EECS 345: Programming Language Concepts, Written Exercise 1
due Friday, February 7, 2020 in class

Problem 1: Consider the following ambiguous BNF grammar:

```
<C>  →  <C> ? <C> : <C> | <V> = <C>
<C>  →  <C> && <C> | <C> || <C> | !<C> | <V> | true | false | (<C>)
<V>  →  x | y | z
```

Rewrite the grammar so that it is no longer ambiguous and has the following properties:
The operators have the following precedence, from highest to lowest: `( )`, `!`, `&&`, `||`, `?:`, `=`.
The `!`, `?:`, and `=` operators are right associative, and the `&&` and `||` operators are left associative.

Problem 2: Consider the following grammar (yes, it is ambiguous but that is unimportant). The subscripts are used to distinguish otherwise identical non-terminals for the purpose of the questions below.

```
<start1>        →  <stmt3> ; <start3>
<start2>        →  <stmt4>
<stmt1>         →  <declare2>
<stmt2>         →  <assign2>
<declare1>      →  <type3> <var>
<type1>         →  int
<type2>         →  double
<assign1>       →  <var> = <expression3>
<expression1>   →  <expression4> <op> <expression5>
<expression2>   →  <value4>
<op>            →  + | - | * | ÷
<value1>        →  <var>
<value2>        →  <integer>
<value3>        →  <float>
<var>           →  a legal name in the language
<integer>       →  a base 10 representation of an integer
<float>         →  a base 10 representation of a floating point number
```

Suppose our static semantic description has five attributes:

```
type              =  { integer, double }
typetable(<var>)  =  { integer, double, error }
inittable(<var>)  =  { true, false, error }
typebinding       =  (<var>, { integer, double })
initialized       =  (<var>, { true, false })
```

`typetable` maps each possible variable name to its declared type, and `inittable` maps each possible variable name to a boolean indicating whether the variable has been assigned a value. Initially, both `typetable` and `inittable` will map all possible variable names to `error` to indicate that the variables have not been declared in the program.

`typebinding` maps a *single* variable name to its declared type, and `initialized` maps a *single* variable name to whether it has been assigned a value.

For each subscripted non-terminal, provide a rule to calculate its *type*, *table*, *inittable*, *typebindig*, and *initialized* attributes, if that non-terminal requires that attribute. Each attribute should either be *inherited* or *synthesized*, but not both. For example, here are two such rules:

```
<value2>.type := integer
<declare1>.initialized := (<var>, false)
```

(Here I am using `:=` to create a mapping so you can use `=` to mean only mathematical equality.)

Problem 3: Suppose we want to enforce the following rules in the grammar from Problem 2:
(a) The type of the expression must match the type of the variable in all assignment statements.
(b) A variable must be declared before it can be used.
(c) A variable must be assigned a value as its first use in the program.
Where in the parse tree should these rules be checked (i.e. at which non-terminals), and write the precise tests that should be done at those non-terminals using the attibutes available.

Problem 4: You have the following code on which the programmer included desired precondition and postconditions and included (hopefully) correct loop invariants on each of the loops. Given the stated post-condition (i.e. goal) for the code, you are to:
a) State whether the given loop invariants are correct, and if not to give corrected loop invariants.
b) For each numbered statement, give the correct *weakest precondition* for that statement.
c) Determine whether the weakest precondition for statement 1 is logically inferred from the stated precondition of the code.

Precondition: `n` $\geq$ `0` and `A` contains `n` elements indexed from 0

```
1.  bound = n;
    /* Loop invariant: A[bound,..,(n-1)] is sorted (non-decreasing) and A[bound] >= A[0,..,bound-1] */
2.  while (bound > 0) {
3.    t = 0;
4.    i = 0;
      /* Loop invariant: A[t] is the largest element of A[0,..,t] */
5.    while (i < bound-1) {
6.      if (A[i] > A[i+1]) {
7.        swap = A[i];
8.        A[i] = A[i+1];
9.        A[i+1] = swap;
10.       t = i+1;
        }
11.     i++;
       }
12.    bound = t;
     }
```

Postcondition: `A[0]` $\leq$ `A[1]` $\leq$ ...$\leq$ `A[n-1]`

Problem 5: Let M_{state} be a denotational semantic mapping that takes a syntax rule and a state and produces a state. Define the M_{state} mapping for the following three syntax rules *assuming we allow side effects*, that is, assuming expressions and conditions can change the values of variables.

```
<assign>  →  <var> = <expression>
  <if>    →  if <condition> then <statement1> else <statement2>
<while>   →  while <condition> <loop body>
```

Assume we have the following mappings defined:
M_{value} takes a syntax rule and a state and produces a numeric value (or an error condition).
$M_{boolean}$ takes a syntax rule and a state and produces a `true / false` value (or an error condition).
M_{name} takes a syntax rule and produces a name (or an error condition).
Add takes a name, a value, and a state and produces a state that adds the pair **(name, value)** to the state.
Remove takes a name and a state and produces a state that removes any pair that contains the name as the first element.

You may assume the *Add* and *Remove* mappings do not produce errors.