CSSE 304 Assignment #18 (5th interpreter assignment) 200 points

You are not required to implement reference parameters or lexical-address in A18. You can (and should) start with your "basic" interpreter from A17a.

A18a:

a. Add `display` and `newline` as primitive procedures (and possibly `printf`) to your interpreted language. You may implement them using the corresponding Scheme procedures. You only need to implement the zero-argument version of `newline` and the one-argument version of `display`. These are mainly for your benefit for debugging purposes. They will not be used in the PLC server's test cases. May be used if we test your code interactively using `rep`.

b. Transform your interpreter to Continuation-Passing Style (CPS). It can be (and preferably should be) based on your A17 interpreter before you added reference parameters and lexical-address. The parser and syntax expander do not need to be in CPS, unless they are called by `top-level-eval` or something that it calls.

c. Which procedures need to be transformed to CPS? Most or all procedures that
   (a) are called (directly or indirectly) by `eval-exp`, **and**
   (b) can also call (directly or indirectly) `eval-exp`
   The bottom line is that your interpreter should correctly interpret `call/cc` and `exit-list`.

   In class we discussed two implementations of the `continuation` abstract datatype; in the first one we represented a continuation by a Scheme procedure; the second one represented each continuation as a record, using `define-datatype`. **The "records *via* `define-datatype`" representation is the one that you must use** for this exercise, for reasons described in class.

   Testing your code: Many of my official tests are going to involve `call/cc` and/or `exit/list`. But before you add those features, I suggest that you make sure that your CPS version of the interpreter works on "normal" code. For example, try it on the test code from assignments 16-17 (you do not have to include code with reference parameters). If it has errors when executing "normal" Scheme code, it will be even more difficult to get it to correctly run code that uses `call/cc`. The A18 tests also include some "legacy" test cases to help you with this.

   Because of the nature of `call/cc`, it is possible that some of the off-line test cases may not work in the context of the usual offline testing framework. You may need to test some of them by hand, by loading your interpreter and pasting them in directly or by using `rep`.

A18b:

d. Add `call/cc` to the interpreted language. You only need to do the version we have discussed in class, where continuations always expect a single argument (in the presence of `call-with-values` and `values`, Scheme continuations sometimes expect multiple arguments or return multiple values, **but you don't have to do this in your interpreter**).

   Obvious restriction: You may not use Scheme's `call/cc` (or anything that is built on top of Scheme's `call/cc`) in your interpreter implementation.

e. Add an `exit-list` procedure to the interpreted language. It is not quite the same as *Chez* Scheme's `exit`; it is more like **`(escaper list)`**. Calling `(exit-list obj1 . . . )` at any point in the user's code causes the pending call to **eval-top-level** to immediately return (as the final result of the current evaluation) a list that contains the values of the arguments to `exit-list`. The call to `exit-list` does not exit the read-eval-print-loop when the code is run interactively. It simply returns the list of its arguments, which will be printed before the *repl* prompts for the next value. You may not use Scheme's `exit` or `call/cc` procedures in the implementation of `exit-list` in your interpreter.

For example,

```
> (eval-one-exp '(+ 4 (- 7 (exit-list 3 5))))
(3 5)
> (eval-one-exp '(+ 3 ((lambda (x)
                        (exit-list (list x (list x)))) 5)))
((5 (5)))
> (rep)
--> (+ 3 (exit-list 5))
(5)
--> (+ 4 (exit-list 5 (exit-list 6 7)))
(6 7)
-->
```

Notes on what to treat as primitives when converting your interpreter to CPS:

Some parts of your interpreter that cannot be treated as primitives (in CPS terminology), so must be converted to CPS:

- `eval-exp`
- Any procedure that `eval-exp` calls that can also call (directly or indirectly) `eval-exp`, including `eval-rands`, `apply-proc`, `apply-primproc`.
- `apply-continuation` (not a CPS procedure, but can only be called in tail position)
- `map` (use `map-cps` instead, and pass it a CPS procedure)

These should not need to be converted to CPS, unless your `eval-exp` calls them and is called by them:

- parse-exp
- top-level-eval
- syntax-expand
- datatype constructors and continuation constructors
- What about the environment procedures? Depends on how you write them and their interaction with the rest of the interpreter.

Fun things to add to your interpreter (not for credit):

Add multiple-value returns to the interpreted language. In particular, implement `values`, `call-with-values`, and `with-values`, as described in section 5.8 of TSPL4. Continuations created by `call/cc` are then allowed to expect multiple values, as described and illustrated in Section 5.8.

Add engines to your interpreted language. Both the engine interface and an approach to implementation are described in TSPL4 Section 12.11. We will also discuss them in class week 9 or early in week 10.

Piazza Questions and Answers from previous terms:

Writing exit-list using given escaper Are we allowed to implement exit-list using the escaper code on the schedule page if we replace the call/cc with our implementation of call/cc?
Instructor answer: Allowed, yes. Advised, no. Once your code is CPS, exit-list can be implemented in a few lines of code, and that does not need escaper or call/cc.

What needs to be CPS in A18? According to the A18 project spec:
These should not need to be converted to CPS, unless your eval-exp calls them **and** is called by them:

- parse-exp
- top-level-eval
- syntax-expand
- datatype constructors and continuation constructors

• What about the environment procedures? Depends on how you write them and their interaction with the rest of the interpreter.

Is this intentionally *and*, or should be *or*? For example, our eval-exp calls parse-exp, but parse-exp does not call eval-exp. Does parse-exp need to be converted to CPS? Likewise, our eval-exp calls apply-env, but apply-env does not call eval-exp. Does apply-env need to be in CPS form?

the students' answer: I believe the "and" is intentional. Only those procedures that are called by eval-exp and call it.
the instructors' answer: The **and** is certainly intentional. Don't forget that the procs that need to be CPS include apply-proc and apply-prim-proc. Also, any time you want to call map or apply within a CPS procedure, you need to call cps versions of those.
followup discussions
A student: You mentioned that we must use the CPS map and apply in our CPS procedures. So in the our apply-prim-proc, we use apply for most of the primitive procedures. If we change them all to apply-cps, do we need to provide cps versions of all the primitive procedures to pass to apply-cps?
Also, can we implement apply-cps using apply, or should we do it another way?

Claude Anderson: My suggestion is to call `apply-k` on the results of the Scheme procedure call for most of the primprocs. But for primprocs `map`, `apply`, and `call/cc` you'll need to do something different.

Another student: Following up on this question: do `apply-env`/`apply-env-ref` need to be in CPS form?

Claude Anderson: For most teams, no. If these procedures do not call eval-exp or call anything that calls eval-exp, you should be okay without converting them to CPS.

If apply-env-ref never calls eval-exp, does it need to be in cps?

In Assignment 18, it says "What about the environment procedures? Depends on how you write them and their interaction with the rest of the interpreter." Since our apply-env-ref is only called (indirectly) from eval-exp but never calls eval-exp, does apply-env-ref need to be cps?
Instructor's answer: No.

Should the fail-proc be in cps?

I noticed in the slide for var-exp the modification is [var-exp (id) (apply-env env id k fail-proc))]
but since fail-proc also will be calling non-primitive procedures, it should be a fail continuation.
However, because it starts with lambda() with no argument, the non-primitive procedures are not evaluated until in apply-env.I'm confused about how to write this fail-proc in cps. Any ideas?

the instructors' answer,
If you need the fail procedure to be in CPS, you can always change the interface to apply-k
to be (lambda (k . args) ...). This allows a continuation to take any number of args, including 0.

CPS in if statements

If you want to make an if statement into CPS
(if 1 2 3) and statement 1 contains non primitive procedures, how do you handle it?

the instructors' answer,
You call the non-primitive procedure first, and put the if in the continuation of that call.