VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT
on

Compiler Design

Submitted by

ADVITHI D (1BM21CS009)

in partial fulfillment for the award of the degree of
BACHELOR OF ENGINEERING
in
COMPUTER SCIENCE AND ENGINEERING

B.M.S. COLLEGE OF ENGINEERING
(Autonomous Institution under VTU)
BENGALURU-560019
Nov-2023 to Feb-2024

B. M. S. College of Engineering,

Bull Temple Road, Bangalore 560019

(Affiliated To Visvesvaraya Technological University, Belgaum)

Department of Computer Science and Engineering

CERTIFICATE

This is to certify that the Lab work entitled "**Compiler Design Lab**" carried out by **Advithi D (1BM21CS009),** who is a bonafide student of **B.M.S. College of Engineering.** It is in partial fulfillment for the award of **Bachelor of Engineering in Computer Science and Engineering** of the Visvesvaraya Technological University, Belgaum during the academic semester June-2023 to Sep-2023. The Lab report has been approved as it satisfies the academic requirements in respect of a **Compiler Design(22CS5PCCPD)** work prescribed for the said degree.

Lohith J J
Assistant Professor
Department of CSE
BMSCE, Bengaluru

Dr. Jyothi S Nayak
Professor and Head
Department of CSE
BMSCE, Bengaluru

INDEX

Part-A: Implementation of Lexical Analyzer, By using C/C++/Java/Python language and using LEX tool.

Part-B: Part-B: Implementation of Parsers (Syntax Analyzers) Using C/C++/Java/Python language)

Part-C: Syntax Directed Translation using YACC tool

Part-A: Implementation of Lexical Analyzer, By using C/C++/Java/Python language and using LEX tool.

1. Write a program to design Lexical Analyzer in C/C++/Java/Python Language (to recognize any five keywords, identifiers, numbers, operators and punctuations)

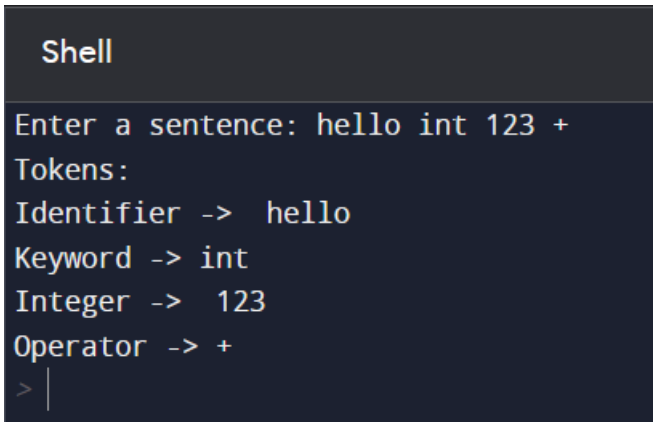
```
import re
def is_operator(char):
    return char in ['+', '-', '*', '/', '>', '<', '=']


def is_valid_identifier(token):
    return token[0].isalpha() and not token.isdigit()
def get_keywords():
    return ["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"]
def is_integer(token):
    try:
        int(token)
        return True
    except ValueError:
        return False
def lexical_analyzer(input_str):
    tokens = re.findall(r'[a-zA-Z_]\w*|[-+*/<>=]|[(),;]|[0-9]+', input_str)
    print('Tokens: ')
    for token in tokens:
        if token in ['+', '-', '*', '/', '>', '<', '=']:
            print(f"Operator -> {token}")
        elif token in [',', ';', '(', ')']:
            print(f"Delimiter -> {token}")
```

```
        elif token in get_keywords():
            print(f"Keyword -> {token}")
        elif is_integer(token):
            print(f"Integer ->  {token}")
        elif is_valid_identifier(token):
            print(f"Identifier ->  {token}")
        else:
            print(f"Unidentified -> {token}")

if __name__ == "__main__":
    input_string = input("Enter a C program code: ")
    lexical_analyzer(input_string)
```

OUTPUT:



2. Write a program in LEX to recognize Floating Point Numbers.

```
%{
#include<stdio.h>
int cnt=0;
%}
sign [+|-]
num [0-9]
dot [.]
%%
{sign}?{num}*{dot}{num}* {printf("Floating point no.");cnt=1;}
{sign}?{num}* {printf("Not Floating point no.");cnt=1;}
%%
int yywrap()
{
}
int main()
{
yylex();
if(cnt==0){
printf("Not floating pnt no.");
}
return 0;
}
```

OUTPUT:

3. Write a program in LEX to recognize different tokens: Keywords, Identifiers, Constants, Operators and Punctuation symbols.

```
%{
#include<stdio.h>
int cnt=0;
%}
letter [a-zA-Z]
digit [0-9]
punc [!|,|.]
oper [+|*|-|/|%]
boole [true|false]
%%
{digit}+|{digit}*.{digit}+ {printf("Constants");}
int|float {printf("Keyword");}
{letter}({digit}|{letter})* {printf("Identifiers");}
{oper} {printf("Operator");}
{punc} {printf("Punctuator");}
%%
int yywrap()
{
}
int main()
{
yylex();
return 0;
}
```

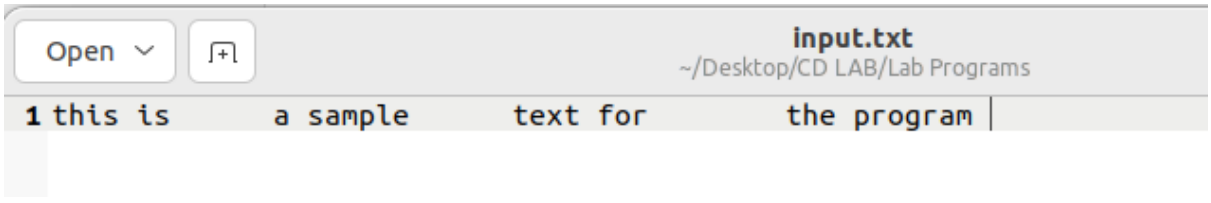
OUTPUT:

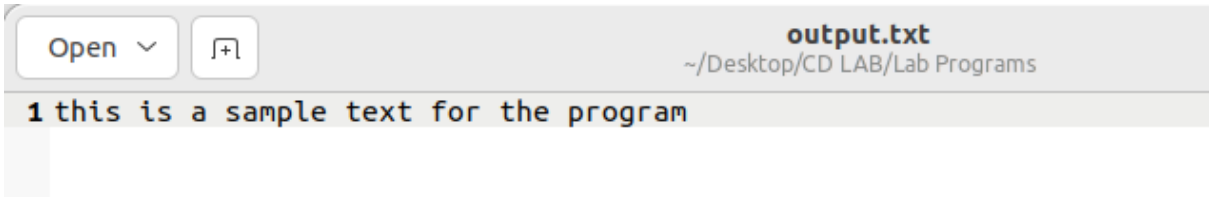
```
Enter the sentence: int
Keyword
abc
Identifiers
+
Operator
!
Punctuator
123
Constants
```

4. **Write a LEX program that copies a file, replacing each nonempty sequence of white spaces by a single blank.**

```
%{
#include<stdio.h>
%}
%%
[\t" "]+ fprintf(yyout," ");
.|\n fprintf(yyout,"%s",yytext);
%%
int yywrap()
{
return 1;
}
int main(void)
{
yyin=fopen("input1.txt","r");
yyout=fopen("output.txt","w");
yylex();
return 0;
}
```

OUTPUT:





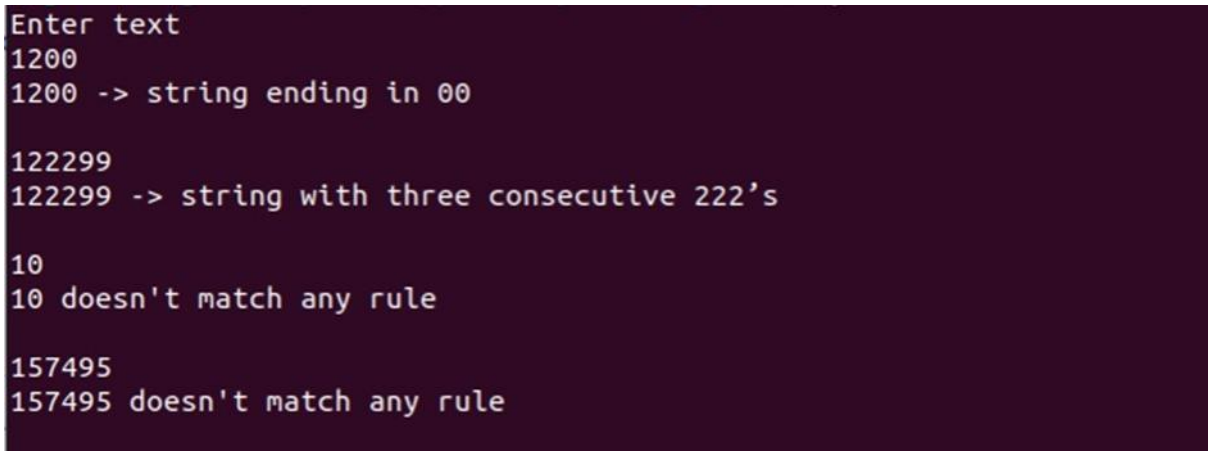
5. **Write a LEX program to recognize the following tokens over the alphabets {0,1,..,9}**
   a) **The set of all string ending in 00.**
   b) **The set of all strings with three consecutive 222's.**
   c) **The set of all string such that every block of five consecutive symbols contains at least two 5's.**
   d) **The set of all strings beginning with a 1 which, interpreted as the binary representation of an integer, is congruent to zero modulo 5.**
   e) **The set of all strings such that the 10th symbol from the right end is 1.**
   f) **The set of all four digits numbers whose sum is 9**
   g) **The set of all four digital numbers, whose individual digits are in ascending order from left to right.**

```
d[0-9]
%{
/* d is for recognising digits */
int c1=0,c2=0,c3=0,c4=0,c5=0,c6=0,c7=0;
/* c1 to c7 are counters for rules a1 to a7 */
%}
%%
({d})*00 { c1++; printf("%s -> string ending in 00\n",yytext);}
({d})*222({d})* { c2++; printf("%s -> string with three consecutive 222's \n",yytext);}
(1(0)*(11|01)(01*01|00*10(0)*(11|1))*0)(1|10(0)*(11|01)(01*01|00*10(0)*(11|1))*10)* {
c4++;
printf("%s -> string beginning with a 1 which, interpreted as the binary representation of an integer, is congruent to zero modulo 5 \n",yytext);
}
({d})*1{d}{9} {
c5++; printf("%s -> string such that the 10th symbol from the right end is 1 \n",yytext);
}
({d})* {
int i,c=0;
if(yyleng<5)
{
printf("%s doesn't match any rule\n",yytext);
```

```
}
else
{
for(i=0;i<5;i++) { if(yytext[i]=='5') {
c++; } }
if(c>=2)
{
for(;i<yyleng;i++)
{
if(yytext[i-5]=='5') {
c--; }
if(yytext[i]=='5') { c++;
}
if(c<2) { printf("%s doesn't match any rule\n",yytext);
break; }
}
if(yyleng==i)
{
printf("%s -> string such that every block of five consecutive symbols contains at least two 5’s\n",yytext); c3++; }
}
else
{
printf("%s doesn't match any rule\n",yytext);
}
}
}
%%
int yywrap()
{ }
int main()
```

```
{
printf("Enter text\n");
yylex();
printf("Total number of tokens matching rules are : \n");
printf("Rule A : %d \n",c1);
printf("Rule B : %d \n",c2);
printf("Rule C : %d \n",c3);
printf("Rule D : %d \n",c4);
printf("Rule E : %d \n",c5);
return 0;
}
```

OUTPUT:



Part-B: Part-B: Implementation of Parsers (Syntax Analyzers) Using C/C++/Java/Python language)

1. Write a program to implement

(a) Recursive Descent Parsing with back tracking (Brute Force Method). S→ cAd , A →ab /a

```
#include<stdio.h>
#include<string.h>
int S();
int A();
char input[100];
int currentIndex = 0;
int match(char symbol) {
   if (input[currentIndex] == symbol) {
      currentIndex++;
      return 1;
   } else {
      return 0;
   }
}
int S() {
   if (match('c')) {
      if (A()) {
         if (match('d')) {
            return 1;
         }
      }
   }
   return 0;
}
int A() {
```

```
    int tempIndex = currentIndex;
    if (match('a')) {
        if (match('b')) {
            return 1;
        }
    }
    currentIndex = tempIndex;
    if (match('a')) {
        return 1;
    }
    return 0;
}
int main() {
    printf("Enter the input string: ");
    scanf("%s", input);
    currentIndex = 0;
    if (S() && currentIndex == strlen(input)) {
        printf("Parsing successful! Input belongs to the given grammar.\n");
    } else {
        printf("Parsing failed! Input does not belong to the given grammar.\n");
    }
    return 0;
}
```

OUTPUT:

```
Output

/tmp/R63NgA7pEx.o
Enter the input string: cad
Parsing successful! Input belongs to the given grammar.
```

```
Output

/tmp/R63NgA7pEx.o
Enter the input string: cabd
Parsing successful! Input belongs to the given grammar.
```

```
Output

/tmp/R63NgA7pEx.o
Enter the input string: caab
Parsing failed! Input does not belong to the given grammar.
```

(b) Recursive Descent Parsing with back tracking (Brute Force Method). S→ cAd , A → a / ab

```
#include<stdio.h>
#include<string.h>
int S();
int A();
char input[100];
int currentIndex = 0;
int match(char symbol) {
   if (input[currentIndex] == symbol) {
      currentIndex++;
      return 1;
   } else {
      return 0;
   }
}
int S() {
   if (match('c')) {
      if (A()) {
         if (match('d')) {
            return 1;
         }
      }
   }
   return 0;
}
int A() {
   int tempIndex = currentIndex;
   if (match('a')) {
      return 1;
   }
```

```
    currentIndex = tempIndex;
    if (match('a')) {
        if (match('b')) {
            return 1;
        }
    }
    currentIndex = tempIndex;
    return 0;
}
int main() {
    printf("Enter the input string: ");
    scanf("%s", input);
    currentIndex = 0;
    if (S() && currentIndex == strlen(input)) {
        printf("Parsing successful! Input belongs to the given grammar.\n");
    } else {
        printf("Parsing failed! Input does not belong to the given grammar.\n");
    }
    return 0;
}
```

OUTPUT:

```
Output
/tmp/R63NgA7pEx.o
Enter the input string: cad
Parsing successful! Input belongs to the given grammar.
```

```
Output
/tmp/R63NgA7pEx.o
Enter the input string: cabd
Parsing failed! Input does not belong to the given grammar.
```

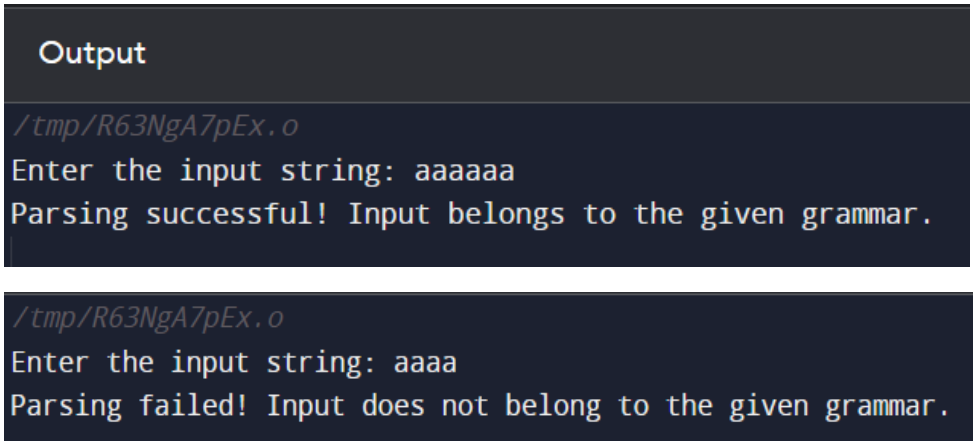
2. Write a program to implement: Recursive Descent Parsing with back tracking (Brute Force Method).

(a) S→ aaSaa | aa

```
#include <stdio.h>
#include <string.h>
int S();
char input[100];
int currentIndex = 0;
int match(char symbol) {
   if (input[currentIndex] == symbol) {
      currentIndex++;
      return 1;
   } else {
      return 0;
   }
}
int S() {
   int tempIndex = currentIndex;
   if (match('a') && match('a')) {
      if (S() && match('a') && match('a')) {
         return 1;
      }
   }
   currentIndex = tempIndex;
   if (match('a') && match('a')) {
      return 1;
   }
   return 0;
}
int main() {
   printf("Enter the input string: ");
```

```
    scanf("%s", input);
    currentIndex = 0;
    if (S() && currentIndex == strlen(input)) {
        printf("Parsing successful! Input belongs to the given grammar.\n");
    } else {
        printf("Parsing failed! Input does not belong to the given grammar.\n");
    }
    return 0;
}
```

OUTPUT:

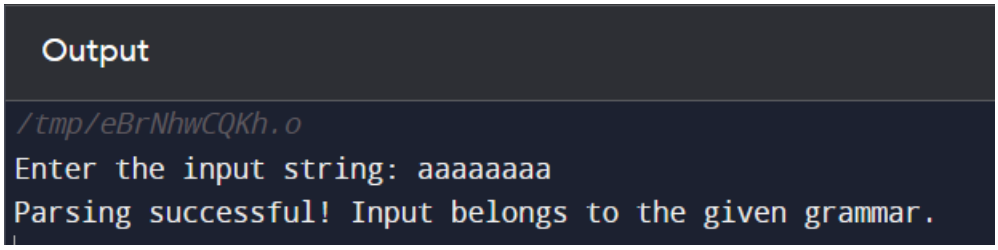


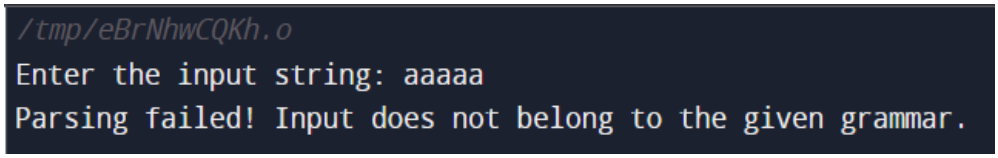
(b) S → aaaSaaa | aa

```
#include <stdio.h>
#include <string.h>
int S();
char input[100];
int currentIndex = 0;
int match(char symbol) {
    if (input[currentIndex] == symbol) {
        currentIndex++;
        return 1;
    } else {
        return 0;
    }
}
int S() {
    int tempIndex = currentIndex;
    if (match('a') && match('a') && match('a')) {
        if (S() && match('a') && match('a') && match('a')) {
            return 1;
        }
    }
    currentIndex = tempIndex;
    if (match('a') && match('a')) {
        return 1;
    }
    return 0;
}
int main() {
    printf("Enter the input string: ");
    scanf("%s", input);
    currentIndex = 0;
```

```
    if (S() && currentIndex == strlen(input)) {
        printf("Parsing successful! Input belongs to the given grammar.\n");
    } else {
        printf("Parsing failed! Input does not belong to the given grammar.\n");
    }
    return 0;
}
```

OUTPUT:



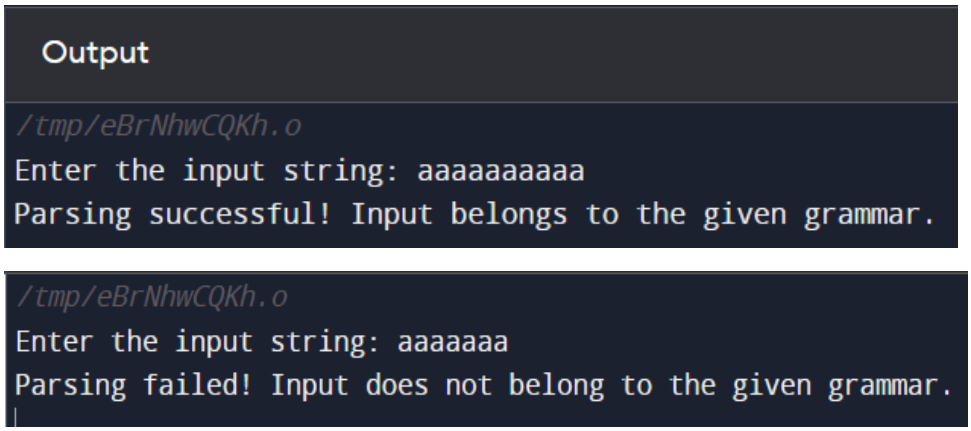


(c) S → aaaaSaaaa | aa

```
#include <stdio.h>
#include <string.h>
int S();
char input[100];
int currentIndex = 0;
int match(char symbol) {
    if (input[currentIndex] == symbol) {
        currentIndex++;
        return 1;
    } else {
        return 0;
    }
}
int S() {
    int tempIndex = currentIndex;
    if (match('a') && match('a') && match('a') && match('a')) {
        if (S() && match('a') && match('a') && match('a') && match('a')) {
            return 1;
        }
    }
    currentIndex = tempIndex;
    if (match('a') && match('a')) {
        return 1;
    }
    return 0;
}
int main() {
    printf("Enter the input string: ");
    scanf("%s", input);
    currentIndex = 0;
```

```
    if (S() && currentIndex == strlen(input)) {
        printf("Parsing successful! Input belongs to the given grammar.\n");
    } else {
        printf("Parsing failed! Input does not belong to the given grammar.\n");
    }
    return 0;
}
```

OUTPUT:

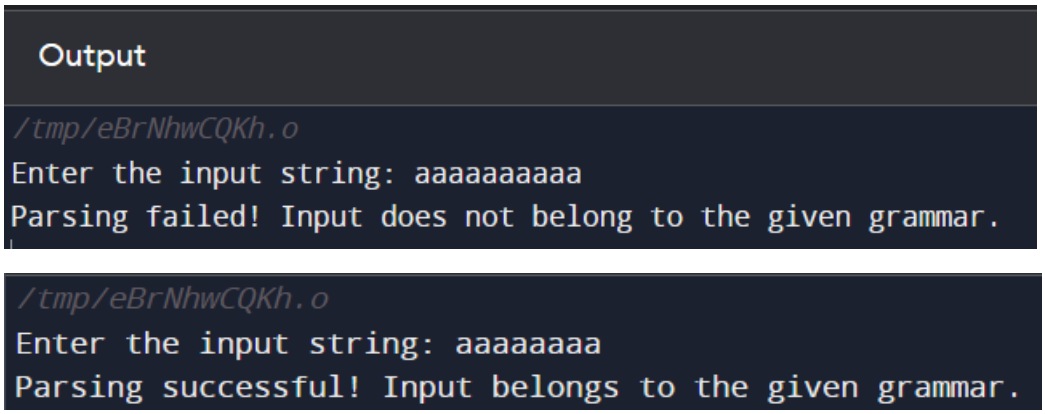


(d) S → aaaSaaa |aSa | aa

```
#include <stdio.h>
#include <string.h>
int S();
char input[100];
int currentIndex = 0;
int match(char symbol) {
    if (input[currentIndex] == symbol) {
        currentIndex++;
        return 1;
    } else {
        return 0;
    }
}
int S() {
    int tempIndex = currentIndex;
    if (match('a') && match('a') && match('a')) {
        if (S() && match('a') && match('a') && match('a')) {
            return 1;
        }
    }
    currentIndex = tempIndex;
    if (match('a') && S() && match('a')) {
        return 1;
    }
    currentIndex = tempIndex;
    if (match('a') && match('a')) {
        return 1;
    }
    return 0;
}
```

```
int main() {
    printf("Enter the input string: ");
    scanf("%s", input);
    currentIndex = 0;
    if (S() && currentIndex == strlen(input)) {
        printf("Parsing successful! Input belongs to the given grammar.\n");
    } else {
        printf("Parsing failed! Input does not belong to the given grammar.\n");
    }
    return 0;
}
```

OUTPUT:



Part-C: Syntax Directed Translation using YACC tool

1. Write a program to design LALR parsing using YACC.

Lex:

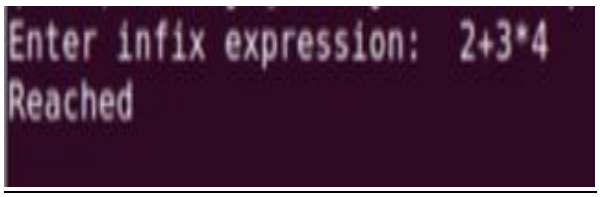
```
%{
 #include "y.tab.h"
 extern int yylval;
%}
%%
//If the token is an Integer number,then return it's value.
[0-9]+ {yylval=atoi(yytext); return digit;}
//If the token is space or tab,then just ignore it.
[\t] ;
//If the token is new line,return 0.
[\n] return 0;
//For any other token, return the first character read since the last
match.
. return yytext[0];
%%
```

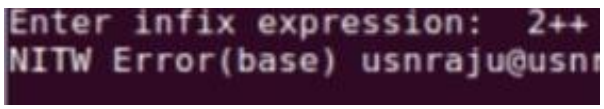
Yacc:

```
%{
#include <math.h>
#include<ctype.h>
#include<stdio.h>
int var_cnt=0;
char iden[20];
%}
%token id
%token digit
%%
S:id '=' E { printf("%s=t%d\n",iden,var_cnt-1); }
```

```
E:E '+' T { $$=var_cnt; var_cnt++; printf("t%d = t%d + t%d;\n", $$, $1, $3 ); }
|E '-' T { $$=var_cnt; var_cnt++; printf("t%d = t%d - t%d;\n", $$, $1, $3 ); }
|T { $$=$1; }
;
T:T '*' F { $$=var_cnt; var_cnt++; printf("t%d = t%d * t%d;\n", $$, $1, $3 ); } |T '/' F { $$=var_cnt; var_cnt++; printf("t%d = t%d / t%d;\n", $$, $1, $3 ); } |F {$$=$1 ; }
F:P '^' F { $$=var_cnt; var_cnt++; printf("t%d = t%d ^ t%d;\n", $$, $1, $3 );} | P { $$ = $1;}
;
P: '(' E ')' { $$=$2; }
|digit { $$=var_cnt; var_cnt++; printf("t%d = %d;\n",$$,$1); } ;
%%
int main()
{
var_cnt=0;
printf("Enter an expression : \n");
yyparse();
return 0;
}
yyerror()
{
printf("error");
}
```

OUTPUT:





2. Use YACC to Convert Binary to Decimal (including fractional numbers).

Lex:

```
%{
#include<stdio.h>
#include<stdlib.h>
#include"y.tab.h"
extern int yylval;
%}
/* rules
if 0 is matched ,make yylval to 0 and return ZERO which is
variable in Yacc program
if 1 is matched ,make yylval to 1 and return ONE which is
variable in Yacc program
if . is matched ,return POINT which is variable in Yacc program
if line change , return 0
otherwise ,ignore*/
%%
0 {yylval=0;return ZERO;}
1 {yylval=1;return ONE;}
"." {return POINT;}
[ \t] {;}
\n return 0;
%%
```

Yacc:

```
/* definition section*/
%{
#include<stdio.h>
#include<stdlib.h>
#include<math.h>
//#define YYSTYPE double
void yyerror(char *s);
```

```
float x = 0;
%}
%token ZERO ONE POINT
%%
L: X POINT Y {printf("%f",$1+x);}
| X {printf("%d", $$);}
X: X B {$$=$1*2+$2;}
| B {$$=$1;}
Y: B Y {x=$1*0.5+x*0.5;}
| {;}
B:ZERO {$$=$1;}
|ONE {$$=$1;};
%%
int main()
{
printf("Enter the binary number : ");
while(yyparse());
printf("\n");
}
void yyerror(char *s)
{
fprintf(stdout,"\n%s",s);
}
```

OUTPUT:

```
Enter the binary number : 10110.1100
22.750000
Enter the binary number : 101101100
364
Enter the binary number : 111.011
7.375000
```

3. Use YACC to implement, evaluator for arithmetic expressions (Desktop calculator)

Lex:

```
%{
#include<stdio.h>
#include "y.tab.h"
extern int yylval;
%}
%%
[0-9]+ {
yylval=atoi(yytext);
 return NUMBER;
}
[\t] ;
[\n] return 0;
. return yytext[0];
%%
int yywrap()
{
return 1;  }
```

Yacc:

```
%{
#include<stdio.h>
int flag=0;
%}
%token NUMBER
%left '+' '-'
%left '*' '/' '%'
%left '(' ')'
%%
ArithmeticExpression: E{
```

```
                printf("\nResult=%d\n", $$);
                return 0;
                };
E:E'+'E {$$=$1+$3;}
|E'-'E {$$=$1-$3;}
|E'*'E {$$=$1*$3;}
|E'/'E {$$=$1/$3;}
|E'%'E {$$=$1%$3;}
|'('E')' {$$=$2;}
| NUMBER {$$=$1;}
;
%%
void main()
{
printf("\nEnter Any Arithmetic Expression: \n");
yyparse();
if(flag==0)
printf("\nEntered arithmetic expression is Valid\n\n");
}
void yyerror()
{
printf("\nEntered arithmetic expression is Invalid\n\n");
flag=1;
}
```

OUTPUT:

```
Enter an expression
2^3^2
Digit : 2
Digit : 3
Digit : 2
Power Operation 3 ^ 2  : 9
Power Operation 2 ^ 9  : 512

Answer : 512
```

```
Enter an expression
2+3*4
Digit : 2
Digit : 3
Digit : 4
Multiplication Operation of 3 and 4 : 12
Addition Operation 2 and 12 : 14

Answer : 14
```

4. Use YACC to convert: Infix expression to Postfix expression.

Lex:

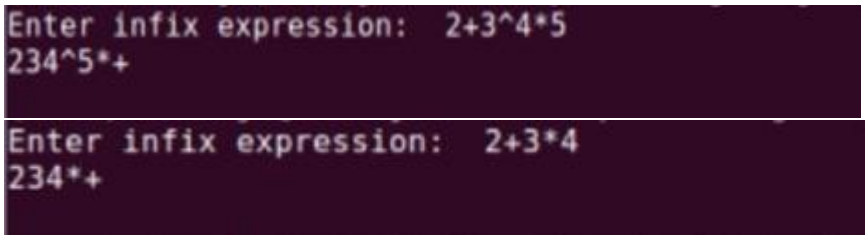
```
%{
#include "y.tab.h"
extern int yylval;
%}
%%
[0-9]+ { yylval=atoi(yytext); return digit;}
[\t] ;
[\n] return 0;
. return yytext[0];
%%
int yywrap()
{ }
```

Yacc:

```
%{
#include <ctype.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
%}
%token digit
%%
S: E { printf("\n\n"); }
  ;
E: E '+' T { printf("+"); }
  | E '-' T { printf("-"); }
  | T
  ;
T: T '*' F { printf("*"); }
  | T '/' F { printf("/"); }
```

```
  | F
  ;
F: F '^' G { printf("^"); }
  | G
  ;
G: '(' E ')'
  | digit { printf("%d", $1); }
  ;
%%
int main()
{
  printf("Enter infix expression: ");
  yyparse();
}
yyerror()
{
  printf("Error");
}
```

OUTPUT:



5. Use YACC to generate Syntax tree for a given expression

Lex:

```
%{
#include "y.tab.h"
extern int yylval;
%}
%%
[0-9]+ { yylval=atoi(yytext); return digit;}
[\t] ;
[\n] return 0;
. return yytext[0];
%%
int yywrap()
{
}
```

Yacc:

```
%{
#include <math.h>
#include<ctype.h>
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
struct tree_node
{
char val[10];
int lc;
int rc;
};
int ind;
struct tree_node syn_tree[100];
void my_print_tree(int cur_ind);
```

```
int mknode(int lc,int rc,char val[10]);
%}
%token digit
%%
S:E { my_print_tree($1); }
;
E:E'+'T { $$= mknode($1,$3,"+"); ; }
|T { $$=$1; }
;
T:T'*'F { $$= mknode($1,$3,"*"); ; }
|F {$$=$1 ; }
;
F:'('E')' { $$=$2; }
|digit {char buf[10]; sprintf(buf,"%d", yylval); $$ = mknode(-1,-1,buf);}
%%
int main()
{
ind=0;
printf("Enter an expression\n");
yyparse();
return 0;
}
int yyerror()
{
printf("NITW Error\n");
}
int mknode(int lc,int rc,char val[10])
{
strcpy(syn_tree[ind].val,val);
syn_tree[ind].lc = lc;
syn_tree[ind].rc = rc;
```

```
ind++;
return ind-1;
}
/*my_print_tree function to print the syntax tree in DLR fashion*/
void my_print_tree(int cur_ind)
{
if(cur_ind==-1) return;
if(syn_tree[cur_ind].lc==-1&&syn_tree[cur_ind].rc==-1)
printf("Digit Node -> Index : %d, Value : %s\n",cur_ind,syn_tree[cur_ind].val); else
printf("Operator Node -> Index : %d, Value : %s, Left Child Index : %d,Right Child Index :
%d  \n",cur_ind,syn_tree[cur_ind].val, syn_tree[cur_ind].lc,syn_tree[cur_ind].rc);
my_print_tree(syn_tree[cur_ind].lc);
my_print_tree(syn_tree[cur_ind].rc);
}
```

OUTPUT:

```
Enter an expression
2+3+(4*5)-6
Operator Node -> Index : 8, Value : -, Left Child Index : 6, Right Child Index : 7
Operator Node -> Index : 6, Value : +, Left Child Index : 2, Right Child Index : 5
Operator Node -> Index : 2, Value : +, Left Child Index : 0, Right Child Index : 1
Digit Node -> Index : 0, Value : 2
Digit Node -> Index : 1, Value : 3
Operator Node -> Index : 5, Value : *, Left Child Index : 3, Right Child Index : 4
Digit Node -> Index : 3, Value : 4
Digit Node -> Index : 4, Value : 5
Digit Node -> Index : 7, Value : 6
```

```
Enter an expression
2+3*4
Operator Node -> Index : 4, Value : +, Left Child Index : 0, Right Child Index : 3
Digit Node -> Index : 0, Value : 2
Operator Node -> Index : 3, Value : *, Left Child Index : 1, Right Child Index : 2
Digit Node -> Index : 1, Value : 3
Digit Node -> Index : 2, Value : 4
```

6. Use YACC to generate 3-Address code for a given expression

Lex:

```
d [0-9]+
a [a-zA-Z]+
%{
#include<stdio.h>
#include<stdlib.h>
#include"y.tab.h"
extern int yylval;
extern char iden[20];
%}
%%
{d} { yylval=atoi(yytext); return digit; }
{a} { strcpy(iden,yytext); yylval=1; return id;}
[ \t] {;}
\n return 0;
. return yytext[0];
%%
int yywrap()
{
}
```

Yacc:

```
%{
#include <math.h>
#include<ctype.h>
#include<stdio.h>
int var_cnt=0;
char iden[20];
%}
%token id
%token digit
```

```
%%
S:id '=' E { printf("%s=t%d\n",iden,var_cnt-1); }
E:E '+' T { $$=var_cnt; var_cnt++; printf("t%d = t%d + t%d;\n", $$, $1, $3 ); }
|E '-' T { $$=var_cnt; var_cnt++; printf("t%d = t%d - t%d;\n", $$, $1, $3 ); }
|T { $$=$1; }
;
T:T '*' F { $$=var_cnt; var_cnt++; printf("t%d = t%d * t%d;\n", $$, $1, $3 ); } |T '/' F { $$=var_cnt; var_cnt++; printf("t%d = t%d / t%d;\n", $$, $1, $3 ); } |F {$$=$1 ; }
F:P '^' F { $$=var_cnt; var_cnt++; printf("t%d = t%d ^ t%d;\n", $$, $1, $3 );} | P { $$ = $1;}
;
P: '(' E ')' { $$=$2; }
|digit { $$=var_cnt; var_cnt++; printf("t%d = %d;\n",$$,$1); } ;
%%
int main()
{
var_cnt=0;
printf("Enter an expression : \n");
yyparse();
return 0;
}
yyerror()
{
printf("error");
}
```

OUTPUT:

```
Enter an expression :
result=2+3*4
t0 = 2;
t1 = 3;
t2 = 4;
t3 = t1 * t2;
t4 = t0 + t3;
result = t4
```

```
Enter an expression :
result=((2^3^1^3)*(2*3^1^3))*(2^2*2)+2+2
t0 = 2;
t1 = 3;
t2 = 1;
t3 = 3;
t4 = t2 ^ t3;
t5 = t1 ^ t4;
t6 = t0 ^ t5;
t7 = 2;
t8 = 3;
t9 = 1;
t10 = 3;
t11 = t9 ^ t10;
t12 = t8 ^ t11;
t13 = t7 * t12;
t14 = t6 * t13;
t15 = 2;
t16 = 2;
t17 = t15 ^ t16;
t18 = 2;
t19 = t17 * t18;
t20 = t14 * t19;
t21 = 2;
t22 = t20 + t21;
t23 = 2;
t24 = t22 + t23;
result = t24
```