



# **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:

1.Introduction .....	1
1.1. Background Study.....	2-4
1.2. Challenges.....	4
2.Project Overview .....	4-8
3.User Manual .....	8-12
4.Conclusion .....	12
5.Appendix.....	13
6.References.....	13

## 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:

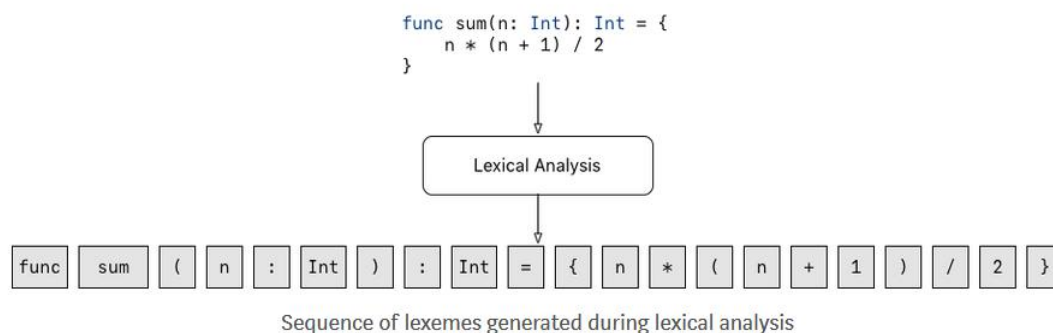
- **Scanning:** The scanner reads one character at a time from the source code and keeps track of which character is present in which line.
- **Lexical Analysis:** The compiler converts the sequence of characters that appear in the source code into a series of strings of characters (known as tokens), 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.



```

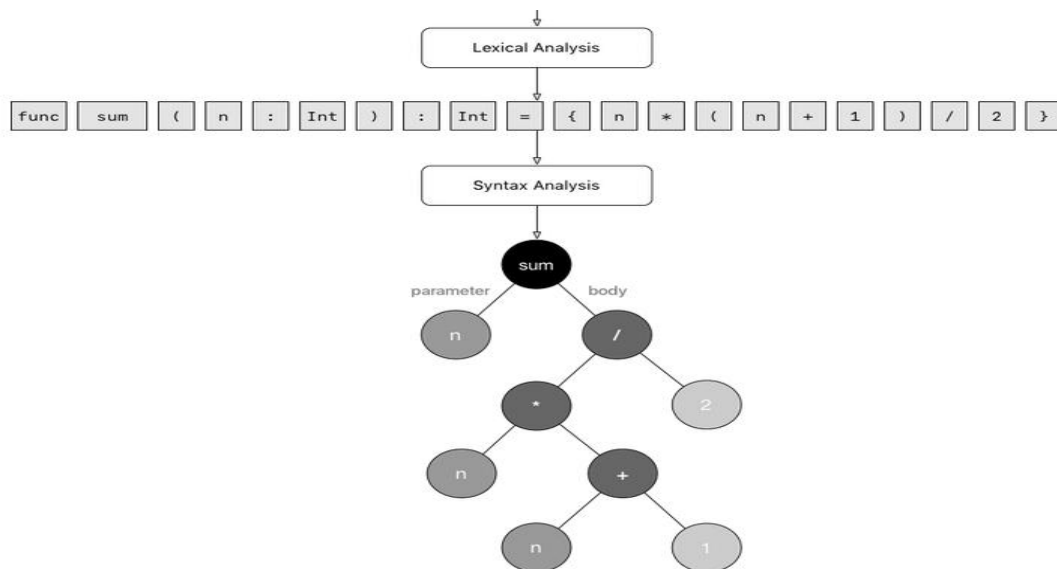
Lexing.cpp - Code::Blocks 17.12
File Edit View Search Project Build Debug Fortran wxSmith Tools Tools+ Plugins DovyBlocks Settings Help
<global>
Start here Lexing.cpp
48
49
50 bool float_literal(string s)
51 {
52     int flag1=0, flag2=0;
53     for(int i=0; i<s.size(); i++)
54     {
55         if((s[i]>='0' && s[i]<='5') || s[i]=='.' || s[i]=='e') flag1=1;
56         else
57         {
58             flag1=0;
59             break;
60         }
61         if(s[i]=='.' || s[i]=='e') flag2++;
62     }
63     if(flag1==1 && flag2==1) return true;
64 }
65
66 bool type_spec(string s)
67 {
68     if(s=="void" || s=="bool" || s=="int" || s=="float") return true;
69 }
70
71 bool ident(string s)
72 {
73     bool flag=false;
74     if((s[0]>='0' && s[0]<='9'))
75     {
76         return false;
77     }
78 }
79
80
81

```

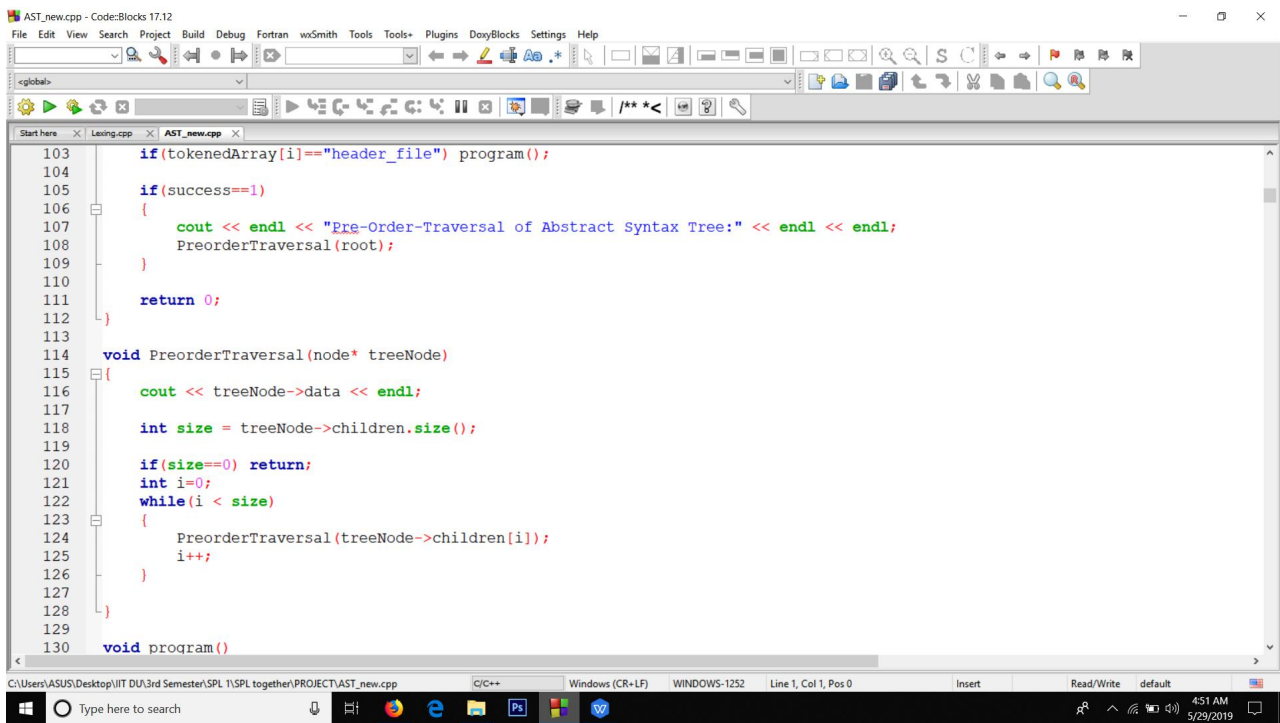
C:\Users\ASUS\Desktop\IIT DU\3rd Semester\SPL 1\SPL together\PROJECT\Lexing.cpp C/C++ Windows (CR+LF) WINDOWS-1252 Line 49, Col 1, Pos 730 Insert Read/Write default 4:45 AM 5/29/2019

## 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



```
103     if(tokenedArray[i]=="header_file") program();
104
105     if(success==1)
106     {
107         cout << endl << "Pre-Order-Traversal of Abstract Syntax Tree:" << endl << endl;
108         PreorderTraversal(root);
109     }
110
111     return 0;
112 }
113
114 void PreorderTraversal(node* treeNode)
115 {
116     cout << treeNode->data << endl;
117
118     int size = treeNode->children.size();
119
120     if(size==0) return;
121     int i=0;
122     while(i < size)
123     {
124         PreorderTraversal(treeNode->children[i]);
125         i++;
126     }
127 }
128
129
130 void program()
```

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 labourious 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:

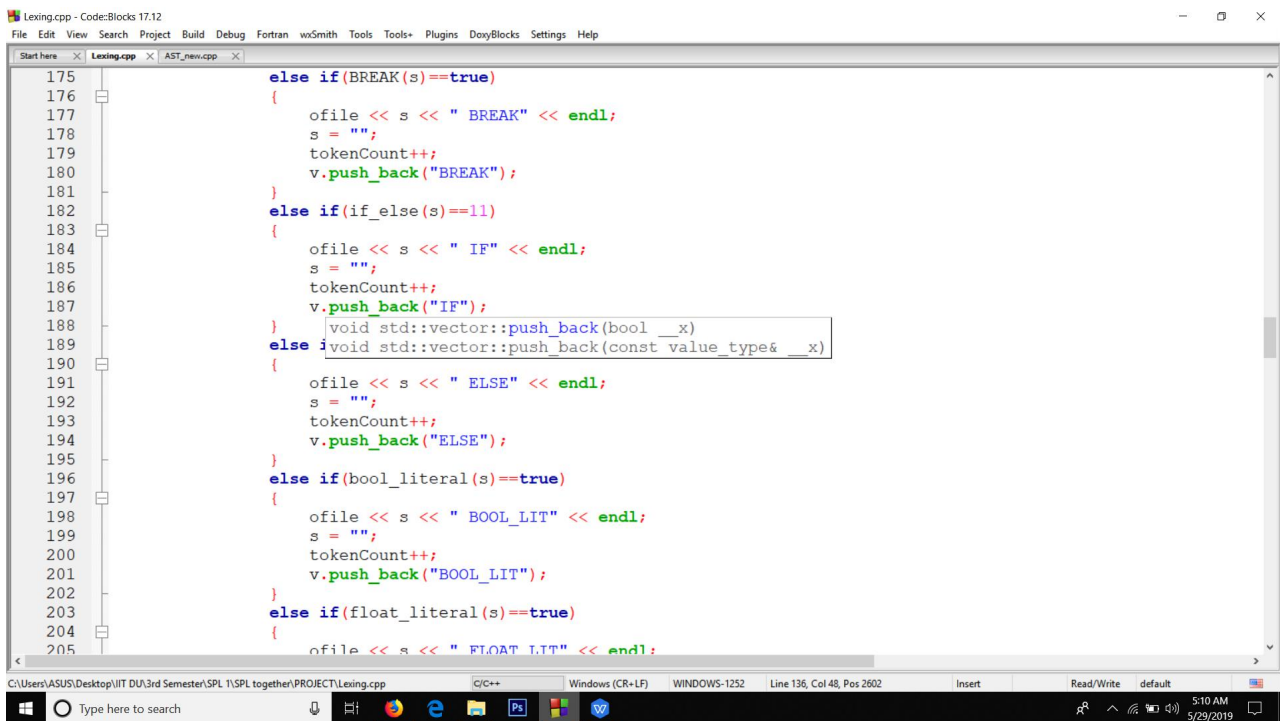


The screenshot shows a Code::Blocks IDE window titled 'Lexing.cpp - Code::Blocks 17.12'. The code is written in C++ and implements a lexical analyzer. It reads characters from a file and identifies tokens based on specific rules. The code is as follows:

```
112     cout << "Code File has been Opened" << endl;
113
114     int string_flag=-1;
115     char prev_ch='?';
116     ifile.get(ch);
117
118     while(true)
119     {
120         if(ch=='\n' || ch=='\t' || ch==' ')
121         {
122             if(type_spec(s)==true)
123             {
124                 ofile << s << " type_spec" << endl;
125                 s = "";
126                 tokenCount++;
127                 v.push_back("type_spec");
128                 ifile.get(ch);
129                 break;
130             }
131
132             if(s[0]=='#' && s[s.size()-1]=='>')
133             {
134                 ofile << s << " header_file" << endl;
135                 tokenCount++;
136                 v.push_back("header_file");
137             }
138
139             s="";
140         }
141         else
142         {
```

The status bar at the bottom indicates the file path: 'C:\Users\ASUS\Desktop\UIT DU\3rd Semester\SPL 1\SPL together\PROJECT\Lexing.cpp'. The window title bar shows 'C/C++', 'Windows (CR+LF)', 'WINDOWS-1252', 'Line 136, Col 48, Pos 2602', 'Insert', 'Read/Write', 'default', and the system clock '5:07 AM 5/29/2019'.

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



```
175     else if(BREAK(s)==true)
176     {
177         ofile << s << " BREAK" << endl;
178         s = "";
179         tokenCount++;
180         v.push_back("BREAK");
181     }
182     else if(if_else(s)==11)
183     {
184         ofile << s << " IF" << endl;
185         s = "";
186         tokenCount++;
187         v.push_back("IF");
188     }
189     else if(void_std::vector::push_back(const value_type& __x)
190     {
191         ofile << s << " ELSE" << endl;
192         s = "";
193         tokenCount++;
194         v.push_back("ELSE");
195     }
196     else if(bool_literal(s)==true)
197     {
198         ofile << s << " BOOL_LIT" << endl;
199         s = "";
200         tokenCount++;
201         v.push_back("BOOL_LIT");
202     }
203     else if(float_literal(s)==true)
204     {
205         ofile << s << " FLOAT_LIT" << endl;
```

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.



```

10
11 void PreorderTraversal(node* treeNode);
12 void program();
13 void decl_list();
14 bool decl();
15 bool var_decl();
16 void type_spec();
17 void function_decl();
18 bool params();
19 bool params_print();
20 bool param_list();
21 bool param_list_print();
22 bool param();
23 bool param_print();
24 bool stmt_list();
25 bool stmt_list_print();
26 bool stmt();
27 bool stmt_print();
28 bool expr_stmt();
29 bool expr_stmt_print();
30 bool while_stmt();
31 bool while_stmt_print();
32 bool compound_stmt();
33 bool compound_stmt_print();
34 bool local_decls();
35 bool local_decls_print();
36 bool local_decl();
37 bool local_decl_print();
38 bool if_stmt();
39 bool if_stmt_print();
40 bool return_stmt();
41 bool return_stmt_print();
42 bool break_stmt();
43 bool break_stmt_print();
44 bool expr();
45 bool expr_print();
46 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.

1	program	→ decl_list
2	decl_list	→ decl_list decl   decl
3	decl	→ var_decl   fun_decl
4	var_decl	→ type_spec IDENT ;   type_spec IDENT [ ] ;
5	type_spec	→ VOID   BOOL   INT   FLOAT
6	fun_decl	→ type_spec IDENT ( params ) compound_stmt
7	params	→ param_list   VOID
8	param_list	→ param_list , param   param
9	param	→ type_spec IDENT   type_spec IDENT [ ]
10	stmt_list	→ stmt_list stmt   ε
11	stmt	→ expr_stmt   compound_stmt   if_stmt   while_stmt   return_stmt   break_stmt
12	expr_stmt	→ expr ;   ;
13	while_stmt	→ WHILE ( expr ) stmt
14	compound_stmt	→ { local_decls stmt_list }
15	local_decls	→ local_decls local_decl   ε
16	local_decl	→ type_spec IDENT ;   type_spec IDENT [ ] ;
17	if_stmt	→ IF ( expr ) stmt   IF ( expr ) stmt ELSE stmt
18	return_stmt	→ RETURN ;   RETURN expr ;
21	The following expressions are listed in order of increasing precedence:	
22	expr	→ IDENT = expr   IDENT [ expr ] = expr
23		→ expr OR expr
24		→ expr EQ expr   expr NE expr
25		→ expr LE expr   expr < expr   expr GE expr   expr > expr
26		→ expr AND expr
27		→ expr + expr   expr - expr
28		→ expr * expr   expr / expr   expr % expr
29		→ ! expr   - expr   + expr
30		→ ( expr )
31		→ IDENT   IDENT [ expr ]   IDENT ( args )   IDENT . size
32		→ BOOL_LIT   INT_LIT   FLOAT_LIT   NEW type_spec [ expr ]
33		
34	arg_list	→ arg_list , expr   expr
35	args	→ arg_list   ε

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

```
113
114 void PreorderTraversal(node* treeNode)
115 {
116     cout << treeNode->data << endl;
117
118     int size = treeNode->children.size();
119
120     if(size==0) return;
121     int i=0;
122     while(i < size)
123     {
124         PreorderTraversal(treeNode->children[i]);
125         i++;
126     }
127 }
128
```

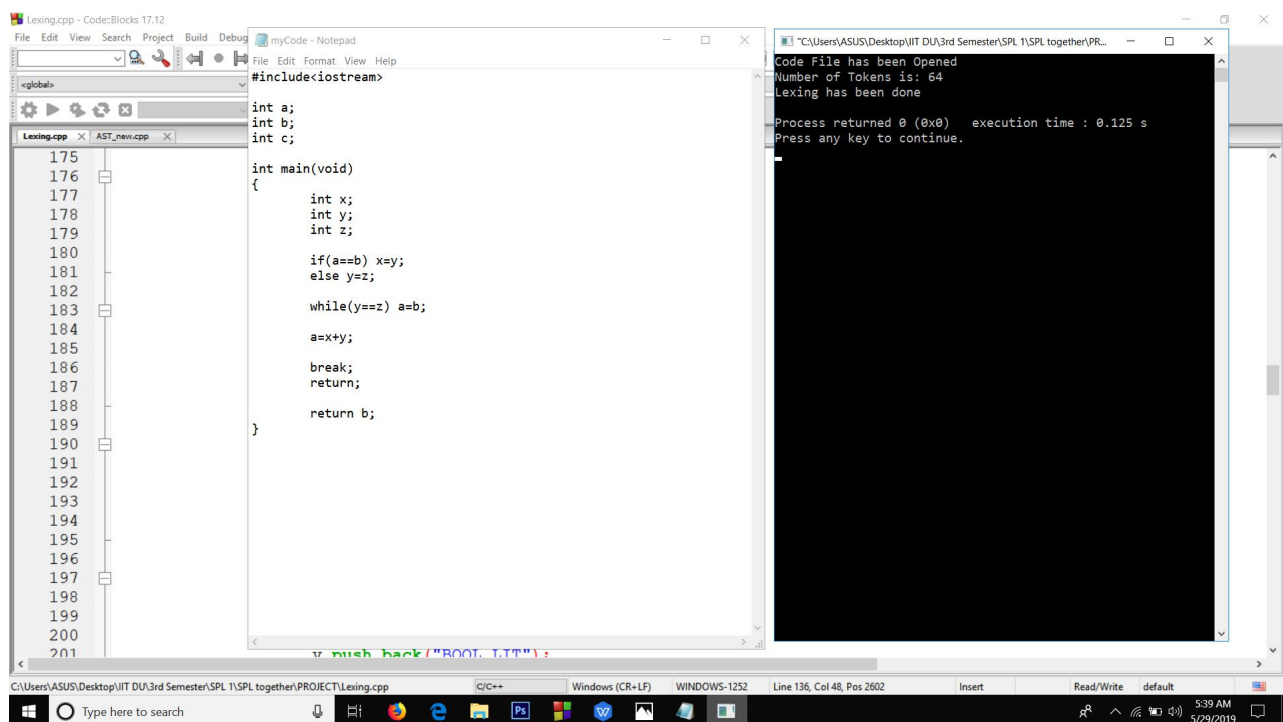
The abstract Syntax Tree is being traversed by the following code which denotes the preorderTraversal of the tree.

```
Lexing.cpp x AST_new.cpp x
283     return;
284 }
285
286
287 void function_decl()
288 {
289     int marker=i;
290
291     if(params()==true && compound_stmt()==true)
292     {
293         node* temp = create_newNode("function_decl");
294         vector<node*> tempVec;
295         tempVec.push_back(temp);
296         leaf->children=tempVec;
297         temp->parent=leaf;
298
299         leaf = tempVec[0];
300         function_decl_leaf = leaf;
301
302         //cout << "function_decl" << endl;
303
304         i=marker;
305         params_print();
306         compound_stmt_print();
307
308         success=1;
309
310         return;
311     }
312     else
313     {
314         cout << "***THE CODE DOES NOT FOLLOW THE CONTEXT FREE GRAMMAR" << endl;
315         return;
316     }
317 }
318
319
```

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.



The screenshot displays a code editor window titled 'Lexing.cpp - Code::Blocks 17.12' with a file named 'myCode - Notepad'. The code in the editor is as follows:

```
#include<iostream>

int a;
int b;
int c;

int main(void)
{
    int x;
    int y;
    int z;

    if(a==b) x=y;
    else y=z;

    while(y==z) a=b;

    a=x+y;

    break;
    return;

    return b;
}
```

Below the code editor, a terminal window shows the output of the program's execution:

```
Code File has been Opened
Number of Tokens is: 64
Lexing has been done

Process returned 0 (0x0)   execution time : 0.125 s
Press any key to continue.
```

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:

```
"C:\Users\ASUS\Desktop\IIT DU\3rd Semester\SPL 1\SPL together\PROJECT\AST_new.exe"
Lexed File has been opened

Pre-Order-Traversal of Abstract Syntax Tree:

program
decl_list
decl
var_decl
type_spec (int)
IDENT (a)
decl
var_decl
type_spec (int)
IDENT (b)
decl
var_decl
type_spec (int)
IDENT (c)
decl
function_decl
params (VOID)
local_decls
local_decl
type_spec (int)
IDENT (x)
local_decl
type_spec (int)
IDENT (y)
local_decl
type_spec (int)
IDENT (z)
stmt_list
stmt
if_stmt
stmt
expr_stmt
IDENT (a)
EQ (==)
IDENT (b)
stmt
expr_stmt
IDENT (x)
ASSIGNMENT (=)
```

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>