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:

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.

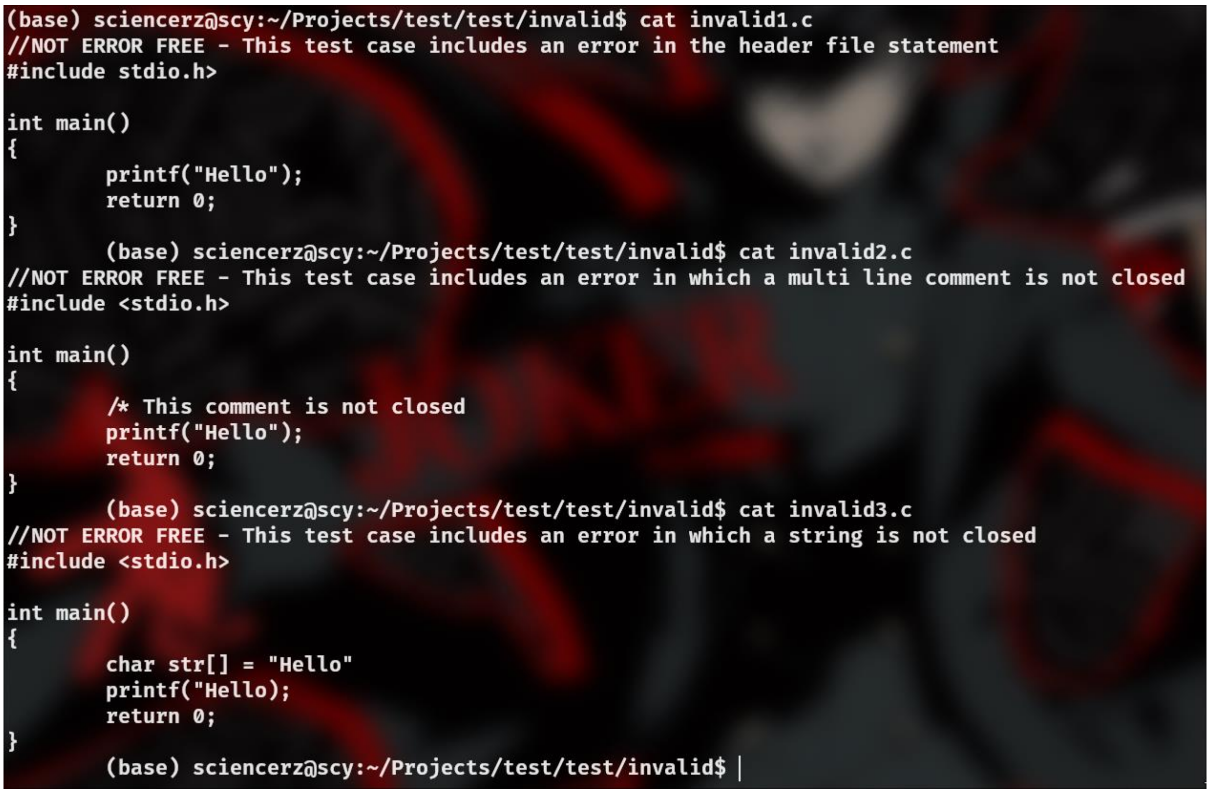
Test Cases(Valid Parse):

```
// Single line comment
#include<stdio.h>

struct book
{
	char name[10];
	char author[10];
};
```

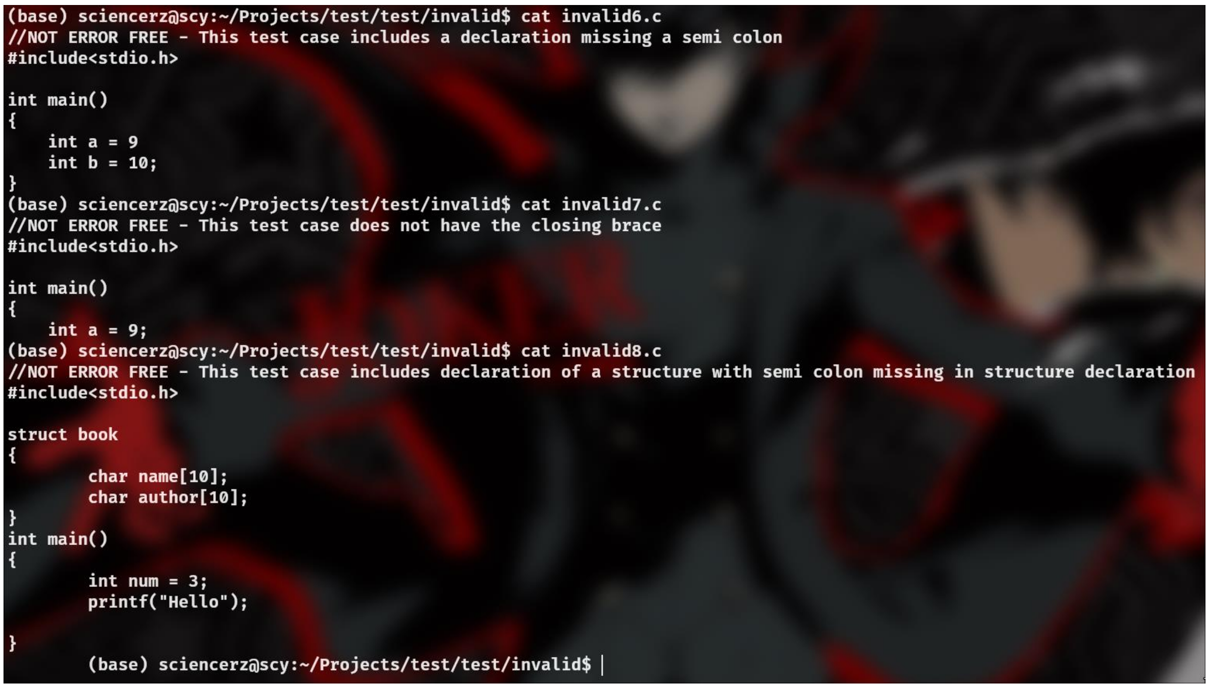
```
int multiply(int a)
{
        return 2*a;
}

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):



```
(base) sciencerz@scy:~/Projects/test/test/invalid$ cat invalid1.c
//NOT ERROR FREE - This test case includes an error in the header file statement
#include stdio.h>

int main()
{
        printf("Hello");
        return 0;
}
        (base) sciencerz@scy:~/Projects/test/test/invalid$ cat invalid2.c
//NOT ERROR FREE - This test case includes an error in which a multi line comment is not closed
#include <stdio.h>

int main()
{
        /* This comment is not closed
        printf("Hello");
        return 0;
}
        (base) sciencerz@scy:~/Projects/test/test/invalid$ cat invalid3.c
//NOT ERROR FREE - This test case includes an error in which a string is not closed
#include <stdio.h>

int main()
{
        char str[] = "Hello"
        printf("Hello);
        return 0;
}
        (base) sciencerz@scy:~/Projects/test/test/invalid$ |
```



```
(base) sciencerz@scy:~/Projects/test/test/invalid$ cat invalid6.c
//NOT ERROR FREE - This test case includes a declaration missing a semi colon
#include<stdio.h>

int main()
{
    int a = 9
    int b = 10;
}
(base) sciencerz@scy:~/Projects/test/test/invalid$ cat invalid7.c
//NOT ERROR FREE - This test case does not have the closing brace
#include<stdio.h>

int main()
{
    int a = 9;
(base) sciencerz@scy:~/Projects/test/test/invalid$ cat invalid8.c
//NOT ERROR FREE - This test case includes declaration of a structure with semi colon missing in structure declaration
#include<stdio.h>

struct book
{
        char name[10];
        char author[10];
}
int main()
{
        int num = 3;
        printf("Hello");

}
        (base) sciencerz@scy:~/Projects/test/test/invalid$ |
```

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 ;

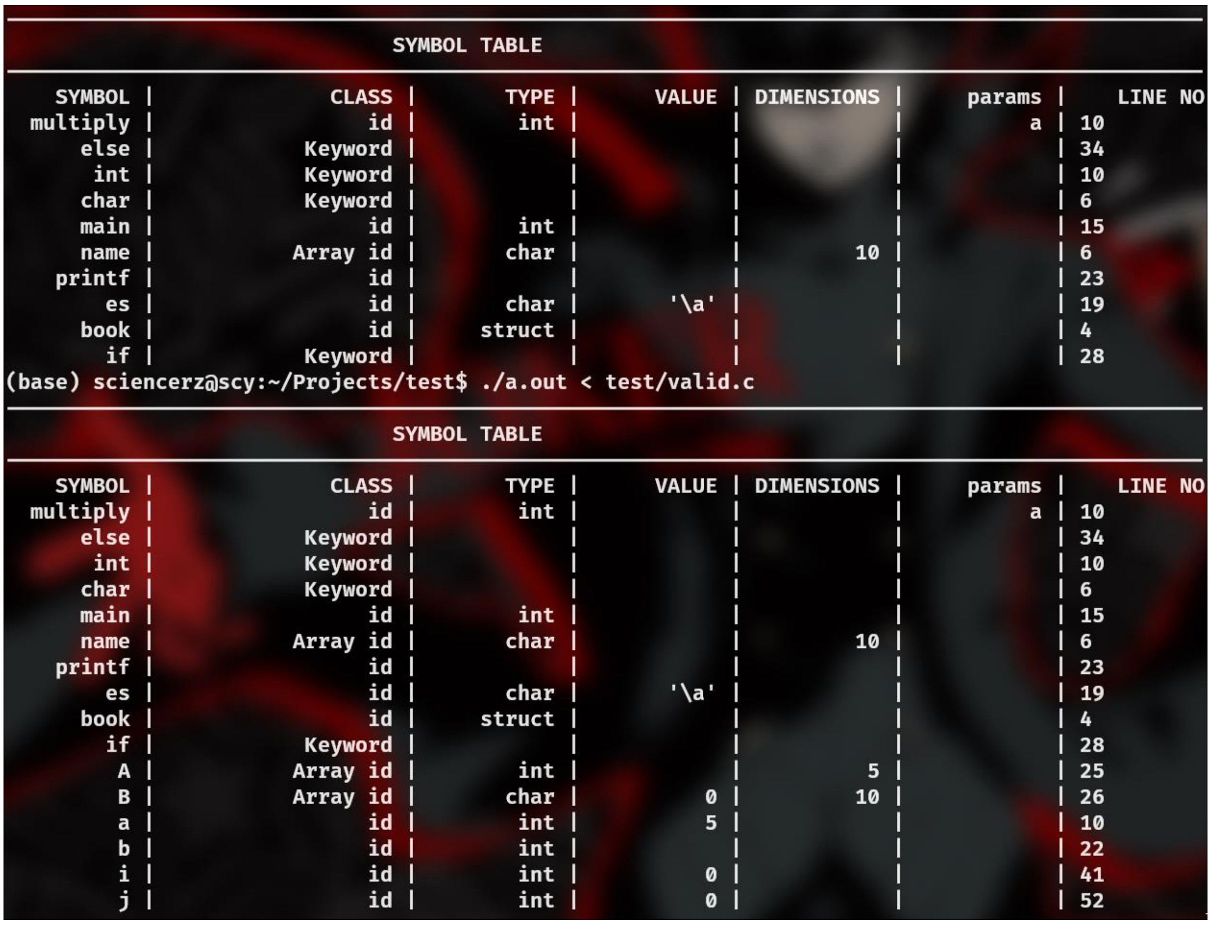
regular_expression -> sum_expression regular_expression_breakup;
regular_expression_breakup -> relational_operators sum_expression |
relational_operators -> GREAT_EQUAL
| LESS_EQUAL
| GREAT
| LESS
| EQUAL
| NOT_EQUAL

sum_expression -> sum_expression sum_operators term

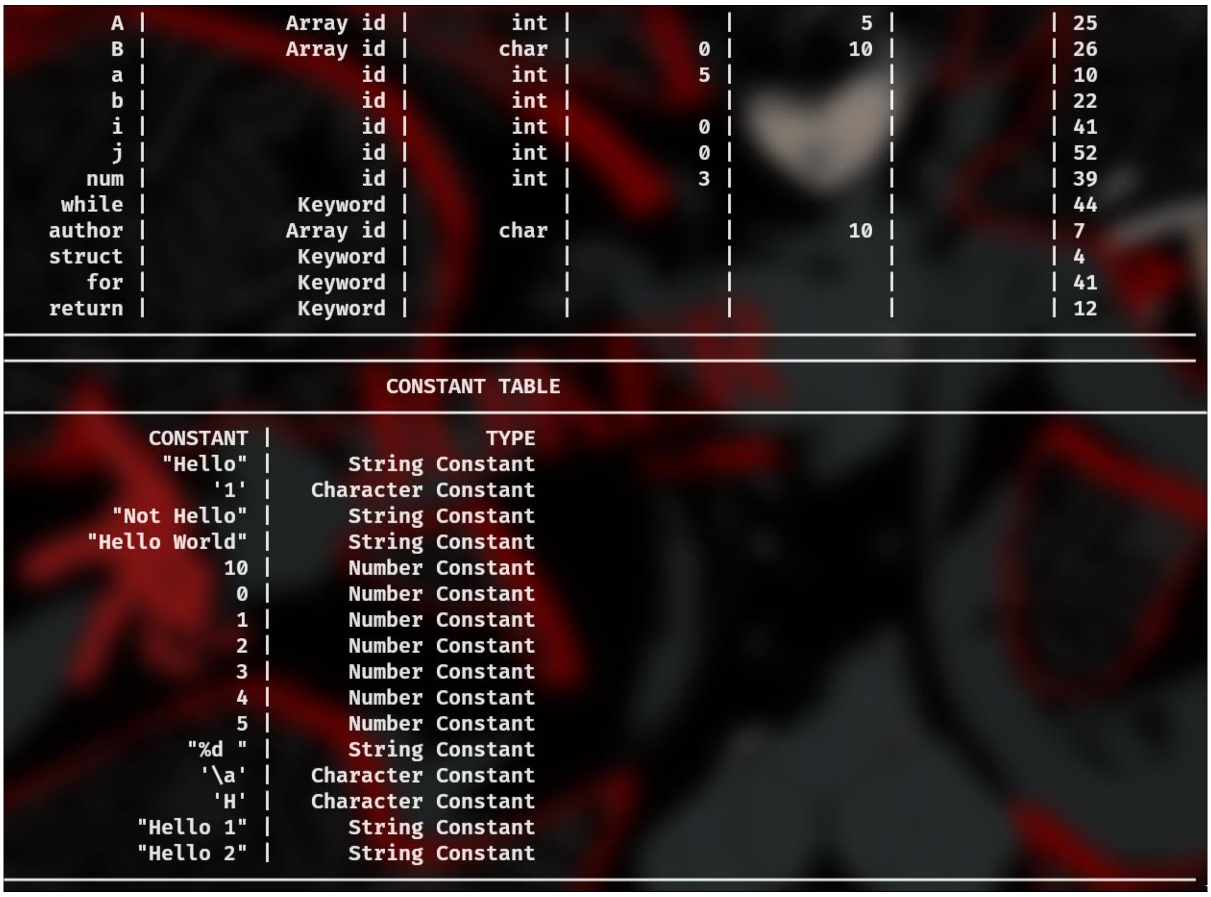
| term

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)

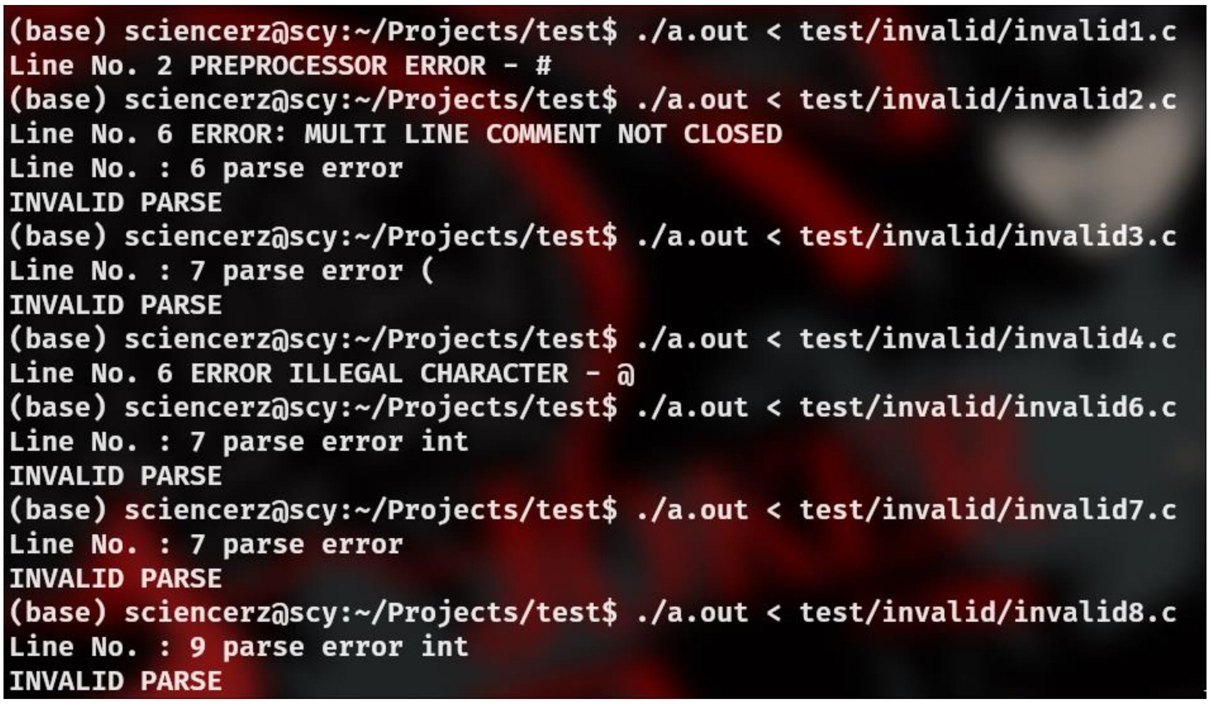


```
                                          SYMBOL TABLE

     SYMBOL |                    CLASS |         TYPE |        VALUE | DIMENSIONS |          params |     LINE NO
   multiply |                       id |          int |              |            |               a |  10
       else |                  Keyword |              |              |            |                 |  34
        int |                  Keyword |              |              |            |                 |  10
       char |                  Keyword |              |              |            |                 |  6
       main |                       id |          int |              |            |                 |  15
       name |                 Array id |         char |              |         10 |                 |  6
     printf |                       id |              |              |            |                 |  23
         es |                       id |         char |         '\a' |            |                 |  19
       book |                       id |       struct |              |            |                 |  4
         if |                  Keyword |              |              |            |                 |  28
(base) sciencerz@scy:~/Projects/test$ ./a.out < test/valid.c

                                          SYMBOL TABLE

     SYMBOL |                    CLASS |         TYPE |        VALUE | DIMENSIONS |          params |     LINE NO
   multiply |                       id |          int |              |            |               a |  10
       else |                  Keyword |              |              |            |                 |  34
        int |                  Keyword |              |              |            |                 |  10
       char |                  Keyword |              |              |            |                 |  6
       main |                       id |          int |              |            |                 |  15
       name |                 Array id |         char |              |         10 |                 |  6
     printf |                       id |              |              |            |                 |  23
         es |                       id |         char |         '\a' |            |                 |  19
       book |                       id |       struct |              |            |                 |  4
         if |                  Keyword |              |              |            |                 |  28
          A |                 Array id |          int |              |          5 |                 |  25
          B |                 Array id |         char |            0 |         10 |                 |  26
          a |                       id |          int |            5 |            |                 |  10
          b |                       id |          int |              |            |                 |  22
          i |                       id |          int |            0 |            |                 |  41
          j |                       id |          int |            0 |            |                 |  52
```



```
          A |                 Array id |          int |              |          5 |                 |  25
          B |                 Array id |         char |            0 |         10 |                 |  26
          a |                       id |          int |            5 |            |                 |  10
          b |                       id |          int |              |            |                 |  22
          i |                       id |          int |            0 |            |                 |  41
          j |                       id |          int |            0 |            |                 |  52
        num |                       id |          int |            3 |            |                 |  39
      while |                  Keyword |              |              |            |                 |  44
     author |                 Array id |         char |              |         10 |                 |  7
     struct |                  Keyword |              |              |            |                 |  4
        for |                  Keyword |              |              |            |                 |  41
     return |                  Keyword |              |              |            |                 |  12

                                          CONSTANT TABLE

        CONSTANT |                 TYPE
         "Hello" |      String Constant
             '1' |   Character Constant
     "Not Hello" |      String Constant
   "Hello World" |      String Constant
              10 |      Number Constant
               0 |      Number Constant
               1 |      Number Constant
               2 |      Number Constant
               3 |      Number Constant
               4 |      Number Constant
               5 |      Number Constant
           "%d " |      String Constant
            '\a' |   Character Constant
             'H' |   Character Constant
       "Hello 1" |      String Constant
       "Hello 2" |      String Constant
```

Result (Invalid Parse):



```
(base) sciencerz@scy:~/Projects/test$ ./a.out < test/invalid/invalid1.c
Line No. 2 PREPROCESSOR ERROR - #
(base) sciencerz@scy:~/Projects/test$ ./a.out < test/invalid/invalid2.c
Line No. 6 ERROR: MULTI LINE COMMENT NOT CLOSED
Line No. : 6 parse error
INVALID PARSE
(base) sciencerz@scy:~/Projects/test$ ./a.out < test/invalid/invalid3.c
Line No. : 7 parse error (
INVALID PARSE
(base) sciencerz@scy:~/Projects/test$ ./a.out < test/invalid/invalid4.c
Line No. 6 ERROR ILLEGAL CHARACTER - @
(base) sciencerz@scy:~/Projects/test$ ./a.out < test/invalid/invalid6.c
Line No. : 7 parse error int
INVALID PARSE
(base) sciencerz@scy:~/Projects/test$ ./a.out < test/invalid/invalid7.c
Line No. : 7 parse error
INVALID PARSE
(base) sciencerz@scy:~/Projects/test$ ./a.out < test/invalid/invalid8.c
Line No. : 9 parse error int
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