CS61A Notes 8 – Attack of the Environmentalists [Solutions v1.0]

Assigning Things To Things And Stuff

1. **Personally – and don't let this leave the room – I think `set!` is useless. I mean, 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`, similar to the one in class:**

<u>The Old Way</u>

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

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

<u>My Brilliant New Way</u>

```
(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. This is because, every time (count) is called, I redefine current to be (+ current 1), but I don't remember that for the next call. That is, after I exit out of the procedure call, the new binding for current is lost.

2. **Consider these definitions:**

```
(define x 3)
(define (z) (set! x 5) x)
```

what would `(list (z) x)` return?

Depends! If we evaluate left to right, then it returns (5 5). If we evaluate right to left, it returns (5 3). Now do you believe me when I say imperative programming is more dangerous?

3. **(SICP ex. 3.8) Keeping number 2 in mind, 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)))))
```

4. **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))))
```

1. **(define (f + -) (+ ((lambda(-) (- 3 5)) -) 10))**
   (f - +)
   -2

2. **(define (hmm n) (lambda(x) (+ x y n)))**
   (define (uhh y)
   (define hmm-y (hmm y))
   (hmm-y 2))
   (uhh 42)
   error: undefined y

3. **(define answer 0)**
   (define (square f x)
   (let ((answer 0))
   (f x)
   answer))
   (square (lambda(n) (set! answer (* n n))) 3)
   0
   (answer becomes 3)

4. **(define a 3)**
   ((lambda(a)
   ((lambda(a) (a))
   (lambda() (set! a ‘myxomatosis)))
   a)
   (* a a))
   myxomatosis

5. **(define a ‘scatterbrain)**
   ((lambda(a b) (b) a)
   a
   (let ((b ‘cuttooth))
   (lambda() (set! a b))))
   a
   scatterbrain; a becomes cuttooth

6. **(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