DAYANANDA SAGAR UNIVERSITY

"SYNTAX ANALYZER FOR MINI C LANGUAGE"

A MINI - PROJECT REPORT

Submitted By:

SANTHOSH GC	**ENG18CS0245**
SANTOSH HUGAR	**ENG18CS0246**
PRAVEEN R	**ENG18CS0216**

Of

6TH SEMESTER

BACHELOR OF TECHNOLOGY

in

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

at

DAYANANDA SAGAR UNIVERSITY

SCHOOL OF ENGINEERING, BANGALORE - 560068

Under the Guidance of:

Prof. ANBUNATHAN RAMAIAH, Dept. of CSE, DSU

DAYANANDA SAGAR UNIVERSITY

CERTIFICATE

This is to certify that the Mini-Project report entitled **"SYNTAX ANALYZER FOR MINI C LANGUAGE"** being submitted by **PRAVEEN.R - ENG18CS0216** to Department of Computer Science and Engineering, School of Engineering, Dayananda Sagar University, Bangalore, for the 6th semester B. Tech C.S.E of this university during the academic year 2020-2021.

Date:___________

____

Signature of the Faculty in Charge

____

Signature of the Chairman

DAYANANDA SAGAR UNIVERSITY

CERTIFICATE

This is to certify that the Mini-Project report entitled **"SYNTAX ANALYZER FOR MINI C LANGUAGE"** being submitted by **SANTOSH HUGAR - ENG18CS0246** to Department of Computer Science and Engineering, School of Engineering, Dayananda Sagar University, Bangalore, for the 6th semester B. Tech C.S.E of this university during the academic year 2020-2021.

Date:___________

____

Signature of the Faculty in Charge

____

Signature of the Chairman

DAYANANDA SAGAR UNIVERSITY

CERTIFICATE

This is to certify that the Mini-Project report entitled **““SYNTAX ANALYZER FOR MINI C LANGUAGE”** being submitted by **SANTHOSH GC - ENG18CS0245** to Department of Computer Science and Engineering, School of Engineering, Dayananda Sagar University, Bangalore, for the 6th semester B .Tech C.S.E of this university during the academic year 2020-2021.

Date:___________

____

Signature of the Faculty in Charge

____

Signature of the Chairman

DECLARATION

We **PRAVEEN.R, SANTHOSH GC, SANTOSH HUGAR** and student of **6th semester B. Tech in Computer Science and Engineering, Dayananda Sagar University, Bengaluru,** hereby declare that titled **""SYNTAX ANALYZER FOR MINI C LANGUAGE"** submitted to the Dayananda Sagar University during the academic year 2020- 2021, is a record of an original work done by me under the guidance of **Prof. ANBUNATHAN RAMAIAH**, Assistant professor, Department of computer science engineering, Dayananda Sagar University, Bengaluru. This project work is submitted in partial fulfilment for the award of the degree of Bachelor of Technology in Computer Science. The result embodied in this thesis has not been submitted to any other university or institute for the award of any degree.

ACKNOWLEDGEMENT

The success and final outcome of this software requirement document required a lot of guidance and assistance from many people and we are extremely privileged to have got this all through the completion of the project. All that we have done is only due to such supervision and assistance and we would not forget to thank them.

We respect and thank our mentor, Professors and the Chairman for providing us an opportunity to do the Software Requirement Document and giving us all support and guidance which made me complete the project duty. We are extremely thankful to him for providing such a nice support and guidance, even though he has a busy schedule managing the departmental affairs.

We owe our deep gratitude to our mentor Prof. ANBUNATHAN RAMAIAH, who took keen interest on our Software Requirement Document and guided us all along, till the completion of our project work by providing all the necessary information for developing a good system document.

We are thankful to, and fortunate enough to get constant encouragement, support and guidance from all Teaching staffs of the Computer Science Engineering department, which helped us in successfully completing our report. Also, we would like to extend our sincere esteems to all staff in the laboratory for their timely support.

Team members:

SANTHOSHGC
ENG18CS0245

SANTOSH HUGAR
ENG18CS0246

PRAVEEN R
ENG18CS0216

ABSTRACT

A compiler is a computer program that transforms source code written in a programming language (the source language) into another computer language (the target language), with the latter often having a binary form known as object code. When executing, the compiler first parses all of the language statements syntactically one after the other and then, in one or more successive stages or passes, builds the output code, making sure that statements that refer to other statements are referred to correctly in the final code. Syntax analysis or parsing is the second phase of a compiler. 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. 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. Parsing, syntax analysis or syntactic analysis is the process of analysing a string of symbols, either in natural language or in computer languages, conforming to the rules of a formal grammar. Syntactic analysis, or parsing, is needed to determine if the series of tokens given are appropriate in a language - that is, whether or not the sentence has the right shape/form. However, not all syntactically valid sentences are meaningful, further semantic analysis has to be applied for this. For syntactic analysis, context-free grammars and the associated parsing techniques are powerful enough to be used - this overall process is called parsing. There are many techniques for parsing algorithms, and the two main classes of algorithm are top-down and bottom-up parsing. The top-down parsing uses leftmost derivation, and the bottom-up parsing uses rightmost derivation.

CONTENTS

INTRODUCTION

Syntax Analysis

In computer science, syntax analysis is the process of checking that the code is syntactically correct. The purpose of syntax analysis or parsing is to check that we have a valid sequence of tokens. Tokens are valid sequence of symbols, keywords, identifiers etc. The parser needs to be able to handle the infinite number of possible valid programs that may be presented to it. The usual way to define the language is to specify a grammar. A grammar is a set of rules (or productions) that specifies the syntax of the language (i.e. what is a valid sentence in the language). There can be more than one grammar for a given language. The parser analyzes the source code (token stream) against the production rules to detect any errors in the code. The output of this phase is a parse tree.

The syntax analyser for the C language by writing two scripts, one that acts as a lexical analyzer (lexer) and outputs a stream of tokens, and the other one that acts as a parser.

The lexer is known as the lex program. Lex reads an input stream specifying the lexical analyzer and outputs source code implementing the lexer in the C programming language.

Structure of Lex Program

The structure of the lex program consists of three sections:
{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 generated source file. These statements presumably contain code called by the rules in the rules section.

Structure of Yacc Program

The parser written is known as the Yacc program. The structure of the Yacc file is similar to that of the lexer, consisting of three sections:

```
{declarations}
%%
{rules}
%%
{routines}
```

The **declarations section** of a yacc file may consist of the following:

- %token - identifies the token names that the yacc file accepts
- %start - identifies a nonterminal name for the start symbol
- %right - identifies tokens that are right-associative with other tokens
- %left - identifies tokens that are left-associative with other tokens
- %nonassoc - identifies tokens that are not associative with other tokens

The **rules section** consists of the context free grammar used to generate the parse tree. A general rule has the following structure:

```
        nonterminal
                : sentential form
                | sentential form
                ................
                | sentential form
                ;
```

Actions may be associated with rules and are executed when the associated sentential form is matched.

The **routines section** may include the C program that specifies the input file, action routines and other user defined functions

CODES

Lex Program : (parser.l file)

```
alpha                   [A-Za-z_]
fl              (f|F|l|L)
ul              (u|U|l|L)*
digit           [0-9]
space                   [ ]
hex             [a-fA-F0-9]
exp             [Ee][+-]?{digit}+

%{
//int yylineno = 1;
char datatype[100] = "dummy";
int tl;
char next;
#include <stdio.h>
#include <string.h>
%}

%%
\n { yylineno++; }
"/*"                            { multicomment(); }
"//"                            { singlecomment(); }

"#include<"({alpha})*".h>" {}

"#define"({space})""({alpha})""({alpha}|{digit})*""({space})""({digit})+""                  {
return DEFINE;}
"#define"({space})""({alpha}({alpha}|{digit})*)""({space})""(({digit}+)\.({digit}+))""              {
return DEFINE;}
"#define"({space})""({alpha}({alpha}|{digit})*)""({space})""({alpha}({alpha}|{digit})*)""           {
return DEFINE;}

{digit}+                                { insertToConstTable(yytext, yylineno, "INT"); return
CONSTANT; }
```

```
({digit}+)\.({digit}+)                    {    insertToConstTable(yytext, yylineno, "FLOAT");
return CONSTANT; }
0[xX]{hex}+{ul}?                 {    insertToConstTable(yytext, yylineno, "FLOAT"); return
CONSTANT; }
{digit}+{ul}?                    {    insertToConstTable(yytext, yylineno, "FLOAT"); return
CONSTANT; }
'(\\.|[^\\'])+'                  {    insertToConstTable(yytext, yylineno, "FLOAT"); return
CONSTANT; }
{digit}+{exp}{fl}?               {    insertToConstTable(yytext, yylineno, "FLOAT"); return
CONSTANT; }
{digit}*"."{digit}+({exp})?{fl}?      { insertToConstTable(yytext, yylineno, "FLOAT"); return
CONSTANT; }
{digit}+"."{digit}*({exp})?{fl}?      { insertToConstTable(yytext, yylineno, "FLOAT"); return
CONSTANT; }

{alpha}?\"(\\.|[^\\"])*\"            {    insertToConstTable(yytext, yylineno, "STRING"); return
STRING_LITERAL; }

"->"            {  return PTR_OP; }
"++"            {  return INC_OP; }
"--"            {  return DEC_OP; }
"<<"            {  return LEFT_OP; }
">>"            {  return RIGHT_OP; }
"<="            {  return LE_OP; }
">="            {  return GE_OP; }
"=="            {  return EQ_OP; }
"!="            {  return NE_OP; }
"&&"            {  return AND_OP; }
"||"            {  return OR_OP; }
"*="            {  return MUL_ASSIGN; }
"/="            {  return DIV_ASSIGN; }
"%="                    { return MOD_ASSIGN; }
"+="            {  return ADD_ASSIGN; }
"-="            {  return SUB_ASSIGN; }
"<<="                   { return LEFT_ASSIGN; }
">>="                   { return RIGHT_ASSIGN; }
"&="            {  return AND_ASSIGN; }
"^="            {  return XOR_ASSIGN; }
"|="            {  return OR_ASSIGN; }
```

```
"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"			{ strcpy(datatype, "FLOAT"); tl = yylineno; return FLOAT; }
"for"			{ return FOR; }
"goto"			{ return GOTO; }
"if"			{ return IF; }
"int"			{ strcpy(datatype, "INT"); tl = yylineno; 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; }


";"			{ strcpy(datatype, "dummy"); return(';'); }
("{"|"<%")		{  return('{'); }
("}"|"%>")		{  return('}'); }
","			{  return(','); }
```

```
":"             {  return(':'); }
"="             {  return('='); }
"("             {  return('('); }
")"             {  return(')'); }
("["|"<:")      {  return('['); }
("]"|":>")      {  return(']'); }
"."             {  return('.'); }
"&"             {  return('&'); }
"!"             {  return('!'); }
"~"             {  return('~'); }
"-"             {  return('-'); }
"+"             {  return('+'); }
"*"             {  return('*'); }
"/"             {  return('/'); }
"%"             {  return('%'); }
"<"             {  return('<'); }
">"             {  return('>'); }
"^"             {  return('^'); }
"|"             {  return('|'); }
"?"             {  return('?'); }
"printf"|"scanf"    { insertToHash(yytext,"PROCEDURE",yylineno); return IDENTIFIER; }
"main"                  {       insertToHash(yytext,"PROCEDURE",yylineno);       return
IDENTIFIER; }
{alpha}({alpha}|{digit})*               {
                                        if(strcmp(datatype, "dummy")==0)
                                                return IDENTIFIER;
                                        else
                                        {
                                                insertToHash(yytext,datatype,yylineno);
                                                return IDENTIFIER;
                                        }
                                }
[ \t\v\n\f]     { }

.               { /* ignore bad characters */ }
%%

struct cnode
{
```

char num[50];

```
    //int lno;
    char type[20];
};
struct cnode ctable[100];
int ccount = 0;

void insertToConstTable(char *num, int l, char *type)
{
    strcpy(ctable[ccount].num, num);
    strcpy(ctable[ccount].type, type);
    //ctable[ccount].lno = l;
    ccount++;
}

void disp()
{
    int i;
    printf("\n\n------------------------------CONSTANT TABLE------------------------------\n");
    printf(" --------------------------------------------------------------------------\n");
    printf("Value \t\t|\tData Type\t\t\n");
    printf(" --------------------------------------------------------------------------\n");
    for(i=0;i<ccount;i++)
    {
            printf("%s\t\t\t", ctable[i].num);
            printf("%s\t\t\t\n", ctable[i].type);
            //printf("\t%d\t\n", ctable[i].lno);
    }
    printf("\n\n");
}

struct node
{
    char token[100];
            char attr[100];
            //int line[100];
            int line_count;
    struct node *next;
};
```

```
struct hash
{
   struct node *head;
   int hash_count;
};

struct hash hashTable[1000];
int eleCount = 1000;

struct node * createNode(char *token, char *attr, int l)
{
   struct node *newnode;
   newnode = (struct node *) malloc(sizeof(struct node));
   strcpy(newnode->token, token);
   strcpy(newnode->attr, attr);
   //newnode->line[0] = l;
   newnode->line_count = 1;
   newnode->next = NULL;
   return newnode;
}

int hashIndex(char *token)
{
   int hi=0;
   int l,i;
   for(i=0;token[i]!='\0';i++)
   {
         hi = hi + (int)token[i];
   }
   hi = hi%eleCount;
   return hi;
}

void insertToHash(char *token, char *attr, int l)
{
   int flag=0;
   int hi;
   hi = hashIndex(token);
   struct node *newnode = createNode(token, attr, l);
```

```
    /* head of list for the bucket with index "hashIndex" */
    if (hashTable[hi].head==NULL)
    {
            hashTable[hi].head = newnode;
           hashTable[hi].hash_count = 1;
           return;
    }
    struct node *myNode;
            myNode = hashTable[hi].head;
    while (myNode != NULL)
            {
                    if (strcmp(myNode->token, token)==0)
                    {
                            flag = 1;
                            //myNode->line[(myNode->line_count)++] = l;
                            if(strcmp(myNode->attr, attr)!=0)
                            {
                                    strcpy(myNode->attr, attr);
                            }
                            break;
                    }

                    myNode = myNode->next;
            }
    if(!flag)
    {

            //adding new node to the list
            newnode->next = (hashTable[hi].head);
            //update the head of the list and no of nodes in the current bucket
            hashTable[hi].head = newnode;
            hashTable[hi].hash_count++;
    }
    return;
}

void display()
{
            struct node *myNode;
            int i,j, k=1;
```

```
                                        printf("\n----------------------------------------Symbol
Table---------------------------------------------\n");

printf("------------------------------------------------------------------------------------------------ ");
        printf("\nToken \t\t|\tToken Type \t\t\t\t\t \n");

printf("------------------------------------------------------------------------------------------------ \n");
        for (i = 0; i < eleCount; i++)
        {
                if (hashTable[i].hash_count == 0)
                        continue;
                myNode = hashTable[i].head;
                if (!myNode)
                        continue;
                while (myNode != NULL)
                {
                        //printf("%d\t\t", k++);
                        printf("%s\t\t\t", myNode->token);
                        printf("%s\t\t\t", myNode->attr);
                        /*for(j=0;j<(myNode->line_count);j++)
                                printf("%d  ",myNode->line[j]);*/
                        printf("\n");
                        myNode = myNode->next;
                }
   }
   printf(" ------------------------------------------------------------------------------------------------\n");
        return;
}

yywrap()
{
   return(1);
}
multicomment()
{
   char c, c1;
   while ((c = input()) != '*' && c != 0);
   c1=input();
   if(c=='*' && c1=='/')
```

```
    {
            c=0;
    }
    if (c != 0)
            putchar(c1);
}
singlecomment()
{
    char c;
    while(c=input()!='\n');
    if(c=='\n')
            c=0;
    if(c!=0)
            putchar(c);
}
```

Yacc Program : (parser.y file)

--

```
%{
    int yylineno;
    char data_type[200];
%}

%expect 19

%name parse
%nonassoc NO_ELSE
%nonassoc ELSE
%left '<' '>' '=' GE_OP LE_OP EQ_OP NE_OP
%left '+' '-'
%left '*' '/' '%'
%left  '|'
%left  '&'
%token IDENTIFIER CONSTANT STRING_LITERAL SIZEOF
%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 DEFINE
%token TYPEDEF EXTERN STATIC AUTO REGISTER
%token CHAR SHORT INT LONG SIGNED UNSIGNED FLOAT DOUBLE CONST
VOLATILE VOID
%token STRUCT UNION ENUM
%token CASE DEFAULT IF ELSE SWITCH WHILE DO FOR GOTO CONTINUE
BREAK RETURN
%start begin

%union{
    char str[1000];
}

%%

begin
    : external_declaration
    | begin external_declaration
    | Define begin
    ;

primary_expression
    : IDENTIFIER { insertToHash($<str>1, data_type , yylineno); }
    | CONSTANT
    | STRING_LITERAL
    | '(' expression ')'
    ;

Define
    : DEFINE
    ;

postfix_expression
    : primary_expression
    | postfix_expression '[' expression ']'
    | postfix_expression '(' ')'
    | postfix_expression '(' argument_expression_list ')'
    | postfix_expression '.' IDENTIFIER
    | postfix_expression PTR_OP IDENTIFIER
    | postfix_expression INC_OP
    | postfix_expression DEC_OP
    ;

argument_expression_list
    : assignment_expression
    | argument_expression_list ',' assignment_expression
    ;

unary_expression
    : postfix_expression
    | INC_OP unary_expression
    | DEC_OP unary_expression
    | unary_operator cast_expression
    | SIZEOF unary_expression
    | SIZEOF '(' type_name ')'
    ;

unary_operator
    : '&'
    | '*'
    | '+'
    | '-'
    | '~'
    | '!'
    ;

cast_expression
    : unary_expression
    | '(' type_name ')' cast_expression
    ;

multiplicative_expression
    : cast_expression
    | multiplicative_expression '*' cast_expression
    | multiplicative_expression '/' cast_expression
    | multiplicative_expression '%' cast_expression
    ;
```

```
additive_expression
    : multiplicative_expression
    | additive_expression '+' multiplicative_expression
    | additive_expression '-' multiplicative_expression
    ;

shift_expression
    : additive_expression
    | shift_expression LEFT_OP additive_expression
    | shift_expression RIGHT_OP additive_expression
    ;

relational_expression
    : shift_expression
    | relational_expression '<' shift_expression
    | relational_expression '>' shift_expression
    | relational_expression LE_OP shift_expression
    | relational_expression GE_OP shift_expression
    ;

equality_expression
    : relational_expression
    | equality_expression EQ_OP relational_expression
    | equality_expression NE_OP relational_expression
    ;

and_expression
    : equality_expression
    | and_expression '&' equality_expression
    ;

exclusive_or_expression
    : and_expression
    | exclusive_or_expression '^' and_expression
    ;

inclusive_or_expression
    : exclusive_or_expression
    | inclusive_or_expression '|' exclusive_or_expression
    ;

logical_and_expression
    : inclusive_or_expression
    | logical_and_expression AND_OP inclusive_or_expression
    ;

logical_or_expression
    : logical_and_expression
    | logical_or_expression OR_OP logical_and_expression
    ;

conditional_expression
    : logical_or_expression
    | logical_or_expression '?' expression ':' conditional_expression
    ;

assignment_expression
    : conditional_expression
    | unary_expression assignment_operator assignment_expression
    ;

assignment_operator
    : '='
    | MUL_ASSIGN
    | DIV_ASSIGN
    | MOD_ASSIGN
    | ADD_ASSIGN
    | SUB_ASSIGN
    | LEFT_ASSIGN
    | RIGHT_ASSIGN
    | AND_ASSIGN
    | XOR_ASSIGN
    | OR_ASSIGN
    ;

expression
    : assignment_expression
    | expression ',' assignment_expression
    ;

constant_expression
    : conditional_expression
    ;

declaration
    : declaration_specifiers ';'
    | declaration_specifiers init_declarator_list ';'
    ;

declaration_specifiers
    : storage_class_specifier
    | storage_class_specifier declaration_specifiers
    | type_specifier    { strcpy(data_type, $<str>1); }
    | type_specifier declaration_specifiers
    ;

init_declarator_list
    : init_declarator
    | init_declarator_list ',' init_declarator
    ;

init_declarator
    : declarator
    | declarator '=' initializer
    ;

storage_class_specifier
    : TYPEDEF
    | EXTERN
    | STATIC
    | AUTO
    | REGISTER
    ;

type_specifier
    : VOID
    | CHAR
    | SHORT
    | INT
    | LONG
    | FLOAT
    | DOUBLE
    | SIGNED
    | UNSIGNED
    | struct_or_union_specifier
    ;

specifier_qualifier_list
    : type_specifier specifier_qualifier_list
    | type_specifier
    | CONST specifier_qualifier_list
    | CONST
    ;

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
    | UNION
    ;

struct_declaration_list
    : struct_declaration
    | struct_declaration_list struct_declaration
    ;

struct_declaration
    : specifier_qualifier_list struct_declarator_list ';'
    ;
```

```
struct_declarator_list
    : declarator
    | struct_declarator_list ',' declarator
    ;

declarator
    : pointer direct_declarator
    | direct_declarator
    ;

direct_declarator
    : IDENTIFIER
    | '(' declarator ')'
    | direct_declarator '[' constant_expression ']'
    | direct_declarator '[' ']'
    | direct_declarator '(' parameter_list ')'
    | direct_declarator '(' identifier_list ')'
    | direct_declarator '(' ')'
    ;

pointer
    : '*'
    | '*' pointer
    ;

parameter_list
    : parameter_declaration
    | parameter_list ',' parameter_declaration
    ;

parameter_declaration
    : declaration_specifiers declarator
    | declaration_specifiers
    ;

identifier_list
    : IDENTIFIER
    | identifier_list ',' IDENTIFIER
    ;
```

```
type_name
    : specifier_qualifier_list
    | specifier_qualifier_list declarator
    ;

initializer
    : assignment_expression
    | '{' initializer_list '}'
    | '{' initializer_list ',' '}'
    ;

initializer_list
    : initializer
    | initializer_list ',' initializer
    ;

statement
    : compound_statement
    | expression_statement
    | selection_statement
    | iteration_statement
    | jump_statement
    ;

compound_statement
    : '{' '}'
    | '{' statement_list '}'
    | '{' declaration_list '}'
    | '{' declaration_list statement_list '}'
    ;

declaration_list
    : declaration
    | declaration_list declaration
    ;

statement_list
    : statement
    | statement_list statement
    ;

expression_statement
    : ';'
    | expression ';'
    ;

selection_statement
    : IF '(' expression ')' statement %prec NO_ELSE
    | IF '(' expression ')' statement ELSE statement
    ;

iteration_statement
    : WHILE '(' expression ')' statement
    | DO statement WHILE '(' expression ')' ';'
    | FOR '(' expression_statement expression_statement ')' statement
    | FOR '(' expression_statement expression_statement expression ')' statement
    ;

jump_statement
    : CONTINUE ';'
    | BREAK ';'
    | RETURN ';'
    | RETURN expression ';'
    ;

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
    ;
%%
```

```
#include  "lex.yy.c"
#include  <stdio.h>
#include <string.h>
int main(int argc, char *argv[])
{
   yyin = fopen(argv[1], "r");
   if(!yyparse())
         printf("\nParsing complete\n");
   else
         printf("\nParsing failed\n");
   fclose(yyin);
   display();
   disp();
   return 0;
}
//extern int lineno;
extern char *yytext;
yyerror(char *s) {
   printf("\nLine %d : %s\n", (yylineno), s);
}
```

IMPLEMENTATION

The Yacc program specifies productions for the following:

- Looping construct: while, for, do-while
- Data types: (signed/unsigned) int, float
- Arithmetic and Relational Operators
- Data structure: Arrays
- User defined functions
- Keywords of C language
- Single and Multi-line comments
- Identifiers and Constant errors
- Selection statement: (nested) if-else

The productions for most of them are straight-forward. A few important ones are:

```
compound_statement
   : '{' '}'
   | '{' statement_list '}'
```

```
    | '{' declaration_list '}'
    | '{' declaration_list statement_list '}'
    ;



selection_statement
    : IF '(' expression ')' statement %prec NO_ELSE
    | IF '(' expression ')' statement ELSE statement
    ;

iteration_statement
    : WHILE '(' expression ')' statement
    | DO statement WHILE '(' expression ')' ';'
    | FOR '(' expression_statement expression_statement ')' statement
    | FOR '(' expression_statement expression_statement expression ')' statement
    ;

jump_statement
    : CONTINUE ';'
    | BREAK ';'
    | RETURN ';'
    | RETURN expression ';'
    ;

statement
    : compound_statement
    | expression_statement
    | selection_statement
    | iteration_statement
    | jump_statement
    ;
```

After parsing, if there are errors then the line numbers of those errors are displayed along with a ‘parsing failed’ message on the terminal. Otherwise, a ‘parsing complete’ message is displayed on the console.

When a grammar is not carefully thought out, the parser generated from the grammar may face two kinds of dilemmas.

Shift-Reduce Conflict:
Enough terms have been read and a grammar rule can be recognized according to the accumulated terms. In this situation, the parser can make a reduction. If, however, there is also another grammar rule which calls for more terms to be accumulated and the look ahead token is just what the second grammar rule expected. In this situation, the parser may also make a shift operation. This dilemma faced by the parser is called the Shift/Reduce Conflict.

Reduce-Reduce Conflict:
Enough terms have been read and two grammar rules are recognized based on the accumulated terms. If the parser then decides to make a reduction, should it reduces the accumulated terms according to the first or second grammar rules? This type of difficulty faced by the parser is called the Reduce/Reduce Conflict.

Handling Shift-Reduce and Reduce-Reduce Conflicts

The yacc command is built with two internal rules for resolving these two ambiguities, sometimes called the disambiguating rules.

1. yacc resolves the Shift/Reduce Conflict in favor of shift operation. In plain English, this just means that it matches the longest input possible.
2. yacc resolves the Reduce/Reduce Conflict in favor of the first grammar rule.

In general, whenever it is possible to apply disambiguating rules to produce a correct parser, it is also possible to rewrite the grammar rules so that the same inputs are read but there are no conflicts. For this reason, it is important to write unambiguous grammar with no conflicts.

Solving Dangling Else Problem

Whenever it is not possible to associate an ‘else’ to a closest ‘if’ in if-else statements, it gives rise to dangling else problem. In this case, the problem of dangling else occurs here, represented by #:

IF '(' expression ')' statement # ELSE statement

The question the parser must answer is "should I shift, or should I reduce". Usually, you want to bind the else to the closest if, which means you want to shift the else token now. Reducing now would mean that you want the else to wait to be bound to an older if.

We should specify the parser generator that "when there is a shift/reduce conflict between the token ELSE and the rule "selection_statement -> IF (expression) statement", then the token must win". To do so, a name is given to the precedence of your rule (e.g., NO_ELSE), and specify that NO_ELSE has less precedence than ELSE.

```
//Precedences go increasing, So, NO_ELSE < ELSE
%nonassoc NO_ELSE
%nonassoc ELSE

%%
selection_statement
    : IF '(' expression ')' statement %prec NO_ELSE
    | IF '(' expression ')' statement ELSE statement
    ;
%%
```

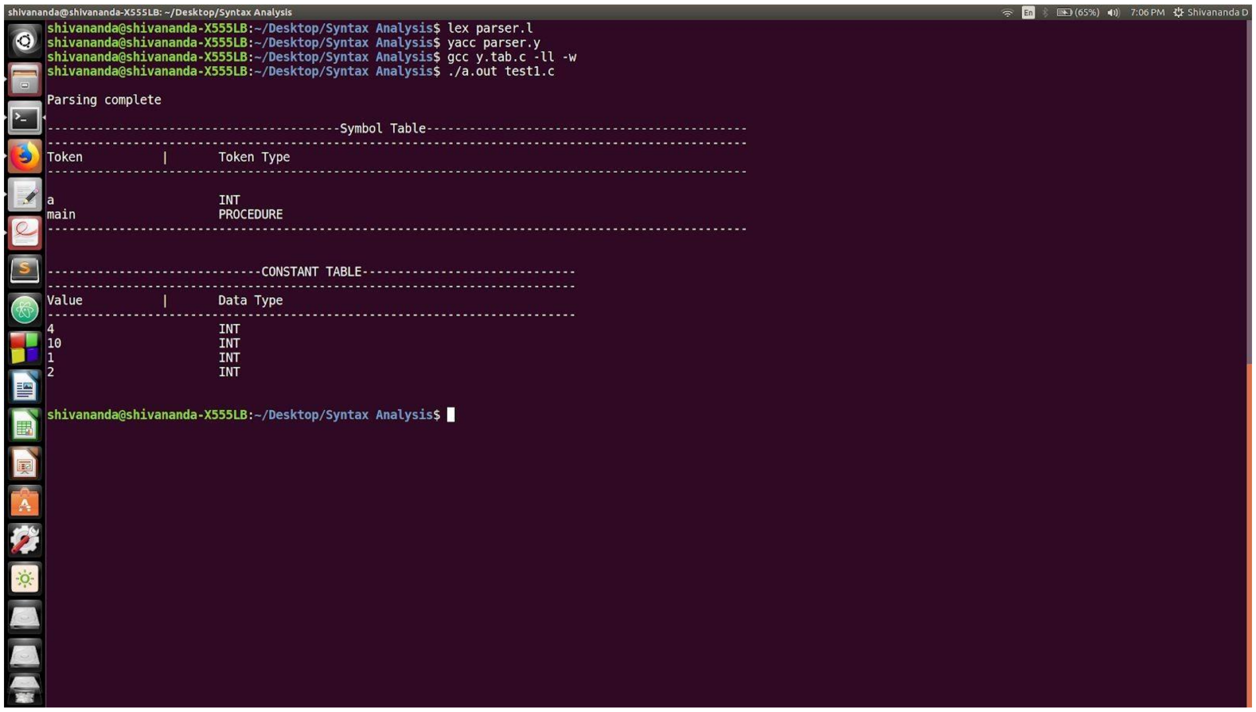
TEST CASES

Without Errors

1. test1.c

```
//without error - nested if-else
#include<stdio.h>
#define x 3
int main()
{
    int a=4;
    if(a<10)
    {
            a = a + 1;
    }
    else
    {
            a = a + 2;
    }

    return;
}
```

Output:

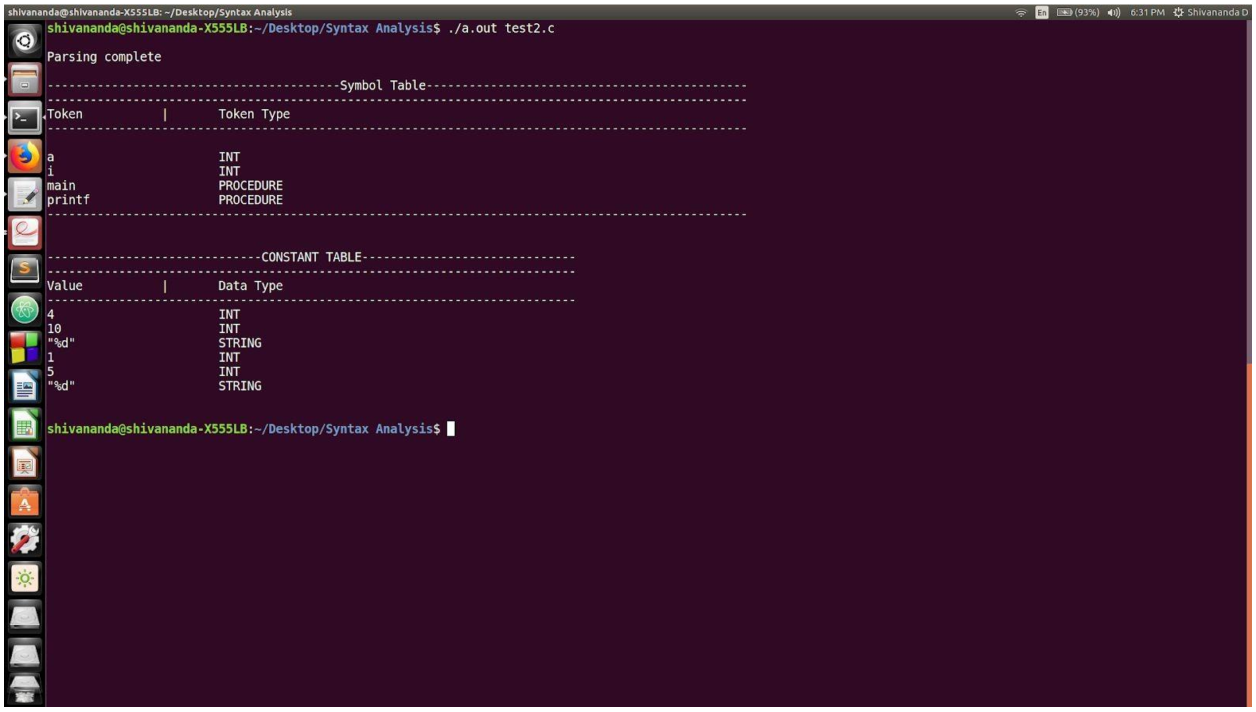


```
shivananda@shivananda-X555LB:~/Desktop/Syntax Analysis$ lex parser.l
shivananda@shivananda-X555LB:~/Desktop/Syntax Analysis$ yacc parser.y
shivananda@shivananda-X555LB:~/Desktop/Syntax Analysis$ gcc y.tab.c -ll -w
shivananda@shivananda-X555LB:~/Desktop/Syntax Analysis$ ./a.out test1.c

Parsing complete

-----------------------------------------Symbol Table-------------------------------------------
------------------------------------------------------------------------------------------------
Token           |       Token Type
------------------------------------------------------------------------------------------------

a                       INT
main                    PROCEDURE
------------------------------------------------------------------------------------------------


------------------------------CONSTANT TABLE-----------------------------
-------------------------------------------------------------------------
Value           |       Data Type
-------------------------------------------------------------------------
4                       INT
10                      INT
1                       INT
2                       INT


shivananda@shivananda-X555LB:~/Desktop/Syntax Analysis$ 
```

2. test2.c

```
//without error - while and for loop
#include<stdio.h>
#define x 3
int main()
{
    int a=4;
    int i;
    while(a<10)
    {
        printf("%d",a);
        a++;
    }
    for(i=1;i<5;i++)
        printf("%d",i);
}
```

Output:



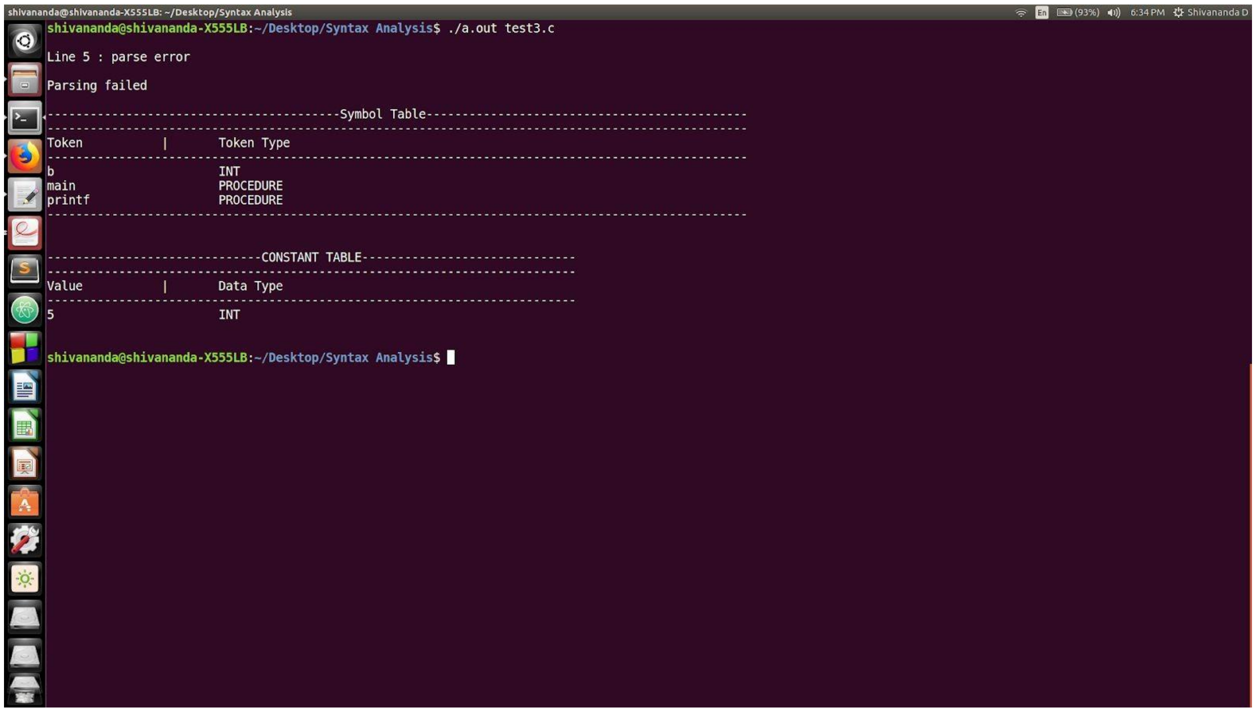
```
shivananda@shivananda-X555LB: ~/Desktop/Syntax Analysis
shivananda@shivananda-X555LB:~/Desktop/Syntax Analysis$ ./a.out test2.c

Parsing complete

----------------------------------------Symbol Table--------------------------------------------
------------------------------------------------------------------------------------------------
Token           |       Token Type
------------------------------------------------------------------------------------------------

a                       INT
i                       INT
main                    PROCEDURE
printf                  PROCEDURE
------------------------------------------------------------------------------------------------


-----------------------------CONSTANT TABLE-----------------------------
------------------------------------------------------------------------
Value           |       Data Type
------------------------------------------------------------------------
4                       INT
10                      INT
"%d"                    STRING
1                       INT
5                       INT
"%d"                    STRING


shivananda@shivananda-X555LB:~/Desktop/Syntax Analysis$ 
```

With Errors

1. test3.c (missing multiple semicolon)

```
#include<stdio.h>
#define x 3
int main()
{
    int b=5
    printf("%d",b);
    while(b<10)
    {
            printf("%d", b)
            b++;
    }
}
```

Output:



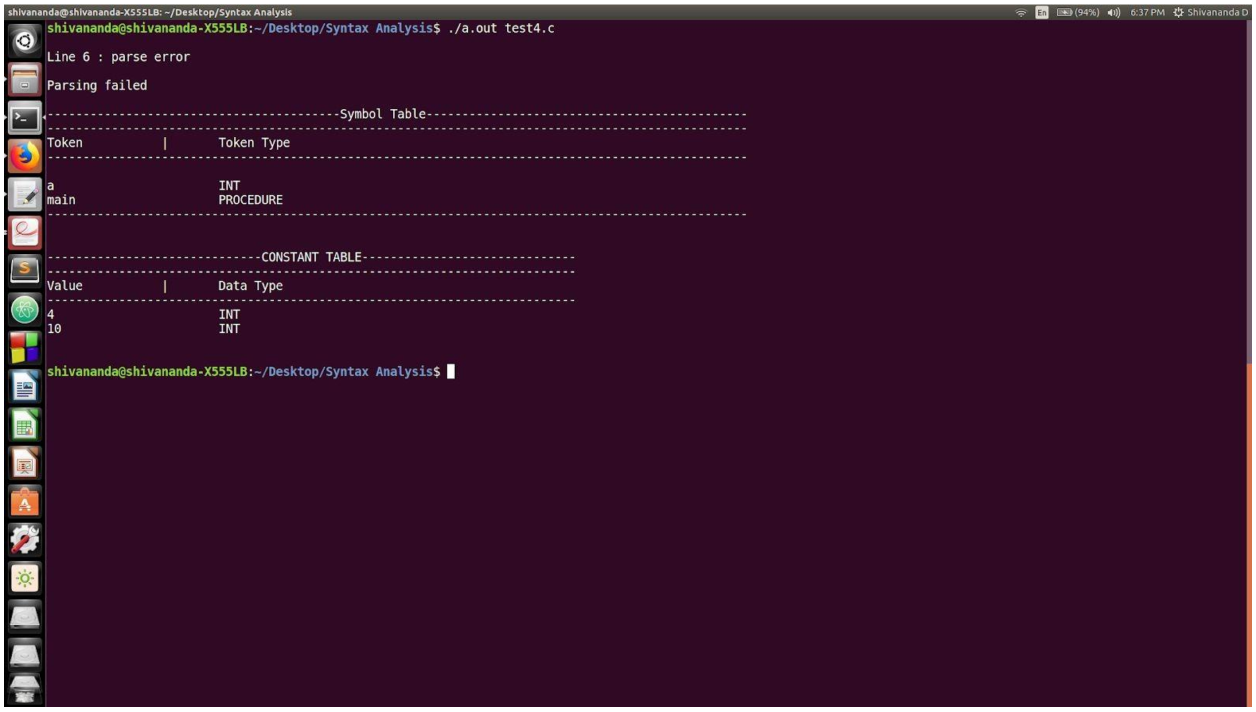
```
shivananda@shivananda-X555LB: ~/Desktop/Syntax Analysis
shivananda@shivananda-X555LB:~/Desktop/Syntax Analysis$ ./a.out test3.c

Line 5 : parse error

Parsing failed

----------------------------------------Symbol Table-------------------------------------------
-----------------------------------------------------------------------------------------------
Token           |       Token Type
-----------------------------------------------------------------------------------------------
b                       INT
main                    PROCEDURE
printf                  PROCEDURE
-----------------------------------------------------------------------------------------------


-----------------------------CONSTANT TABLE-----------------------------
------------------------------------------------------------------------
Value           |       Data Type
------------------------------------------------------------------------
5                       INT


shivananda@shivananda-X555LB:~/Desktop/Syntax Analysis$
```

2. test4.c (misspelled keyword 'while' and unmatched parentheses)

```
#include<stdio.h>
#define x 3
int main()
{
    int a=4;
    whle(a<10)
    {
            printf("%d",a);
            j=1;
```

```
        while(j<=4)
                j++;
    }    //unmatched parantheses
    }
}
```

Output:

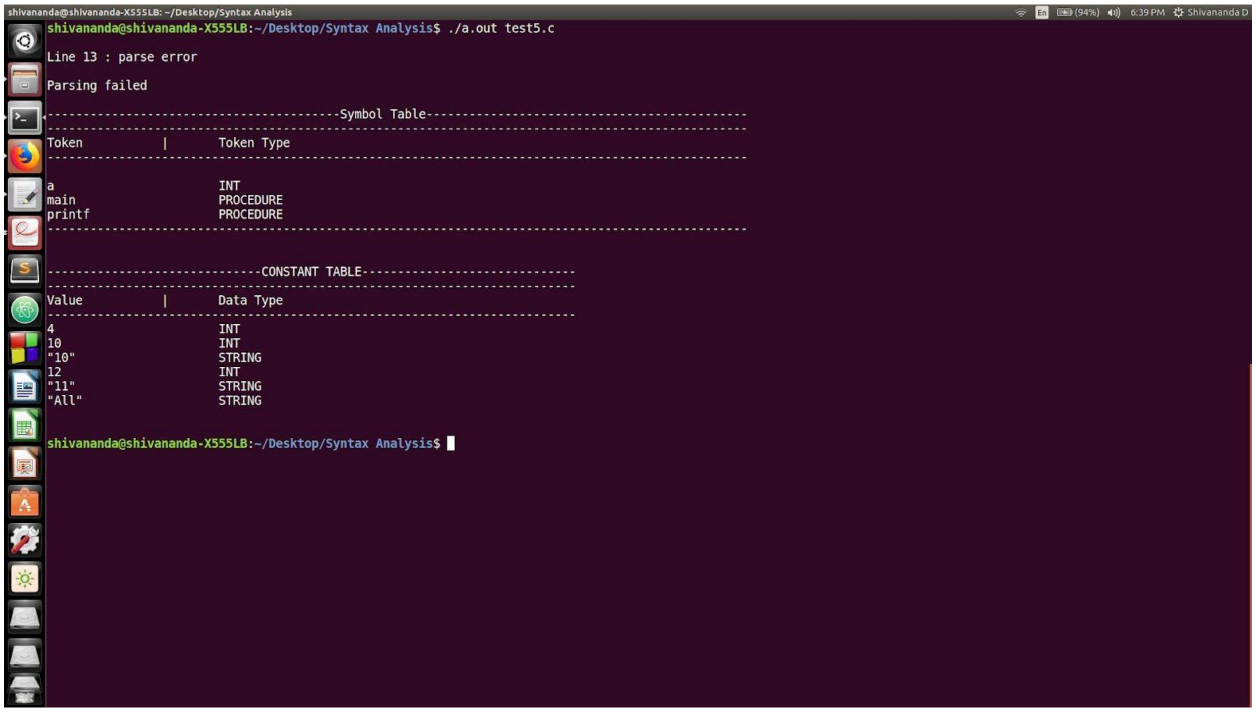


```
shivananda@shivananda-X555LB:~/Desktop/Syntax Analysis$ ./a.out test4.c

Line 6 : parse error

Parsing failed

----------------------------------------Symbol Table--------------------------------------------
------------------------------------------------------------------------------------------------
Token           |       Token Type
------------------------------------------------------------------------------------------------

a                       INT
main                    PROCEDURE
------------------------------------------------------------------------------------------------


-----------------------------CONSTANT TABLE-----------------------------
-----------------------------------------------------------------------
Value           |       Data Type
-----------------------------------------------------------------------
4                       INT
10                      INT


shivananda@shivananda-X555LB:~/Desktop/Syntax Analysis$
```

3. test5.c (dangling else problem)

```
#include<stdio.h>
#define x 3
int main()
{
    int a=4;
    if(a<10)
        printf("10");
    else
    {
            if(a<12)
                        printf("11");
            else
                        printf("All");
            else
                        printf("error");
    }
}
```

Output:

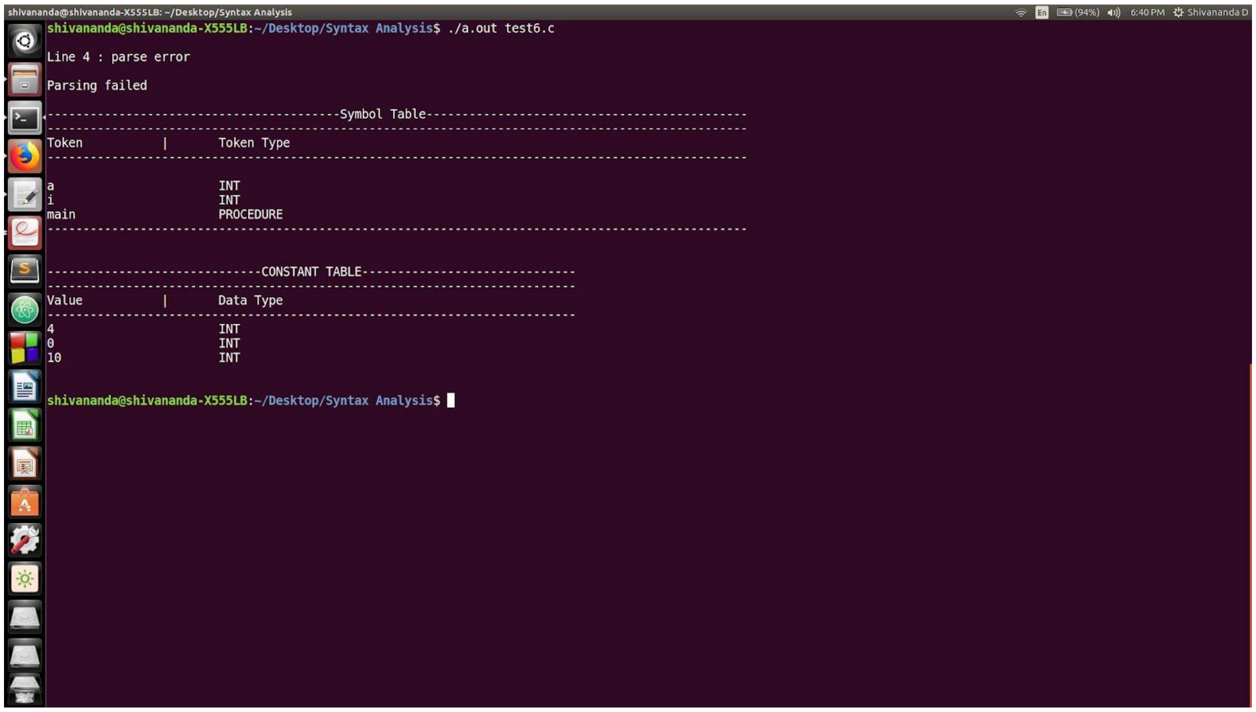


```
shivananda@shivananda-X555LB: ~/Desktop/Syntax Analysis
shivananda@shivananda-X555LB:~/Desktop/Syntax Analysis$ ./a.out test5.c

Line 13 : parse error

Parsing failed

----------------------------------------Symbol Table-------------------------------------------
-----------------------------------------------------------------------------------------------
Token           |       Token Type
-----------------------------------------------------------------------------------------------

a                       INT
main                    PROCEDURE
printf                  PROCEDURE
-----------------------------------------------------------------------------------------------


-----------------------------CONSTANT TABLE-----------------------------
------------------------------------------------------------------------
Value           |       Data Type
------------------------------------------------------------------------
4                       INT
10                      INT
"10"                    STRING
12                      INT
"11"                    STRING
"All"                   STRING


shivananda@shivananda-X555LB:~/Desktop/Syntax Analysis$ 
```

4. Test6.c (for loop syntax error)

```
#include<stdio.h>
int main()
{
    int a=4, i;
    for(i=0;i<10)
    {
            printf("%d",i);
    }
}
```

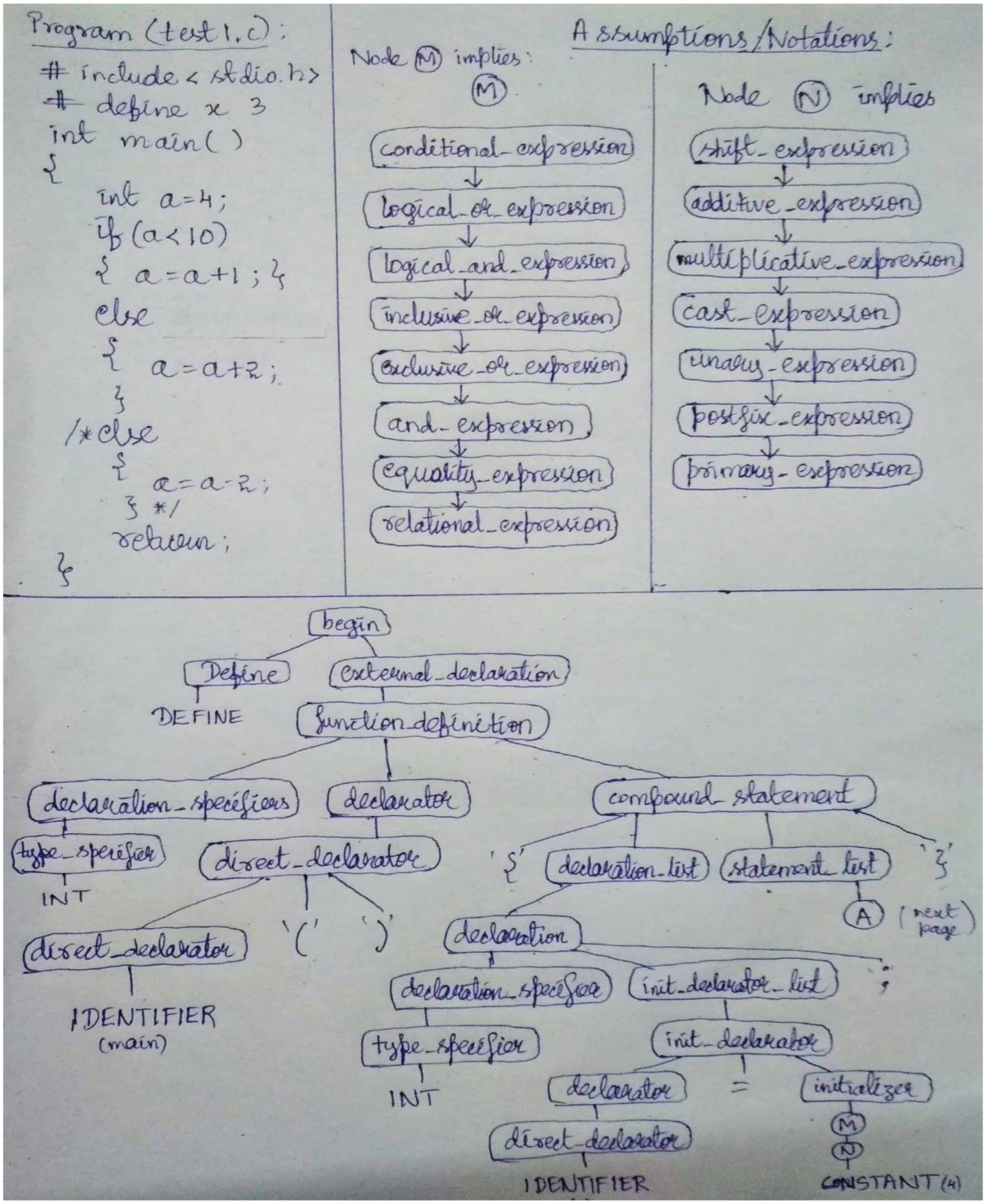
Output:



```
shivananda@shivananda-X555LB:~/Desktop/Syntax Analysis$ ./a.out test6.c

Line 4 : parse error

Parsing failed

----------------------------------------Symbol Table--------------------------------------------
------------------------------------------------------------------------------------------------
Token           |       Token Type
------------------------------------------------------------------------------------------------

a                       INT
i                       INT
main                    PROCEDURE
------------------------------------------------------------------------------------------------


-----------------------------CONSTANT TABLE-----------------------------
------------------------------------------------------------------------
Value           |       Data Type
------------------------------------------------------------------------
4                       INT
0                       INT
10                      INT


shivananda@shivananda-X555LB:~/Desktop/Syntax Analysis$ 
```

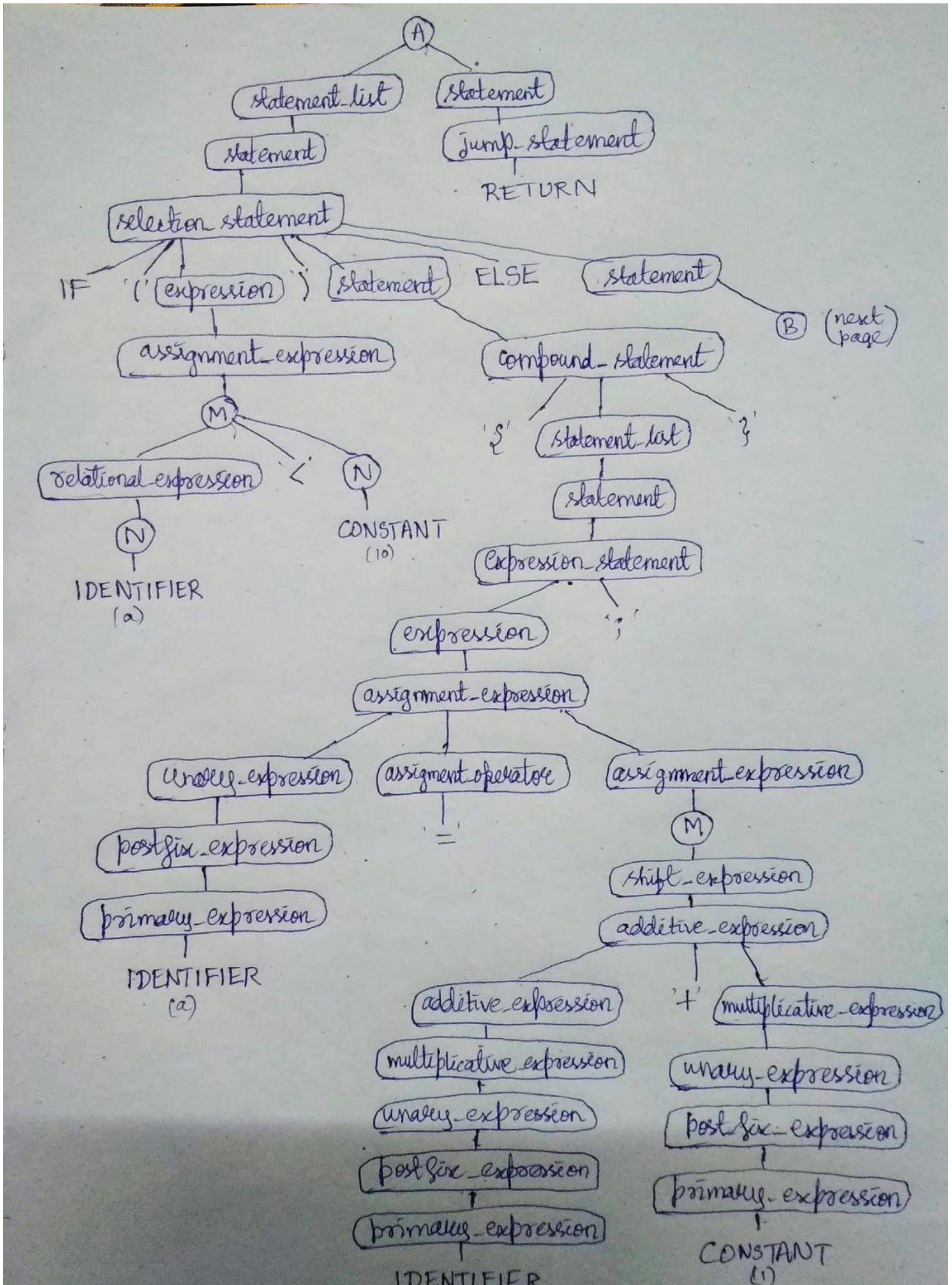
PARSE TREE

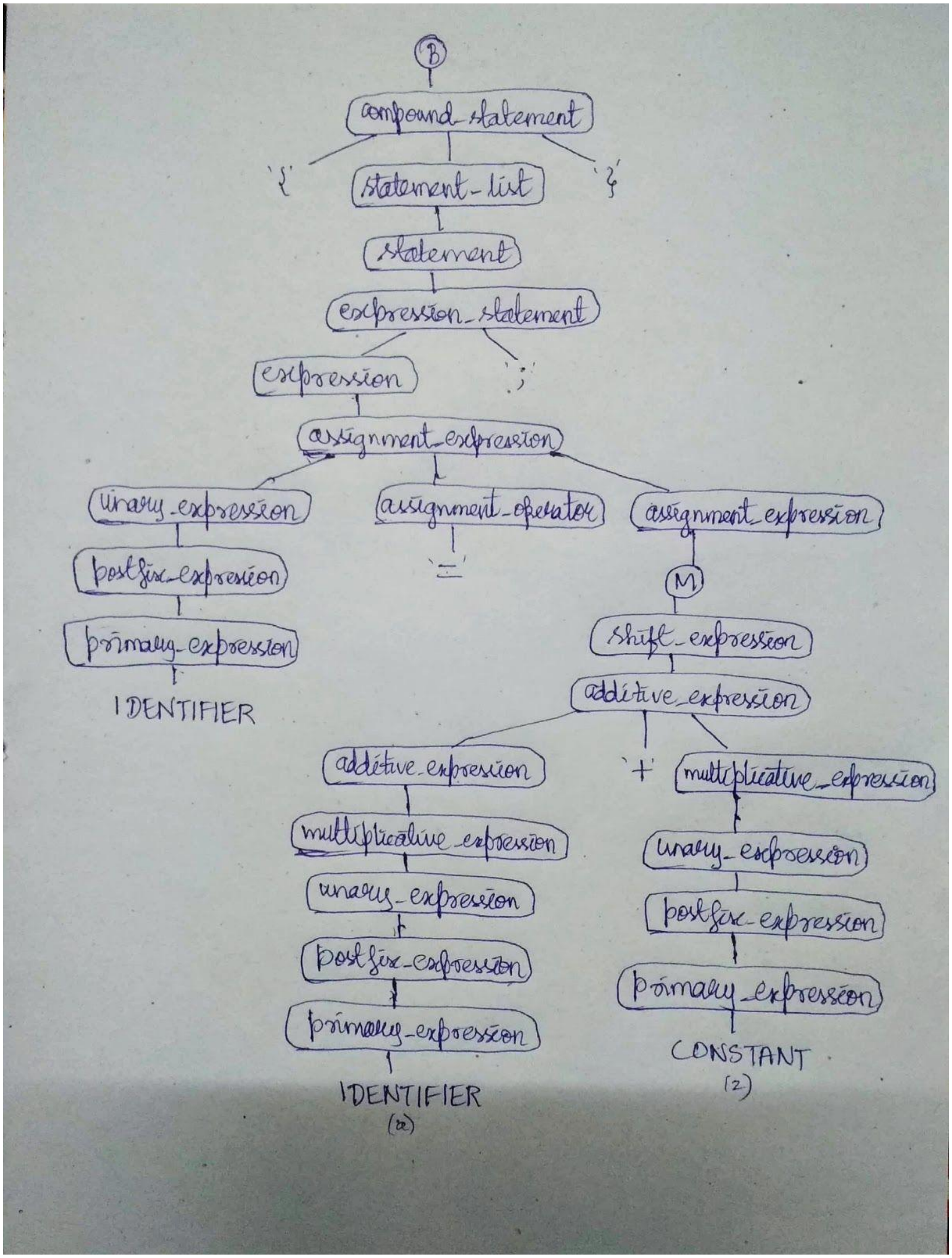
(input program: test1.c)



Program (test1.c):

```
#include <stdio.h>
#define x 3
int main( )
{
    int a=4;
    if (a<10)
    { a=a+1; }
    else
     {
        a=a+2;
     }
  /*else
     {
        a=a-2;
     } */
    return;
}
```


A
statement_list
statement
statement
jump_statement
RETURN
selection_statement
IF
'('
expression
')'
statement
ELSE
statement
B (next page)
assignment_expression
compound_statement
M
'{'
statement_list
'}'
relational_expression
'<'
N
statement
N
CONSTANT (10)
expression_statement
IDENTIFIER (a)
expression
';'
assignment_expression
unary_expression
assigment_operator
assignment_expression
'='
postfix_expression
M
primary_expression
shift_expression
IDENTIFIER (a)
additive_expression
additive_expression
'+'
multiplicative_expression
multiplicative_expression
unary_expression
unary_expression
postfix_expression
postfix_expression
primary_expression
primary_expression
CONSTANT (1)
IDENTIFIER (a)


B
compound_statement
'{'
statement_list
'}'
statement
expression_statement
expression
';'
assignment_expression
unary_expression
assignment_operator
assignment_expression
postfix_expression
'='
M
primary_expression
shift_expression
IDENTIFIER
additive_expression
additive_expression
'+'
multiplicative_expression
multiplicative_expression
unary_expression
unary_expression
postfix_expression
postfix_expression
primary_expression
primary_expression
CONSTANT
(2)
IDENTIFIER
(x)

CONCLUSION

The lexical analyzer and the syntax analyzer for a subset of C language, which include selection statements, compound statements, iteration statements, jumping statements, user defined functions and primary expressions is generated. It is important to note that conflicts (shift-reduce and reduce-reduce) may occur in case of syntax analyzer if proper care is not taken while specifying the context-free grammar for the language. We should always specify unambiguous grammar for the parser to work properly.

FUTURE WORK

We have implemented the parser for only a subset of C language. The future work may include specifying the grammar for more pre-defined functions in C (like string functions, file read and write functions), goto jump statements and enumerations.

REFERENCES

1. http://epaperpress.com/lexandyacc/download/LexAndYaccTutorial.pdf - Lex and Yacc Tutorial by Tom Nieman
2. Compilers – Principles, Techniques and Tools By Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman
3. https://www.programiz.com/c-programming/precedence-associativity-operators - C Precedence and Associativity