Week 4 – 04-02-12
Midterm 1 Review

Scheme Questions
What will Scheme print in response to the following expressions? If an expression produces an error or runs forever without producing a result, just say “error”; If the value of an expression is a procedure, just say “procedure.”

(word ‘(+ 2 3) (+ 2 3)) ***ERROR***

((lambda (x y z) (* 5 y)) 3 4 7) ***20***

((if 3 - *) 23 2) ***21***

(lambda (x) (/ x 0)) ***procedure***

(butfirst ‘(help)) ***()***

```
(let ((+ -))                                  6
   (+ 8 2))
```

(every – (filter number? ‘(the 1 after 909))) ***(-1 -909)***

```
(let ((a 2) (b (+ a 3)))                      ERROR
   (word a b))
```

((lambda (a b c) (b c a)) 1 + 4) ***5***

(se (‘tell ‘me ‘why)) ***ERROR***

(every pigl (se ‘() (word 61 ‘a) (se ‘is ‘a) ‘great ‘(class)))

(a61ay isay aay eatgray assclay)

```
 (let ((a (square 2))
       (b (+ 3 4)))
   (let ((c (+ a (let ((d 3))
                  (+ b d))))
          (e 14))
      (* (+ a b) (- e c))))
```

```
( (lambda (a b)
     ( (lambda (c e)
           (* (+ a b) (- e c)))
        (+ a ( (lambda (d) (+ b d)) 3 )
        14 )
  (square 2)
  (+ 3 4) )
```

Higher Order Function

Write a procedure called **make-manip** which takes two procedures, *pred* and *manip* and returns a manipulator! A manipulator is a procedure that takes a sentence as its argument and returns a sentence in which every element from which *pred* returns true is manipulated with *manip*, and all of the other elements are the same. For example:

```
>((make-manip odd? 1+) ‘(3 6 9 12))
    (4 6 10 12)
```

No Helper functions!

Write one version using HOF and no explicit recursion.

```
(define (make-manip pred funct)
  (lambda (x)
       (every (lambda (y) (if (pred y) (funct y) y)) x)) )
```

Write another using no HOF.

```
(define (make-manip pred funct)
  (lambda (x)
       (cond ((empty? x) '()
            ((pred (first x))
             (se (funct (first x)) ((make-manip pred funct) (bf x))))
            (else (se (first x) ((make-manip pred funct) (bf x))))))))
```

Normal vs. Applicative Order
True or False:

```
(define (f x) (* x x x))
```

Evaluating (f (g y)) evaluates (g y) more often in applicative order than in normal order.

FALSE

Suppose you were given the following definitions:

```
(define (double x) (+ x x))
(define (foo x y z) (+ x y z))
(define (bar x y z k) k)
```

How many times is + called for

```
        (foo (double (+ 1 1)) (double (+ 1 1)) (+ 1 1))
```

…under normal order? ***8***

…under applicative order? ***6***

How many times is + called for

```
        (bar (double (+ 1 1)) (double (+ 1 1)) (+ 1 1) 1)
```

…under normal order? ***0***

…under applicative order? ***5***

Big O
True or False:
If foo is Theta(n) and bar is Theta(n^2), then you can always compute (foo 1000) faster than (bar 1000) on the same computer.

TRUE

Given These Definitions:

```
 (define (f x)
   (if (< x 0)
       1
       (f (- x 3))))

(define (g y)
   (if (< y 104)
       0
```

```
      (* (f y) (f (- y 4)))))

(define (h z)
   (if (< z 4)
       0
       (+ (h (- z 2))
           (h (- z 1)))))
```

State whether or not these statements are true or false:

FALSE h generates an iterative process (i.e. uses Θ(1) space)

TRUE f is Θ(x).

FALSE h is Θ(z2)

TRUE f and g have the same order of growth

FALSE g and h have the same order of growth

Project Questions
Write a strategy four-cards that hits only if a player has fewer than four cards

```
(define (four-cards customer-hand-so-far dealer-up-card)
   (< (count customer-hand-so-far) 4))
```

Write a procedure n-cards that takes an argument n and returns a strategy that hits only if a player has fewer than n cards

```
(define (n-cards n)
  (lambda (customer-hand-so-far dealer-up-card)
       (< (count customer-hand-so-far) n)))
```

Recursive vs. Iterative

Write a procedure **(insert value insert-before sent)** that’ll return a sentence with ‘value’ inserted in the list (counting from 1):

```
(insert ‘a 3 ‘(1 2 3 4)) → (1 2 a 3 4)
(insert x ‘a ‘(a b c d)) → (x a b c d)
(insert a ‘4 ‘(1 2 3 4)) → (1 2 3 a 4)
```

You can assume that **insert-before** will always be in the sentence. You may not use any mutators (if you know of them)

Write a version using a recursive process....

```
(define (insert val insert-before  sent)
  (if (equal? insert-before (first sent))
     (se val sent)
     (se (first sent)
          (insert val insert-before (bf sent)))))
```

and another with an iterative process.

```
(define (insert val insert-before sent)
  (define (helper sent-so-far sent)
    (if (equal? insert-before (first sent))
       (se sent-so-far (first val sent))
        (helper (se sent-so-far (first sent))
              (bf sent))) )
  (helper '() sent) )
```

OR

```
(define (insert val insert-before sent)
  (define (helper sent-so-far sent)
    (cond ((empty? sent) sent-so-far)
          ((equal? insert-before (first sent))
           (helper (se sent-so-far (se val sent))
                 '()))
          (else (helper (se sent-so-far (first sent))
                     (bf sent)))) )
  (helper '() sent) )
```

Programming Methodology
Greg wanted to write a procedure that would split a non-empty word into a sentence of consecutive, identical letters as follows:

```
(split ‘(aaabbcdddaa) → (aaa bb c ddd aa)
(split ‘abababab) → (a b a b a b a b)
(split ‘aaa) → (aaa)
(split ‘a) → (a)
```

Here’s what he wrote:

```
1:  (define (split wd)
2:      (split-help (first wd) (bf wd)))
3:
4:  (define (split-help cur wd)
5:      (cond ((empty? wd) (se))
6:              ((equal? cur (first wd))
7:               (split-help (word cur (first wd)) (bf wd)))
8:              (else
9:                (se cur (split-help (first wd) (bf wd))))))
```

There are two bugs.

Part A:
What does (split ‘abc) return? ***(a b)***
On which line number is the bug that causes this error? Line ***5***
What should the line say?
(cond ((empty? wd) (se cur)) …

Part B:
Where’s the other bug? Line ***6***
What should the line say?
((equal? (first cur) (first wd)) …