Turn in the code for your assignment using CMS (`http://cms.csuglab.cornell.edu`). Hand in the rest of your assignment (problems 1, 2, 3, and 4c) next Friday in class.

1. Extend the large-step semantics of IMP to support the following constructs:

   (a) Richer operations for boolean expressions: $b ::= \ldots \mid \mathsf{not}\ b \mid b_1\ \mathsf{or}\ b_2 \mid b_1\ \mathsf{and}\ b_2$. Your rules must enforce the short-circuited evaluation of boolean expressions.

   (b) Another loop construct: $\mathsf{do}\ c\ \mathsf{until}\ b$, which iteratively executes c until the test expression b holds. Your semantic rules for this construct should not rely on if commands or while commands.

2. Suppose we further extend IMP to support auto-increment arithmetic expressions:

$$a ::= \ldots \mid x\texttt{++} \mid \texttt{++}x$$

   Both constructs increment the value of x; expression x++ returns the old value of x, and expression ++x returns the new value of x. How does these constructs affect the large-step semantic model of IMP? More precisely:

   (a) What changes are necessary in the semantic domains and relations to support such expressions? Explain why you need the changes.

   (b) Indicate which evaluation rules need to be changed and show two of the updated rules: one for expressions, and one for commands;

   (c) Show the evaluation rules for the new auto-increment expressions.

3. Show that the following command equivalences hold (where $c \sim c'$ if $\langle c, s\rangle \Downarrow s' \Leftrightarrow \langle c', s\rangle \Downarrow s'$):

   (a) $\mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2 \ \sim\ \mathsf{if}\ (\mathsf{not}\ b)\ \mathsf{then}\ c_2\ \mathsf{else}\ c_1$

   (b) $\mathsf{do}\ c\ \mathsf{until}\ b \ \sim\ c\,;\ \mathsf{while}\ (\mathsf{not}\ b)\ \mathsf{do}\ c$

4. Implement an interpreter for IMP in SML based on the small-step semantics of the language. Assume the following datatype declarations:

```
type var = 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

type store = var -> int
```

(a) You have to implement the following one-step evaluation functions:

```
onestep_aexp : aexp * store -> aexp
onestep_bexp : bexp * store -> bexp
onestep_com : com * store -> com * store
```

(b) Extend your interpreter to support the constructs from problem 1, as well multiplication and equality tests:

```
datatype aexp = ... | Times of aexp * aexp
datatype bexp = ... | Equals of aexp * aexp | LessEq of aexp * aexp
               | Not of bexp | And of bexp * bexp | Or of bexp * bexp
datatype com = ... | Until of bexp * com
```

(c) Would an implementation of the interpreter using large-step semantics have been easier or more efficient? Explain.