# Homework 5

Jason Hemann

December 29, 2017

## 1 Preamble

- Do consult class notes, online lecture notes, and test cases when completing this assignment.
- Zip your `main.ss` and `pmatch`, using a command like the following: (`zip <username>-hw5.zip main.ss pmatch.ss`)
- Load `pmatch.ss` in the top of your file: `(load "pmatch.ss")`
- You should be writing recursive functions over our extended language of `lambda`-calculus expressions.
- Upload the `.zip` file to the PLC grading server
- Your interpreters must work for at least these test cases. Of course, these tests are not exhaustive; you should use your own tests as well.
- You should use the following interpreter as your initial starting point: this interpreter is representation independent w.r.t. environments, and uses a functional representation of closures. The language of this interpreter is familiar.

```
(define value-of
  (lambda (exp env)
    (pmatch exp
      [,c (guard (or (number? c) (boolean? c))) c]
      [(zero? ,n) (zero? (value-of n env))]
      [(sub1 ,n) (sub1 (value-of n env))]
      [(* ,ne1 ,ne2) (* (value-of ne1 env) (value-of ne2 env))]
      [(if ,test ,conseq ,alt) (if (value-of test env)
                                   (value-of conseq env)
                                   (value-of alt env))]
      [,y (guard (symbol? y)) (apply-env env y)]
      [(lambda (,x) ,body) (lambda (a) (value-of body (extend-env x a env)))]
      [(,rator ,rand) ((value-of rator env)
                       (value-of rand env))])))
```

Succinctly, starting with this template, you should build the following, and pass the below tests:

| Convention | Interpreter Name |
|---|---|
| call-by-value | value-of-cbv |
| call-by-reference | value-of-cbr |
| call-by-name | value-of-cbname |
| call-by-need | value-of-cbneed |

## 2 Part 1 : Call-by-value, with assignments

Copy and rename the interpreter above, add forms `set!` and `begin2` to your interpreter, and ensure that your interpreter implements a **call-by-value** parameter-passing convention.

```
> (value-of-cbv
    '((lambda (a)
        ((lambda (p) (begin2 (p a) a)) (lambda (x) (set! x 4))))
      3)
    (empty-env))
3
> ;; ...but returns 55 under CBV!  You can change the "begin2" to
  ;; "begin" and evaluate this in the Scheme REPL as evidence that
  ;; Scheme uses CBV.
(value-of-cbv
  '((lambda (f)
      ((lambda (g) ((lambda (z) (begin2 (g z) z)) 55))
        (lambda (y) (f y))))
     (lambda (x) (set! x 44)))
  (empty-env))
      55
> ;; ...but returns 33 under CBV.
(value-of-cbv
  '((lambda (swap)
      ((lambda (a) ((lambda (b) (begin2 ((swap a) b) a)) 44)) 33))
     (lambda (x)
       (lambda (y)
         ((lambda (temp) (begin2 (set! x y) (set! y temp))) x))))
  (empty-env))
33
```

## 3 Part 2 : Call-by-reference, with assignments

Copy and rename your `value-of-cbv` interpreter with forms `set!` and `begin2`, change your interpreter to implement a **call-by-reference** parameter-passing convention.

```
> ;; Making sure set! works
(value-of-cbr
  '((lambda (x) (begin2 (set! x #t) (if x 3 5))) #f)
  (empty-env))
```

```
3
> ;; Returns 4 under CBR...
(value-of-cbr
  '((lambda (a)
      ((lambda (p) (begin2 (p a) a)) (lambda (x) (set! x 4))))
     3)
  (empty-env))
4
> ;; returns 44 under CBR...
(value-of-cbr
  '((lambda (f)
      ((lambda (g) ((lambda (z) (begin2 (g z) z)) 55))
        (lambda (y) (f y))))
     (lambda (x) (set! x 44)))
  (empty-env))
44
;; Returns 44 under CBR...
(value-of-cbr
  '((lambda (swap)
      ((lambda (a) ((lambda (b) (begin2 ((swap a) b) a)) 44)) 33))
     (lambda (x)
       (lambda (y)
         ((lambda (temp) (begin2 (set! x y) (set! y temp))) x))))
  (empty-env))
44
> (value-of-cbr
    '((lambda (swap)
        (((lambda (a) (lambda (b) (begin2 ((swap a) b) (sub1 a))))
           5)
          3))
       (lambda (x)
         (lambda (y)
           ((lambda (temp) (begin2 (set! y x) (set! x temp))) y))))
    (empty-env))
4
> ;; Make sure that changing a value that's referenced by an
  ;; already-created closure still shows the updated value.
(value-of-cbr
  '((lambda (x)
      ((lambda (func) (begin2 (set! x 3) (func 1)))
        (lambda (_) x)))
     5)
  (empty-env))
3
```

# 4 Part 3 : Call-by-name

Copy and rename your `value-of-cbr` interpreter, and remove forms `set!` and `begin2`. Add the `random` form. Then change your interpreter to implement a **call-by-name** parameter-passing convention.

```
> (let ([random-sieve '((lambda (n)
                          (if (zero? n)
                              (if (zero? n)
                                  (if (zero? n)
                                      (if (zero? n)
                                          (if (zero? n)
                                              (if (zero? n)
                                                  (if (zero? n) #t #f)
                                                  #f)
                                              #f)
                                          #f)
                                      #f)
                                  (if (zero? n)
                                      #f
                                      (if (zero? n)
                                          #f
                                          (if (zero? n)
                                              #f
                                              (if (zero? n)
                                                  #f
                                                  (if (zero? n)
                                                      #f
                                                      (if (zero? n) #f #t))))))))
                          (random 2))])
    (value-of-cbname random-sieve (empty-env)))
#f
> (value-of-cbname
    '((lambda (z) 100) ((lambda (x) (x x)) (lambda (x) (x x))))
    (empty-env))
100
> (value-of-cbname
    '(((lambda (pi)
         ((lambda (omega) (omega omega))
           (lambda (beta) (pi (beta beta)))))
       (lambda (fact)
         (lambda (n) (if (zero? n) 1 (* n (fact (sub1 n)))))))
      5)
    (empty-env))
120
```

## 5 Part 3 : Call-by-need

Copy and rename your `value-of-cbname` interpreter. Then change your interpreter to implement a **call-by-need** parameter-passing convention.

```
> (value-of-cbneed
    '((lambda (z) 100) ((lambda (x) (x x)) (lambda (x) (x x))))
    (empty-env))
100
> (let ([random-sieve '((lambda (n)
                          (if (zero? n)
                              (if (zero? n)
                                  (if (zero? n)
                                      (if (zero? n)
                                          (if (zero? n)
                                              (if (zero? n)
                                                  (if (zero? n) #t #f)
                                                  #f)
                                              #f)
                                          #f)
                                      #f)
                                  (if (zero? n)
                                      #f
                                      (if (zero? n)
                                          #f
                                          (if (zero? n)
                                              #f
                                              (if (zero? n)
                                                  #f
                                                  (if (zero? n)
                                                      #f
                                                      (if (zero? n) #f #t))))))))
                        (random 2))])
    (value-of-cbneed random-sieve (empty-env)))
#t
> (value-of-cbneed
    '((lambda (f) ((lambda (x) (x x)) (lambda (x) (f (x x)))))
       (lambda (!)
         (lambda (n) (if (zero? n) 1 (* n (! (sub1 n)))))))
    (empty-env))
5
```