Mutation Revisited

- ⟨⟨PLAI Chapter 14⟩⟩
- We saw in homework 6 that as a side effect of our `lookup` implementation it was easy to implement

```
1   {set! <id> <FLANG>}
```

via

```
2   (set-box! (lookup <id> <env>)
              (eval <FLANG> <env>))
```

Return value of set!

- What should be the value of a 'set!' expression? There are three obvious choices:
  1. return some bogus value,
  2. return the value that was assigned,
  3. return the value that was previously in the box.
- Option 2 (return new value) is useful in C

```
x = y = 0

while (c = getchar()) { ... }
```

- Option 3 (return old value) is also potentially useful. E.g.:

4

```
x = y = x;
```

- the expression ”$x = x + 1$” has the meaning of C’s ”$++x$” when option (2) is used, and ”$x++$” when option (3) is used.)
- Racket chooses the first option, separating mutation from access.
- The programmer must explicitely choose what value to return.
- This also means that some form of sequencing is needed (e.g. Racket `begin` forms)

Mutation is a big change

- The code change is small, but
- the (target) language change is large
- bindings are not fixed; substitution no longer makes sense as a semantic model.

State and Environments

- There can be only one:

```
(define counter
  (let ([the-box (box 0)])
    (lambda ()
      (begin
        (set-box! the-box (+ 1 (unbox
            the-box)))
        (unbox the-box)))))
```

- Or not:

```
(define (make-counter)
  (let ([counter (box 0)])
    (lambda ()
      (begin
        (set-box! counter (+ 1 (unbox
            counter)))
        (unbox counter)))))

(define foo (make-counter))
(define bar (make-counter))
```

7

- We don't need the pass by reference semantics of boxes here:

```
(define (make-counter)
  (let ([counter 0])
    (lambda ()
      (begin (set! counter (+ 1 counter))
         counter))))

(define foo (make-counter))
(define bar (make-counter))
```

- Mutation and lexical scope are working **together** here

```
(define counter
  (let ([counter 0])
    (lambda ()
      (begin
        (set! counter (+ 1 counter))
        (if (zero? (modulo counter 4))
            'tock 'tick)))))
```

- Internal state is invisible (can't even tell it is integer).


```
(define counter
  (let [(state 'ichi)]
    (lambda ()
      (case state
        [(ichi) (begin (set! state 'ni) 'tick)]
        [(ni)   (begin (set! state 'san)
           'tock)]
        [(san)  (begin (set! state 'shi)
           'tick)]
        [(shi)  (begin (set! state 'ichi)
           'boom)]))))
```

Implementing Objects with State

- Mutable state encapsulated by closures gives us a natural implementation of objects

```
(define (make-point x y)
  (let ([the-x x]
        [the-y y])
    (lambda (msg)
      (case msg
        [(getx) the-x]
        [(gety) the-y]
        [(incx) (set! the-x (add1 the-x))]))))

(define P (make-point 0 0))
(P 'incx)
(P 'getx)
```

- How do we make this typecheck


```
(define (make-point x y)
  (let ([the-x x]
        [the-y y])
    (lambda (msg)
      (case msg
        [(getx) the-x]
        [(gety) the-y]
        ;[(incx) ....]
```

- Side effects can easily be incorporated in operations

```
(define (make-point x y)
  (let ([the-x x]
        [the-y y])
    (lambda (msg)
      (case msg
        [(getx) the-x]
        [(gety) the-y]
        [(incx) (begin (set! the-x (add1
            the-x))
                       (update-screen))]))))
```

- A simple imitation of inheritance can be achieved using delegation to an instance of the super-class:

```
(define (make-colored-point x y color)
  (let ([p (make-point x y)])
    (lambda (msg)
      (case msg
        [(getcolor) color]
        [else (p msg)]))))

(define P (make-colored-point 0 0 'red))
(P 'incx)
(P 'getcolor)
```

Preprocessing yields something like the racket default class system

14

```
(define point%
  (class object%
    (init x y)
    (super-new)
    (define current-x x)
    (define current-y y)
    (define/public (getx) current-x)
    (define/public (gety) current-x)
    (define/public (incx)
             (set! current-x
                   (add1 current-x)))))

(define P (new point% [x 10] [y 10]))
(send P incx)
(send P getx)
```