CSSE 304 Assignment #18 (5th interpreter assignment) 200 points Updated for Spring, 2016

Additional test cases may be added. If so, there may be adjustments to point values of some existing tests and/or total assignment points.

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`. This is mainly for your benefit for debugging purposes. Will not be used in my test cases.

b. Transform your Assignment 17 interpreter to Continuation-Passing Style (CPS). The parser and syntax expander do not need to be in CPS, unless they are called by `top-level-eval` or something it calls. Most procedures that `eval-exp` calls need to be in CPS. The bottom line is that your interpreter correctly interprets `call/cc` and `exit-list`, without using Scheme's `call/cc` anywhere in your code.
You are not required to implement reference parameters in A18.

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 are to 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, you should make sure that your CPS version of the interpreter works on "normal" code. For example, try it on the test code from assignments 15-17 (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`. 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 testing framework. You may need to test some of them by hand, by loading your interpreter and pasting them in directly.

c. 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 worry about this in your interpreter).

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

d. 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 or that can call (directly or indirectly) `eval-exp`
- `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 or 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. If there is time, we will discuss them in class.