Experiment-1

Aim:

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

Or

To design a lexical analyzer for given language and the lexical analyzer should ignore redundant spaces, tabs and new lines

```
#include<string.h>
#include<ctype.h>
#include<stdio.h>
void keyword(char str[10])
{
if(strcmp("for",str)==0||strcmp("while",str)==0||strcmp("do",str)==0||
strcmp("int",str)==0||strcmp("float",str)==0||strcmp("char",str)==0||
strcmp("double",str)==0||strcmp("static",str)==0||strcmp("switch",str)==0||
strcmp("case",str)==0)
printf("\n%s is a keyword",str);
else
printf("\n%s is an identifier",str);
}
main()
{
FILE *f1,*f2,*f3;
char c,str[10],st1[10];
int num[100],lineno=0,tokenvalue=0,i=0,j=0,k=0;
      printf("\nEnter the c program");/*gets(st1);*/
f1=fopen("input","w");
while((c=getchar())!=EOF)
putc(c,f1);
fclose(f1);
f1=fopen("input","r");
f2=fopen("identifier","w");
f3=fopen("specialchar","w");
while((c=getc(f1))!=EOF)
{
if(isdigit(c))
{
tokenvalue=c-'0';
c=getc(f1);
while(isdigit(c))
{
tokenvalue*=10+c-'0';
c=getc(f1);
}
num[i++]=tokenvalue;
ungetc(c,f1);
```

```
}
else if(isalpha(c))
{
putc(c,f2);
c=getc(f1);
while(isdigit(c)||isalpha(c)||c=='_'||c=='$')
{
putc(c,f2);
c=getc(f1);
}
putc(' ',f2);
ungetc(c,f1);
}
else if(c==' '||c=='\t')
printf(" ");
else if(c=='\n')
lineno++;
else
putc(c,f3);
}
fclose(f2);
fclose(f3);
fclose(f1);
printf("\nThe no's in the program are");
for(j=0;j<i;j++)
printf("%d",num[j]);
printf("\n");
f2=fopen("identifier","r");
k=0;
printf("The keywords and identifiersare:");
while((c=getc(f2))!=EOF)
{
if(c!=' ')
str[k++]=c;
else
{
      str[k]='\0';
                  keyword(str);
k=0;
}
}
fclose(f2);
f3=fopen("specialchar","r");
printf("\nSpecial characters are");
while((c=getc(f3))!=EOF)
printf("%c",c);
```

```
printf("\n");
fclose(f3);
printf("Total no. of lines are:%d",lineno);
}
```

Output:

```
Enter the C program
a+b*c
Ctrl-D
The no's in the program are:
The keywords and identifiers are:
a is an identifier and terminal
b is an identifier and terminal
c is an identifier and terminal

Special characters are:
+ *
Total no. of lines are: 1
```

EXPERIMENT-2

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

```
%{
  /* program to recognize a c program */
    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 |
do |
if |
break |
continue |
void |
switch |
case |
long |
struct |
const |
```

```
typedef |
return |
else |
goto       {printf("\n\t%s is a KEYWORD",yytext);}
"/*" {COMMENT = 1;}
    /*{printf("\n\n\t%s is a COMMENT\n",yytext);}*/
"*/" {COMMENT = 0;}
    /* printf("\n\n\t%s is a COMMENT\n",yytext);}*/
{identifier}\(    {if(!COMMENT)printf("\n\nFUNCTION\n\t%s",yytext);}
\{   {if(!COMMENT) printf("\n BLOCK BEGINS");}
\}   {if(!COMMENT) printf("\n 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\t%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)
{
 if (argc > 1)
 {
  FILE *file;
  file = fopen(argv[1],"r");
  if(!file)
  {
   printf("could not open %s \n",argv[1]);
   exit(0);
  }
  yyin = file;
 }
 yylex();
 printf("\n\n");
 return 0;
}
int yywrap()
{
  return 0;
}
```

Input:

```
$vi var.c
#include<stdio.h>
main()
{
  int a,b;
}
```

Output:

```
$lex lex.l
$cc lex.yy.c
$./a.out var.c
#include<stdio.h> is a PREPROCESSOR DIRECTIVE
FUNCTION
    main (

        )
BLOCK BEGINS
    int is a KEYWORD
a IDENTIFIER
b IDENTIFIER
BLOCK ENDS
```

EXPERIMENT-3

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

```
#include<stdio.h>
#include<ctype.h>
void main()
{
char a[10];
int flag,i;
printf("\n enter an identifier:");
scanf("%c",a);
if(isalpha(a[0]))
{
    flag=1;
}
else
{
    printf("\n Not a valid Identifier");
}
while(a[i]!='\0')
{
    if(!isdigit(a[i])&&!isalpha(a[i]))
    {
```

```
        flag=0;
        break;
    }
    i++;
    if(flag==1)
    {
        printf("\nvalid identifier");
    }
}
}
```

Output:

enter an identifier:itsme
valid identifier

EXPERIMENT-4

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

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

void check(void);
void set_value_backtracking(void);
void get_value_backtracking(void);
void display_output_string(void);

int iptr=0,optr=0,current_optr=0;
char output_string[20],current_output_string[20],input_string[20],temp_string[20];

int main(){
printf("\nEnter the string to check: ");
scanf("%s",input_string);
check();
return 0;}

void check(void){

   int flag=1,rule2_index=1;
    strcpy(output_string,"S");

    printf("\nThe output string in different stages are:\n");

    while(iptr<=strlen(input_string)){

        if(strcmp(output_string,temp_string)!=0){
            display_output_string();}
```

```
        if((iptr!=strlen(input_string)) || (optr!=strlen(output_string))){
            if(input_string[iptr]==output_string[optr]){
                iptr=iptr+1;
                optr=optr+1;}

            else{
                if(output_string[optr]=='S'){
                    memset(output_string,0,strlen(output_string));
                    strcpy(output_string,"cAd");}


                else if(output_string[optr]=='A'){
                    set_value_backtracking();

                    if(rule2_index==1){
                    memset(output_string,0,strlen(output_string));
                    strcpy(output_string,"cabd");}


                    else{
                    memset(output_string,0,strlen(output_string));
                    strcpy(output_string,"cad");}}

                else if(output_string[optr]=='b' && input_string[iptr]=='d'){
                    rule2_index=2;
                    get_value_backtracking();
                    iptr=iptr-1;}

                else{
                    printf("\nThe given string, '%s' is invalid.\n\n",input_string);
                    break;}}}

        else{
            printf("\nThe given string, '%s' is valid.\n\n",input_string);
            break;}}}


void set_value_backtracking(void){ //setting values for backtracking
    current_optr=optr;
    strcpy(current_output_string,output_string);
    return;}

void get_value_backtracking(void){ //backtracking and obtaining previous values
    optr=current_optr;
    memset(output_string,0,strlen(output_string));
    strcpy(output_string,current_output_string);
```

```
    return;}

void display_output_string(void){
    printf("%s\n",output_string);
    memset(temp_string,0,strlen(temp_string));
    strcpy(temp_string,output_string);
    return;}
```

Output:

```
1    Enter the string to check: cad
The output string in different stages are:
S
cAd
cabd
cAd
cad
The given string, 'cad' is valid.


2    Enter the string to check: cbd
The output string in different stages are:
S
cAd
cabd

The given string, 'cbd' is invalid.
```

EXPERIMENT-5

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

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

char input[10];
int i,error;
void E();
void T();
void Eprime();
void Tprime();
void F();
        main()
        {
i=0;
error=0;
            printf("Enter an arithmetic expression   :  "); // Eg: a+a*a
```

```
        gets(input);
        E();
        if(strlen(input)==i&&error==0)
                printf("\nAccepted..!!!\n");
        else printf("\nRejected..!!!\n");
                  }



void E()
{
    T();
    Eprime();
}
void Eprime()
{
    if(input[i]=='+')
    {
    i++;
    T();
    Eprime();
    }
    }
void T()
{
    F();
    Tprime();
}
void Tprime()
{
    if(input[i]=='*')
    {
                i++;
                F();
                Tprime();
                }
                }
    void F()
    {
        if(isalnum(input[i]))i++;
        else if(input[i]=='(')
        {
        i++;
        E();
        if(input[i]==')')
        i++;
```

```
        else error=1;
          }

        else error=1;
        }
```

Output:

1 Enter an arithmetic expression : a+a
Accepted..!!!
2 Enter an arithmetic expression : a++
Rejected..!!!

EXPERIMENT-6

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

```
#include<stdio.h>
#include<ctype.h>
char a[8][8];
struct firTab
{
   int n;
   char firT[5];
};
struct folTab
{
   int n;
   char folT[5];
};
struct folTab follow[5];
struct firTab first[5];
int col;
void findFirst(char,char);
void findFollow(char,char);
void folTabOperation(char,char);
void firTabOperation(char,char);
```

```
void main()
{
   int i,j,c=0,cnt=0;
   char ip;
   char b[8];
   printf("\nFIRST AND FOLLOW SET \n\nenter 8 productions in format A->B+T\n");
   for(i=0;i<8;i++)
   {
   scanf("%s",&a[i]);
   }
   for(i=0;i<8;i++)
   {   c=0;
   for(j=0;j<i+1;j++)
   {
      if(a[i][0] == b[j])
      {
         c=1;
         break;
      }
      }
   if(c !=1)
   {
     b[cnt] = a[i][0];
     cnt++;
   }
   }
    printf("\n");

   for(i=0;i<cnt;i++)
   {   col=1;
```

```
 first[i].firT[0] = b[i];
 first[i].n=0;
 findFirst(b[i],i);
 }
 for(i=0;i<cnt;i++)
 {
 col=1;
 follow[i].folT[0] = b[i];
 follow[i].n=0;
 findFollow(b[i],i);
  }
 printf("\n");
for(i=0;i<cnt;i++)
{
 for(j=0;j<=first[i].n;j++)
 {
        if(j==0)
        {
            printf("First(%c) : {",first[i].firT[j]);
        }
        else
        {
            printf(" %c",first[i].firT[j]);
        }
 }
 printf(" } ");
 printf("\n");
 }
  printf("\n");
for(i=0;i<cnt;i++)
```

```
  {
   for(j=0;j<=follow[i].n;j++)
   {
         if(j==0)
         {
            printf("Follow(%c) : {",follow[i].folT[j]);
         }
         else
         {
            printf(" %c",follow[i].folT[j]);
         }
   }
   printf(" } ");
   printf("\n");
   }
}
void findFirst(char ip,char pos)
{
   int i;
   for(i=0;i<8;i++)
   {
      if(ip == a[i][0])
      {
         if(isupper(a[i][3]))
         {
            findFirst(a[i][3],pos);
         }
         else
      {
      first[pos].firT[col]=a[i][3];
```

```
        first[pos].n++;
        col++;
           }
        }
    }
}
void findFollow(char ip,char row)
{   int i,j;
    if(row==0 && col==1)
    {
        follow[row].folT[col]= '$';
        col++;
        follow[row].n++;
    }
    for(i=0;i<8;i++)
    {
        for(j=3;j<7;j++)
        {
           if(a[i][j] == ip)
           {
              if(a[i][j+1] == '\0')
              {
                 if(a[i][j] != a[i][0])
                 {
                    folTabOperation(a[i][0],row);
                 }
              }
              else if(isupper(a[i][j+1]))
              {   if(a[i][j+1] != a[i][0])
                 {
```

```
                    firTabOperation(a[i][j+1],row);
                }
                }
                else
                {
                    follow[row].folT[col] = a[i][j+1];
                    col++;
                    follow[row].n++;
                }
            }
        }
    }
}
void folTabOperation(char ip,char row)
{   int i,j;
    for(i=0;i<5;i++)
    {
        if(ip == follow[i].folT[0])
        {
            for(j=1;j<=follow[i].n;j++)
            {
                follow[row].folT[col] = follow[i].folT[j];
                col++;
                follow[row].n++;
            }
        }
    }
}
void firTabOperation(char ip,char row)
{
```

```
    int i,j;
  for(i=0;i<5;i++)
  {
    if(ip == first[i].firT[0])
    {
      for(j=1;j<=first[i].n;j++)
      {
        if(first[i].firT[j] != '0')
        {
          follow[row].folT[col] = first[i].firT[j];
          follow[row].n++;
          col++;
        }
        else
        {
          folTabOperation(ip,row);
        }
      }
    }
  }

}

/*Input productions

E->TA
A->+TA
A->0
T->FB
B->*FB
```

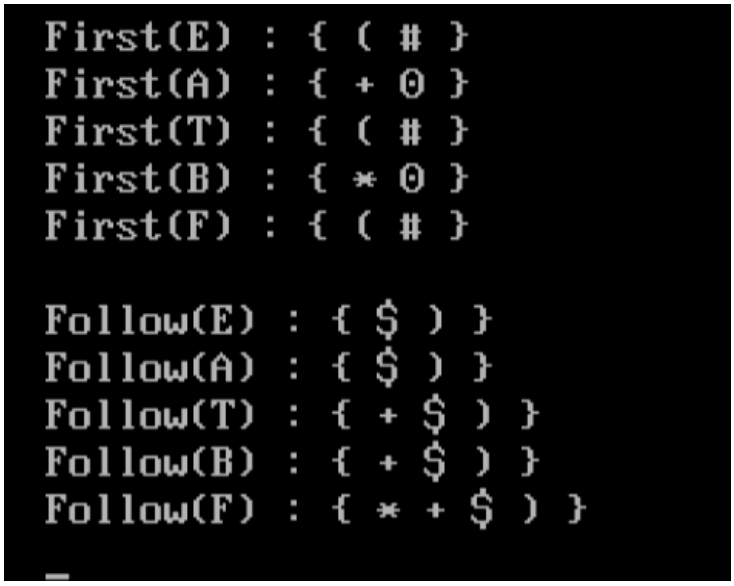
B->0

F->(E)

F->#

*/

Output:



EXPERIMENT-7

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

```
#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(" %s->%s", l, r);
 printf("%s", r);
while(sscanf(r+consumed, " % [^l] s", temp) == 1 &&consumed<=strlen(r))
{
 if(temp[0] == l[0])
 {
 flag = 1;
 sprintf(productions[i++], "%s->%s%s ‘\0", l,temp+1,1);
 }
else
```

```
 sprintf(productions[i++], "%s->%s%s ‘\0",l, temp,1);
 consumed += strlen(temp)+1;
}
if(flag==1)
{
 sprintf(productions[i++], "%s->€ \0", 1);
 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+T
 The productions after eliminating Left Recursion are: E->+TE’
E->
Enter the Productions:
T->T*F
The productions after eliminating Left Recursion are: T-> *FT’
T->
 Enter the Productions:
F->id
The Given Grammar has no Left Recursion
```

EXPERIMENT-8

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

```
#include<stdio.h>
#include<string.h>
int spt=0,ipt=0;
char s[20],ip[15];
char *m[5][6]={{"TG","\0","\0","TG","\0","\0"},
      {"\0","+TG","\0","\0","e","e"},
      {"FH","\0","\0","FH","\0","\0"},
               {"\0","e","*FH","\0","e","e"},
               {"i","\0","\0","(E)","\0","\0"}};
char nt[5]={'E','G','T','H','F'};
char t[6]={'i','+','*','(',')','$'};
int nti(char c)
{
   int i;
   for(i=0;i<5;i++){
if(nt[i]==c)
```

```
  return(i);
   }
   return(6);
}
int ti(char c)
{
   int i;
   for(i=0;i<6;i++) {
if(t[i]==c)
  return(i);
   }
   return(7);
}
main()
{
  char prod[4],temp[4];
  int l,k,j;
  printf("enter input string:");
  scanf("%s",ip);
  strcat(ip,"$");
  s[0]='$';
  s[1]='E';
  s[2]='\0';
  spt=1;
  while(1) {
    if(ip[ipt]=='$'&&s[spt]=='$')
break;
    if(ti(s[spt])<5||s[spt]=='$')
    {
if(s[spt]==ip[ipt])
       {
spt--;
ipt++;
       }
else
error();
    }
    else if(nti(s[spt]<6)){
strcpy(prod,m[nti(s[spt])][ti(ip[ipt])]);
if(prod=='\0')
error();
l=strlen(prod);
for(k=l-1,j=0;k>=0&&j<=l;k--,j++)
temp[j]=prod[k];
for(k=0;k<l;k++)
prod[k]=temp[k];
```

```
s[spt--]='\0';
strcat(s,prod);
spt=spt+l;
if(s[spt]=='e')
s[spt--]='\0';
   }
   else
error();
 }
 if(s[spt]=='$'&&ip[ipt]=='$')
printf("\n input is parsed\n");

 else
    error();
 return 0;
}
error()
{
 printf("input is not parsed\n");
 exit(1);
 return 0;
}
```

Output:

```
1   enter input string:i+i*i$
input is parsed
2    enter input string:a*b%c
input is not parsed
```

EXPERIMENT-9

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

```
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
int axn[][6][2]={
{{100,5},{-1,-1},{-1,-1},{100,4},{-1,-1},{-1,-1}},
{{-1,-1},{100,6},{-1,-1},{-1,-1},{-1,-1},{102,102}},
{{-1,-1},{101,2},{100,7},{-1,-1},{101,2},{101,2}},
{{-1,-1},{101,4},{101,4},{-1,-1},{101,4},{101,4}},
{{100,5},{-1,-1},{-1,-1},{100,4},{-1,-1},{-1,-1}},
{{-1,-1},{101,6},{101,6},{-1,-1},{101,6},{101,6}},
{{100,5},{-1,-1},{-1,-1},{-1,-1},{-1,-1},{-1,-1}},
{{100,5},{-1 -1},{-1,-1},{100,4},{-1,-1},{-1,-1}},
{{-1,-1},{100,6},{-1,-1},{-1,-1},{100,11},{-1,-1}},
{{-1,-1},{101,1},{100,7},{-1,-1},{101,1},{101,1}},
{{-1,-1},{101,3},{101,3},{-1,-1},{101,3},{101,3}},
{{-1,-1},{101,5},{101,5},{-1,-1},{101,5},{101,5}},
```

```
};
int gotot[12][3]={1,2,3,-1,-1,-1-1,-1,-1,-1,-1,-1,8,2,3,-1,-1,-1,-
1,9,3,-1,-1,10,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1};
int a[10];
char b[10];
int top=-1,btop=-1,i;
void push(int k)
{
   if(top<9)
   a[++top]=k;
}
void pushb(char k)
{
  if(btop<9)
    b[++btop]=k;
}
char TOS()
{
  return a[top];
}

void pop() {
  if(top>=0)
    top--;
}
void popb() {
  if(btop>=10)
    b[btop--]='\0';
}
void display() {
  for(i=0;i<top;i++)
    printf("%d%c",a[i],b[i]);
}
void display1(char p[],int m) {
  int l;
  printf("\t\t");
  for(l=m;p[l]!='\0';l++)
     printf("%c",p[l]);
  printf("\n");
}
void error() {
    printf("syntax error");
}
void reduce(int p) {
   int k,ad;
   char src,*dest;
   switch(p) {
      case 1:dest="E+T";
             src='E';
             break;
      case 2:dest="T";
             src='E';
            break;
      case 3:dest="T*F";
```

```
              src='T';
              break;
       case 4:dest="F";
              src='T';
              break;
       case 5:dest="(E)";
              src='F';
              break;
       case 6:dest="i";
              src='F';
              break;
       default :dest="\0";
              src='\0';
              break;
   }
   for(k=0;k<strlen(dest);k++) {
      pop();
      popb(); }
      pushb(src);
      switch(src) {
           case 'E':ad=0;
break;
case 'T':ad=1;
break;
case 'F':ad=2;
break;
default:ad=-1;
break; }
push(gotot[TOS()][ad]); }
int main() {

int j,st,ic;
char ip[20]="\0",an;
printf("enter any string");
scanf("%s",ip);
push(0);
display();
printf("\t%s\n",ip);
for(j=0;ip[j]!='\0';)
{
st=TOS();
an=ip[j];
if(an>='a'&&an<='z') ic=0;
else if(an=='+') ic=1;
else if(an=='*') ic=2;
else if(an=='(') ic=3;
else if(an==')') ic=4;
else if(an=='$') ic=5;
else {
error();
break; }
if(axn[st][ic][0]==100) {
pushb(an);
push(axn[st][ic][1]);
```

```
display();
j++;
display1(ip,j); }
if(axn[st][ic][0]==101) {
reduce(axn[st][ic][1]);
display();
display1(ip,j);

}
if(axn[st][ic][1]==102) {
printf("given string is accepted");

break;
}
}
return(0);
}
```

Output:

```
enter any string i+i*i
i+i*i
       i+i*i
0i             +i*i
0i             +i*i
0i             +i*i
0i             +i*i
0i1F           i*i
0i1F6T         *i
0i1F6T         *i
```

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.

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
char ip_sym[15],stack[15];
int ip_ptr=0,st_ptr=0,len,i;
char temp[2],temp2[2];
char act[15];
void check();
void main()
{
printf("\n\t\t SHIFT REDUCE PARSER\n");
printf("\n GRAMMER\n");
printf("\n E->E+E\n  E->a/b \n");
printf("\n enter the input symbol:\t");
scanf("%c",ip_sym);
printf("\n\t stack implementation table");
printf("\n stack\t\t input symbol\t\t action");
printf("\n______\t\t _____________\t\t ______\n");
```

```
printf("\n $\t\t%s$\t\t\t--",ip_sym);
strcpy(act,"shift ");
temp[0]=ip_sym[ip_ptr];
temp[1]='\0';
strcat(act,temp);
len=strlen(ip_sym);
for(i=0;i<=len-1;i++)
{
stack[st_ptr]=ip_sym[ip_ptr];
stack[st_ptr+1]='\0';
ip_sym[ip_ptr]=' ';
ip_ptr++;
printf("\n $%s\t\t%s$\t\t\t%s",stack,ip_sym,act);
strcpy(act,"shift ");
temp[0]=ip_sym[ip_ptr];
temp[1]='\0';
strcat(act,temp);
check();
st_ptr++;
}
st_ptr++;
check();
}
void check()
{
int flag=0;
temp2[0]=stack[st_ptr];
temp2[1]='\0';
if((!strcmp(temp2,"a"))||(!strcmp(temp2,"b")))
{
 stack[st_ptr]='E';
 if(!strcmp(temp2,"a"))
  printf("\n $%s\t\t%s$\t\t\tE->a",stack, ip_sym);
 else
  printf("\n $%s\t\t%s$\t\t\tE->b",stack,ip_sym);
 flag=1;
}
if((!strcmp(temp2,"+"))||(strcmp(temp2,"*"))||(!strcmp(temp2,"/")))
{
 flag=1;
}
if((!strcmp(stack,"E+E"))||(!strcmp(stack,"E\E"))||(!strcmp(stack,"E*E")))
{
strcpy(stack,"E");
st_ptr=0;
if(!strcmp(stack,"E+E"))
```

```
printf("\n $%s\t\t%s$\t\t\tE->E+E",stack,ip_sym);
else
if(!strcmp(stack,"E\E"))
printf("\n $%s\t\t %s$\t\t\tE->E\E",stack,ip_sym);
else
printf("\n $%s\t\t%s$\t\t\tE->E*E",stack,ip_sym);
flag=1;
}
if(!strcmp(stack,"E")&&ip_ptr==len)
{
printf("\n $%s\t\t%s$\t\t\tACCEPT",stack,ip_sym);
exit(0);
}
if(flag==0)
{
printf("\n%s\t\t\t%s\t\t reject",stack,ip_sym);
exit(0);
}
return;
}
```

Output:

```
SHIFT REDUCE PARSER

 GRAMMER

 E->E+E
  E->a/b

 enter the input symbol: a-b
 stack implementation table
 stack input symbol action
_____ ___________ _____

 $ a$ --
 $a $ shift a
 $E $ E->a
 $E $ ACCEPT
```

EXPERIMENT-11

Aim: . Simulate the calculator using LEX and YACC tool

//Implementation of calculator using LEX and YACC

LEX PART:

```
%{
```

```
#include<stdio.h>
#include "y.tab.h"
extern int yylval;
%}

%%
[0-9]+ {
        yylval=atoi(yytext);
        return NUMBER;
      }
[\t] ;
[\n] return 0;
. return yytext[0];
%%
int yywrap()
{
return 1;
}
```

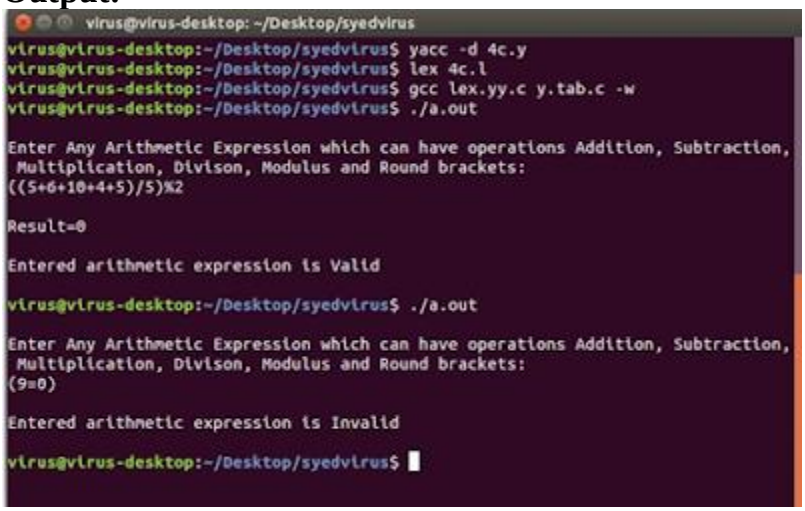
YACC PART:

```
%{
   #include<stdio.h>
   int flag=0;

%}
```

```
%token NUMBER


%left '+' '-'

%left '*' '/' '%'

%left '(' ')'

%%

ArithmeticExpression: E{

        printf("\nResult=%d\n",$$);

        return 0;

       };

E:E'+'E {$$=$1+$3;}

 |E'-'E {$$=$1-$3;}

 |E'*'E {$$=$1*$3;}

 |E'/'E {$$=$1/$3;}

 |E'%'E {$$=$1%$3;}

 |'('E')' {$$=$2;}

 | NUMBER {$$=$1;}

;

%%


void main()

{

   printf("\nEnter Any Arithmetic Expression which can have operations Addition,
Subtraction, Multiplication, Divison, Modulus and Round brackets:\n");
```

```
   yyparse();
  if(flag==0)
   printf("\nEntered arithmetic expression is Valid\n\n");

}
void yyerror()
{
   printf("\nEntered arithmetic expression is Invalid\n\n");
   flag=1;
}
```

Output:



EXPERIMENT-12

Aim: . Generate YACC specification for a few syntactic categories.

Program name:arith_id.l

```
%{
/* This LEX program returns the tokens for the expression */
#include “y.tab.h”
%}
```

```
%%
“=” {printf(“\n Operator is EQUAL”);}
“+” {printf(“\n Operator is PLUS”);}
“-“ {printf(“\n Operator is MINUS”);}
“/” {printf(“\n Operator is DIVISION”);}
“*” {printf(“\n Operator is MULTIPLICATION”);}

[a-z A-Z]*[0-9]* {
printf(“\n Identifier is %s”,yytext);
return ID;
}
return yytext[0];
\n return 0;
%%

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

Program Name : arith_id.y

```
%{
#include
/* This YYAC program is for recognizing the Expression */
%}
%%
statement: A’=’E
| E {
printf(“\n Valid arithmetic expression”);
$$ = $1;
};

E: E’+’ID
| E’-’ID
| E’*’ID
| E’/’ID
| ID
;
%%
extern FILE *yyin;
main()
{
do
{
```

```
yyparse();
}while(!feof(yyin));
}

yyerror(char*s)
{
}
```

Output:

```
[root@localhost]# lex arith_id.1
[root@localhost]# yacc –d arith_id.y
[root@localhost]# gcc lex.yy.c y.tab.c
[root@localhost]# ./a.out
x=a+b;

Identifier is x
Operator is EQUAL
Identifier is a
Operator is PLUS
Identifier is b
```