# CSE340 Fall 2019 - Homework 1 Solution

Due: Wednesday September 11 2019 by 11:59 PM on Canvas

**Problem 1.** Consider the grammar

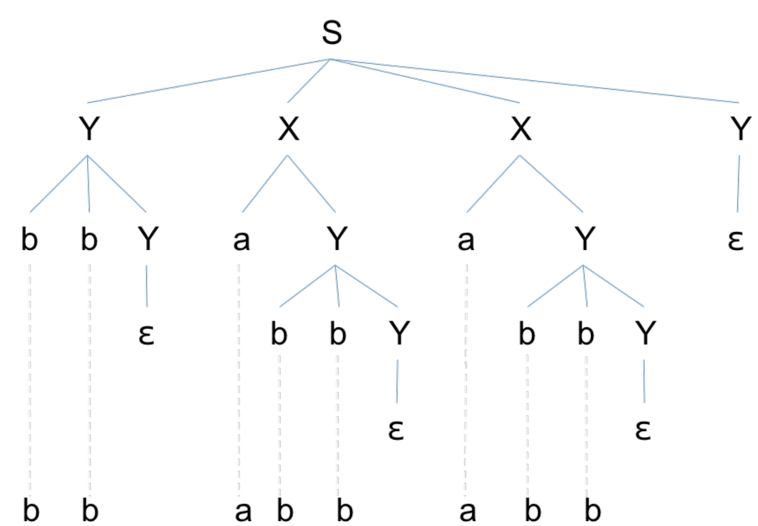
```
S → Y X X Y
X → a Y | Y
Y → b b Y | X | ε
```

where a and b are tokens. Remember that ε represent the empty string. Y → ε means that Y does not have to match any tokens. Draw a parse tree for input string (sequence of tokens):

```
bbabbabb
```

**The parse tree should have height less than or equal to 5.**

**Answer** The following parse tree satisfies the problem's requirements




I drew the input under the parse tree to show how the sequence of leaves from left to right matches the input.

The parse tree corresponds to the following leftmost derivation

S ⇒ YXXY ⇒ bbYXXY ⇒ bbXXY ⇒ bbaYXY ⇒ bbabbYXY ⇒ bbabbXY ⇒ bbabbaYY

⇒ bbabbabbYY ⇒ bbabbabbY ⇒ bbabbabb

**Problem 2.** Consider the grammar

```
S → a S b S c S
S → A
A → a S b S
A → d
```

1. What are the non-terminals?

   **Answer.** The non-terminals are `S` and `A` only. By convention, and unless otherwise noted, the non-terminals are the left-hand sides of rules.

2. What is the start symbol?

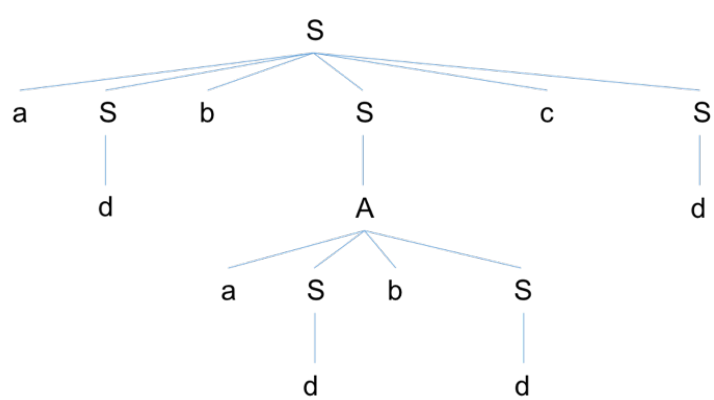
   **Answer.** The start symbol is `S`. By convention, and unless otherwise noted, the left-hand side of the first rule is the start symbol.

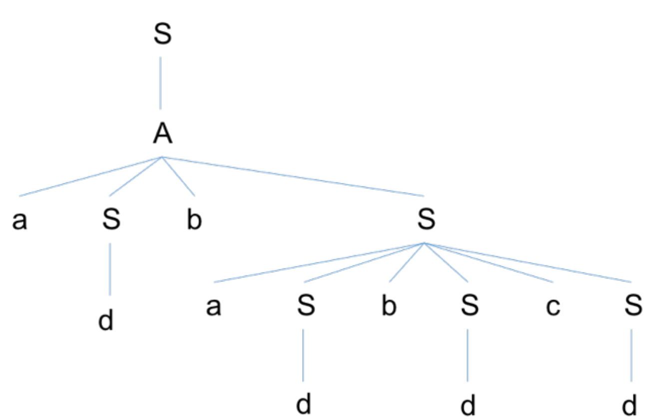
3. What are the terminals?

   **Answer.** The terminals are `a` , `b` , `c`, and `d`. By convention, and unless otherwise noted, the terminals are all the symbols that do not appear on the left-hand side of any rule.

4. Show that this grammar is ambiguous by giving a string that has two parse trees

   **Answer.** This problem can take some trial and error to find the right string. The string I will use to show that the grammar is ambiguous is the following `a d b a d b d c d`. Note that giving a string without explanation (two parse tree, two left most derivations, or two rightmost derivations is not sufficient)



**Problem 3.** Consider the grammar

S → A | B

A → a A b | c

B → b B c | D

D → d | ε

Write a recursive descent parser for this grammar. You should write the functions parse_B(), parse_S() and parse_input(), but you do not have to write the function parse_A(). You can assume that parse_A() is available and your functions can call it as needed.

I encourage you to try to write a complete parser in C++ and to execute it on a number of inputs to get a better understanding of recursive descent parsers , but that is not required and for the homework solution.

```
parse_input()
{
      parse_S();                  // parse start symbol and make sure
      t = lexer.getToken();       // that the input completely matches
      if ( t != EOF )             // the start symbol, which means that
            syntax_error();       // EOF should be found after S
}

parse_S()
{
      t = lexer.getToken();       // get and unget a token to peek at
      lexer.ungetToken(t);        // the upcoming token

      if ( t.type == a-type || t.type == c-type )    // S -> A
      {                                              // if the token is in
            parse_A();                               // FIRST(A) = { a , c }
                                                     // then we should parse
                                                     // the righthand side A

      } else if ( t.type == b-type || t.type == d-type )
                                                     // S -> B
                                                     // if the token is in
      {                                              // FIRST(B) = { b , d }
            parse_B();                               // then we should parse the
                                                     // righthand side B

      } else if ( t.type == EOF )                    // S -> B
                                                     // since epsilon is in
      {                                              // FIRST(S), we should
            parse_B();                               // check if the token is in
                                                     // FOLLOW(S) = { EOF }.
                                                     // if it is, we should parse
                                                     // the righthand side that
                                                     // can generate epsilon
                                                     // namely B

      } else                                         // otherwise, we determine
      {                                              // that there is syntax error
            syntax_error();
      }
}
```

**Note.** We can combine the conditions for righthand side B into one condition that checks for FIRST(B) and FOLLOW(S). I keep them separate to make it clearer.

```
parse_B()
{
       t = lexer.getToken();      // get and unget a token to peek at
       lexer.ungetToken(t);       // the upcoming token

       if ( t.type == b-type )                        // B -> b B c
                                                      // if the token is in
                                                      // FIRST(b B c) = { b },
       {                                              // we should parse the
                                                      // righthand side b B c

              t = getToken();                         // first, we match b
              if ( t.type != b-type )
                     syntax_error();
              parse_B();                              // then we parse B
              t = getToken();
              if ( t.type != c )                      // and finally we match c
                     syntax_error();

       } else if ( t.type == d-type )                 // B -> D
                                                      // if the token is in
       {                                              // FIRST(D) = { d }
              parse_D();                              // then we should parse the
                                                      // righthand side D

       } else if ( t.type == c-type || t.type = EOF)  // B -> D
       {                                              // since epsilon is in FIRST(B),
              parse_D();                              // we check if the token is in
                                                      // FOLLOW(B) = { c , EOF }.
                                                      // if it is, we should parse
                                                      // the righthand side that
                                                      // can generate epsilon
                                                      // namely D

       } else                                         // otherwise, we determine
       {                                              // that there is syntax error
              syntax_error();
       }
}
```

**Note.** We can combine the conditions for righthand side D into one condition that checks for FIRST(D) and FOLLOW(B). I keep them separate to make it clearer.

**For all questions, you should explain your answers.**