Hand in your assignment next Friday in class and turn in the programming part using CMS. Write your solution very clearly, or else type your assignment.

1. Consider the analysis that computes signs of variables, as described in class.
   (a) Write down an analysis for a do-until loop.
   (b) Extend the sign analysis presented in class such that it identifies, for each variable x, not only whether $x > 0$, $x < 0$, and $x = 0$, but also whether $x \geq 0$, $x \leq 0$, and $x \neq 0$. Show the changes you need to make to the existing analysis.
   (c) The test conditions in if and while commands can provide useful information about the sign of variables. If I encounter a test $x < 0$ at a point where I don't know the sign of x, then I learn that x is negative on the control path where the test succeeds, and that it is non-negative otherwise. Starting with the analysis from the previous part, re-define the following evaluations to take advantage of the information in conditional tests:
   $$\mathcal{C}'[\![\mathsf{if}\ (x < a)\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2]\!]\ S$$
   $$\mathcal{C}'[\![\mathsf{while}\ (x < a)\ \mathsf{do}\ c]\!]\ S$$
   (d) Show a program c for which the analysis $\mathcal{C}'[\![c]\!]$ from the previous part yields more precise information than the previous analysis (indicate the result that you get with each analysis).

2. Suppose that we change the syntax of IMP such that variables can assigned not only integers, but boolean values too. In that case boolean expressions also include variables. So the syntax changes are:
   $$c ::= \ldots \mid x := b$$
   $$b ::= \ldots \mid x$$
   First, we want to ensure that arithmetic operations always operate on integers, and test conditions always hold boolean values. Otherwise, the program execution yields an error. Second, we want to ensure that variables are always initialized before being used.

   Write an analysis in SML that checks both of these properties. You will use the extended version of IMP from the last assignment (the one with do-until loops and richer sets of operators), but with the new syntax discussed above. You must implement a function `analyze : com -> unit` that analyzes a given command c and issues at most two warnings for each variable: one warning if the variable has been used without being uninitialized in c; and one warning if it had the incorrect kind of value when used in c.

   Note: This is exactly what the Java bytecode verifier does. To minimize the number of local variables, Java bytecodes allow variables to be reused and hold different kinds of values at different points in the program. The bytecode verifier uses an analysis to check that variables are always initialized and are consistent with the constructs that they are occur in.