Homework 6

Jason Hemann

January 17, 2018

1 Preamble

- Do consult class notes, online lecture notes, and test cases when completing this assignment.
- You can upload your file directly, or in a `.zip` archive, but you will not need `pmatch` for this assignment.
- If you choose to zip it, zip your file as `main.ss`, using a command like the following: (`zip <username>-hw6.zip main.ss`)
- You should be writing your functions in proper CPS form.
- 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 Assignment

For this assignment, you will convert several short programs to continuation-passing style. Please observe the following guidelines:

Use the following definition of `empty-k`

```
(define empty-k
  (lambda ()
    (let ((once-only #f))
      (lambda (v)
        (if once-only
            (error 'empty-k "You can only invoke the empty continuation once")
            (begin (set! once-only #t) v))))))
```

You have seen `empty-k` defined as the following:

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

However, the one above is much better, in that it will help you better detect if you have made a mistake in CPSing.

1. Define and test a procedure `binary-to-decimal-cps` that is a CPSed version of the following `binary-to-decimal` procedure:

   ```
   (define binary-to-decimal
     (lambda (n)
       (cond
         [(null? n) 0]
         [else (+ (car n) (* 2 (binary-to-decimal (cdr n))))])))
   ```

   `binary-to-decimal` uses little-endian binary numbers; you should consider binary sequences with one or more trailing 0s to be ill-formed binary numbers (bad data). Here are a few sample calls to make the meaning clear.

   ```
   > (binary-to-decimal '())
   0
   > (binary-to-decimal '(1))
   1
   > (binary-to-decimal '(0 1))
   2
   > (binary-to-decimal '(1 1 0 1))
   11
   ```

2. Define and test a procedure `times-cps` that is a CPSed version of the following times procedure.

   ```
   (define times
     (lambda (ls)
       (cond
         [(null? ls) 1]
         [(zero? (car ls)) 0]
         [else (* (car ls) (times (cdr ls)))])))
   ```

   Here are some examples of calls to `times`:

   ```
   > (times '(1 2 3 4 5))
   120
   > (times '(1 2 3 0 3))
   0
   ```

3. Define a modified version of your `times-cps` above, called `times-cps-shortcut` that doesn't apply `k` in the `zero?` case. Instead, maintain the behavior of the `zero?` case in `times` - simply return the `0` and not performing further computation. While this certainly violates the standard rules of CPSing the program, it provides an interesting look at optimizations CPSing allows us.

4. Define and test a procedure plus-cps that is a CPSed version of the following plus procedure:

```
(define plus
  (lambda (m)
    (lambda (n)
      (+ m n))))
```

Here are some examples of calls to plus:

```
> ((plus 2) 3)
5
> ((plus ((plus 2) 3)) 5)
10
```

5. Define and test a procedure `remv-first-9*-cps` that is a CPSed version of the following `remv-first-9*` procedure, which removes the first 9 in a preorder walk of the arbitrarily nested list ls:

```
(define remv-first-9*
  (lambda (ls)
    (cond
      [(null? ls) '()]
      [(pair? (car ls))
       (cond
         [(equal? (car ls) (remv-first-9* (car ls)))
          (cons (car ls) (remv-first-9* (cdr ls)))]
         [else (cons (remv-first-9* (car ls)) (cdr ls))])]
      [(eqv? (car ls) '9) (cdr ls)]
      [else (cons (car ls) (remv-first-9* (cdr ls)))])))
```

Here are some example calls to `remv-first-9*`:

```
> (remv-first-9* '((1 2 (3) 9)))
((1 2 (3)))
> (remv-first-9* '(9 (9 (9 (9)))))
((9 (9 (9))))
> (remv-first-9* '(((((9) 9) 9) 9) 9))
((((() 9) 9) 9) 9)
```

6. Define and test a procedure `cons-cell-count-cps` that is a CPSed version of the following `cons-cell-count` procedure:

```
(define cons-cell-count
  (lambda (ls)
    (cond
      [(pair? ls)
       (add1 (+ (cons-cell-count (car ls)) (cons-cell-count (cdr ls))))]
      [else 0])))
```

7. Define and test a procedure `find-cps` that is a CPSed version of the following `find` procedure:

```
(define find
  (lambda (u s)
    (let ((pr (assv u s)))
      (if pr (find (cdr pr) s) u))))
```

   Here are some sample calls to find:

```
> (find 5 '((5 . a) (6 . b) (7 . c)))
a
> (find 7 '((5 . a) (6 . 5) (7 . 6)))
a
> (find 5 '((5 . 6) (9 . 6) (2 . 9)))
6
```

8. Define and test a procedure `ack-cps` that is a CPSed version of the following `ack` procedure:

```
;; ack: computes the Ackermann function
;; (http://en.wikipedia.org/wiki/Ackermann_function).  Warning: if you
;; run this program with m >= 4 and n >= 2, you'll be in for a long
;; wait.
(define ack
  (lambda (m n)
    (cond
      [(zero? m) (add1 n)]
      [(zero? n) (ack (sub1 m) 1)]
      [else (ack (sub1 m)
                 (ack m (sub1 n)))])))
```

9. Define and test a procedure `M-cps` that is a CPSed version of `M`, which is a curried version of `map`. Assume for the CPSed version that any `f` passed in will also be CPSed.

```
(define M
  (lambda (f)
    (lambda (ls)
      (cond
        ((null? ls) '())
        (else (cons (f (car ls)) ((M f) (cdr ls))))))))
```

10. Consider the corresponding call to `M`, called `use-of-M`. Using your CPSed `M-cps`, re-write `use-of-M` to call `M-cps`, and make all the appropriate changes (including CPSing the argument). Name it `use-of-M-cps`.

```
(define use-of-M
  ((M (lambda (n) (add1 n))) '(1 2 3 4 5)))
```