CSE 6341, Assignment #2
Due: Sept. 26, '18.

Note: The first mid-term will be on Friday, Sept. 28. Topics will be everything we discuss in class by then. Also, this homework must be turned in by start of classtime on Sept. 26. No late homeworks.

1. (6 points). In our *translational semantics*, we have used, whenever necessary, a set of *temporary locations* named `T1, T2`, ..., as needed. In this problem, you have to change that to instead use a *stack* (which we will assume is unlimited in size) that the system provides. The assembly language provides two instructions,

   ```
   PUSH   XYZ
   POP    XYZ
   ```

   The `PUSH` will copy the variable `XYZ` to the top of the stack and leave `XYZ` unchanged; the `POP` instruction will copy the value at the top of the stack to `XYZ` and remove that value from the stack. Also, the arithmetic instructions work only with the stack. Thus:

   ```
   SUB
   ```

   will subtract the value at the top of the stack from the value immediately below it on the stack, remove both values from the stack, and store the result at the top of the stack. So, for example, if you want to add `X1` and `X2` and store the result in `X3`, you can do it as follows:

   ```
   PUSH   X1
   PUSH   X2
   ADD
   POP    X3
   ```

   Also, `PUSH 42` will copy the constant 42 to the top of the stack.

   How would you change the translational semantics we have discussed to use the stack in place of the temporaries? Explain by writing down the revised rules, conditions, etc. Your answer doesn't have to be exhaustive but the key ideas should be clear.

2. (10 points). In our *translational semantics*, we have used, in the assembly language code that we produce, the same variable names as the ones used in the source code. The goal of this problem is to do memory allocation for the variables in the source code and use, in the assembly language code, the addresses corresponding to the source code variables rather than the variable names themselves. So, for example, if we have, in the source program, the statement `X := Y + Z;`, we will produce the following code; no use of stacks:

   ```
   LOAD   100
   ADD    101
   STORE  102
   ```

   where I have assumed that the addresses of `Y, Z, X` are 100, 101, and 102 respectively.

   The most sensible approach to do this is to produce, using appropriate attributes, a *symbol table* in which we keep track of the addresses corresponding to the source variables so that we can look them up as needed. This is somewhat like the *Nest/Decs* attributes, except it won't be quite as complex. Also, to simplify this task, we will change the BNF grammar as follows:

   ```
       <prog>   ::=  <decl seq> <stmt>
   <decl seq>   ::=  <decl> | <decl> <decl seq>
       <decl>   ::=  int <id list>;
    <id list>   ::=  <id> | <id>, <id list>
   ```

The production for `<stmt>` remains as before. In other words, while previously we just used variables in the statements without declaring them, now all the variables used in the `<prog>` must be declared in its `<decl seq>`. Assume that all variables are properly declared; you do not need to check for that.

Define the appropriate attributes, attribute evaluation rules, and conditions, that will correspond to this change so that in the code you produce, we use memory addresses in place of variable names used in the source code. Again your answer doesn't have to be exhaustive but the key ideas should be clear.

Note: In practice, things are a bit more complex because we cannot really allocate memory addresses to variables in the source program (why not?). Instead, what is determined at compile time (and expressed in the AG rules) for the variables declared in a given function are the *off-sets*, corresponding to each of those variables, within any *activation frame* for any call to the function. The compiler can then produce (and we can express in the AG rules) code that will use the value in the *stack pointer* and the off-sets to obtain, at runtime, the actual addresses of the variables.

3. (4 points). Rewrite the following s-expressions using the *list* notation of LISP/Scheme; if it cannot be done for a particular s-expression, explain why not; if it can done partially, do so as much as possible:

```
a. (4 . NIL)
b. ((3 . NIL) . (4 . NIL))
c. (3 . ((4 . NIL) . (5 . NIL)))
d. (3 . (4 . 5))
```