Environment Diagrams

1. Why do set!, when we can always just redefine a variable using a define statement? Instead of doing (set! x 3), why don’t we just do (define x 3) again? I propose the following alternative implementation of counter.

The Old Way

```
(define count
  (let ((current 0))
    (lambda()
      (set! current (+ current))
      current))

(count) ==> 1
(count) ==> 2
```

My Brilliant New Way

```
(define count
  (let ((current 0))
    (lambda ()
      (define current
        (+ current 1))
      current))
```

How dumb am I? What happens when I use my new brilliant implementation?

My brilliant implementation will always return 1. define will always create a new binding in the frame, and each procedure call will always create a new frame.

2. Define a procedure f so that, given the procedure call (+ (f 0) (f 1)) if STk evaluates from left to right, it returns 0, and if STk evaluates from right to left, it returns 1.

```
(define f
  (let ((first-call #t))
    (lambda(x)
      (cond (first-call (set! first-call #f) x)
            (else 0)))))
```

3. Define a procedure fib so that, every time it is called, it returns the next Fibonacci number, starting from 1:

```
(fib) ==> 1; (fib) ==> 2; (fib) ==> 3; (fib) ==> 5; (fib) ==> 8, etc.

(define fib
  (let ((a 0) (b 1))
    (lambda ()
      (let ((old-a a))
        (set! a b)
        (set! b (+ a old-a))
        b))))
```
4. Draw environment diagrams for each of the following:

\[
\text{(define } (f + -) (+ ((\lambda(-)(- 3 5)) -) 10)) \]
\[
(f - +) \]

-2

\[
\text{(define } (hmm n) (\lambda(x) (+ x y n))) \]
\[
\text{(define } (uhh y) \]
\[
\text{    (define hmm-y (hmm y))} \]
\[
\text{    (hmm-y 2))} \]
\[
(uhh 42) \]

\text{error: undefined } y \]

\[
\text{(define answer 0)} \]
\[
\text{(define } (square f x) \]
\[
\text{    (let } ((answer 0)) \]
\[
\text{        (f x)} \]
\[
\text{        answer))} \]
\[
\text{(square } (\lambda(n) \text{ (set! answer } (* n n))) 3) \]
(define a 3)
((lambda(a)
    ((lambda(a) (a))
     (lambda() (set! a 'myxomatosis)))
   a)
  (* a a))

myxomatosis

(define a 'scatterbrain)
((lambda(a b) (b) a)
 a
  (let ((b 'cuttooth))
   (lambda() (set! a b)))
 a)

scatterbrain
cuttooth
(define (slow-op-maker op)
  (let ((old-result #f))
    (lambda(x)
      (let ((return old-result))
        (set! old-result (op x))
        return))))

(define slow-sqr (slow-op-maker square))
(slow-sqr 3)
(slow-sqr 5)
(define slow-cube (slow-op-maker cube))
(slow-cube 3)
(slow-cube 5)

#f, 9, #f, 27

(define slow-op-maker
  (let ((old-result #f))
    (lambda(op)
      (lambda(x)
        (let ((return old-result))
          (set! old-result (op x))
          return))))

(define slow-sqr (slow-op-maker square))
(slow-sqr 3)
(slow-sqr 5)
(define slow-cube (slow-op-maker cube))
(slow-cube 3)

#f, 9, 25