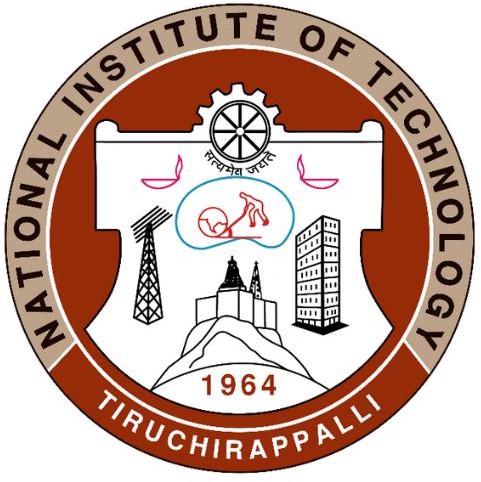
**NATIONAL INSTITUTE OF TECHNOLOGY, TIRUCHIRAPPALLI**



CSPC62

COMPILER DESIGN

TOPIC: C++ Compiler

LAB REPORT-2

Sub Topic : Syntax Analyzer

**DONE BY:**

|  |  |  |
| --- | --- | --- |
| **S.No** | **Name** | **RollNo** |
| 1. | Hema Sai Dorababu K | 106121065 |
| 2. | Chetan Reddy T | 106121139 |
| 3. | Harshith Babu B | 106121029 |

***SynTAX ANALYZER***

**Introduction**

**What is a Syntax Analyzer?**

A syntax analyzer, also known as a parser, is a program that checks the grammatical structure of an input program written in a specific programming language. It verifies if the code follows the language's rules and constructs.

**Role in Compilers:**

Syntax analysis is a crucial stage in the compilation process, typically the second phase after lexical analysis.

1. **Lexical Analysis:** Breaks down the source code into smaller meaningful units called tokens (keywords, identifiers, operators, literals).
2. **Syntax Analysis:** Checks if the sequence of tokens adheres to the grammar rules of the programming language.

**How it Works:**

* The syntax analyzer uses a set of predefined rules, that define valid program structures.
* It reads the token stream generated by the lexical analyzer one token at a time.
* It applies the grammar rules to determine if the sequence of tokens forms a valid construct in the language.
* If a violation is found, it reports a syntax error with details like the line number and potential error message.

**Benefits:**

* **Early Error Detection:** Syntax analysis helps identify errors early in the development process, saving time and effort compared to debugging runtime errors.
* **Improved Code Quality:** By enforcing language rules, syntax analysis promotes well-structured and maintainable code.
* **Enables Further Compilation Stages:** A successfully parsed program can proceed to semantic analysis (checking types) and code generation (creating machine code).

**Limitations:**

* **Focuses on Structure:** Syntax analyzers only verify the code's structure, not its logical correctness or efficiency.
* **Language Specific:** Each language has its own grammar rules, so a syntax analyzer is designed for a particular programming language.

**Applications:**

Syntax analyzers are used in various contexts:

* **Compilers and Interpreters:** As a core component for translating code into machine code or executing it directly.
* **Text Editors and IDEs:** To provide real-time syntax highlighting and error checking as developers write code.
* **Data Validation:** To ensure data entered in a specific format adheres to defined rules.

**Key Functionalities:**

* **Lexical Analysis:** The code likely uses the yyin stream and yytext variable to handle the input source code. Functions like yyerror might be used for error reporting during the lexical analysis phase (not shown in the provided code).
* **Grammar Parsing:** The code utilizes the yylex function to retrieve tokens from the input. The %token section defines the recognized tokens, including keywords, operators, identifiers, and literals.
* **Abstract Syntax Tree (AST) Construction:** The parser uses a recursive descent approach to build an AST representing the program structure. The create\_Node function creates AST nodes with labels, values, and references to child nodes.
* **Error Handling:** The yyerror function reports syntax errors encountered during parsing, potentially including line numbers and error messages.

**Parsing Rules:**

The grammar rules are defined using BNF (Backus-Naur Form) notation within the %% section. These rules specify how different tokens can be combined to form valid program constructs. Here are some key observations:

* The program can start with a sequence of headers, followed by a function declaration and more program statements, or just statements alone.
* Function declarations include return type, identifier, parameter list, and a compound statement containing local declarations and statements.
* Statements can be variable declarations, assignments, control flow statements (if, else, while, for), input/output statements (cout, cin), return statements, or error handling.
* Expressions are built using arithmetic operators (+, -, \*, /), comparison operators (<=, >=, ==, !=, <, >), and unary operators.

**1. Create a parser for your programming language:**

**Code Snippets:**

**Lexer.l:**

%option yylineno

%{

#include "temp2.tab.h"

#include<stdio.h>

#include<stdlib.h>

#include <string.h>

#include <stdarg.h>

#include "TreeNode.h"

int st[100];

int top,count,currscope,up,declared = 0;

char decl[20];

int flag = 0;

void installID(char \*text,int nm,int scp);

void display();

struct entry

{

char arr[20];

int scope;

char dtype[10];

int value;

};

struct entry symbolTable[100];

TreeNode \*createNode(char \*label, int value, char \*value\_str, int num\_children, ...) {

TreeNode \*newNode = (TreeNode \*)malloc(sizeof(TreeNode));

newNode->label = label;

newNode->value = value;

newNode->value\_str = (char\*)malloc(sizeof(char) \* strlen(value\_str) + 1);

strcpy(newNode->value\_str,value\_str);

newNode->num\_children = num\_children;

newNode->children = malloc(sizeof(TreeNode\*) \* num\_children);

va\_list args;

va\_start(args, num\_children);

for (int i = 0; i < num\_children; i++) {

newNode->children[i] = va\_arg(args, TreeNode\*);

}

va\_end(args);

return newNode;

}

%}

/\* regular definitions \*/

delim [ \t]

ws {delim}+

letter [a-zA-Z]

digit [0-9]

id {letter}({letter}|{digit})\*

relop (<=|>=|==|!=|<|>)

logicalop (&&|[|][|])

leftshift (<<)

rightshi (>>)

plus [+]

minus [-]

mult [\*]

div [/]

num ({digit}+)

float ({num}\.{num})

arithmeticop ({plus}|{minus}|{mult}|{div})

increment {plus}{plus}

decrement {minus}{minus}

assignop =

string (\"(\\.|[^\\"])\*\")

character (\'(\\.|[^\\'])\*\')

keyword (if|else|const|while|for|int|float|return|void|main|char|"long long"|double|short|long|unsigned|signed|define|struct|enum|typedef|sizeof|sta c|register|auto|break|case|con nue|default|do|goto|switch|cout|cin|endl|bool|using|namespace|std|include|iostream|vector|map|set|queue|stack|push\_back|pop\_back|pop|push|top|front|priority\_queue)

inval (({digit}+{id}))

comment (\/\/.\*|\/\\*([^\*]|(\\*+[^\*/]))\*\\*+\/[\n]\*)

%%

[\n] {printf("newline\n");return EOL;}

#include<[^>]+> {printf( "header:%s\n",yytext); yylval.node = createNode("header",-1,yytext,0); return header;};

#include\"([^\"]+)\" {printf( "header:%s\n",yytext); yylval.node = createNode("header",-1,yytext,0); return header;};

if { printf("if\n"); yylval.node = createNode("if",-1,yytext,0); return if\_x; }

else { printf("else\n"); yylval.node = createNode("else",-1,yytext,0); return else\_x; }

while { printf("while\n"); yylval.node = createNode("while",-1,yytext,0); return while\_x; }

for {printf("for\n"); yylval.node = createNode("for",-1,yytext,0); return for\_x; }

return { printf("return \n"); yylval.node = createNode("return",-1,yytext,0); return return\_x; }

printf { printf("printf\n"); yylval.node = createNode("printf",-1,yytext,0); return printf\_x; }

cout { printf("cout\n"); yylval.node = createNode("cout",-1,yytext,0); return cout; }

cin { printf("cin\n"); yylval.node = createNode("cin",-1,yytext,0); return cin; }

"<<" { printf("insert\n"); yylval.node = createNode("<<",-1,yytext,0); return insert; }

">>" { printf("extract\n"); yylval.node = createNode(">>",-1,yytext,0); return extract; }

{assignop} {printf("assignop:%s\n", yytext); yylval.node = createNode("assignop",-1,yytext,0); return assignmentop; }

{relop} { ; printf("compop:%s\n", yytext); strcpy(yylval.str,yytext); yylval.node = createNode("comparisionop",-1,yytext,0); return comparisionop; }

{logicalop} { printf("logicalop:%s\n", yytext); return logicalop; }

int|float|double|char|string|"long long"|short|long { printf("datatype:%s\n", yytext);declared = 1;strcpy(decl, yytext); yylval.node = createNode("datatype",-1,yytext,0); return datatype; }

{num} { printf("num:%s\n", yytext); yylval.node = createNode("number",atoi(yytext),"NULL",0); return number; }

{increment} { ; printf("unary:%s\n", yytext); yylval.node = createNode("unary",-1,yytext,0); return unary; }

{decrement} { ; printf("unary:%s\n", yytext); yylval.node = createNode("unary",-1,yytext,0); return unary; }

{id} { if(declared == 1) {installID(yytext, yylineno, st[top]);} printf("id:%s\n", yytext); yylval.node = createNode("id",-1,yytext,0); return identifier; }

{character} { printf("character:%s\n", yytext); yylval.node = createNode("character",-1,yytext,0); return character;}

{string} { printf("string:%s\n", yytext); yylval.node = createNode("string",-1,yytext,0); return string; }

\n\* ;

{ws} ;

{plus} { yylval.node = createNode("plus",-1,yytext,0); return PLUS;}

{minus} { yylval.node = createNode("minus",-1,yytext,0); return MINUS;}

{mult} { yylval.node = createNode("mult",-1,yytext,0); return MUL;}

{div} { yylval.node = createNode("div",-1,yytext,0); return DIV;}

"{" { printf("Lbrace:%s\n",yytext); currscope++; yylval.node = createNode("{",-1,yytext,0); return LBRACE; }

"}" { currscope--;printf("Rbrace:%s\n",yytext); yylval.node = createNode("}",-1,yytext,0); return RBRACE; }

"(" {printf ("LPAREN: %s, line: %d\n", yytext, yylineno); yylval.node = createNode("LPAREN",-1,yytext,0); return LPAREN;}

")" {printf ("RPAREN: %s, line: %d\n", yytext, yylineno); yylval.node = createNode("RPAREN",-1,yytext,0); return RPAREN;}

"[" {printf ("LBRACKET: %s, line: %d\n", yytext, yylineno); yylval.node = createNode("[",-1,yytext,0); return yytext[0];}

"]" {printf ("RBRACKET: %s, line: %d\n", yytext, yylineno); yylval.node = createNode("]",-1,yytext,0); return yytext[0];}

";" { declared = 0; printf("semicolon :%s,line:%d\n",yytext,yylineno); yylval.node = createNode(";",-1,yytext,0); return SEMICOLON; }

"," { printf("comma :%s,line:%d\n",yytext,yylineno); yylval.node = createNode(",",-1,yytext,0); return COMMA; }

{comment} { };

. { return yytext[0]; }

%%

void installID(char \*text, int nm, int scp)

{

int present = 0;

for (int i = 0; i <= up; i++)

{

if (strcmp(symbolTable[i].arr, text) == 0 && symbolTable[i].scope == currscope)

{

present = 1;

break;

}

}

if (!present)

{

strcpy(symbolTable[up].arr, text);

symbolTable[up].scope = currscope;

strcpy(symbolTable[up].dtype, decl);

// symbolTable[up].value = up;

up++;

}

}

void display()

{

printf("\nSymbol Table\n");

printf("Symbol\t\tscope\t\tdtype\n");

for (int i = 0; i < up; i++)

{

printf("%s\t\t%d\t\t%s\n", symbolTable[i].arr,symbolTable[i].scope, symbolTable[i].dtype);

}

}

int yywrap()

{

return 1;

}

**Parser.y:**

%{

#include "TreeNode.h"

#include<stdio.h>

#include<stdlib.h>

#include<string.h>

int yylex();

int yyerror(const char \*s);

int yyparse();

extern void display();

struct entry{

char arr[20];

int scope;

char dtype[10];

int value;

};

extern struct entry symbolTable[100];

TreeNode \*head = NULL;

struct TreeNode \*create\_Node(char \*label, int value, char \*value\_str, int num\_children, ...) {

struct TreeNode \*newNode = (struct TreeNode \*)malloc(sizeof(struct TreeNode));

newNode->label = label;

newNode->value = value;

newNode->value\_str = (char\*)malloc(sizeof(char) \* strlen(value\_str) + 1);

strcpy(newNode->value\_str,value\_str);

newNode->num\_children = num\_children;

newNode->children = NULL;

if(num\_children<1) return newNode;

newNode->children = malloc(sizeof(TreeNode\*) \* num\_children);

va\_list args;

va\_start(args, num\_children);

for (int i = 0; i < num\_children; i++) {

newNode->children[i] = va\_arg(args, struct TreeNode\*);

}

va\_end(args);

return newNode;

}

void printTree(TreeNode \*root,int level){

if(root == NULL){

return;

}

for(int i = 0;i<level;i++){

printf(" ");

}

if(root->value ==-1 && strcmp(root->value\_str,"NULL") == 0){

printf("%d.%s\n",level, root->label);

}else if(root->value == -1){

printf("%d.%s\n", level,root->value\_str);

}else{

printf("%d.(%s,%d)\n",level, root->label, root->value);

}

// printf("(%s,%d)\n", root->label,root->value);

for(int i = 0;i<root->num\_children;i++){

printTree(root->children[i],level+1);

}

}

%}

%union {

int num;

char \*str;

struct TreeNode \*node;

}

%start program

%token EOL

%error-verbose

%token<node> PLUS MINUS MUL DIV number if\_x else\_x while\_x for\_x return\_x printf\_x main\_x assignmentop comparisionop logicalop datatype unary identifier string character cout cin insert extract header LBRACE RBRACE LPAREN RPAREN SEMICOLON COMMA

%type<node> E T F assignment\_statement statement\_list function\_declaration declaration\_statement id\_list insert\_statement extract\_statement if\_statement else\_statement if\_else\_statement while\_statement for\_statement return\_statement cout\_statement cin\_statement statement headers parameter\_list program

/\* rules \*/

%%

program: headers function\_declaration program { printf("program No: %d\n",$$); $$ = create\_Node("program", -1, "NULL", 3,$1,$2,$3);head = $$;}

| statement\_list program {$$ = create\_Node("program", -1, "NULL", 2,$1,$2); head = $$;}

| EOL program {$$ = create\_Node("program", -1, "NULL", 1,$2); head = $$;}

| EOL {$$ = NULL;}

|;

headers: header {printf("headers1\n"); $$ = create\_Node("headers", -1, "NULL",1,$1);}

| headers EOL headers {printf("headers\n"); $$ = create\_Node("headers", -1,"NULL", 2, $1, $3);}

| EOL headers {printf("headers\n"); $$ = create\_Node("headers", -1, "NULL",1, $2);}

| EOL{ $$ = NULL; };

function\_declaration: datatype identifier LPAREN parameter\_list RPAREN LBRACE statement\_list RBRACE statement\_list {printf("Function NO: %d\n",$$); $$ = create\_Node("function\_declaration", -1, "NULL", 9, $1,$2,$3, $4, $5, $6, $7, $8, $9); };

parameter\_list: datatype identifier COMMA parameter\_list { $$ = create\_Node("parameter\_list", -1, "NULL",4, $1, $2, $3, $4); }

| datatype identifier { $$ = create\_Node("parameter\_list", -1, "NULL",2, $1, $2);}

| EOL {$$ = NULL;}

| { $$ = NULL; };

statement\_list:

statement statement\_list {printf("statement\_list\n"); $$ = create\_Node("statement\_list", -1, "NULL", 2, $1, $2);}

| EOL statement\_list { $$ = create\_Node("statement\_list", -1, "NULL", 1, $2);}

| EOL {$$ = NULL;}

|

;

statement: declaration\_statement {printf("declaration\_statement\n"); $$ = create\_Node("declaration\_statement", -1, "NULL",0); }

| assignment\_statement SEMICOLON {printf("assignment\_statement\n"); $$ = create\_Node("assignment\_statement", -1,"NULL",2, $1, $2);}

| for\_statement {printf("for\_statement\n"); $$ = create\_Node("for\_statement", -1, "NULL",1,$1); }

| if\_statement {printf("if\_statement\n"); $$ = create\_Node("if\_statement", -1, "NULL",1,$1); }

| if\_else\_statement {printf("if\_else\_statement\n"); $$ = create\_Node("if\_else\_statement", -1, "NULL",1,$1); }

| while\_statement {printf("while\_statement\n"); $$ = create\_Node("while\_statement", -1, "NULL",1,$1); }

| cout\_statement {printf("cout\_statement\n"); $$ = create\_Node("cout\_statement", -1, "NULL",1,$1); }

| cin\_statement {printf("cin\_statement\n"); $$ = create\_Node("cin\_statement", -1, "NULL",1,$1); }

| return\_statement {printf("return\_statement\n"); $$ = create\_Node("return\_statement", -1, "NULL",1,$1); }

| error SEMICOLON { $$ = create\_Node("error", -1, "NULL",0); }

| EOL { $$ = NULL; }

;

if\_statement: if\_x LPAREN E RPAREN LBRACE statement\_list RBRACE { $$ = create\_Node("if\_statement", -1,"NULL", 7,$1,$2,$3,$4,$5,$6,$7);}

| if\_x LPAREN E RPAREN statement { $$ = create\_Node("if\_statement", -1,"NULL", 5,$1,$2,$3,$4,$5);}

| if\_statement EOL { $$ = create\_Node("if\_statement", -1,"NULL", 1,$1);}

else\_statement: else\_x LBRACE statement\_list RBRACE { $$ = create\_Node("else\_statement", -1, "NULL", 3, $1, $2, $3); }

| else\_x statement { $$ = create\_Node("else\_statement", -1, "NULL", 2, $1, $2); }

| else\_statement EOL { $$ = create\_Node("else\_statement", -1, "NULL", 1, $1); }

;

if\_else\_statement: if\_statement else\_statement { $$ = create\_Node("if\_else\_statement", -1, "NULL", 2, $1, $2); };

while\_statement: while\_x LPAREN E RPAREN LBRACE statement\_list RBRACE { $$ = create\_Node("while\_statement", -1, "NULL", 7, $1, $2, $3, $4, $5, $6, $7); };

for\_statement: for\_x LPAREN declaration\_statement E SEMICOLON E RPAREN LBRACE statement\_list RBRACE { $$ = create\_Node("for\_statement", -1, "NULL", 9, $1, $2, $3, $4, $5, $6, $7, $8, $9); };

return\_statement: return\_x E SEMICOLON { $$ = create\_Node("return\_statement", -1,"NULL", 2, $1, $2); }

| return\_x SEMICOLON { $$ = create\_Node("return\_statement", -1, "NULL", 2, $1,$2); };

cout\_statement: cout insert\_statement SEMICOLON { $$ = create\_Node("cout\_statement", -1,"NULL", 2, $1, $2); };

insert\_statement: insert E insert\_statement { $$ = create\_Node("insert\_statement", -1,"NULL", 3, $1, $2, $3); }

| insert string insert\_statement { $$ = create\_Node("insert\_statement", -1, "NULL", 3, $1, $2, $3); }

| insert E { $$ = create\_Node("insert\_statement", -1, "NULL", 2, $1, $2); }

| insert string { $$ = create\_Node("insert\_statement", -1, "NULL", 2, $1, $2); }

cin\_statement: cin extract\_statement SEMICOLON { $$ = create\_Node("cin\_statement", -1, "NULL", 3, $1, $2, $3); };

extract\_statement: extract identifier extract\_statement { $$ = create\_Node("extract\_statement", -1, "NULL", 3, $1, $2, $3); }

| extract identifier { $$ = create\_Node("extract\_statement", -1, "NULL", 2, $1, $2); }

;

declaration\_statement: datatype id\_list SEMICOLON { $$ = create\_Node("declaration\_statement", -1, "NULL", 3, $1, $2, $3); };

id\_list: identifier COMMA id\_list { $$ = create\_Node("id\_list", -1, "NULL", 3, $1, $2, $3); }

| assignment\_statement COMMA id\_list { $$ = create\_Node("id\_list", -1, "NULL", 3, $1, $2, $3); }

| assignment\_statement { $$ = create\_Node("id\_list", -1, "NULL", 1, $1); }

| identifier { $$ = create\_Node("id\_list", -1, "NULL", 1, $1); }

assignment\_statement:

F assignmentop E { printf("assignment-statement\n");$$ = create\_Node("assignment\_statement", -1, "NULL", 3, $1, $2, $3); $1->value = $3->value; }

;

E: F assignmentop E { $$ = create\_Node("E", -1, "NULL", 3, $1, $2, $3); $1->value = $3->value; }

| E comparisionop T { $$ = create\_Node("E", -1, "NULL", 3, $1, $2, $3);

if (strcmp($2->value\_str, "<=") == 0) {

$$->value = $1->value <= $3->value;

} else if (strcmp($2->value\_str, ">=") == 0) {

$$->value = $1->value >= $3->value;

} else if (strcmp($2->value\_str, "==") == 0) {

$$->value = $1->value == $3->value;

} else if (strcmp($2->value\_str, "!=") == 0) {

$$->value = $1->value != $3->value;

}

else if (strcmp($2->value\_str, "<") == 0) {

$$->value = $1->value < $3->value;

}

else if (strcmp($2->value\_str, ">") == 0) {

$$->value = $1->value > $3->value;

}

}

| E PLUS T { $$ = create\_Node("E", $1->value+$3->value, "NULL", 3, $1, $2, $3); }

| E MINUS T { $$ = create\_Node("E", $1->value-$3->value, "NULL", 3, $1, $2, $3); }

| T { $$ = create\_Node("E", $1->value, "NULL", 1, $1); }

;

T:

T MUL F { $$ = create\_Node("T", $1->value\*$3->value, "NULL", 3, $1, $2, $3);}

| T DIV F { $$ = create\_Node("T", $1->value/$3->value, "NULL", 3, $1, $2, $3);}

| F { $$ = create\_Node("T", $1->value, "NULL", 1, $1); }

;

F:

number { $$ = create\_Node("F", -1, "NULL" , 1, $1); }

| character { $$ = create\_Node("F", -1, "NULL", 1, $1); }

| LPAREN E RPAREN { $$ = create\_Node("F",$2->value, "NULL", 3, $1, $2, $3); }

| identifier { $$ = create\_Node("identifier", -1, "NULL", 1, $1); }

| unary identifier { $$ = create\_Node("F", -1, "NULL", 2, $1, $2); }

| identifier unary { $$ = create\_Node("F", -1, "NULL", 2, $1, $2); }

;

%%

#include <ctype.h>

int yyerror(const char \*s)

{

extern int yylineno;

// valid = 0;

if(yylineno != 27){

printf("Line no: %d \n The error is: %s\n",yylineno,s);

}

}

extern FILE \*yyin;

int main(int argc,char \*\*argv){

if(argc<2)

{

printf("Usage: %s <filename>\n",argv[0]);

return 1;

}

FILE \*fp = fopen(argv[1], "r");

if(fp==NULL)

{

printf("Error: File not found\n");

return 1;

}

extern char \*yytext;

yyin=fp;

yyparse();

printf("---------------\n");

printTree(head,0);

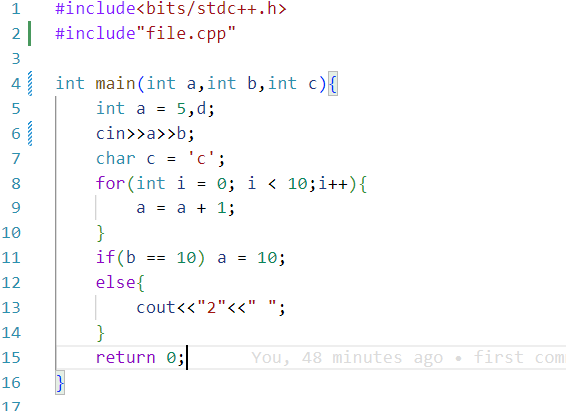
printf("---------------\n");

display();

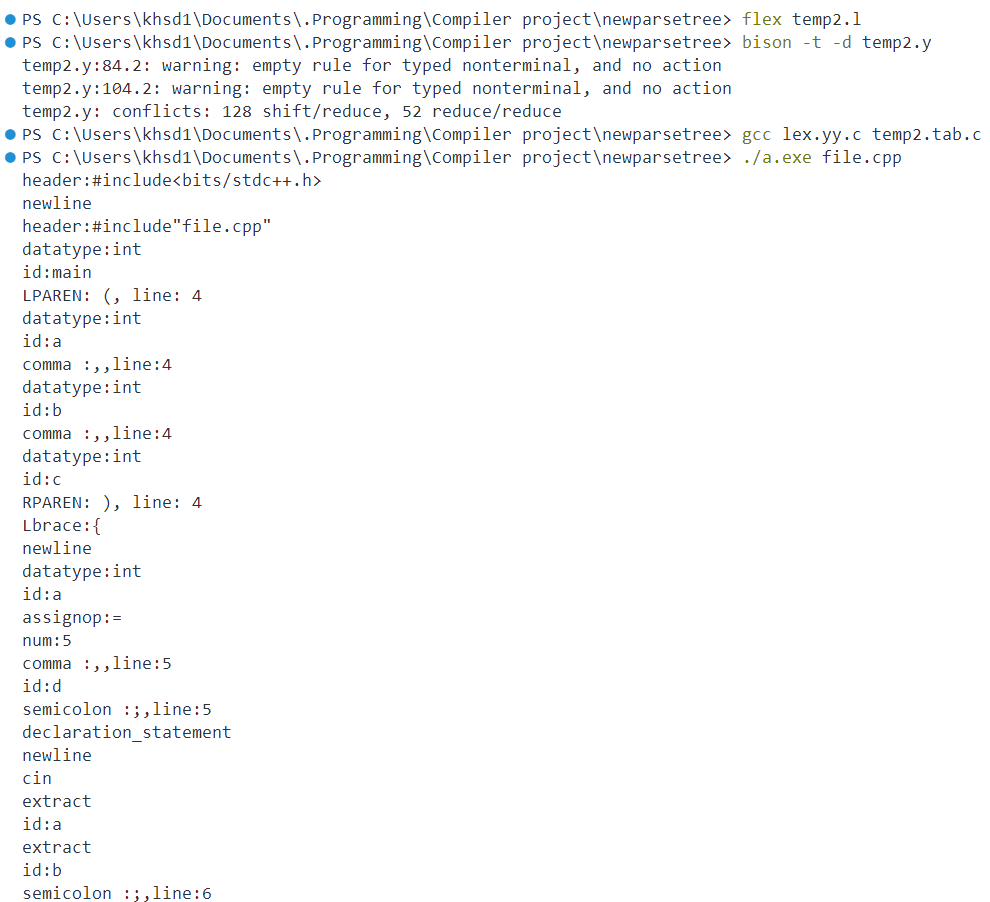
}

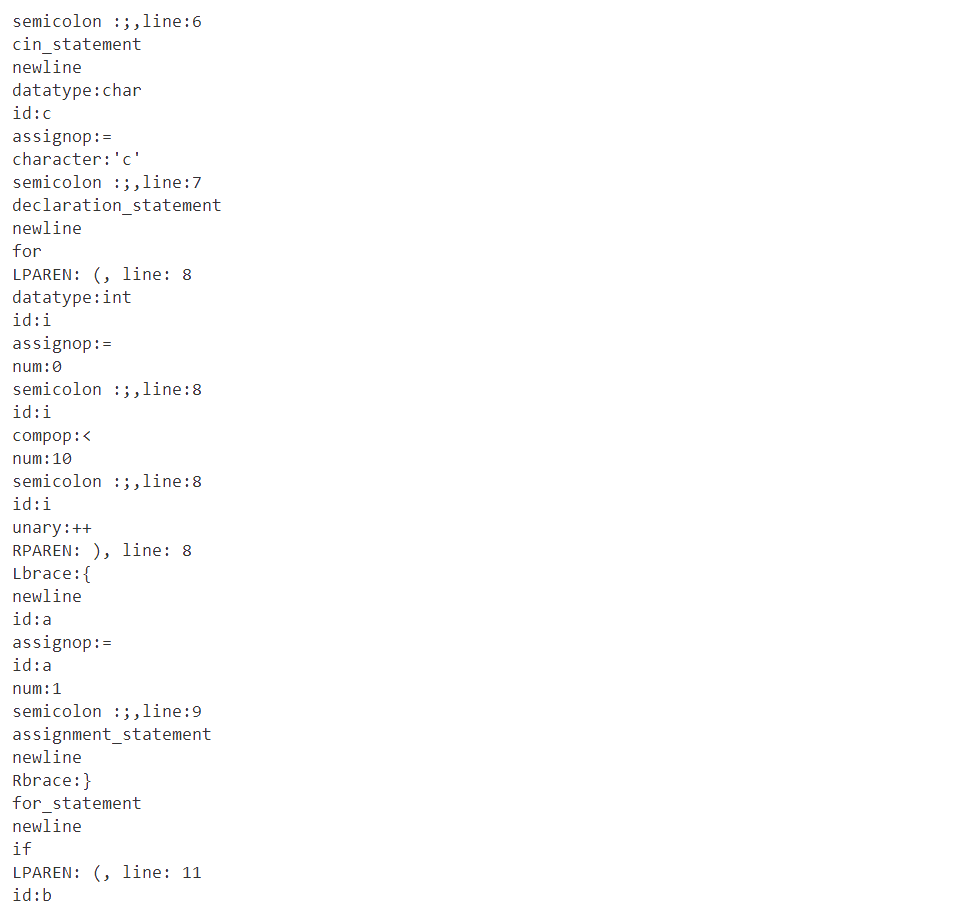
**2. Show that this parser correctly parses the input token generated by your lexical analyser for any program written in your programming language as well as identifies errors.**

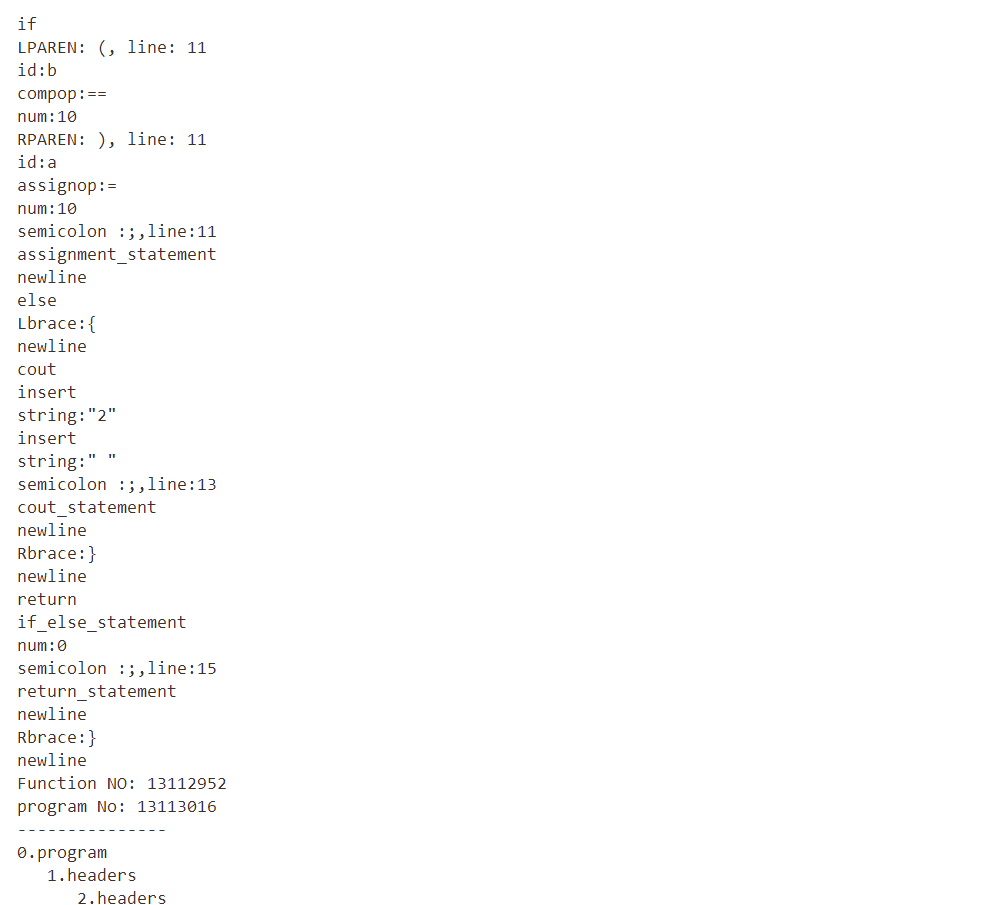
**Sample Code A:**



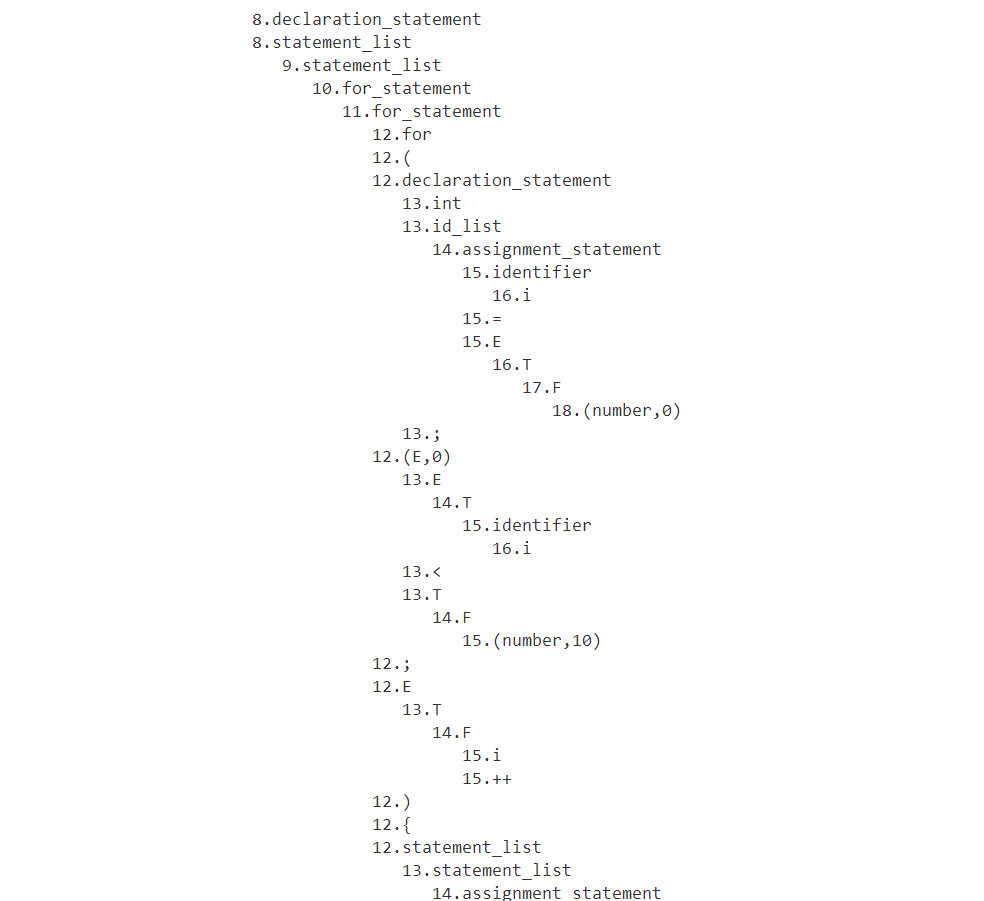
**Output Parse Tree:**

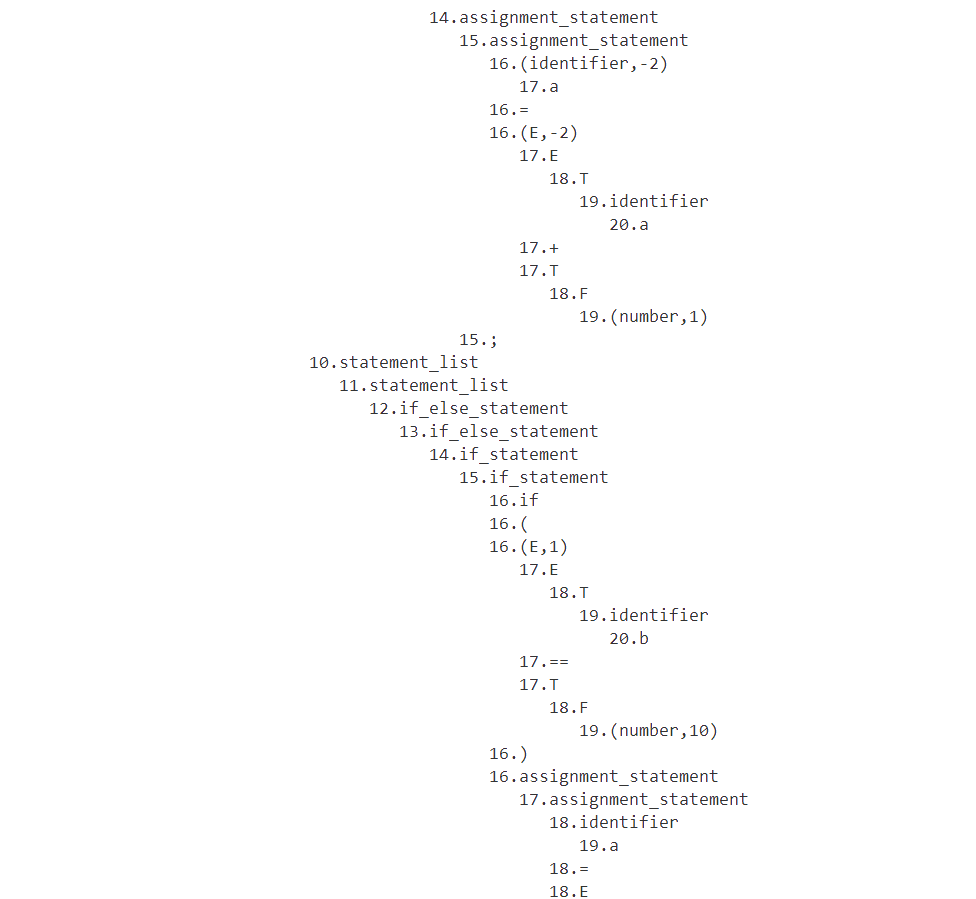


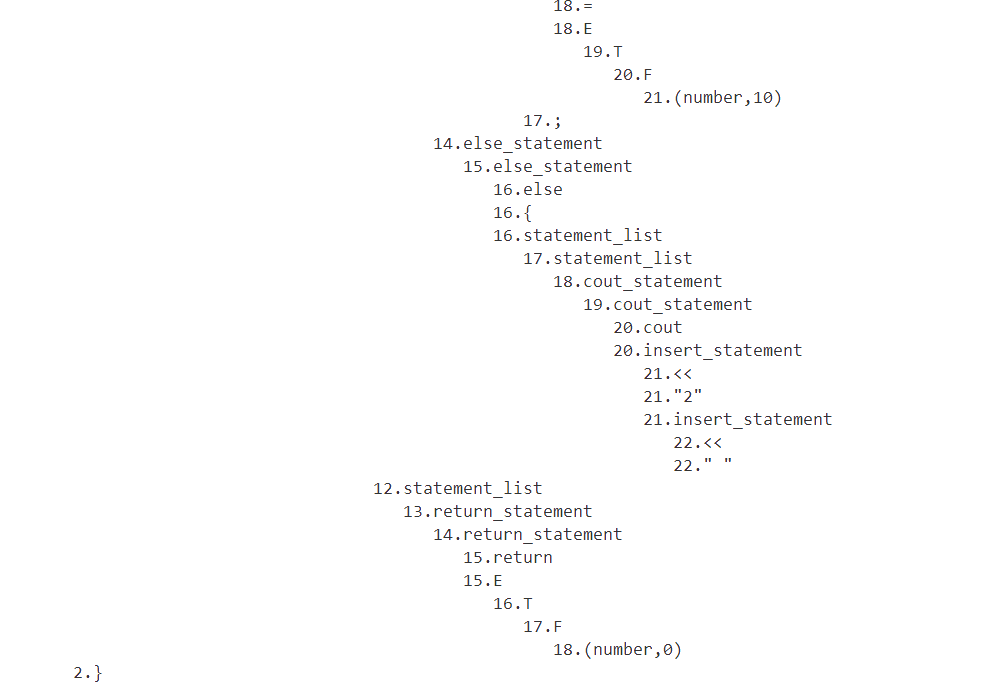




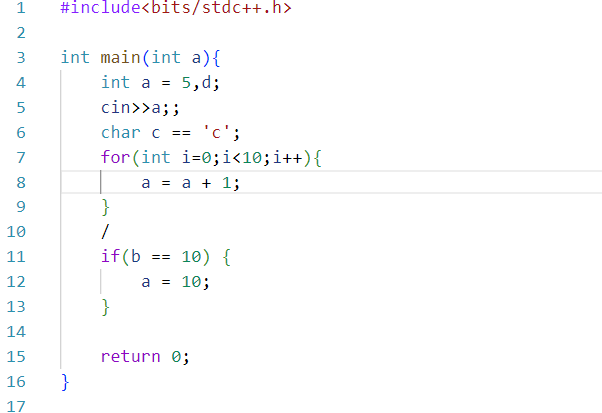




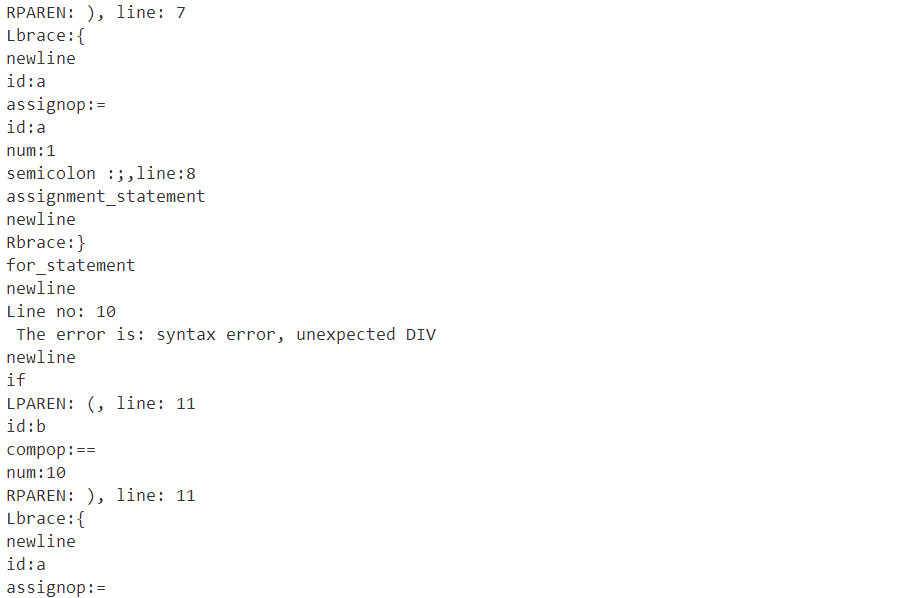
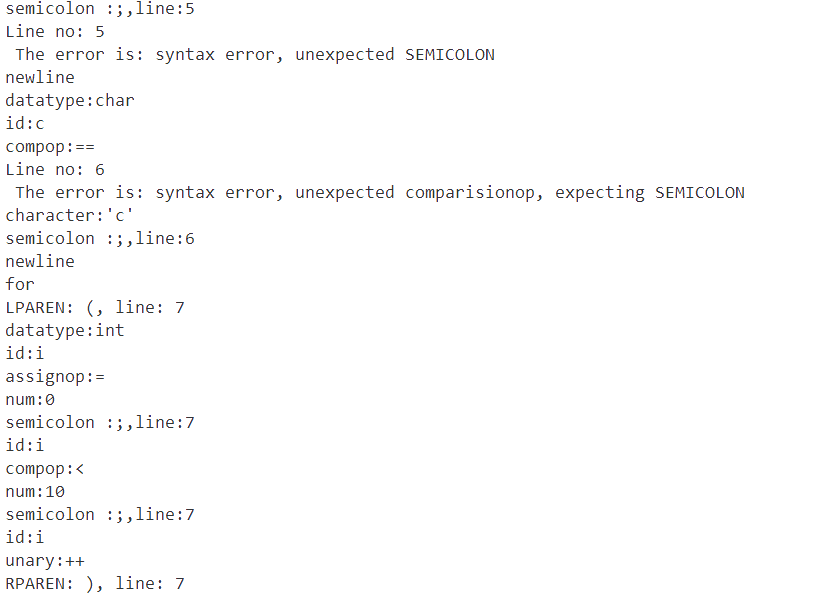
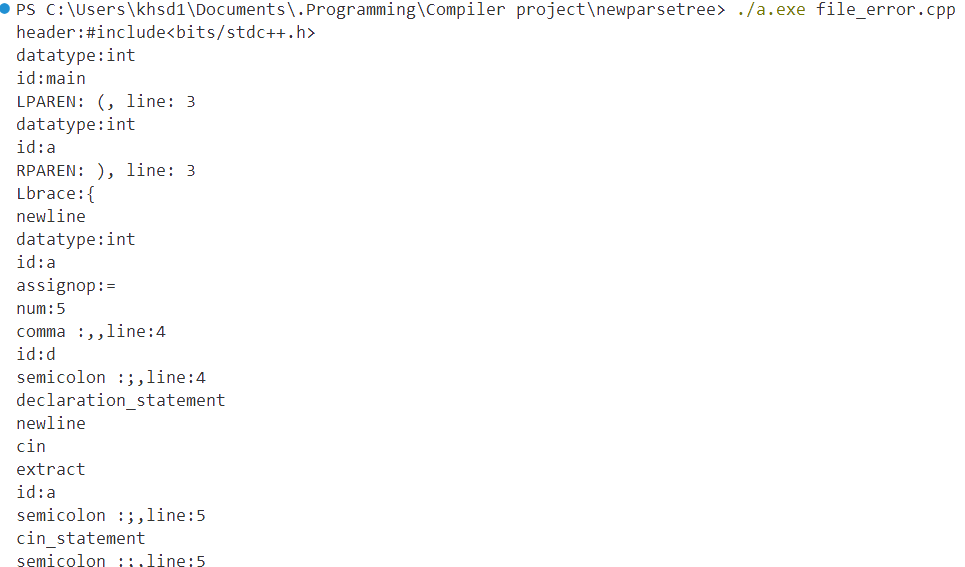


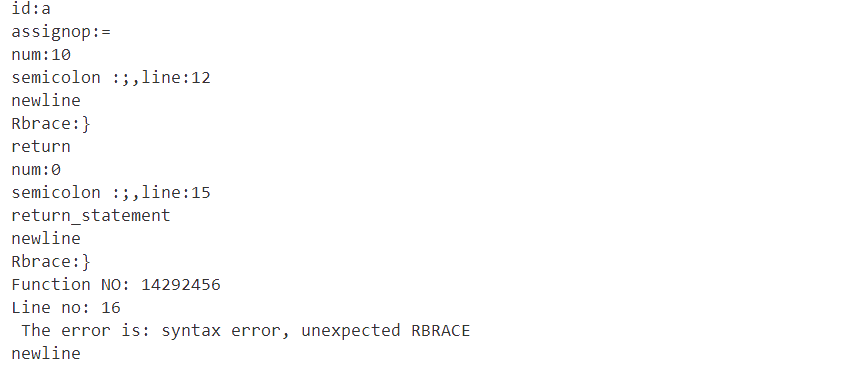


**Sample Code B: (For Error detection)**

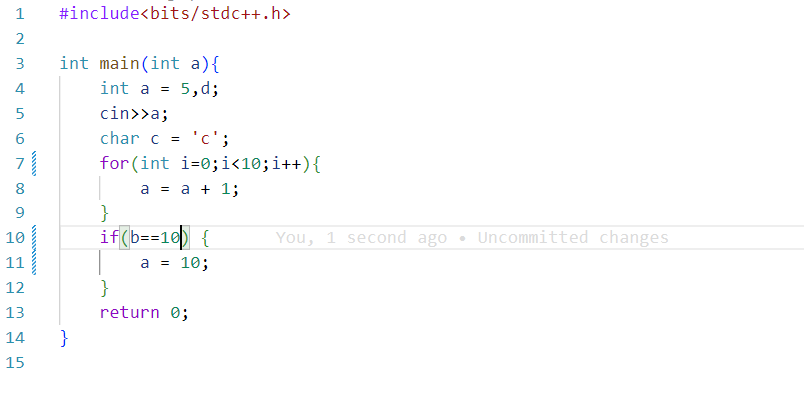


**Erros Detecting Output:**

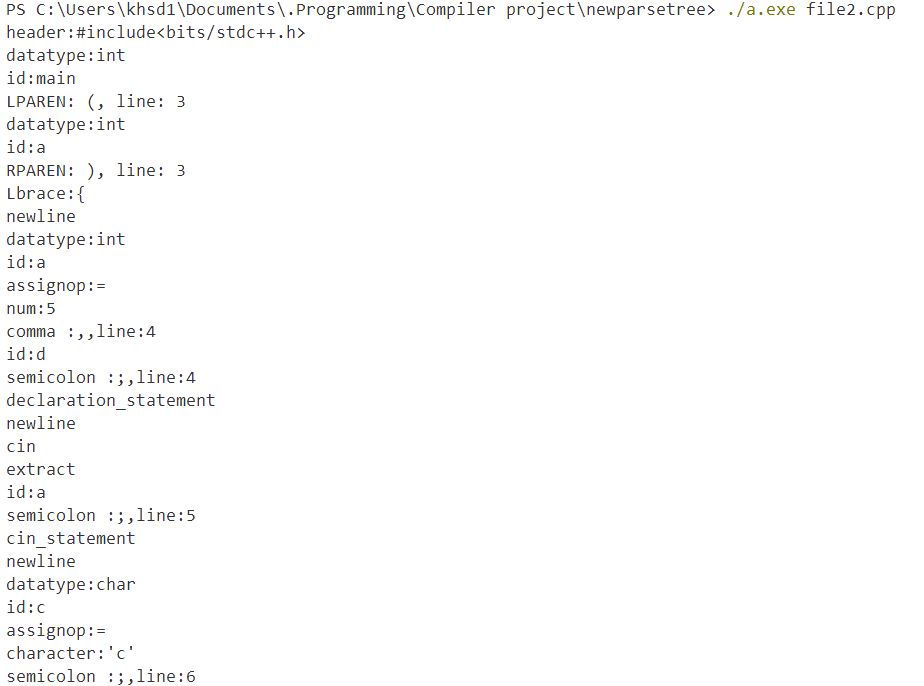


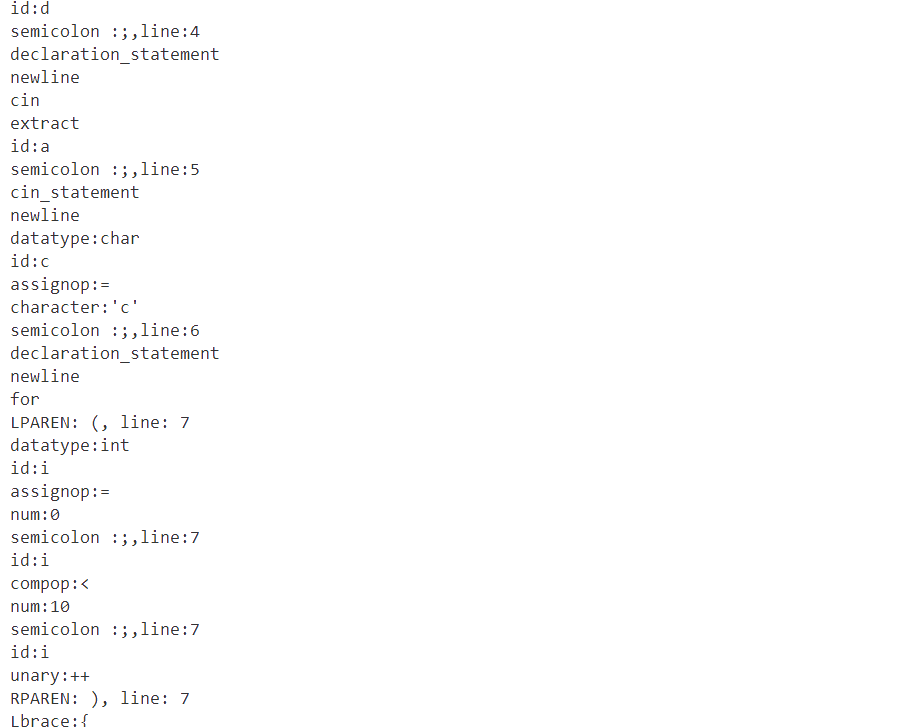


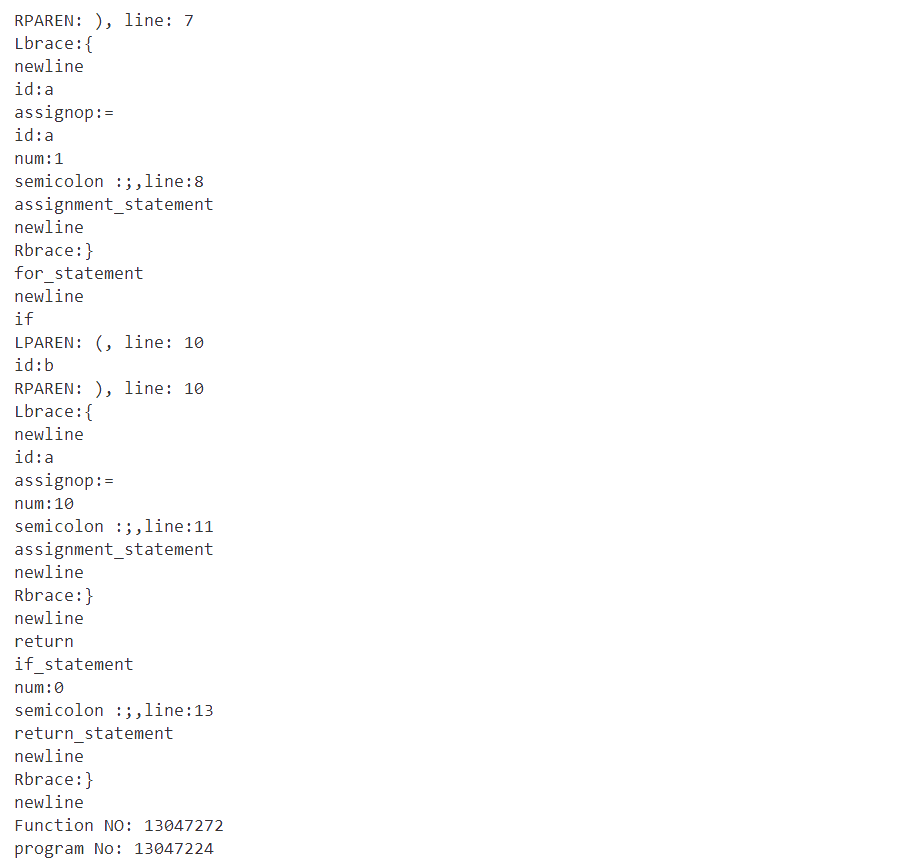
**3. Write a simple program in your language with all kinds of tokens and keywords and show that your compiler is correctly detecting the tokens and errors. Parse the program using your parser. Print step by step parsing process and draw the parse tree.**

****

**Output:**

****

****

****



