COMPILER DESIGN PROJECT REPORT - 2

PARSER FOR C PROGRAMMING LANGUAGE

Submitted By :

Arvind Ramachandran - 15CO111

Aswanth P P - 15CO112

DATE : 12/02/2018

TABLE OF CONTENTS

INTRODUCTION

When an input string (source code or a program in some language) is given to a compiler, the compiler processes it in several phases, starting from lexical analysis (scans the input and divides it into tokens) to target code generation. The aim of this phase of a compiler is to implement a parser for C language. The lexical analyzer generated in the first phase reads the source program and generates tokens that are given as input to the parser which then creates a syntax tree in accordance with the grammar, consequently leading to the generation of intermediate code that is fed into the synthesis phase, to obtain the correct, equivalent machine level code. We have seen that a lexical analyzer can identify tokens with the help of regular expressions and pattern rules. But a lexical analyzer cannot check the syntax of a given sentence due to the limitations of the regular expressions. Regular expressions cannot check balancing tokens, such as parenthesis. Therefore, this phase uses context-free grammar (CFG), which is recognized by push-down automata.

SYNTAX ANALYSIS

Syntax Analysis or **Parsing** is the second phase, i.e. after lexical analysis. A syntax analyzer or parser takes the input from a lexical analyzer in the form of token streams. The parser analyzes the source code (token stream) against the production rules to detect any errors in the code. It checks the syntactic structure of the given input, i.e. whether the given input is in the correct syntax (of the language in which the input has been written) or not. It does so by building a data structure, called a **Parse tree** or **Syntax tree**. The parse tree is constructed by using the predefined Grammar of the language and the input string. 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. The Grammar for a Language consists of Production rules.

CONTEXT-FREE GRAMMAR

A context-free grammar has four components:

- A set of **non-terminals** (V). Non-terminals are syntactic variables that denote sets of strings. The non-terminals define sets of strings that help define the language generated by the grammar.
- A set of tokens, known as **terminal symbols** (Σ). Terminals are the basic symbols from which strings are formed.

- A set of **productions** (P). The productions of a grammar specify the manner in which the terminals and non-terminals can be combined to form strings. Each production consists of a **non-terminal** called the left side of the production, an arrow, and a sequence of tokens and/or **on- terminals**, called the right side of the production.
- One of the non-terminals is designated as the start symbol (S); from where the production begins.

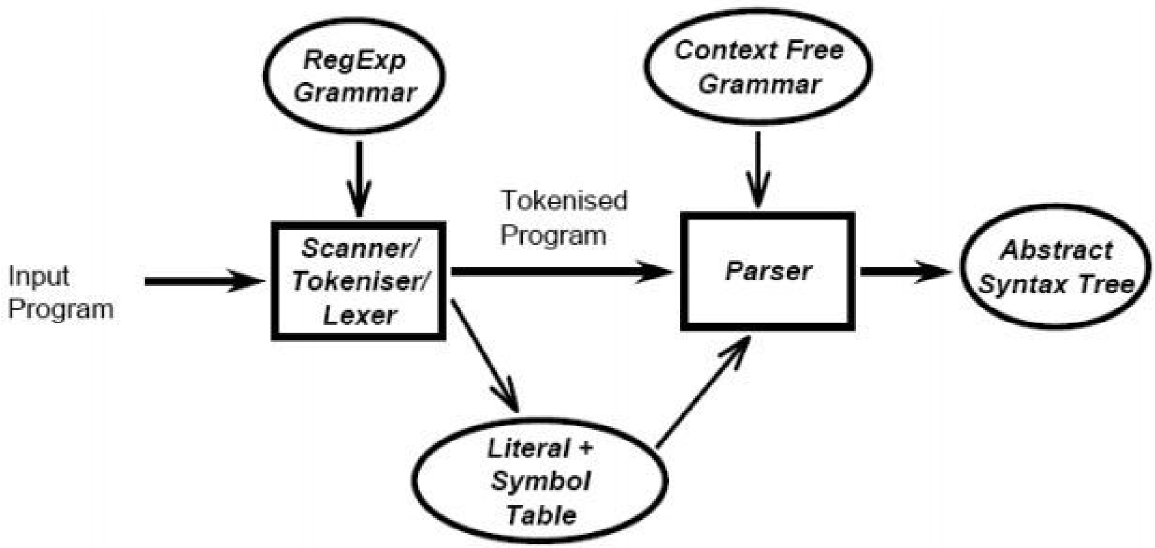


Fig 1: Flowchart of a parser

The strings are derived from the start symbol by repeatedly replacing a non-terminal (initially the start symbol) by the right side of a production, for that non-terminal.

PARSE TREE

A parse tree is a graphical depiction of a derivation. It is convenient to see how strings are derived from the start symbol. The start symbol of the derivation becomes the root of the parse tree.

Consider the following example with the given Production rules and Input String.

Input string : id + id * id

Production rules:

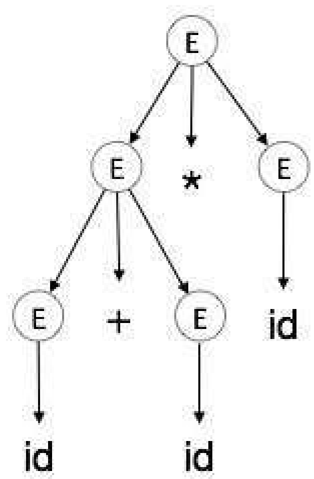
- E → E + E
- E → E * E

- E → id

The left-most derivation is:

- E → E * E
- E → E + E * E
- E → id + E * E
- E → id + id * E
- E → id + id * id

The parse tree for the given input string is :



In a parse tree:

- All leaf nodes are terminals.
- All interior nodes are non-terminals.
- In-order traversal gives original input string.

A parse tree depicts associativity and precedence of operators. The deepest sub-tree is traversed first, therefore the operator in that sub-tree gets precedence over the operator which is in the parent nodes.

YACC SCRIPT

YACC stands for Yet Another Compiler-Compiler. It is a parser generator for LALR(1) grammars. Given a description of the grammar,it generates a C source for the parser. The input is a file that contains the grammar description with a formalism similar to the BNF (Backus-Naur Form) notation for language specification :

- Non-terminal symbols - lowercase identifiers

  - Expr, stmt
- Terminal symbols - uppercase identifiers or single characters
  - INTEGER, FLOAT, IF, WHILE, ‘;’, ‘.’
- Grammar rules (Production rules)
  - expr: expr ‘+’ expr | expr ‘*’ expr ; E→E+E|E*E

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. Yacc then generates a function to control the input process. This function, called a parser, calls the user-supplied low-level input routine (the lexical analyzer) to pick up the basic items (called tokens) from the input stream. These tokens are organized according to the input structure rules, called grammar rules; 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.

BASIC SPECIFICATIONS

Names refer to either tokens or nonterminal symbols. Yacc requires token names to be declared as such. Every specification file consists of three sections: the declarations, (grammar) rules, and programs. The sections are separated by double percent ``%%" marks. (The percent ``%" is generally used in Yacc specifications as an escape character.)

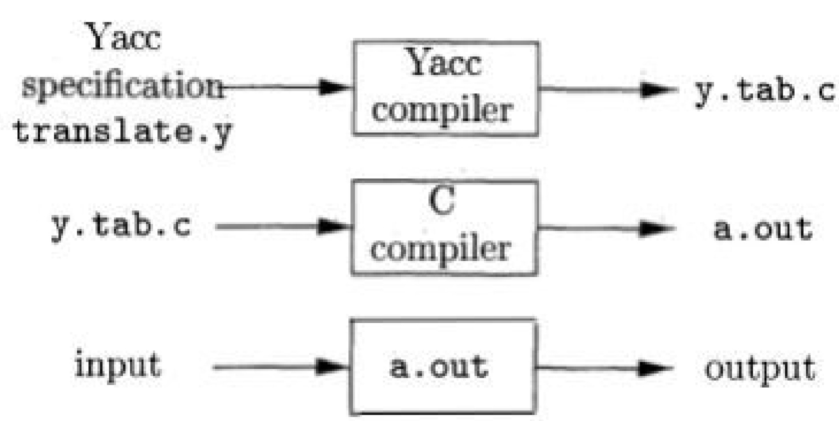


Fig 2: Creating a syntactical analyzer with Yacc

In other words, a full specification file looks like :

declarations
%%
rules
%%
programs

The declaration section may be empty. Moreover, if the programs section is omitted, the second %% mark may be omitted also; thus, the smallest legal Yacc specification is

%%
rules

The rules section is made up of one or more grammar rules. A grammar rule has the form:

A : BODY ;

A represents a nonterminal name, and BODY represents a sequence of zero or more names and literals.

ACTIONS

With each grammar rule, the user may associate actions to be performed each time the rule is recognized in the input process. These actions may return values, and may obtain the values returned by previous actions. Moreover, the lexical analyzer can return values for tokens, if desired. An action is an arbitrary C statement, and as such can do input and output, call subprograms, and alter external vectors and variables. An action is specified by one or more statements, enclosed in curly braces "{" and "}".

How the Parser Works

Yacc turns the specification file into a C program, which parses the input according to the specification given. The parser produced by Yacc consists of a finite state machine with a stack. The parser is also capable of reading and remembering the next input token (called the lookahead token). The current state is always the one on the top of the stack. A move of the parser is done as :

1. Based on current state, the parser decides whether it needs a lookahead token to decide what action should be done; if it needs and does not have one, it calls yylex to obtain the next token.

2. Using the current state, and the lookahead token if needed, the parser decides on its next action, and carries it out. This may result in states being pushed onto the stack, or popped off of the stack, and in the lookahead token being processed or left alone.

SOURCE CODE

Lexical Analyzer

This is the lex program that contains regular expressions and returns whatever tokens are required.

```
%{
#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}                     { lineCount++; }

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

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

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

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

FIG. 1

```
"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); }
"goto"                  {  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(TYPEDEF); }
"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); }
```

FIG. 2

```
{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(';'); }
("{"|"<%")              {  return('{'); }
("}"|"%>")              {  return('}'); }
","                     {  return(','); }
":"                     {  return(':'); }
"="                     {  return('='); }
"("                     {  return('('); }
")"                     {  return(')'); }
("["|"<:")              {  return('['); }
("]"|":>")              {  return(']'); }
"."                     {  return('.'); }
"&"                     {  return('&'); }
"!"                     {  return('!'); }
"~"                     {  return('~'); }
"-"                     {  return('-'); }
"+"                     {  return('+'); }
"*"                     {  return('*'); }
"/"                     {  return('/'); }
"%"                     {  return('%'); }
"<"                     {  return('<'); }
```

FIG. 3

```
">"                     {  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);
}
```

FIG. 4

Syntax Analyzer

This is the main parser code, a yacc file which contains the declarations, rules and programs and defines the actions which should be taken for various cases.

```
%{
#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
```

FIG. 1

```
%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); }
```

FIG. 2

```
        | '*' { 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); }
        ;

```

FIG. 3

```
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); }
```

FIG. 4

```
        | 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
```

FIG. 5

```
        : 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); }
        ;
```

FIG. 6

```

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); }
```

FIG. 7

```
        | '*' 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); }
```

FIG. 8

```
        | '[' ']'                                                  { 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); }
        ;


//  statements
statement
        : labeled_statement
        | compound_statement
        | expression_statement
        | selection_statement
        | iteration_statement
        | jump_statement
        ;
//1
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); }
        ;
//2
compound_statement
        : '{' '}'
        | '{' statement_list '}'
        | '{' declaration_list '}'
        | '{' declaration_list statement_list '}'
        ;
```

FIG. 9

```
declaration_list
        : declaration
        | declaration_list declaration
        ;

statement_list
        : statement
        | statement_list statement
        ;
//3
expression_statement
        : ';'                           { makeList(";", 'p', lineCount); }
        | expression ';'        { makeList(";", 'p', lineCount); }
        ;
//4
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); }
        ;
//5
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); }
        ;
//6
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);}
```

FIG. 10

```
        ;


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){
```

FIG. 11

```
                errorFlag=1;
                printf("%s :                                    %d : Nested Comment\n",sourceCode,lineCount);
        }

        if(!errorFlag){

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


                FILE *writeParsed=fopen("parsedTable.txt","w");
                fprintf(writeParsed,"\n\t\t\t\tParsed   Table\n\n\t\tToken\t\t\tType\t\t\t\t\t\tLineNumber\n");
                for(tokenList *ptr=parsedPtr;ptr!=NULL;ptr=ptr->next){
                        fprintf(writeParsed,"\n%20s%30.30s%60s",ptr->token,ptr->type,ptr->line);
                }


                FILE *writeSymbol=fopen("symbolTable.txt","w");
                fprintf(writeSymbol,"\n\t\t\t\tSymbolTable\n\n\t\tToken\t\t\t\t\tType\t\t\t\t\t\tLine Number\n");
                for(tokenList *ptr=symbolPtr;ptr!=NULL;ptr=ptr->next){
                        fprintf(writeSymbol,"\n%20s%30.30s%60s",ptr->token,ptr->type,ptr->line);
                }

                FILE *writeConstant=fopen("constantTable.txt","w");
                fprintf(writeConstant,"\n   \t\t\t\t\t\t\tConstant Table \n\n \t\t\t\t\tValue\t\t\t\t\t\tLine Number\n");
                for(tokenList *ptr=constantPtr;ptr!=NULL;ptr=ptr->next)
                fprintf(writeConstant,"\n%50s%60s",ptr->token,ptr->line);


                fclose(writeSymbol);
                fclose(writeConstant);
        }
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':
```

FIG. 12

```
                        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;
        }
```

FIG. 13

```
        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;
```

FIG. 14

```
                }
                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;
                }
        }
}
```

FIG. 15

EXECUTION OF THE CODE

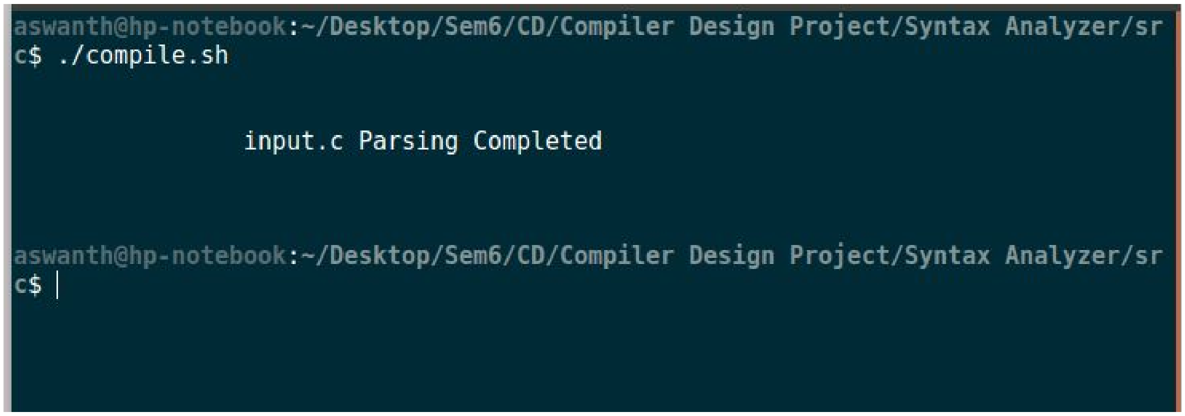
The following is a shell script to automate the compilation and execution of the parser.

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

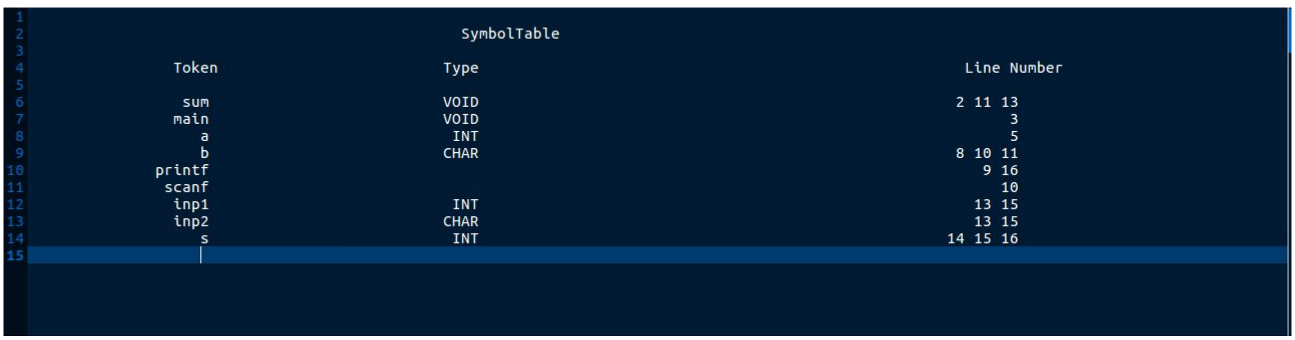
Sample Input

```
#include<stdio.h>
void sum(int ,char);
void main()
{
        int a=5;
        int ;
        short ;
        char b;
        printf("\nEnter Number:");
        scanf("%d",&b);
        sum(5,b);
}
void sum(int inp1,char inp2){
        int s=0;
        s=inp1+inp2;
        printf("\nSum : %d",s);
        return ;
}
```

Sample Output



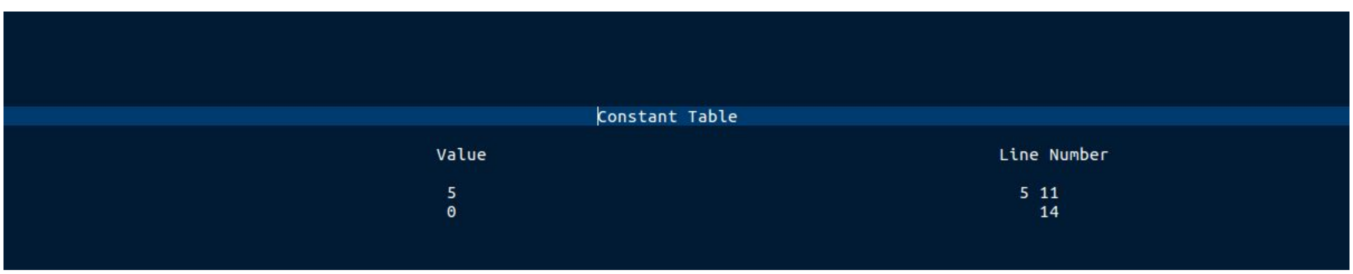
Symbol Table



SymbolTable

Token	Type	Line Number
sum	VOID	2 11 13
main	VOID	3
a	INT	5
b	CHAR	8 10 11
printf		9 16
scanf		10
inp1	INT	13 15
inp2	CHAR	13 15
s	INT	14 15 16

Constant Table



Constant Table

Value	Line Number
5	5 11
0	14

IMPLEMENTATION

The implementation is as follows:

- Firstly, a file is created with a " .y " extension, which represents a yacc file. The file created here is " **syntaxChecker.y** ".
- The file, like all yacc files, contains three main sections: Declarations, Rules, and Programs.
- The declaration section contains the structure for each entries of symbol table that is identifier name,data type and attribute values (in our case, line number)
- Also the declaration section contains all the pointer used in lexical phase to track each token that is each pointer from "**lexicalAnalyzer.l**" is imported here as extern type and used accordingly to raise lexical errors as well as syntactical errors using the production rules for each token
- Also ,declaration contains function declaration for symbol table and constant table management (that is **void makList(char *, char, int)**
- The Rules section contains the various grammatical rules that are to be followed while parsing.
- The Programs section is where the yyparse() and the yyerror() functions are called,in order to generate the error message at the line where the syntax error has occurred.
- The Program section makes symbol table ("**symblTable.txt**") and constant table ("**constantTable.txt**") for each node of the link list **struct tokenList**
- If a production rule contains identifier or constants **makeList( )** function will be invoked with respective type as 'v' or 'c' along with line number accordingly linked list for them is maintained
- **void makeList(char *tokenName , char tokenType , int lineNumber)** will add the identifier and constants accordingly by using linked list if an identifier is again came which is already there then it's line number is appended to the first entry hence no duplicate entry for each identifier in symbol table
- Once the file is made, it is run using the shell script by giving the following command: ./compile.sh

TESTCASES

Case1.c

```c
//Testcase to check arithmatic logical and relational statements

#include <stdio.h>
int main(){
	int a=10,c=40,b=56,d;
	a=b+c;
	a+=c;
	a=b+c*d;
	d=b*(c+d);
	int d[100];
	d[20]=a*d[10]+c;

	b=a**c;
	c=a++c;


```

Case2.c

```c
// Testcase to check control statements

#include<stdio.h>
int main(){
	int a=10,b=50,c=10;
	if(a>b){
		printf("\na is greater than b");
	}
	else{
		printf("\n a is smaller than b");
	}
	else{
		printf("\nUnbalanced Else");
	}

	if(a>b && b<c){
		if(a!=c){
			printf("\nHello world");
		}
	    else{
		printf("\nComputer Science");
	    }
	 }

```

Case3.c

```c
// Testcase to check  looping statements
#include<stdio.h>
int main(){

        int l=10,a=0,i;
        for(i=0;i<l;i++){
                printf("\nHello World");
        }
        for(a<l;a++){
                printf("\nInvalid Syntax");
        }
        while(i<10){
                printf("\nCompiler Design");
        }

}
```

Case4.c

```c
//Testcase to check function decleration and parameter passing

#include<stdio.h>
int main(){

        int a[]={1,2,3,4};
        function1(a);
}
void function1(){
        printf("\nHello World");
}
```

Case5.c

```c
//Testcase to check printf and scanf errors
#include<stdio.h>
int main(){
        int a=10;
        printf("\nHello world");
        printf("%d",a);
        printf("hello",);
        printf("%d");

        scanf("%d",&a);
        scanf("%d",a);
        scanf("%d",);

}
```

Case6.c

```c
//Testcase to check missing semicolon ,unbalanced parenthesis
#include<stdio.h>
int main(){
        int a=0,b=45,c=10;
        a=a+b

}
}
```

Case7.c

```c
//testcase to check struct and uninon operations
#include<stdio.h>
struct Node {
        int data;
};
union Node2{
        int w;
};

int main(){
        struct Node x;
        printf("\nEnjter Number : ");
        scanf("%d",&x->data);
        return 1;
}
```

TEST CASE EVALUATION

File Name	Purpose	Expected Output	Explanation	Status
Case1.c	Check arithmetic ,relational and logical statements	**Case1.c : 11 : Syntax Error**	Cannot use two arithmetic operator consecutively Unary plus should not have more than one operand	**Passed**
Case2.c	Check control and selection statements	**Case2.c : 13 : Syntax Error**	There is an unbalanced else in line number 13 ,ie there should be else for each if statements	**Passed**

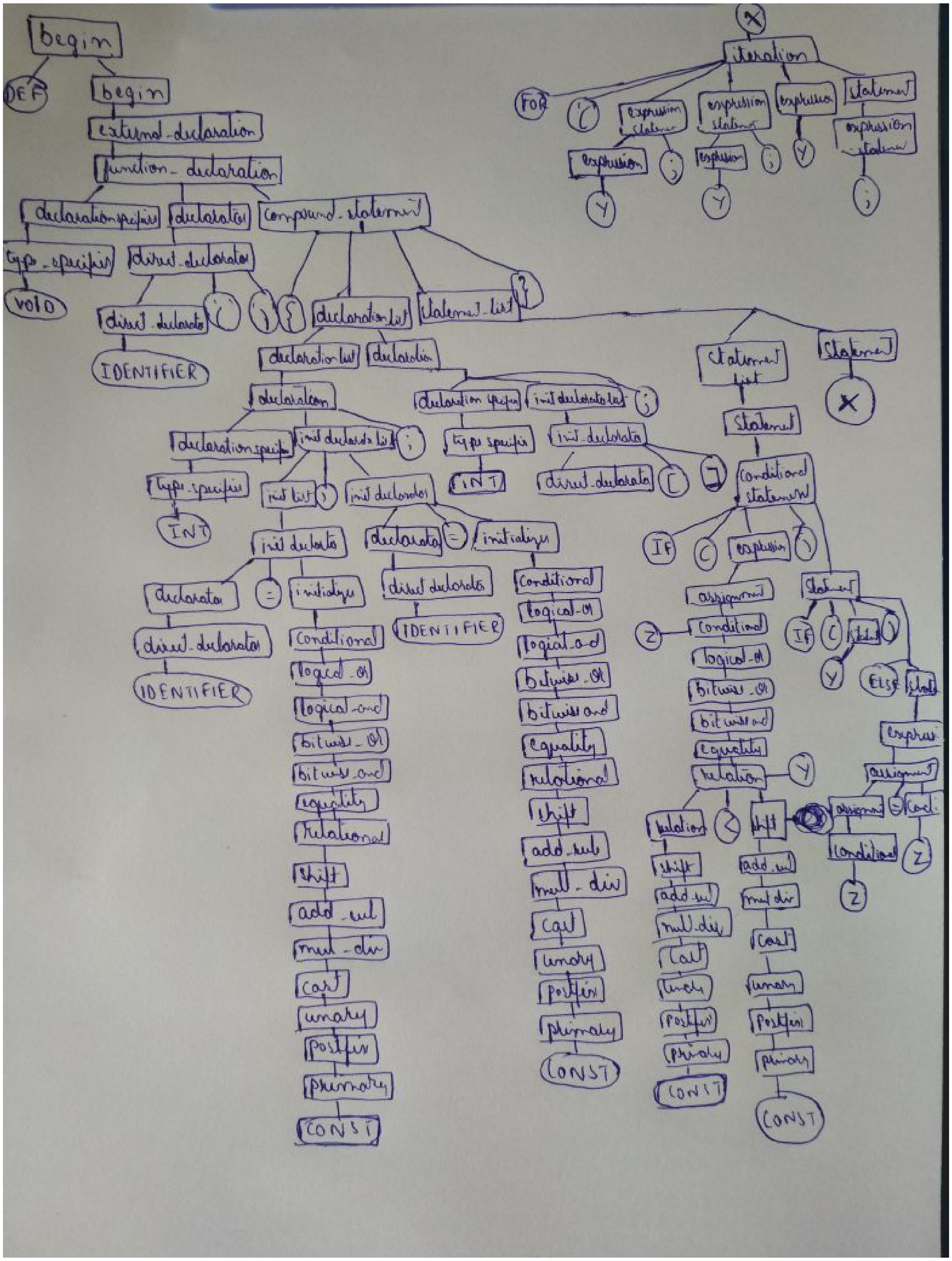
Case3.c	Check for iteration statement validity	**Case3.c : 10 : Syntax Error**	There should be exactly 2 semicolon separates three statements in for loop syntax	**Passed**
Case4.c	Check for function declaration and parameter passing	**Case4.c : Parsing Complete**	All the function declaration and function invoking follows the grammar	**Passed**
Case5.c	Check printf and scanf function invoke errors	**Case5.c : 8 : Syntax Error**	Printf arguments should be separated by comma and should not end with comma	**Passed**
Case6.c	Check unbalanced parenthesis and missing semicolon	**Case6.c : 8: Syntax Error**	Unbalanced parenthesis at the end of the code	**Passed**
Case7.c	Check declaration syntax for structure and union variable	**Case7.c : Parsing Complete**	All the declaration follow the grammar rule	**Passed**

PARSE TREE CONSTRUCTION

We have drawn the parse tree for the following two source programs:

1)

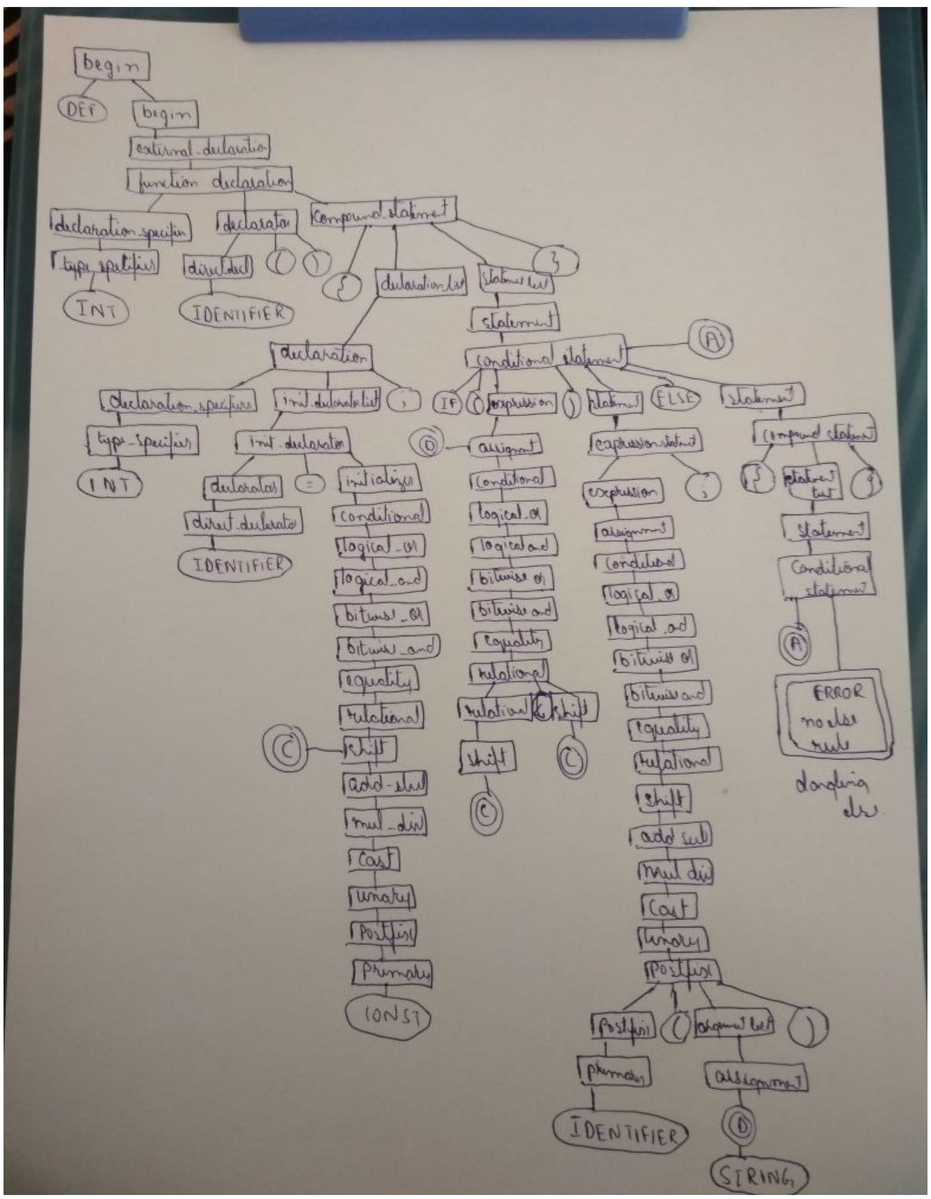
```
//testcase 1 for parse tree
#include<stdio.h>
#define x 3
void main()
{
    int a=4,b=5;
    int d[2]={3,5};
    if(a<10)
        if(a>10)
            a=a+1;
        else
            a=a-1;
for(;i<j;i++,j++);
}
```


begin
DEF
begin
IDENTIFIER
INT
CONST
iteration
FOR
IF
ELSE

2)

```
//testcase 2 for parse tree

//dangling else error
#include<stdio.h>
#define x 3
int main(int argc,int *argv[])
{
int a=4;
if(a<10)
printf("10");
else
{
if(a<12)
printf("11");
else
printf("All");
else
printf("error");}
}
```



CONCLUSION

After the second phase of this project, we have successfully implemented and added a Syntactic Analyzer/Parser to the C Compiler. This takes in the stream of tokens generated by the Lexical Analyzer implemented in the first phase as input. For the output, it displays syntax errors along with their corresponding line number. Symbol table and Constant table have also been added.

The following list of features have been included in this phase :

1. Syntax checking for Arithmetic, Logical and Relational Expressions
2. IF, IF...ELSE and Nested IF statements
3. Validation of Unary Operators
4. Validation of all loops
    a. FOR
    b. WHILE
    c. DO...WHILE
5. Proper declaration of functions and parameter passing
6. Array declaration
7. Structure and Union
8. Scanf and Printf errors
9. Missing semicolon at the end of statements
10. Unbalanced Parentheses