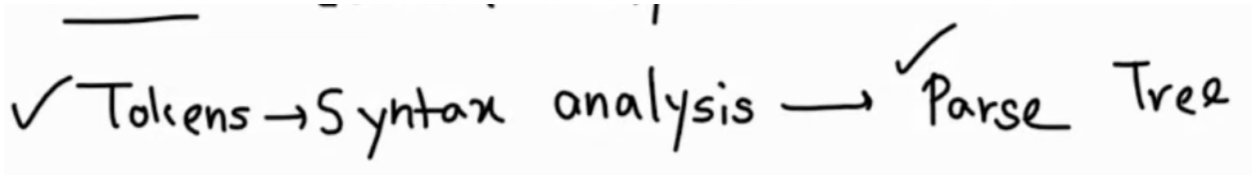
The compiler converts a high-level language to a low-level language.
After the high-level language goes through lexical analysis we get tokens.

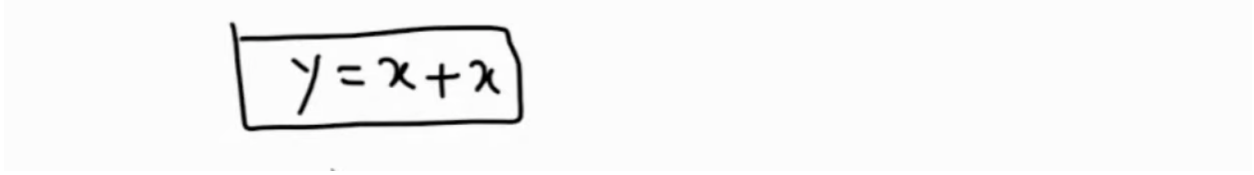
For all the lexemes of the high-level language, we get tokens after lexical analysis.

After the tokens go through syntax analysis we get a parse tree.
According to the given grammar, the rules will be checked. If the rules get matched we generate the parse tree and say the tokens to be synthetically organised. In short, if the tokens are synthetically organised finally a parse tree gets generated.



For this lab, if the nodes are grammatically correct then we will print them.

The grammar contains the necessary rules that a function needs to run.



The grammar given is for a function. But this is not a function, also y and x is not defined and the semi colon is missing. So this is grammatically incorrect.

So this will be printed.

symbol_info.h | TreeNode.h | syntax_analyzer.y

C: > Users > Asus > Desktop > Lab 03 Final > syntax_analyzer.y

```
extern int yyparse(void);
extern int yylex(void);

extern FILE *yyin;

extern YYSTYPE yylval;

std::ofstream outlog;

int lines = 1;

TreeNode* rootNode = nullptr;

void yyerror(const char *s)
{
    outlog << "At line " << lines << " " << s << std::endl << std::endl;
}

%}

%token IF ELSE FOR WHILE DO BREAK INT CHAR FLOAT DOUBLE VOID RETURN SWITCH CASE DEFAULT CONTINUE PRINTF ADDOP MUL
```

A root node variable is created where the reference of the root node is saved. (root node = first node) so that the tree can be traversed in post order.

symbol_info.h | TreeNode.h | syntax_analyzer.y

C: > Users > Asus > Desktop > Lab 03 Final > TreeNode.h

```
#ifndef TREE_NODE_H
#define TREE_NODE_H

#include <string>
#include <vector>

class TreeNode {
private:
    std::string type;
    std::string value;
    std::vector<TreeNode*> children;

public:
    TreeNode(const std::string& type, const std::string& value = "")
        : type(type), value(value) {}

    static TreeNode* createNonTerminalNode(const std::string& type) {
        return new TreeNode(type);
    }

    static TreeNode* createTerminalNode(const std::string& type, const std::string& value = "") {
        return new TreeNode(type, value);
```

In type, the tokens are stored

Value = lexeme → things that are read from the input file.

```
%nonassoc LOWER_THAN_ELSE
%nonassoc ELSE

%start program

%%

program : program unit
    {
        TreeNode* node = TreeNode::createNonTerminalNode("program");
        node->addChild($1->getNode());
        node->addChild($2->getNode());
        $$ = new symbol_info("program", "non_terminal", node);
        rootNode = node;
    }
    | unit
    {

    }
    ;

unit : var_declaration
```

As createNonTerminalNode is called, inside the TreeNode.h file, a non-terminal node will be created.

When addChild method is called, it adds the child under the respective node on which it was called upon.

```
func_definition : type_specifier id_name LPAREN parameter_list RPAREN compound_statement
        {
            TreeNode* node = TreeNode::createNonTerminalNode("func_definition"); /// WILL THERE
            node->addChild($1->getNode());
            node->addChild($2->getNode());
            node->addChild(TreeNode::createTerminalNode("LPAREN", "("));
            node->addChild($4->getNode());
            node->addChild(TreeNode::createTerminalNode("RPAREN", ")"));
            node->addChild($6->getNode());
            $$ = new symbol_info("func_definition", "non_terminal", node); /// WILL THERE
        }
```

Node contains the reference of the tree node class object

```
    TreeNode(const std::string& type, const std::string& value = "")
        : type(type), value(value) {}

    static TreeNode* createNonTerminalNode(const std::string& type) {
        return new TreeNode(type);
    }

    static TreeNode* createTerminalNode(const std::string& type, const std::string& value = "") {
        return new TreeNode(type, value);
    }

    void addChild(TreeNode* child) {
        children.push_back(child);
    }

    const std::string& getType() const {
        return type;
    }

    const std::string& getValue() const {
        return value;
    }
```

Getnode method returns the already created branch which basically gets added as a child.

$$ ← symbol table

```
40  program : program unit
41      {
42          TreeNode* node = TreeNode::createNonTerminalNode("program");
43          node->addChild($1->getNode());
44          node->addChild($2->getNode());
45          $$ = new symbol_info("program", "non_terminal", node);
46          rootNode = node;
```

Symbol info class is created and is added as a pointer in the symbol table($$)

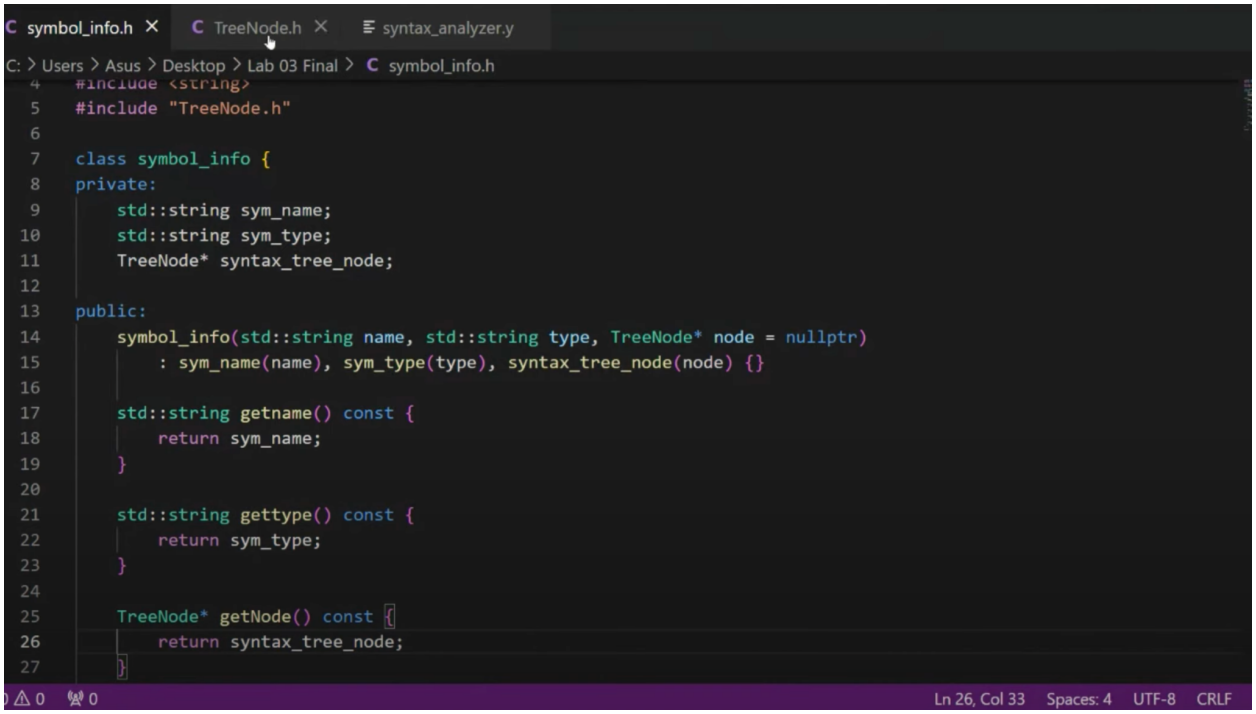
```
369     if (rootNode) {
370         rootNode->postOrderTraversal(outlog);
371     }
```

Outlog parameter is passed which is the output file to write the value i.e. the lexeme that was read in the input file.

```
50          if (!value.empty()) {
51              outFile << " " << value;
52          }
```

If the node is not empty then values will be printed along with space.

To summarize we are creating nodes based on terminal and non-terminal. After that, we add children by calling addChild method. Then getnode method is used to add the already created branch. Finally, the symbol info class is created and is added under the symbol info table.



Syntax tree node contains the reference of the symbol info class.