CSSE 304 Assignment #15 Updated for Spring, 2016

This is an individual assignment, You may continue working on interpreter enhancements with your partner, but you should do this assignment yourself.

Programming problems:

1. (90 points) Procedures to convert to continuation-passing style (CPS).

In this exercise, you will write CPS procedures that use the "Scheme procedure" representation of the continuation ADT. To make the code as representation-independent as possible, **you are required to call `apply-continuation` whenever you apply a continuation**, even though this procedural representation would allow you to apply the continuation directly.

In your code, 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, these criteria must be met:

- The procedure passes the grading program's tests.
- Every application of a continuation must be done *via* the **apply-continuation** procedure, as in the class examples. This will be checked by hand.
- 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. I.e., answers are always passed to a continuation, and all calls to non-primitive procedures (including `apply-continuation`) are in tail position.

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

1. The continuation that you pass into 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. Note that there may be an occasional single level of indentation (or even two levels), but the maximum level should not depend on the size or complexity of the data that is passed to the procedure.

(a) (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)
```

(b) (10 points) Here is my solution to an exercise from an earlier 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?` Treat `member?` As a non- primitive, so that you must use `member?-cps` in your solution.

```
> (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)
```

(c) (30 points) Here is my solution to `domain`, an exercise from another term

```
(define 1st car)

(define set-of   ; removes duplicates to make a set
  (lambda (s)
    (cond [(null? s) '()]
      [(member (car s) (cdr s))
       (set-of (cdr s))]
      [else (cons (car s)
                  (set-of (cdr s)))])))

 (define domain             ; finds the domain of a relation.
   (lambda (rel)
     (set-of (map 1st rel))))
```

You are to write `domain-cps`, which is a transformation-to-cps of the above code. You will also need to write the following four cps procedures that `domain-cps` calls, and make sure that the calls to them are in in tail position:

```
(set-of-cps L k)
(map-cps proc-cps L k)   ; any procedure that map-cps takes as its first argument must be in CPS form.
(1st-cps L k)            ; A CPS version of 1st, so it can be used as an argument to map-cps.
(member?-cps item L k)   ; Is item an element of L?
```

```
> (domain-cps '((1 2) (3 4) (1 3) (2 7) (1 6) (4 3) (3 8))
    (lambda (answer) (format "domain is ~a" answer)))
"domain is (2 1 4 3)"
```

(d) (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 will only apply `make-cps` to Scheme's built-in procedures. (I.e. you are not allowed to apply it to any procedures that you write.) 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)
```

(e) (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)
```

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) `cps-snlist-recur`

`cps-snlist-recur` is not itself a cps procedure, but it expects any of its arguments that are procedures to be cps procedures. It produces a cps-procedure that does the `snlist-recur` recursion pattern.

You may start with my definition of `sn-list-recur`

```
(define snlist-recur
  (lambda (seed item-proc list-proc)
    (letrec ([helper
              (lambda (ls)
                (if (null? ls)
                    seed
                    (let ([c (car ls)])
                      (if (or (pair? c) (null? c))
                          (list-proc (helper c) (helper (cdr ls)))
                          (item-proc c (helper (cdr ls)))))))])
      helper)))
```

or with your own definition. For example, the solution might begin like this

```
(define cps-snlist-recur
  (lambda (base-value item-proc-cps list-proc-cps)
    (letrec
      ([helper (lambda (ls k)
                 ; you fill in the details.
       ]))))
```

You may need to create cps versions of some "primitive" CPS procedures, for use with `cps-snlist-recur`. For example

```
(define +-cps
  (lambda (a b k)
    (apply-k k (+ a b))))
```

You should only do this sort of thing with primitive procedures that are inherently non-recursive. If you need a cps-version of a recursive procedure (such as `length` or `append`), you should do the recursion yourself in cps.

Using `cps-snlist-recur` to define a recursive function:

```
(define sn-list-sum-cps
  (cps-snlist-recur 0 +-cps +-cps))
```

Example of its use:

```
> (sn-list-sum-cps '((1 (2 3 (()) 4) 5))
                   (lambda (x) (format "answer is ~s" x)))
"answer is 15"
```

You are to define `cps-snlist-recur`, and then use it to define the following procedures (based on the similarly-named procedures from assignment 9). Each of those takes an extra argument, which is a continuation. As in the `sn-list-recur` assignment, all recursion in these procedures must come from `cps-sn-list-recur`, not from directly recursive calls in your code for the three procedures.

```
sn-list-reverse-cps
sn-list-occur-cps
sn-list-depth-cps
```

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].

(10) In addition, answer the following question (put your answer in comments at the very beginning of your `15.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 21 PowerPoint slides?

Caution: *Chez* Scheme has a function, `make-hash-table`, which is similar to `make-hashtable`. You probably want to use `make-hashtable`.

3. (25 points) subst-leftmost using multi-value returns. The interface to the `subst-leftmost` procedure is the same as in previous assignments, 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 to a helper procedure; now you should use `values` to do that return, and use `call-with-values` (or `with-values` or `mv-let`; these were (or soon will be) demonstrated in class and are linked from the schedule page) to receive the multiple return values.