

Parser for the C Language



National of Technology Karnataka Surathkal

Date: 23-02-2020

Submitted To: Dr. P. Santhi Thilagam

Group Members:

Veena Nagar (17CO151)

Rounak Modi (17CO236)

Shweta Hariharan Iyer(17CO245)

Abstract:

Aim : Using yacc to implement a syntax analyser/ parser for a subset of the C language.

This report contains the details of the tasks finished as a part of the Phase Two of Compiler Lab. We have developed a Parser for C language which makes use of the C lexer to parse the given C input file.

A parser is a compiler or interpreter component that breaks data into smaller elements for easy translation into another language. A parser takes input in the form of a sequence of tokens or program instructions and usually builds a data structure in the form of a parse tree or an abstract syntax tree. The parser generates a list of identifiers and functions with their types and also specifies syntax errors if any. The parser code has functionality of taking input through a file or through standard input. This makes it more user friendly and efficient at the same time.

It will have the following features:

- Check the syntax of variable declaration of int, float and char type and also short, long, signed, unsigned subtypes.
- Declaration of arrays with specified datatype (eg: int arr[10]).
- Proper usage of looping constructs such as while, nested while.
- Error display with suitable messages for syntax analyser.
- Precedence and associativity of the operators.
- Identifying valid and invalid strings by displaying error messages.
- Displaying identifiers, their type and their line number.

Result :

- Displaying error in the source program with program name, error line number and error message.
- Output file contains the parsed table, the symbol table and constant table of the source program.

Tools : Flex and Yacc.

Contents:

- Introduction
 - Parser / Syntactic analysis
 - Yacc Script
 - C program
- Design of programs
 - Lexer code
 - Parser code
 - Execution of the code
- Test Case Table (Expected)
 - Without error
 - With error
- Implementation
- 5 Test Cases: Sample Programs with most features of C covered (valid C programs)
- 5 Test Cases: Sample Programs with most features of C covered (invalid C programs)
- Results and Conclusion
- Future work
- References

List of Figures and Tables:

1. Table 1: 5 Test Cases without errors
2. Table 2: 5 Test cases with errors
3. Figures : 5 Sample programs without error

(Each program contains the following figures :

Input file, Execution status, parse table, symbol table, constant table)

4. Figures : 5 Sample programs with error

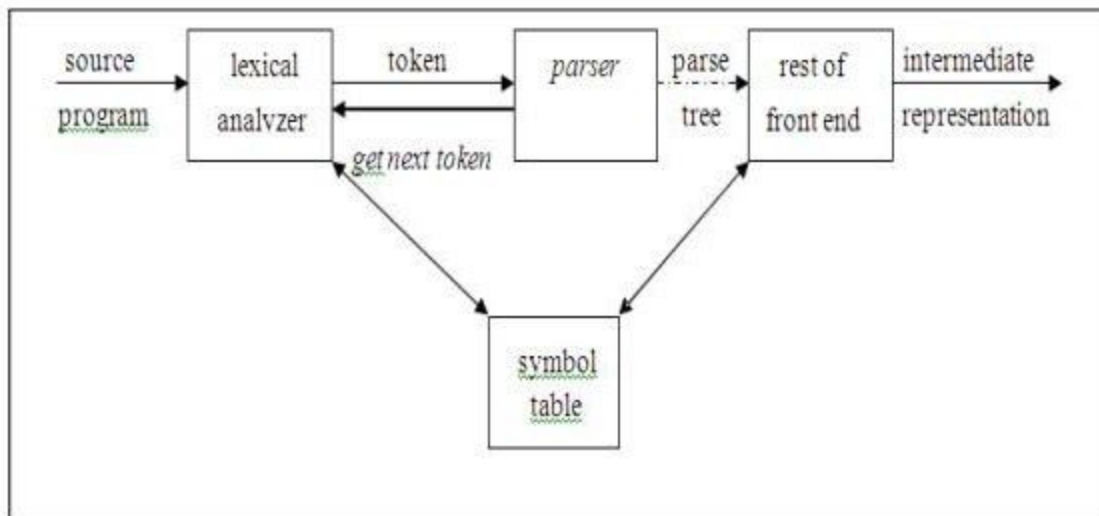
(Each program contains the following figures :Input file, Execution status)

Introduction

Parser/Syntactic Analysis

Parsing or syntactic analysis is the process of analyzing a string of symbols, either in natural language, computer languages or data structures, conforming to the rules of a formal grammar.

A parser is a software component that takes input data and builds a data structure- often some kind of parse tree. The parser obtains a string of tokens from the lexical analyzer and verifies that the string of token names can be generated by the grammar for the source language. We expect the parser to report any syntax errors in an intelligible fashion and to recover from commonly occurring errors to continue processing the remainder of the program. Conceptually, for well-formed programs, the parser constructs a parse tree using the predefined Grammar of the language and the input string. and passes it to the rest of the compiler for further processing.



If the given input string can be produced with the help of the syntax tree (in the derivation process), the input string is found to be in the correct syntax. There are three general types of parsers for grammars: universal, top-down, and bottom-up.

The most efficient top-down and bottom-up methods work only for sub-classes of grammars, but several of these classes, particularly, LL and LR grammars, are expressive enough to describe most of the syntactic constructs in modern programming languages.

Yacc Script

Yacc provides a general tool for describing the input to a computer program. The Yacc user specifies the structures of his input, together with code to be invoked as each such structure is recognized. Yacc turns such a specification into a subroutine that handles the input process; frequently, it is convenient and appropriate to have most of the flow of control in the user's application handled by this subroutine.

The input subroutine produced by Yacc calls a user-supplied routine to return the next basic input item. Thus, the user can specify his input in terms of individual input characters, or in terms of higher level constructs such as names and numbers. The user-supplied routine may also handle idiomatic features such as comment and continuation conventions, which typically defy easy grammatical specification.

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 given below; files are divided into three sections, separated by lines that contain only two percent signs, as follows:

Definition section

%%

Rules section

%% *C code section*

The definition section defines macros and imports header files written in C. It is also possible to write any C code here, which will be copied verbatim into the generated source file.

The rules section associates regular expression patterns with C statements. When the lexer sees text in the input matching a given pattern, it will execute the associated C code.

The C code section contains C statements and functions that are copied verbatim to the generated source file. These statements presumably contain code called by the rules in the rules section. In large programs it is more convenient to place this code in a separate

file linked in at compile time.

C Program

This section describes the input C program which is fed to the yacc script for parsing. The workflow is explained as under:

- Compile the script using Yacc tool
 - `$ yacc -d c_parser.y`
- Compile the flex script using Flex tool
 - `$ flex c_lexer.l`
- After compiling the lex file, `lex.yy.c` file is generated. Also, `y.tab.c` and `y.tab.h` files are generated after compiling the yacc script.
- The three files, `lex.yy.c`, `y.tab.c` and `y.tab.h` are compiled together with the options `-ll` and `-ly`
 - `$ gcc -o compiler lex.yy.c y.tab.h y.tab.c -ll -ly`
- The executable file is generated, which on running parses the C file given as a command line input
 - `$./compiler test.c`

Design of Programs

Lexer Code:

*****Lexer Code*****

```
%{  
#include <stdio.h>  
#include "y.tab.h"  
int lineCount=1;  
int nestedCommentCount=0;  
int commentFlag=0;  
char *tablePtr;  
void addToken(char*);
```

```

%}
digit          [0-9]
letter         [a-zA-Z_]
hex            [a-fA-F0-9]
E              [Ee][+-]?{digit}+
FS             (f|F|l|letter)
IS             (u|U|l|letter)*
singlelineComment (\\/\\.*)
multilineCommentStart (\\/\\*)
multilineCommentEnd (\\*\\/)
%x DETECT_COMMENT
%%

{singlelineComment}          { }

{multilineCommentStart}      { BEGIN(DETECT_COMMENT);
                               nestedCommentCount++; }

<DETECT_COMMENT>{multilineCommentStart} { nestedCommentCount++;
                                           if(nestedCommentCount>1)
                                             commentFlag = 1;
                                           }

<DETECT_COMMENT>{multilineCommentEnd}    { BEGIN(INITIAL);
                                           if(nestedCommentCount>0)
                                             nestedCommentCount--;
                                           if(nestedCommentCount==0)
                                             BEGIN(INITIAL);}

<DETECT_COMMENT>\n          {lineCount++;}
<DETECT_COMMENT>.          {}

```

```
"auto"           { return(AUTO); }
"break"          { return(BREAK); }
"case"           { return(CASE); }
"char"           { return(CHAR); }
"const"          { return(CONST); }
"continue"       { return(CONTINUE); }
"default"        { return(DEFAULT); }
"do"             { return(DO); }
"double"         { return(DOUBLE); }
"else"           { return(ELSE); }
"enum"           { return(ENUM); }
"extern"         { return(EXTERN); }
"float"          { return(FLOAT); }
"for"            { return(FOR); }
"go to"          { return(GOTO); }
"if"             { return(IF); }
"int"            { return(INT); }
"long"           { return(LONG); }
"register"       { return(REGISTER); }
"return"         { return(RETURN); }
"short"         { return(SHORT); }
"signed"         { return(SIGNED); }
"sizeof"         { return(SIZEOF); }
"static"         { return(STATIC); }
"struct"         { return(STRUCT); }
"switch"         { return(SWITCH); }
```



```

"typedef"      { return(TYPDEF); }
"union"        { return(UNION); }
"unsigned"     { return(UNSIGNED); }
"void"         { return(VOID); }
"volatile"     { return(VOLATILE); }
"while"        { return(WHILE); }

{letter}({letter}|{digit})*  { addToken(yytext);
return(IDENTIFIER); }

{letter}?\"(\\.|[^\"])*\"    { addToken(yytext);
return(STRING_LITERAL); }

0[xX]{hex}+{IS}?            { addToken(yytext);
return(CONSTANT); }

0{digit}+{IS}?              { addToken(yytext);
return(CONSTANT); }

{hex}+{IS}?                 { addToken(yytext);
return(CONSTANT); }

{letter}?' (\\.|[^\'] )+'    { addToken(yytext);
return(CONSTANT); }

{digit}+{E}{FS}?            { addToken(yytext);
return(CONSTANT); }

{digit}*\".\"{digit}+({E})?{FS}? { addToken(yytext);
return(CONSTANT); }

{digit}+\".\"{digit}*({E})?{FS}? { addToken(yytext);
return(CONSTANT); }

```

```

"..."      { return(ELLIPSIS); }
">>="      { return(RIGHT_ASSIGN); }
"<<="      { return(LEFT_ASSIGN); }
"+="        { return(ADD_ASSIGN); }
"-="        { return(SUB_ASSIGN); }
"*="        { return(MUL_ASSIGN); }
"/="        { return(DIV_ASSIGN); }
"%="        { return(MOD_ASSIGN); }
"&="        { return(AND_ASSIGN); }
"^="        { return(XOR_ASSIGN); }
"|="        { return(OR_ASSIGN); }
">>"        { return(RIGHT_OP); }
"<<"        { return(LEFT_OP); }
"++"        { return(INC_OP); }
"--"        { return(DEC_OP); }
"->"        { return(PTR_OP); }
"&&"        { return(AND_OP); }
"||"        { return(OR_OP); }
"<="        { return(LE_OP); }
">="        { return(GE_OP); }
"=="        { return(EQ_OP); }
"!="        { return(NE_OP); }
";"         { return(';'); }
("{" | "<%" ) { makeList("{", 'p', lineCount); return('{'); }
("}" | "%>") { makeList("}", 'p', lineCount); return('}'); }
","         { return(','); }
":"         { return(':'); }

```

```

"="          { return('='); }
"("          { return('('); }
")"          { return(')'); }
("[ " | "<:") { return('['); }
("] " | ">:") { return(']'); }
"."          { return('.'); }
"&"          { return('&'); }
"!"          { return('!'); }
"~"          { return('~'); }
"-"          { return('-'); }
"+"          { return('+'); }
"*"          { return('*'); }
"/"          { return('/'); }
"%"          { return('%'); }
"<"          { return('<'); }
">"          { return('>'); }
"^"          { return('^'); }
"|"          { return('|'); }
"?"          { return('?'); }
"#include"(.)*"\n" { lineCount++; }
"#define"(.)*"\n"  { lineCount++; }
[ ]          {}
[\t\v\f]     {}
[\n]         {lineCount++;}
.            { }
%%
yywrap()

```

```
{
    return(1);
}

void addToken(char *yytext)
{
    int len = strlen(yytext);
    tablePtr = (char*)malloc((len+1)*sizeof(char));
    strcpy(tablePtr, yytext);
}
```

Parser Code:

*****Parser Code here*****

```
%{    #include <stdio.h>

#include <string.h>

#include <stdlib.h>

#include "y.tab.h"

struct tokenList

{
    char *token, type[20], line[100];

    struct tokenList *next;
};
```

```
typedef struct tokenList tokenList;
```

```
extern FILE *yyin;
```

```
extern int lineCount;
```

```
extern char *tablePtr;
```

```
extern int nestedCommentCount;
```

```
extern int commentFlag;
```

```
char typeBuffer=' ';
```

```
tokenList *symbolPtr = NULL;
```

```
tokenList *constantPtr = NULL;
```

```
tokenList *parsedPtr=NULL;
```

```
char *sourceCode=NULL;
```

```
int errorFlag=0;
```

```
void makeList(char *,char,int);
```

```
%}
```

```
%token  AUTO BREAK  CASE CHAR  CONST  CONTINUE  DEFAULT  DO DOUBLE
```

ELSE ENUM

%token EXTERN FLOAT FOR GOTO IF INT LONG REGISTER RETURN SHORT

SIGNED

%token SIZEOF STATIC STRUCT SWITCH TYPEDEF UNION UNSIGNED VOID

VOLATILE WHILE

%token IDENTIFIER

%token CONSTANT STRING_LITERAL

%token ELLIPSIS

%token PTR_OP INC_OP DEC_OP LEFT_OP RIGHT_OP LE_OP GE_OP EQ_OP

NE_OP

%token AND_OP OR_OP MUL_ASSIGN DIV_ASSIGN MOD_ASSIGN ADD_ASSIGN

%token SUB_ASSIGN LEFT_ASSIGN RIGHT_ASSIGN AND_ASSIGN

%token XOR_ASSIGN OR_ASSIGN TYPE_NAME

%nonassoc LOWER_THAN_ELSE

%nonassoc ELSE

```
%start translation_unit
```

```
%%
```

```
primary_expression
```

```
    : IDENTIFIER          { makeList(tablePtr, 'v', lineCount); }
```

```
    | CONSTANT           { makeList(tablePtr, 'c', lineCount); }
```

```
    | STRING_LITERAL      { makeList(tablePtr, 's', lineCount); }
```

```
    | '(' expression ')'  { makeList("(", 'p', lineCount);
```

```
makeList(")", 'p', lineCount); };
```

```
postfix_expression
```

```
    : primary_expression
```

```
    | postfix_expression '[' expression ']'          {
```

```
makeList("[", 'p', lineCount); makeList("]", 'p', lineCount); }
```

```
    | postfix_expression '(' ')'                    { makeList("(",
```

```
'p', lineCount); makeList(")", 'p', lineCount); }
```

```
    | postfix_expression '(' argument_expression_list ')' {
```

```
makeList("(", 'p', lineCount); makeList(")", 'p', lineCount); }
```

```
    | postfix_expression '.' IDENTIFIER          {
```

```
makeList(tablePtr, 'v', lineCount);}
```

```
    | postfix_expression PTR_OP IDENTIFIER        {
```

```
makeList(tablePtr, 'v', lineCount);}
```

```
    | postfix_expression INC_OP                    {
```

```
makeList(tablePtr, 'o', lineCount);}
```

```
    | postfix_expression DEC_OP                    {
```

```
makeList(tablePtr, 'o', lineCount);}    ;
```

```
argument_expression_list
```

```
    : assignment_expression
```

```
    | argument_expression_list ',' assignment_expression {
```

```
makeList(",", 'p', lineCount); }    ;
```

```
unary_expression
```

```
    : postfix_expression
```



```

        | INC_OP unary_expression      { makeList("++", 'o',
lineCount); }

        | DEC_OP unary_expression      { makeList("--", 'o',
lineCount); }

        | unary_operator cast_expression

        | SIZEOF unary_expression      { makeList("sizeof", 'o',
lineCount); }

        | SIZEOF '(' type_name ')'      { makeList("sizeof", 'o',
lineCount); }

                                { makeList("(", 'p', lineCount);

makeList(")", 'p', lineCount); }    ;

unary_operator

: '&' { makeList("&", 'o', lineCount); }

| '*' { makeList("*", 'o', lineCount); }

| '+' { makeList("+", 'o', lineCount); }

```

```

| '-' { makeList("-", 'o', lineCount); }

| '~' { makeList("~", 'o', lineCount); }

| '!' { makeList("!", 'o', lineCount); }

;

```

cast_expression

```

: unary_expression

| '(' type_name ')' cast_expression { makeList("(", 'p',
lineCount); makeList(")", 'p', lineCount); } ;

```

multiplicative_expression

```

: cast_expression

| multiplicative_expression '*' cast_expression {
makeList("*", 'o', lineCount); }

| multiplicative_expression '/' cast_expression {
makeList("/", 'o', lineCount); }

```

```

        | multiplicative_expression '%' cast_expression {
makeList("%", 'o', lineCount); }      ;

additive_expression

        : multiplicative_expression

        | additive_expression '+' multiplicative_expression {
makeList("+", 'o', lineCount); }

        | additive_expression '-' multiplicative_expression {
makeList("-", 'o', lineCount); }      ;

shift_expression

        : additive_expression

        | shift_expression LEFT_OP additive_expression      {
makeList("<<", 'o', lineCount); }

        | shift_expression RIGHT_OP additive_expression {
makeList(">>", 'o', lineCount); }      ;

relational_expression

```

```

        : shift_expression

        | relational_expression '<' shift_expression

        | relational_expression '>' shift_expression

        | relational_expression LE_OP shift_expression {
makeList("<=", 'o', lineCount); }

        | relational_expression GE_OP shift_expression {
makeList(">=", 'o', lineCount); };

equality_expression

        : relational_expression

        | equality_expression EQ_OP relational_expression {
makeList("==", 'o', lineCount); }

        | equality_expression NE_OP relational_expression {
makeList("!= ", 'o', lineCount); }    ;

and_expression

        : equality_expression

```

```

        | and_expression '&' equality_expression      { makeList("&",
'o', lineCount);}    ;

```

exclusive_or_expression

```

        : and_expression

        | exclusive_or_expression '^' and_expression      {
makeList("^", 'o', lineCount); };

```

inclusive_or_expression

```

        : exclusive_or_expression

        | inclusive_or_expression '|' exclusive_or_expression {
makeList("|", 'o', lineCount); };

```

logical_and_expression

```

        : inclusive_or_expression

        | logical_and_expression AND_OP inclusive_or_expression {
makeList("&&", 'o', lineCount); };

```

logical_or_expression

```

        : logical_and_expression

        | logical_or_expression OR_OP logical_and_expression {
makeList("||", 'o', lineCount); };

conditional_expression

        : logical_or_expression

        | logical_or_expression '?' expression ':'

conditional_expression { makeList("?:",'o', lineCount); };

assignment_expression

        : conditional_expression

        | unary_expression assignment_operator assignment_expression;

assignment_operator

        : '='          { makeList("=", 'o', lineCount); }

        | MUL_ASSIGN   { makeList("*=", 'o', lineCount); }

        | DIV_ASSIGN   { makeList("/=", 'o', lineCount); }

        | MOD_ASSIGN   { makeList("%=", 'o', lineCount); }

```

```

| ADD_ASSIGN    { makeList("+=", 'o', lineCount); }

| SUB_ASSIGN    { makeList("-=", 'o', lineCount); }

| LEFT_ASSIGN   { makeList("<=", 'o', lineCount); }

| RIGHT_ASSIGN  { makeList(">=", 'o', lineCount); }

| AND_ASSIGN    { makeList("&=", 'o', lineCount); }

| XOR_ASSIGN    { makeList("^=", 'o', lineCount); }

| OR_ASSIGN     { makeList("|=", 'o', lineCount); };

```

expression

```

: assignment_expression

| expression ',' assignment_expression { makeList(",", 'p',

```

lineCount); };

constant_expression

```

: conditional_expression;

```

declaration

```

: declaration_specifiers ';' { makeList(";",

```

```
'p', lineCount);typeBuffer=' ' ; }
```

```
| declaration_specifiers init_declarator_list ';' {
```

```
makeList(";", 'p', lineCount); typeBuffer=' ' ;} ;
```

```
declaration_specifiers
```

```
: storage_class_specifier
```

```
| storage_class_specifier declaration_specifiers
```

```
| type_specifier
```

```
| type_specifier declaration_specifiers
```

```
| type_qualifier
```

```
| type_qualifier declaration_specifiers
```

```
;
```

```
init_declarator_list
```

```
: init_declarator
```

```
| init_declarator_list ',' init_declarator { makeList(";",
```

```
'p', lineCount); } ;
```


init_declarator

```

        : declarator
    };    | declarator '=' initializer { makeList("=", 'o', lineCount);

```

storage_class_specifier

```

        : TYPEDEF      { makeList("typedef", 'k', lineCount);}

    | EXTERN  { makeList("extern", 'k', lineCount);}

    | STATIC  { makeList("static", 'k', lineCount);}

    | AUTO      { makeList("auto", 'k', lineCount);}

    | REGISTER  { makeList("register", 'k', lineCount);}

    ;

```

type_specifier

```

        : VOID      { makeList("void", 'k', lineCount);typeBuffer='v';}

    | CHAR      { makeList("char", 'k', lineCount);typeBuffer='c';}

    | SHORT     { makeList("short", 'k', lineCount);}

```

```

| INT      { makeList("int", 'k', lineCount); typeBuffer='i';}

| LONG     { makeList("long", 'k', lineCount);}

| FLOAT { makeList("float", 'k', lineCount); typeBuffer='f';}

| DOUBLE  { makeList("double", 'k', lineCount);}

| SIGNED  { makeList("signed", 'k', lineCount);}

| UNSIGNED { makeList("unsigned", 'k', lineCount);}

| struct_or_union_specifier

| enum_specifier

| TYPE_NAME

;

```

struct_or_union_specifier

```

: struct_or_union IDENTIFIER '{' struct_declaration_list '}'

| struct_or_union '{' struct_declaration_list '}'

| struct_or_union IDENTIFIER

;

```

struct_or_union

```

: STRUCT  { makeList("struct", 'k', lineCount);}

| UNION   { makeList("union", 'k', lineCount);}

;

```

struct_declaration_list

```

: struct_declaration

| struct_declaration_list struct_declaration

;

```

struct_declaration

```

: specifier_qualifier_list struct_declarator_list ';'

{ makeList(";", 'p', lineCount); };

```

specifier_qualifier_list

```

: type_specifier specifier_qualifier_list

| type_specifier

| type_qualifier specifier_qualifier_list

```

```
| type_qualifier;
```

struct_declarator_list

```
: struct_declarator
```

```
| struct_declarator_list ',' struct_declarator
```

```
{ makeList(",", 'p', lineCount); };
```

struct_declarator

```
: declarator
```

```
| ':' constant_expression {makeList(":", 'p', lineCount);}
```

```
| declarator ':' constant_expression
```

```
{ makeList(":", 'p', lineCount); };
```

enum_specifier

```
: ENUM '{' enumerator_list '}' { makeList("enum", 'k',
```

```
lineCount);}
```

```
| ENUM IDENTIFIER '{' enumerator_list '}' {
```

```
makeList("enum", 'k', lineCount); makeList(tablePtr, 'v',
lineCount); }
```

```
    | ENUM IDENTIFIER    { makeList("enum", 'k', lineCount);
makeList(tablePtr, 'v', lineCount); };
```

```
enumerator_list
```

```
    : enumerator
```

```
    | enumerator_list ',' enumerator { makeList(",", 'p',
lineCount); };
```

```
enumerator
```

```
    : IDENTIFIER    { makeList(tablePtr, 'v', lineCount); }
```

```
    | IDENTIFIER '=' constant_expression    { makeList("=", 'o',
lineCount); makeList("tablePtr", 'v', lineCount); };
```

```
type_qualifier
```

```
    : CONST    { makeList("const", 'k', lineCount); }
```

```
    | VOLATILE    { makeList("volatile", 'k', lineCount); };
```

declarator

: pointer direct_declarator

| direct_declarator;

direct_declarator

: IDENTIFIER { makeList(tablePtr, 'v', lineCount); }

| '(' declarator ')' { makeList("(", 'p', lineCount);
makeList(")", 'p', lineCount); }

| direct_declarator '[' constant_expression ']' {

makeList("[", 'p', lineCount); makeList("]", 'p', lineCount); }

| direct_declarator '[' ']' { makeList("[", 'p',

lineCount); makeList("]", 'p', lineCount); }

| direct_declarator '(' parameter_type_list ')' {

makeList("(", 'p', lineCount); makeList(")", 'p', lineCount); }

| direct_declarator '(' identifier_list ')' {

makeList("(", 'p', lineCount); makeList(")", 'p', lineCount); }

```

    | direct_declarator '(' ')' { makeList("(", 'p',
lineCount); makeList(")", 'p', lineCount); } ;

```

pointer

```

    : '*' { makeList("*", 'o', lineCount); }

    | '*' type_qualifier_list { makeList("*", 'o', lineCount); }

    | '*' pointer { makeList("*", 'o', lineCount); }

    | '*' type_qualifier_list pointer
{ makeList("*", 'o', lineCount); };

```

type_qualifier_list

```

    : type_qualifier

    | type_qualifier_list type_qualifier;

```

parameter_type_list

```

    : parameter_list

    | parameter_list ',' ELLIPSIS { makeList(",", 'p',
lineCount); makeList("::", 'o', lineCount); };

```

parameter_list

```

    : parameter_declaration

    | parameter_list ',' parameter_declaration { makeList(",",
'p', lineCount); };

```

parameter_declaration

```

    : declaration_specifiers declarator

    | declaration_specifiers abstract_declarator

    | declaration_specifiers ;

```

identifier_list

```

    : IDENTIFIER {makeList(tablePtr, 'v', lineCount);}

    | identifier_list ',' IDENTIFIER { makeList(tablePtr, 'v',
lineCount); makeList(",", 'p', lineCount); };

```

type_name

```

    : specifier_qualifier_list

    | specifier_qualifier_list abstract_declarator;

```


abstract_declarator

: pointer

| direct_abstract_declarator

| pointer direct_abstract_declarator;

direct_abstract_declarator

: '(' abstract_declarator ')' { makeList("(", 'p', lineCount); makeList(")", 'p', lineCount); }

| '[' ']' { makeList("[", 'p', lineCount); makeList("]", 'p',

lineCount); }

| '[' constant_expression ']' { makeList("[", 'p',

lineCount); makeList("]", 'p', lineCount); }

| direct_abstract_declarator '[' ']' { makeList("[", 'p',

lineCount); makeList("]", 'p', lineCount); }

| direct_abstract_declarator '[' constant_expression ']' {

makeList("[", 'p', lineCount); makeList("]", 'p', lineCount); }

```

    | '(' ')' { makeList("(", 'p', lineCount); makeList(")", 'p',
lineCount); }

```

```

    | '(' parameter_type_list ')' { makeList("(", 'p',
lineCount); makeList(")", 'p', lineCount); }

```

```

    | direct_abstract_declarator '(' ')' { makeList("(", 'p',
lineCount); makeList(")", 'p', lineCount); }

```

```

    | direct_abstract_declarator '(' parameter_type_list ')' {
makeList("(", 'p', lineCount); makeList(")", 'p', lineCount); };

```

```

initializer

```

```

    : assignment_expression

```

```

    | '{' initializer_list '}'

```

```

    | '{' initializer_list ',' '}' ;

```

```

initializer_list

```

```

    : initializer

```

```

    | initializer_list ',' initializer { makeList(",", 'p',

```

```
lineCount); };
```

```
statement
```

```
    : labeled_statement
```

```
    | compound_statement
```

```
    | expression_statement
```

```
    | selection_statement
```

```
    | iteration_statement
```

```
    | jump_statement;
```

```
labeled_statement
```

```
    : IDENTIFIER ':' statement    { makeList(tablePtr, 'v',
```

```
lineCount); }
```

```
    | CASE constant_expression ':' statement    { makeList(":",
```

```
'p', lineCount); makeList("case", 'k', lineCount);}
```

```
    | DEFAULT ':' statement          { makeList(":", 'p',
```

```
lineCount); makeList("default", 'k', lineCount); };
```

compound_statement

: '{' '}'

| '{' statement_list '}'

| '{' declaration_list '}'

| '{' declaration_list statement_list '}'

;

declaration_list

: declaration

| declaration_list declaration

;

statement_list

: statement

| statement_list statement

;

expression_statement

```

: ';'          { makeList(";", 'p', lineCount); }

| expression ';' { makeList(";", 'p', lineCount); }

;

```

selection_statement

```

: IF '(' expression ')' statement %prec LOWER_THAN_ELSE

      { makeList("if", 'k', lineCount);

makeList("(", 'p', lineCount); makeList(")", 'p', lineCount);}

| IF '(' expression ')' statement ELSE statement

{ makeList("if", 'k', lineCount); makeList("else", 'k',

lineCount); makeList("(", 'p', lineCount);

makeList(")", 'p', lineCount); }

| SWITCH '(' expression ')' statement { makeList("switch",

'k', lineCount); makeList("(", 'p', lineCount); makeList(")",

'p', lineCount); };

```

iteration_statement

```

        : WHILE '(' expression ')' statement { makeList("while",
'k', lineCount); makeList("(", 'p', lineCount); makeList(")", 'p',
lineCount); }

```

```

        | DO statement WHILE '(' expression ')' ';' { makeList("do",
'k', lineCount); makeList("while", 'k', lineCount); makeList("(",
'p', lineCount);
makeList(")", 'p',
lineCount); makeList(";", 'p', lineCount); }

```

```

        | FOR '(' expression_statement expression_statement ')'
statement { makeList("for", 'k', lineCount); makeList("(", 'p',
lineCount); makeList(")", 'p', lineCount); }

```

```

        | FOR '(' expression_statement expression_statement
expression ')' statement { makeList("for", 'k', lineCount);
makeList("(", 'p', lineCount); makeList(")", 'p', lineCount); }

;

```

```

jump_statement: GOTO IDENTIFIER ';' { makeList("goto", 'k',

```

```
lineCount); makeList(";", 'p', lineCount); makeList(tablePtr, 'v',
lineCount);}
```

```
    | CONTINUE ';'          { makeList("continue", 'k',
lineCount); makeList(";", 'p', lineCount); }
```

```
    | BREAK ';'            { makeList("break", 'k', lineCount);
makeList(";", 'p', lineCount);}
```

```
    | RETURN ';'           { makeList("return", 'k', lineCount);
makeList(";", 'p', lineCount);}
```

```
    | RETURN expression ';' { makeList("return", 'k',
lineCount); makeList(";", 'p', lineCount);};
```

```
translation_unit
```

```
    : external_declaration
```

```
    | translation_unit external_declaration
```

```
    ;
```

```
external_declaration
```

: function_definition

| declaration

;

function_definition

: declaration_specifiers declarator declaration_list

compound_statement

| declaration_specifiers declarator compound_statement

| declarator declaration_list compound_statement

| declarator compound_statement

;

%%

void yyerror()

{

errorFlag=1;

fflush(stdout);


```

        printf("\n%s : %d :Syntax error \n",sourceCode,lineCount);

    }

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

    if(argc<=1){

printf("Invalid ,Expected Format : ./a.out <\\"sourceCode\ "> \n");

        return 0;

    }

    yyin=fopen(argv[1],"r");

    sourceCode=(char *)malloc(strlen(argv[1])*sizeof(char));

    sourceCode=argv[1];

    yyparse();

    if(nestedCommentCount!=0){

        errorFlag=1;

        printf("%s : %d : Comment Does Not

End\n",sourceCode,lineCount);

```

}

```
if(commentFlag==1){
```

```
errorFlag=1;
```

```
printf("%s : %d : Nested", name, depth);
```

```
Comment\n", sourceCode, lineCount);
```

}

```
if(!errorFlag){
```

```
printf("\n\n\t\t%s Parsing Completed\n\n",sourceCode);
```

```
FILE *writeFile=fopen("Output.txt","w");
```

```
fprintf(writeFile, "-----")
```

-----\n");

[illegible]

```
fprintf(writeFile, "-----\n");
```

```
fprintf(writeFile, "\t\tIdentifier
```

```

\t\t\t\tToken\t\t\t\t\t Line Number\n");

fprintf(writeFile, "-----\n");

for(tokenList *ptr=parsedPtr;ptr!=NULL;ptr=ptr->next){

    fprintf(writeFile, "\n%20s%40s%60s", ptr->token, ptr->type,

        ptr->line);

    }

fprintf(writeFile, "\n \n \n");

fprintf(writeFile, "-----\n");

fprintf(writeFile, "\n\t\t\t\t\t\t\t\t\t\t\tSYMBOL TABLE\n");

fprintf(writeFile, "-----\n");

fprintf(writeFile, "\tIdentifier \t\t\t\t\tType\t\t\t\t\t Line

Number\n");

fprintf(writeFile, "-----

```

[illegible]

```
        fclose(writeFile);

    }

    printf("\n\n");

}

void makeList(char *tokenName, char tokenType, int tokenLine)

{

    char line[39], lineBuffer[19];

    snprintf(lineBuffer, 19, "%d", tokenLine);

    strcpy(line, " ");

    strcat(line, lineBuffer);

    char type[20];

    switch(tokenType)

    {

        case 'c':

            strcpy(type, "Constant");
```

```
                break;

case 'v':

    strcpy(type, "Identifier");

    break;

case 'p':

    strcpy(type, "Punctuator");

    break;

case 'o':

    strcpy(type, "Operator");

    break;

case 'k':

    strcpy(type, "Keyword");

    break;

case 's':

    strcpy(type, "String Literal");
```

```

        break;

    case 'd':

        strcpy(type, "Preprocessor Statement");

        break;

}

for(tokenList *p=parsedPtr;p!=NULL;p=p->next)

    if(strcmp(p->token, tokenName)==0){

        strcat(p->line, line);

        goto xx;

    }

    tokenList *temp=(tokenList *)malloc(sizeof(tokenList));

    temp->token=(char *)malloc(strlen(tokenName)+1);

    strcpy(temp->token, tokenName);

    strcpy(temp->type, type);

    strcpy(temp->line, line);

```

```
temp->next=NULL;

tokenList *p=parsedPtr;

if(p==NULL){

    parsedPtr=temp;

}

else{

    while(p->next!=NULL){

        p=p->next;

    }

    p->next=temp;

}

xx:

if(tokenType == 'c')

{

    for(tokenList *p=constantPtr;p!=NULL;p=p->next)
```



```
        if(strcmp(p->token, tokenName)==0){

            strcat(p->line, line);

            return;

        }

tokenList *temp=(tokenList *)malloc(sizeof(tokenList));

temp->token=(char *)malloc(strlen(tokenName)+1);

strcpy(temp->token, tokenName);

strcpy(temp->type, type);

strcpy(temp->line, line);

temp->next=NULL;

tokenList *p=constantPtr;

if(p==NULL){

    constantPtr=temp;

}

else{
```

```
        while(p->next!=NULL){

            p=p->next;

        }

        p->next=temp;

    }

}

if(tokenType=='v')

{

    for(tokenList *p=symbolPtr;p!=NULL;p=p->next)

        if(strcmp(p->token,tokenName)==0){

            strcat(p->line,line);

            return;

        }

    tokenList *temp=(tokenList *)malloc(sizeof(tokenList));

    temp->token=(char *)malloc(strlen(tokenName)+1);
```

```
strcpy(temp->token, tokenName);

switch(typeBuffer){

case 'i': strcpy(temp->type, "INT"); break;

case 'f': strcpy(temp->type, "FLOAT");break;

case 'v' :strcpy(temp->type, "VOID");break;

case 'c': strcpy(temp->type, "CHAR");break;

}

strcpy(temp->line, line);

temp->next=NULL;

tokenList *p=symbolPtr;

if(p==NULL){

symbolPtr=temp;

}

else{

while(p->next!=NULL){
```

```
                p=p->next;

            }

            p->next=temp;

        }

    }

}
```

Execution of the code:

The following is a shell script to automate the compilation and execution of the parser.

```
#!/bin/sh
lex lexicalAnalyzer.l
yacc -d syntaxChecker.y
gcc lex.yy.c y.tab.c -w -g
./a.out file_name.c
rm y.tab.c y.tab.h lex.yy.c
```

Test Cases:

Without Errors:

Serial No	Test Case	Expected Output	Status
1.	<pre>#include<stdio.h> void main() { int a=5,d=10,c; printf("\nEnter Number:"); c = a+d-10; return; }</pre>	<pre>arith1.c Parsing Completed ----SYMBOL TABLE---- Identifier Type Line Number ---- main VOID 2 a INT 4 6 d INT 4 6 c INT 4 6 printf 5 ---- END OF SYMBOL TABLE ----</pre>	PASS
2.	<pre>//SingleLine comments #include<stdio.h> void main() { int i; for(i=0;i<10;i+=2){ printf("Value of i=", i); /* loops the value of i by 2 at a time */ } return; /*multiline comment */ }</pre>	<pre>comments2.c Parsing Completed ----SYMBOL TABLE---- Identifier Type Line Number ---- main VOID 2 i INT 6 7 7 8 printf 8 ---- END OF SYMBOL TABLE ----</pre>	PASS
3.	<pre>#include<stdio.h> void main() { int a=10; if(a>5) printf("a is greater than 5"); else printf("a is less than equal to 5"); return; }</pre>	<pre>ifelse.c Parsing Completed ----SYMBOL TABLE---- Identifier Type Line Number ---- main VOID 2 a INT 4 5 printf 6 8 ---- END OF SYMBOL TABLE ----</pre>	PASS

4.	<pre>#include<stdio.h> void main() { int i; for(i=0;i<10;i+=2){ printf("Value of i=", i); } return; }</pre>	loops.c Parsing Completed ----SYMBOL TABLE---- Identifier Type Line Number ---- ---- ----- main VOID 2 i INT 4 5 5 5 6 printf 6 8 ---- END OF SYMBOL TABLE ----	PASS
5.	<pre>#include<stdio.h> void main() { char s[30]; s= "nitk surathkal"; printf("\nString is:",s); return; }</pre>	string.c Parsing Completed ----SYMBOL TABLE---- Identifier Type Line Number ---- ---- ----- main VOID 2 s char 4 5 6 printf 6 ---- END OF SYMBOL TABLE ----	PASS

With Errors:

Serial No	Test Case	Expected Output	Status
1.	<pre>#include<stdio.h> void main() { int a=5,d=10,c; printf("\nEnter Number:"); a+d-10 = c; return; }</pre>	arith_error.c : 6 :Syntax error	PASS

2.	<pre>//SingleLine comments #include<stdio.h> void main() { int i; for(i=0;i<10;i+=2){ printf("Value of i=", i); /* loops the value of i by 2 at a time */ } return; }</pre>	<pre>comments_error.c : 13 :Syntax error comments_error.c : 13 : Comment Does Not End comments_error.c : 13 : Nested Comment</pre>	PASS
3.	<pre>#include<stdio.h> void main() { int a=10; if(a>5) printf("a is greater than 5"); else printf("a is less than equal to 5"); els return; }</pre>	<pre>ifelse_error.c : 10 :Syntax error</pre>	PASS
4.	<pre>#include<stdio.h> void main() { int i; for(i=0;i<10){ printf("Value of i=", i); } return; }</pre>	<pre>loops_error.c : 5 :Syntax error</pre>	PASS
5.	<pre>#include<stdio.h> void main() { char s[30]; s= "nitk surathkal; printf("\nString is:",s); return; }</pre>	<pre>string_error.c : 5 :Syntax error</pre>	PASS

Implementation

The lexer code submitted in the previous phase took care of most of the features of C using regular expressions. Some special corner cases were taken care of using custom reg-ex. These were:

- **The regex for Identifiers**
- **Multiline comments should be supported**
- **Literals**
- **Error Handling for Incomplete String**
- **Error Handling for Nested Comments**

The parser code requires exhaustive token recognition and because of this reason, we utilised the lexer code given under the C specifications with the parser. The parser implements C grammar using a number of production rules.

Parser takes tokens from the lexer output, one at a time and applies the corresponding production rules to generate the symbol table with the variable and function types. If the parsing is not successful, the parser outputs the line number with the corresponding error. We have written a function, symlook(), which takes a string (name of the identifier) as input and –

1. If the identifier is already present in the symbol table, returns the struct symtab entry corresponding to the identifier.
2. If not already present, creates a new struct symtab entry, adds it to the symbol table, and returns the struct symtab.

5 Test Cases: Sample Programs with most features of C covered

Input 1:

```
#include<stdio.h>
void main()
{
    int a=5,d=10,c;
    printf("\nEnter Number:");
    c = a+d-10;
    return;
}
```


Output 1: Parser Output

```
veena@veena:~/Documents/6th/CD LAB/compiler-design/Parser/Syntax Analyser$ bash compile.sh
```

```
arith1.c Parsing Completed
```

PARSED TABLE		
Identifier	Token	Line Number
void	Keyword	2
main	Identifier	2
(Punctuator	2 5
)	Punctuator	2 5
{	Punctuator	3
int	Keyword	4
a	Identifier	4 6
5	Constant	4
=	Operator	4 4 6
d	Identifier	4 6
10	Constant	4 6
,	Punctuator	4 4
c	Identifier	4 6
;	Punctuator	4 5 6 7
printf	Identifier	5
"\nEnter Number:"	String Literal	5
+	Operator	6
-	Operator	6
return	Keyword	7
}	Punctuator	8

SYMBOL TABLE		
Identifier	Type	Line Number
main	VOID	2
a	INT	4 6
d	INT	4 6
c	INT	4 6
printf		5

CONSTANT TABLE	
Constant Value	Line Number
5	4
10	4 6

Input 2:

```

1 | #include<stdio.h>
2 | int main(){
3 |     int a; //this is a comment
4 |     return 0;
5 |     /*abc*/
6 | }

```

Output 2: Parser Output

```

veena@veena:~/Documents/6th/CD LAB/compiler-design/Parser/Syntax Analyser$ bash compile.sh

comments2.c Parsing Completed

```

PARSED TABLE		
Identifier	Token	Line Number
int	Keyword	2 3
main	Identifier	2
(Punctuator	2
)	Punctuator	2
{	Punctuator	2
a	Identifier	3
;	Punctuator	3 4
0	Constant	4
return	Keyword	4
}	Punctuator	6

SYMBOL TABLE		
Identifier	Type	Line Number
main	INT	2
a	INT	3

CONSTANT TABLE	
Constant Value	Line Number
0	4

Input 3:

```
#include<stdio.h>
void main()
{
    int a=10;
    if(a>5)
        printf("a is greater than 5");
    else
        printf("a is less than equal to 5");
    return;
}
```

Output 3: Parser Output

```
veena@veena:~/Documents/6th/CD LAB/compiler-design/Parser/Syntax Analyser$ bash compile.sh
Help
```

```
ifelse3.c Parsing Completed
```

PARSED TABLE		
Identifier	Token	Line Number
void	Keyword	2
main	Identifier	2
(Punctuator	2 6 8 8
)	Punctuator	2 6 8 8
{	Punctuator	3
int	Keyword	4
a	Identifier	4 5
10	Constant	4
=	Operator	4
;	Punctuator	4 6 8 9
5	Constant	5
printf	Identifier	6 8
"a is greater than 5"	String Literal	6
"a is less than equal to 5"	String Literal	8
if	Keyword	8
else	Keyword	8
return	Keyword	9
}	Punctuator	10

SYMBOL TABLE		
Identifier	Type	Line Number
main	VOID	2
a	INT	4 5
printf		6 8

CONSTANT TABLE	
Constant Value	Line Number
10	4
5	5

Input 4:

```
#include<stdio.h>
void main()
{
    int i;
    for(i=0;i<10;i+=2){
        printf("Value of i=", i);
    }
    return;
}
```

Output 4: Parser Output

```
veena@veena:~/Documents/6th/CD LAB/compiler-design/Parser/Syntax Analyser$ bash compile.sh

loops4.c Parsing Completed
```

PARSED TABLE		
Identifier	Token	Line Number
void	Keyword	2
main	Identifier	2
(Punctuator	2 6 7
)	Punctuator	2 6 7
{	Punctuator	3 5
int	Keyword	4
i	Identifier	4 5 5 5 6
;	Punctuator	4 5 5 6 8
=	Operator	5
0	Constant	5
10	Constant	5
+=	Operator	5
2	Constant	5
printf	Identifier	6
"Value of i="	String Literal	6
,	Punctuator	6
}	Punctuator	7 9
for	Keyword	7
return	Keyword	8

SYMBOL TABLE		
Identifier	Type	Line Number
main	VOID	2
i	INT	4 5 5 5 6
printf		6

CONSTANT TABLE	
Constant Value	Line Number
0	5
10	5
2	5

Input 5:

```
#include<stdio.h>
void main()
{
    char s[30];
    s= "nitk surathkal";
    printf("\nString is:",s);
    return;
}
```

Output 5: Parser Output

```
veena@veena:~/Documents/6th/CD LAB/compiler-design/Parser/Syntax Analyser$ bash compile.sh
```

```
string.c Parsing Completed
```

PARSED TABLE		
Identifier	Token	Line Number
void	Keyword	2
main	Identifier	2
(Punctuator	2 6
)	Punctuator	2 6
{	Punctuator	3
char	Keyword	4
s	Identifier	4 5 6
30	Constant	4
[Punctuator	4
]	Punctuator	4
;	Punctuator	4 5 6 7
=	Operator	5
"nitk surathkal"	String Literal	5
printf	Identifier	6
"\nString is:"	String Literal	6
,	Punctuator	6
return	Keyword	7
}	Punctuator	8

SYMBOL TABLE		
Identifier	Type	Line Number
main	VOID	2
s	CHAR	4 5 6
printf		6

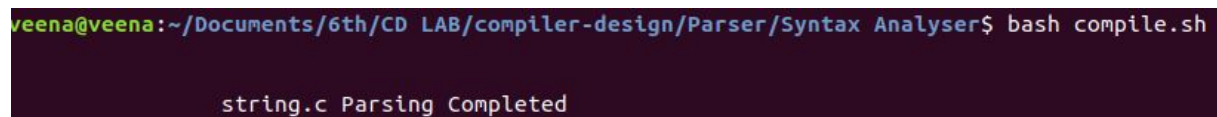
CONSTANT TABLE	
Constant Value	Line Number
30	4

Test Case: Sample C program with convoluted constructions

Input file :

```
1 #include<stdio.h>
2 struct st{
3     int a;
4 };
5 |
6 void main()
7 {
8     char s[30];
9     s= "nitk surathkal";
10    printf("\nString is:",s);
11    return;
12 }
```

Output :

A terminal window with a dark purple background. The prompt is 'veena@veena:~/Documents/6th/CD LAB/compiler-design/Parser/Syntax Analyser\$'. The command 'bash compile.sh' has been executed, resulting in the output 'string.c Parsing Completed' on the next line.

```
veena@veena:~/Documents/6th/CD LAB/compiler-design/Parser/Syntax Analyser$ bash compile.sh
string.c Parsing Completed
```

PARSED TABLE		
Identifier	Token	Line Number
struct	Keyword	2
{	Punctuator	2 7
int	Keyword	3
a	Identifier	3
;	Punctuator	3 4 8 9 10 11
}	Punctuator	4 12
void	Keyword	6
main	Identifier	6
(Punctuator	6 10
)	Punctuator	6 10
char	Keyword	8
s	Identifier	8 9 10
30	Constant	8
[Punctuator	8
]	Punctuator	8
=	Operator	9
"nitk surathkal"	String Literal	9
printf	Identifier	10
"\nString is:"	String Literal	10
,	Punctuator	10
return	Keyword	11

SYMBOL TABLE		
Identifier	Type	Line Number
a	INT	3
main	VOID	6
s	CHAR	8 9 10
printf		10

CONSTANT TABLE	
Constant Value	Line Number
30	8

5 Test Cases: Invalid C Programs

Input 6:

```
#include<stdio.h>
void main()
{
    int a=5,d=10,c;
    printf("\nEnter Number:");
    a+d-10 = c;
    return;
}
```


Output 6:

```
shweta@shweta-vostro-15-3568:~/Compiler-Design/Syntax Analyzer/src$ bash compile.sh
arith_error.c : 6 :Syntax error
```

Input 7:

```
#include<stdio.h>
int main(){
    int a; //this is a comment
    return 0;
    /*abc**/
}
```

Output 7:

```
veena@veena:~/Documents/6th/CD LAB/compiler-design/Parser/Syntax Analyser$ bash compile.sh
comments_error.c : 7 : Comment Does Not End
comments_error.c :                               7 : Nested Comment
```

Input 8:

```
#include<stdio.h>
void main()
{
    int a=10;
    if(a>5)
        printf("a is greater than 5");
    else
        printf("a is less than equal to 5");
    else
        return;
}
```

Output 8:

```
shweta@shweta-vostro-15-3568:~/Compiler-Design/Syntax Analyzer/src$ bash compile.sh
ifelse_error.c : 10 :Syntax error
```

Input 9:

```
#include<stdio.h>
void main()
{
    int i;
    for(i=0;i<10 ){
        printf("Value of i=", i);
    }
    return;
}
```

Output 9:

```
shweta@shweta-vostro-15-3568:~/Compiler-Design/Syntax Analyzer/src$ bash compile.sh
loops_error.c : 5 :Syntax error
```

Input 10:

```
#include<stdio.h>
void main()
{
    char s[30];
    s= "nitk surathkal;
    printf("\nString is:",s);
    return;
}
```

Output 10:

```
shweta@shweta-vostro-15-3568:~/Compiler-Design/Syntax Analyzer/src$ bash compile.sh
string_error.c : 5 :Syntax error
```

Results and Conclusion

We were able to successfully parse the tokens recognized by the flex script for C. The output displays the set of identifiers and constants present in the program with their types. The functions present in the C program are also identified as functions and not as identifiers. The parser generates error messages in case of any syntactical errors in the test program.

Future work:

The yacc script presented in this report takes care of all the parsing rules of C language, but is not fully exhaustive in nature. Our future work would include making the script even more robust in order to handle function parameters and other corner cases, and thus make it more effective.

References :

- <http://dinosaur.compilertools.net/lex/>
- Compilers-Principles, Techniques And Tools By Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman
- <https://wikipedia.org>
- <https://silcnitc.github.io/yacc.html>
- <https://www.geeksforgeeks.org/introduction-to-yacc/>