EXPERIMENT NO: 01

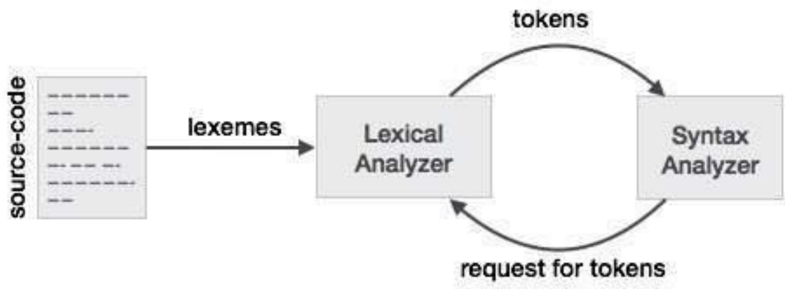
LEXICAL ANALYZER

AIM:

Design and implement a lexical analyzer for given language using C. Program should recognize the tokens such as identifiers, keywords, arithmetic operators, relational operators.

THEORY:

Lexical analysis is the first phase of a compiler. It takes the modified source code from language preprocessors that are written in the form of sentences. The lexical analyzer breaks these syntaxes into a series of tokens, by removing any whitespace or comments in the source code. If the lexical analyzer finds a token invalid, it generates an error. The lexical analyzer works closely with the syntax analyzer. It reads character streams from the source code, checks for legal tokens, and passes the data to the syntax analyzer when it demands.



Tokens :

Lexemes are said to be a sequence of characters (alphanumeric) in a token. There are some predefined rules for every lexeme to be identified as a valid token. These rules are defined by grammar rules, by means of a pattern. A pattern explains what can be a token, and these patterns are defined by means of regular expressions.

In programming language, keywords, constants, identifiers, strings, numbers, operators and punctuations symbols can be considered as tokens.

contains the tokens:

int (keyword), value (identifier), =(operator), 100(constant) and ;(symbol).

Specifications of Tokens

Alphabets :

Any finite set of symbols {0,1} is a set of binary alphabets, {0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F} is a set of Hexadecimal alphabets, {a-z, A-Z} is a set of English language alphabets.

Strings :

Any finite sequence of alphabets is called a string. Length of the string is the total number of occurrence of alphabets. A string having no alphabets, i.e. a string of zero length is known as an empty string and is denoted by ε (epsilon).

Special Symbols

A typical high-level language contains the following symbols:-

Arithmetic Symbols	Addition(+), Subtraction(-), Modulo(%), Multiplication(*), Division(/)
Punctuation	Comma(,), Semicolon(;), Dot(.), Arrow(->)
Assignment	=
Special Assignment	+=, /=, *=, -=
Comparison	==, !=, <, <=, >, >=
Preprocessor	#
Location Specifier	&
Logical	&, &&, |, ||, !
Shift Operator	>>, >>>, <<, <<<

ALGORITHM

Step 1: start

Step 2: Declare ch, ch1, Buffer, operators, file pointer fp and l=1,i=0,j=0

Step 3: Open file program.txt using fp

Step 4: if fp = NULL Display “error while opening “ and go to step 17

```
Step 5: Repeat step 6 to 12 until end of file is reached
Step 6: Read next character in file to ch
Step 7: for i from 0 to 5
        Step 7.1: if ch= operator[i] Display ch is an arithmetic operator in line no l
Step 8: if ch is a digit or "_" and j not equal to 0 then Buffer[j+1] = ch
Step 9: if ch is an alphabet then Buffer[j+1]=ch
Step 10: if ch is a space and j not equal to 0
         Step 10.1: Buffer[j]='\0'
         Step 10.2: j=0
         Step 10.3 : call function isKeyword ,if it returns 1, display "Buffer[] is a Keyword in line, l"
                     otherwise display "Buffer is an identifier
Step 11: if ch= newline
         Step 11.1: incrementl
         Step 11.2: if j is not equal to 0
              Step 11.2.1: check Buffer[j] equal to '\0'
              Step 11.2.2: j=0
              Step 11.2.3: call function isKeyword ,if it returns 1, display "Buffer[] is a Keyword in
                           line, l-1"  otherwise display "Buffer is an identifier in l-1"
Step 12: if ch equals to <,> or =
         Step 12.1 : read next character to ch1
         Step 12.2: if ch1 is equal to '=' then display "chch1 is a relational operator in l"
         Step 12.3: if ch1 is equal to '>'  or '<' then display "chch1 is a relational operator in l"
                    Otherwise move filepointer to previous character and display"ch is a relational
                    operator in line l"
step 13: stop
Algorithm  IsKeyword ()
Step1: Declare all 32 C keywords in Keyword[][]
Step 2: for i from 0 to 31, if keywords[i] equals to buffer then flag =1
Step 3: return flag
Step 4: stop
```

PROGRAM

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<ctype.h>
int isKeyword(char buffer[ ]) {
char Keyword[32][10] =
{‘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’, ‘typdef’, ‘union’, ‘unsigned’, ‘void’, ‘volitile’, ‘while’};
int I, flag =0;
for i=0, i<32;i++){
if(strcmp[Keyword[i],buffer)==0){
flag=1;
break;
}}
return flag;
}
Void main(){
Char ch,ch1, buffer[50], operator[]=’+ - * / %’;
FILE *fp;
int i, j=0, l=1;
fp=fopen(‘program.txt’, ‘r’);
if(fp==NULL)
{
Printf(“\n error while opening file’);
Exit(0);
}
Else{
Printf(‘LEXEME \t LINE NUMBER\t TOKEN’);
While((ch=fgetc(fp))!=EOF)
```

```
{
for(i=0;i<5;++i){
if(ch==operator[i])
{
Printf('\n %c \t\t %d\t arithmetic operator ', ch,l);
Break;
}
}
If(i==5){
If(isalnum(ch)){
If(isdigit(ch) && j==0){ }
Else {
Buffer[j++] = ch;
Ch1= fgetc(fp);
If(isalnum(ch1))
Fseek(fp, -1, SEEK_CUR);
Else {
Fseek (fp,-1,SEEK_CUR);
Buffer[j]='\)';
If(isKeyword(buffer)==1){
Printf('\n%s  \t\t %d \t keyword  ', buffer, l);
J=0;
}
Else{
If(strcmp(buffer, 'main')!=0)
Printf('\n%s \t\t %d\t identifier', buffer, l);
J=0;}
}
}
}
if(ch=='<' || ch=='>'){
ch1= fgetc(fp);
if(ch1== ‘=’)
printf(‘\n%c%c  \t\t %d \t relational operator’,ch,ch1,l);
else{
fseek(fp,-1,SEEK_CUR);
printf(‘\n%c \t\t %d \t relational operator’, ch,l);
}
}
Else if(ch==’=’){
Ch1=fgetc(fp);
if(ch1== ‘=’) {
Printf(‘\n%c%c \t\t %d \t relational operator’, ch, ch1, l);
}
Else{
fseek(fp, -1, SEEK_CUR);
Printf(‘\n%c \t\t %d \t Assignment operator’,ch,l);
}
}
if(ch==’\n’)
L++;
}
}
}}
```

Program.txt

```
Void main()
{
int a,b;
a=5;
b=0;
while(a>0)
{
b=b+1;
a=a-1;
}
}
```

OUTPUT

```
LEXEME    TOKEN                     LINE NUMBER
Void       keyword                  1
int        keyword                  3
a          identifier               3
b          identifier               3
a          identifier               4
=          Assignment operator      4
b          identifier               5
=          Assignment operator      5
while      keyword                  6
a          identifier               6
>          Relational operator      6
b          identifier               8
=          Assignment operator      8
b          identifier               8
+          Arithmetic operator      8
a          identifier               9
=          Assignment operator      9
a          identifier               9
-          Arithmetic operator      9
```

RESULT

The program was executed successfully.

VIVA QUESTIONS

1. What is a token? Give examples for tokens and non-tokens.

2. How does Lexical Analyzer function?

3. Explain lexical analysis phase of compiler.

EXPERIMENT NO: 02

OPERATOR PRECEDENCE PARSER

AIM

Develop an operator precedence parser for a given language

THEORY

A grammar that is generated to define the mathematical operators is called operator grammar with some restrictions on grammar. An operator precedence grammar is a context-free grammar that has the property that no production has either an empty right-hand side (null productions) or two adjacent non-terminals in its right-hand side. An operator precedence parser is a one of the bottom-up parser that interprets an operator-precedence grammar. This parser is only used for operator grammars. *Ambiguous grammars are not allowed* in case of any parser except operator precedence parser. There are two methods for determining what precedence relations should hold between a pair of terminals:

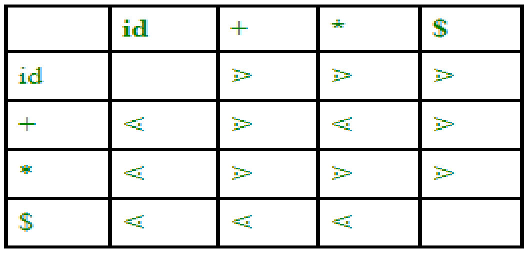
1. Use the conventional associatively and precedence of operator.
2. The second method of selecting operator-precedence relations is first to construct an unambiguous grammar for the language, a grammar that reflects the correct associatively and precedence in its parse trees.

This parser relies on the following three precedence relations: **⋖, ≐, ⋗**

a ⋖ b This means a “yields precedence to” b.

a ⋗ b This means a “takes precedence over”b.

a ≐ b This means a “has precedence as” b.



	id	+	*	$
id		⋗	⋗	⋗
+	⋖	⋗	⋖	⋗
*	⋖	⋗	⋗	⋗
$	⋖	⋖	⋖	

Figure – Operator precedence relation table for grammar E->E+E/E*E/id

There is not given any relation between id and id as id will not be compared and two variables cannot come side by side. There is also a disadvantage of this table as if we have n operators than size of table will be n*n and complexity will be 0(n^2). In order to increase the size of table, use **operator function table**. The operator precedence parsers usually do not store the precedence table with the relations; rather they are implemented in a special way. Operator precedence parsers use **precedence functions** that map terminal symbols to integers, and so the precedence relations between the symbols are implemented by numerical comparison. The parsing table can be encoded by two precedence functions **f** and **g** that map terminal symbols to integers. We select f and g such that:

1. f(a) < g(b) whenever a is precedence to b
2. f(a) = g(b) whenever a and b having precedence
3. f(a) > g(b) whenever a takes precedence over b

ALGORITHM

```
Initialize: Set ip to point to the first symbol of the input string w$
Repeat: Let b be the top stack symbol, a the input symbol pointed to by ip
if (a is $ and b is $)
return
else
if a ·> b or a =· b then
push a onto the stack
advance ip to the next input symbol
else if a <· b then
repeat
c = pop the stack
until (c .> stack-top)
else error
end
```

PROGRAM

```
#include<stdio.h>
```

```
#include<string.h>

char *input;
int i=0;
char lasthandle[6],stack[50],handles[][5]={")E(","E*E","E+E","i","E^E"};
int top=0,l;
char prec[9][9]={  /*stack    +    - * / ^ i ( ) $  */

            /* + */ '>', '>','<','<','<','<','<','>','>',
            /*  - */  '>', '>','<','<','<','<','<','>','>',
            /*  * */  '>', '>','>','>','<','<','<','>','>',
            /* / */ '>', '>','>','>','<','<','<','>','>',
            /* ^ */ '>', '>','>','>','<','<','<','>','>',
            /* i */ '>', '>','>','>','>','e','e','>','>',
            /*  ( */  '<', '<','<','<','<','<','<','>','e',
            /*  ) */  '>', '>','>','>','>','e','e','>','>',
            /* $ */ '<', '<','<','<','<','<','<','<','>',
                };
int getindex(char c)
{
switch(c)
    {
    case '+':return 0;
    case '-':return 1;
    case '*':return 2;
    case '/':return 3;
    case '^':return 4;
    case 'i':return 5;
    case '(':return 6;
    case ')':return 7;
    case '$':return 8;
    }
}
int shift()
{
stack[++top]=*(input+i++);
stack[top+1]='\0';
}
int reduce()
{
```

```
int i,len,found,t;
for(i=0;i<5;i++)//selecting handles
   {
   len=strlen(handles[i]);
   if(stack[top]==handles[i][0]&&top+1>=len)
      {
      found=1;
      for(t=0;t<len;t++)
         {
         if(stack[top-t]!=handles[i][t])
            {
            found=0;
            break;
            }
         }
      if(found==1)
         {
         stack[top-t+1]='E';
         top=top-t+1;
         strcpy(lasthandle,handles[i]);
         stack[top+1]='\0';
         return 1;//successful reduction
         }
      }
  }
return 0;
}
void dispstack()
{
int j;
for(j=0;j<=top;j++)
   printf("%c",stack[j]);
}
void dispinput()
{
int j;
for(j=i;j<l;j++)
   printf("%c",*(input+j));
}
void main()
```

```
{
int j;
input=(char*)malloc(50*sizeof(char));
printf("\nEnter the string\n");
scanf("%s",input);
input=strcat(input,"$");
l=strlen(input);
strcpy(stack,"$");
printf("\nSTACK\tINPUT\tACTION");
while(i<=l)
                {
                shift();
                printf("\n");
                dispstack();
                printf("\t");
                dispinput();
                printf("\tShift");
                if(prec[getindex(stack[top])][getindex(input[i])]=='>')
                    {
                    while(reduce())
                        {
                        printf("\n");
                        dispstack();
                        printf("\t");
                        dispinput();
                        printf("\tReduced: E->%s",lasthandle);
                        }
                    }
                }
```

OUTPUT:

```
Enter the string
i*(i+i)*i

STACK          INPUT          ACTION
$i             *(i+i)*i$      Shift
$E             *(i+i)*i$      Reduced: E->i
$E*            (i+i)*i$       Shift
$E*            ( i+i)*i$      Shift
$E*(           i +i)*i$       Shift
```

```
$E*(E           +i)*i$          Reduced: E->i
$E*(E+          i)*i$           Shift
$E*(E+i         )*i$            Shift
$E*(E+E         )*i$            Reduced: E->i
$E*(E           )*i$            Reduced: E->E+E
$E*(E)          *i$             Shift
$E*E            *i$             Reduced: E->)E(
$E*             i$              Reduced: E->E*E
$E* I           $               Shift
$E*I            $               Shift
$E*E            $               Reduced: E->i
$E $                            Reduced: E->E*E
$E$                             Shift
$E$                             Shift
Accepted;
```

RESULT

The program was executed successfully.

VIVA QUESTIONS

1. What are the disadvantages of operator precedence parser?

2. Examples for bottom up parser.

3. What are the advantages of operator precedence parser?

EXPERIMENT NO: 03

FIRST AND FOLLOW

AIM

To write a program to find First and Follow of any given grammar

THEORY

Syntax Analysis or Parsing is the second phase of a compiler. It checks the syntactical structure of the given input, i.e. whether the given input is in the correct syntax or not. It does so by building a data structure, called a Parse tree or Syntax tree. If the given input string can be produced with the help of the syntax tree (in the derivation process), the input string is found to be in the correct syntax. The Grammar for a Language consists of Production rules. If the compiler would have come to know in advance, that what is the “first character of the string produced when a production rule is applied”, and comparing it to the current character or token in the input string it sees, it can wisely take decision on which production rule to apply.

FIRST(X) for a grammar symbol X is the set of terminals that begin the strings derivable from X.

ALGORITHM

To compute FIRST set:

1. If x is a terminal, then FIRST(x) = { ‘x’ }
2. If x-> Є, is a production rule, then add Є to FIRST(x).
3. If X->Y1 Y2 Y3….Yn is a production,
   a. FIRST(X) = FIRST(Y1)
   b. If FIRST(Y1) contains Є then FIRST(X) = { FIRST(Y1) – Є } U { FIRST(Y2) }
   c. If FIRST (Yi) contains Є for all i = 1 to n, then add Є to FIRST(X).

To compute FOLLOW set:

1. FOLLOW(S) = { $ } // where S is the starting Non-Terminal

2. If A -> pBq is a production, where p, B and q are any grammar symbols, then everything in FIRST(q) except Є is in FOLLOW(B).
3. If A->pB is a production, then everything in FOLLOW(A) is in FOLLOW(B).
4. If A->pBq is a production and FIRST(q) contains Є, then FOLLOW(B) contains { FIRST(q) – Є } U FOLLOW(A)

PROGRAM

```
#include<stdio.h>
#include<string.h>
int i,j,l,m,n=0,o,p,nv,z=0,x=0;
char str[10],temp,temp2[10],temp3[20],*ptr;
struct prod
{
   char lhs[10],rhs[10][10],ft[10],fol[10];
   int n;
}pro[10];
void findter()
{
   int k,t;
   for(k=0;k<n;k++)
   {
      if(temp==pro[k].lhs[0])
      {
         for(t=0;t<pro[k].n;t++)
         {
            if( pro[k].rhs[t][0]<65 || pro[k].rhs[t][0]>90 )
               pro[i].ft[strlen(pro[i].ft)]=pro[k].rhs[t][0];
            else if( pro[k].rhs[t][0]>=65 && pro[k].rhs[t][0]<=90 )
            {
               temp=pro[k].rhs[t][0];
               if(temp=='S')
                  pro[i].ft[strlen(pro[i].ft)]='#';
```

```
                findter();
            }
        }
        break;
    }
  }
}
void findfol()
{
    int k,t,p1,o1,chk;
    char *ptr1;
    for(k=0;k<n;k++)
    {
        chk=0;
        for(t=0;t<pro[k].n;t++)
        {
            ptr1=strchr(pro[k].rhs[t],temp);
            if( ptr1 )
            {
                p1=ptr1-pro[k].rhs[t];
                if(pro[k].rhs[t][p1+1]>=65 && pro[k].rhs[t][p1+1]<=90)
                {
                    for(o1=0;o1<n;o1++)
                        if(pro[o1].lhs[0]==pro[k].rhs[t][p1+1])
                        {
                                strcat(pro[i].fol,pro[o1].ft);
                                chk++;
                        }
                }
                else if(pro[k].rhs[t][p1+1]=='\0')
                {
```

```
                temp=pro[k].lhs[0];
                if(pro[l].rhs[j][p]==temp)
                    continue;
                if(temp=='S')
                    strcat(pro[i].fol,"$");
                findfol();
                chk++;
            }
            else
            {
                pro[i].fol[strlen(pro[i].fol)]=pro[k].rhs[t][p1+1];
                chk++;
            }
        }
    }
    if(chk>0)
        break;
  }
}
int main()
{
    FILE *f;
    //clrscr();
    for(i=0;i<10;i++)
        pro[i].n=0;
       f=fopen("tab5.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++;
}
  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]);
pro[0].ft[0]='#';
for(i=0;i<n;i++)
{
    for(j=0;j<pro[i].n;j++)
    {
        if( pro[i].rhs[j][0]<65 || pro[i].rhs[j][0]>90 )
        {
            pro[i].ft[strlen(pro[i].ft)]=pro[i].rhs[j][0];
        }
        else if( pro[i].rhs[j][0]>=65 && pro[i].rhs[j][0]<=90 )
        {
            temp=pro[i].rhs[j][0];
            if(temp=='S')
                pro[i].ft[strlen(pro[i].ft)]='#';
            findter();
            }
        }
    }
    printf("\n\nFIRST\n");
    for(i=0;i<n;i++)
    {
        printf("\n%s -> ",pro[i].lhs);
        for(j=0;j<strlen(pro[i].ft);j++)
        {
            for(l=j-1;l>=0;l--)
                if(pro[i].ft[l]==pro[i].ft[j])
                    break;
            if(l==-1)
                printf("%c",pro[i].ft[j]);
        }
    }
    for(i=0;i<n;i++)
        temp2[i]=pro[i].lhs[0];
    pro[0].fol[0]='$';
    for(i=0;i<n;i++)
    {
        for(l=0;l<n;l++)
        {
            for(j=0;j<pro[i].n;j++)
            {
                ptr=strchr(pro[l].rhs[j],temp2[i]);
                if( ptr )
                {
                    p=ptr-pro[l].rhs[j];
                    if(pro[l].rhs[j][p+1]>=65 && pro[l].rhs[j][p+1]<=90)
                    {
```

```
                for(o=0;o<n;o++)
                    if(pro[o].lhs[0]==pro[l].rhs[j][p+1])
                            strcat(pro[i].fol,pro[o].ft);
            }
            else if(pro[l].rhs[j][p+1]=='\0')
            {
                temp=pro[l].lhs[0];
                if(pro[l].rhs[j][p]==temp)
                    continue;
                if(temp=='S')
                    strcat(pro[i].fol,"$");
                findfol();
            }
            else
                pro[i].fol[strlen(pro[i].fol)]=pro[l].rhs[j][p+1];
        }
    }
  }
}
printf("\n\nFOLLOW\n");
for(i=0;i<n;i++)
{
    printf("\n%s -> ",pro[i].lhs);
    for(j=0;j<strlen(pro[i].fol);j++)
    {
        for(l=j-1;l>=0;l--)
            if(pro[i].fol[l]==pro[i].fol[j])
                break;
        if(l==-1)
            printf("%c",pro[i].fol[j]);
    }
    }
    printf("\n");
    //getch();
}
```

INPUT

THE GRAMMAR IS AS FOLLOWS

S -> ACB|Cbb|Ba

A -> da|BC

B-> g|Є

C-> h| Є

OUTPUT

FIRST set

FIRST(S) = FIRST(A) U FIRST(B) U FIRST(C) = { d, g, h, Є, b, a}

FIRST(A) = { d } U FIRST(B) = { d, g, h, Є }

FIRST(B) = { g, Є }

FIRST(C) = { h, Є }

FOLLOW Set

FOLLOW(S) = { $ }

FOLLOW(A) = { h, g, $ }

FOLLOW(B) = { a, $, h, g }

FOLLOW(C) = { b, g, $, h }

RESULT

The program was executed successfully.

VIVA QUESTIONS

1. Why do we use first and follow?

2. What are the rules to compute first set?

3. What are the rules to compute follow set?

EXPERIMENT NO: 04

RECURSIVE DESCENT PARSER

AIM:

Construct a recursive descent parser for an expression according to the grammar below

E->TE’

E’->+TE’/$

T->FT’

T’->*FT’/$

F->(E)/ID

THEORY:

A parser is the module of an interpreter or compiler which performs parsing. It takes a sequence of tokens from the lexical analyzer, finds a derivation for the sequence of tokens, and builds a parse tree (also known as a syntax tree) representing the derivation. Parse trees are very important in figuring out the meaning of a program (or part of a program).

Recursive descent is a simple parsing algorithm that is very easy to implement. It is a top-down parsing algorithm because it builds the parse tree from the top (the start symbol) down. The main limitation of recursive descent parsing (and all top-down parsing algorithms in general) is that they only work on grammars with certain properties. For example, if a grammar contains any left recursion, recursive descent parsing doesn't work. The input is read from left to right. It uses procedures for every terminal and non-terminal entity. This parsing technique recursively parses the input to make a parse tree, which may or may not require back-tracking. But the grammar associated with it (if not left factored) cannot avoid back-tracking. A form of recursive-descent parsing that does not require any back-tracking is known as predictive parsing. This parsing technique is regarded recursive as it uses context-free grammar which is recursive in nature.

Back-tracking:

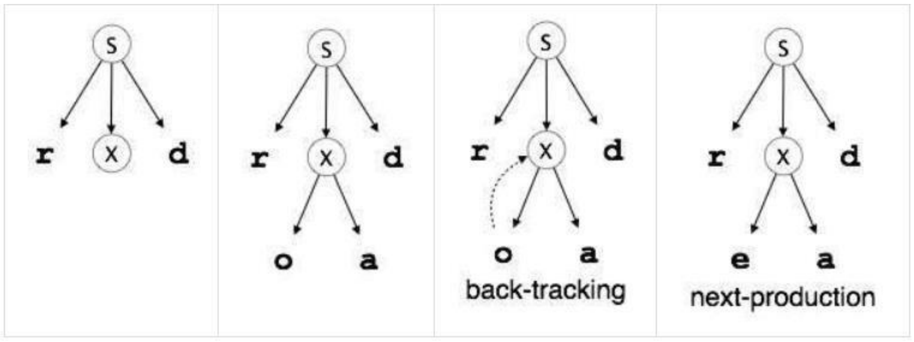
Top- down parsers start from the root node (start symbol) and match the input string against the production rules to replace them (if matched). To understand this, take the following example of CFG:

$S \rightarrow rXd \mid rZd$

$X \rightarrow oa \mid ea$

$Z \rightarrow ai$

For an input string: read, a top-down parser, will start with S from the production rules and will match its yield to the left-most letter of the input, i.e. 'r'. The very production of S ($S \rightarrow rXd$) matches with it. So the top-down parser advances to the next input letter (i.e. 'e'). The parser tries to expand non-terminal 'X' and checks its production from the left ($X \rightarrow oa$). It does not match with the next input symbol. So the top-down parser backtracks to obtain the next production rule of X, ($X \rightarrow ea$). Now the parser matches all the input letters in an ordered manner. The string is accepted.



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.

ALGORITHM

1. Whenever a Non-terminal spend first time then go with the first alternative and compare with the given I/P String
2. If matching doesn't occur then go with the second alternative and compare with the given I/P String.

3. If matching again not found then go with the alternative and so on.
4. Moreover, If matching occurs for at least one alternative, then the I/P string is parsed successfully.

PROGRAM

```
#include<iostream>
#include <string>
#define MAX_ROW 50
using namespace std;
class PARSER {
private:
string input;
string output [MAX_ROW][2];
int tokenptr;
int tokenize(){
string temp;
int row=0;
for(int i=0; i<input.size();i++){
if(isalpha(input[i]) || isdigit(input[i])){
temp.append(1, input[i]);
}
else
{
output[row++][0] = temp;
temp.clear();
if(input[i] == '+' || input[i] == '*') {
temp.append(1, input[i]);
output[row++][0] = temp;
temp.clear();
}
}
```

```
}
output[row++][0] = temp;
return row;
}
int recognize(){
int token_n,j;
token_n = this->tokenize();
//cout<<token_n;
for(int i=0; i<token_n;i++){
string s = output[i][0];
int state = 0;
bool flag = true;
for(j=0; j<s.size() && flag; j++){
switch(state)
{
case 0 :
if(s[j] == 'i')
state = 1;
else if(s[j] == 'e')
state = 3;
else if(s[j] == 'c')
state = 9;
else
flag = false;
break;
case 1 :
if(s[j] == 'n')
state = 7;
else if(s[j] == 'f')
state = 2;
else
```

```
flag = false;
break;
case 2 :
flag = false;
break;
case 3 :
if(s[j] == 'l')
state = 4;
else
flag = false;
break;
case 4 :
if(s[j] == 's')
state = 5;
else
flag = false;
break;
case 5 :
if(s[j] == 'e')
state = 6;
else
flag = false;
break;
case 6 :
flag = false;
break;
case 7 :
if(s[j] == 't')
state = 8;
else
flag = false;
```

```
break;
case 8 :
flag = false;
break;
case 9 :
if(s[j] == 'h')
state = 10;
else
flag = false;
break;
case 10 :
if(s[j] == 'a')
state = 11;
else
flag = false;
break;
case 11:
if(s[j] == 'r')
state = 12;
else
flag = false;
break;
case 12 :
flag = false;
break;
}
}
if(flag == true && (state == 8 || state == 2 || state == 6 ||state == 12 )){
output[i][1] = "keyword";
continue;
}
```

```
flag = true;
state = 0;
for(j=0; j<s.size() && flag; j++){
switch(state)
{
case 0 :
if(isalpha(s[j]))
state = 1;
else
flag = false;
break;
case 1 :
if(isalpha(s[j]) || isdigit(s[j]))
state = 1;
else
flag = false;
break;
}
}
if(flag == true && state == 1){
output[i][1] = "identifier";
continue;
}
flag = true;
state = 0;
for( j=0; j<s.size() && flag; j++){
switch(state)
{
case 0 :
if(s[j] == '+')
state = 1;
else if(s[j] == '*')
state = 2;
else
flag =2;
break;
}
}
if(flag == true && (state == 1 || state == 2)){
if(s[j]=='+')
output[i][1] = "plus";
else if(s[j]=='*')
output[i][1] = "times";
continue;
}
output[i][1] = "invalid";
continue;
}
return token_n;
}
bool E(){
bool v1 = T();
if(v1){
bool v2 = E_();
if(v2)
return true;
else
return false;
}
}
bool E_(){
if(output[tokenptr][1]=="plus"){
```

```
tokenptr++;
if(T()){
if(E_())
return true;
else
return false;
}
else
return false;
}
else
return true; // for epsilon production
}
bool T(){
bool v1 = F();
if(v1){
bool v2 = T_();
if(v2)
return true;
else
return false;
}
}
bool T_(){
if(output[tokenptr][1]=="times"){
tokenptr++;
if(F()){
if(T_())
return true;
else
return false;
}
else
return false;
}
else
return true; // for epsilon production
}
bool F(){
if(output[tokenptr][1]=="identifier"){
tokenptr++;
return true;
}
else
return false;
}
public:
void parse(){
int table_dim = this->recognize();
output[table_dim][0] = "$";
output[table_dim][0] = "dollar"; // to add dollar sign to the end of the table. To make sure the
E_() and T_() return true
if(E())
cout<<"The string is parsed succesfully\n";
else
cout<<"The string is not parsed\n";
}
PARSER(string inp_string) {
input.clear();
for(int i=0 ; i<MAX_ROW; i++)
for(int j=0; j<2; j++)
output[i][j].clear();
```

```
input = inp_string;
tokenptr = 0;
}
};
int main(){
string inp_string;
cout<<"Enter the input string \n";
cin>>inp_string;
PARSER obj(inp_string);
obj.parse();
return 0;
}
```

OUTPUT

```
Enter Expression : a+b*c
String accepted
```

RESULT

The program was executed successfully.

VIVA QUESTIONS

1. What is predictive parser?

2. Which are the different types of LR parsers?

3. What do L and R stand for in LR parser?

EXPERIMENT NO: 05

INTERMEDIATE CODE GENERATION

AIM

Implement Intermediate code generation for arithmetic expressions.

THEORY

In the analysis-synthesis model of a compiler, the front end of a compiler translates a source program into an independent intermediate code, then the back end of the compiler uses this intermediate code to generate the target code (which can be understood by the machine). The benefits of using machine independent intermediate code are:

- Because of the machine independent intermediate code, portability will be enhanced. For ex, suppose, if a compiler translates the source language to its target machine language without having the option for generating intermediate code, then for each new machine, a full native compiler is required. Because, obviously, there were some modifications in the compiler itself according to the machine specifications.
- Retargeting is facilitated
- It is easier to apply source code modification to improve the performance of source code by optimizing the intermediate code.

Intermediate Representation:

Intermediate codes can be represented in a variety of ways and they have their own benefits.

- High Level IR - High-level intermediate code representation is very close to the source language itself. They can be easily generated from the source code and we can easily apply code modifications to enhance performance. But for target machine optimization, it is less preferred.
- Low Level IR - This one is close to the target machine, which makes it suitable for register and memory allocation, instruction set selection, etc. It is good for machine-dependent optimizations.

Intermediate code can be either language specific (e.g., Byte Code for Java) or language independent (three-address code).

Three-Address Code:

Intermediate code generator receives input from its predecessor phase, semantic analyzer, in the form of an annotated syntax tree. That syntax tree then can be converted into a linear representation, e.g., postfix notation. Intermediate code tends to be machine independent code. Therefore, code generator assumes to have unlimited number of memory storage (register) to generate code.

For example:

a = b + c * d;

The intermediate code generator will try to divide this expression into sub-expressions and then generate the corresponding code.

r1 = c * d;

r2 = b + r1;

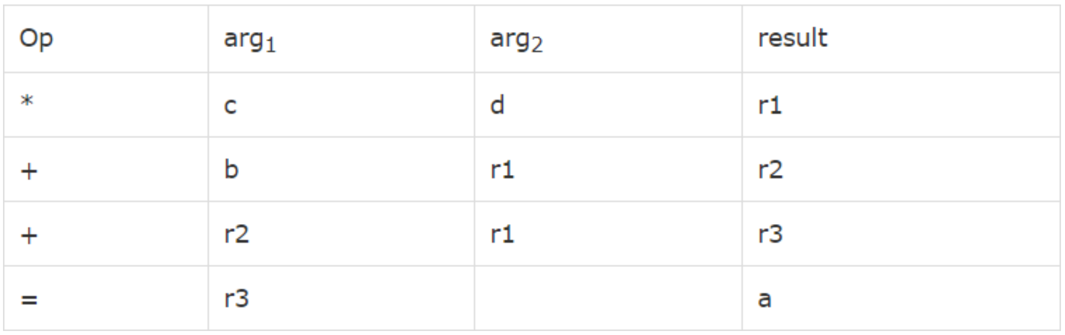
a = r2

r being used as registers in the target program.

A three-address code has at most three address locations to calculate the expression. A three-address code can be represented in two forms : quadruples and triples.

Quadruples:

Each instruction in quadruples presentation is divided into four fields: operator, arg1, arg2, and result. The above example is represented below in quadruples format:



Op	arg_1	arg_2	result
*	c	d	r1
+	b	r1	r2
+	r2	r1	r3
=	r3		a

Triples:

Each instruction in triples presentation has three fields : op, arg1, and arg2.The results of respective sub-expressions are denoted by the position of expression. Triples represent similarity with DAG and syntax tree. They are equivalent to DAG while representing expressions.

Op	arg_1	arg_2
*	c	d
+	b	(0)
+	(1)	(0)
=	(2)	

Triples face the problem of code immovability while optimizations, as the results are positional and changing the order or position of an expression may cause problems.

Indirect Triples:

This representation is an enhancement over triples representation. It uses pointers instead of position to store results. This enables the optimizers to freely re-position the sub-expression to produce an optimized code.

ALGORITHM

1. Read the input string.
2. Convert it into postfix form.
3. Pop two operands and insert into three address code table.
4. Mark result of each row with line number in three address code table
5. Return

PROGRAM

```
#include<iostream>
#include<string>
#include<sstream>
using namespace std;
#define MAXROWS 6
#define MAX 20
class Stack{
int top; // Stack Top
```

```
public:
string elements[MAX];    //Maximum size of Stack
Stack()
{
top = -1;
}
void push(string x)
{
elements[++top]=x;
}
string pop()
{
if(!isEmpty())
return elements[top--];
return NULL;
}
bool isEmpty()
{
if(top==-1)
return true;
else
return false;
}
};
class ICG{
private:
string input; //The input string
string IC[MAXROWS][4]; // Output in Quadruple format
string output[MAX];
int counter;
public:
```

```
ICG(string str)
{
this->input=str;
this->counter=0;
}

void GenerateIC()
{
string out_string;
stringstream ss;
int tempCounter=1;
int n=tokenize();
Stack st;
string str1,str2;
for(int i=0; i<n ; i++)
{
if(output[i] == "+" || output[i] == "*")
{
str2=st.pop();
str1=st.pop();
IC[counter][0]=output[i] ;
IC[counter][1]=str2 ;
IC[counter][2]=str1 ;
ss.str("");
ss << tempCounter;
out_string.clear();
out_string = ss.str();
str1.clear();
str1 = "Temp"+out_string ;
IC[counter][3]=str1;
counter++;
```

```
tempCounter++;
st.push(str1);
}
else
{
st.push(output[i]);
}
}
printICG();
}
void printICG()
{
cout<<endl<<"The Intermediate Representation is : "<<endl;
for(int i=0; i<counter; i++)
{
cout<<endl<<IC[i][0]<<"\t"<<IC[i][1]<<"\t"<<IC[i][2]<<"\t"<<IC[i][3];
}
}
int tokenize()
{
string temp;
int row=0;
/*for(int i=0;i<input.size();i++)
{
output[row++]= input[i];
}*/
for(int i=0; i<input.size();i++)
{
if(input[i]!=' ')
{
temp.append(1, input[i]);
```

```
}
else
{
output[row++]= temp;
temp.clear();
}
}
output[row++] = temp;
return row;
}
};
main()
{
ICG cg("ab b1 c d2 * * + ");
cg.GenerateIC();
cout<<endl;
}
```

OUTPUT

```
Enter expression : a+b*c
Postfix Expression : abc*+
Quadruple code:
```

*	B	c	T1
+	A	T1	T2

Triplet code:

2	B	c
+	A	1

RESULT

The program was executed successfully.

VIVA QUESTIONS

1. What are the benefits of using machine independent intermediate code?
2. What the commonly used intermediate code representation?
3. What are the different types of Three-Address Codes?

EXPERIMENT NO: 06

THREE ADDRESS CODE AND PRODUCE 8086 INSTRUCTION

AIM

To write a program to implement back end of the compiler which takes the three address code and produce the 8086 assembly language instructions that can be assembled and run using an 8086 assembler.

THEORY

A compiler is a computer program that implements a programming language specification to "translate" programs, usually as a set of files which constitute the source code written in source language, into their equivalent machine readable instructions(the target language, often having a binary form known as object code). This translation process is called compilation.

BACK END :

- Some local optimization
- Register allocation
- Peep-hole optimization
- Code generation
- Instruction scheduling

The main phases of the back end include the following:

Analysis: This is the gathering of program information from the intermediate representation derived from the input; data-flow analysis is used to build use-define chains, together with dependence analysis, alias analysis, pointer analysis, escape analysis etc.

Optimization: The intermediate language representation is transformed into functionally equivalent but faster (or smaller) forms. Popular optimizations are expansion, dead, constant, propagation, loop transformation, register allocation and even automatic parallelization.

Code generation: The transformed language is translated into the output language, usually the native machine language of the system. This involves resource and storage decisions, such as deciding which variables to fit into registers and memory and the selection and scheduling of appropriate machine instructions along with their associated modes. Debug data may also need to be generated to facilitate debugging.

ALGORITHM

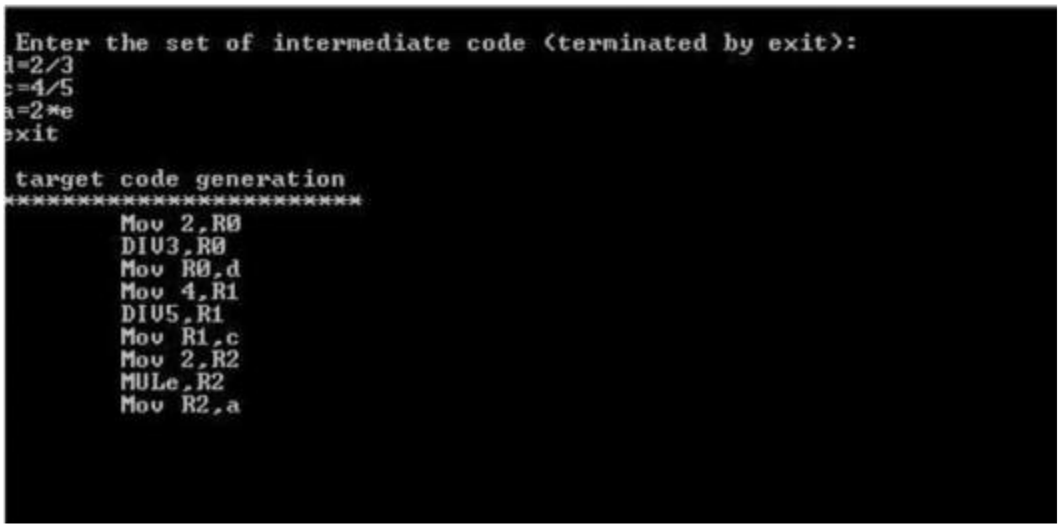
1. Start the program
2. Open the source file and store the contents as quadruples.
3. Check for operators, in quadruples, if it is an arithmetic operator generator it or if assignment operator generates it, else perform unary minus on register C.
4. Write the generated code into output definition of the file in outp.c
5. Print the output
6. Stop the program.

PROGRAM

```
#include<stdio.h>
//#include<conio.h>
#include<string.h>
void main()
{
 char icode[10][30],str[20],opr[10];
int i=0;
//clrscr();
printf("\n Enter the set of intermediate code (terminated by exit):\n");
Do
{
 scanf("%s",icode[i]);
} while(strcmp(icode[i++],"exit")!=0);
printf("\n target code generation");
printf("\n************************");
i=0;
Do
{
 strcpy(str,icode[i]);
 switch(str[3])
 {
 case '+':
 strcpy(opr,"ADD");
 break;
 case '-':
 strcpy(opr,"SUB");
 break;
 case '*':
 strcpy(opr,"MUL");
 break;
 case '/':
 strcpy(opr,"DIV");
 break;
 }
 printf("\n\tMov %c,R%d",str[2],i);

 printf("\n\t%s%c,R%d",opr,str[4],i);
 printf("\n\tMov R%d,%c",i,str[0]);
 }while(strcmp(icode[++i],"exit")!=0);
//getch();
}
```

OUTPUT



```
 Enter the set of intermediate code (terminated by exit):
d=2/3
c=4/5
a=2*e
exit

 target code generation
************************
        Mov 2,R0
        DIV3,R0
        Mov R0,d
        Mov 4,R1
        DIV5,R1
        Mov R1,c
        Mov 2,R2
        MULe,R2
        Mov R2,a
```

RESULT

The program was executed successfully.

VIVA QUESTIONS

1. What are the register allocation issues that can arise in code generation phase?

2. How is instruction selection done in code generation phase?

3. What are the approaches to code generation issues?

EXPERIMENT NO: 07

LOOP UNROLLING

AIM:

Write a program to perform loop unrolling.

THEORY:

Loop unrolling, also known as loop unwinding, is a loop transformation technique that attempts to optimize a program's execution speed at the expense of its binary size, which is an approach known as space–time tradeoff. The transformation can be undertaken manually by the programmer or by an optimizing compiler.

The goal of loop unwinding is to increase a program's speed by reducing or eliminating instructions that control the loop, such as pointer arithmetic and "end of loop" tests on each iteration; reducing branch penalties; as well as hiding latencies including the delay in reading data from memory. To eliminate this computational overhead, loops can be re-written as a repeated sequence of similar independent statements.

Loop unrolling is also part of certain formal verification techniques, in particular bounded model checking.

Advantages:

The overhead in "tight" loops often consists of instructions to increment a pointer or index to the next element in an array (pointer arithmetic), as well as "end of loop" tests. If an optimizing compiler or assembler is able to pre-calculate offsets to each individually referenced array variable, these can be built into the machine code instructions directly, therefore requiring no additional arithmetic operations at run time.

- Significant gains can be realized if the reduction in executed instructions compensates for any performance reduction caused by any increase in the size of the program.
- Branch penalty is minimized.
- If the statements in the loop are independent of each other (i.e. where statements that occur earlier in the loop do not affect statements that follow them), the statements can potentially be executed in parallel.

- Can be implemented dynamically if the number of array elements is unknown at compile time (as in Duff's device).

Optimizing compilers will sometimes perform the unrolling automatically, or upon request.

Disadvantages:

- Increased program code size, which can be undesirable, particularly for embedded applications. Can also cause an increase in instruction cache misses, which may adversely affect performance.
- Unless performed transparently by an optimizing compiler, the code may become less readable.
- If the code in the body of the loop involves function calls, it may not be possible to combine unrolling with inlining, since the increase in code size might be excessive. Thus there can be a trade-off between the two optimizations.
- Possible increased register usage in a single iteration to store temporary variables[dubious – discuss], which may reduce performance, though much will depend on possible optimizations.
- Apart from very small and simple codes, unrolled loops that contain branches are even slower than recursions.

Example:

Normal loop	After loop unrolling
`int x; for (x = 0; x < 100; x++) { delete(x); }`	`int x; for (x = 0; x < 100; x += 5 ) { delete(x); delete(x + 1); delete(x + 2); delete(x + 3); delete(x + 4); }`

ALGORITHM

```
Step 1: start
Step 2: declare variables word ,len=0, i=0,c,y.
Step 3: open sample.c file read mode
Step 4: read file word by word. For each word do step 5 to
Step 5: assign the length of word to len
Step 6: if
```

PROGRAM

```
#include<stdio.h>
#define BLOCKSIZE (8)
void main(void)
{
int i = 0;
int limit = 33; /* could be anything */
int blocklimit;

/* The limit may not be divisible by BLOCKSIZE,
 * go as near as we can first, then tidy up.
 */
blocklimit = (limit / BLOCKSIZE) * BLOCKSIZE;

/* unroll the loop in blocks of 8 */
while( i < blocklimit )
{
    printf("process(%d)\n", i);
    printf("process(%d)\n", i+1);
    printf("process(%d)\n", i+2);
    printf("process(%d)\n", i+3);
    printf("process(%d)\n", i+4);
    printf("process(%d)\n", i+5);
    printf("process(%d)\n", i+6);
    printf("process(%d)\n", i+7);
    /* update the counter */
    i += 8;
}
/*
 * There may be some left to do.
 * This could be done as a simple for() loop,
 * but a switch is faster (and more interesting)
 */
if( i < limit )
{
    /* Jump into the case at the place that will allow
     * us to finish off the appropriate number of items.
     */
    switch( limit - i )
    {
        case 7 : printf("process(%d)\n", i); i++;
        case 6 : printf("process(%d)\n", i); i++;
        case 5 : printf("process(%d)\n", i); i++;
        case 4 : printf("process(%d)\n", i); i++;
        case 3 : printf("process(%d)\n", i); i++;
        case 2 : printf("process(%d)\n", i); i++;
        case 1 : printf("process(%d)\n", i);
    }
}
}
```

RESULT

The program was executed successfully.

VIVA QUESTIONS

1. What are the types of Code Optimization?
2. What are the different ways of code optimization?
3. What are the objectives of code optimization?

EXPERIMENT NO: 08

LEXICAL ANALYZER USING LEX TOOL

AIM

Implementation of lexical analyzer using Lex Tool to recognize tokens such as identifiers, keywords, operators.

THEORY

Lexical analysis is the first phase of a compiler. It is also known as scanner. It takes the modified source code from language preprocessors that are written in the form of sentences. The lexical analyzer breaks these syntaxes into a series of tokens, by removing any whitespace or comments in the source code. If the lexical analyzer finds a token invalid, it generates an error.

The lexical analyzer works closely with the syntax analyzer. It reads character streams from the source code, checks for legal tokens, and passes the data to the syntax analyzer when it demands. Lexical analyzer removes white spaces and comments from the program. It also provides help in generating error message by providing row number and column number.

Lexemes are said to be a sequence of characters (alphanumeric) in a token. There are some predefined rules for every lexeme to be identified as a valid token. These rules are defined by grammar rules, by means of a pattern. A pattern explains what can be a token, and these patterns are defined by means of regular expressions. In programming language, keywords, constants, identifiers, strings, numbers, operators and punctuations symbols can be considered as tokens.

For example, in C language, the variable declaration line int value = 100; contains the tokens: int (keyword), value (identifier), = (operator), 100 (constant) and ; (symbol).

PROGRAM

```
#include<iostream>
#include <string>
```

```
#define MAX_ROW 50
using namespace std;
class LEXICALANALYZER {
        private:
                string input;
                string output [MAX_ROW][2];
                int tokenize(){
                        string temp;
                        int row=0;
                        for(int i=0; i<input.size();i++){
                                if(isalpha(input[i]) || isdigit(input[i])){
                                        temp.append(1, input[i]);
                                }
                                else
                                {
                                        output[row++][0] = temp;
                                        temp.clear();
                                        if(input[i] == '+' || input[i] == '*') {

                                        temp.append(1, input[i]);
                                        output[row++][0] = temp;
                                        temp.clear();
                                        }
                                }
                        }
                        output[row++][0] = temp;
                        return row;
                }
        public:
                void recognize(){
                        int token_n;
```

```
token_n = this->tokenize();
//cout<<token_n;
for(int i=0; i<token_n;i++){
        string s = output[i][0];
        int state = 0;
        bool flag = true;
        for(int j=0; j<s.size() && flag; j++){
                switch(state)
                {
                        case 0 :
                                if(s[j] == 'i')
                                        state = 1;
                                else if(s[j] == 'e')
                                        state = 3;
                                else if(s[j] == 'c')
                                         state = 9;
                                else
                                        flag = false;
             break;
                        case 1 :
                                if(s[j] == 'n')
                                        state = 7;
                                else if(s[j] == 'f')
                                        state = 2;
                                else
                                        flag = false;
                                break;
                        case 2 :
                                flag = false;
                                break;
                        case 3 :
```

```
                        if(s[j] == 'l')
                                state = 4;
                        else
                                flag = false;
                        break;
                case 4 :
                        if(s[j] == 's')
                                state = 5;
                        else
                                flag = false;
                        break;
                case 5 :
                        if(s[j] == 'e')
                                state = 6;
                        else
                                flag = false;
                        break;
                case 6 :
                        flag = false;
                        break;
                case 7 :
                        if(s[j] == 't')
                                state = 8;
                        else
                                flag = false;
                        break;
                case 8 :
                        flag = false;
                        break;
                case 9 :
                        if(s[j] == 'h')
```

```
                                        state = 10;
                                    else
                                        flag = false;

                                    break;
                                case 10 :
                                    if(s[j] == 'a')
                                        state = 11;
                                    else
                                        flag = false;
                                   break;
                                case 11:
                                    if(s[j] == 'r')
                                        state = 12;
                                    else
                                        flag = false;
                                    break;
                                case 12 :
                                    flag = false;
                              break;
                        }
                    }
            if(flag == true && (state == 8 || state == 2 || state == 6 ||state == 12 )){
                        output[i][1] = "keyword";
                        continue;
                    }
                    flag = true;
                    state = 0;
                    for(int j=0; j<s.size() && flag; j++){
                        switch(state)
                        {
```

```
                    case 0 :
                        if(isalpha(s[j]))
                            state = 1;
                        else
                            flag = false;
                         break;
                    case 1 :
                        if(isalpha(s[j]) || isdigit(s[j]))
                            state = 1;
                        else
                            flag = false;
                        break;
                }
            }
            if(flag == true && state == 1){
                output[i][1] = "identifier";
                continue;
            }
            flag = true;
            state = 0;
        for(int j=0; j<s.size() && flag; j++){
                switch(state)
                {
                    case 0 :
                        if(s[j] == '+')
                            state = 1;
                        else if(s[j] == '*')
                            state = 2;
```

```
else break;


flag =2;

                              }
                                        }
                                        if(flag == true && (state == 1 || state == 2)){
                                                  output[i][1] = "operator";
                                                  continue;
                                        }

                                        output[i][1] = "invalid";
                                        continue;

                              }
                              cout<<"------------------------"<<endl;
                              for(int i=0; i<token_n;i++){
                                        cout<<output[i][0]<<"  "<<output[i][1]<<endl;

                              }
                    cout<<"------------------------"<<endl;
          }

                    LEXICALANALYZER(string inp_string) {
                              input.clear();
                              for(int i=0 ; i<MAX_ROW; i++)
                                        for(int j=0; j<2; j++)
                                                  output[i][j].clear();
                              input = inp_string;
                    }
```

```
};
int main(){
        string inp_string;
        cout<<"Enter the input string \n";
        cin>>inp_string; LEXICALANALYZER obj(inp_string); obj.recognize();
        return 0;
}
```

VIVA QUESTIONS

1. What is the Structure of a Lex file?

2. How can we use Lex with other programming tools?

3. What is the function of utility ‘make’?

EXPERIMENT NO: 09

YACC

AIM

Generate YACC specification for the implementation of calculator that perform +,-,* and /

THEORY

YACC (Yet Another Compiler-Compiler) is a computer program for the Unix operating system developed by Stephen C. Johnson. It is a Look Ahead Left-to-Right (LALR) parser generator, generating a parser, the part of a compiler that tries to make syntactic sense of the source code, specifically a LALR parser, based on an analytic grammar written in a notation similar to Backus–Naur Form (BNF). YACC is supplied as a standard utility on BSD and AT&T Unix. GNU-based Linux distributions include Bison, a forward-compatible YACC replacement.

The input to YACC is a grammar with snippets of C code (called "actions") attached to its rules. Its output is a shift-reduce parser in C that executes the C snippets associated with each rule as soon as the rule is recognized. Typical actions involve the construction of parse trees. Using an example from Johnson, if the call node(label, left, right) constructs a binary parse tree node with the specified label and children, then the rule.

```
expr : expr '+' expr { $$ = node('+', $1, $3); }
```

Recognizes summation expressions and constructs nodes for them. The special identifiers $$, $1 and $3 refer to items on the parser's stack.

YACC produces only a parser (phrase analyzer); for full syntactic analysis this requires an external lexical analyzer to perform the first tokenization stage (word analysis), which is then followed by the parsing stage proper. Lexical analyzer generators, such as Lex or Flex are widely available. The IEEE POSIX P1003.2 standard defines the functionality and requirements for both Lex and YACC.

ALGORITHM

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

Declarations %% translation rules %% 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. v. 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

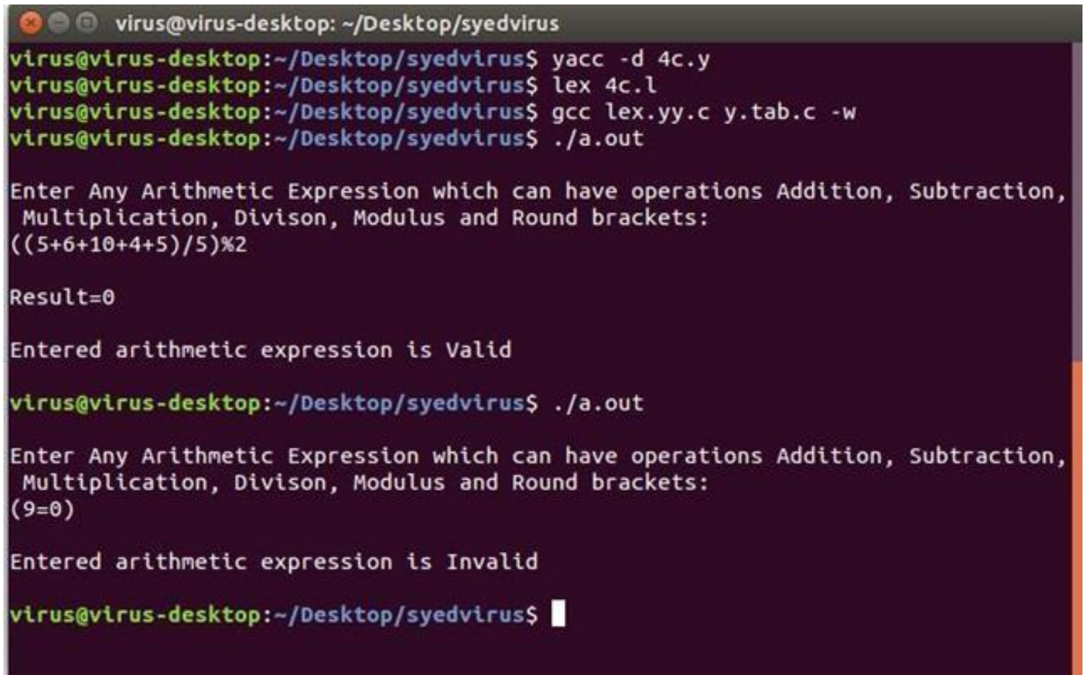
program.

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.

EXPECTED OUTPUT



```
virus@virus-desktop: ~/Desktop/syedvirus
virus@virus-desktop:~/Desktop/syedvirus$ yacc -d 4c.y
virus@virus-desktop:~/Desktop/syedvirus$ lex 4c.l
virus@virus-desktop:~/Desktop/syedvirus$ gcc lex.yy.c y.tab.c -w
virus@virus-desktop:~/Desktop/syedvirus$ ./a.out

Enter Any Arithmetic Expression which can have operations Addition, Subtraction,
 Multiplication, Divison, Modulus and Round brackets:
((5+6+10+4+5)/5)%2

Result=0

Entered arithmetic expression is Valid

virus@virus-desktop:~/Desktop/syedvirus$ ./a.out

Enter Any Arithmetic Expression which can have operations Addition, Subtraction,
 Multiplication, Divison, Modulus and Round brackets:
(9=0)

Entered arithmetic expression is Invalid

virus@virus-desktop:~/Desktop/syedvirus$ 
```

VIVA QUESTIONS

1. What is the basic operational sequence of YACC?

2. What does YACC stand for?

3. Which grammar can be compiled using YACC?

EXPERIMENT NO: 10

ε CLOSURE

AIM:

Write a program to find ε closure of all states of any given NFA with ε transition.

THEORY:

In automata theory, a finite state machine is called a deterministic finite automaton (DFA), if each of its transitions is uniquely determined by its source state and input symbol, and reading an input symbol is required for each state transition.

A nondeterministic finite automaton (NFA), or nondeterministic finite state machine, does not need to obey these restrictions. In particular, every DFA is also an NFA. Sometimes the term NFA is used in a narrower sense, referring to a NFA that is not a DFA, but not in this article. Using the subset construction algorithm, each NFA can be translated to an equivalent DFA, i.e. a DFA recognizing the same formal language. Like DFAs, NFAs only recognize regular languages.

NFAs were introduced in 1959 by Michael O. Rabin and Dana Scott,[2] who also showed their equivalence to DFAs. NFAs are used in the implementation of regular expressions: Thompson's construction is an algorithm for compiling a regular expression to an NFA that can efficiently perform pattern matching on strings.

NFAs have been generalized in multiple ways, e.g., nondeterministic finite automaton with ε-moves, finite state transducers, pushdown automata, alternating automata, ω-automata, and probabilistic automata. Besides the DFAs, other known special cases of NFAs are unambiguous finite automata (UFA) and self-verifying finite automata (SVFA).

Given a set M of NFA states, we define ϵ-closure(M) to be the least (in terms of the subset relation) solution to the set equation ϵ-closure(M) = M ∪ { t| s ∈ ϵ-closure(M) and sϵt∈T}, where T is the set of transitions in the NFA.

ALGORITHM

```
# Input: a set of states, S
# Output: epsilon_closure(S)
    Stack P.addAll(S) #a stack containing all states in S
    Set C.addAll(S) #the closure initially contains the states in S
    while ! P.empty() do
         s = P.pop()
         for r in m(s, epsilon) do
            # m(s, epsilon) is a set of states
            if r not in C then
                P.push(r)
                C.add(r)
            end if
        end for
    end while
    return C
```

EXPECTED OUTPUT

(a+b)*

A	0	EMPTY	EMPTY	{1 7}
B	1	EMPTY	EMPTY	{2 4}
C	2	{3}	EMPTY	EMPTY
D	3	EMPTY	EMPTY	{6}
E	4	EMPTY	{5}	EMPTY
F	5	EMPTY	EMPTY	6
G	6	EMPTY	EMPTY	{1 7}
H	7	EMPTY	EMPTY	EMPTY

A(0)= {A(0)B(1),H(7),C(2),E(4)}

B(1)={B(1),C(2),E(4)}

C(2)= {C(2)}

D(3)= {B(1),C(2),D(3),G(6),E(4),H(7)}

E(4)={E(4)}

F(5)= {B(1),C(2),F(5),E(4),G(6),H(7)}

G(6)= {B(1),C(2),E(4),G(6),H(7)}

H(7)={H(7)}

VIVA QUESTIONS

1. What is epsilon closure?
2. What are the steps to Convert NFA with ε-move to DFA?
3. How is NFA with ε-move represented?

EXPERIMENT NO: 11

CONVERSION OF NFA TO DFA

AIM

To convert NFA to DFA

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, $\sum$, δ, q_0, 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 $\delta: Q \times \sum \rightarrow Q$
- q_0 is the initial state from where any input is processed ($q_0 \in Q$).
- 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.

In NDFA, for a particular input symbol, the machine can move to any combination of the states in the machine. In other words, the exact state to which the machine moves cannot be determined. Hence, it is called Non-deterministic Automaton. As it has finite number of states, the machine is called Non-deterministic Finite Machine or Non-deterministic Finite Automaton.

Formal Definition of an NDFA

An NDFA can be represented by a 5-tuple (Q, $\sum$, δ, q_0, F) where −

- Q is a finite set of states.

- $\sum$ is a finite set of symbols called the alphabets.
- δ is the transition function where $\delta: Q \times \sum \rightarrow 2^Q$

  (Here the power set of Q (2^Q) has been taken because in case of NDFA, from a state, transition can occur to any combination of Q states)
- q_0 is the initial state from where any input is processed ($q_0 \in Q$).
- F is a set of final state/states of Q ($F \subseteq Q$).

Graphical Representation of an NDFA: (same as DFA)

An NDFA 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.

Input − An NDFA

Output − An equivalent DFA

Step 1 − Create state table from the given NDFA.

Step 2 − Create a blank state table under possible input alphabets for the equivalent DFA.

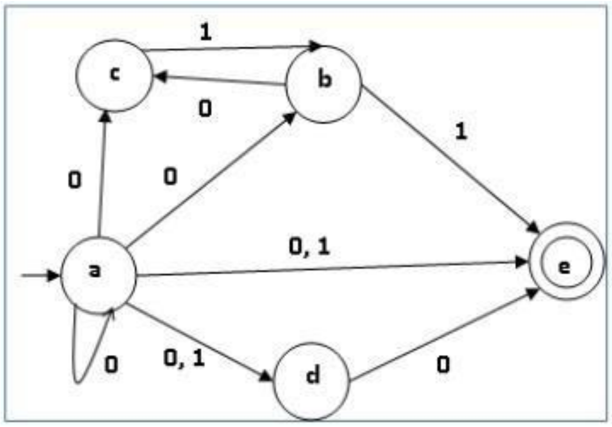
Step 3 − Mark the start state of the DFA by q0 (Same as the NDFA).

Step 4 − Find out the combination of States $\{Q_0, Q_1, \ldots , Q_n\}$ for each possible input alphabet.

Step 5 − Each time we generate a new DFA state under the input alphabet columns, we have to apply step 4 again, otherwise go to step 6.

Step 6 − The states which contain any of the final states of the NDFA are the final states of the equivalent DFA.

Let us consider the NDFA shown in the figure below.

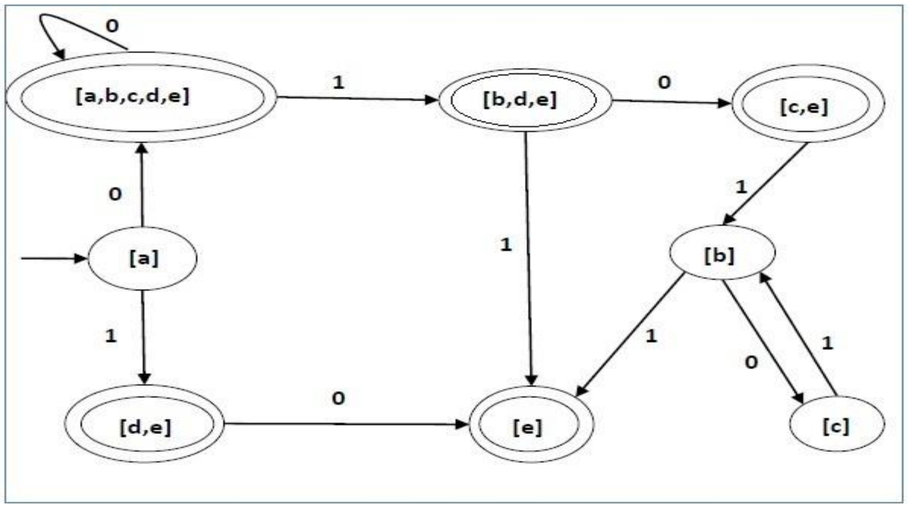


Q	δ(q,0)	δ(q,1)
A	{a,b,c,d,e}	{d,e}
B	{c}	{e}
C	∅	{b}
D	{e}	∅
E	∅	∅

Using the above algorithm, we find its equivalent DFA. The state table of the DFA is shown in below.

Q	δ(q,0)	δ(q,1)
[a]	[a,b,c,d,e]	[d,e]
[a,b,c,d,e]	[a,b,c,d,e]	[b,d,e]
[d,e]	[e]	∅
[b,d,e]	[c,e]	[e]
[e]	∅	∅
[c, e]	∅	[b]
[b]	[c]	[e]
[c]	∅	[b]

The state diagram of the DFA is as follows −



ALGORITHM

Input − An NDFA

Output − An equivalent DFA

Step 1 − Create state table from the given NDFA.

Step 2 − Create a blank state table under possible input alphabets for the equivalent DFA.

Step 3 − Mark the start state of the DFA by q0 (Same as the NDFA).

Step 4 − Find out the combination of States {Q0, Q1,... , Qn} for each possible input alphabet.

Step 5 − Each time we generate a new DFA state under the input alphabet columns, we have to apply step 4 again, otherwise go to step 6.

Step 6 − The states which contain any of the final states of the NDFA are the final states of the equivalent DFA.

EXPECTED OUTPUT

```
Input : 6
        2
        FC - BF
        - C -
        - - D
        E A -
        A - BF
        - - -
Output :
 STATES OF NFA :        A, B, C, D, E, F
 GIVEN SYMBOLS FOR NFA:     0, 1, eps
 NFA STATE TRANSITION TABLE
STATES    |0    |1    eps
--------+------------------------------------
A    |FC     |-     |BF
B    |-     |C     |-
C    |-     |-     |D
D    |E     |A     |-
E    |A     |-     |BF
F    |-     |-     |-

 e-Closure (A) :    ABF
 e-Closure (B) :    B
 e-Closure (C) :    CD
 e-Closure (D) :    D
 e-Closure (E) :    BEF
 e-Closure (F) :    F
*******************************************************
         DFA TRANSITION STATE TABLE
 STATES OF DFA :        ABF, CDF, CD, BEF,
 GIVEN SYMBOLS FOR DFA:     0, 1,

STATES    |0    |1
--------+-----------------------
ABF    |CDF     |CD
CDF    |BEF     |ABF
CD    |BEF     |ABF
```

```
BEF    |ABF     |CD
```

VIVA QUESTIONS

1. How is NFA represented?
2. How is NFA with ε moves represented?
3. How is DFA seen as a special kind of NFA?

VIVA QUESTIONS

1. What is Translator?
2. What is source language?
3. What is target language?
4. What is compiler?
5. What is high-level translator?
6. What is disassemble?
7. What is decompiler?
8. What is source program?
9. What is object program?
10. What is implementation language?
11. What is tombstone diagram?
12. What is cross compiler?
13. What is Portable program?
14. What is interpretive compiler?
15. What is a compiler?
16. What is the difference between compilers & interpreters?
17. What is Language processor?
18. What is Symbol table?
19. Explain Different Phases Of A Compiler With An Example?
20. What is Risc?
21. What is CISC?
22. Application of compiler technology?
23. What is Lexical Analyzer?
24. What is Tokens, Patterns, Lexemes?
25. What are Regular Expressions?
26. What are Regular Definitions?
27. What is deterministic Finite Automata (DFA)?
28. What is Nondeterministic Finite Automata (NFA)?
29. What is Synthesized Attributes?
30. What is Syntax-Directed Definitions?

31. What is Parsing?
32. What is Predictive Parsing?
33. What is Left Recursion?
34. Issues in the Design of Code generator?
35. What is Lex?
36. What is Lexical error?
37. Types of parsers?
38. What does LL and LR mean?
39. What is compile-and-Go Loaders?
40. What is Absolute Loader?
41. What is Relocation loader?
42. What is Dynamic Linking?
43. What is Code Generation?
44. What is code optimization?
45. What Is A Compiler?
46. List The Sub Parts Or Phases Of Analysis Part?
47. List The Various Phases Of A Compiler?
48. Write A Regular Expression For An Identifier?
49. Mention The Basic Issues In Parsing?
50. Define A Context Free Grammar?
51. Mention The Types Of LR Parser?
52. What Are The Problems With Top Down Parsing?
53. What is YACC?