Project Part 1

Jason Hemann

January 19, 2018

1 Preamble

- Do consult class notes, online lecture notes, and test cases when completing this assignment.
- Name your file `main.ss`, and use a command like the following: (`zip <username>-p1.zip main.ss`)
- When CPSing, you may treat built-in procedures such as `null?`, `add1`, `assv`, `car`, `<`, and the like as "simple".
- Test your CPSed procedures using the initial continuation returned from `empty-k`.

2 To begin: `lex`

Just when you thought you'd seen the last of it ... `lex` is back! If you've got the previous version working, then all we're asking for is a handful of changes.

- Replace your application line with the following `[`(,rator ,rand) `(app ,(lex rator acc) ,(lex rand acc))]`
- Replace your `zero?` line with the following `[`(zero? ,nexp) `(zero ,(lex nexp acc))]`
- Replace your `*` line with the following `[`(* ,nexp1 ,nexp2) `(mult ,(lex nexp1 acc) ,(lex nexp2 acc))]`
- Add lines to `lex` that implement `letcc` (a `lex`-ed form of `let/cc`) and `throw`. Of these, `letcc` is the more interesting one.
- These first three changes aren't required to correctly perform lexical addressing, but will make your `lex` useful later on.

3 The interpreter itself

You should begin with the interpreter below. This is an interpreter for a `lex`-ed language, like the output of `lex` from several assignments now.

```
(define value-of
  (lambda (expr env)
    (pmatch expr
      [(const ,expr) expr]
      [(mult ,x1 ,x2) (* (value-of x1 env) (value-of x2 env))]
      [(sub1 ,x) (sub1 (value-of x env))]
      [(zero ,x) (zero? (value-of x env))]
      [(if ,test ,conseq ,alt) (if (value-of test env)
                                   (value-of conseq env)
                                   (value-of alt env))]
      ;; [(letcc ,body) ... ]
      ;; [(throw ,k-exp ,v-exp) ...]
      [(let ,e ,body) (let ((a (value-of e env)))
                        (value-of body (lambda (y) (if (zero? y) a (env (sub1 y))))))]
      [(var ,expr) (env expr)]
      [(lambda ,body) (lambda (a) (value-of body (lambda (y) (if (zero? y) a (env (sub1 y))))))]
      [(app ,rator ,rand) ((value-of rator env) (value-of rand env))])))

(define empty-env
  (lambda ()
    (lambda (y)
      (error 'value-of "unbound identifier"))))

(define empty-k
  (lambda ()
    (lambda (v)
      v)))
```

CPS this interpreter. Call it `value-of-cps`. Use the "let trick" to eliminate the let binding when you CPS the `let` line. You might consider always using `k^` as the additional continuation variable to your extended environments. Do not apply `k^` to the call to error in `empty-env`. This is similar to the behavior of `times-cps-shortcut` from Assignment 6. Scheme's `call/cc` may not be used in your CPSed interpreter. You might consider comment out some of your `pmatch` clauses, and CPSing the interpreter a few lines at a time. But try to finish this step entirely before you move on to the next one. Since your closures and environments are both implemented as functions, you should consider renaming them to `env-cps` and `c-cps`, respectively.