CS411 Homework 2
IMP Semantics *Due: 15 February 2007*

Correction: a typo has been fixed in part 3b.

Turn in the code for your assignment (problem 4) Thursday evening using CMS. Hand in the rest of your assignment (problems 1, 2, 3) next morning 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: $\mathsf{do}\ c\ \mathsf{while}\ b$, which iteratively executes c as long as 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. Informally, we say that two commands c and c' are semantically equivalent if their execution yields the same results for all inputs.

   (a) Write a formal definition for the semantic equivalence using large-step semantics.

   (b) Prove that the commands "$\mathsf{do}\ c\ \mathsf{while}\ b$" and "$c$; $\mathsf{while}\ (b)\ \mathsf{do}\ c$" are equivalent.

4. Implement an interpreter for IMP in Ocaml based on the large-step semantics of the language. Stores are represented using association lists containing pairs of variables and their integer values in the store. Assume the following datatype declarations:

```
type var = string
type aexp = Var of var | Int of int | Plus of aexp * aexp
type bexp = True | False | Less of aexp * aexp

type com = Skip | Seq of com * com | Assign of var * aexp |
           If of bexp * com * com | While of bexp * com

type store = (var * int) list
```

(a) Finish up the implementation of function `lokkup` that retrieves the value of a variable from a store. If the variable is not defined, your function should raise an exception.

(b) Implement the following evaluation functions, according to the large-step evaluation rules. Use comments in your code to indicate where each rule is implemented in the code:

```
evala : aexp * store -> int
evalb : bexp * store -> bool
evalc : com  * store -> store
```