# PRACTICAL - 1

AIM :

Implement a lexical analyzer for a subset of C using LEX Implementation should support Error handling.

THEORY :

Lexical analysis is the starting phase of the compiler. It gathers modified source code that is written in the form of sentences from the language preprocessor. The lexical analyzer is responsible for breaking these syntaxes into a series of tokens, by removing whitespace in the source code. If the lexical analyzer gets any invalid token, it generates an error. The stream of character is read by it and it seeks the legal tokens, and then the data is passed to the syntax analyzer, when it is asked for.

Architecture of Lexical Analyzer:

To read the input character in the source code and produce a token is the most important task of a lexical analyzer. The lexical analyzer goes through with the entire source code and identifies each token one by one. The scanner is responsible to produce tokens when it is requested by the parser. The lexical analyzer avoids the whitespace and comments while creating these tokens. If any error occurs, the analyzer correlates these errors with the source file and line number.
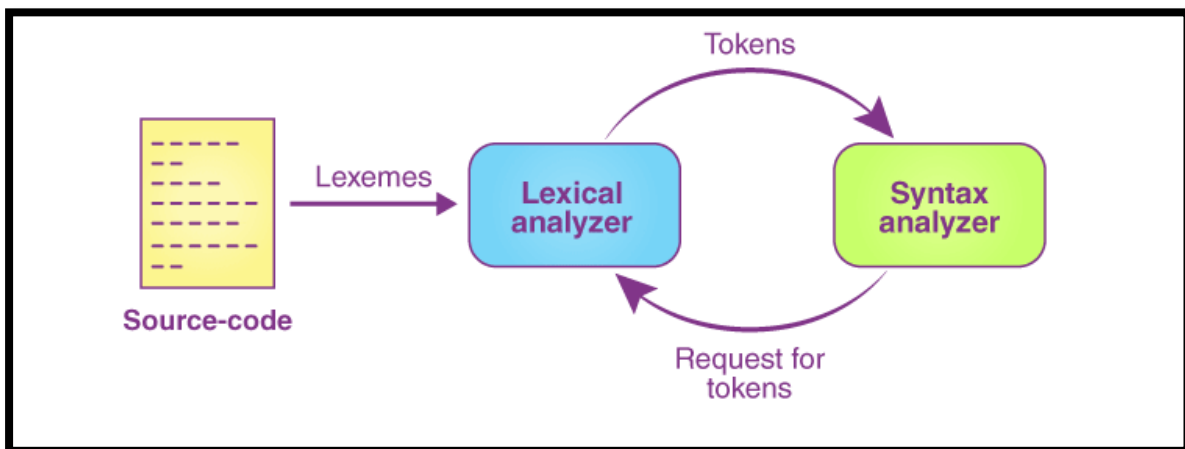
IMPLEMENTATION :

- lex <filename with .l extension>
- gcc<newly created .c file> -o <file name for exe file>
- <filename of exe file>

In this case, create an extra text file named abc.txt which will contain some C code to work as input for lexical analysis.

CODE :

```
%%
"#" {printf("\n %s \t Preprocessor",yytext);}

"main"|"printf"|"scanf" {printf("\n%s\tfunction",yytext);}

"if"|"else"|"int"|"unsigned"|"long"|"char"|"switch"|"case"|"struct"|"do"|"while"|"void"|"for"|"float"|"continue"|"break"|"include" { printf("\n%s\tKeyword",yytext); }

[_a-zA-Z][_a-zA-Z0-9]* {printf("\n%s\tIdenifier",yytext);}

"+"|"/"|"*"|"-" {printf("\n%s\tOperator",yytext);}

"="|"<"|">"|"!="|"=="|"<="|">=" {printf("\n%s\tRelational Operator",yytext);}

"%d"|"%s"|"%c"|"%f" {printf("\n%s\tTokenizer",yytext);}

"stdio.h"|"conio.h"|"math.h"|"string.h"|"graphics.h"|"dos.h"            {printf("\n%s\tHeader File",yytext);}

";"|"," {printf("\n%s\tDelimiter",yytext);}


"{" {printf("\n%s\tStart Of Function/Loop",yytext);}

"}" {printf("\n%s\tEnd of Function",yytext);}
%%
int yywrap(void)
{
  return 1;
}
int main()
{
int i;
FILE *fp;
fp=fopen("a.txt","r");
        if(fp==NULL)
{
        printf("Unable To Open File");
}
else
{
        yyin=fp;
```

}

yylex();

return 0;

}

OUTPUT :

```
D:\Education\SEM 7 PRACTICALS\DLP\Practical 1>gcc lex.yy.c -o prac1
pract1.l: In function 'yylex':
pract1.l:11:5: warning: implicit declaration of function 'strcmp' [-Wimplicit-function-declaration]
 "("|")" {if(strcmp(yytext,"(")==0)
     ^~~~~~
```

```
D:\Education\SEM 7 PRACTICALS\DLP\Practical 1>prac1

 #       Preprocessor
include Keyword

a11     Idenifier
D:\Education\SEM 7 PRACTICALS\DLP\Practical 1>prac1

 #       Preprocessor
include Keyword
<       Relational Operator
stdio.h Header File
>       Relational Operator

void    Keyword
main    function
(       Opening Parenthesis
)       Closing Parenthesis

{       Start Of Function/Loop

int     Keyword
ab      Idenifier
;       Delimiter

printf  function
(       Opening Parenthesis"
hii     Idenifier"
)       Closing Parenthesis
;       Delimiter

}       End of Function
D:\Education\SEM 7 PRACTICALS\DLP\Practical 1>_
```

CONCLUSION :

In this practical, we learnt about lex files and implemented a program for lexical analysis.

PRACTICAL - 2

AIM :

Implement a lexical analyzer for identification of numbers.

THEORY :

The main task of lexical analysis is to read input characters in the code and produce tokens.Lexicalanalyzer scans the entire source code of the program. It identifies each token one by one. Scanners are usually implemented to produce tokens only when requested by a parser. Here is how recognition of tokens in compiler design works-

1. "Get next token" is a command which is sent from the parser to the lexical analyzer.
2. On receiving this command, the lexical analyzer scans the input until it finds the next token.
3. It returns the token to Parser.

Lexical Analyzer skips whitespaces and comments while creating these tokens. If any error is present, then Lexical analyzer will correlate that error with the source file and line number.

Roles of the Lexical analyzer

- Helps to identify token into the symbol table
- Removes white spaces and comments from the source program
- Correlates error messages with the source program
- Helps you to expands the macros if it is found in the source program
- Read input characters from the source program

IMPLEMENTATION :

- lex <filename with .l extension>
- gcc<newly created .c file> -o <file name for exe file>
- <filename of exe file>

CODE :

bin (0|1)+

char [A-Za-z]+

digit [0-9]

oct [0-7]

dec [0-9]*

float {digit}+("."{digit}+)

exp {digit}+("."{digit}+)?("E"("+"|"-")?{digit}+)?

hex [0-9a-fA-F]+

```
%%
{bin} {printf("\n %s= it is a binary number",yytext);}
{char} {printf("\n %s=it is a char",yytext);}
{oct} {printf("\n %s=it is a octal number",yytext);}
{digit} {printf("\n %s=it is a digit",yytext);}
{dec} {printf("\n %s=it is a decimal",yytext);}
{float} {printf("\n %s=it is a float",yytext);}
{exp} {printf("\n %s=it is a exp",yytext);}
{hex} {printf("\n %s=it is a hex",yytext);}
%%
int yywrap(){
  return 1;}
int main(){
printf("Enter the number=");
yylex();
  return 0;}
```
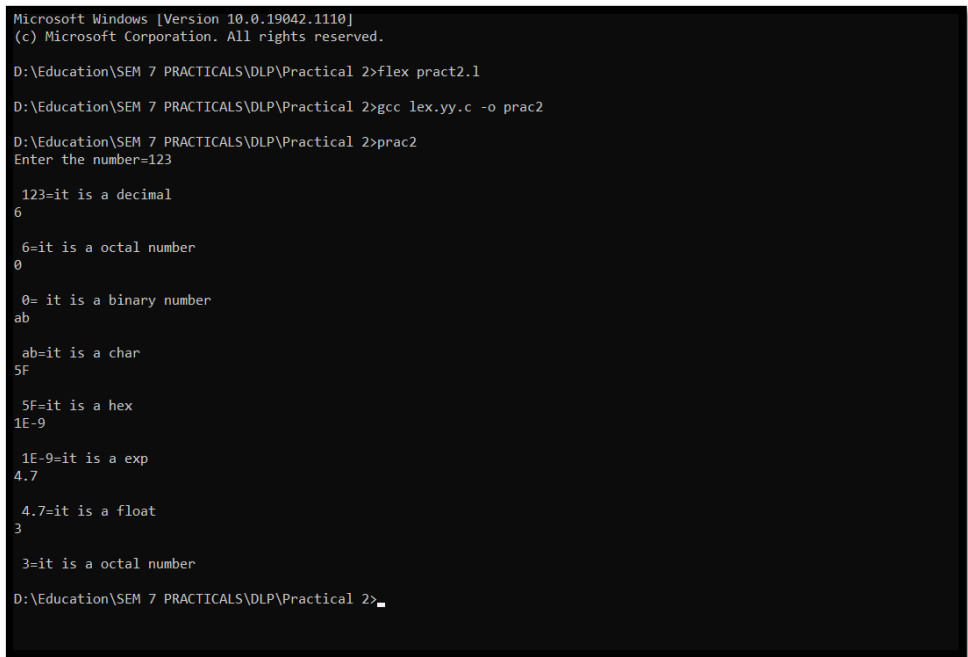
OUTPUT :

CONCLUSION :

In this practical, we learnt about lexical analysis for numbers and characters.

PRACTICAL - 3

AIM :

Write an ambiguous CFG to recognize an infix expression and implement a parser that recognizes the infix expression using YACC.

THEORY :

YACC:

YACC (Yet another Compiler Compiler) is a tool used to generate a parser. This document is a tutorial for the use of YACC to generate a parser for ExpL. YACC translates a given Context Free Grammar (CFG) specifications (input in input_file.y) into a C implementation (y.tab.c) of a corresponding push down automaton (i.e., a finite state machine with a stack). This C program when compiled, yields an executable parser.

Parser:

A parser is a program that checks whether its input (viewed as a stream of tokens) meets a given grammar specification. The syntax of SIL can be specified using a Context Free Grammar. As mentioned earlier, YACC takes this specification and generates a parser for SIL.

Context Free Grammar (CFG):

A context free grammar is defined by a four tuple (N, T, P, S) - a set N of non-terminals, a set T of terminals (in our project, these are the tokens returned by the lexical analyzer and hence we refer to them as tokens frequently), set P of productions and a start variable S. Each production consists of a non-terminal on the left side (head part) and a sequence of tokens and non-terminals (of zero or more length) on the right side (body part).

IMPLEMENTATION :

- yacc<filename with .y extension>
- gcc<newly created .c file> -o <file name for exe file>
- <filename of exe file>

CODE :

%{

/*** Auxiliary declarations section ***/

#include<stdio.h>

#include<stdlib.h>

#include<string.h>

/* Custom function to print an operator*/

void print_operator(char op);

```
/* Variable to keep track of the position of the number in the input */
int pos=0;
char p;
%}
 /*** YACC Declarations section ***/
%token NUM
%left '+'
%left '*'
%%
/*** Rules Section ***/
start : expr '\n'        {exit(1);}
    ;
expr: expr '+' expr    {print_operator('+');}
   | expr '*' expr    {print_operator('*');}
   | '(' expr ')'
   | NUM          {printf("%c ",p);}
   ;
%%
/*** Auxiliary functions section ***/
void print_operator(char c){
   switch(c){
     case '+' :printf("+ ");
           break;
     case '*' :printf("* ");
break;  }
   return;}
yyerror(char const *s){
printf("yyerror %s",s);}
yylex(){
   char c;
   c = getchar();
   p=c;
```

```
    if(isdigit(c)){
pos++;
        return NUM;
    }
    else if(c == ' '){
yylex();
    }
    else {
        return c;}}
main(){
        yyparse();
        return 1;}
```

OUTPUT :

```
D:\Education\SEM 7 PRACTICALS\DLP\Practical 3>yacc infix.y

D:\Education\SEM 7 PRACTICALS\DLP\Practical 3>gcc infix.tab.c -o prac3
infix.tab.c: In function 'yyparse':
infix.tab.c:589:16: warning: implicit declaration of function 'yylex' [-Wimplicit-function-declaration]
 # define YYLEX yylex ()
                ^~~~~
infix.tab.c:1249:16: note: in expansion of macro 'YYLEX'
      yychar = YYLEX;
               ^~~~~
infix.tab.c:1403:7: warning: implicit declaration of function 'yyerror'; did you mean 'yyerrok'? [-Wimplicit-function-declaration]
      yyerror (YY_("syntax error"));
      ^~~~~~~
      yyerrok
infix.y: At top level:
infix.y:47:1: warning: return type defaults to 'int' [-Wimplicit-int]
 yyerror(char const *s)
 ^~~~~~~
infix.y:52:1: warning: return type defaults to 'int' [-Wimplicit-int]
 yylex(){
 ^~~~~
infix.y: In function 'yylex':
infix.y:56:8: warning: implicit declaration of function 'isdigit' [-Wimplicit-function-declaration]
     if(isdigit(c)){
        ^~~~~~~
infix.y: At top level:
infix.y:68:1: warning: return type defaults to 'int' [-Wimplicit-int]
 main()
 ^~~~
```

```
D:\Education\SEM 7 PRACTICALS\DLP\Practical 3>prac3
(1 + 2) * 3
1 2 + 3 *
D:\Education\SEM 7 PRACTICALS\DLP\Practical 3>_
```

CONCLUSION :

In this practical, we learnt about yacc and performed infix to postfix conversion.

PRACTICAL - 4

AIM :

Implement a Calculator using LEX and YACC.

THEORY :

Step1: A Yacc source program has three parts as follows:

Declarations %% translationrules %% supporting C routines

Step2: Declarations Section: This section contains entries that:

- Include standard I/O header file.
- Define global variables.
- Define the list rule as the place to start processing.
- Define the tokens used by the parser.
- Define the operators and their precedence.

Step3: Rules Section: The rules section defines the rules that parse the input stream. Each rule of a grammar production and the associated semantic action.

Step4: Programs Section: The programs section contains the following subroutines. Because these subroutines are included in this file, it is not necessary to use the yacc library when processing this file.

Step5: Main- The required main program that calls the yyparse subroutine to start the program.

Step6:yyerror(s) -This error-handling subroutine only prints a syntax error message.

Step7:yywrap -The wrap-up subroutine that returns a value of 1 when the end of input occurs. The calc.lex file contains include statements for standard input and output, as programmar file information if we use the -d flag with the yacc command. The y.tab.h file contains definitions for the tokens that the parser program uses.

Step8:calc.lex contains the rules to generate these tokens from the input stream.

IMPLEMENTATION :

- lex <filename with .l extension>
- yacc<filename with .y extension>
- gcc<newly created .c file from yacc> -o <file name for exe file>
- <filename of exe file>

CODE :

DIGIT [0-9]

%option noyywrap

%%

```
{DIGIT} { yylval=atof(yytext); return NUM;}
\n|. {return yytext[0];}
```

Yacc File:

```
%{
#include<ctype.h>
#include<stdio.h>
#define YYSTYPE double
%}
%token NUM
%left '+' '-'
%left '*' '/'
%%
S : S E '\n' { printf("Answer: %g \nEnter:\n", $2); }
| S '\n'
|
| error '\n' { yyerror("Error: Enter once more�\n" );yyerrok; }
;
E : E '+' E { $$ = $1 + $3; }
| E'-'E { $$=$1-$3;}
| E'*'E {$$=$1*$3;}
| E'/'E {$$=$1/$3;}
| NUM
;
%%
#include "lex.yy.c"
int main()
{
printf("Enter the expression: ");
yyparse();
}
yyerror (char * s)
{
```

printf ("% s \n", s);

exit (1);

}

OUTPUT :

CONCLUSION :

In this practical, we learnt implemented a calculator using lex and yacc which takes expression as input and perform basic arithmetic operations.

PRACTICAL - 5

AIM :

Implementation of Syntax Tree.

THEORY :

An ordered rooted tree that describes the syntactic structure of a string according to context-free grammar is known as a parse tree, parsing tree, derivation tree, or syntax tree. The phrase parse tree is mostly used in computer linguistics; the word syntax tree is more commonly used in theoretical syntax.

Each leaf node represents an operand, whereas each inner node represents an operator in a syntax tree. The Syntax Tree is an abbreviation for the Parse Tree.

Rules for constructing Syntax Tree:

The syntax tree nodes can all be treated as data with several fields. The operator is identified by one node element, whereas the remaining areas include a pointer to the operand nodes. The node's label is also known as the operator. The following functions are used to generate the syntax tree nodes for expressions using binary operators. Each method returns a pointer to the most recently developed node.

- **Mknode (op, left, right):** It creates an operator node with the name op and two fields, containing left and right pointers.
- **Mkleaf (id, entry):** It creates an identifier node with the label id and the entry field, which contains a reference to the identifier's symbol table entry.
- **Mkleaf (num, Val):** It creates a number node with the name num and a field containing the number's value, Val. For example, construct a syntax tree for an expression $p - 4 + q$. In this sequence, a1, a2…. a5 are pointers to the symbol table entries for identifier 'p' and 'q' respectively.

IMPLEMENTATION :

- gcc<newly created .c file> -o <file name for exe file>
- <filename of exe file>

In this case, create a syntax.txt file as input for the executable which will contain following statements.

t1=a+b

t2=c-d

t3=e+t2
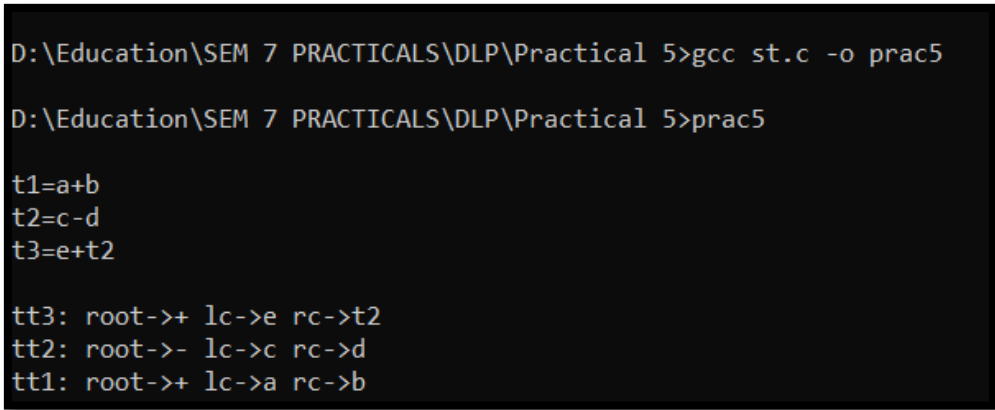
t4=t1-t3

CODE :

```c
#include<conio.h>
#include<stdio.h>
int main(){
FILE *fp;
int i=0,j=0,k,l,row,col,s,x;
char a[10][10],ch,main[50],search;
//clrscr();
fp=fopen("syntax.txt","r+");
while((ch=fgetc(fp))!=EOF){
if(ch=='\n'){
row=i;
col=j;
j=0;
i++;
}
else{
a[i][j]=ch
; j++;
}}
printf("\n");
for(k=0;k<row+1;k++){
for(l=0;l<col;l++){
printf("%c",a[k][l]);
}
printf("\n");
}
i=0;
s=0;
for(k=0;k<row+1;k++){
main[i]=a[k][1];
i++;
```

```
if(a[k][3]=='t'){
search=a[k][4];
for(l=0;l<i;l++){
if(main[l]==search){
main[i]=main[l];
i++;
break;
}
}
main[i]=a[k][5];
s=5;
i++;
}
else{
main[i]=a[k][3];
// printf("\n%c",main[i]);
i++;
main[i]=a[k][4];
// printf(",%c\n",main[i]);
s=4;
i++;
}
s++;
if(a[k][s]=='t'){
s++;
search=a[k][s];
for(l=0;l<i;l++){
if(main[l]==search)
{
main[i]=main[l];
i++;
break;}}}
```

else{

main[i]=a[k][s];

i++;}}

for(x=i-1;x>=0;x=x-4){

printf("\ntt%c: root->%c ",main[x-3],main[x-1]);

if(main[x-2]>48 &&main[x-2]<59)

printf("lc->t%c",main[x-2]);

else

printf("lc->%c ",main[x-2]);

if(main[x]>48 &&main[x]<59)

printf("rc->t%c",main[x]);

else

printf("rc->%c ",main[x]);

}

getch();

}

OUTPUT :

```
D:\Education\SEM 7 PRACTICALS\DLP\Practical 5>gcc st.c -o prac5

D:\Education\SEM 7 PRACTICALS\DLP\Practical 5>prac5

t1=a+b
t2=c-d
t3=e+t2

tt3: root->+ lc->e rc->t2
tt2: root->- lc->c rc->d
tt1: root->+ lc->a rc->b
```

CONCLUSION :

In this practical, we learnt about syntax tree and implemented the concept using C.

PRACTICAL - 6

AIM :

Implementation of Context Free Grammar.

THEORY :

Context-free grammars (CFGs) are used to describe context-free languages. A context-free grammar is a set of recursive rules used to generate patterns of strings. A context-free grammar can describe all regular languages and more, but they cannot describe all possible languages.

Context-free grammars can generate context-free languages. They do this by taking a set of variables which are defined recursively, in terms of one another, by a set of production rules. Context-free grammars are named as such because any of the production rules in the grammar can be applied regardless of context—it does not depend on any other symbols that may or may not be around a given symbol that is having a rule applied to it.

A context-free grammar (CFG) consisting of a finite set of grammar rules is a quadruple (N, T, P, S) where

- N is a set of non-terminal symbols.
- T is a set of terminals where N ∩ T = NULL.
- P is a set of rules, P: N → (N ∪ T)*, i.e., the left-hand side of the production rule P does have any right context or left context.
- S is the start symbol.

IMPLEMENTATION :

- gcc<our .c file> -o <file name for exe file>
- <filename of exe file>

In this case, create a syntax.txt file as input for the executable which will contain following statements.

S aBaA

S AB

A Bc

B c

CODE :

#include<stdio.h>

#include<string.h>

#include<conio.h>

int i,j,k,l,m,n=0,o,p,nv,z=0,t,x=0;

```c
char str[10],temp[20],temp2[20],temp3[20];
struct prod
{
   char lhs[10],rhs[10][10];
   int n;
}pro[10];


void findter()
{
   for(k=0;k<n;k++)
   {
      if(temp[i]==pro[k].lhs[0])
      {
         for(t=0;t<pro[k].n;t++)
         {
            for(l=0;l<20;l++)
               temp2[l]='\0';
            for(l=i+1;l<strlen(temp);l++)
               temp2[l-i-1]=temp[l];
            for(l=i;l<20;l++)
               temp[l]='\0';
            for(l=0;l<strlen(pro[k].rhs[t]);l++)
               temp[i+l]=pro[k].rhs[t][l];
strcat(temp,temp2);
            if(str[i]==temp[i])
               return;
            else if(str[i]!=temp[i] && temp[i]>=65 && temp[i]<=90)
               break;
         }
         break;
      }
   }
```

```c
    if(temp[i]>=65 && temp[i]<=90)

findter();

}

int main()

{

    FILE *f;

//    clrscr();


    for(i=0;i<10;i++)

        pro[i].n=0;


    f=fopen("input.txt","r");

    while(!feof(f))

    {

fscanf(f,"%s",pro[n].lhs);

        if(n>0)

        {

if( strcmp(pro[n].lhs,pro[n-1].lhs) == 0 )

            {

                pro[n].lhs[0]='\0';

fscanf(f,"%s",pro[n-1].rhs[pro[n-1].n]);

                pro[n-1].n++;

                continue;

            }

        }

fscanf(f,"%s",pro[n].rhs[pro[n].n]);

        pro[n].n++;

        n++;

    }

    n--;


printf("\n\nTHE GRAMMAR IS AS FOLLOWS\n\n");
```
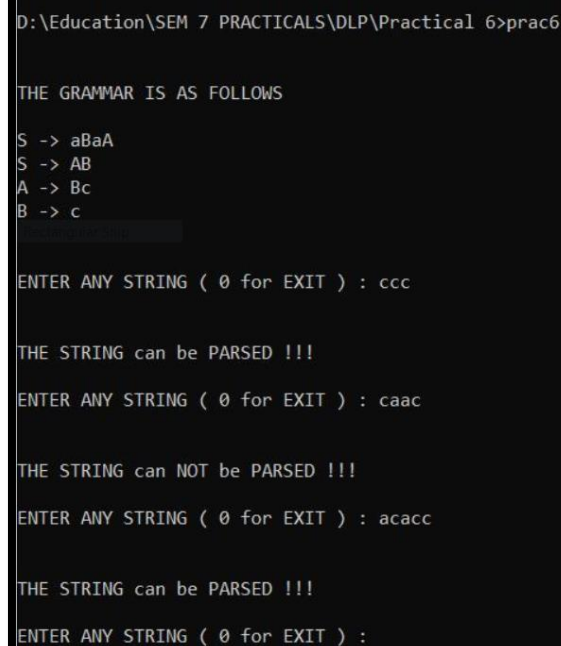
```
  for(i=0;i<n;i++)
     for(j=0;j<pro[i].n;j++)
printf("%s -> %s\n",pro[i].lhs,pro[i].rhs[j]);


while(1)
   {
     for(l=0;l<10;l++)
str[0]=NULL;


printf("\n\nENTER ANY STRING ( 0 for EXIT ) : ");
scanf("%s",str);
     if(str[0]=='0')
printf("Exit");
  //     exit(1);
     for(j=0;j<pro[0].n;j++)
     {
       for(l=0;l<20;l++)
          temp[l]=NULL;
strcpy(temp,pro[0].rhs[j]);
       m=0;
       for(i=0;i<strlen(str);i++)
       {
         if(str[i]==temp[i])
           m++;
         else if(str[i]!=temp[i] && temp[i]>=65 && temp[i]<=90)
         {
findter();
            if(str[i]==temp[i])
              m++;
         }
         else if( str[i]!=temp[i] && (temp[i]<65 || temp[i]>90) )
            break;
```

```
        }
        if(m==strlen(str) &&strlen(str)==strlen(temp)){
                printf("\n\nTHE STRING can be PARSED !!!");
                break;
           }
        }
        if(j==pro[0].n)
        printf("\n\nTHE STRING can NOT be PARSED !!!");
    }
getch();
}
```

OUTPUT :

```
D:\Education\SEM 7 PRACTICALS\DLP\Practical 6>prac6

THE GRAMMAR IS AS FOLLOWS

S -> aBaA
S -> AB
A -> Bc
B -> c


ENTER ANY STRING ( 0 for EXIT ) : ccc


THE STRING can be PARSED !!!

ENTER ANY STRING ( 0 for EXIT ) : caac


THE STRING can NOT be PARSED !!!

ENTER ANY STRING ( 0 for EXIT ) : acacc


THE STRING can be PARSED !!!

ENTER ANY STRING ( 0 for EXIT ) :
```

CONCLUSION :

In this practical, we learnt about Context Free Grammar and implemented the concept using
C.

PRACTICAL - 7

AIM :

Design of a Predictive parser.

THEORY :

Predictive parser is a recursive descent parser, which has the capability to predict which production is to be used to replace the input string. The predictive parser does not suffer from backtracking.

To accomplish its tasks, the predictive parser uses a look-ahead pointer, which points to the next input symbols. To make the parser back-tracking free, the predictive parser puts some constraints on the grammar and accepts only a class of grammar known as LL (k) grammar.

Predictive parsing uses a stack and a parsing table to parse the input and generate a parse tree. Both the stack and the input contain an end symbol $ to denote that the stack is empty and the input is consumed. The parser refers to the parsing table to take any decision on the input and stack element combination.

In recursive descent parsing, the parser may have more than one production to choose from for a single instance of input, whereas in predictive parser, each step has at most one production to choose. There might be instances where there is no production matching the input string, making the parsing procedure to fail.

IMPLEMENTATION :

- gcc<our .c file> -o <file name for exe file>
- <filename of exe file>

CODE:

```
#include<stdio.h>

#include<conio.h>

#include<string.h>

void main(){

char fin[10][20],st[10][20],ft[20][20],fol[20][20];

int a=0,e,i,t,b,c,n,k,l=0,j,s,m,p;

//clrscr();

printf("enter the no. of coordinates\n");

scanf("%d",&n);

printf("enter the productions in a grammar\n");

for(i=0;i<n;i++)

scanf("%s",st[i]);
```

```
for(i=0;i<n;i++)
fol[i][0]='\0';
for(s=0;s<n;s++){
for(i=0;i<n;i++)
{
j=3;
l=0;
a=0;
l1:if(!((st[i][j]>64)&&(st[i][j]<91))){
for(m=0;m<l;m++){
if(ft[i][m]==st[i][j])
goto s1;
}
ft[i][l]=st[i][j];
l=l+1;
s1:j=j+1;
}
else{
if(s>0){
while(st[i][j]!=st[a][0]){
a++;
}
b=0;
while(ft[a][b]!='\0'){
for(m=0;m<l;m++){
if(ft[i][m]==ft[a][b])
goto s2;
}
ft[i][l]=ft[a][b];
l=l+1;
s2:b=b+1;
}}}
```

```
while(st[i][j]!='\0'){
if(st[i][j]=='|'){
j=j+1;
goto l1;
}
j=j+1;
}
ft[i][l]='\0';
}}
printf("first pos\n");
for(i=0;i<n;i++)
printf("FIRS[%c]=%s\n",st[i][0],ft[i]);
fol[0][0]='$';
for(i=0;i<n;i++){
k=0;
j=3;
if(i==0)
l=1;
else
l=0;
k1:while((st[i][0]!=st[k][j])&&(k<n)){
if(st[k][j]=='\0'){
k++;
j=2;
}
j++;
}
j=j+1;
if(st[i][0]==st[k][j-1]){
if((st[k][j]!='|')&&(st[k][j]!='\0')){
a=0;
if(!((st[k][j]>64)&&(st[k][j]<91))){
```

```
for(m=0;m<l;m++){
if(fol[i][m]==st[k][j])
goto q3;
}
fol[i][l]=st[k][j];
q3:l++;
    }
else{
while(st[k][j]!=st[a][0]){
a++;
}
p=0;
while(ft[a][p]!='\0'){
if(ft[a][p]!='@'){
for(m=0;m<l;m++){
if(fol[i][m]==ft[a][p])
goto q2;
}
fol[i][l]=ft[a][p];
l=l+1;
}
else
e=1;
q2:p++;
}
if(e==1){
e=0;
goto a1;
}}}
else{
a1:c=0;
a=0;
```

```
while(st[k][0]!=st[a][0]){

a++;

}

while((fol[a][c]!='\0')&&(st[a][0]!=st[i][0])){

for(m=0;m<l;m++){

if(fol[i][m]==fol[a][c])

goto q1;

}

fol[i][l]=fol[a][c];

l++;

q1:c++;

}}

goto k1;

}

fol[i][l]='\0';

}

printf("follow pos\n");

for(i=0;i<n;i++)

printf("FOLLOW[%c]=%s\n",st[i][0],fol[i]);

printf("\n");

s=0;

for(i=0;i<n;i++){

j=3;

while(st[i][j]!='\0'){

if((st[i][j-1]=='|')||(j==3)){

for(p=0;p<=2;p++){

fin[s][p]=st[i][p];

}

t=j;

for(p=3;((st[i][j]!='|')&&(st[i][j]!='\0'));p++){

fin[s][p]=st[i][j];

j++;
```

```
}
fin[s][p]='\0';
if(st[i][k]=='@'){
b=0;
a=0;
while(st[a][0]!=st[i][0]){
a++;
}
while(fol[a][b]!='\0'){
printf("M[%c,%c]=%s\n",st[i][0],fol[a][b],fin[s]);
b++;
}}
else if(!((st[i][t]>64)&&(st[i][t]<91)))
printf("M[%c,%c]=%s\n",st[i][0],st[i][t],fin[s]);
else{
b=0;
a=0;
while(st[a][0]!=st[i][3]){
a++;
}
while(ft[a][b]!='\0'){
printf("M[%c,%c]=%s\n",st[i][0],ft[a][b],fin[s]);
b++;}}
s++;}
if(st[i][j]=='|')
j++;}}
getch();
}
```

OUTPUT :

```
enter the no. of coordinates
2
enter the productions in a grammar
S->CC
C->eC | d
first pos
FIRS[S]=e
FIRS[C]=e
follow pos
FOLLOW[S]=$
FOLLOW[C]=e$

M[S,e]=S->CC
M[C,e]=C->eC
```

CONCLUSION :

In this practical, we learnt about first and follow and implemented predictive parser using C.

PRACTICAL - 8

AIM :

Implementation of code generator.

THEORY :

The final phase in compiler model is the code generator. It takes as input an intermediate representation of the source program and produces as output an equivalent target program.

The code generation techniques presented below can be used whether or not an optimizing phase occurs before code generation.

Fig. 4.1 Position of code generator
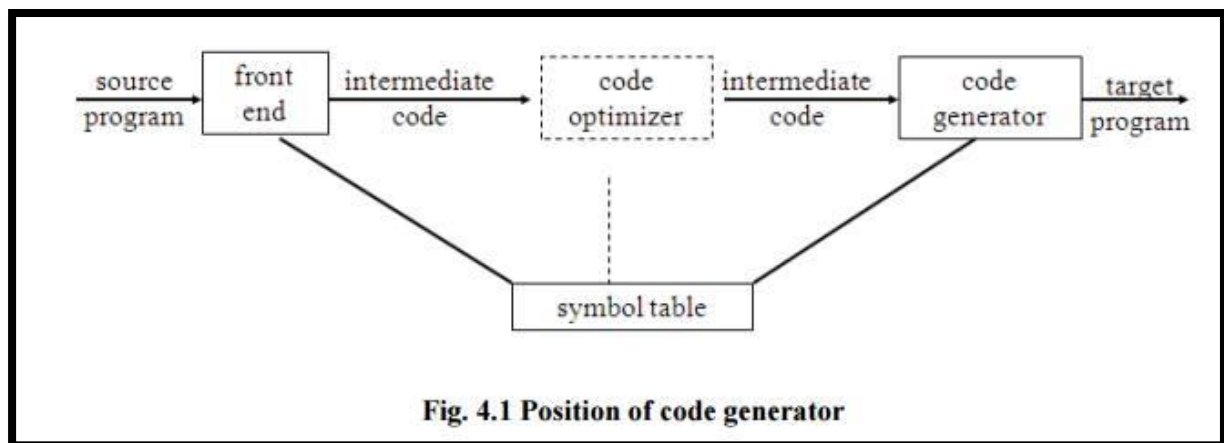
- The code generated by the compiler is an object code of some lower-level programming language, for example, assembly language.
- The source code written in a higher-level language is transformed into a lower-level language that results in a lower-level object code, which should have the following minimum properties:
- It should carry the exact meaning of the source code.
- It should be efficient in terms of CPU usage and memory management.

IMPLEMENTATION :

- gcc<our .c file> -o <file name for exe file>
- <filename of exe file>

Content of Input1.txt:

a=b+c;

d=n+s;

p=q;

CODE :

```c
#include<stdio.h>
#include<string.h>
struct table{
char op1[2];
char op2[2];
char opr[2];
char res[2];
}tbl[100];
void add(char *res,char *op1, char *op2,char *opr){
      FILE *ft;
      char string[20];
      char sym[100];
      ft=fopen("result.asm","a+");
      if(ft==NULL)
            ft=fopen("result.asm","w");
      printf("\nUpdating Assembly Code for the Input File : File : Result.asm ; Status [ok]\n");
      //sleep(2);
      strcpy(string,"mov r0,");
      strcat(string,op1);
      if(strcmp(opr,"&")==0){
      }
      else{
            strcat(string,"\nmov r1,");
            strcat(string,op2);}
      fputs(string,ft);
      if(strcmp(opr,"+")==0)
            strcpy(string,"\nadd r0,r1\n");
      else if(strcmp(opr,"-")==0)
            strcpy(string,"\nsub r0,r1\n");
      else if(strcmp(opr,"/")==0)
```

```
                strcpy(string,"\ndiv r0,r1\n");
        else if(strcmp(opr,"*")==0)
                strcpy(string,"\nmul r0,r1\n");
        else if(strcmp(opr,"&")==0)
                strcpy(string,"\n");
        else
                strcpy(string,"\noperation r0,r1\n");
        fputs(string,ft);
        strcpy(string,"mov ");
        strcat(string,res);
        strcat(string,", r0\n");
        fputs(string,ft);
        fclose(ft);
        string[0]='\0';
        sym[0]='\0';
}
main(){
        int res,op1,op2,i,j,opr;
        FILE *fp;
        char filename[50];
        char s,s1[10];
        system("clear");
        remove("result.asm");
        remove("result.sym");
        res=0;op1=0;op2=0;i=0;j=0;opr=0;
        printf("\n Enter the Input Filename with no white spaces:");
        scanf("%s",filename);
        fp=fopen(filename,"r");
        if(fp==NULL){
                printf("\n cannot open the input file !\n");
                return(0);
        }
```

```
        else{
                while(!feof(fp)){
                        s=fgetc(fp);
                                if(s=='='){
                                        res=1;
                                        op1=op2=opr=0;
                                        s1[j]='\0';
                                        strcpy(tbl[i].res,s1);
                                        j=0;}
                                else if(s=='+'||s=='-'||s=='*'||s=='/'){
                                        op1=1;
                                        opr=1;
                                        s1[j]='\0';
                                        tbl[i].opr[0]=s;
                                        tbl[i].opr[1]='\0';
                                        strcpy(tbl[i].op1,s1);
                                        j=0;}
                                else if(s==';'){
                                        if(opr) {
                                                op2=1;
                                                s1[j]='\0';
                                                strcpy(tbl[i].op2,s1);}
                                        else if(!opr){
                                                op1=1;
                                                op2=0;
                                                s1[j]='\0';
                                                strcpy(tbl[i].op1,s1);
                                                strcpy(tbl[i].op2,"&");
                                                strcpy(tbl[i].opr,"&"); }
                                        add(tbl[i].res,tbl[i].op1,tbl[i].op2,tbl[i].opr);
                                        i++;
                                        j=0;
```
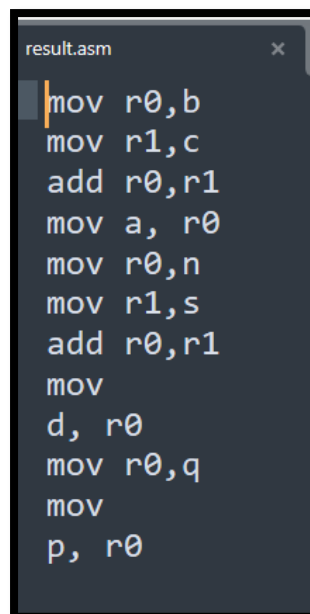
```
                                        opr=op1=op2=res=0;
                        }
                        else{
                                s1[j]=s;
                                j++;}}
        system("clear");}
        return 0;
}
```

OUTPUT :

```
 Enter the Input Filename with no white spaces:input1.txt

Updating Assembly Code for the Input File : File : Result.asm ; Status [ok]

Updating Assembly Code for the Input File : File : Result.asm ; Status [ok]

Updating Assembly Code for the Input File : File : Result.asm ; Status [ok]
```

```
result.asm                    ×
mov r0,b
mov r1,c
add r0,r1
mov a, r0
mov r0,n
mov r1,s
add r0,r1
mov
d, r0
mov r0,q
mov
p, r0
```

CONCLUSION :

In this practical, we learnt about code generation and implemented the same using C.

cc

```c
char *p, *l;
char temp, t;
char *tem;
//clrscr();
printf("enter no of values=");
scanf("%d", &n);
//n=5;
for (i = 0; i< n; i++){
printf("\t left: \t");
scanf(" %c", &op[i].l);
printf("\t right: \t");
scanf("%s", op[i].r);
}
/*for (i = 0; i< n; i++){
printf("\n right: \t");
scanf("%s", op[i].r);
}*/
printf(" intermediate Code\n");
for (i = 0; i< n; i++){
printf(" %c=", op[i].l);
printf(" %s\n", op[i].r);
}
for (i = 0; i< n - 1; i++){
  temp = op[i].l;
  for (j = 0; j< n; j++){
    p = strchr(op[j].r,temp);
   if (p)
   {
        pr[z].l= op[i].l;
        strcpy(pr[z].r, op[i].r);
        z++;
}}}
```

```c
pr[z].l= op[n - 1].l;
strcpy(pr[z].r, op[n - 1].r);
z++;
printf("\n after dead code elimination \n");
for (k = 0; k < z; k++)
{
printf("%c = \t ",pr[k].l);
printf("%s \n",pr[k].r);
}
for (m = 0; m < z; m++)
{ tem = pr[m].r;
  for (j = m + 1; j < z; j++)
{ p = strstr(tem, pr[j].r);
     if (p)
      {
           t = pr[j].l;
            pr[j].l= pr[m].l;
              for (i = 0; i< z; i++)
               {
           l= strchr(pr[i].r, t);
           if (l){
                    a = l - pr[i].r;
                    //printf("pos: %d",a);
                    pr[i].r[a] = pr[m].l;
                    }
                }
             }
        }
}
printf("eliminate common expression\n");
for(i=0;i<z;i++){
printf("%c\t =", pr[i].l);
```

```c
printf("%s\n", pr[i].r);
}
// duplicate production elimination
for (i = 0; i< z; i++){
for (j = i + 1; j < z; j++){
q = strcmp(pr[i].r, pr[j].r);
if ((pr[i].l == pr[j].l) && !q)
{
pr[i].l = '\0';
//pr[i].r = "NULL";
strcpy( pr[i].r , "NULL");
}}}
printf("optimized code \n");
for (i = 0; i< z; i++){
        if (pr[i].l != '\0'){
        printf("%c =", pr[i].l);
        printf("%s \n", pr[i].r);
}}
getch();
}
```

OUTPUT :

```
D:\Education\SEM 7 PRACTICALS\DLP\Practical 9>gcc cop.c -o x

D:\Education\SEM 7 PRACTICALS\DLP\Practical 9>x
enter no of values=3
        left:  a
        right:          b+c
        left:  b
        right:          5
        left:  c
        right:          7*2
 intermediate Code
 a= b+c
 b= 5
 c= 7*2

 after dead code elimination
b =        5
c =        7*2
eliminate common expression
b        =5
c        =7*2
optimized code
b =5
c =7*2
```

CONCLUSION :

In this practical, we learnt about code optimization and implemented the same using C.

PRACTICAL - 10

AIM :

Implement Deterministic Finite Automata.

THEORY :

In DFA, for each input symbol, one can determine the state to which the machine will move. Hence, it is called Deterministic Automaton. As it has a finite number of states, the machine is called Deterministic Finite Machine or Deterministic Finite Automaton.Formal Definition of a DFA.

A DFA can be represented by a 5-tuple $(Q, \sum, \delta, q0, F)$ where −

- Q is a finite set of states.
- $\sum$ is a finite set of symbols called the alphabet.
- δ is the transition function where $\delta: Q \times \sum \rightarrow Q$
- Q0 is the initial state from where any input is processed $(q0 \in Q)$.
- F is a set of final state/states of Q $(F \subseteq Q)$.

Graphical Representation of a DFA

A DFA is represented by digraphs called state diagram.

- The vertices represent the states.
- The arcs labeled with an input alphabet show the transitions.
- The initial state is denoted by an empty single incoming arc.
- The final state is indicated by double circles.

IMPLEMENTATION :

- gcc<our .c file> -o <file name for exe file>
- <filename of exe file>

CODE :

```
#include <stdio.h>
#include <stdlib.h>
struct node{
 int id_num;
 int st_val;
 struct node *link0;
 struct node *link1;
};
struct node *start, *q, *ptr;
```

```c
int vst_arr[100], a[10];
int main(){
 int count, i, posi, j;
 char n[10];
 //clrscr();
printf("=-=-=-=-=-=-=-=-==-=-=-=-=-=-=-=-=-=\n");
printf("Enter the number of states in the m/c:");
scanf("%d",&count);
 q=(struct node *)malloc(sizeof(struct node)*count);
 for(i=0;i<count;i++){
  (q+i)->id_num=i;
printf("State Machine::%d\n",i);
printf("Next State if i/p is 0:");
scanf("%d",&posi);
  (q+i)->link0=(q+posi);
printf("Next State if i/p is 1:");
scanf("%d",&posi);
  (q+i)->link1=(q+posi);
printf("Is the state final state(0/1)?");
scanf("%d",&(q+i)->st_val);}
printf("Enter the Initial State of the m/c:");
scanf("%d",&posi);
 start=q+posi;
printf("=-=-=-=-=-=-=-==-=-=-=-=-=-=-=-=-=\n");
while(1){
printf("=-=-=-=-=-=-=-==-=-=-=-=-=-=-=-=-=\n");
printf("Perform String Check(0/1):");
scanf("%d",&j);
  if(j){
ptr=start;
printf("Enter the string of inputs:");
scanf("%s",n);
```

```c
posi=0;
   while(n[posi]!='\0'){
  a[posi]=(n[posi]-'0');
  //printf("%c\n",n[posi]);
  //printf("%d",a[posi]);
posi++;}
i=0;
printf("The visited States of the m/c are:");
do{
vst_arr[i]=ptr->id_num;
  if(a[i]==0){
ptr=ptr->link0; }
  else if(a[i]==1){
ptr=ptr->link1;}
else{
printf("iNCORRECTiNPUT\n");
   return;}
printf("[%d]",vst_arr[i]);
i++;
}while(i<posi);
printf("\n");
printf("Present State:%d\n",ptr->id_num);
printf("String Status:: ");
   if(ptr->st_val==1)
printf("String Accepted\n");
   else
printf("String Not Accepted\n");}
  else
   return 0;}
printf("=-=-=-=-=-=-=-==-=-=-=-=-=-=-=-=-=\n");
 return 0;
}
```

OUTPUT :

```
D:\Education\SEM 7 PRACTICALS\DLP\Practical 10>gcc "DFA (1).c" -o x
DFA (1).c: In function 'main':
DFA (1).c:77:7: warning: 'return' with no value, in function returning non-void
      return;
      ^~~~~~
DFA (1).c:15:5: note: declared here
 int main(){
     ^~~~

D:\Education\SEM 7 PRACTICALS\DLP\Practical 10>x
=-=-=-=-=-=-=-==-=-=-=-=-=-=-=-=-=
Enter the number of states in the m/c:2
State Machine::0
Next State if i/p is 0:1
Next State if i/p is 1:0
Is the state final state(0/1)?0
State Machine::1
Next State if i/p is 0:0
Next State if i/p is 1:1
Is the state final state(0/1)?1
Enter the Initial State of the m/c:0
=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
Perform String Check(0/1):1
Enter the string of inputs:00
The visited States of the m/c are:[0][1]
Present State:0
String Status:: String Not Accepted
=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
Perform String Check(0/1):1
Enter the string of inputs:1010
The visited States of the m/c are:[0][0][1][1]
Present State:0
String Status:: String Not Accepted
=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
Perform String Check(0/1):1
Enter the string of inputs:0
The visited States of the m/c are:[0]
Present State:1
String Status:: String Accepted
=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
Perform String Check(0/1):0
```

CONCLUSION :

In this practical, we learnt about deterministic finite automata and implemented the same using C.