Turn in the written part next Friday in class. Turn in the programming part using CMS. Write your solutions very neatly.

1. Give a sample program fragment that produces different results under each of the following parameter-passing methods:

   (a) Call-by-value

   (b) Call-by-reference

   (c) Call-by-value-result

   (d) Call-by-name

2. (a) Write a translation of exceptions into a language with if expressions, let constructs, pairs, and selection operators for the first and second components of the tuple;

   (b) Extend the above translation to support constructs that pass values along with exceptions, e.g., "e_1 `handle` $X(x)$ `=>` e_2" and "`throw` $X(e)$".

3. We want to model non-local control in an imperative setting.

   (a) First, we extend IMP with labels and goto statements:

   $$c ::= \dots \mid l : s \mid \texttt{goto}\ l$$
   $$l \in \mathsf{Labels}$$

   The jump command `goto` l transfers the control at the point where the label command $l : s$ occurs. For simplicity, assume that the label command occurs before the jump during the execution.

   We'd like to model configurations as tuples (s, m, k, c) containing: a store s, a map m from labels to continuations, a current continuation k, and a currently executed command c. The nice fact about this setting is that we can simply model continuations as commands, so m maps labels to commands, and k is some command c'. The evaluation of assignments is:

   $$(s, m, c, x := a) \to (s, m, c, x := a') \qquad \text{if } \langle a, s\rangle \to \langle a', s\rangle$$
   $$(s, m, c, x := v) \to (s[x \mapsto v], m, c, \texttt{skip})$$

   Write the remaining rules for `skip`, $c_1; c_2$, `if` b `then` c_1 `else` c_2, `while` b `do` c, $l : s$, and `goto` l. Note: make sure that while loops work correctly in the presence of goto's.

   (b) Next, we extend the language with break and continue statements:

   $$c ::= \dots \mid \texttt{break} \mid \texttt{continue}$$

   A continue command transfers the control to the next iteration of the innermost enclosing loop; a break command transfers the control to the statement that follows after the innermost enclosing loop.

In the presence of these commands, the evaluation rule for while loops must capture the control points for break and continue. For this, you should use two distinguished labels, `LB` for the break point and `LC` for the continue point, and re-write the rule for while loops as follows:

$$(s, m, c_c, \texttt{while}\ b\ \texttt{do}\ c) \rightarrow (s, m, \texttt{LC} : (\texttt{while}\ b\ \texttt{do}\ c);\ \texttt{LB} : c_c, c')$$

for some appropriately chosen c'. Labels `LB` and `LC` differ from standard labels in the fact that they may occur multiple times in the continuation (for programs with nested loops). For that reason, `LB` and `LC` are never entered in the mapping m (otherwise, each would be mapped to a single continuation).

Show the appropriate command c' above and write the evaluation rules for `break` and `continue`.

(c) Describe in 2-3 sentences what you need to do to allow jumps anywhere in the program.

(d) Describe in one sentence the notion of a continuation at the level of machine code.

(e) Implement all of the constructs in part (a) and (b) in IMP, allowing jumps anywhere in the program. Assume the following datatypes:

```
type var = string
type store = var -> int
type label = string

datatype aexp = Var of var | Int of int | Plus of aexp * aexp
datatype bexp = True | False | Less of aexp * aexp

datatype com = Skip | Seq of com * com | Assign of var * aexp |
               If of bexp * com * com | While of bexp * com |
               Label of label * com | Goto of label |
               Break | Continue
```

Write an evaluation function that maps an input store and a command to an output store:

```
  eval : com * store -> store
```

The evaluation function must raise an exception when it encounters an error:

```
  exception RuntimeError of string
```

Note: You can use the evaluation functions for expressions from assignment 2.