Assignment 4
Parser Implementation

Note: All code must be compilable and executable on Windows systems.
Code that does not compile will receive 0 marks. There will be no evaluations. C/C++ and JAVA will be accepted.

In the previous assignment, you designed a parser for the language JAVA--. Your task now is to implement it.

The Parser shall take the file *words.txt* generated by the Lex, and analyze its syntax according to the JAVA-- grammar. In the case of incorrect syntax, it shall gracefully halt execution and print an appropriate error message. In the case of completely correct syntax, it shall produce a parse tree of the code (a sample given in *below*). It shall also add to the symbol table the datatypes of all identifiers.

Following is a sample program written in c--.

```
int numPrint (int num, int length)
{
        int i, j, first, temp;
        char a;
        a <- ‘x’;
        jOut ("enter number");
        jIn (i);
        jOut(i);
        i <- length;
        while (i > 0)
        {
                first<- 0;              /*this line contains a comment*/
                j <-1;
                while (j < i)
                {
                        jOut( j);
                        j <- j + 1;
                }
                /* this is a comment */
                i<- i - 1;
                /*This is a
                Multiline
                Comment*/
        }
        jOut( “temp is “);
        jOut( temp);
        return i;
}
```

The language contains the following elements:

data types: int char

Keywords: if else while return jIn jOut

arithmetic operators: + - * /

relational operators: < <= > >= == !=

comments: /* enclose comment in */
identifier: a letter followed by any number of letters or digits
numeric constants: only integers
literal constants: a letter enclosed in single quotes
strings: no need to store as variables, only used in print statements
parenthesis, braces, square brackets
assignment operator <-
semi colon
colon
comma

Sample Parse Tree:

```
X
|__X
|__X
:    |__X
:    |__X
:    :    |__^
|__ID (sum)
|__X
:    |__X
:    |__X
:    :    |__ID (i)
:    :    |__A'
:    :    :    |__COMMA
:    :    :    |__D
:    :    :    :    |__INT

...


:    :    :    :    :    |__X
:    :    :    :    :    :    |__^
:    :    :    |__X
:    :    :    :    |__^
:    :    |__SEMICOLON
:    |__X
|__X
:    |__^
```

Your parse tree will have the appropriate tokens in the places marked X.

Requirements:

- A test file that successfully runs your compiler (comprising of both lex and parser). Name this *test.cmm*

- Complete (previous) code of your lex
- Complete code of your Parser, which will generate the following text files:
- *parsetree.txt*: a parse tree for any given test.cmm
  The parse tree should print the names of all functions visited, as well as all tokens and non-null lexemes. It should clearly illustrate function depth using whitespace and non-alphabetic characters.
- *parser-symboltable.txt*: the names and datatypes of all the identifiers

This will be a continuation of Phase 2. The input will still be a .cmm file containing JAVA-- code. Your compiler must run the lex first, and use the freshly generated output to subsequently run the parser. Do not submit an isolated parser with static *words.txt* and *symboltable.txt* files. Such submissions will not be checked.

Interface:

The program will take file name like test.cmm as input and run the whole program.

Make sure you follow all given instructions, and name the files as instructed.

Submission Instructions:
Submit files unzipped
Do not submit executables.

*This deliverable will be marked without an evaluation, so make sure the code is compilable **ON WINDOWS**.*

There will be zero tolerance for plagiarism. Your assignments will be checked far more thoroughly than you are anticipating. Once detected, no appeals for removal of plagiarism will be entertained.