CSSE 304 Day 9

1. What are some characteristics of “objects” in the traditional OOP sense?

2. Consider this attempt at writing a constructor for "stack objects". How does it work? What needs to be fixed?

```
(define make-stack
 (lambda ()
  (lambda (msg  . args )
    (let ([stk '()])
      (case msg
        [(empty?) (null? stk)]
        [(push)
         (set! stk (cons (car args)
                         stk))]
        [(pop)
         (let ([top (car stk)])
           (set! stk (cdr stk))
           top)]
        [else
         (error
          'stack
          "illegal stack message: ~a"
          msg)]))))))
```

```
> (define s1 (make-stack))
> (define s2 (make-stack)))
> (s1 'push 'a)
> (s2 'push 'z)
> (s1 'push 'b)
> (s1 'pop)
b
(s1 'empty?)
#f
```

```
> (s2 'push (s1 'pop))
> (s1 'empty?)
#t
> (s2 'pop)
a
> (s2 'pop)
z
> (s2 'pop)
Exception in car: () is not a
pair
```

`case-lambda` provides a more specific interface for writing variable-arity procedures. Examples are in the slides.

Your notes and questions:

PRACTICE PROBLEMS (we may not get to these until Day 11 (Day 10 is procedural abstraction).

3. Consider the following Scheme transcript (from which I have removed part of the definition) Fill in the missing part. You might want to try it in Scheme to see if your code works.

```
> (define a                                  )  ;fill it in!
> (list ((car a)  4 7)
        ((cadr a) 4 7))
(11 28)
```

4. What is the result of entering each of the following lines of code in the given order the following code? Before you enter the last two lines, draw the box-and-pointer diagrams, figure out what they will print, then try running them in Scheme.

```
(define a '(1 2 3 . 4))
(define b (cons (car a) (cdr a)))
(set-car! a 5)
(set-car! (cdr a) 6)
a
b
```

Draw (above) a box-and-pointer diagram showing the final states of `a` and `b`. Why does the code output what it does?

5. `remove` may do less than you think!

Can you figure out why the last output is what it is?

Draw box-and-pointer diagrams

```
> (define a '(1 2 3 4 5 6))
> (define b (remove 2 a))
> (define c (remove 3 a))
> (list a b c)
((1 2 3 4 5 6) (1 3 4 5 6) (1 2 4 5 6))
> (set-car! (cddr b) 10)
> (list a b c)
((1 2 3 10 5 6) (1 3 10 5 6) (1 2 10 5 6))
>
```

6. In-class exercise (if there is time) Do this with another student or two.

write `reverse` and `reverse!` Procedures. For `reverse!`, you will need to use `set-cdr!`
Both should be O(n). These transcripts might help you to distinguish between them.

```
> (reverse '())
()
> (reverse '(a b c))
(c b a)
> (let* ([x '(a b c)]
         [y (cdr x)]
         [z (reverse x)])
    (list x z (eq? y (cdr z))))
((a b c) (c b a) #f)
```

```
> (reverse! '())
()
> (reverse! '(a b c))
(c b a)
> (define L '(a b c d))
> (reverse! L)
(d c b a)
> L
(a)
> (let* ([x '(a b c)]
         [y (cdr x)]
         [z (reverse! x)])
    (list x z (eq? y (cdr z))))
((a) (c b a) #t)
```