Course Objectives:

To enlighten the student with knowledge base in compiler design and its applications

Course Outcomes:

The end of the course student will be able to x Design simple lexical analyzers x Determine predictive parsing table for a CFG x Apply Lex and Yacc tools x Examine LR parser and generating SLR Parsing table x Relate Intermediate code generation for subset C language

List of Experiments:

1. Write a C program to identify different types of Tokens in a given Program.
2. Write a Lex Program to implement a Lexical Analyzer using Lex tool.
3. Write a C program to Simulate Lexical Analyzer to validating a given input String.
4. Write a C program to implement the Brute force technique of Top down Parsing.
5. Write a C program to implement a Recursive Descent Parser.
6. Write C program to compute the First and Follow Sets for the given Grammar.
7. Write a C program for eliminating the left recursion and left factoring of a given grammar
8. Write a C program to check the validity of input string using Predictive Parser.
9. Write a C program for implementation of LR parsing algorithm to accept a given input string.
10. Write a C program for implementation of a Shift Reduce Parser using Stack Data Structure to accept a given input string of a given grammar.
11. Simulate the calculator using LEX and YACC tool.
12. Generate YACC specification for a few syntactic categories.
13. Write a C program for generating the three address code of a given expression/statement.
14. Write a C program for implementation of a Code Generation Algorithm of a given expression/statement.

Text Books & Reference Books : 1. Compilers: Principles, Techniques and Tools, Second Edition, Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffry D. Ullman, Pearson Publishers, 2007. 2. John R Levine, Tony Mason, Doug Brown, "Lex and Yacc", Orielly, 2nd Edition, 2009.

Experiment -1

AIM: Write a C program to identify different types of Tokens in a given Program

Lexical analyzer reads the characters from source code and converts it into tokens.

Different tokens or lexemes are:

- Keywords
- Identifiers
- Operators
- Constants

Take below example.

c = a + b;

After lexical analysis a symbol table is generated as given below.

Token	Type
c	identifier
=	operator
a	identifier
+	operator
b	identifier
;	separator

Now below I have given implementation of very simple lexical analyzer which reads source code from file and then generate tokens.

```
#include <stdbool.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
bool  isValidDelimiter(char ch) {
  if (ch == ' ' || ch == '+' || ch == '-' || ch == '*' ||
  ch == '/' || ch == ',' || ch == ';' || ch == '>' ||
  ch == '<' || ch == '=' || ch == '(' || ch == ')' ||
  ch == '[' || ch == ']' || ch == '{' || ch == '}')
  return (true);
  return (false);
}
bool  isValidOperator(char ch){
  if (ch == '+' || ch == '-' || ch == '*' ||
  ch == '/' || ch == '>' || ch == '<' ||
  ch == '=')
  return (true);
  return (false);
}
// Returns 'true' if the string is a VALID IDENTIFIER.
bool isvalidIdentifier(char* str){
  if (str[0] == '0' || str[0] == '1' || str[0] == '2' ||
  str[0] == '3' || str[0] == '4' || str[0] == '5' ||
  str[0] == '6' || str[0] == '7' || str[0] == '8' ||
  str[0] == '9' || isValidDelimiter(str[0]) == true)
```

```
  return (false);

  return (true);

}

bool  isValidKeyword(char* str) {

  if (!strcmp(str, "if") || !strcmp(str, "else") || !strcmp(str, "while") || !strcmp(str, "do") ||
!strcmp(str, "break") || !strcmp(str, "continue") || !strcmp(str, "int")

  || !strcmp(str, "double") || !strcmp(str, "float") || !strcmp(str, "return") || !strcmp(str,     "char") ||
!strcmp(str, "case") || !strcmp(str, "char")

  || !strcmp(str, "sizeof") || !strcmp(str, "long") || !strcmp(str, "short") || !strcmp(str, "typedef") ||
!strcmp(str, "switch") || !strcmp(str, "unsigned")

  || !strcmp(str, "void") || !strcmp(str, "static") || !strcmp(str, "struct") || !strcmp(str, "goto"))

  return (true);

  return (false);

}

bool isValidInteger(char* str) {

  int i, len = strlen(str);

  if (len == 0)

  return (false);

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

    if (str[i] != '0' && str[i] != '1' && str[i] != '2'&& str[i] != '3' && str[i] != '4' && str[i] != '5'

    && str[i] != '6' && str[i] != '7' && str[i] != '8' && str[i] != '9' || (str[i] == '-' && i > 0))

    return (false);

  }

  return (true);

}

bool isRealNumber(char* str) {

  int i, len = strlen(str);

  bool hasDecimal = false;
```

```
  if (len == 0)
  return (false);
  for (i = 0; i < len; i++) {
    if (str[i] != '0' && str[i] != '1' && str[i] != '2' && str[i] != '3' && str[i] != '4' && str[i]    != '5'
&& str[i] != '6' && str[i] != '7' && str[i] != '8'
    && str[i] != '9' && str[i] != '.' || (str[i] == '-' && i > 0))
    return (false);
      if (str[i] == '.' )
    hasDecimal = true;
  }
  return  (hasDecimal);
}
char* subString(char* str, int left, int right) {
  int i;
  char* subStr = (char*)malloc( sizeof(char) * (right - left + 2));
  for (i = left; i <= right; i++)
    subStr[i - left] = str[i];
  subStr[right - left + 1] = '\0';
  return (subStr);
}
void detectTokens(char* str) {
  int left = 0, right = 0;
  int length = strlen(str);
  while (right <= length && left <= right) {
    if  (isValidDelimiter(str[right])  ==  false)
    right++;
    if (isValidDelimiter(str[right]) == true && left == right) {
```

```
        if (isValidOperator(str[right]) == true)
        printf("Valid operator : '%c'
", str[right]);
        right++;
        left = right;
    } else if (isValidDelimiter(str[right]) == true && left != right || (right == length && left != right)) {
        char* subStr = subString(str, left, right - 1);
        if (isValidKeyword(subStr) == true)
          printf("Valid keyword : '%s'
", subStr);
        else if (isValidInteger(subStr) == true)
          printf("Valid Integer : '%s'
", subStr);
        else if (isRealNumber(subStr) == true)
          printf("Real Number : '%s'
", subStr);
        else if (isvalidIdentifier(subStr) == true
          && isValidDelimiter(str[right - 1]) == false)
        printf("Valid Identifier : '%s'
", subStr);
        else if (isvalidIdentifier(subStr) == false
          && isValidDelimiter(str[right - 1]) == false)
        printf("Invalid Identifier : '%s'
", subStr);
        left = right;
    }
```

```
    }
    return;
}
int main(){
    char str[100] = "float x = a + 1b; ";
    printf("The Program is : '%s'
", str);
    printf("All Tokens are :
");
    detectTokens(str);
    return (0);

}
```

Output

```
The Program is : 'float x = a + 1b; '

All Tokens are :

Valid keyword : 'float'

Valid Identifier : 'x'

Valid operator : '='

Valid Identifier : 'a'

Valid operator : '+'

Invalid Identifier : '1b'
```

Experiment -2

AIM: Write a Lex Program to implement a Lexical Analyzer using Lex tool.

ALGORITHM:

Step1: Lex program contains three sections: definitions, rules, and user subroutines. Each section must be separated from the others by a line containing only the delimiter, %%. The format is as follows: definitions %% rules %% user_subroutines

Step2: In definition section, the variables make up the left column, and their definitions make up the right column. Any C statements should be enclosed in %{..}%. Identifier is defined such that the first letter of an identifier is alphabet and remaining letters are alphanumeric.

Step3: In rules section, the left column contains the pattern to be recognized in an input file to yylex(). The right column contains the C program fragment executed when that pattern is recognized. The various patterns are keywords, operators, new line character, number, string, identifier, beginning and end of block, comment statements, preprocessor directive statements etc.

Step4: Each pattern may have a corresponding action, that is, a fragment of C source code to execute when the pattern is matched.

Step5: When yylex() matches a string in the input stream, it copies the matched text to an external character array, yytext, before it executes any actions in the rules section.

Step6: In user subroutine section, main routine calls yylex(). yywrap() is used to get more input.

Step7: The lex command uses the rules and actions contained in file to generate a program, lex.yy.c, which can be compiled with the cc command. That program can then receive input, break the input into the logical pieces defined by the rules in file, and run program fragments contained in the actions in file.

PROGRAM CODE:

```
//Implementation of Lexical Analyzer using Lex tool

%{
```

```
int COMMENT=0;
%}
identifier [a-zA-Z][a-zA-Z0-9]*
%%
#.* {printf("\n%s is a preprocessor directive",yytext);}
int |
float |
char |
double |
while |
for |
struct |
typedef |
do |
if |
break |
continue |
void |
switch |
return |
else |
goto {printf("\n\t%s is a keyword",yytext);}
"/*" {COMMENT=1;}{printf("\n\t %s is a COMMENT",yytext);}
{identifier}\( {if(!COMMENT)printf("\nFUNCTION \n\t%s",yytext);}
\{ {if(!COMMENT)printf("\n BLOCK BEGINS");}
\} {if(!COMMENT)printf("BLOCK ENDS ");}
```

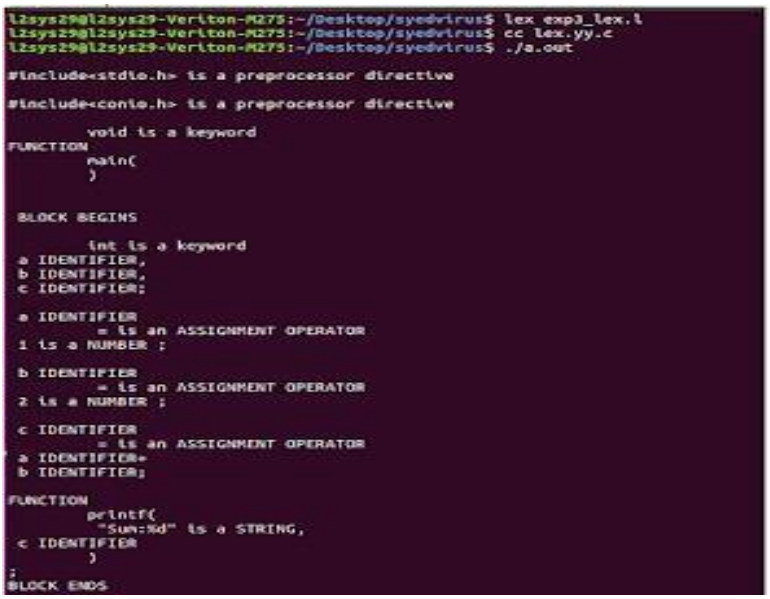
```
{identifier}(\[[0-9]*\])? {if(!COMMENT) printf("\n %s IDENTIFIER",yytext);}
\".*\" {if(!COMMENT)printf("\n\t %s is a STRING",yytext);}
[0-9]+ {if(!COMMENT) printf("\n %s is a NUMBER ",yytext);}
\)(\:)?  {if(!COMMENT)printf("\n\t");ECHO;printf("\n");}
\( ECHO;
= {if(!COMMENT)printf("\n\t %s is an ASSIGNMENT OPERATOR",yytext);}
\<= |
\>= |
\< |
== |
\> {if(!COMMENT) printf("\n\t%s is a RELATIONAL OPERATOR",yytext);}
%%
int main(int argc, char **argv)
{
FILE *file;
file=fopen("var.c","r");
if(!file)
{
printf("could not open the file");
exit(0);
}
yyin=file;
yylex();
printf("\n");
return(0);
}
```

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

INPUT:

```
//var.c
#include<stdio.h>
#include<conio.h>
void main()
{
int a,b,c;
a=1;
b=2;
c=a+b;
printf("Sum:%d",c);
}
```

OUTPUT:



Experiment -3

AIM: Write a C program to Simulate Lexical Analyzer to validating a given input String.

PROGRAM LOGIC :

Read the given input.

If the given input matches with any operator symbol.

Then display in terms of words of the particular symbol.

Else print not a operator.

PROCEDURE:

Go to debug -> run or press CTRL + F9 to run the program.

PROGRAM:

```
#include<stdio.h>
#include<conio.h>
void main()
{
char s[5];
```

```
clrscr();
printf("\n Enter any operator:");
gets(s);
switch(s[0])
{
case'>': if(s[1]=='=')
printf("\n Greater than or equal");
else
printf("\n  Greater  than");
break;
case'<': if(s[1]=='=')
printf("\n Less than or equal");
else
printf("\nLess than");
break;
case'=': if(s[1]=='=')
printf("\nEqual to");
else
printf("\nAssignment");
break;
case'!': if(s[1]=='=')
printf("\nNot Equal");
else
printf("\n Bit Not");
break;
case'&': if(s[1]=='&')
```

```
printf("\nLogical AND");
else
printf("\n Bitwise  AND");
break;
case'|': if(s[1]=='|')
printf("\nLogical OR");

else
printf("\nBitwise OR");
break;
case'+': printf("\n Addition");
break;
case'-': printf("\nSubstraction");
break;
case'*': printf("\nMultiplication");
break;
case'/': printf("\nDivision");
break;
case'%': printf("Modulus");
break;
default: printf("\n Not a operator");
}
getch();
}
```

INPUT & OUTPUT:

Input

Enter any operator: *

Output

Multiplication

Experiment -4

AIM: Write a C program to implement the Brute force technique of Top down Parsing.

```
#include<stdio.h>
#include<conio.h>
#include<iostream.h>
void main()
{
int a[30];
clrscr();
int min=10000,temp=0,i,lev,n,noofc,z;
printf("please enter how many number");
cin>>n;
for(i=0;i<n;i++)
a[i]=0;
```

```
cout<<"enter value of root";
cin>>a[0];
for(i=1;i<=n/2;i++)
{
cout<<"please enter no of child of parent with value"<<a[i-1]<<":";
cin>>noofc;
for(int j=1;j<=noofc;j++)
{z=(i)*2+j-2;
cout<<"please enter value of child";
cin>>a[z];
}
}
for(i=n-1;i>=n/2;i--)
{
temp=0;
for(int j=i+1;j>=1;j=j/2)
temp=temp+a[j-1];
if(temp<min)
min=temp;
cout<<"temp  min  is"<<temp<<"\n";
}
cout<<"min is"<<min;
getch();
}
```

Experiment -5

AIM: Write a C program to implement a Recursive Descent Parser

PROGRAM LOGIC:

Read the input string.

Write procedures for the non terminals

Verify the next token equals to non terminals if it satisfies match the non terminal.

If the input string does not match print error.

PROCEDURE:

Go to debug -> run or press CTRL + F9 to run the program.

PROGRAM:

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
char input[100];
int i,l;
void main()
{
clrscr();
printf("\nRecursive descent parsing for the following grammar\n");
printf("\nE->TE'\nE'->+TE'/@\nT->FT'\nT'->*FT'/@\nF->(E)/ID\n");
printf("\nEnter the string to be checked:");
gets(input);
if(E())
{
if(input[i+1]=='\0')
printf("\nString is accepted");
else
```

```
printf("\nString is not accepted");
}
else
printf("\nString not accepted");
getch();
}
E()
{
if(T())
{
if(EP())
return(1);
else
return(0);
}
else
return(0);
}
EP()
{
if(input[i]=='+')
{
i++;
if(T())
{
if(EP())
return(1);
else
```

```
return(0);


}
else
return(0);
}
else
return(1);
}
T()
{
if(F())
{
if(TP())
return(1);
else
return(0);
}
Else
return(0);
}
TP()
{
if(input[i]=='*')
{
i++;
if(F())
{
```

```
if(TP())
return(1);
else
return(0);
}
else
return(0);
}
else
return(1);
}
F()
{
if(input[i]=='(')
{
i++;
if(E())
{
if(input[i]==')')
{
i++;
return(1);

}
else
return(0);
}
else
```

```
return(0);
}
else    if(input[i]>='a'&&input[i]<='z'||input[i]>='A'&&input[i]<='Z')

{
i++;
return(1);
}
else
return(0);
}
```

INPUT & OUTPUT:

Recursive descent parsing for the following grammar

E->TE'

E'->+TE'/@

T->FT'

T'->*FT'/@

F->(E)/ID

Enter the string to be checked:(a+b)*c

String is accepted

Recursive descent parsing for the following grammar

E->TE'

E'->+TE'/@

T->FT'

T'->*FT'/@

F->(E)/ID

Enter the string to be checked: a/c+d

String is not accepted

Experiment-6

AIM: Write C program to compute the First and Follow Sets for the given Grammar

PROGRAM LOGIC:

Read the input string.

By using the FIRST AND FOLLOW values.

Verify the FIRST of non terminal and insert the production in the FIRST value

If we have any @ terms in FIRST then insert the productions in FOLLOW values

Constructing the predictive parser table

PROCEDURE:

Go to debug -> run or press CTRL + F9 to run the program.

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");
```

```
znsoftech@znsoftech-OEM:~$ gcc coders_hub.c
znsoftech@znsoftech-OEM:~$ ./a.out


THE GRAMMAR IS AS FOLLOWS

S -> ABE
S -> a
A -> p
A -> t
B -> Aq
S -> f
A -> w
 ->


FIRST

S -> #pta
A -> pt
B -> pt
S -> f
A -> w
 ->

FOLLOW

S -> $
A -> ptq
B ->
S ->
A -> ptq
 -> $pt
znsoftech@znsoftech-OEM:~$
```

Experiment-7

AIM: . Write a C program for eliminating the left recursion and left factoring of a given grammar

```
/*A program to remove left recursion in C with sscanf*/

#include<stdio.h>

#include<string.h>

void main()  {

   char  input[100],l[50],r[50],temp[10],tempprod[20],productions[25][50];

   int i=0,j=0,flag=0,consumed=0;

   printf("Enter the productions: ");

   scanf("%1s->%s",l,r);

   printf("%s",r);

   while(sscanf(r+consumed,"%[^|]s",temp) == 1 && consumed <= strlen(r)) {

      if(temp[0] == l[0]) {

         flag = 1;
         sprintf(productions[i++],"%s->%s%s'\0",l,temp+1,l);

      }

      else

         sprintf(productions[i++],"%s'->%s%s'\0",l,temp,l);

      consumed += strlen(temp)+1;

   }

   if(flag == 1) {

      sprintf(productions[i++],"%s->ε\0",l);

      printf("The productions after eliminating Left Recursion are:\n");

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

         printf("%s\n",productions[j]);

   }

   Else
```

```
    printf("The Given Grammar has no Left Recursion");
}
```

OUTPUT

```
Enter the productions: E->E+E|T
The productions after eliminating Left Recursion are:
E->+EE'
E'->TE'
E->ε
```

What is Left Factoring ?

Consider a part of regular grammar,

```
E->aE+bcD
E->aE+cBD
```

Here, grammar is non-left recursive, and unambiguous but there is left factoring.

How to resolve ?

```
E=aB | aC | aD | .....
```

then,

```
E=aX
X=B | C | D |.....
```

So, the above grammar will be as :

```
E=aE+X
X=bcD | cBD
```

Program :

```
1#include<stdio.h>
#include<string.h>
int main()
 {
    char  gram[20],part1[20],part2[20],modifiedGram[20],newGram[20],tempGram[20];
    int i,j=0,k=0,l=0,pos;
    printf("Enter Production : A->");
   gets(gram);
   for(i=0;gram[i]!='|';i++,j++)
        part1[j]=gram[i];
    part1[j]='\0';
    for(j=++i,i=0;gram[j]!='\0';j++,i++)
       part2[i]=gram[j];
    part2[i]='\0';
    for(i=0;i<strlen(part1)||i<strlen(part2);i++)
    {
```

```
        if(part1[i]==part2[i])
        {
            modifiedGram[k]=part1[i];
            k++;
            pos=i+1;
        }
    }
    for(i=pos,j=0;part1[i]!='\0';i++,j++){
        newGram[j]=part1[i];
    }
   newGram[j++]='|';
    for(i=pos;part2[i]!='\0';i++,j++){
        newGram[j]=part2[i];
    }
        modifiedGram[k]='X';

 modifiedGram[++k]='\0';

    newGram[j]='\0';

    printf("\n  A->%s",modifiedGram);
    printf("\n X->%s\n",newGram);

}
```

Experiment -8

AIM: Write a C program to check the validity of input string using Predictive Parser.

Experiment -9

AIM: Write a C program for implementation of LR parsing algorithm to accept a given input string.

Read the input string.

Push the input symbol with its state symbols in to the stack by referring lookaheads

We perform shift and reduce actions to parse the grammar.

Parsing is completed when we reach $ symbol.

PROCEDURE:

Go to debug -> run or press CTRL + F9 to run the program.

PROGRAM:

```
/*LALR PARSER

E->E+T

E->T

T->T*F
T->F

F->(E)

F->i

*/

#include<stdio.h>

#include<conio.h>

#include<stdlib.h>

#include<string.h>

void push(char *,int *,char);

char stacktop(char *);

void isproduct(char,char);

int ister(char);

int isnter(char);

int isstate(char);
```

```
void error();
void isreduce(char,char);
char pop(char *,int *);
void printt(char *,int *,char [],int);
void rep(char [],int);
struct action
{
char row[6][5];
};


const struct action A[12]={
{"sf","emp","emp","se","emp","emp"},
{"emp","sg","emp","emp","emp","acc"},
{"emp","rc","sh","emp","rc","rc"},
{"emp","re","re","emp","re","re"},
{"sf","emp","emp","se","emp","emp"},
{"emp","rg","rg","emp","rg","rg"},
{"sf","emp","emp","se","emp","emp"},
{"sf","emp","emp","se","emp","emp"},
{"emp","sg","emp","emp","sl","emp"},
{"emp","rb","sh","emp","rb","rb"},
{"emp","rb","rd","emp","rd","rd"},
{"emp","rf","rf","emp","rf","rf"}
};
struct gotol
{
char r[3][4];
};
```

```
const struct gotol G[12]={
{"b","c","d"},
{"emp","emp","emp"},
{"emp","emp","emp"},
{"emp","emp","emp"},
{"i","c","d"},
{"emp","emp","emp"},
{"emp","j","d"},
{"emp","emp","k"},
{"emp","emp","emp"},
{"emp","emp","emp"},
 };
char ter[6]={'i','+','*',')','(','$'};
char nter[3]={'E','T','F'};
char states[12]={'a','b','c','d','e','f','g','h','m','j','k','l'};
char stack[100];
int top=-1;
char temp[10];
struct grammar
{
char left;
char right[5];
};
const struct grammar rl[6]={
{'E',"e+T"},
{'E',"T"},
{'T',"T*F"},
{'T',"F"},
```

```
{'F',"(E)"},
{'F',"i"},
 };
void main()
{
char inp[80],x,p,dl[80],y,bl='a';
int i=0,j,k,l,n,m,c,len;
clrscr();
printf(" Enter the input :");
scanf("%s",inp);
len=strlen(inp);
inp[len]='$';
inp[len+1]='\0';
push(stack,&top,bl);
printf("\n stack \t\t\t input");
printt(stack,&top,inp,i);
do
{
x=inp[i];
p=stacktop(stack);
isproduct(x,p);
if(strcmp(temp,"emp")==0)
error();
if(strcmp(temp,"acc")==0)
break;
else
{
if(temp[0]=='s')
```

```
{
push(stack,&top,inp[i]);
push(stack,&top,temp[1]);
i++;
}
else
{
if(temp[0]=='r')
{
j=isstate(temp[1]);
strcpy(temp,rl[j-2].right);
dl[0]=rl[j-2].left;
dl[1]='\0';
n=strlen(temp);
for(k=0;k<2*n;k++)
pop(stack,&top);
for(m=0;dl[m]!='\0';m++)
push(stack,&top,dl[m]);
l=top;
y=stack[l-1];
isreduce(y,dl[0]);
for(m=0;temp[m]!='\0';m++)
push(stack,&top,temp[m]);
}
}
}
printt(stack,&top,inp,i);
}while(inp[i]!='\0');
```

```
if(strcmp(temp,"acc")==0)
printf(" \n accept the input ");
else
printf(" \n do not accept the input ");
getch();
}
void push(char *s,int *sp,char item)
{
if(*sp==100)
printf(" stack is full ");
else
{
*sp=*sp+1;

s[*sp]=item;
}
}
char stacktop(char *s)
{
char i;
i=s[top];
return i;
}
void isproduct(char x,char p)
{
int k,l;
k=ister(x);
l=isstate(p);
```

```
strcpy(temp,A[l-1].row[k-1]);
}
int ister(char x)
{
int i;
for(i=0;i<6;i++)
if(x==ter[i])
return i+1;
return 0;
}
int isnter(char x)
{
int i;
for(i=0;i<3;i++)
if(x==nter[i])
return i+1;
return 0;
}
int isstate(char p)
{
int i;
for(i=0;i<12;i++)
if(p==states[i])

return i+1;
return 0;
```

```
}
void error()
{
printf(" error in the input ");
exit(0);
}
void isreduce(char x,char p)
{
int k,l;
k=isstate(x);
l=isnter(p);
strcpy(temp,G[k-1].r[l-1]);
}
char pop(char *s,int *sp)
{
char item;
if(*sp==-1)
printf(" stack is empty ");
else
{
item=s[*sp];
*sp=*sp-1;
}
return item;
}
void printt(char *t,int *p,char inp[],int i)
```

```
{
int r;
printf("\n");
for(r=0;r<=*p;r++)
rep(t,r);
printf("\t\t\t");
for(r=i;inp[r]!='\0';r++)

printf("%c",inp[r]);
}
void rep(char t[],int r)
{
char c;
c=t[r];
switch(c)
{
case 'a': printf("0");
break;
case 'b': printf("1");
break;
case 'c': printf("2");
break;
case 'd': printf("3");
break;
case 'e': printf("4");
break;
```

```
case 'f': printf("5");
break;
case 'g': printf("6");
break;
case 'h': printf("7");
break;
case 'm': printf("8");
break;
case 'j': printf("9");
break;
case 'k': printf("10");
break;
case 'l': printf("11");
break;
default :printf("%c",t[r]);
break;
}
}
```

Experiment -10

AIM: Write a C program for implementation of a Shift Reduce Parser using Stack Data Structure to accept a given input string of a given grammar.

Shift reduce parser is a type of bottom-up parser. It uses a stack to hold grammar symbols. A parser goes on shifting the input symbols onto the stack until a handle comes on the top of the stack. When a handle occurs on the top of the stack, it implements reduction.

There are the various steps of Shift Reduce Parsing which are as follows −

- It uses a stack and an input buffer.
- Insert $ at the bottom of the stack and the right end of the input string in Input Buffer.

- Shift: Parser shifts zero or more input symbols onto the stack until the handle is on top of the stack.
- Reduce: Parser reduce or replace the handle on top of the stack to the left side of production, i.e., R.H.S. of production is popped, and L.H.S is pushed.
- Accept: Step 3 and Step 4 will be repeated until it has identified an error or until the stack includes start symbol (S) and input Buffer is empty, i.e., it contains $.

- Error: Signal discovery of a syntax error that has appeared and calls an error recovery routine.

For example, consider the grammar

S → aAcBe

A → Ab|b

B → d

and the string is abbcde.

It can reduce this string to S. It can scan string abbcde looking for the substring that matches the right side of some production. The substrings b and d qualify.

Let us select the left-most b and replace it with A, the left side of the production A → b, and obtain the string aAbcde. It can identify that Ab, b, and d each connect the right side of some production. Suppose this time it can select to restore the substring Ab by A, the left side of the production A → Ab and it can achieve aAcde.

Thus replacing d by B, the left side of the production B → d, and can achieve aAcBe. It can replace this string by S. Each replacement of the right-side of a production by the left side in the process above is known as reduction.

Drawbacks of Shift Reduce Parsing

- **Shift|Reduce Conflict** − Sometimes, the SR parser cannot determine whether to shift or to reduce.
- **Reduce|Reduce conflict** − Sometimes, the Parser cannot determine which of Production should be used for reduction.

Example − To stack implementation of shift-reduce parsing is done, consider the grammar −

E → E + E

E → E ∗ E

E → (E)

E → id

and input string as id1+id2∗id3��1+��2∗��3.

Stack	Input String	Action
$	$id_1 + id_2 * id_3$	Shift
$ id_1	$+id_2 * id_3$	Reduce by E → id
$ E	$+id_2 * id_3$	Shift
$ E +	$id_2 * id_3$	Shift
$ E + id_2	$* id_3$	Reduce by E → id
$E + E	$* id_3$	Shift
Stack	**Input String**	**Action**
$E + E *	id_3	Shift
$E + E * id_3	$	Reduce by E → id
$E + E * E	$	Reduce by E → E * E
$E + E	$	Reduce by E → E + E
$E	$	Accept

Program:

```
#include<stdio.h>
#include<string.h>
int k=0,z=0,i=0,j=0,c=0;
char a[16],ac[20],stk[15],act[10];
void check();
int main()
  {

    puts("GRAMMAR is E->E+E \n E->E*E \n E->(E) \n E->id");
    puts("enter input string ");
    gets(a);
    c=strlen(a);
    strcpy(act,"SHIFT->");
```

```
      puts("stack \t input \t action");
      for(k=0,i=0;  j<c;  k++,i++,j++)
       {
        if(a[j]=='i' && a[j+1]=='d')
          {
            stk[i]=a[j];
            stk[i+1]=a[j+1];
            stk[i+2]='\0';
            a[j]=' ';
            a[j+1]=' ';
            printf("\n$%s\t%s$\t%sid",stk,a,act);
            check();
          }
        else
          {
            stk[i]=a[j];
            stk[i+1]='\0';
            a[j]=' ';
            printf("\n$%s\t%s$\t%ssymbols",stk,a,act);
            check();
          }
       }

  }
void check()
  {
    strcpy(ac,"REDUCE  TO  E");
    for(z=0; z<c; z++)
     if(stk[z]=='i' && stk[z+1]=='d')
       {
         stk[z]='E';
         stk[z+1]='\0';
         printf("\n$%s\t%s$\t%s",stk,a,ac);
         j++;
       }
    for(z=0; z<c; z++)
     if(stk[z]=='E' && stk[z+1]=='+' && stk[z+2]=='E')
       {
         stk[z]='E';
         stk[z+1]='\0';
         stk[z+2]='\0';
         printf("\n$%s\t%s$\t%s",stk,a,ac);
         i=i-2;
       }
    for(z=0; z<c; z++)
     if(stk[z]=='E' && stk[z+1]=='*' && stk[z+2]=='E')
       {
         stk[z]='E';
         stk[z+1]='\0';
         stk[z+1]='\0';
         printf("\n$%s\t%s$\t%s",stk,a,ac);
         i=i-2;
```

```
    }

  for(z=0; z<c; z++)
    if(stk[z]=='(' && stk[z+1]=='E' && stk[z+2]==')')
    {
      stk[z]='E';
      stk[z+1]='\0';
      stk[z+1]='\0';
      printf("\n$%s\t%s$\t%s",stk,a,ac);
      i=i-2;
    }  }
```

Output :

```
GRAMMAR is E->E+E
E->E*E
E->(E)
E->id enter input string
id+id*id+id
stack          input          action
$id           +id*id+id$       SHIFT->id
$E            +id*id+id$       REDUCE TO E
$E+             id*id+id$      SHIFT->symbols
$E+id             *id+id$      SHIFT->id
$E+E              *id+id$      REDUCE TO E
$E               *id+id$       REDUCE TO E
$E*               id+id$       SHIFT->symbols
$E*id               +id$       SHIFT->id
$E*E                +id$       REDUCE TO E
$E                 +id$        REDUCE TO E
$E+                 id$        SHIFT->symbols
$E+id                 $        SHIFT->id
$E+E                  $        REDUCE TO E
$E                    $        REDUCE TO E
```

Experiment -11

AIM: Simulate the calculator using LEX and YACC tool.

The commands for executing the LEX program are:

lex abc.l (abc is the file name)

cc lex.yy.c -efl

./a.out

Let's see LEX program to implement a simple calculator.

Examples:
Input :

3+3

Output :

6.0

Input :

5*4

Output :

20.0

/*lex program to implement - a simple calculator.*/

```
% {
int op = 0,i;
float a, b;
% }

dig  [0-9]+|([0-9]*)"."([0-9]+)
add "+"
sub  "-"
```

```
mul "*"

div "/"

pow "^"

ln \n

%%


/* digi() is a user defined function */

{dig}  {digi();}

{add}  {op=1;}

{sub} {op=2;}

{mul}  {op=3;}
{div} {op=4;}

{pow} {op=5;}

{ln} {printf("\n The Answer :%f\n\n",a);}


%%

digi()

{

if(op==0)

/* atof() is used to convert

        - the ASCII input to float */

a=atof(yytext);

else

{

b=atof(yytext);

switch(op)

{
case 1:a=a+b;

        break;
```

```
    case2:a=a-b;
break;
    case 3:a=a*b;
    break;
    case 4:a=a/b;
    break;
    case 5:for(i=a;b>1;b--)
    a=a*i;
    break;
    }
    op=0;
    }
    }
    main(int argv,char *argc[])
    {
    yylex();
    }
    yywrap()
    {
    return 1;
    }
```

Experiment -12

Generate YACC specification for a few syntactic categories.

AIM:-

To write a program that recognize a valid arithmetic expression that uses operator +,-,* and /.

PROGRAM:-

```
%{      #include <stdio.h>
#include <ctype.h>
#include <stdlib.h>
%}
%token      num  let
%left  '+'   '-'
%left    '*'  '/'
%%
Stmt :          Stmt  „\n‟          {  printf (“\n.. Valid Expression.. \n”);  exit(0);  }
  |    expr
  |
  |    error    „\n‟       {    printf (“\n..Invalid ..\n”);     exit(0); }
  ;
expr :          num
           |    let
  |    expr  ‟ + ‟  expr
           |    expr   ‟ - ‟   expr
           |    expr   ‟ * ‟   expr
           |    expr  ‟ / ‟  expr
           |    '('     expr    ')'
           ;
           %%
```

```
main ( )
{
printf (“Enter an expression to validate : ”);
yyparse ( );
}

yylex()
{
        int ch;
        while ( ( ch = getchar() ) == ' ');
        if ( isdigit(ch) )
           return  num;                          // return token num
        if ( isalpha(ch) )
                    return  let;                  // return token let
        return ch;
}

yyerror (char *s)
{
    printf ( “%s”, s );
}
```

OUTPUT:-

```
$ yacc  -d  prog.y
$ cc  y.tab.c    -ll
./a.out
 4 + t                                          Accepted
( 5 * d + 5 )                                   Accepted
 a + * 5                                        Rejected
```

AIM:-

To write a program that recognizesa valid variable which starts with letter followed by a digit using yacc tool.

PROGRAM:-

```
%{
#include "y.tab.h"
%}
%%
[0-9] return digit;
[a-z] return letter;
[\n] return yytext[0];
. return 0;
%%
/* Yacc program to validate the given variable */
%{
#include<type.h>
%}
% token digit letter;
%%
ident : expn '\n' { printf ("valid\n"); exit (0); }
;
expn : letter
| expn letter
| expn digit
| error { yyerror ("invalid \n"); exit (0); }
;
%%
main()
{
yyparse();
}
yyerror (char *s)
{
printf("%s", s);
}
/* Yacc program which has c program to pass tokens */
%{
#include <stdio.h>
#include <ctype.h>
%}
%token LETTER DIGIT
%%
st:st LETTER DIGIT '\n' {printf("\nVALID");}
| st '\n'
|
| error '\n' {yyerror("\nINVALID");yyerrok;}
```

```
;
%%
main()
{
yyparse();
}

yylex()
{
char c;
while((c=getchar())==' ');
if(islower(c)) return LETTER;
if(isdigit(c)) return DIGIT;
return c;
}
yyerror(char *s)
{
printf("%s",s);
}
```

OUTPUT:-

```
a(a+0)
Valid
0(0+b)
Invalid
```

Experiment -13

Write a C program for generating the three address code of a given expression/statement.

AIM :

To write a C program to generate a three address code for a given expression.

ALGORITHM:

Step1: Begin the program
Step2 : The expression is read from the file using a file pointer
Step3 : Each string is read and the total no. of strings in the file is calculated.

Step4: Each string is compared with an operator; if any operator is seen then the previous string and next string are concatenated and stored in a first temporary value and the three address code expression is printed

Step5 : Suppose if another operand is seen then the first temporary value is concatenated to the next string using the operator and the expression is printed.
Step6 : The final temporary value is replaced to the left operand value.
Step7 : End the program.

PROGRAM: //program for three address code generation

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
#include<string.h>

struct three
{
char data[10],temp[7];
}s[30];
void main()
{
char d1[7],d2[7]="t";
int i=0,j=1,len=0;
FILE *f1,*f2;
clrscr();
f1=fopen("sum.txt","r");
f2=fopen("out.txt","w");
while(fscanf(f1,"%s",s[len].data)!=EOF)
len++;
itoa(j,d1,7);
strcat(d2,d1);
strcpy(s[j].temp,d2);
strcpy(d1,"");
```

```
strcpy(d2,"t");
if(!strcmp(s[3].data,"+"))
{
fprintf(f2,"%s=%s+%s",s[j].temp,s[i+2].data,s[i+4].data);
j++;
}
else  if(!strcmp(s[3].data,"-"))
{
fprintf(f2,"%s=%s-%s",s[j].temp,s[i+2].data,s[i+4].data);
j++;
}
for(i=4;i<len-2;i+=2)
{
itoa(j,d1,7);
strcat(d2,d1);
strcpy(s[j].temp,d2);
if(!strcmp(s[i+1].data,"+"))
fprintf(f2,"\n%s=%s+%s",s[j].temp,s[j-1].temp,s[i+2].data);
else if(!strcmp(s[i+1].data,"-"))
fprintf(f2,"\n%s=%s-%s",s[j].temp,s[j-1].temp,s[i+2].data);
strcpy(d1,"");
strcpy(d2,"t");
j++;
}
fprintf(f2,"\n%s=%s",s[0].data,s[j-1].temp);
fclose(f1);
fclose(f2);
getch();
}
```

Input: sum.txt

```
out = in1 + in2 + in3 - in4
```

Output : out.txt

```
t1=in1+in2
t2=t1+in3
t3=t2-in4
out=t3
```

Experiment -14

Write a C program for implementation of a Code Generation Algorithm of a given expression/statement.