CSE 6341, Assignment #1
Due: Sept. 14, '18.

1. (6 points). Consider the following *BNF* grammar:
   $\langle lstring \rangle ::= \langle s \rangle \langle s \rangle$
   $\langle s \rangle ::= a \mid b \mid c \mid a\langle s \rangle \mid b\langle s \rangle \mid c\langle s \rangle$
   Add appropriate attributes and conditions to the grammar so that only $\langle lstring \rangle$s that satisfy the following condition are allowed: every occurrence of 'a' is immediately preceded by a 'b' *and* followed by a 'c'. In other words, each occurrence of 'a' is sandwiched between b and c. (Thus, for example, '$bacbbcbac$' is legal, but not '$babbcbc$'.) Use synthesized attributes or inherited attributes, or a combination. Do not change the *BNF* grammar; do not compute the whole string and pass it up to the root – use only simple arithmetic functions in your attribute evaluation rules and conditions. If the problem cannot be solved under these constraints, explain why not.

2. (6 points). Consider the following BNF grammar of expressions:

   $\langle exp \rangle \ ::= \ \langle simple \rangle \mid \langle exp \rangle \ + \ \langle exp \rangle \mid \langle exp \rangle \ * \ \langle exp \rangle$

   $\langle simple \rangle \ ::= \ \langle number \rangle \mid \langle variable \rangle$

   where $\langle number \rangle$ and $\langle variable \rangle$ correspond to numbers and (program) variables.

   This grammar does not impose the proper precedence between $+$ and $*$. Without changing the productions, introduce appropriate attributes, evaluation rules, and conditions such that the proper precedence is enforced (i.e., parse trees that do not have the proper precedence, although legal according to the BNF productions, are ruled out by having one or more of the conditions evaluating to *false*).

3. (8 points). Consider the following simple programming language. A program consists of a set of declarations and statements but no nested blocks. The key complication is that declarations and statements may be interleaved in any order. Two kinds of variables, `int` and `bool` may be declared. If `X` is declared at some point, it becomes accessible to statements that *follow* the declaration (but not to the ones that *precede* that declaration). Also, if there is already a declaration for `X`, the new one will override that declaration. This is relevant because we have two different types. A statement may be an assignment or if-then-else, while-loop etc. Here is the BNF grammar:

```
    <prog> ::=  <decsNStmts>
<decsNstmts> ::= <decl>|<stmt>|<decl> <decsNstmts>|<stmt> <decsNstmts>
     <decl> ::= int <id>; | bool <id>;
   <stmt> ::= <assign> | if <id> then <decsNstmts> else <decsNstmts> end;
            |  while <id> do  <decsNstmts> end;
    <assign> ::= <id> = <boolExp>; | <id> = <intExp>;
```

   We won't worry about the details of `<boolExp>` and `<intExp>`.

   Here is the problem: Clearly, this language includes context-sensitive conditions, specifically, that if an `X` is used as the `<id>` in an if-statement or while-statement, it should have been declared previously and its most recent declaration should be as a `<bool>` . [If we introduced details of the `<boolExp>` and `<intExp>`, there will be further conditions to check; but ignore them for this problem.]

   There is one further complication: what happens if there is a declaration inside the *then*-portion or the *else*-portion of an if-statement, or in the body of a loop? What happens when we come of those respective portions? Do we still have access to those declarations? The answer is "no". What you have to do solve this problem is to add appropriate attributes, evaluation rules, and conditions to the above grammar to capture these context-sensitive conditions. Don't just write the grammar; provide a brief intuitive explanation of how your grammar works.