**CSSE 304 Assignment #16 (5th interpreter assignment)**

As usual, when you do your turn-in for credit, include other team member's usernames on the submission form (unless they did not contribute). Late day rules are the same as on earlier interpreter assignments.

@. (0 points but convenient for testing) Add display and newline as primitive procedures in 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.

a. (50 points) Transform your Assignment 15 interpreter to Continuation-Passing Style. You are allowed to treat the procedures mentioned in Slide 8 of Day 30 as primitives (in the sense that we used that word with respect to CPS), The parser and syntax expander can only be treated as primitives if they are not called directly or indirectly by eval-exp). Testing this part of the assignment will be done by running tests that could have been tests for A15, then checking by hand to make sure that your code really is in CPS form.  
  
In class we discussed two implementations of the continuation datatype; one represented continuations by Scheme procedures; the other represented continuations by records, 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.

b. (30 points) Add call/cc to the interpreted language. Here is a slightly revised version of its description from TSPL:

(call/cc receiver) obtains the current continuation and passes it to receiver, which must accept one argument. The continuation itself is represented by a procedure. Each time this continuation procedure is applied to (a) value(s), it returns the value(s) to the continuation of the call/cc application (see values and call-with-values below). That is, when the continuation procedure is applied, it returns the value of its argument(s)as the result(s) of the application of call/cc. If receiver returns normally when applied to this continuation procedure, the value returned by the call/cc application is the value returned by receiver.

You may not use Scheme's call/cc in your implementation of call/cc.

c. (20 points) Add an exit procedure to the interpreted language (not quite the same as *Chez* Scheme's procedure with the same name; it is more like **(escaper list)** ). Calling (exit obj1 . . . ) at any point in the user's code causes the pending call to **eval-top-level** to immediately return a list that contains the values of the arguments to exit. The call to exit 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 r-e-p loop prompts for the next value.

For example,

**>** **(eval-one-exp '(+ 4 (- 7 (exit 3 5))))**

(3 5)

> **(eval-one-exp '(+ 3 ((lambda (x) (exit (list x (list x)))) 5)))**

((5 (5)))

> (rep)

--> **(+ 3 (exit 5))**

(5)

--> **(+ 4 (exit 5 (exit 6 7)))**

(6 7)

-->

**Extra credit:** The 100 points for the previous parts will be considered a perfect score on this assignment, but you may earn additional points for getting the following things working. Of course, you may not use the corresponding Scheme features in your implementation of these.

d. (40 points) Add multiple-value returns to the interpreted language. In particular, implement values, call-with-values, and with-values, as described in section 5.7 of TSPL. Continuations created by **call/cc** are then allowed to expect multiple values, as described and illustrated in Section 5.7.

e. (30 points). Add engines to your interpreted language. Both the engine interface and an approach to implementation are described in TSPL Section 9.11.

f. (100 points) Interpreter that interprets itself from A15. If you do this one, arrange a time to demo it to your instructor.