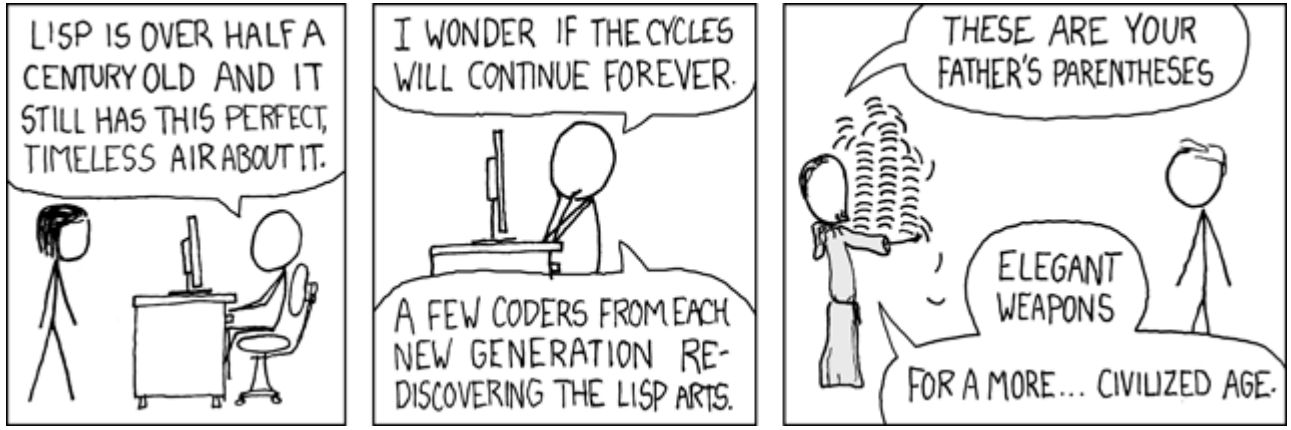
Login

Assignment 5: Parameter-Passing Conventions



Note

You may find the following resources to be useful as you work on this assignment:

* An in-depth look at parameter-passing variations from notes on Sep 29, 2011 and Feb 6, 2014.

You should be able to use the a5 student tests to test your assignment; as usual these should not be considered exhaustive tests.

Assignment

Part 1

Below is an interpreter that is complete except for its five helpers `empty-env`, `extend-env`, `apply-env`, `make-closure`, and `apply-closure`. This interpreter should look quite familiar to you, except that it adds two new forms to our language, `begin2` and `random`.

```
(define value-of
  (lambda (exp env)
    (match exp
      [`,b #:when (boolean? b) b]
      [`,n #:when (number? n)  n]
      [`(zero? ,n) (zero? (value-of n env))]
```

```
      [`(sub1 ,n) (sub1 (value-of n env))]
      [`(* ,n1 ,n2) (* (value-of n1 env) (value-of n2 env))]
      [`(if ,test ,conseq ,alt) (if (value-of test env)
                                    (value-of conseq env)
                                    (value-of alt env))]
      [`(begin2 ,e1 ,e2) (begin (value-of e1 env) (value-of e2 env))]
      [`(random ,n) (random (value-of n env))]
      [`,y #:when (symbol? y) (apply-env env y)]
      [`(lambda (,x) ,body) (make-closure x body env)]
      [`(,rator ,rand) (apply-closure (value-of rator env)
                                      (value-of rand env))]))))
```

Your task is to implement four new versions of this interpreter, each one using a different parameter-passing convention.

Name your interpreters as follows:

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

- You will need to implement the `empty-env`, `extend-env`, `apply-env`, `make-closure`, and `apply-closure` helpers. Use a **functional representation** of both environments and closures.
- For the most part, you can use the same set of helpers for every interpreter. However, since `make-closure` calls the interpreter, you'll need to implement versions of `make-closure` to go along with each interpreter.
- All interpreters must handle the following: booleans, numbers, variables, `lambda`, application, `zero?`, `sub1`, `*`, `if`, and `random`.
- Your `val-of-cbr` and `val-of-cbv` interpreters (not the other two) must also handle `begin2` and `set!`.
- You should use *boxes* to help implement parameter-passing conventions. For more about boxes and the operations you can perform with them, see the Racket Documentation [http://docs.racket-lang.org/reference/boxes.html].

Suggested Tests

Remember, these tests are just a few examples; always write your own tests to help verify your code.

```
;; Making sure set! works
> (val-of-cbr
   '((lambda (x) (begin2 (set! x #t)
                         (if x 3 5))) #f)
   (empty-env))
3
;; Returns 4 under CBR...
```

```
> (val-of-cbr
   '((lambda (a)
       ((lambda (p)
          (begin2
           (p a)
           a)) (lambda (x) (set! x 4)))) 3)
   (empty-env))
4
;; ...but returns 3 under CBV.
> (val-of-cbv
   '((lambda (a)
       ((lambda (p)
          (begin2
           (p a)
           a)) (lambda (x) (set! x 4)))) 3)
   (empty-env))
3
;; returns 44 under CBR...
(test "interesting-cbr-2"
  (val-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)
;; ...but returns 55 under CBV!  You can change the "begin2" to
;; "begin" and evaluate this in the Racket REPL as evidence that
;; Racket uses CBV.
> (val-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
;; Returns 44 under CBR...
> (val-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
;; ...but returns 33 under CBV.
> (val-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
> (define 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) #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)))
;; call-by-name
;;P(false positive) <= .01
> (val-of-cbname random-sieve (empty-env))
#f
;; call-by-need
> (val-of-cbneed random-sieve (empty-env))
#t
;; Does not terminate with val-of-cbr or val-of-cbv -- try it!
> (val-of-cbname
   '((lambda (z) 100)
     ((lambda (x) (x x)) (lambda (x) (x x))))
   (empty-env))
100
```

Brainteaser

One of the highly influential papers during the 1970s in PL was the paper "CONS should not Evaluate its Arguments" [http://www.cs.indiana.edu/pub/techreports/TR44.pdf], which points out the need for lazy data structures. Cons that does not evaluate its arguments is easy to implement as a macro in Racket, but you will be adding it to your interpreter. To your “val-of-cbv” interpreter, add “cons^” that does NOT evaluate its arguments strictly (in other words, it evaluates them lazily). You should also create the corresponding versions of car and cdr (“car^” and “cdr^”) that operate on “cons^”. You should add the strict versions (regular “cons”, “car”, “cdr”) to your `val-of-cbv` interpreter along with add1, empty list, let, and null?.

```
> (define cons-test
    '(let ((fix (lambda (f)
                  ((lambda (x) (f (lambda (v) ((x x) v))))
                   (lambda (x) (f (lambda (v) ((x x) v))))))))
       (let ((map (fix (lambda (map)
                         (lambda (f)
                           (lambda (l)
                              (if (null? l)
                                  '()
                                  (cons^ (f (car^ l))
                                         ((map f) (cdr^ l))))))))))
         (let ((take (fix (lambda (take)
                            (lambda (l)
                              (lambda (n)
                                (if (zero? n)
                                    '()
                                     (cons (car^ l)
                                           ((take (cdr^ l)) (sub1 n))))))))))
           ((take ((fix (lambda (m)
                          (lambda (i)
                            (cons^ 1 ((map (lambda (x) (add1 x))) (m i))))))) 0)) 5)))))
> (val-of-cbv cons-test (empty-env))
(1 2 3 4 5)
```

Just Dessert

Loeb (and Moeb) are cool functions. Read this [https://github.com/quchen/articles/blob/master/loeb-moeb.md] and do something neat with it. You could get it working in your call-by-need interpreter, for instance. Or make a spreadsheet thingy with it, I guess. If you have trouble with the syntax, well, then learn you a Haskell for great good! [http://learnyouahaskell.com/].