# CSE340 Spring 2019 - Homework 1

Due: Friday January 18 by 11:59 PM on Blackboard

**All submissions should be typed. Exception can only be made for drawing parse trees, which can be hand drawn and scanned in the submitted document.**

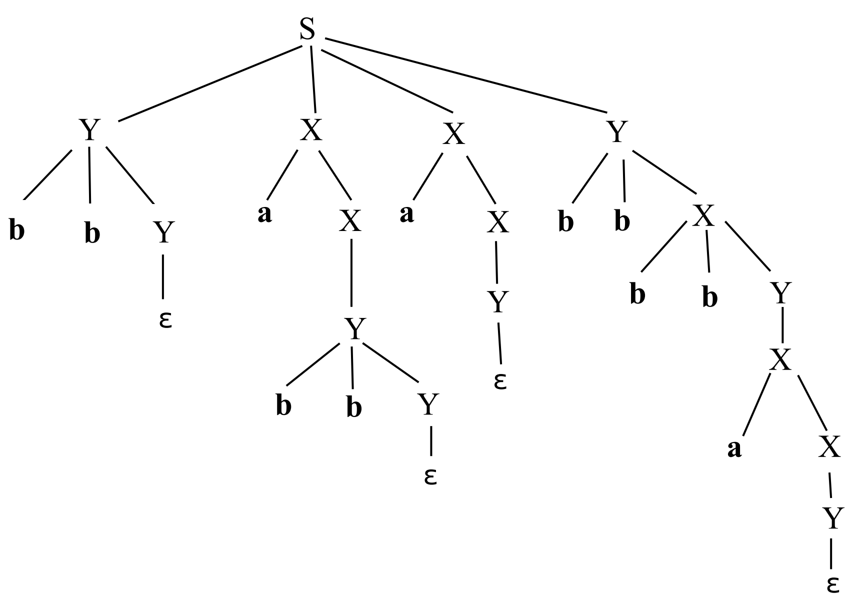
You should write your solutions in order. Solution to problem 1 should be first, then solution to problem 2, and finally solution to problem 3.

**Problem 1.** Consider the grammar

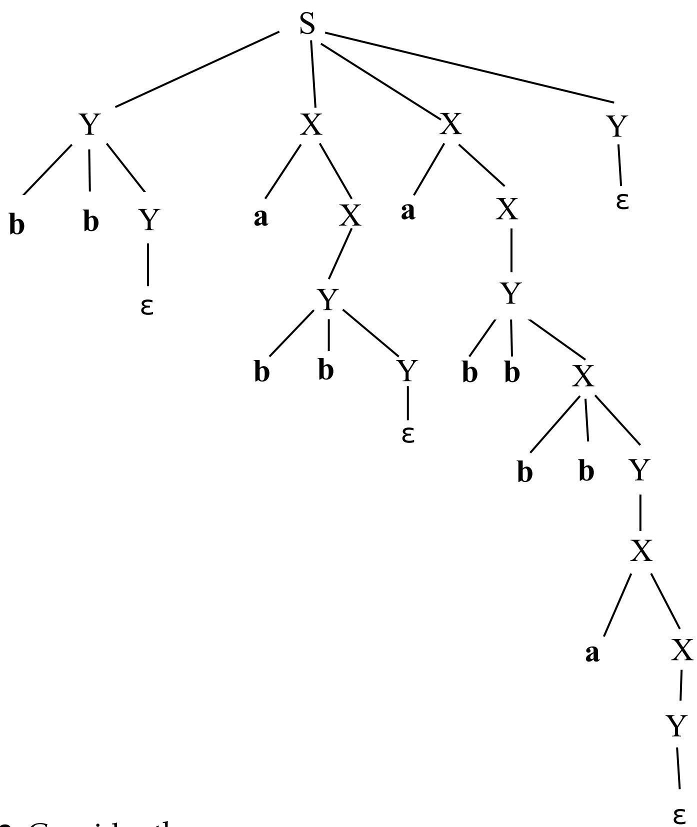
```
S → Y X X Y

X → a X | Y

Y → b b Y | X | ε
```

Draw a parse tree for input string bbabbabbbba.

**Solution 1**




**There is no unique answer to this question. Another parse tree is given on the next page.**




**Problem 2.** Consider the grammar

```
S → a S b | A
S → b S a | B
A → a b A | a b
B → b a B | b a
```

1. What are the non-terminals?

**A.** Unless otherwise noted, non-terminals are the symbols which are on the left side of the grammar rules. In the given grammar, the non-terminals are: **S, A, B**

2. What is the start symbol?

**A.** Unless otherwise specified, the start symbol is the left-hand side of the first rule of the grammar. In this grammar, the start symbol is: **S**

3. What are the terminals?

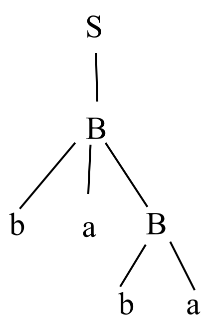
**A.** Unless otherwise specified, terminals are the symbols which are only found in the right side of a grammar rule. Therefore, the terminals are all the symbols except the non-terminals in the right side of a grammar rule. In this grammar, the terminals are: **a, b**

4. Show that this grammar is ambiguous by giving a string that starts with **b** and that has two parse trees

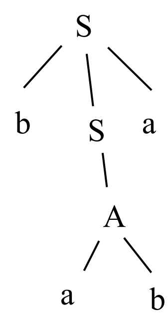
**A.** Generally, in these type of questions, one may apply a brute-force search for ambiguous strings. Start from finding smaller ambiguous string and then extend if it does not work. In this question, we take advantage of start symbol **S** appearing on the right side of the grammar rule.

We get the string: **b a b a**

**Tree 1**



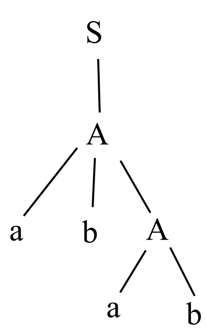

**Tree 2**



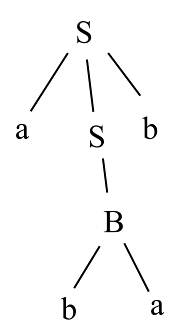

5. Show that this grammar is ambiguous by giving a string that starts with **a** and that has two parse trees

**A.** The ambiguous string: **a b a b**

**Tree 1**




**Tree 2**




**Problem 3.** Consider the grammar

```
S → A | B

A → a A b | c

B → b B | b
```

Write a recursive descent parser for this grammar. You can assume that the function `parse_B()` is already written for you. You only need to write `parse_S()` and `parse_A()`.

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 I only need the two functions `parse_S()` and `parse_A()`. For all questions, you should explain your answers.

**A.** We assume that getToken() and ungetToken() functions of lexer class are given to us.

```
parse_S()
{
    Token t = lexer.getToken();
    if(t.type == a.type || t.type == c.type) // Check for the start rule
    {
        lexer.ungetToken(t); // Unget token after we figure out which
                             // righthand side needs to be parsed because
                             // the token is part of A  or B not a
                             // terminal that appears in the righthand
                             // side
        parse_A();
        return ;
        // there is no need to check for EOF in parse_S(). We assume
        // that is done in parse_input() function which you are
        // not required to write.
        // for this grammar handling EOF in parse_S() is not a mistake
    }
    else if(t.type == b.type)
    {
        lexer.ungetToken(t);
        parse_B();
        return ;

        // same comment as above regarding EOF
    }
    else
    {
        syntax_error();      // if the token is not in FIRST(A) or
                             // FIRST(B), we have syntax error
    }
}
```

```
parse_A()
{
    Token t = lexer.getToken();
    if(t.type == a.type)
    {                 // We do not unget the token here because the token
                      // matches the terminal in the righthand side of the
                      // rule A -> a A b and this is the rule we are
                      // trying to parse.
        parse_A();    // Call Parse_A() after we encounter "a"
                      // at this point we have seen a A of the
                      // rihghtand side a A b
                      // only b remains to be seen and we successfully
                      // finish parsing a A b
        Token t1 = lexer.getToken();
        if(t1.type == b.type)
        {
            Return;
        }
        else
        {
            syntax_error();
        }
    }
    else if(t.type == c.type)
    {
        Return;
    }
    else
    {
        syntax_error();
    }
}
```

Note : the code above either returns successfully or throws a syntax error. Alternatively, we could have had the parse functions return a Boolean value (true or false). In grading, we will count both as correct.

The following function was not required for the homework but I include it for completeness.

```
Parse_B()
{
    Token t = lexer.getToken();
    if(t.type == b.type)
    {
        Token t1 = lexer.getToken();
        If (t1.type == b.type)       // Check whether first or second
                                     // rule applies
        {
            lexer.ungetToken(t1);
            Parse_B();
            Return;
        }
        else if(t1.type == EOF)      // if the rule is simply B -> b
                                     // the token that we should get
                                     // should be in FOLLOW(B) = {EOF}
        {
            lexer.ungetToken(t1);
            return ;
        }
        else
        {
            syntax_error();
        }
    }
    else
    {
        syntax_error();
    }
}
```