CSSE 304 Assignment #15 Updated for Winter, 2019-20

This is an individual assignment. You should continue working on interpreter milestones with your partner, but you should do this assignment yourself. A15 does not depend on A14, and A16 does not depend on it, so you can work on this assignment at times when you are waiting on your partner to be able to get together to work on those assignments.

Problems 1 and 2 involve writing code in continuation-passing-style. In Problem 1, you will represent continuations by Scheme procedures (as we did in the in-class live coding day 23 (winter, 2019-20). In Problem 2, represent continuations by record structures, using define-datatype; this will be a warm-up for A18.

For problems 1 and 2, all calls to substantial procedures must be in tail position. If not, you will receive zero credit even if the PLC server gives you all of the points. I suggest that you have a friend look over your code to see if they can spot any substantial calls that are not in tail position.

1. **(55 points)** Procedures to convert to continuation-passing style (CPS) using the "scheme procedure" representation of continuations.

Warm-up (not for credit). In class we wrote `intersection-cps`. For the non-null case we could call `intersection-cps` on the cdr first and place the call to `memq-cps` in its continuation, or we could call `memq-cps` first and place the call(s) to `intersection-cps` in its continuation. In class I chose the first option. It would be good practice for you to try the second option.

Important note: `apply-k` is a substantial procedure that can only be called in tail position, `make-k` is not substantial.

In this problem, 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-k` whenever you apply a continuation and call `make-k` whenever you make a continuation**, even though this Scheme procedure representation would allow you to apply the continuation directly. In the next problem, you will implement continuations as data structures, so the use of `apply-k` will be unavoidable.

Don't forget: any continuation to a CPS procedure must contain *all* of the information necessary to complete the entire original computation.

In order to receive full credit for any 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-k** procedure, as in the class examples. This will be checked by hand.
- Every continuation creation must be done *via* the **make-k** procedure applied to a procedure that can take one argument.
- A by-hand analysis by one of the graders shows that the required procedure and any non-primitive helper procedures that it calls actually are in CPS form. I.e., answers are always passed to a continuation, and all calls to substantial 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) (30 points) Here is my solution to `domain`, an Assignment 3 exercise from a previous 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 all calls to them are 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))
              (make-k (lambda (answer) (format "domain is ~a" answer))))
"domain is (2 1 4 3)"
```

(b) (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 and other non-recursive procedures. (I.e. you are not allowed to apply `make-cps` to any recursive procedures that you write.) This procedure may be helpful in a subsequent part of this problem.

Examples:

```
> (let ([car-cps (make-cps car)])
     (car-cps '(1 2 3) (make-k list)))
(1)
> (let ([count 0])
   (andmap-cps
    (make-cps (lambda (x)
                (set! count (+ 1 count))
                (positive? x)))
    '(4 3 9 0 1)
    (make-k (lambda (v) (list v count)))))
(#f 4)
```

(c) (15 points) Write **`andmap-cps`**.

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.

Form: `(andmap-cps pred-cps list continuation)`, where `pred-cps` is a cps version of a predicate. Your `andmap-cps` must short-circuit.

Examples:

```
> (andmap-cps (make-cps number?) '(2 3 4 5) (make-k list))
(#t)
> (andmap-cps (make-cps number?) '(2 3 a 5) (make-k list))
(#f)
> (andmap-cps (lambda (L k) (member?-cps 'a L k)) '((b a) (c b a)) (make-k list))
(#t)
```

```
> (andmap-cps (lambda (L k) (member?-cps 'a L k)) '((b a) (c b)) (make-k not))
#t
> (let* ([count 0]  ; check for short-circuit
         [check-and-increment-cps
           (lambda (x k)
             (set! count (+ 1 count))
             (apply-k (number? x)))])
    (andmap-cps check-and-increment-cps
                '(3 4 5 #f #t)
                (make-k (lambda (v)
                          (cons count v)))))
(4 . #f)
```

(d) (0 points) `cps-snlist-recur`

This is a great problem. But since I am adding the "continuation as datatype" problem, I thought I should keep the workload down by removing a problem. I left the description here in case you want extra practice, **but this problem is not required**.

`cps-snlist-recur` is not itself a cps procedure, but it expects all 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.

How to use `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))
                   (make-k (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. (30 points) CPS with data structures continuations.
You will want to refer to the live code that we developed on Day 24 (Winter, 2019-20). It is in the Live-in-class folder. As we did in the live coding, you will need to define the continuation datatype, including the `init-k`, `list-k`, and `not-k` variants. For this part, name your continuation-application procedure `apply-k-ds` so that it will not interfere with your `apply-k` procedure from the previous problem.

Important note: `apply-k-ds` is a substantial procedure that can only be called in tail position. The constructors for the various `continuation` variants are not substantial.

(a) Here is my solution to the `free-vars` problem from A10, along with some supporting procedures:

```
; This is my solution to the free-vars problem from A10.
; It was for the original lambda-calculus expressions where lambdas
; have only one parameter and one body, and applications have only one operand.

(define 1st car)
(define 2nd cadr)
(define 3rd caddr)

; This procedure is not needed for the given solution because memq is built-in.
; But you are required to call it from the CPS version of union.
(define memq-cps
  (lambda (sym ls k)
    (cond [(null? ls)
           (apply-k-ds k #f)]
          [(eq? (car ls) sym)
           (apply-k-ds k #t)]
          [else (memq-cps sym (cdr ls) k)])))

(define union ; s1 and s2 are sets of symbols.
  (lambda (s1 s2)
    (let loop ([s1 s1])
      (cond [(null? s1) s2]
            [(memq (car s1) s2) (loop (cdr s1))]
            [else (cons (car s1) (loop (cdr s1)))]))))

(define remove ; removes the first occurrence of sym from los.
  (lambda (sym los)
    (cond [(null? los) '()]
          [(eq? sym (car los)) (cdr los)]
          [else (cons (car los) (remove sym (cdr los)))])))

(define free-vars ; convert to CPS.  You should first convert
  (lambda (exp)   ; union and remove.
    (cond [(symbol? exp) (list exp)]
          [(eq? (1st exp) 'lambda)
           (remove (car (2nd exp))
                   (free-vars (3rd exp)))]
          [else (union (free-vars (1st exp))
                       (free-vars (2nd exp)))])))
```

Write and test CPS versions of these procedures (and use the CPS version of the (substantial) `memq` from the in-class code).

You may find the `exp?` procedure from the in-class code to be helpful in your `define-datatype` code.

A few test cases:

```
> (union-cps '(a c e g r) '(b a g d t) (init-k))
(c e r b a g d t)
> (remove-cps 'a '(b c e a d a a ) (list-k))
((b c e d a a))
> (remove-cps 'b '(b c e a d a a ) (list-k))
((c e a d a a))
> (free-vars-cps '(a (b ((lambda (x) (c (d (lambda (y) ((x y) e))))) f)))
                 (init-k))
(a b c d e f)
```

3. (40 points, including 15 for answering a question) 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)) ; the actual interface you are to use for memoize is described below
> (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.

(25 points) 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.

(15 points) 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 PowerPoint slides? Note that in the past many students have given a naïve and incorrect answer to this question.

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

4. (25 points) subst-leftmost using multi-value returns. The interface to the `subst-leftmost` procedure is the same as in previous assignments, and the restrictions that your code must short-circuit and 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` to receive the multiple return values. These were (or soon will be) demonstrated in class and are linked from the schedule page) .

Some past Piazza questions and answers

Is car not a primitive procedure?

1c has us create 1st-cps so that we can use it as an argument call. I thought we were allowed to use primitive procedures in non tail positions, and isn't 1st just a renaming of car?

the students' answer 1st-cps takes two arguments, a list and a continuation. Car just jakes a list. Once you start writing domain-cps you will see the distinction.

HW15 set-of-cps member or member?

There is a "member" procedure in the given code of set-of. Do we treat it as a primitive procedure, or we need to substitute it with member?-cps?

```
(define set-of
   (lambda (s)
      (cond
         [(null? s) '()]
         [(member (car s) (cdr s)) (set-of (cdr s))]
         [else (cons (car s) (set-of (cdr s)))])))
```

the students' answer You need to use member?-cps.

tail position?

is the last argument to apply-continuation in tail position? for example in (apply-continuation k (some-cps 'x another-k)) is the call to some-cps in tail position?

the students' answer, I don't think it's technically in tail position since nothing in a function application is in tail position.

the instructors' answer Not in tail position, so you are not allowed to write that code.

followup discussions Ok, so I see it's not in tail position then. So you'd have to rearrange the code so that some-cps happens before that and apply the continuation to the result.

Claude Anderson Yes.

pair? in HW15

Could we just use pair? Directly? Or we have to write a pair?-cps instead?

the students' answer I believe pair? is a primitive procedure, so we can probably use it directly

map-cps

Is the continuation taken by map-cps just passed to the cps-procedure? Wouldn't map-cps need a continuation for both the procedure and the list itself?

the instructors' answer, The continuation passed to map-cps is the continuation of the call to map-cps. map-cps needs to create the continuations that it passes to the calls to the procedure being mapped.

map-cps is not a trivial procedure to write; my version is 10 lines long.

Restriction on Identity procedure

The assignment says that if

"The continuation that you pass into one of your recursive calls is the identity procedure: (lambda (v) v)."

is a clue that your program may not be in proper CPS form. Does this mean that within a function, we cannot call another cps function and give it the identity procedure?

For example, in set?-cps we can't do (member?-cps .... (lambda (v) v))?

I'm just not sure how else we can get the value that we need since we'll be applying some continuation.

the instructors' answer, A call to a CPS procedure (like member?-cps) does not return a value. When it gets a value, it passes that value to its continuation. That continuation must be able to do ALL of the rest of the computation.

If you think you need to pass the identity procedure as the continuation, I expect that your call to member?-cps is not in tail position, a violation of the "tail-recursive" rules. You probably need to call member?-cps and include in the continuation of that call a call to whatever function you were going to pass the result of that identity procedure to.

Applying a continutation

Is there a difference between doing

```
(apply-continuation k v)
```

and just doing

```
(k v)
```

?

the instructors' answer, The difference is representation-independence. Don't forget that we are going to be changing our representation of continuations soon. If you use apply-continuation in your code, you will not have to change it when we change the representation. It will also make your code more readable. And it will make the use of trace for debugging be more effective.

CPS: Should make-k and apply-k be treated as primitives?

make-k: primitive

For now, it just wraps a lambda. In a couple of weeks when we do data-structures representations of continuations, make-k will just make a datatype object, no recursion involved. Thus calls to make-k do not have to be (and almost never are) in tail position.

apply-k: not primitive

When we apply a continuation, that may result in a call to a non-primitive procedure. Thus all calls to apply-k must be in tail position.

make-cps?

I tried testing map-cps but it said that make-cps is not bound. Was that a procedure that we had to write?

the students' answer,
where students collectively construct a single answer

It's part of 1d. It's one of those times where a test refers to something you haven't written yet.

the instructors' answer,
where instructors collectively construct a single answer

I could not find an easy way to avoid circularity in the dependence among the various procedures.

And why would we want make-cps except to produce procedures to feed to map?

Is list a primitive

In terms of what needs to be in tail form is list, as in (list a 4 5) -> (a 4 5), a primitive procedure

the instructors' answer,
where instructors collectively construct a single answer

Actions

Yes

"cps-snlist-recur is not itself a cps procedure...

, but it expects all of its arguments that are procedures to be cps procedures"

Does that mean when implementing cps-snlist-recur, all the rules of non-primitive, tail position calls go out the window?

the students' answer,
where students collectively construct a single answer

No.
The procedure returned by cps-snlist-recur should still be cps. You can always test your code by tracing all the non-primitive procedures.
I put the following lines at the end of my document. When testing on any of the test cases, it should print no indentation.

```
(trace apply-k)
(trace andmap-cps)
(trace make-cps)
(trace domain-cps)
(trace map-cps)
(trace set-of-cps)
(trace 1st-cps)
(trace set?-cps)
(trace member?-cps)
(trace +-cps)
(trace cps-snlist-recur)
(trace sn-list-sum-cps)
(trace sn-list-depth-cps)
(trace sn-list-occur-cps)
(trace sn-list-reverse-cps)
```

the instructors' answer,
where instructors collectively construct a single answer

The procedures that **cps-snlist-recur** takes as arguments, must be cps. And also the procedure that it returns is cps.

A15 Problem 3

Just in case other people have this problem. with-values is not included in scheme library. There is define-syntax code for with-values under call-with-values. Hope this helps.

PS: call-with-values takes procedures, not valu