework 5
  – 5a assigned today
  – 5b assigned Wednesday

This week
• Monday & Tuesday
  – How local state works
  – Reading: (3.1 - 3.2)
• Wednesday & Thursday
  – Mutable data
  – Reading: (3.3.1 - 3.3.3)
Review - state variables

- **state variable** - a variable that remembers its value from one invocation to the next with **global state**

```lisp
(counter: int)
(define counter 0)

(count: () → int)
(define (count)
  (set! counter (+ counter 1))
  counter)
```

STk> (count)
1
STk> (count)
2

with **local state**

```lisp
(count: () → int)
(define count
  (let ((result 0))
    (lambda ()
      (set! result (+ result 1))
      result)))
```

STk> (count)
1
STk> (count)
2

Doesn't Work

```lisp
(count: () → int)
(define (count)
  (let ((counter 0))
    (set! counter (+ counter 1))
    counter))
```

STk> (count)
1
STk> (count)
1

Doesn't Work:
```
define count (lambda () (let ...))
```

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

Review - state variables

make-count: each time you call it, makes a new counter

```lisp
(make-count: () → ( () → int ))
(define make-count
  (lambda ()
    (let ((counter 0))
      (lambda ()
        (set! counter (+ counter 1))
        counter))))
```

STk> (define dracula (make-count))
STk> (dracula)
1
STk> (define monte-cristo (make-count))
STk> (dracula)
1
STk> (monte-cristo)
2

STk> (define (square x) (* x x))
```

⇒
49

Environment diagrams

- **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 ...

Evaluation models

- **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.

```lisp
(square:int → int)
```

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

⇒
49

- **Substitution model**
  1. Substitute the actual argument value(s) for the formal parameter(s) in the body of the function;
  2. Evaluate the resulting expression.

```
substitution of 7 for x in (* x x) gives (* 7 7)
```
**Environments and OOP**

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

  ```scheme
  (define make-count
    (let ((glob 0))
      (lambda ()
        (let ((loc 0))
          (lambda ()
            (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 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.

**Back to make-count**

- The structure of alternating lets and lambdas is the same, but the inner `lambda` now generates a dispatch procedure.
Environments and OOP

• Here’s how we say the same thing in OOP notation:
  
  ```lisp
  (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))
  ```