Project Report
Software Project Lab 1
SE 305

A SMALL SCALE COMPILER

Submitted By:
Name: Ahmed Ryan
Roll No: BSSE-1011
Session: 2017-18

Supervised By:
Kishan Kumar Ganguly
Lecturer
Institute of Information Technology,
University of Dhaka

Table of Contents:

Introduction:

A compiler is a software program that transforms high-level source code that is written by a developer in a high-level programming language into a low level object code (binary code) in machine language, which can be understood by the processor. The process of converting high-level programming into machine language is known as compilation.

The processor executes object code, which indicates when binary high and low signals are required in the arithmetic logic unit of the processor.

A compiler executes four major steps:

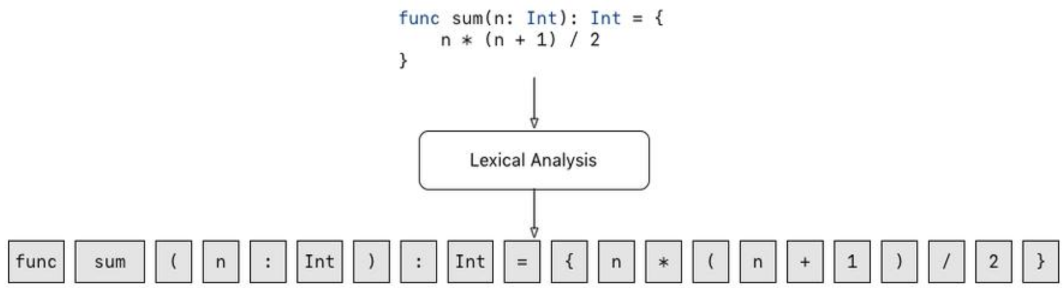
- **Scanning**: The scanner reads one character at a time from the source code and keeps track of which character is present in which line.
- **Lexical Analysis**: The compiler converts the sequence of characters that appear in the source code into a series of strings of characters (known as tokens), which are associated by a specific rule by a program called a lexical analyzer. A symbol table is used by the lexical analyzer to store the words in the source code that correspond to the token generated.
- **Syntactic Analysis**: In this step, syntax analysis is performed, which involves preprocessing to determine whether the tokens created during lexical analysis are in proper order as per their usage. The correct order of a set of keywords, which can yield a desired result, is called syntax. The compiler has to check the source code to ensure syntactic accuracy.
- **Semantic Analysis**: This step is comprised of several intermediate steps. First, the structure of tokens is checked, along with their order with respect to the grammar in a given language. The meaning of the token structure is interpreted by the parser and analyzer to finally generate an intermediate code, called object code. The object code includes instructions that represent the processor action for a corresponding token when encountered in the program. Finally, the entire code is parsed and interpreted to check if any optimizations are possible.

Background Study:

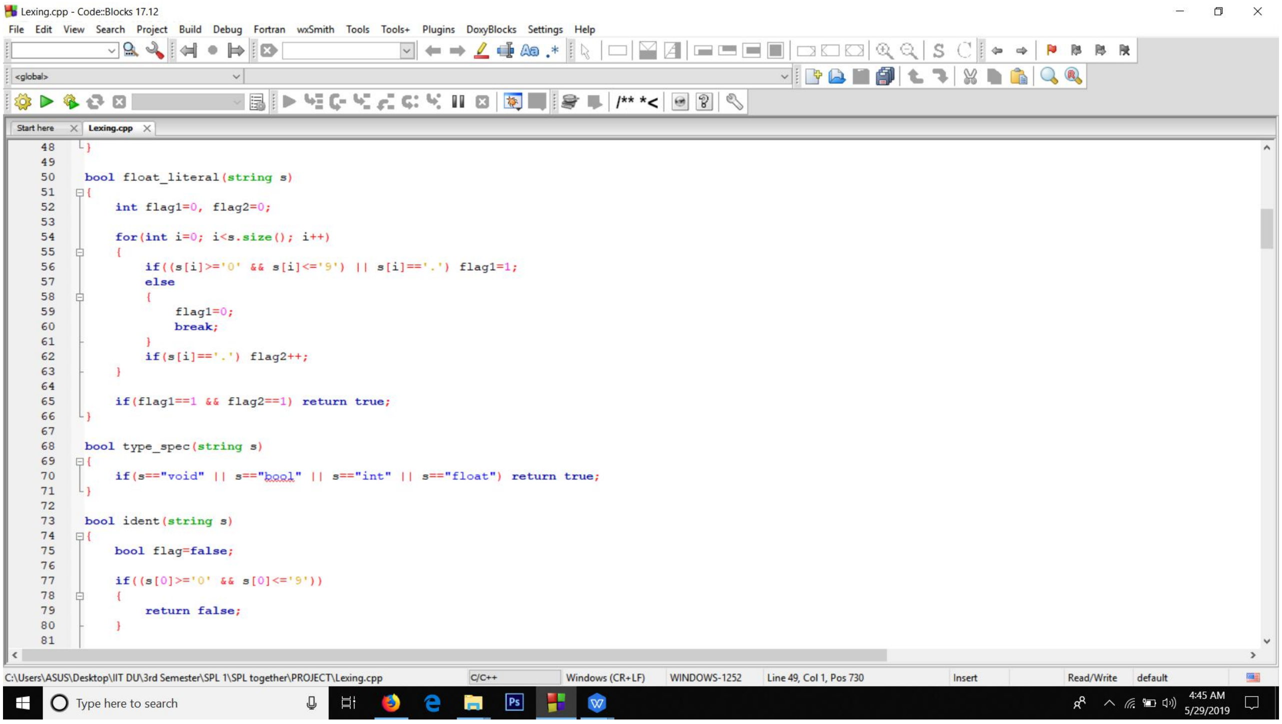
Lexical Analysis:

The first phase of the compiler is the *lexical analysis*. In this phase, the compiler breaks the submitted source code into meaningful elements called **lexemes** and generates a sequence of **tokens** from the lexemes.

A *token* is an object describing a *lexeme*. Along with the value of the *lexeme* (the actual string of characters of the lexeme), it contains information such as its type (*is it a keyword? an identifier? an operator?* …) and the position (line and/or column number) in the source code where it appears.

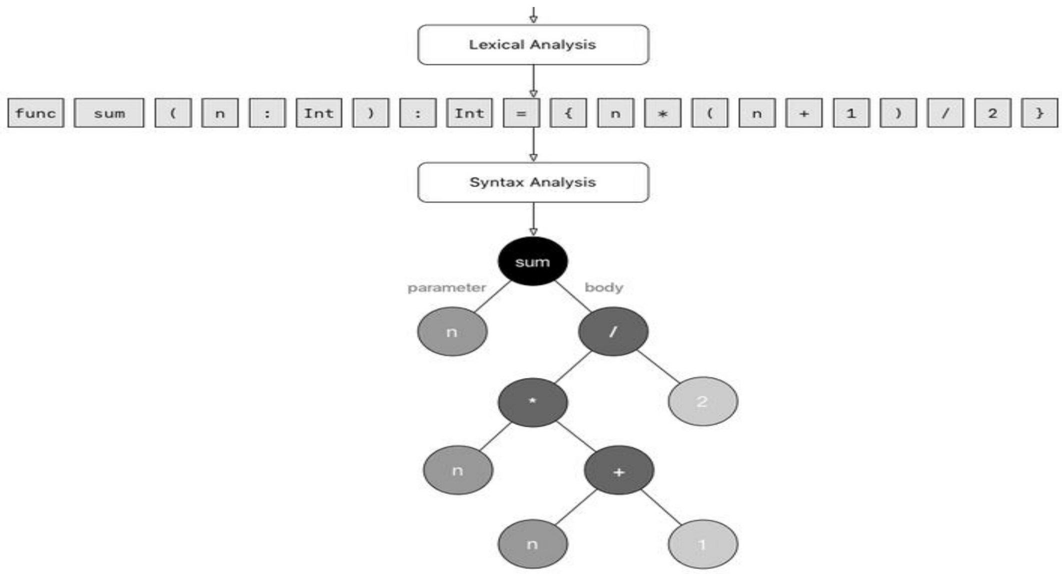


Sequence of lexemes generated during lexical analysis

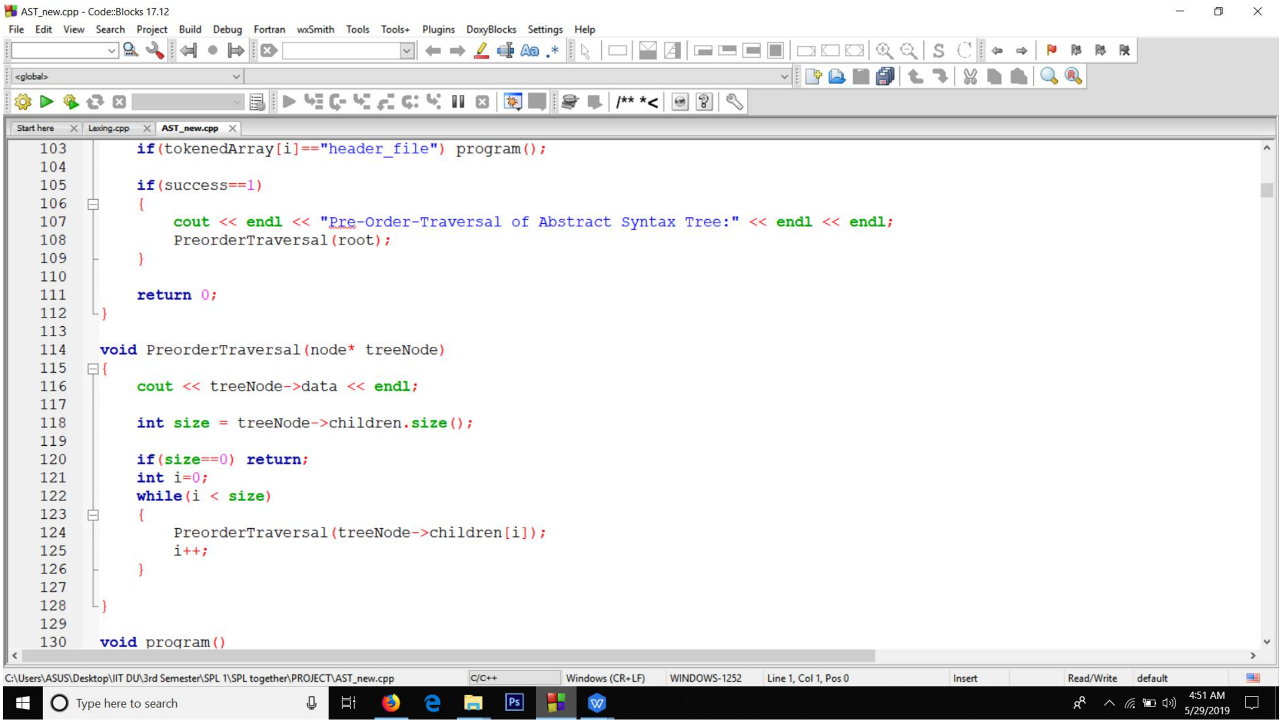


Syntax Analysis

During syntax analysis, the compiler uses the sequence of *tokens* generated during the lexical analysis to generate a tree-like data structure called **Abstract Syntax Tree**, **AST** for short. The *AST* reflects the syntactic and logical structure of the program.



Abstract Syntax Tree generated after syntax analysis



Syntax analysis is also the phase where eventual syntax errors are detected and reported to the user in the form of informative messages. For instance, in the example above, if we forget the closing brace } after the definition of the sum function, the compiler should return an error stating that there is a missing } and the error should point to the line and column where the } is missing.

Challenges:

New Challenges are always faced while implementing a software solution. While Implementing the lexical analyser and the abstract syntax tree, a lot of challenges were faced namely:

1. Parsing was new to me and parsing a code was very difficult in the first place.
2. Parsing character by character was a very labourios job.
3. Dividing the entire source code into tokens was a bit difficult.
4. Learning the Context Free Grammar took so much time.
5. Implementing the production rules was the most difficult of all.
6. Lack of resources in the internet gave a lot of difficulties.
7. Representing the code into an abstract Syntax Tree was very confusing and it took a lot of time.

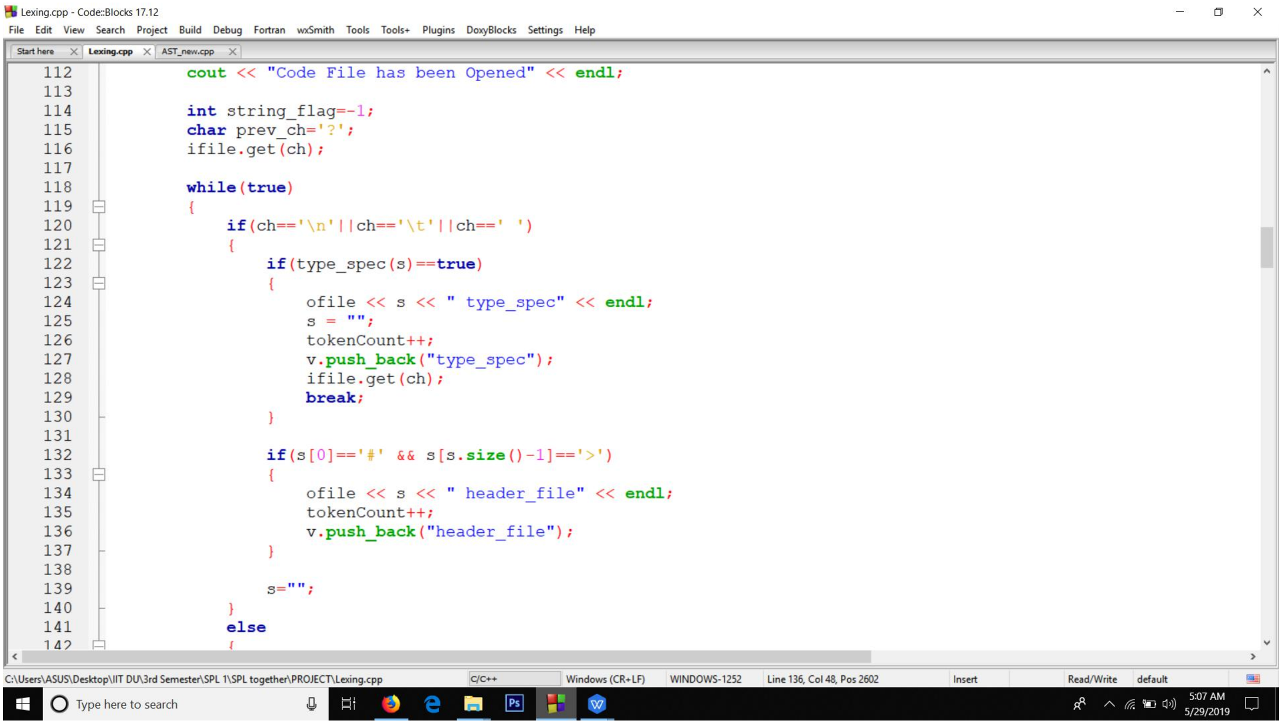
Project Overview:

The project has mainly two steps:

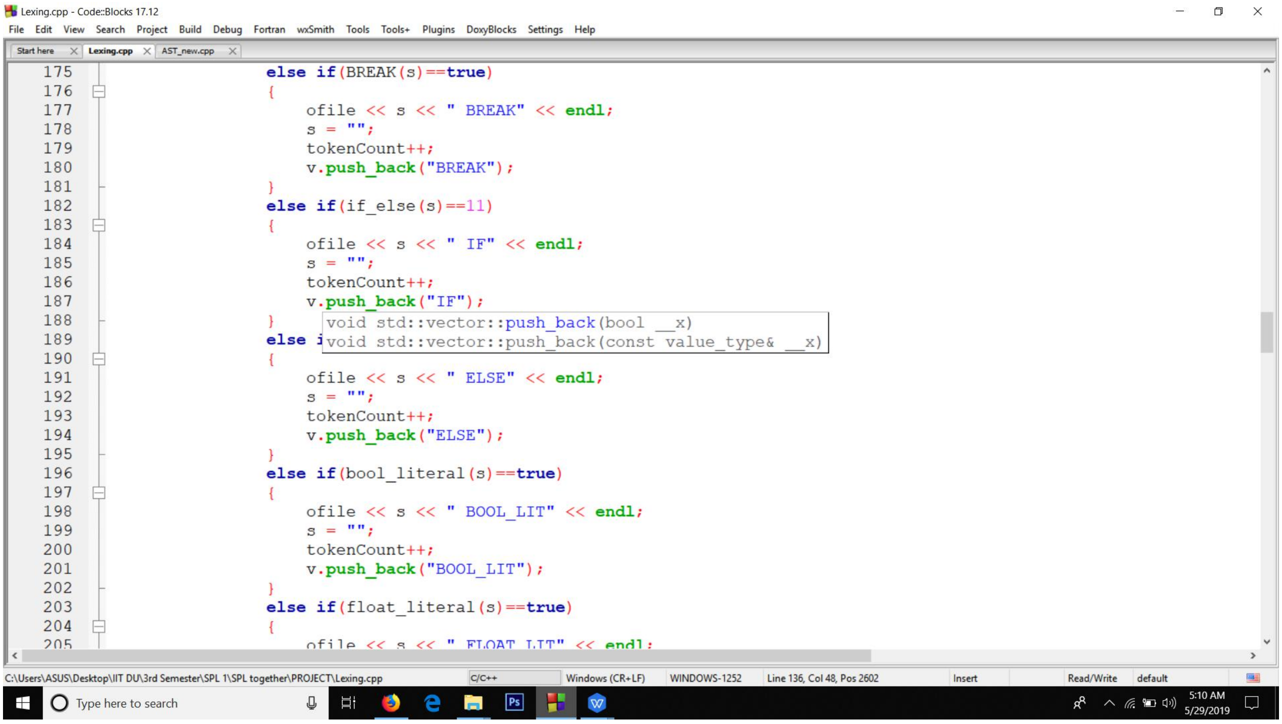
1. Lexical Analysis
2. Syntactical Analysis

Lexical Analysis:

The source code is first converted into tokens so that the syntactical analysis can be done on the Lexed Code. Several Steps were taken during these time. Some of the steps are illustrated below:



Here, we can see type specifiers and header files are being lexed from the source code and delimeters endline, tab and space are being used for the cause

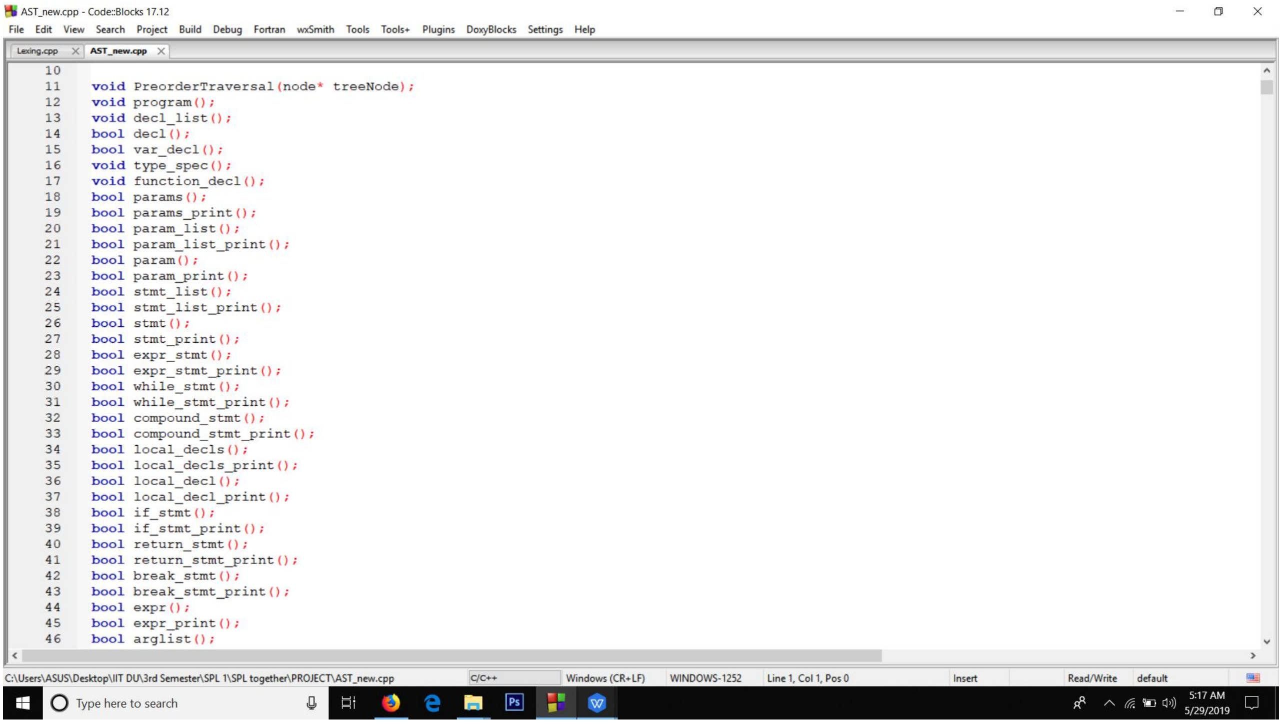


Here, we have detected break, if, else, boolean literals and float literals and converted them into tokens for further use.

Next let us show about the syntactical analysis of the lexed code.

Syntactical Analysis:

The lexed code will be analysed syntactically and an abstract syntax tree will be formed with the tokens. Any irregularities with the Context Free Grammar will be determined from here. The prototypes of functions used in the Analyser will be shown below whose activities can be guessed from their names.



```
void PreorderTraversal(node* treeNode);
void program();
void decl_list();
bool decl();
bool var_decl();
void type_spec();
void function_decl();
bool params();
bool params_print();
bool param_list();
bool param_list_print();
bool param();
bool param_print();
bool stmt_list();
bool stmt_list_print();
bool stmt();
bool stmt_print();
bool expr_stmt();
bool expr_stmt_print();
bool while_stmt();
bool while_stmt_print();
bool compound_stmt();
bool compound_stmt_print();
bool local_decls();
bool local_decls_print();
bool local_decl();
bool local_decl_print();
bool if_stmt();
bool if_stmt_print();
bool return_stmt();
bool return_stmt_print();
bool break_stmt();
bool break_stmt_print();
bool expr();
bool expr_print();
bool arglist();
```

These are the functions used in the Syntactical Analyser. Now the **Context Free Grammar** whose help has been taken in building the Abstract Syntax Tree will be illustrated below.

```
program                   → decl_list
decl_list                 → decl_list decl | decl
decl                      → var_decl | fun_decl
var_decl                  → type_spec IDENT  ; | type_spec IDENT [ ] ;
type_spec                 → VOID | BOOL | INT | FLOAT
fun_decl                  → type_spec IDENT ( params ) compound_stmt
params                    → param_list | VOID
param_list                → param_list , param | param
param                     → type_spec IDENT | type_spec IDENT [ ]
stmt_list                 → stmt_list stmt | ε
stmt                      → expr_stmt | compound_stmt | if_stmt | while_stmt |
                                  return_stmt | break_stmt
expr_stmt                 → expr ; | ;
while_stmt                → WHILE ( expr ) stmt
compound_stmt    → { local_decls stmt_list }
local_decls               → local_decls local_decl | ε
local_decl                → type_spec IDENT ; | type_spec IDENT [ ] ;
if_stmt                   → IF ( expr ) stmt | IF ( expr ) stmt ELSE stmt
return_stmt               → RETURN ; | RETURN expr ;

The following expressions are listed in order of increasing precedence:
expr                      → IDENT = expr | IDENT [ expr ] = expr
                                  → expr OR expr
                                  → expr EQ expr | expr NE expr
                                  → expr LE expr | expr < expr | expr GE expr  | expr > expr
                                  → expr AND expr
                                  → expr + expr | expr - expr
                                  → expr * expr | expr / expr  | expr % expr
                                  → ! expr | - expr | + expr
                                  → ( expr )
                                  → IDENT | IDENT [ expr ] | IDENT ( args ) | IDENT . size
                                  → BOOL_LIT | INT_LIT  | FLOAT_LIT | NEW type_spec [ expr ]

arg_list                  → arg_list , expr | expr
args                      → arg_list | ε
```

Some significant functions of the code is shown below and explained briefly.

```
void PreorderTraversal(node* treeNode)
{
    cout << treeNode->data << endl;

    int size = treeNode->children.size();

    if(size==0) return;
    int i=0;
    while(i < size)
    {
        PreorderTraversal(treeNode->children[i]);
        i++;
    }

}
```

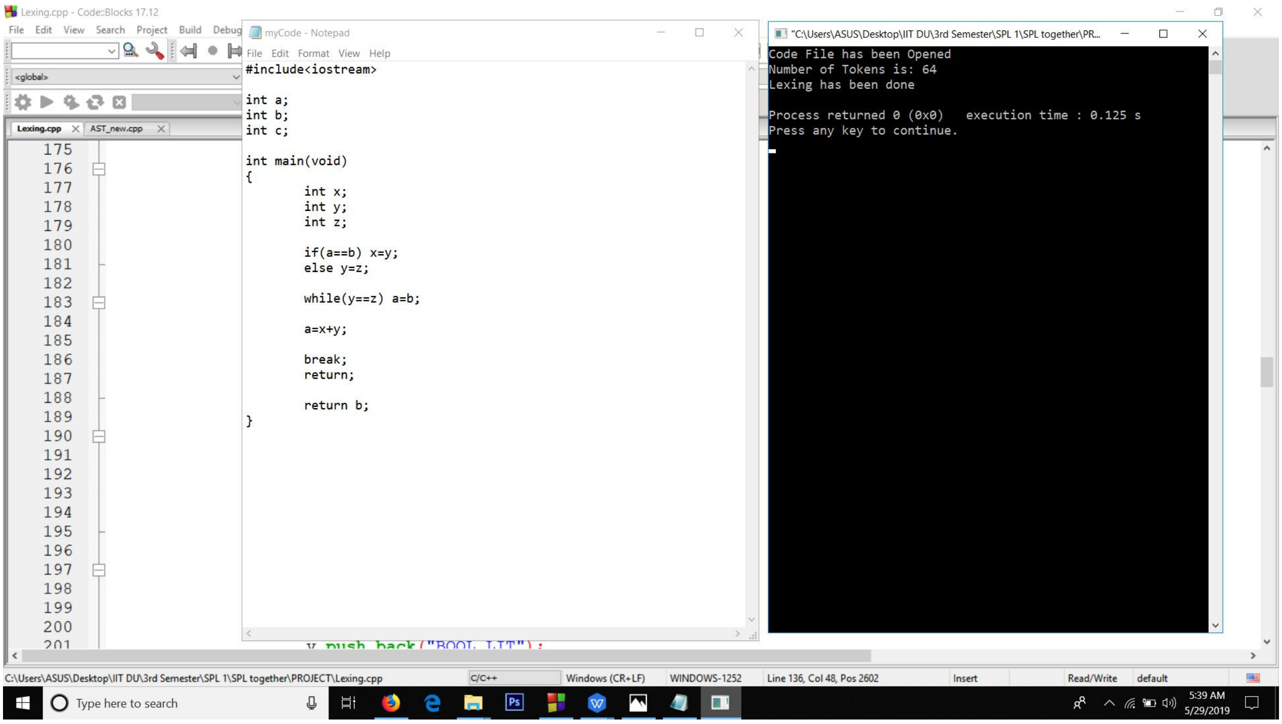
The abstract Syntax Tree is being traversed by the following code which denotes the preorderTraversal of the tree.

```
Lexing.cpp  AST_new.cpp
    return;
}


void function_decl()
{
    int marker=i;

    if(params()==true && compound_stmt()==true)
    {
        node* temp = create_newNode("function_decl");
        vector <node*> tempVec;
        tempVec.push_back(temp);
        leaf->children=tempVec;
        temp->parent=leaf;

        leaf = tempVec[0];
        function_decl_leaf = leaf;

        //cout << "function_decl" << endl;

        i=marker;
        params_print();
        compound_stmt_print();

        success=1;

        return;
    }
    else
    {
        cout << "**THE CODE DOES NOT FOLLOW THE CONTEXT FREE GRAMMAR" << endl;
        return;
    }

}
```

This segment of the code is used to detect a function declaration from the lexed code.

USER MANUAL:

There are two parts of the compiler which have been implemented. The two codes are separated for better understanding. The Lexical Analyser turns the Source Code into a String of tokens.



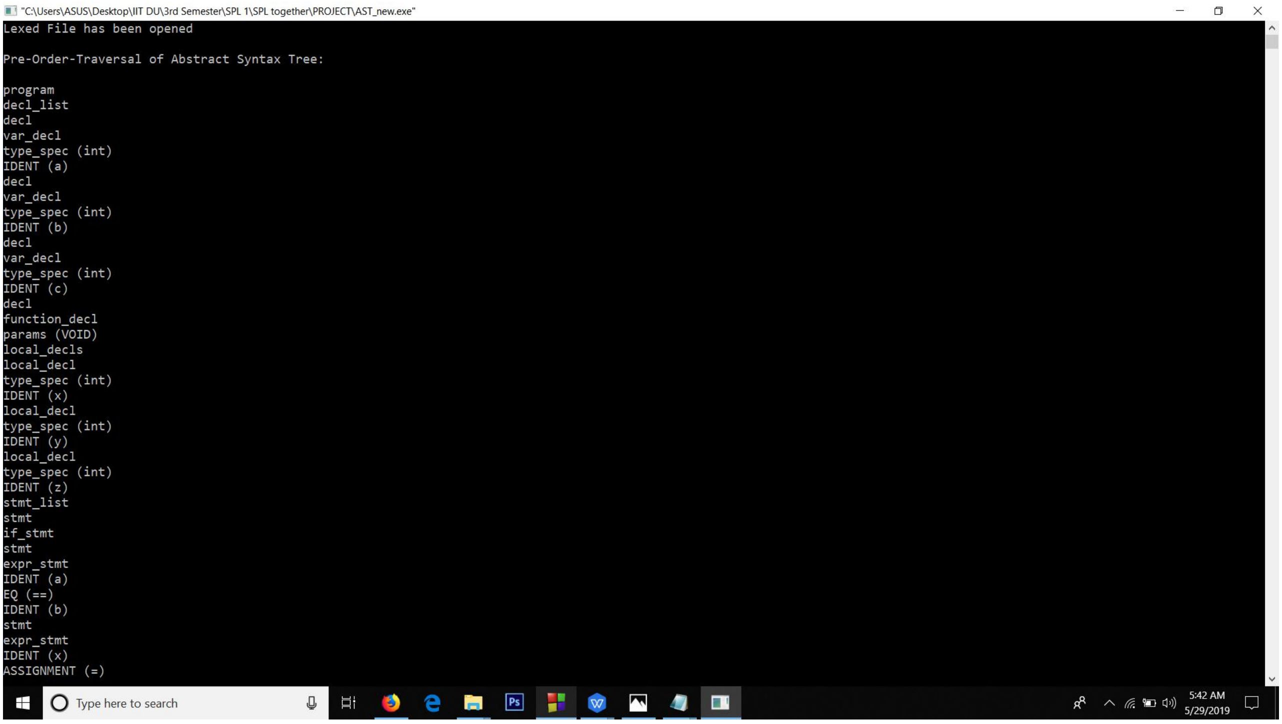
After the lexing of the user Code into lexed file we get the following file.

Lexing - Notepad

File Edit Format View Help

```
#include<iostream> header_file
int type_spec
a IDENT
; SEMICOLON
int type_spec
b IDENT
; SEMICOLON
int type_spec
c IDENT
; SEMICOLON
int type_spec
main IDENT
( FIRST_BRACKET_OPEN
void VOID
) FIRST_BRACKET_CLOSE
{ SECOND_BRACKET_OPEN
int type_spec
x IDENT
; SEMICOLON
int type_spec
y IDENT
; SEMICOLON
int type_spec
z IDENT
; SEMICOLON
if IF
( FIRST_BRACKET_OPEN
a IDENT
== EQ
b IDENT
) FIRST_BRACKET_CLOSE
x IDENT
= ASSIGNMENT
y IDENT
; SEMICOLON
else ELSE
y IDENT
= ASSIGNMENT
z IDENT
; SEMICOLON
while WHILE
```

Then these tokens are constructed into an AST which is shown below:



Therefore, the abstract syntax tree has been constructed in the above stated manner.

Conclusion:

The project helped me understand how the compiler works and it has helped me so much in a number of ways. It taught me how to be more manageable while coding and how to handle large lines of code. . This project was quiet challenging as I didn’t have any knowledge about computer compilers before doing this project and I gained a lot of experience from it. Hope this knowledge will help me in my future compiler and parsing related projects. I want to thank my supervisor for guiding me a lot during this project.

APPENDIX:

I could complete upto Syntactical Analysis. I have plans for this project in the future. I would like to generate the object code from the source code and also detect errors from the source code.

Reference:

1.Definitions:
https://hackernoon.com/compilers-and-interpreters-3e354a2e41cf
2.Some photos:
https://hackernoon.com/compilers-and-interpreters-3e354a2e41cf
3.More Definitions:
https://www.techopedia.com/definition/3912/compiler