**CSSE 304 Assignment #14 Updated for Spring, 2014**

**Reading:** See the reading assignments on the Schedule page.

**Programming problems:**

1. **(70 points)** Six procedures to convert to continuation-passing style (CPS).

In order for a procedure definition to be in CPS, all calls to non-primitive (as defined in class during our discussion of CPS) procedures must be in tail position, and thus any call to a CPS procedure must produce the final answer for the computation that involves that call. That is, the procedure passed as a continuation to any CPS procedure must contain *all* of the information necessary to complete the computation.

In order to receive credit for each part of this problem, two criteria must be met:

* The procedure passes the grading program’s tests.
* A by-hand analysis by one of the graders shows that the required procedure and any non-primitive helper procedures that it calls really are in CPS form.

Here are some clues that your program may not be in proper CPS form:

1. The continuation of one of your recursive calls is the identity procedure: (lambda (v) v).
2. If you trace all of your CPS procedures for one part of this problem and run it on non-trivial test cases, you see the   
   | | | indentation pattern when you run your test cases, indicating that tail-recursion is not in play here.
3. (5 points) Write member?-cps. Form: (member?-cps item list continuation). What it calculates: same thing as member? Examples:

> **(member?-cps 1 '(3 2 4 1 5) (lambda (v) v))**

#t

> **(member?-cps 7 '(3 2 4 1 5) list)**

(#f)

1. (10 points) Here is my solution to an exercise from a previous assignment:

(define set?

(lambda (ls)

(cond

[(null? ls) #t]

[(not (pair? ls)) #f]

[(member? (car ls) (cdr ls)) #f]

[else (set? (cdr ls))])))

You are to write set?-cps, a CPS version of set? Since member? is not primitive, you must instead use member?-cps in your solution.

**Examples:**

> **(set?-cps '(a b c b d) (lambda (v) v))**

#f

> **(set?-cps '(a b c)**

**(lambda (v)**

**(set?-cps '()**

**(lambda (w)**

**(list v w)))))**

(#t #t)

1. (10 points) Here is my solution to another exercise:

(define intersection

(lambda (los1 los2)

(cond

[(null? los1) '()]

[(member? (car los1) los2)

(cons (car los1)

(intersection (cdr los1) los2))]

[else (intersection (cdr los1) los2)])))

You are to write intersection-cps, which should of course use member?-cps

> **(intersection-cps '(a b c d e) '(f e t b c) (lambda (v) v))**

(b c e)

> **(intersection-cps '(a b c d e)**

**'(f e t b c)**

**(lambda (v) (intersection-cps v '(a c e d) list)))**

((c e))

1. (10 points) Sometimes we may want to use a non-CPS procedure in a context where a CPS procedure is expected. This is akin to the adapter pattern (<http://en.wikipedia.org/wiki/Adapter_pattern>) but applied to procedures instead of classes. Write an adapter procedure called make-cps that takes a one-argument non-cps procedure and produces a corresponding two-argument procedure that can be called in a CPS context. We would usually apply make-cps to one of Scheme’s built-in procedures. This procedure may be helpful in a subsequent part of this exercise.  
     
   **Examples:**

> **(let ([car-cps (make-cps car)])**

**(car-cps '(1 2 3) list))**

(1)

> **(let ([count 0])**

**(andmap-cps**

**(make-cps (lambda (x)**

**(set! count (+ 1 count))**

**(positive? x)))**

**'(4 3 9 0 1)**

**(lambda (v) (list v count))))**

(#f 4)

1. (10 points) Write **andmap-cps**.   
   Form: (andmap-cps pred-cps list continuation), where pred-cps is a cps version of a predicate. Your andmap-cps must short-circuit. I use make-cps in my tests of andmap-cps.  
     
   **Examples:**

> **(andmap-cps (make-cps number?) '(2 3 4 5) list)**

(#t)

> **(andmap-cps (make-cps number?) '(2 3 a 5) list)**

(#f)

> **(andmap-cps (lambda (L k) (member?-cps 'a L k)) '((b a) (c b a)) list)**

(#t)

> **(andmap-cps (lambda (L k) (member?-cps 'a L k)) '((b a) (c b)) not)**

#t

> **(let\* ([count 0]** ; check for short-circuit

**[check-and-increment-cps**

**(lambda (x k)**

**(set! count (+ 1 count))**

**(k (number? x)))])**

**(andmap-cps check-and-increment-cps**

**'(3 4 5 #f #t)**

**(lambda (v)**

**(cons count v))))**

(4 . #f)

In the A11 document, one of my tests for make-cps calls andmap-cps, and vice-versa.

I used tests like that for the grading program also. Thus you won't get full credit for either until you have written both.

(f) (25 points) This is considerably harder than the other parts of this problem!

The following code checks to see if its argument is a matrix (in the sense of assignment 3). Note that matrix entries do not have to be of any particular type.

(define matrix?

(lambda (m)

(and (list? m)

(not (null? m))

(not (null? (car m)))

(andmap list? m)

(andmap (lambda (L) (= (length L) (length (car m))))

(cdr m)))))

You are to write matrix?-cps, a CPS version of matrix?

Note that andmap, length and list? are not primitives (in the sense that we have used *primitive* with respect to CPS procedures), so you must write and use andmap-cps, list?-cps, and length-cps.

You will also need to expand and into nested ifs, since only the last part of the and in my code is in tail position.

**Examples:**

> **(matrix?-cps '((1 2) (3 4)) list)**

(#t)

> **(matrix?-cps '((1 2) (3 4 5)) not)**

#t

> **(matrix?-cps '(()) (lambda (v) v))**

#f

> **(matrix?-cps '((2)) (lambda (v) v))**

#t

> **(matrix?-cps '((2))**

**(lambda (v)**

**(matrix?-cps #f**

**(lambda (w) (list v w)))))**

(#t #f)

**2. (25 points) memoize.** In class we saw a memoized version of Fibonacci. It stores all function values that it has previously calculated, so that it does not have to recompute them later. We can write a general **memoize** function that takes any function **f** and returns a function that takes the same arguments and returns the same thing as **f** but also caches all previously-computed values so it does not have to recompute them.

> **(define (fib n)**

**(if (< n 2)**

**1**

**(+ (fib (- n 1))**

**(fib (- n 2)))))**

> **(define fib-memo (memoize fib))**

> **(fib-memo 12)**

233

It is hard to tell from the above transcript that fib-memo is any different than fib. But a test that includes timing info may be able to tell.

(15) You are to write the memoize function. Of course it should pass the grading program tests, but it will also be checked by hand. Think about what kind of test the grading program might use to determine whether it is likely that your function does indeed create a memoized version of the function that is passed to it.

In order to make the memoized, function more efficient, you should use a hash table to store the previously computed values. Scheme's make-hashtable constructor requires a hash function and an equivalence test as arguments, so a call to memoize will look like (memoize f hash equiv?), where the hash function is appropriate for the list of arguments passed to f. Details are in [TSPL, Section 6.13](http://www.scheme.com/tspl4/objects.html#./objects:h13).

(10) In addition, answer the following question (put your answer in comments at the very beginning of your 14.ss code). Why is the time savings (compared to fib) for the above definition of fib-memo less dramatic than the time savings for the definition of fib-memo in the Day 20 PowerPoint slides?

**3. (25 points) subst-leftmost using multi-value returns.** The interface to the subst-leftmost procedure is the same as in Assignment 08, and the restriction that your code may not recursively descend into the same subtree twice is still in place. Most likely you used a list to return multiple values from each recursive call; now you should use values to do that and call-with-values (or with-values or mv-let; these were defined in session 21 class and linked form the schedule page) to receive the multiple return values.