CSSE 304 Assignment #19 (individual assignment)

This is an individual assignment, not a partner assignment. You may discuss ideas with others and get help with debugging, but the code you write should be your own.

Programming part (80 points)

Convert the CPS code that I provide to imperative form. In imperative form,

- All calls to non-primitive procedures must be in tail position.
- All non-primitive procedures must be thunks (procedures that are called with no arguments).

You will need to convert the lambda CPS to data structure CPS — that is, you should use something like `(define-datatype continuation continuation? …` in your final code. In your continuation datatype, you must provide the following continuations that contain no fields: `id-k`, `list-k`, `length-k`. I also recommend another one, `r-f-p-k`, which can be used by `read-flatten-print`. `read-flatten-print` will not be called by my test code, but it may be the most convenient approach for debugging your code. I used it to debug my code.

All of the continuation constructors may be considered primitive, but `apply-k` is not primitive, so it must be a thunk that is only called in tail position.
Other non-primitives include flatten-cps, list-sum-cps, the `helper` procedure in `cps-snlist-recur`, `+cps`, `append-cps`, `cons-cps`.

You must use `cps-snlist-recur` to produce the `flatten-cps` and `sum-cps` procedures, but `cps-snlist-recur` itself is not a CPS procedure; it takes CPS arguments and returns a CPS procedure

Code that gets all of the points from the server but does not follow the above rules will receive reduced credit, possibly zero credit.

This starting code contains many comments and test code that I hope you will find useful:
http://www.rose-hulman.edu/class/csse/csse304/201920/Homework/Assignment_19/A19-starting-code.ss

Interface for grading purposes:
Our tests on the server and offline will not use `read-flatten-print`. Instead they will assume that you have defined global variables `slist` and `k`, and they will call `flatten-cps` as in the following example:

```
(begin (set! slist '(((a d () (e) c ) g b) t))
       (set! k (list-k))
       (flatten-cps))
```

The other tests will have the same format, but different initial values for `slist` and `k`. When `k` is `init-k`, `apply-k` should simply return the value of `v`.

In addition, some of the test-cases will call `sum-cps`, which you will also create using `cps-snlist-recur`:

```
(begin
  (set! slist '((1) () 2 (3)))
  (set! k (id-k))
  (sum-cps))
```

Part 2 – Exam prep (40 points) Not assigned for Fall 2020.

I intend for this part to accomplish two things:
1. Get you started preparing for the final exam.
2. Provide a collection of additional practice problems that everyone can use.

You are to come up with at least two questions/problems that you think would be good problems for a comprehensive final exam (an exam that emphasizes things that were not included in the possible materials for Exam 2), along with answers/solutions to those problems. I will grade each question based on my subjective evaluation of its correctness and how appropriate a problem it is. Problems you submit should not be from the homework (or its test cases), slides, past exams that have been posted, etc.

Especially useful: `call/cc` problems and interpreter enhancement/modification problems.

Bonus possibility: If I *really* like one of your problems, so much that I decide to actually use it on the exam:
(a) it is likely that you will get the problem correct on the exam.
(b) I will give you final exam bonus points that are equal to the number of points that I assign to that problem on the exam.

Submission:

Submit your problems/questions and answers/solutions to the A19 Drop Box on Moodle
Then submit only the questions/problems to the A19 Forum on Moodle

---

The original plan for the programming part of this assignment (toned down considerably for the real assignment). You do not have to do this stuff, but it may be worth thinking about how you would do it, and the implications of it:

The purpose of this assignment is to show that we can write this interpreter in a low-level language, not relying on Scheme-specific constructs.

The user-facing interface of your interpreter will be the same as for previous interpreter assignments. `eval-one-exp` and `rep` will be used in exactly the same ways. Internally, the flow will be different.

All of the major interpreter procedures (such as `eval-exp`, `eval-rands`, `apply-proc`, `eval-bodies`, `apply-prim-proc`, `eval-exp`, `apply-k`, `map-cps`, and anything else that needed to be in CPS in Assignment 18, must be in imperative form. You can treat most built-in Scheme procedures as primitives that do not have to be in CPS, also all datatype constructors. Unless `eval-exp` both calls them and is called by them, you may treat `parse-exp` and `syntax-expand` as primitives that do not have to be in CPS.

Not all of A18: To make your task quite a bit smaller, the test cases will not contain any of the specific features that were to be added for A17 (I.e., `define`, `set!`, reference parameters). They will also not include `while`. They will include `call/cc`. You do not have to use `lexical-address`.