**CS304- COMPILER DESIGN LAB**

A REPORT ON THE PROJECT ENTITLED

**SYNTAX ANALYZER FOR THE C LANGUAGE**



**Group Members:**

**Hemang J Jamadagni Adithya S Ubaradka Adithya G**

221CS129 221CS105 221CS106

V SEMESTER B-TECH CSE- S1

**DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING**

**NATIONAL INSTITUTE OF TECHNOLOGY KARNATAKA**

**SURATHKAL**

**2024 – 2025**

**Abstract:**

The project focuses on the design and implementation of a syntax analyzer for the C programming language using **Lex** and **Yacc** tools. Lex is utilized for lexical analysis, where the source code is converted into a sequence of tokens such as keywords, identifiers, operators, and literals. Yacc, a parser generator, is employed for syntax analysis, where these tokens are checked against the formal grammar of the C language to construct a syntax tree, ensuring that the code adheres to the C language's syntactic rules.

The syntax analyzer is capable of identifying common syntactical errors, such as missing semicolons, unmatched parentheses, and incorrect operator usage, providing clear and detailed error messages for developers. This project also explores the integration of context-free grammar (CFG) for C within Yacc, enabling the parser to handle complex language constructs like loops, conditionals, function definitions, and more.

By using Lex and Yacc, this project demonstrates the process of converting high-level C code into its syntactic representation, facilitating further steps like semantic analysis or code generation in a compiler or interpreter pipeline. The syntax analyzer provides a foundation for deeper exploration into compiler design and error detection, offering a valuable tool for both educational and practical applications in programming language development.

**Contents**: **Page No**

* Introduction 3
  + Syntax Analyzer
  + Flex Script
  + C Program
* Design of Programs 5
  + Code
  + Explanation
* Test Cases 23
  + Valid Parse
  + Invalid Parse
* First and Follow Sets 26
  + First Sets
  + Follow Sets
* Implementation 41
* Results 41
* Future Work 44
* References 44

**Introduction**

**Syntax analysis** (also known as **parsing**) is a key stage in the compilation or interpretation of programming languages. It involves analyzing the structure of a sequence of tokens (which are produced during lexical analysis) to determine their grammatical structure according to a specific language's grammar rules.

Here's a more detailed breakdown of what syntax analysis involves:

1. **Input**: Tokens (e.g., keywords, operators, identifiers, literals) from the lexical analysis phase.
2. **Output**: A parse tree (or syntax tree) that represents the grammatical structure of the input according to the language's grammar.
3. **Purpose**: To ensure that the sequence of tokens follows the rules of the language's formal grammar (context-free grammar).
4. **Tools**: Syntax analyzers (or parsers) like LL parsers, LR parsers, and recursive descent parsers are commonly used.

### Key Concepts:

* **Context-Free Grammar (CFG)**: Defines the syntactical structure of a language. It's typically expressed in terms of production rules.
* **Parse Tree (or Syntax Tree)**: A hierarchical tree structure that represents the grammatical structure of the input code. Each node in the tree corresponds to a construct occurring in the source code.
* **Top-Down Parsing**: Begins from the start symbol and tries to derive the input sequence using the grammar rules.
* **Bottom-Up Parsing**: Starts from the input sequence and works backwards to derive the start symbol.

In summary, syntax analysis checks if the code is well-formed, ensuring it adheres to the language’s grammar before proceeding to later stages like semantic analysis or code generation.

**YACC Script**

Yacc stands for Yet Another Compiler-Compiler. Yacc is essentially a parser

generator. Yacc provides a general tool for imposing structure on the input to a

computer program. The Yacc user prepares a specification of the input process;

this includes rules describing the input structure, code to be invoked when these

rules are recognized, and a low-level routine to do the basic input. A function is

then generated by Yacc to control the input process. This function is called the

parser which calls the lexical analyzer to get a stream of tokens from the input.

Based on the input structure rules, called grammar rules, the tokens are

organized. When one of these rules has been recognized, then user code supplied

for this rule, an action, is invoked. Actions have the ability to return values and

make use of the values of other actions.

Yacc is written in portable C. The class of specifications accepted is a very

general one, LALR(1) grammars with disambiguating rules.

The structure of our yacc script is divided into three sections, separated by lines

that contain only two percent signs, as follows:

DECLARATIONS

%%

RULES

%%

AUXILIARY FUNCTIONS

The Declarations Section defines macros and imports header files written in C.

It is also possible to write any C code here, which will be copied directly into the

generated source file. We also define all parameters related to the parser here,

specifications like using leftmost derivations or rightmost derivations,

precedence, left and right associativity are declared here, data types and tokens

which will be used by the lexical analyzer are also declared at this stage.

The Rules Section contains the entire grammar which is used for deciding if the

input text is legally correct according to the specifications of the language. Yacc

uses these rules for reducing the token stream received from the lexical analysis

stage. All rules are linked to each other from the start state.

Yacc generates C code for the rules specified in the Rules section and places this

code into a single function called yyparse(). The Auxiliary Functions Section

contains C statements and functions that are copied directly to the generated

source file. These statements usually contain code called by the different rules.

This section essentially allows the programmer to add to the generated source

code.

**C Program**

The parser takes C source files as input for parsing. The input file is specified in

the auxiliary functions section of the yacc script.

The workflow for testing the parser is as follows:

1. Compile the yacc script using the yacc tool

$ yacc -d parser.y

2. Compile the flex script using the flex tool

$ lex lexer.l

3. The first two steps generate lex.yy.c, y.tab.c, and y.tab.h. The header file is

included in lexer.l file. Then, lex.yy.c and y.tab.c are compiled together.

$ gcc lex.yy.c y.tab.c

4. Run the generated executable file

$ ./a.out

**Design of Programs**

**Code:**

**Lexical Analyzer (lexer.l)**

%{

#include <stdio.h>

#include <string.h>

#include "y.tab.h"

struct ConstantTable{

char constant\_name[100];

char constant\_type[100];

int exist;

}CT[1000];

struct SymbolTable{

char symbol\_name[100];

char symbol\_type[100];

char array\_dimensions[100];

char class[100];

char value[100];

char params[100];

int line\_number;

int exist;

}ST[1000];

unsigned long hash(unsigned char \*str)

{

unsigned long hash = 5381;

int c;

while (c = \*str++)

hash = ((hash << 5) + hash) + c;

return hash;

}

int search\_ConstantTable(char\* str){

unsigned long temp\_val = hash(str);

int val = temp\_val%1000;

if(CT[val].exist == 0){

return 0;

}

else if(strcmp(CT[val].constant\_name, str) == 0)

{

return 1;

}

else

{

for(int i = val+1 ; i!=val ; i = (i+1)%1000)

{

if(strcmp(CT[i].constant\_name,str)==0)

{

return 1;

}

}

return 0;

}

}

int search\_SymbolTable(char\* str){

unsigned long temp\_val = hash(str);

int val = temp\_val%1000;

if(ST[val].exist == 0){

return 0;

}

else if(strcmp(ST[val].symbol\_name, str) == 0)

{

return 1;

}

else

{

for(int i = val+1 ; i!=val ; i = (i+1)%1000)

{

if(strcmp(ST[i].symbol\_name,str)==0)

{

return 1;

}

}

return 0;

}

}

void insert\_ConstantTable(char\* name, char\* type){

int index = 0;

if(search\_ConstantTable(name)){

return;

}

else{

unsigned long temp\_val = hash(name);

int val = temp\_val%1000;

if(CT[val].exist == 0){

strcpy(CT[val].constant\_name, name);

strcpy(CT[val].constant\_type, type);

CT[val].exist = 1;

return;

}

for(int i = val+1; i != val; i = (i+1)%1000){

if(CT[i].exist == 0){

index = i;

break;

}

}

strcpy(CT[index].constant\_name, name);

strcpy(CT[index].constant\_type, type);

CT[index].exist = 1;

}

}

void insert\_SymbolTable(char\* name, char\* class){

int index = 0;

if(search\_SymbolTable(name)){

return;

}

else{

unsigned long temp\_val = hash(name);

int val = temp\_val%1000;

if(ST[val].exist == 0){

strcpy(ST[val].symbol\_name, name);

strcpy(ST[val].class, class);

ST[val].line\_number = yylineno;

ST[val].exist = 1;

return;

}

for(int i = val+1; i != val; i = (i+1)%1000){

if(ST[i].exist == 0){

index = i;

break;

}

}

strcpy(ST[index].symbol\_name, name);

strcpy(ST[val].class, class);

ST[index].exist = 1;

}

}

void insert\_SymbolTable\_type(char \*str1, char \*str2)

{

for(int i = 0 ; i < 1000 ; i++)

{

if(strcmp(ST[i].symbol\_name,str1)==0)

{

strcpy(ST[i].symbol\_type,str2);

}

}

}

void insert\_SymbolTable\_value(char \*str1, char \*str2)

{

for(int i = 0 ; i < 1000 ; i++)

{

if(strcmp(ST[i].symbol\_name,str1)==0)

{

strcpy(ST[i].value,str2);

}

}

}

void insert\_SymbolTable\_arraydim(char \*str1, char \*dim)

{

for(int i = 0 ; i < 1000 ; i++)

{

if(strcmp(ST[i].symbol\_name,str1)==0)

{

strcpy(ST[i].array\_dimensions,dim);

}

}

}

void insert\_SymbolTable\_functionparam(char \*str1, char \*param)

{

for(int i = 0 ; i < 1000 ; i++)

{

if(strcmp(ST[i].symbol\_name,str1)==0)

{

strcat(ST[i].params," ");

strcat(ST[i].params,param);

}

}

}

void insert\_SymbolTable\_line(char \*str1, int line)

{

for(int i = 0 ; i < 1000 ; i++)

{

if(strcmp(ST[i].symbol\_name,str1)==0)

{

ST[i].line\_number = line;

}

}

}

void printConstantTable(){

printf("%20s | %20s\n", "CONSTANT","TYPE");

for(int i = 0; i < 1000; ++i){

if(CT[i].exist == 0)

continue;

printf("%20s | %20s\n", CT[i].constant\_name, CT[i].constant\_type);

}

}

void printSymbolTable(){

printf("%10s | %18s | %10s | %10s | %10s | %10s | %10s\n","SYMBOL", "CLASS", "TYPE","VALUE","DIMENSIONS","params","LINE NO");

for(int i = 0; i < 1000; ++i){

if(ST[i].exist == 0)

continue;

printf("%10s | %18s | %10s | %10s | %10s | %10s | %d\n", ST[i].symbol\_name, ST[i].class, ST[i].symbol\_type, ST[i].value,ST[i].array\_dimensions,ST[i].params, ST[i].line\_number);

}

}

char current\_id[20];

char current\_type[20];

char current\_value[20];

char current\_function[20];

char previous\_operator[20];

int flag;

%}

num [0-9]

alpha [a-zA-Z]

alphanum {alpha}|{num}

escape\_sequences 0|a|b|f|n|r|t|v|"\\"|"\""|"\'"

ws [ \t\r\f\v]+

%x MLCOMMENT

DE "define"

IN "include"

%%

int nested\_count = 0;

int check\_nested = 0;

\n {yylineno++;}

"#include"[ ]\*"<"{alpha}({alphanum})\*".h>" { }

"#define"[ ]+(\_|{alpha})({alphanum})\*[ ]\*(.)+ { }

"//".\* { }

"/\*" { BEGIN MLCOMMENT; }

<MLCOMMENT>"/\*" { ++nested\_count;

check\_nested = 1;

}

<MLCOMMENT>"\*"+"/" { if (nested\_count) --nested\_count;

else{ if(check\_nested){

check\_nested = 0;

BEGIN INITIAL;

}

else{

BEGIN INITIAL;

}

}

}

<MLCOMMENT>"\*"+ ;

<MLCOMMENT>[^/\*\n]+ ;

<MLCOMMENT>[/] ;

<MLCOMMENT>\n ;

<MLCOMMENT><<EOF>> { printf("Line No. %d ERROR: MULTI LINE COMMENT NOT CLOSED\n", yylineno); return 0;}

"[" {return \*yytext;}

"]" {return \*yytext;}

"(" {return \*yytext;}

")" {return \*yytext;}

"{" {return \*yytext;}

"}" {return \*yytext;}

"," {return \*yytext;}

";" {return \*yytext;}

"char" { strcpy(current\_type,yytext); insert\_SymbolTable(yytext, "Keyword"); return CHAR;}

"double" { strcpy(current\_type,yytext); insert\_SymbolTable(yytext, "Keyword"); return DOUBLE;}

"else" { insert\_SymbolTable\_line(yytext, yylineno); insert\_SymbolTable(yytext, "Keyword"); return ELSE;}

"float" { strcpy(current\_type,yytext); insert\_SymbolTable(yytext, "Keyword");return FLOAT;}

"while" { insert\_SymbolTable(yytext, "Keyword"); return WHILE;}

"do" { insert\_SymbolTable(yytext, "Keyword"); return DO;}

"for" { insert\_SymbolTable(yytext, "Keyword"); return FOR;}

"if" { insert\_SymbolTable(yytext, "Keyword"); return IF;}

"int" { strcpy(current\_type,yytext); insert\_SymbolTable(yytext, "Keyword");return INT;}

"long" { strcpy(current\_type,yytext); insert\_SymbolTable(yytext, "Keyword"); return LONG;}

"return" { insert\_SymbolTable(yytext, "Keyword"); return RETURN;}

"short" { strcpy(current\_type,yytext); insert\_SymbolTable(yytext, "Keyword"); return SHORT;}

"signed" { strcpy(current\_type,yytext); insert\_SymbolTable(yytext, "Keyword"); return SIGNED;}

"sizeof" { insert\_SymbolTable(yytext, "Keyword"); return SIZEOF;}

"struct" { strcpy(current\_type,yytext); insert\_SymbolTable(yytext, "Keyword"); return STRUCT;}

"unsigned" { insert\_SymbolTable(yytext, "Keyword"); return UNSIGNED;}

"void" { strcpy(current\_type,yytext); insert\_SymbolTable(yytext, "Keyword"); return VOID;}

"break" { insert\_SymbolTable(yytext, "Keyword"); return BREAK;}

"continue" { insert\_SymbolTable(yytext, "Keyword"); return CONTINUE;}

"goto" { insert\_SymbolTable(yytext, "Keyword"); return GOTO;}

"switch" { insert\_SymbolTable(yytext, "Keyword"); return SWITCH;}

"case" { insert\_SymbolTable(yytext, "Keyword"); return CASE;}

"default" { insert\_SymbolTable(yytext, "Keyword"); return DEFAULT;}

("\"")[^\n\"]\*("\"") {strcpy(current\_value,yytext); insert\_ConstantTable(yytext,"String Constant"); return string\_constant;}

("\"")[^\n\"]\* { printf("Line No. %d ERROR: UNCLOSED STRING - %s\n", yylineno, yytext); return 0;}

("\'")(("\\"({escape\_sequences}))|.)("\'") {strcpy(current\_value,yytext); insert\_ConstantTable(yytext,"Character Constant"); return character\_constant;}

("\'")(((("\\")[^0abfnrtv\\\"\'][^\n\']\*))|[^\n\''][^\n\'']+)("\'") {printf("Line No. %d ERROR: NOT A CHARACTER - %s\n", yylineno, yytext); return 0; }

{num}+(\.{num}+)?e{num}+ {strcpy(current\_value,yytext); insert\_ConstantTable(yytext, "Floating Constant"); return float\_constant;}

{num}+\.{num}+ {strcpy(current\_value,yytext); insert\_ConstantTable(yytext, "Floating Constant"); return float\_constant;}

{num}+ {strcpy(current\_value,yytext); insert\_ConstantTable(yytext, "Number Constant"); return integer\_constant;}

(\_|{alpha})({alpha}|{alpha}|\_)\* {strcpy(current\_id,yytext);insert\_SymbolTable(yytext,"id"); return id;}

(\_|{alpha})({alpha}|{alpha}|\_)\*/\[ {strcpy(current\_id,yytext);insert\_SymbolTable(yytext,"Array id"); return id;}

{ws} ;

"+" {return \*yytext;}

"-" {return \*yytext;}

"\*" {return \*yytext;}

"/" {return \*yytext;}

"=" {return \*yytext;}

"%" {return \*yytext;}

"&" {return \*yytext; }

"^" {return \*yytext; }

"++" {return INCREMENT;}

"--" {return DECREMENT;}

"!" {return NOT;}

"+=" {return ADD\_EQUAL;}

"-=" {return SUBTRACT\_EQUAL;}

"\*=" {return MULTIPLY\_EQUAL;}

"/=" {return DIVIDE\_EQUAL;}

"%=" {return MOD\_EQUAL;}

"&&" {return AND\_AND;}

"||" {return OR\_OR;}

">" {return GREAT;}

"<" {return LESS;}

">=" {return GREAT\_EQUAL;}

"<=" {return LESS\_EQUAL;}

"==" {return EQUAL;}

"!=" {return NOT\_EQUAL;}

. {

if(yytext[0] == '#')

printf("Line No. %d PREPROCESSOR ERROR - %s\n", yylineno, yytext);

else

printf("Line No. %d ERROR ILLEGAL CHARACTER - %s\n", yylineno, yytext);

exit(0);}

**YACC Parser (parser.y)**

%{

void yyerror(char\* s);

int yylex();

#include "stdio.h"

#include "stdlib.h"

#include "ctype.h"

#include "string.h"

void insert\_type();

void insert\_value();

void insert\_dimensions();

void insert\_params();

int insert\_flag = 0;

extern char current\_id[20];

extern char current\_type[20];

extern char current\_value[20];

extern char current\_function[20];

extern char previous\_operator[20];

%}

%name parse

%nonassoc IF

%token INT CHAR FLOAT DOUBLE LONG SHORT SIGNED UNSIGNED STRUCT

%token RETURN MAIN

%token VOID

%token WHILE FOR DO

%token BREAK CONTINUE GOTO

%token ENDIF

%token SWITCH CASE DEFAULT

%expect 2

%token id

%token integer\_constant string\_constant float\_constant character\_constant

%nonassoc ELSE

%right MOD\_EQUAL

%right MULTIPLY\_EQUAL DIVIDE\_EQUAL

%right ADD\_EQUAL SUBTRACT\_EQUAL

%right '='

%left OR\_OR

%left AND\_AND

%left '^'

%left EQUAL NOT\_EQUAL

%left LESS\_EQUAL LESS GREAT\_EQUAL GREAT

%left '+' '-'

%left '\*' '/' '%'

%right SIZEOF

%right NOT

%left INCREMENT DECREMENT

%start program

%%

program

: declarations;

declarations

: declaration declarations

|

;

declaration

: var\_dec

| function\_dec

| struct\_dec;

struct\_dec

: STRUCT id { insert\_type(); } '{' struct\_content '}' ';';

struct\_content : var\_dec struct\_content | ;

var\_dec

: dtype vars ';'

| struct\_initialize;

struct\_initialize

: STRUCT id vars;

vars

: id\_name multiple\_vars;

multiple\_vars

: vars ','

| ;

id\_name

: id { insert\_type(); } extended\_id;

extended\_id : array\_id | '='{strcpy(previous\_operator,"=");} expression ;

array\_id

: '[' array\_dims

| ;

array\_dims

: integer\_constant {insert\_dimensions();} ']' init

| ']' string\_init;

init

: string\_init

| array\_init

| ;

string\_init

: '='{strcpy(previous\_operator,"=");} string\_constant { insert\_value(); };

array\_init

: '='{strcpy(previous\_operator,"=");} '{' array\_values '}';

array\_values

: integer\_constant multiple\_array\_values;

multiple\_array\_values

: ',' array\_values

| ;

dtype

: INT | CHAR | FLOAT | DOUBLE

| LONG long\_grammar

| SHORT short\_grammar

| UNSIGNED unsigned\_grammar

| SIGNED signed\_grammar

| VOID ;

unsigned\_grammar

: INT | LONG long\_grammar | SHORT short\_grammar | ;

signed\_grammar

: INT | LONG long\_grammar | SHORT short\_grammar | ;

long\_grammar

: INT | ;

short\_grammar

: INT | ;

function\_dec

: function\_dtype function\_params;

function\_dtype

: dtype id '(' {strcpy(current\_function,current\_id); insert\_type();};

function\_params

: params ')' statement;

params

: dtype all\_parameter\_ids | ;

all\_parameter\_ids

: parameter\_id multiple\_params;

multiple\_params

: ',' params

| ;

parameter\_id

: id { insert\_params(); insert\_type(); } extended\_parameter;

extended\_parameter

: '[' ']'

| ;

statement

: expression\_statement | multiple\_statement

| conditional\_statements | loop\_statements

| return\_statement | break\_statement

| var\_dec;

multiple\_statement

: '{' statements '}' ;

statements

: statement statements

| ;

expression\_statement

: expression ';'

| ';' ;

conditional\_statements

: IF '(' simple\_expression ')' statement extended\_conditional\_statements;

extended\_conditional\_statements

: ELSE statement

| ;

loop\_statements

: WHILE '(' simple\_expression ')' statement

| FOR '(' for\_init simple\_expression ';' expression ')'

| DO statement WHILE '(' simple\_expression ')' ';';

for\_init

: var\_dec

| expression ';'

| ';' ;

return\_statement

: RETURN return\_suffix;

return\_suffix

: ';'

| expression ';' ;

break\_statement

: BREAK ';' ;

expression

: identifier expressions

| simple\_expression ;

expressions

: '='{strcpy(previous\_operator,"=");} expression

| ADD\_EQUAL{strcpy(previous\_operator,"+=");} expression

| SUBTRACT\_EQUAL{strcpy(previous\_operator,"-=");} expression

| MULTIPLY\_EQUAL{strcpy(previous\_operator,"\*=");} expression

| DIVIDE\_EQUAL{strcpy(previous\_operator,"/=");} expression

| MOD\_EQUAL{strcpy(previous\_operator,"%=");} expression

| INCREMENT

| DECREMENT ;

simple\_expression

: and\_expression simple\_expression\_breakup;

simple\_expression\_breakup

: OR\_OR and\_expression simple\_expression\_breakup | ;

and\_expression

: unary\_relation\_expression and\_expression\_breakup;

and\_expression\_breakup

: AND\_AND unary\_relation\_expression and\_expression\_breakup

| ;

unary\_relation\_expression

: NOT unary\_relation\_expression

| regex ;

regex

: sum\_expression regex\_breakup;

regex\_breakup

: relops sum\_expression

| ;

relops

: GREAT\_EQUAL{strcpy(previous\_operator,">=");}

| LESS\_EQUAL{strcpy(previous\_operator,"<=");}

| GREAT{strcpy(previous\_operator,">");}

| LESS{strcpy(previous\_operator,"<");}

| EQUAL{strcpy(previous\_operator,"==");}

| NOT\_EQUAL{strcpy(previous\_operator,"!=");} ;

sum\_expression

: sum\_expression sum\_operators term

| term ;

sum\_operators

: '+'

| '-' ;

term

: term multiply\_operators factor

| factor ;

multiply\_operators

: '\*' | '/' | '%' ;

factor

: function | identifier ;

identifier

: id

| identifier extended\_identifier;

extended\_identifier

: '[' expression ']'

| '.' id;

function

: '('{strcpy(previous\_operator,"(");} expression ')'

| function\_call | constant;

function\_call

: id '('{strcpy(previous\_operator,"(");} args ')';

args

: args\_list | ;

args\_list

: expression extended\_args;

extended\_args

: ',' expression extended\_args

| ;

constant

: integer\_constant { insert\_value(); }

| string\_constant { insert\_value(); }

| float\_constant { insert\_value(); }

| character\_constant{ insert\_value(); };

%%

extern FILE \*yyin;

extern int yylineno;

extern char \*yytext;

void insert\_SymbolTable\_type(char \*,char \*);

void insert\_SymbolTable\_value(char \*, char \*);

void insert\_ConstantTable(char \*, char \*);

void insert\_SymbolTable\_arraydim(char \*, char \*);

void insert\_SymbolTable\_functionparam(char \*, char \*);

void printSymbolTable();

void printConstantTable();

int main()

{

yyparse();

printf("%30s SYMBOL TABLE \n", " ");

printf("%30s %s\n", " ", "---------------------------------------------------------------------");

printSymbolTable();

printf("\n\n%30s CONSTANT TABLE \n", " ");

printf("%30s %s\n", " ", "--------------");

printConstantTable();

}

void yyerror(char \*s)

{

printf("Line No. : %d %s %s\n",yylineno, s, yytext);

printf("INVALID PARSE\n");

exit(0);

}

void insert\_type()

{

insert\_SymbolTable\_type(current\_id,current\_type);

}

void insert\_value()

{

if(strcmp(previous\_operator, "=") == 0)

{ insert\_SymbolTable\_value(current\_id,current\_value);

}

}

void insert\_dimensions()

{

insert\_SymbolTable\_arraydim(current\_id, current\_value);

}

void insert\_params()

{

insert\_SymbolTable\_functionparam(current\_function, current\_id);

}

int yywrap()

{

return 1;

**Explanation:**

The lex code is used to detect tokens and generate a stream of tokens from the

input C source code. In the first phase of the project, we only stored the different

symbols and constants their respective tables and printed out the different tokens

with their corresponding line numbers. For this stage, we need to return the

tokens identified by the lexer to the parser so that the parser is able to use it for

further computation. In addition to the functions used in the previous stage, we

added functions to help the parser insert the type, value, function parameter, and

array dimensions into the symbol table.

In the definition section of the yacc program, we include all the required header

files, function definitions and other variables. All the tokens which are returned

by the lexical analyzer are also listed in the order of their precedence in this

section. Operators are also declared here according to their associativity and

precedence. This helps ensure that the grammar given to the parser is

unambiguous.

In this section, grammar rules for the C Programming Language is written. The

grammar rules are written in such a way that there is no left recursion and the

grammar is also deterministic. Non deterministic grammar is converted by

applying left factoring. The grammar productions does the syntax analysis of the

source code. When the complete statement with proper syntax is matched by the

parser, the parser recognizes that it is a valid parse and prints the symbol and

constant table. If the statement is not matched, the parser recognizes that there is

an error and outputs the error along with the line number.

The yyparse() function was called to run the program on the given input file.

After that, both the symbol table and the constant table were printed in order to

show the result.

**Test Cases(Valid Parse):**

// Single line comment

#include<stdio.h>

struct book

{

char name[10];

char author[10];

};

int multiply(int a)

{

return 2\*a;

}

int main()

{

//Single Line Comment

int a;

char es = '\a';

/\* This is the declaration

of an integer value \*/

int a = 5;

int b = multiply(a);

printf("%d ", b);

int A[5] = {1,2,3,4,5};

char B[10] = "Hello";

if(B[0] == 'H'){

if(A[0] == '1')

printf("Hello 1");

else

printf("Hello 2");

}

else

printf("Not Hello");

printf("Hello World");

int num = 3;

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

printf("Hello");

while(num > 0)

{

printf("Hello");

num--;

}

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

{

for(int j = 0; j < num; j++)

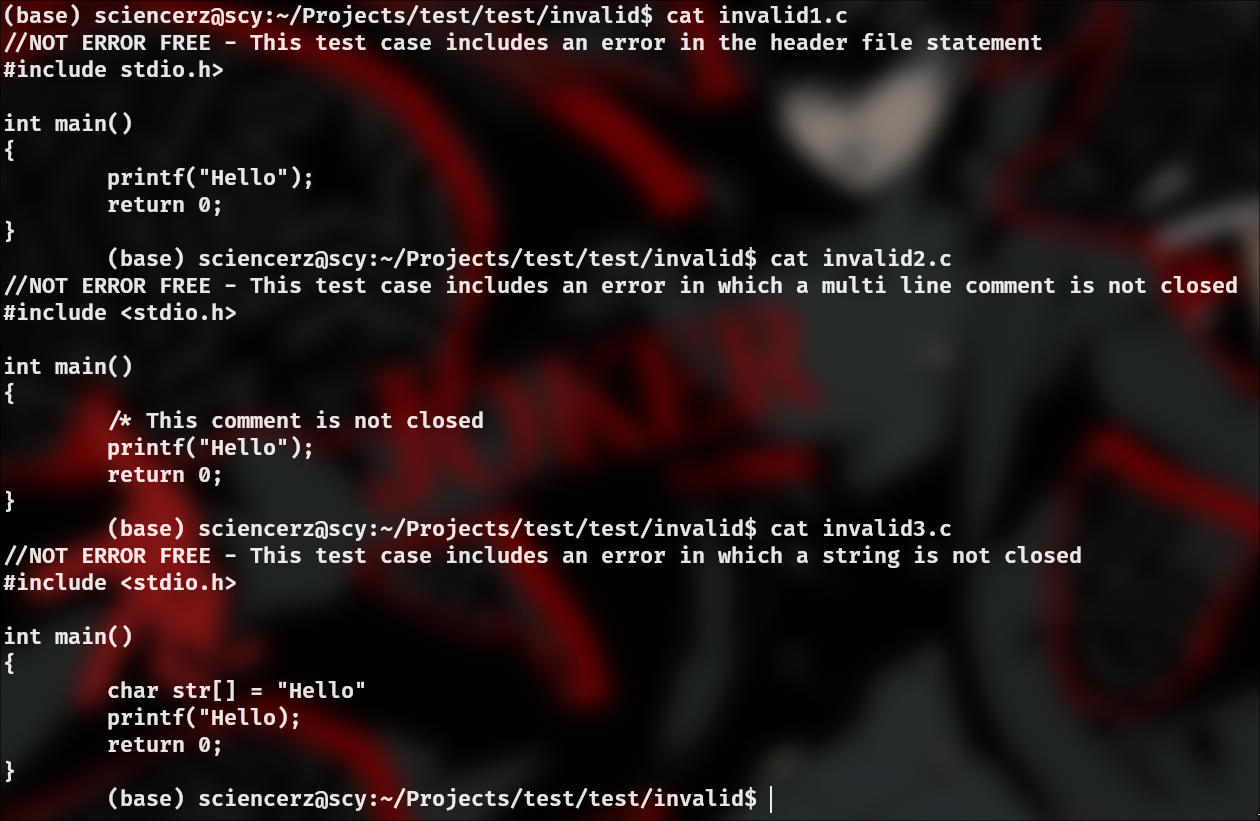
printf("Hello");

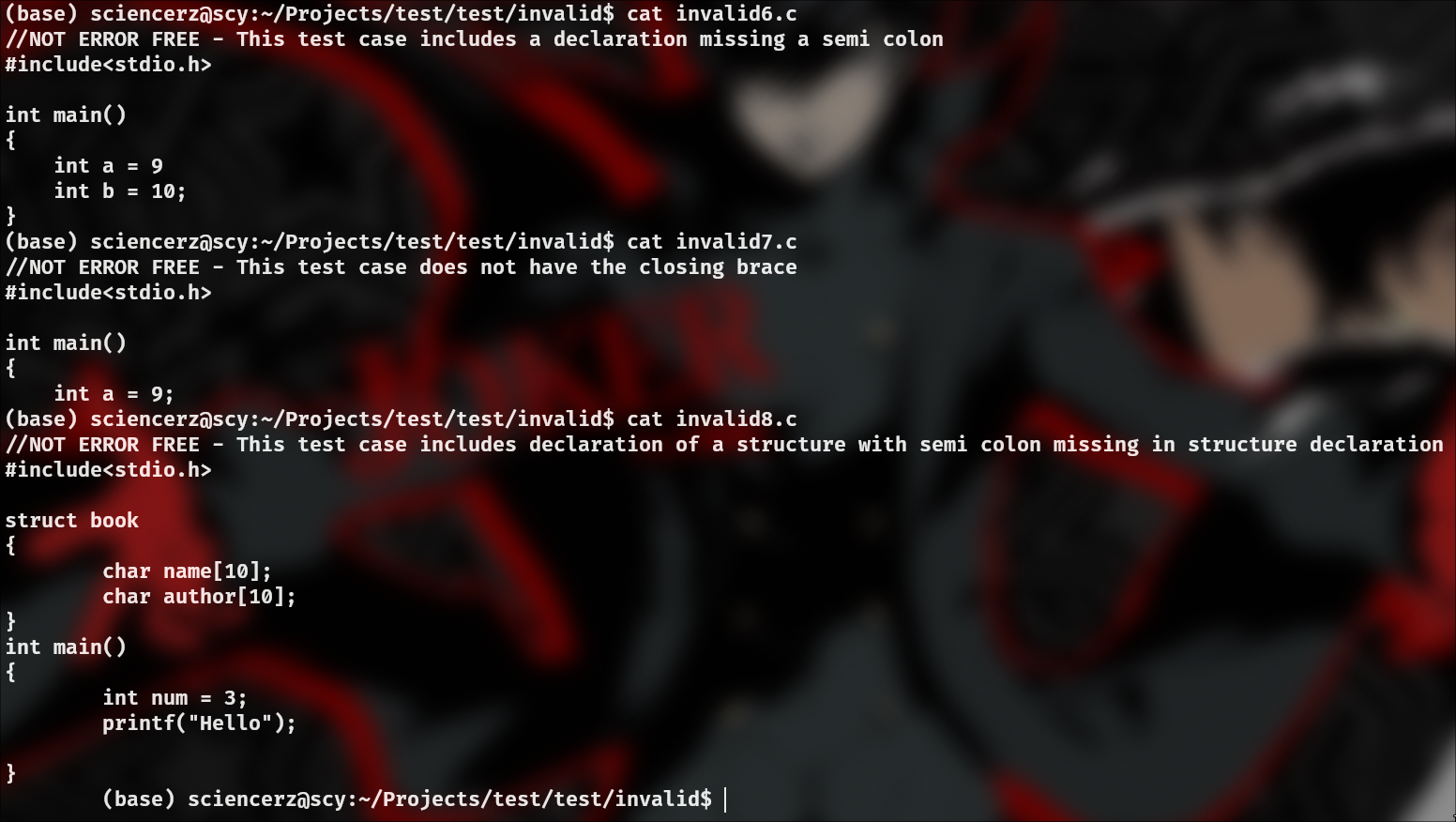
}

return 0;

}

**Test Cases(Invalid Parse):**





**First and Follow Sets**

**First Sets**

First( program ) : { ε, INT, CHAR, FLOAT, DOUBLE, LONG, SHORT, UNSIGNED,

SIGNED, VOID, STRUCT }

First( declarations ) : {ε, INT, CHAR, FLOAT, DOUBLE, LONG,SHORT, UNSIGNED,

SIGNED, VOID, STRUCT}

First( declaration ) : {INT, CHAR, FLOAT, DOUBLE, LONG, SHORT,UNSIGNED,

SIGNED, VOID, STRUCT}

First( struct\_dec ) : {STRUCT}

First( struct\_content ) : {ε, INT, CHAR, FLOAT, DOUBLE, LONG, SHORT,

UNSIGNED, SIGNED, VOID, STRUCT}

First( var\_dec ) : { INT, CHAR, FLOAT, DOUBLE, LONG, SHORT, UNSIGNED,

SIGNED, VOID, STRUCT }

First( struct\_init ) : { STRUCT }

First( vars ) : { id }

First( multiple\_vars ) : { , , ε}

First( function\_dec ) : { INT, CHAR, FLOAT, DOUBLE, LONG, SHORT, UNSIGNED,

SIGNED, VOID}

First( function\_datatype ) : { INT, CHAR, FLOAT, DOUBLE, LONG, SHORT,

UNSIGNED, SIGNED, VOID}

First( function\_parameters ) : { ), ε, INT, CHAR, FLOAT, DOUBLE, LONG, SHORT,

UNSIGNED, SIGNED, VOID}

First( parameters ) : { ε, INT, CHAR, FLOAT, DOUBLE, LONG, SHORT, UNSIGNED,

SIGNED, VOID }

First( all\_parameter\_ids ) : { ) }

First( multiple\_parameters ) : { , , ε }

First( parameter\_id ) : { id}

First( extended\_parameter ) : { [, ε }

First( id\_name ) : { id}

First( extended\_id ) : { =, [, ε }

First( array\_id ) : { [, ε }

First( array\_dims ) : { integer\_constant, ] }

First( init ) : { ε, = }

First( string\_init ) : { = }

First( array\_init) : { = }

First( array\_values ): { integer\_constant }

First( multiple\_array\_values ) : { , , ε }

First( datatype ) : { INT, CHAR, FLOAT, DOUBLE, LONG, SHORT, UNSIGNED,

SIGNED, VOID }

First( unsigned\_grammar ) : { INT, LONG, SHORT, ε }

First( signed\_grammar ) : { INT, LONG, SHORT, ε }

First( long\_grammar ) : { INT, ε }

First( short\_grammar ) : { INT, ε }

First( statement ) : { ;, id, NOT, (, integer\_constant, string\_constant,

float\_constant, character\_constant, IF, RETURN, INT, CHAR, FLOAT, DOUBLE,

LONG, SHORT, UNSIGNED, SIGNED, VOID, STRUCT, {, BREAK, WHILE,FOR, DO }

First( multiple\_statement ) : { { }

First( statements ) : { ε, ;, id, NOT, (, integer\_constant, string\_constant,

float\_constant, character\_constant, IF, RETURN, INT, CHAR, FLOAT, DOUBLE,

LONG, SHORT, UNSIGNED, SIGNED, VOID, STRUCT, {, BREAK, WHILE,FOR, DO }

First( expression\_statement ) : { ;, id, NOT, (, integer\_constant,

string\_constant, float\_constant, character\_constant }

First( conditional\_statements ) : { IF }

First( extended\_conditional\_statements ) : { ELSE, ε }

First( iterative\_statements ) : { WHILE, FOR, DO }

First( for\_init ) : { ;, INT, CHAR, FLOAT, DOUBLE, LONG, SHORT,

UNSIGNED, SIGNED, VOID, STRUCT, id, NOT, (, integer\_constant,

string\_constant, float\_constant, character\_constant }

First( return\_statement ) : { RETURN }

First( return\_suffix ) : { ;, id, NOT, (, integer\_constant, string\_constant,

float\_constant, character\_constant }

First( break\_statement ) : { BREAK }

First( expression ) : { id, NOT, (, integer\_constant, string\_constant,

float\_constant, character\_constant }

First( expressions ) : { =, ADD\_EQUAL, SUBTRACT\_EQUAL, MULTIPLY\_EQUAL,

DIVIDE\_EQUAL, MOD\_EQUAL, INCREMENT, DECREMENT }

First( simple\_expression ) : { NOT, (, id, integer\_constant, string\_constant,

float\_constant, character\_constant }

First( simple\_expression\_breakup ) : { OR\_OR, ε }

First( and\_expression ) : { NOT, (, id, integer\_constant, string\_constant,

float\_constant, character\_constant }

First( and\_expression\_breakup ) : { AND\_AND, ε }

First( unary\_relation\_expression ) : { NOT, (, id, integer\_constant,

string\_constant, float\_constant, character\_constant }

First( regex ) : { (, id, integer\_constant, string\_constant,

float\_constant, character\_constant }

First( regex\_breakup ) : { ε, GREAT\_EQUAL, LESS\_EQUAL, GREAT,

LESS, EQUAL, NOT\_EQUAL }

First( relops ) : { GREAT\_EQUAL, LESS\_EQUAL, GREAT, LESS, EQUAL,

NOT\_EQUAL }

First( sum\_expression ) : { (, id, integer\_constant, string\_constant,

float\_constant, character\_constant }

First( sum\_expression' ) : { ε, +, - }

First( sum\_operators ) : { +, - }

First( term ) : { (, id, integer\_constant, string\_constant, float\_constant,

character\_constant }

First( term' ) : { ε, \*, /, % }

First( multiply\_operators ) : { \*, /, % }

First( factor ) : { (, id, integer\_constant, string\_constant, float\_constant,

character\_constant }

First( identifier ) : { id }

First( identifier’ ) : { ε, [, . }

First( extended\_identifier ) : { [, . }

First( function ) : { (, id, integer\_constant, string\_constant, float\_constant,

character\_constant }

First( func\_call ):{ id }

First( args ):{ ε, id, NOT, (, integer\_constant,string\_constant,

float\_constant, character\_constant }

First( args\_list ):{ id, NOT, (, integer\_constant,string\_constant,

float\_constant, character\_constant }

First( extended\_args ):{ ,, ε }

First( constant ):{ integer\_constant, string\_constant, float\_constant,

character\_constant }

**Follow Sets**

Follow( program ) : { $ }

Follow( declarations ) : { $ }

Follow( declaration ) : {INT, CHAR, FLOAT, DOUBLE, LONG, SHORT, UNSIGNED,

SIGNED, VOID, STRUCT, $}

Follow( struct\_dec ) : {INT, CHAR, FLOAT, DOUBLE, LONG, SHORT, UNSIGNED,

SIGNED, VOID, STRUCT, $}

Follow( struct\_content ) : { }}

Follow( var\_dec ) : { INT, CHAR, FLOAT, DOUBLE, LONG, SHORT, UNSIGNED,

SIGNED, VOID, STRUCT, NOT, (, id, integer\_constant, string\_constant,

float\_constant, character\_constant, WHILE, ELSE, ;, IF, RETURN, {,BREAK, FOR, DO,

}, $}

Follow( struct\_init ) : { INT, CHAR, FLOAT, DOUBLE, LONG, SHORT,

UNSIGNED, SIGNED, VOID, STRUCT, NOT, (, id, integer\_constant,

string\_constant, float\_constant, character\_constant, WHILE, ELSE, ;, IF, RETURN,

{,BREAK, FOR, DO, }, $ }

Follow( vars ) : { ;, INT, CHAR, FLOAT, DOUBLE, LONG, SHORT, UNSIGNED,

SIGNED, VOID, STRUCT, NOT, (, id, integer\_constant, string\_constant,

float\_constant, character\_constant, WHILE, ELSE, IF, RETURN, {,BREAK, FOR, DO, },

$ }

Follow( multiple\_vars ) : { ;, INT, CHAR, FLOAT, DOUBLE, LONG, SHORT,

UNSIGNED, SIGNED, VOID, STRUCT, NOT, (, id, integer\_constant,

string\_constant, float\_constant, character\_constant, WHILE, ELSE, IF, RETURN,

{,BREAK, FOR, DO, }, $}

Follow( function\_dec ) : { INT, CHAR, FLOAT, DOUBLE, LONG, SHORT, UNSIGNED,

SIGNED, VOID, STRUCT, $}

Follow( function\_datatype ) : { ), INT, CHAR, FLOAT, DOUBLE, LONG, SHORT,

UNSIGNED, SIGNED, VOID}

Follow( function\_parameters ):{ INT, CHAR, FLOAT, DOUBLE, LONG, SHORT,

UNSIGNED, SIGNED, VOID, STRUCT, $}

Follow( parameters ) : { ) }

Follow( all\_parameter\_ids ) : { ) }

Follow( multiple\_parameters ) : { )}

Follow( parameter\_id ) : { ,, ) }

Follow( extended\_parameter ): { ,, ) }

Follow( id\_name ) : { ,, ;, INT, CHAR, FLOAT, DOUBLE, LONG, SHORT,

UNSIGNED, SIGNED, VOID, STRUCT, NOT, (, id, integer\_constant,

string\_constant, float\_constant, character\_constant, WHILE, ELSE, IF, RETURN,

{,BREAK, FOR, DO, }, $ }

Follow( extended\_id ) : { ,, ;, INT, CHAR, FLOAT, DOUBLE, LONG, SHORT,

UNSIGNED, SIGNED, VOID, STRUCT, NOT, (, id, integer\_constant,

string\_constant, float\_constant, character\_constant, WHILE, ELSE, IF, RETURN,

{,BREAK, FOR, DO, }, $}

Follow( array\_id ) : { ,, ;, INT, CHAR, FLOAT, DOUBLE, LONG, SHORT,

UNSIGNED, SIGNED, VOID, STRUCT, NOT, (, id, integer\_constant,

string\_constant, float\_constant, character\_constant, WHILE, ELSE, IF, RETURN,

{,BREAK, FOR, DO, }, $ }

Follow( array\_dims ) : { ,, ;, INT, CHAR, FLOAT, DOUBLE, LONG, SHORT, UNSIGNED,

SIGNED, VOID, STRUCT, NOT, (, id, integer\_constant, string\_constant,

float\_constant, character\_constant, WHILE, ELSE, IF, RETURN, {,BREAK, FOR, DO, },

$ }

Follow( init ) : { ,, ;, INT, CHAR, FLOAT, DOUBLE, LONG, SHORT, UNSIGNED,

SIGNED, VOID, STRUCT, NOT, (, id, integer\_constant, string\_constant,

float\_constant, character\_constant, WHILE, ELSE, IF, RETURN, {,BREAK, FOR, DO, },

$}

Follow( string\_init ) : { ,, ;, INT, CHAR, FLOAT, DOUBLE, LONG, SHORT,

UNSIGNED, SIGNED, VOID, STRUCT, NOT, (, id, integer\_constant,

string\_constant, float\_constant, character\_constant, WHILE, ELSE, IF, RETURN,

{,BREAK, FOR, DO, }, $ }

Follow( array\_init) : { ,, ;, INT, CHAR, FLOAT, DOUBLE, LONG, SHORT,

UNSIGNED, SIGNED, VOID, STRUCT, NOT, (, id, integer\_constant,

string\_constant, float\_constant, character\_constant, WHILE, ELSE, IF, RETURN,

{,BREAK, FOR, DO, }, $ }

Follow( array\_values ) : { } }

Follow( multiple\_array\_values ) : { } }

Follow( datatype ) : { id }

Follow( unsigned\_grammar ) : { id }

Follow( signed\_grammar ) : { id }

Follow( long\_grammar ) : { id }

Follow( short\_grammar ) : { id }

Follow( statement ) : { WHILE, ELSE, ;, id, NOT, (, integer\_constant,

string\_constant, float\_constant, character\_constant, IF, RETURN, INT, CHAR,

FLOAT, DOUBLE, LONG, SHORT, UNSIGNED, SIGNED, VOID, STRUCT, {,BREAK, FOR,

DO, }, $ }

Follow( multiple\_statement ) : { WHILE, ELSE, ;, id, NOT, (,

integer\_constant, string\_constant, float\_constant, character\_constant, IF,

RETURN, INT, CHAR, FLOAT, DOUBLE, LONG, SHORT, UNSIGNED, SIGNED, VOID,

STRUCT, {,BREAK, FOR, DO, }, $ }

Follow( statements ) : { } }

Follow( expression\_statement ) : { WHILE, ELSE, ;, id, NOT, (,

integer\_constant, string\_constant, float\_constant, character\_constant, IF,

RETURN, INT, CHAR, FLOAT, DOUBLE, LONG, SHORT, UNSIGNED, SIGNED, VOID,

STRUCT, {,BREAK, FOR, DO, }, $ }

Follow( conditional\_statements ) : { WHILE, ELSE, ;, id, NOT, (,

integer\_constant, string\_constant, float\_constant, character\_constant, IF,

RETURN, INT, CHAR, FLOAT, DOUBLE, LONG, SHORT, UNSIGNED, SIGNED,VOID,

STRUCT, {,BREAK, FOR, DO, }, $ }

Follow( extended\_conditional\_statements ) : { WHILE, ELSE, ;, id, NOT,(,

integer\_constant, string\_constant, float\_constant, character\_constant, IF,

RETURN, INT, CHAR, FLOAT, DOUBLE, LONG, SHORT, UNSIGNED, SIGNED,VOID,

STRUCT, {,BREAK, FOR, DO, }, $ }

Follow( iterative\_statements ) : { WHILE, ELSE, ;, id, NOT, (,

integer\_constant, string\_constant, float\_constant, character\_constant, IF,

RETURN, INT, CHAR, FLOAT, DOUBLE, LONG, SHORT, UNSIGNED, SIGNED,VOID,

STRUCT, {,BREAK, FOR, DO, }, $ }

Follow( for\_init ) : { NOT, (, id, integer\_constant, string\_constant,

float\_constant, character\_constant }

Follow( return\_statement ) : { WHILE, ELSE, ;, id, NOT, (, integer\_constant,

string\_constant, float\_constant, character\_constant, IF, RETURN, INT, CHAR,

FLOAT, DOUBLE, LONG, SHORT, UNSIGNED, SIGNED, VOID, STRUCT, {,BREAK,FOR,

DO, }, $ }

Follow( return\_suffix ):{ WHILE, ELSE, ;, id, NOT, (, integer\_constant,

string\_constant, float\_constant, character\_constant, IF, RETURN, INT, CHAR,

FLOAT, DOUBLE, LONG, SHORT, UNSIGNED, SIGNED, VOID, STRUCT, {,BREAK,FOR,

DO, }, $ }

Follow( break\_statement ) : { WHILE, ELSE, ;, id, NOT, (, integer\_constant,

string\_constant, float\_constant, character\_constant, IF, RETURN, INT, CHAR,

FLOAT, DOUBLE, LONG, SHORT, UNSIGNED, SIGNED, VOID, STRUCT, {,BREAK,FOR,

DO, }, $ }

Follow( expression ) : { ,, ), ], ;, INT, CHAR, FLOAT, DOUBLE, LONG, SHORT,

UNSIGNED, SIGNED, VOID, STRUCT, NOT, (, id, integer\_constant,

string\_constant, float\_constant, character\_constant, WHILE, ELSE, IF, RETURN,

{,BREAK, FOR, DO, }, $ }

Follow( expressions ) : { ,, ), ], ;, INT, CHAR, FLOAT, DOUBLE, LONG, SHORT,

UNSIGNED, SIGNED, VOID, STRUCT, NOT, (, id, integer\_constant,

string\_constant, float\_constant, character\_constant, WHILE, ELSE, IF, RETURN,

{,BREAK, FOR, DO, }, $ }

Follow( simple\_expression ) : { ), ;, ,, ], INT, CHAR, FLOAT, DOUBLE, LONG, SHORT,

UNSIGNED, SIGNED, VOID, STRUCT, NOT, (, id, integer\_constant,

string\_constant, float\_constant, character\_constant, WHILE, ELSE, IF, RETURN,

{,BREAK, FOR, DO, }, $ }

Follow( simple\_expression\_breakup ) : { ), ;, ,, ], INT, CHAR, FLOAT, DOUBLE,

LONG, SHORT, UNSIGNED, SIGNED, VOID, STRUCT, NOT, (, id,

integer\_constant, string\_constant, float\_constant, character\_constant, WHILE,

ELSE, IF, RETURN, {,BREAK, FOR, DO, }, $ }

Follow( and\_expression ) : { OR\_OR, ), ;, ,, ], INT, CHAR, FLOAT, DOUBLE, LONG,

SHORT, UNSIGNED, SIGNED, VOID, STRUCT, NOT, (, id, integer\_constant,

string\_constant, float\_constant, character\_constant, WHILE, ELSE, IF,RETURN, {,

BREAK, FOR, DO, }, $ }

Follow( and\_expression\_breakup ) : { OR\_OR, ), ;, ,, ], INT, CHAR, FLOAT, DOUBLE,

LONG, SHORT, UNSIGNED, SIGNED, VOID, STRUCT, NOT, (, id,

integer\_constant, string\_constant, float\_constant, character\_constant, WHILE,

ELSE, IF,RETURN, {, BREAK, FOR, DO, }, $ }

Follow( unary\_relation\_expression ) : { AND\_AND, OR\_OR, ), ;, ,, ], INT, CHAR,

FLOAT, DOUBLE, LONG, SHORT, UNSIGNED, SIGNED, VOID, STRUCT, NOT, (,

id, integer\_constant, string\_constant, float\_constant, character\_constant,

WHILE,ELSE, IF, RETURN, {, BREAK, FOR, DO, }, $ }

Follow( regex ) : { AND\_AND, OR\_OR, ), ;, ,, ], INT, CHAR, FLOAT,

DOUBLE, LONG, SHORT, UNSIGNED, SIGNED, VOID, STRUCT, NOT, (, id,

integer\_constant, string\_constant, float\_constant, character\_constant,

WHILE,ELSE, IF, RETURN, {, BREAK, FOR, DO, }, $ }

Follow( regex\_breakup ) : { AND\_AND, OR\_OR, ), ;, ,, ], INT, CHAR,

FLOAT, DOUBLE, LONG, SHORT, UNSIGNED, SIGNED, VOID, STRUCT, NOT, (,

id, integer\_constant, string\_constant, float\_constant, character\_constant,

WHILE,ELSE, IF, RETURN, {, BREAK, FOR, DO, }, $ }

Follow( relops ) : { (, id, integer\_constant, string\_constant,

float\_constant, character\_constant }

Follow( sum\_expression ) : { GREAT\_EQUAL, LESS\_EQUAL, GREAT, LESS, EQUAL,

NOT\_EQUAL, AND\_AND, OR\_OR, ), ;, ,, ], INT, CHAR, FLOAT, DOUBLE, LONG,

SHORT, UNSIGNED, SIGNED, VOID, STRUCT, NOT, (, id,integer\_constant,

string\_constant, float\_constant, character\_constant, WHILE, ELSE, IF, RETURN, {,

BREAK, FOR, DO, }, $ }

Follow( sum\_expression' ) : { GREAT\_EQUAL, LESS\_EQUAL, GREAT, LESS, EQUAL,

NOT\_EQUAL, AND\_AND, OR\_OR, ), ;, ,, ], INT, CHAR, FLOAT, DOUBLE, LONG,

SHORT, UNSIGNED, SIGNED, VOID, STRUCT, NOT, (, id,integer\_constant,

string\_constant, float\_constant, character\_constant, WHILE, ELSE, IF, RETURN, {,

BREAK, FOR, DO, }, $ }

Follow( sum\_operators ) : { (, id, integer\_constant, string\_constant,

float\_constant, character\_constant }

Follow( term ) : { +, -, GREAT\_EQUAL, LESS\_EQUAL, GREAT, LESS, EQUAL,

NOT\_EQUAL, AND\_AND, OR\_OR, ), ;, ,, ], INT, CHAR, FLOAT, DOUBLE, LONG,

SHORT, UNSIGNED, SIGNED, VOID, STRUCT, NOT, (, id,integer\_constant,

string\_constant, float\_constant, character\_constant, WHILE, ELSE, IF, RETURN, {,

BREAK, FOR, DO, }, $ }

Follow( term' ) : { +, -, GREAT\_EQUAL, LESS\_EQUAL, GREAT, LESS, EQUAL,

NOT\_EQUAL, AND\_AND, OR\_OR, ), ;, ,, ], INT, CHAR, FLOAT, DOUBLE, LONG,

SHORT, UNSIGNED, SIGNED, VOID, STRUCT, NOT, (, id,integer\_constant,

string\_constant, float\_constant, character\_constant, WHILE, ELSE, IF, RETURN, {,

BREAK, FOR, DO, }, $ }

Follow( multiply\_operators ) : { (, id, integer\_constant, string\_constant,

float\_constant, character\_constant}

Follow( factor ) : { \*, /, %, +, -, GREAT\_EQUAL, LESS\_EQUAL, GREAT, LESS, EQUAL,

NOT\_EQUAL, AND\_AND, OR\_OR, ), ;, ,, ], INT, CHAR, FLOAT, DOUBLE, LONG,

SHORT, UNSIGNED, SIGNED, VOID, STRUCT, NOT, (,id, integer\_constant,

string\_constant, float\_constant, character\_constant, WHILE, ELSE, IF, RETURN, {,

BREAK, FOR, DO, }, $ }

Follow( identifier ) : { =, ADD\_EQUAL, SUBTRACT\_EQUAL, MULTIPLY\_EQUAL,

DIVIDE\_EQUAL, MOD\_EQUAL, INCREMENT, DECREMENT, \*, /, %, +, -,

GREAT\_EQUAL, LESS\_EQUAL, GREAT, LESS, EQUAL, NOT\_EQUAL,AND\_AND,

OR\_OR, ), ;, ,, ], INT, CHAR, FLOAT, DOUBLE, LONG, SHORT, UNSIGNED, SIGNED,

VOID, STRUCT, NOT, (, id, integer\_constant, string\_constant,

float\_constant, character\_constant, WHILE,ELSE, IF, RETURN, {, BREAK, FOR, DO, },

$ }

Follow( identifier’ ) : { =, ADD\_EQUAL, SUBTRACT\_EQUAL, MULTIPLY\_EQUAL,

DIVIDE\_EQUAL, MOD\_EQUAL, INCREMENT, DECREMENT, \*, /, %, +, -,

GREAT\_EQUAL, LESS\_EQUAL, GREAT, LESS, EQUAL, NOT\_EQUAL,AND\_AND,

OR\_OR, ), ;, ,, ], INT, CHAR, FLOAT, DOUBLE, LONG, SHORT, UNSIGNED, SIGNED,

VOID, STRUCT, NOT, (, id, integer\_constant, string\_constant,

float\_constant, character\_constant, WHILE,ELSE, IF, RETURN, {, BREAK, FOR, DO, },

$ }

Follow( extended\_identifier ) : { [, ., =, ADD\_EQUAL, SUBTRACT\_EQUAL,

MULTIPLY\_EQUAL, DIVIDE\_EQUAL, MOD\_EQUAL, INCREMENT, DECREMENT, \*, /,

%, +, -, GREAT\_EQUAL, LESS\_EQUAL, GREAT, LESS, EQUAL,NOT\_EQUAL,

AND\_AND, OR\_OR, ), ;, ,, ], INT, CHAR, FLOAT, DOUBLE, LONG, SHORT, UNSIGNED,

SIGNED, VOID, STRUCT, NOT, (, id, integer\_constant, string\_constant,

float\_constant,character\_constant, WHILE, ELSE, IF, RETURN, {, BREAK, FOR, DO, },

$ }

Follow( function ) : { \*, /, %, +, -, GREAT\_EQUAL, LESS\_EQUAL, GREAT, LESS, EQUAL,

NOT\_EQUAL, AND\_AND, OR\_OR, ), ;, ,, ], INT, CHAR, FLOAT, DOUBLE, LONG,

SHORT, UNSIGNED, SIGNED, VOID, STRUCT, NOT, (,id, integer\_constant,

string\_constant, float\_constant, character\_constant, WHILE, ELSE, IF, RETURN, {,

BREAK, FOR, DO, }, $ }

Follow( func\_call ) : { \*, /, %, +, -, GREAT\_EQUAL, LESS\_EQUAL, GREAT, LESS,

EQUAL, NOT\_EQUAL, AND\_AND, OR\_OR, ), ;, ,, ], INT, CHAR, FLOAT, DOUBLE,

LONG, SHORT, UNSIGNED, SIGNED, VOID, STRUCT, NOT, (,id,

integer\_constant, string\_constant, float\_constant, character\_constant, WHILE,

ELSE, IF, RETURN, {, BREAK, FOR, DO, }, $ }

Follow( args ) : { ) }

Follow( args\_list ) : { ) }

Follow( extended\_args ) : { ) }

Follow( constant ) : { \*, /, %, +, -, GREAT\_EQUAL, LESS\_EQUAL, GREAT, LESS,

EQUAL, NOT\_EQUAL, AND\_AND, OR\_OR, ), ;, ,, ], INT, CHAR, FLOAT, DOUBLE,

LONG, SHORT, UNSIGNED, SIGNED, VOID, STRUCT, NOT, (,id,

integer\_constant, string\_constant, float\_constant, character\_constant, WHILE,

ELSE, IF, RETURN, {, BREAK, FOR, DO, }, $ }

**Implementation**

1. Start Symbol

In our implementation of the grammar for the C language, we use begin\_parse as

the start variable. This is done with the help of %start begin\_parse. In case the

start symbol is not declared explicitly, Yacc automatically assumes the first non

terminal on the left side as the start symbol.

2. Declarations

A C program is essentially made up of a bunch of declarations. Any code is made

up of a function, variable or structure declarations.

declarations -> declaration declarations |

declaration -> variable\_dec | function\_dec | structure\_dec;

3. Variable Declaration

variable\_dec -> datatype variables ';'

| structure\_initialize

variables -> identifier\_name multiple\_variables

multiple\_variables -> ',' variables |

identifier\_name -> identifier extended\_identifier

extended\_identifier -> array\_identifier | '=' expression

array\_identifier -> '[' array\_dims |

array\_dims -> integer\_constant ']' initilization

| ']' string\_initilization;

The above rules are being used to construct variable declaration. The

multiple\_variables rule helps us to declare multiple identifiers in a single

statement and also all statements should end with a semi-colon. These rules also

allow variables to be initialized. These rules also allow declaration and

initialization of array variables.

4. Function Declarations

function\_dec ->function\_datatype function\_parameters

function\_datatype -> datatype identifier '('

function\_parameters -> parameters ')' statement

parameters -> datatype all\_parameter\_identifiers |

all\_parameter\_identifiers -> parameter\_identifier multiple\_parameters

multiple\_parameters -> ',' parameters |

The above rules are used to define as well as declare functions. function\_dec is

used for declarations. It produces all cases of valid declarations with return type,

name and parameter list. To generate all valid cases of parameter list a non-

terminal parameters is used. To generate more than one parameter

all\_parameter\_identifiers and multiple\_parameters work together.

5. Structure Declaration and Initialization

structure\_dec -> STRUCT identifier '{' structure\_content '}' ';'

structure\_content -> variable\_dec structure\_content |

structure\_initialize -> STRUCT identifier variables;

The above production rules are used to identify structure declarations and

initializations.

6. Statements

statement -> expression\_statment | multiple\_statement

| conditional\_statements | iterative\_statements

| return\_statement | break\_statement

| variable\_dec

multiple\_statement -> '{' statments '}'

statments -> statement statements |

The above rules are used to generate any kind of statements in C. These statements usually arise inside functions. Multiple statements with curly brackets

are generated using the multiple\_statement rule. This rule on its own can produce

a list of statements denoted by statements which is made up of multiple

statements or can also be empty.

7. If-Else Statements

conditional\_statements -> IF '(' simple\_expression ')' statement

extended\_conditional\_statements

extended\_conditional\_statements -> ELSE statement |

This rule is used to verify the syntax of all if-else statements. This rule also

handles the dangling else problem. We use the non-terminal

simple\_expression to derive all possible inputs to if statement for evaluation.

The non-terminal statement is used to signify all possible blocks of code

which can come after an if or an else statement.

8. Iterative Statements

iterative\_statements -> WHILE '(' simple\_expression ')' statement

| FOR '(' for\_initialization simple\_expression ';'

expression ')'

| DO statement WHILE '(' simple\_expression ')' ';'

for\_initialization ->variable\_dec

| expression ';'

| ';'

The iterative\_statements production rule is used for identifying all iterative

programs in the C language, here we have included rules for While loops, for

loops and do while loops. The rules follow the simple syntactical specifications

of C. The statement non terminal is used for all code blocks that will follow the

loop statements. The expression and simple\_expression non terminals are used

for identifying statements inside the loop. The for\_initialization non-terminal

handles the initialization of variables inside a for loop.

9. Expressions

expression -> iden expression | simple\_expression

expressions -> '=' expression

| ADD\_EQUAL expression

| SUBTRACT\_EQUAL expression

| MULTIPLY\_EQUAL expression

| DIVIDE\_EQUAL expression

| MOD\_EQUAL expression

| INCREMENT

| DECREMENT

simple\_expression -> and\_expression simple\_expression\_breakup

simple\_expression\_breakup -> OR\_OR and\_expression

simple\_expression\_breakup |

and\_expression -> unary\_relation\_expression and\_expression\_breakup;

and\_expression\_breakup -> AND\_AND unary\_relation\_expression

and\_expression\_breakup |

unary\_relation\_expression -> NOT unary\_relation\_expression

| regular\_expression ;

regular\_expression -> sum\_expression regular\_expression\_breakup;

regular\_expression\_breakup -> relational\_operators sum\_expression |

relational\_operators -> GREAT\_EQUAL

| LESS\_EQUAL

| GREAT

| LESS

| EQUAL

| NOT\_EQUAL

sum\_expression -> sum\_expression sum\_operators term

| term

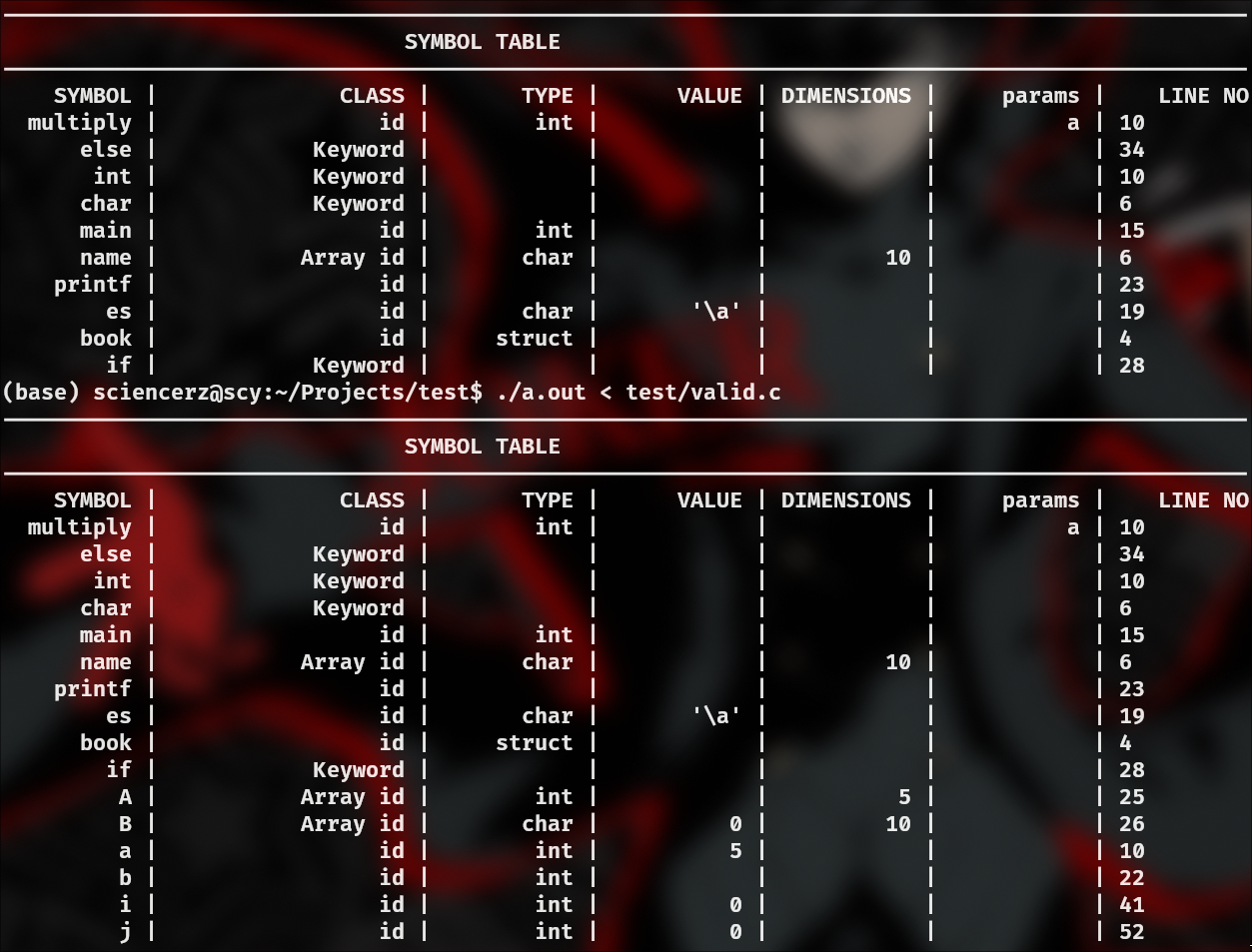
The above rules are pretty straightforward and are used to derive a large

subset of the allowed expressions in the C language. Expression can be an

assignment expression or a simple expression. The above grammar thus

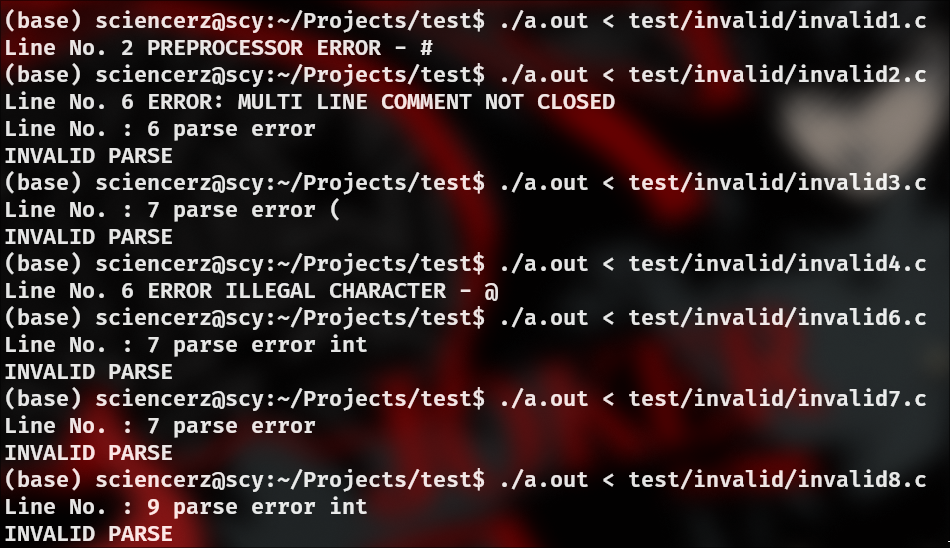
allows multi-assignment statements.

**Result ( Valid Parse )**





**Result (Invalid Parse):**



**Future work:**

The parser was able to successfully parse the tokens recognized by the flex script.

The symbol and constant table are populated and the parser also generated error

messages in the case of invalid parses. Thus, the parsing stage is an essential part

of the compiler and is needed for the simplification of the design of the compiler. It

helps improve the efficiency of the compiler while also speeding up the

compilation process.

**References**

**<https://www.lysator.liu.se/c/ANSI-C-grammar-y.html>**