CSCI 305 Homework 2

Due Date: February 23, 2018 @ Beginning of Class

Name:____

Syntax and Semantics

1. Starting with the following BNF grammar:

```
<exp> ::= <exp> + <mulexp> | <mulexp>
<mulexp> ::= <mulexp> * <rootexp> | <rootexp>  |
<rootexp> ::= ( <exp> )
          | a | b | c
```

Construct an EBNF grammar from this grammar with the following modifications. Use the EBNF extensions wherever possible to simply the grammars. Include whatever notes to the reader required to make the associativity of the operators clear:

a. Add subtraction and division operators (- and /) with the customary precedence and associativity.

b. Then add a left-associative operator % between + and * in precedence.

c. Then add a right-associative operator = at lower precedence than any of the other operators.

1. Show that the following grammar is ambiguous. (Note: To show that a grammar is ambiguous, you must demonstrate that it can generate two parse trees for the same string.)

   ```
   <person> ::= <woman> | <man>
   <woman> ::= wilma | betty | <empty>
   <man> ::= fred | barney | <empty>
   ```

2. For the grammar in Question 2, construct an unambiguous grammar for the same language.

Language Systems

1. Suppose the target assembly language for a compiler has these five instructions:

   ```
   push address
   add
   sub
   mul
   pop address
   ```

   In these instructions, and *address* is the name of a static variable (whose actual address will be filled in by the loader). The machine maintains a stack of integers, which can grow to any size. The `push` instruction pushes the integer from the given memory address to the top of the stack. The `add` instruction adds the top integer on the stack to the next-from-the-top integer, pops both off, and pushes the result onto the stack. The `sub` instruction subtracts the top integer on the stack from the next-from-the-top integer, pops both off, and pushes the result onto the stack. The `mul` instruction multiplies the top integer on the stack by the next-from-the-top integer, pops both off, and pushes the result onto the stack. The `pop` instruction pops an integer off the stack and stores it at the given memory address. So, for example, the compiler might translate the assignment `result := offset + (width * n)` into this:

```
push offset
push width
push n
mul
add
pop result
```

Using this assembly language, give translations of the following assignment statements using as few instructions as possible:

a. `net := gross - costs`

b. `cube := (x * x) * x`

c. `final := ((a - abase) * (b - bbase)) * (c - cbase)`

2. Investigate the `COMMON` keyword in Fortran. Describe how Fortran common blocks work and give an example. What happens if two named common blocks with the same name contain different variables? What is the difference between a blank common and a named common? What does the linker have to do to make this language construct work?