# COMPILER DESIGN LAB (CO351)

**REPORT**

**Project 2 : Parser(Syntax Analyzer) for C-language**

**Submitted by :-**

**(1) Patel Harsh Rakeshkumar (15CO233)**

**(2) Shah Harsh Kamlesh (15CO243)**

---

## INDEX

## ABSTRACT

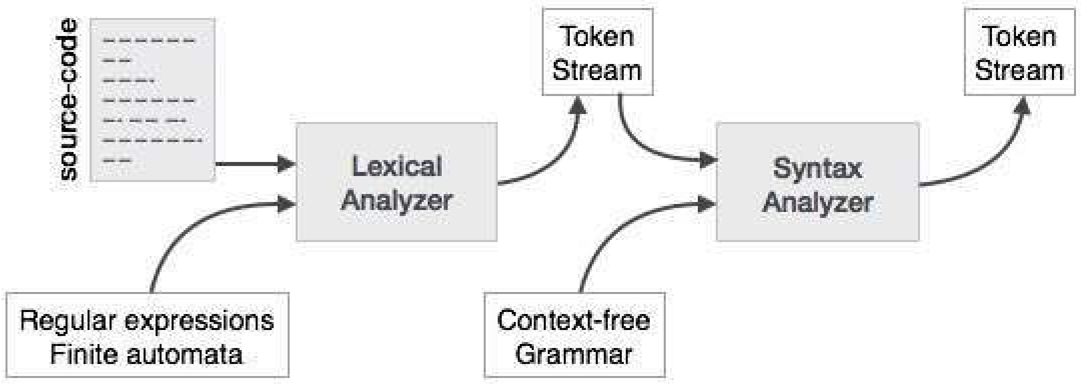
**The parser will have the following functionalities :-**

- We are using YACC to build a parser (syntax analyzer) for C language.
- Our parser takes in stream of tokens from the lexical analyzer and checks if the syntax is correct or not.
- The parser generates the symbol table containing identifiers from the stream of tokens provided to it by the scanner.
  The parser also generates the constant table containing numeric constants and string literals.
- The parser recognizer syntax errors and it also gives an error message along with the line number where the error is identified.
  Also the parser can handle multiple syntax errors at the same time and it also shows the lexical errors along with the syntax errors.
- The parser can recognize complex expressions and handle normal declarations and definitions.
- The parser parses for do-while, while and for loops. It also parser if-else ladder and switch-case statements.
- The parser can parse function declarations, function definitions and function calls.

---

## SYNTAX ANALYSIS (PARSING)

Syntax analysis or parsing is the second phase of a compiler.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. The output of this phase is a **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.
This way, the parser accomplishes two tasks, i.e., parsing the code, looking for errors and generating a parse tree as the output of the phase.
Parsers are expected to parse the whole code even if some errors exist in the program. Parsers use error recovering strategies

## Limitations of Syntax Analyzers

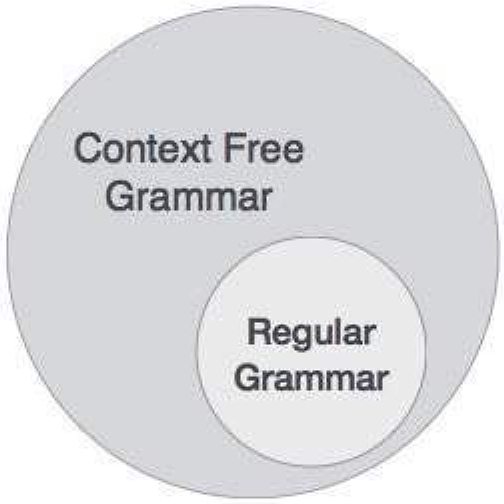
Syntax analyzers receive their inputs, in the form of tokens, from lexical analyzers. Lexical analyzers are responsible for the validity of a token supplied by the syntax analyzer. Syntax analyzers have the following drawbacks -

- it cannot determine if a token is valid,
- it cannot determine if a token is declared before it is being used,
- it cannot determine if a token is initialized before it is being used,
- it cannot determine if an operation performed on a token type is valid or not.

## CONTEXT FREE GRAMMARS

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.

CFG, is a superset of Regular Grammar, as depicted below:



It implies that every Regular Grammar is also context-free, but there exists some problems, which are beyond the scope of Regular Grammar. CFG is a helpful tool in describing the syntax of programming languages.
A context free grammar consists of a finite set of **Non-Terminal** symbols (V), a set of **Terminal** symbols Σ, A **start symbol** from the set of non-terminals S, and a set of **Productions**.
G = ( V, Σ, P, S )

**Derivation -** A derivation is basically a sequence of production rules, in order to get the input string. During parsing, we take two decisions for some sentential form of input:

- Deciding the non-terminal which is to be replaced.
- Deciding the production rule, by which, the non-terminal will be replaced.

To decide which non-terminal to be replaced with production rule, we can have two options.

**Left-most Derivation -** If the sentential form of an input is scanned and replaced from left to right, it is called left-most derivation. The sentential form derived by the left-most derivation is called the left-sentential form.

**Right-most Derivation -** If we scan and replace the input with production rules, from right to left, it is known as right-most derivation. The sentential form derived from the right-most derivation is called the right-sentential form.

**Associativity -** If an operand has operators on both sides, the side on which the operator takes this operand is decided by the associativity of those operators. If the operation is left-associative, then the operand will be taken by the left operator or if the operation is right-associative, the right operator will take the operand.

**Precedence -** If two different operators share a common operand, the precedence of operators decides which will take the operand. That is, 2+3*4 can have two different parse trees, one corresponding to (2+3)*4 and another corresponding to 2+(3*4). By setting precedence among operators, this problem can be easily removed. As in the previous example, mathematically * (multiplication) has precedence over + (addition), so the expression 2+3*4 will always be interpreted as: 2 + (3 * 4) These methods decrease the chances of ambiguity in a language or its grammar.

---

## Yacc

Yacc will read your grammar and generate C code for a syntax analyzer or parser. The syntax analyzer uses grammar rules that allow it to analyze tokens from the lexical analyzer and create a syntax tree. The syntax tree imposes a hierarchical structure the tokens. For example, operator precedence and associativity are apparent in the syntax tree.

Grammars for yacc are described using a variant of Backus Naur Form (BNF). This technique,pioneered by John Backus and Peter Naur, was used to describe ALGOL 60. A BNF grammar can be used to express context-free languages. Most constructs in modern programming language scan be represented in BNF.Names refer to either tokens or nonterminal symbols. Yacc requires token names to be declared as such. In addition, for reasons discussed in Section 3, it is often desirable to include the lexical analyzer as part of the specification file; it may be useful to include other programs as well. Thus, 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.)

In other words, a full specification file looks like:

```
declarations
      %%
      rules
      %%
      programs
```

Yacc turns the grammar 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. The states of the finite state machine are given small integer labels. Initially, the machine is in state 0, the stack contains only state 0, and no lookahead token has been read.

The machine has only four actions available to it, called shift, reduce, accept, and error.

A move of the parser is done as follows:

1. Based on its current state, the parser decides whether it needs a lookahead token to decide what action should be done; if it needs one, 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.

Whenever a shift action is taken, there is always a lookahead token. The reduce action keeps the stack from growing without bounds. The accept action indicates that the entire input has been seen and that it matches the specification. This action appears only when the lookahead token is the endmarker, and indicates that the parser has successfully done its job. The error action, on the other hand, represents a place where the parser can no longer continue parsing according to the specification. The input

tokens it has seen, together with the lookahead token, cannot be followed by anything that would result in a legal input. The parser reports an error, and attempts to recover the situation and resume parsing.
When there are shift/reduce or reduce/reduce conflicts, Yacc still produces a parser. It does this by selecting one of the valid steps wherever it has a choice. A rule describing which choice to make in a given situation is called a disambiguating rule.
Yacc invokes two disambiguating rules by default:

1. In a shift/reduce conflict, the default is to do the shift.
2. In a reduce/reduce conflict, the default is to reduce by the earlier grammar rule (in the input sequence).

Rule 1 implies that reductions are deferred whenever there is a choice, in favor of shifts.

Rule 2 gives the user rather crude control over the behavior of the parser in this situation, but reduce/reduce conflicts **should be avoided** whenever possible.

Conflicts may arise because of mistakes in input or logic, or because the grammar rules, while consistent, require a more complex parser than Yacc can construct. The use of actions within rules can also cause conflicts, if the action must be done before the parser can be sure which rule is being recognized. In these cases, the application of disambiguating rules is inappropriate, and leads to an incorrect parser. For this reason, Yacc always reports the number of shift/reduce and reduce/reduce conflicts resolved by Rule 1 and Rule 2.

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, most previous parser generators have considered conflicts to be fatal errors. Our experience has suggested that this rewriting is somewhat unnatural, and produces slower parsers; thus, Yacc will produce parsers even in the presence of conflicts.

---

# CODE

## Parser

The following code blocks contains the complete code for the parse which takens in a stream of tokens as input from the parser and has a set of grammar rules defined in it, using which it checks for the syntactical correctness of the code. It also generates the symbol table (containing identifiers) and constant table (containing numeric constants and string literals) from the input stream of tokens which is also printed as output along with the status of parsing i.e. successful or unsuccessful.

```
%token AUTO BREAK CASE CHAR CONST CONTINUE DEFAULT DO DOUBLE ELSE ENUM EXTERN FLOAT
FOR GOTO IF INT LONG REGISTER RETURN SHORT SIGNED SIZEOF STATIC STRUCT SWITCH
TYPEDEF UNION UNSIGNED VOID VOLATILE WHILE
%token IDENTIFIER CONSTANT STRING_LITERAL ASSIGN_OP REL_OP ADD_OP MUL_OP INCDEC_OP
EQU_OP  LAND LOR BAND BXOR BOR NOT_OP

%{
#include<stdio.h>
#include<stdlib.h>
#include<ctype.h>

extern FILE *yyin;
extern int lineno;
extern int comment_nesting;

int flag = 0;
int errorFlag=0;
struct stable{
      char name[100];
      char type[50];
};
struct ctable{
      char name[100];
      char type[50];
};
extern struct stable symbol_table[1000];
extern struct ctable constant_table[1000];
extern char yyval[100];
extern char yycons[100];
extern char yystr[100];
void insert_symbol_table()
{
    int k=hash_cal(yyval);
                                struct stable temp;
                                strcpy(temp.name,yyval);
                                strcpy(temp.type,"identifier");
                                int i=0;
```

```
                                 int flag=0;
                                 for(i=0;i<1000;i++)
                                 {
                                        if(symbol_table[k].name[0]=='\0')
                                        {
                                              break;
                                        }
                                        else
                                        {

if(strcmp(yyval,symbol_table[k].name)==0)
                                              {
                                                     flag=1;
                                                     break;
                                              }
                                              k=(k+1)%1000;
                                        }
                                 }
                                 if(flag==0)
                                 {
                                        symbol_table[k]=temp;
                                 }
}

void insert_constant_table()
{
                int k=hash_cal(yycons);
                                 struct ctable temp;
                                 strcpy(temp.name,yycons);
                                 strcpy(temp.type,"numeric constant");
                                 int i=0;
                                 int flag=0;
                                 for(i=0;i<1000;i++)
                                 {
                                        if(constant_table[k].name[0]=='\0')
                                        {
                                              break;
                                        }
                                        else
                                        {

if(strcmp(yycons,constant_table[k].name)==0)
                                              {
                                                     flag=1;
                                                     break;
                                              }
```

```
                                                        k=(k+1)%1000;
                                                }
                                        }
                                        if(flag==0)
                                        {
                                                constant_table[k]=temp;
                                        }
}
void insert_constant_table_str()
{
    int k=hash_cal(yystr);
                                        struct ctable temp;
                                        strcpy(temp.name,yystr);
                                        strcpy(temp.type,"string constant");
                                        int i=0;
                                        int flag=0;
                                        for(i=0;i<1000;i++)
                                        {
                                                if(constant_table[k].name[0]=='\0')
                                                {
                                                        break;
                                                }
                                                else
                                                {

if(strcmp(yystr,constant_table[k].name)==0)
                                                        {
                                                                flag=1;
                                                                break;
                                                        }
                                                        k=(k+1)%1000;
                                                }
                                        }
                                        if(flag==0)
                                        {
                                                constant_table[k]=temp;
                                        }
}

%}

%nonassoc LOWER_THAN_ELSE
%nonassoc ELSE

%right ASSIGN_OP '='
%left LOR
```

```
%left LAND
%left BOR
%left BXOR
%left BAND
%left EQU_OP
%left REL_OP
%left ADD_OP
%left MUL_OP
%right NOT_OP
%left INCDEC_OP

%start PROGRAM

%%

PROGRAM
    :EXTERNAL_DECLARATION
    |PROGRAM EXTERNAL_DECLARATION
    ;

EXTERNAL_DECLARATION
    :FUNCTION_DEFINITION
    |DECLARATION
    ;

DECLARATION
    : DECLARATION_SPECIFIER INIT_DECLARATOR_LIST ';'
    ;

DECLARATION_SPECIFIER
    : STORAGE_CLASS_SPECIFIER
    | STORAGE_CLASS_SPECIFIER DECLARATION_SPECIFIER
    | TYPE_SPECIFIER
    | TYPE_SPECIFIER DECLARATION_SPECIFIER
    ;

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

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

INIT_DECLARATOR_LIST
    : INIT_DECLARATOR
    | INIT_DECLARATOR_LIST ',' INIT_DECLARATOR
    ;

INIT_DECLARATOR
    : DECLARATOR
    | DECLARATOR '=' EXPRESSION
    | DECLARATOR '=' STRING_LITERAL {insert_constant_table_str();}
    ;

DECLARATOR
    : IDENTIFIER {insert_symbol_table();}
    | '(' DECLARATOR ')'
    | DECLARATOR '[' CONSTANT ']' {insert_constant_table();}
    | DECLARATOR '[' ']'
    | FUNCTION_DECLARATION
    ;

FUNCTION_DECLARATION
    : IDENTIFIER '(' DECLARATION_SPECIFIER_LIST ')'  {insert_symbol_table();}
    | IDENTIFIER '(' ')' {insert_symbol_table();}
    ;

FUNCTION_DEFINITION
    : DECLARATION_SPECIFIER IDENTIFIER '('  ')' COMPOUND_STATEMENT
{insert_symbol_table();}
    | DECLARATION_SPECIFIER IDENTIFIER '(' DEFINITION_SPECIFIER_LIST ')'
COMPOUND_STATEMENT {insert_symbol_table();}
    ;


DECLARATION_SPECIFIER_LIST
    : DECLARATION_SPECIFIER_LIST ',' DECLARATION_SPECIFIER
    | DECLARATION_SPECIFIER
    ;
```

```
DEFINITION_SPECIFIER_LIST
    :VARIABLE_DECLARATION
    | DEFINITION_SPECIFIER_LIST ',' VARIABLE_DECLARATION
    ;
VARIABLE_DECLARATION
    :DECLARATION_SPECIFIER IDENTIFIER {insert_symbol_table();}
    ;
COMPOUND_STATEMENT
    :'{' STATEMENT_LIST '}'
    |'{''}'
    ;

STATEMENT_LIST
    :STATEMENT_LIST STATEMENT
    |STATEMENT
    ;



STATEMENT
    : LABELED_STATEMENT
    | COMPOUND_STATEMENT
    | EXPRESSION_STATEMENT
    | ITERATION_STATEMENT
    | SELECTION_STATEMENT
    | JUMP_STATEMENT
    | DECLARATION
    | error ';'
    ;

LABELED_STATEMENT
    : CASE EXPRESSION ':' STATEMENT
    | DEFAULT ':' STATEMENT
    ;

EXPRESSION_STATEMENT
    : ';'
    | EXPRESSION ';'
    ;

EXPRESSION
    : IDENTIFIER ASSIGN_OP EXPRESSION {insert_symbol_table();}
    | IDENTIFIER '=' EXPRESSION {insert_symbol_table();}
    | EXPRESSION EQU_OP EXPRESSION
    | EXPRESSION REL_OP EXPRESSION
    | EXPRESSION ADD_OP EXPRESSION
    | EXPRESSION MUL_OP EXPRESSION
    | EXPRESSION LAND EXPRESSION
    | EXPRESSION LOR EXPRESSION
    | EXPRESSION BOR EXPRESSION
    | EXPRESSION BAND EXPRESSION
    | EXPRESSION BXOR EXPRESSION
    | NOT_OP EXPRESSION
    | EXPRESSION INCDEC_OP
    | INCDEC_OP EXPRESSION
    | '(' EXPRESSION ')'
    | IDENTIFIER {insert_symbol_table();}
    | CONSTANT {insert_constant_table();}
    | FUNCTION_CALL
    ;

FUNCTION_CALL
    : IDENTIFIER '(' ')' {insert_symbol_table();}
    | IDENTIFIER '(' EXPRESSION_LIST ')' {insert_symbol_table();}
    ;

EXPRESSION_LIST
    : EXPRESSION_LIST ',' EXPRESSION
    | EXPRESSION
    ;



SELECTION_STATEMENT
    : IF '(' EXPRESSION ')' STATEMENT  %prec LOWER_THAN_ELSE {printf("if\n");}
    | IF '(' EXPRESSION ')' STATEMENT ELSE STATEMENT {printf("ie\n");}
    | SWITCH '(' EXPRESSION ')' 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
    ;

%%

int yyerror()
{
    flag = 1;
    printf("PARSING ERROR at Line Number - %d\n",lineno);
    return (1);
}

main()
{

    yyin=fopen("test.txt","r");



    yyparse();

    if(comment_nesting!=0)
        printf("LEXICAL ERROR : Unterminated Comment\n");

    printf("\n*****Symbol Table******\n\n");
    int i;
      for(i=0;i<1000;i++)
      {
             if(symbol_table[i].name[0]!='\0')
             {
                    printf("%s   %s\n",symbol_table[i].name,symbol_table[i].type);
             }
      }


      printf("\n*****Constant Table******\n\n");
      for(i=0;i<1000;i++)
      {
             if(constant_table[i].name[0]!='\0')
             {
                    printf("%s
%s\n",constant_table[i].name,constant_table[i].type);
             }
      }
    if(!flag)
    {
        printf("\nParsing SUCCESSFUL.\n");
    }
}
```

## Scanner

The following code block contains the code for the lexical analyser which is used to recognize tokens and to send the recognized tokens to the above mentioned parser.

```
%{
#include "y.tab.h"
#include<string.h>
#include<stdio.h>

int comment_nesting = 0;
int lineno=1;

struct stable{
    char name[100];
    char type[50];
}symbol_table[1000];

struct ctable{
    char name[100];
    char type[50];
}constant_table[1000];
char yyval[100];
char yycons[100];
char yystr[100];
int hash_cal(char * str)
{
    int i;
    int sum=0;
    for(i=0;i<100;i++)
    {
       if(str[i]=='\0')
       {
             break;
       }
       else{
             sum+=(int)str[i];
             sum=sum%1000;
```

```
        }
    }
    return sum;
}

%}

IDENTIFIER [A-Za-z_]([A-Za-z_]|[0-9])*
CONSTANT (([0-9])*)(\.(([0-9])+))?(([eE])([\+\-]?)(([0-9])+))?
WHITESPACE " "|"    "|"\n"

%x SC_COMMENT

%%

"/*"         { comment_nesting++; BEGIN(SC_COMMENT); }
<SC_COMMENT>{
  "/*"       { comment_nesting++; }
  "*"+"/"    { comment_nesting=0;
                     BEGIN(INITIAL);  }
  "*"+       ;
  [^/*\n]+   ;
  [/]        ;
  \n         { lineno++;}
}

\/\/[^\n]*[\n]? { lineno++; }

"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); }

{IDENTIFIER}     {    //printf("%s - identifier\n",yytext);
                           strcpy(yyval,yytext);
                    return (IDENTIFIER);
                    }

{CONSTANT} {//printf("%s - constant\n",yytext);
                           strcpy(yycons,yytext);
                           return (CONSTANT);
              }

\"[^\n]*\" {//printf("%s - string\n",yytext);

                    strcpy(yystr,yytext);

                    return(STRING_LITERAL);
              }

"+="|"-="|"*="|"/="|"%="    { return(ASSIGN_OP); }
"=="|"!="                               { return(EQU_OP); }
"<="|">="|">"|"<"                       { return(REL_OP); }
"+"|"-"                                        { return(ADD_OP); }
"*"|"/"|"%"                                    { return(MUL_OP); }
"++"|"--"                               { return(INCDEC_OP); }
"&&"         {return(LAND);}
"||"         {return(LOR);}
"&"          {return(BAND);}
"|"          {return(BOR);}
"^"          {return(BXOR);}
"!"|"~"      {return(NOT_OP);}
"="           { return('='); }
"{"           { return('{'); }
"}"           { return('}'); }
"("           { return('('); }
")"           { return(')'); }
"["           { return('['); }
"]"           { return(']'); }
";"           { return(';'); }
","           { return(','); }
"."           { return('.'); }
"?"           { return('?'); }
":"           { return(':'); }

{WHITESPACE}    { if(yytext[0]=='\n') { lineno++; } };

. {printf("LEXICAL ERROR - %s : invalid character at line [%d]\n",yytext,lineno);}

%%



int yywrap()
{
    return 1;
}
```

---

## **TEST CASES**

The following is a list of all the test cases for the parser. The format of the test cases is - input file first and then the output from the terminal. Also, a little description is given above each test case.

1. **Main Test Case**
   This test case contains all the possible inputs from various declarations, loops such as for, while, do-while, expressions, else-if ladder, function definition, function declaration, function call, string constants, etc.
   The output expected from this test cases is “successfully parsed”.

<u>INPUT</u>

```
int fun(int, int);
int fun(int a,int b)
{
      for(i=0;i<10;i++)
      {
      if(a%2==0)
      {
            a++;
      }
      else if(b%2==0)
      {
            b++;
      }
      else
      {
            a--;
            b--;
      }
      }
      int z= a*b;
      return z;
}

int main()
{
      int a,b,c;
      int c = fun(a,b);
      int x=100;
      while(x>=0)
      {
      c/=2;
      do
      {
            switch(c)
            {
                  default : c=50;
                  case 1 : c++;
                        break;
                  case 2 : c+=2;
                        break;

            }
      }
      while(c>0);
      }
      int d;
      d = (~(5+3*2>=7!=1))&&6;

}
```

OUTPUT

```
*****Symbol Table******

a    identifier
b    identifier
c    identifier
d    identifier
i    identifier
x    identifier
z    identifier
fun    identifier

*****Constant Table******

0    numeric constant
1    numeric constant
2    numeric constant
3    numeric constant
5    numeric constant
6    numeric constant
7    numeric constant
```

```
10    numeric constant
50    numeric constant
100    numeric constant

Parsing SUCCESSFUL.
```

2. **Semicolon Missing**
   This test case contains a simple variable declaration with a semicolon missing at the end which should result in a parsing error.

INPUT

```
int main()
{
      int a
      a=1;
}
```

OUTPUT

```
PARSING ERROR at Line Number - 4

*****Symbol Table******

a    identifier

*****Constant Table******
```

3. **Unmatched Braces**
   This test case contains an expression which has 2 opening braces and 3 closing braces and thus should return a parsing error.

INPUT

```
int main()
{
      int a;
      a=(3*(1+5)-2));
}
```

OUTPUT

```
PARSING ERROR at Line Number - 4

*****Symbol Table******

a    identifier

*****Constant Table******

1    numeric constant
2    numeric constant
3    numeric constant
5    numeric constant
```

4. **For Loop with just one semicolon**
   This test case contains a for loop where there is just one semicolon in the for() and thus the parser is expected to return an error.

INPUT

```
int main()
{
      int a,b;
      for(i=0;i++)
      {
      a++;
      }
      return 0;
}
```

OUTPUT

```
PARSING ERROR at Line Number - 4

*****Symbol Table******

a    identifier
b    identifier
i    identifier

*****Constant Table******
```

```
0    numeric constant
```

5. **Incorrect Expression in While**
   This test case contains a while loop where an incorrect expression is supplied as the argument and thus the parser should return an error for this test case.

INPUT

```
int main()
{
      int a,b;
      while(a<)
      {
      b--;
      }
}
```

OUTPUT

```
PARSING ERROR at Line Number - 4

*****Symbol Table******

a    identifier
b    identifier

*****Constant Table******
```

6. **Do-While Looping Construct with Missing Argument**
   This test case has a do-while loop where no argument is passed in the while() and thus the parser is expected to return an error.

INPUT

```
int main()
{
      int a,b;
      do
      {
      b--;
      }
      while();
}
```

OUTPUT

```
PARSING ERROR at Line Number - 8

*****Symbol Table******

a    identifier
b    identifier

*****Constant Table******
```

7. **Nested Looping**
   This test case contains a while loop within a for loop where the code is syntactically correct. Thus, the parser is expected to parse successfully.

INPUT

```
int main()
{
	int a,b;
	for(i=0;;i++)
	{
	while(a>0)
	{
		b++;
	}
	}
}
```

OUTPUT

```
*****Symbol Table******

a    identifier
b    identifier
i    identifier

*****Constant Table******

0    numeric constant

Parsing SUCCESSFUL.
```

8. **Dangling If-Else**
   This test case has a if else statement along with a if statement which is parsed successfully. The main check here is about if the dangling if-else problem is solved or not, which is solved and can be checked by printing some constants at the parsing of each rule.

INPUT

```
int main()
{
      int a,b;
      a=1;
      b=3;

      if(a==1)
      if(b>5)
      b++;
      else
      b--;

}
```

OUTPUT

```
*****Symbol Table******

a    identifier
b    identifier

*****Constant Table******

1    numeric constant
3    numeric constant
5    numeric constant

Parsing SUCCESSFUL.
```

9. **Function Call with incorrect parameters**
   This test case contains a function declaration and the main function as a function

call to that function wherein one parameter is missing and empty space is sent instead. The parser is expected to report an error in this case.

INPUT

```
int fun(int a, int b)
{
      return a+b;
}

int main()
{
      int a,b;
      int c = fun(a,);
      return 0;
}
```

OUTPUT

```
PARSING ERROR at Line Number - 9

*****Symbol Table******

a    identifier
b    identifier
c    identifier

*****Constant Table******

0    numeric constant
```

10. **Lexical and Syntax Errors Mixed**
This test case contains an invalid character in definition of a variable. This is expected to return a Lexical Error as well as a Parser Error as the character is not matched to any token definition rule and also not to any grammar rule.

INPUT

```
int main()
{
      int a,b;
      a=$;
      b=(3*(1+5)-2));
```

```
}
```

OUTPUT

```
LEXICAL ERROR - $ : invalid character at line [4]
PARSING ERROR at Line Number - 4
PARSING ERROR at Line Number - 5

*****Symbol Table******

a    identifier
b    identifier

*****Constant Table******

1    numeric constant
2    numeric constant
3    numeric constant
5    numeric constant
```

11. **Multiple Parsing Errors**
    This test case contains 3 possible parsing errors - a missing semicolon in declaration, unmatched braces in an expression and no argument in the while() call. Thus it should return 3 errors at the expected lines.

INPUT

```
int main()
{
	int a,b
	a=1;

	b=(3*(1+5)-2));

	int c = a+b;

	while()
	{
	c++;
	}
}
```

OUTPUT

```
PARSING ERROR at Line Number - 4
PARSING ERROR at Line Number - 6
PARSING ERROR at Line Number - 10

*****Symbol Table******

a    identifier
b    identifier
c    identifier

*****Constant Table******

1    numeric constant
2    numeric constant
3    numeric constant
5    numeric constant
```

## CONCLUSION

We studied Context Free Grammars and applied the rules to make a working parser which checks the syntactical correctness of the program and the supports basic declarations and definitions, expressions, function declaration, definition and function call, looping constructs (for, while, do-while) and conditional statements (if-else ladder, switch-case). Thus, the parsing unit under the analysis phase of compiler design is studied and implemented.

## THE END