

Exp. No. 1

Develop a lexical Analyzer to identify identifiers, constants, operators using C program.

Program:

```
#include<stdio.h>
#include<ctype.h>
#include<string.h>
int main()
{
    int i,ic=0,m,cc=0,oc=0,j;
    char b[30],operators[30],identifiers[30],constants[30];
    printf("enter the string : ");
    scanf("%[^\\n]s",&b);
    for(i=0;i<strlen(b);i++)
    {
        if(isspace(b[i]))
        {
            continue;
        }
        else if(isalpha(b[i]))
        {
            identifiers[ic] =b[i];
            ic++;
        }
        else if(isdigit(b[i]))
        {
            m=(b[i]-'0');
            i=i+1;
            while(isdigit(b[i]))
            {
                m=m*10 + (b[i]-'0');
                i++;
            }
            i=i-1;
            constants[cc]=m;
            cc++;
        }
    }
}
```

```

else
    {
    if(b[i]=='*')
        {
            operators[oc]='*';
            oc++;
        }
    else if(b[i]=='-')
        {
            operators[oc]='-';
            oc++;
        }
    else if(b[i]=='+')
        {
            operators[oc]='+';
            oc++;
        }
    else if(b[i]=='=')
        {
            operators[oc]='=';
            oc++;
        }
    }
}

printf(" identifiers : ");
for(j=0;j<ic;j++)
    {
        printf("%c ",identifiers[j]);
    }
printf("\n constants : ");
for(j=0;j<cc;j++)
    {
        printf("%d ",constants[j]);
    }
printf("\n operators : ");
for(j=0;j<oc;j++)
    {

```

```

        printf("%c ",operators[j]);
    }
}

```

Output:

```

enter the string : a = b + c * e + 100
identifiers : a b c e
constants : 100
operators : = + * +

```

Exp. No. 2

Develop a lexical Analyzer to identify whether a given line is a comment or not using C

Program:

```

#include<stdio.h>
#include<conio.h>
int main()
{
    char com[30];
    int i=2,a=0;
    printf("\n Enter comment:");
    gets(com);
    if(com[0]=='/')
    {
        if(com[1]=='/')
            printf("\n It is a comment");
        else if(com[1]=='*')
        {
            for(i=2;i<=30;i++)
            {
                if(com[i]=='*'&&com[i+1]=='/')
                {
                    printf("\n It is a comment");
                    a=1;
                    break;
                }
            }
            else

```

```

                                continue;
                            }
                            if(a==0)
                                printf("\n It is not a comment");
                        }
                    else
                        printf("\n It is not a comment");
                }
            else
                printf("\n It is not a comment");
        }
    }
}

```

Output:

Input: Enter comment: //hello

Output: It is a comment

Input: Enter comment: hello

Output: It is not a comment

Exp. No. 3

Design a lexical Analyzer for given language should ignore the redundant spaces, tabs and new lines and ignore comments using C

Program:

```

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

int isKeyword(char buffer[]){
    char keywords[32][10] =
    {"main","auto","break","case","char","const","continue","default",
    "do","double","else","enum","extern","float","for","goto",
    "if","int","long","register","return","short","signed",
    "sizeof","static","struct","switch","typedef",
    "unsigned","void","printf","while"};
    int i, flag = 0;

```

```

for(i = 0; i < 32; ++i){
    if(strcmp(keywords[i], buffer) == 0){
        flag = 1;
        break;
    }
}
return flag;
}

```

```

int main(){
    char ch, buffer[15], operators[] = "+-*/%=";
    FILE *fp;
    int i,j=0;
    fp = fopen("3lex_input.txt","r");
    if(fp == NULL){
        printf("error while opening the file\n");
        exit(0);
    }
    while((ch = fgetc(fp)) != EOF){
        for(i = 0; i < 6; ++i){
            if(ch == operators[i])
                printf("%c is operator\n", ch);
        }

        if(isalnum(ch)){
            buffer[j++] = ch;
        }
        else if((ch == ' ' || ch == '\n') && (j != 0)){
            buffer[j] = '\0';
            j = 0;

            if(isKeyword(buffer) == 1)
                printf("%s is keyword\n", buffer);
            else
                printf("%s is identifier\n", buffer);
        }
    }
}

```

```
}  
fclose(fp);  
return 0;  
}
```

Input: 3lex_input.txt

```
main ( )  
{  
    int a, b, c ;  
    c = b + c;  
    printf ( "%d" ,c ) ;  
}
```

Output:

main is keyword
int is keyword
a is indentifier
b is indentifier
c is indentifier
c is indentifier
= is operator
b is indentifier
+ is operator
c is indentifier
printf is keyword
% is operator
d is indentifier
c is indentifier

Exp. No. 4

Design a lexical Analyzer to validate operators to recognize the operators +,-,*,/ using regular arithmetic operators using C

Program:

```
#include<stdio.h>  
#include<conio.h>  
int main()  
{
```

```
char s[5];
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("\n Less than");
        break;
    case '=':
        if(s[1]=='=')
            printf("\n Equal to");
        else
            printf("\n Assignment");
        break;
    case '!':
        if(s[1]=='=')
            printf("\n Not Equal");
        else
            printf("\n Bit Not");
        break;
    case '&':
        if(s[1]=='&')
            printf("\n Logical 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");
}
}

```

Output:

Enter any operator:<=
 Less than or equal

Exp. No. 5

Design a lexical Analyzer to find the number of whitespaces and newline characters using C.

Program:

```

#include <stdio.h>
int main()
{
    char str[100]; //input string with size 100

```



```

int words=0,newline=0,characters=0; // counter variables

scanf("%[^~]",&str);//scanf formatting

for(int i=0;str[i]!='\0';i++)
{
    if(str[i] == ' ')
    {
        words++;
    }
    else if(str[i] == '\n')
    {
        newline++;
        words++;//since with every next line new words start. corner case 1
    }
    else if(str[i] != ' ' && str[i] != '\n'){
        characters++;
    }
}
if(characters > 0)//Corner case 2,3.
{
    words++;
    newline++;
}
printf("Total number of words : %d\n",words);
printf("Total number of lines : %d\n",newline);
printf("Total number of characters : %d\n",characters);
return 0;
}

```

Output:

```

void main()
{
    int a;
    int b;
    a = b + c;
    c = d * e;
}

```

```
}
```

Total number of words : 18

Total number of lines : 7

Exp. No. 6

Develop a lexical Analyzer to test whether a given identifier is valid or not using C.

Program:

```
#include<stdio.h>
#include<conio.h>
#include<ctype.h>
int main()
{
    char a[10];
    int flag, i=1;
    printf("\n Enter an identifier:");
    gets(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("\n Valid identifier");
}
```

Output:

Enter an identifier:abc123

Valid identifier

Exp. No. 7

Write a C program to find FIRST() - predictive parser for the given grammar

$S \rightarrow AaAb / BbBa$

$A \rightarrow \epsilon$

$B \rightarrow \epsilon$

Program:

```
#include<stdio.h>
#include<ctype.h>
void FIRST(char[],char );
void addToResultSet(char[],char);
int numOfProductions;
char productionSet[10][10];
int main()
{
    int i;
    char choice;
    char c;
    char result[20];
    printf("How many number of productions ? :");
    scanf(" %d",&numOfProductions);
    for(i=0;i<numOfProductions;i++)//read production string eg: E=E+T
    {
        printf("Enter productions Number %d : ",i+1);
        scanf(" %s",productionSet[i]);
    }
    do
    {
        printf("\n Find the FIRST of :");
        scanf(" %c",&c);
        FIRST(result,c); //Compute FIRST; Get Answer in 'result' array
        printf("\n FIRST(%c)= { ",c);
        for(i=0;result[i]!='\0';i++)
            printf(" %c ",result[i]);    //Display result
        printf("}\n");
        printf("press 'y' to continue : ");
        scanf(" %c",&choice);
    }
```

```

    }
    while(choice=='y' | | choice == 'Y');
}
/*
*Function FIRST:
*Compute the elements in FIRST(c) and write them
*in Result Array.
*/
void FIRST(char* Result,char c)
{
    int i,j,k;
    char subResult[20];
    int foundEpsilon;
    subResult[0]='\0';
    Result[0]='\0';
    //If X is terminal, FIRST(X) = {X}.
    if(!(isupper(c)))
    {
        addToResultSet(Result,c);
        return ;
    }
    //If X is non terminal
    //Read each production
    for(i=0;i<numOfProductions;i++)
    {
        //Find production with X as LHS
        if(productionSet[i][0]==c)
        {
            //If  $X \rightarrow \epsilon$  is a production, then add  $\epsilon$  to FIRST(X).
            if(productionSet[i][2]=='$') addToResultSet(Result,'$');
            //If X is a non-terminal, and  $X \rightarrow Y_1 Y_2 \dots Y_k$ 
            //is a production, then add a to FIRST(X)
            //if for some i, a is in FIRST( $Y_i$ ),
            //and  $\epsilon$  is in all of FIRST( $Y_1$ ), ..., FIRST( $Y_{i-1}$ ).
            else
            {
                j=2;

```

```

        while(productionSet[i][j]!='\0')
        {
            foundEpsilon=0;
            FIRST(subResult,productionSet[i][j]);
            for(k=0;subResult[k]!='\0';k++)
                addToResultSet(Result,subResult[k]);
            for(k=0;subResult[k]!='\0';k++)
                if(subResult[k]=='$')
                {
                    foundEpsilon=1;
                    break;
                }
            //No  $\epsilon$  found, no need to check next element
            if(!foundEpsilon)
                break;
            j++;
        }
    }
}

return ;
}

/* addToResultSet adds the computed
*element to result set.
*This code avoids multiple inclusion of elements
*/
void addToResultSet(char Result[],char val)
{
    int k;
    for(k=0 ;Result[k]!='\0';k++)
        if(Result[k]==val)
            return;
    Result[k]=val;
    Result[k+1]='\0';
}

```

Output:

How many number of productions ? :4

Enter productions Number 1 : S=AaAb

Enter productions Number 2 : S=BbBa

Enter productions Number 3 : A=\$

Enter productions Number 4 : B=\$

Find the FIRST of :S

FIRST(S)= { \$ a b }

press 'y' to continue : y

Find the FIRST of :A

FIRST(A)= { \$ }

press 'y' to continue : y

Find the FIRST of :B

FIRST(B)= { \$ }

press 'y' to continue : n

Exp. No. 8

Write a C program to find FOLLOW() - predictive parser for the given grammar

$S \rightarrow AaAb / BbBa$

$A \rightarrow \epsilon$

$B \rightarrow \epsilon$

Program:

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

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

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

```
int limit, x = 0;
```

```
char production[10][10], array[10];
```

```
void find_first(char ch);
```

```
void find_follow(char ch);
```

```
void Array_Manipulation(char ch);
```

```
int main()
```

```
{
    int count;
    char option, ch;
    printf("\nEnter Total Number of Productions:\t");
    scanf("%d", &limit);
    for(count = 0; count < limit; count++)
    {
        printf("\nValue of Production Number [%d]:\t", count + 1);
        scanf("%s", production[count]);
    }
    do
    {
        x = 0;
        printf("\nEnter production Value to Find Follow:\t");
        scanf(" %c", &ch);
        find_follow(ch);
        printf("\nFollow Value of %c:\t{ ", ch);
        for(count = 0; count < x; count++)
        {
            printf("%c ", array[count]);
        }
        printf("}\n");
        printf("To Continue, Press Y:\t");
        scanf(" %c", &option);
    }while(option == 'y' || option == 'Y');
    return 0;
}
```

```
void find_follow(char ch)
```

```
{
    int i, j;
    int length = strlen(production[i]);
    if(production[0][0] == ch)
    {
```

```

        Array_Manipulation('$');
    }
    for(i = 0; i < limit; i++)
    {
        for(j = 2; j < length; j++)
        {
            if(production[i][j] == ch)
            {
                if(production[i][j + 1] != '\0')
                {
                    find_first(production[i][j + 1]);
                }
                if(production[i][j + 1] == '\0' && ch != production[i][0])
                {
                    find_follow(production[i][0]);
                }
            }
        }
    }
}

```

```

void find_first(char ch)
{
    int i, k;
    if(!(isupper(ch)))
    {
        Array_Manipulation(ch);
    }
    for(k = 0; k < limit; k++)
    {
        if(production[k][0] == ch)
        {
            if(production[k][2] == '$')
            {
                find_follow(production[i][0]);
            }
            else if(islower(production[k][2]))

```



```

        {
            Array_Manipulation(production[k][2]);
        }
        else
        {
            find_first(production[k][2]);
        }
    }
}

```

```

void Array_Manipulation(char ch)
{
    int count;
    for(count = 0; count <= x; count++)
    {
        if(array[count] == ch)
        {
            return;
        }
    }
    array[x++] = ch;
}

```

Output:

Enter Total Number of Productions: 4

Value of Production Number [1]: S=AaAb

Value of Production Number [2]: S=BbBa

Value of Production Number [3]: A=\$

Value of Production Number [4]: B=\$

Enter production Value to Find Follow: S

Follow Value of S: { \$ }

To Continue, Press Y: y

Enter production Value to Find Follow: A

Follow Value of A: { a b }

To Continue, Press Y: y

Enter production Value to Find Follow: B

Follow Value of B: { b a }

To Continue, Press Y: n

Exp. No. 9

Implement a C program to eliminate left recursion from a given CFG.

$S \rightarrow (L) / a$

$L \rightarrow L, S / S$

Program:

```
#include<stdio.h>
#include<string.h>
#define SIZE 10
int main () {
    char non_terminal;
    char beta,alpha;
    int num;
    char production[10][SIZE];
    int index=3; /* starting of the string following "->" */
    printf("Enter Number of Production : ");
    scanf("%d",&num);
    printf("Enter the grammar as E->E-A :\n");
    for(int i=0;i<num;i++){
        scanf("%s",production[i]);
    }
    for(int i=0;i<num;i++){
        printf("\nGRAMMAR : : : %s",production[i]);
        non_terminal=production[i][0];
```

```

if(non_terminal==production[i][index]) {
    alpha=production[i][index+1];
    printf(" is left recursive.\n");
    while(production[i][index]!=0 && production[i][index]!='|')
        index++;
    if(production[i][index]!=0) {
        beta=production[i][index+1];
        printf("Grammar without left recursion:\n");
        printf("%c->%c%c'",non_terminal,beta,non_terminal);
        printf("\n%c'->%c%c'|E\n",non_terminal,alpha,non_terminal);
    }
    else
        printf(" can't be reduced\n");
}
else
    printf(" is not left recursive.\n");
index=3;
}
}

```

Output:

Enter Number of Production : 2

Enter the grammar as E->E-A :

S->(L)|a

L->L,S|S

GRAMMAR : : : S->(L)|a is not left recursive.

GRAMMAR : : : L->L,S|S is left recursive.

Grammar without left recursion:

L->SL'

L'->,L'|E

Exp. No. 10

Implement a C program to eliminate left factoring from a given CFG.

$S \rightarrow iEtS / iEtSeS / a$

$E \rightarrow b$

Program:

```
#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 : S->");
    gets(gram);
    for(i=0;gram[i]!='\0';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++]='\0';
    for(i=pos;part2[i]!='\0';i++,j++){
        newGram[j]=part2[i];
    }
}
```

```

    }
    modifiedGram[k]='X';
    modifiedGram[++k]='\0';
    newGram[j]='\0';
    printf("\n S->%s",modifiedGram);
    printf("\n X->%s\n",newGram);
}

```

Output:

Enter Production : S->iEtS|iEtSeS|a

S->iEtSX

X->|eS|a

Exp. No. 11

Implement a C program to perform symbol table operations.

Program:

```

#include<stdio.h>
#include<stdlib.h>
#include<string.h>
int cnt=0;
struct symtab
{
    char label[20];
    int addr;
}
sy[50];
void insert();
int search(char *);
void display();
void modify();
int main()
{
    int ch,val;
    char lab[10];

```

```

do
{
    printf("\n1.insert\n2.display\n3.search\n4.modify\n5.exit\n");
    scanf("%d",&ch);
    switch(ch)
    {
        case 1:
            insert();
            break;
        case 2:
            display();
            break;
        case 3:
            printf("enter the label");
            scanf("%s",lab);
            val=search(lab);
            if(val==1)
                printf("label is found");
            else
                printf("label is not found");
            break;
        case 4:
            modify();
            break;
        case 5:
            exit(0);
            break;
    }
}while(ch<5);
}

void insert()
{
    int val;
    char lab[10];
    int symbol;
    printf("enter the label");
    scanf("%s",lab);

```

```

        val=search(lab);
        if(val==1)
            printf("duplicate symbol");
        else
        {
            strcpy(sy[cnt].label,lab);
            printf("enter the address");
            scanf("%d",&sy[cnt].addr);
            cnt++;
        }
    }
    int search(char *s)
    {
        int flag=0,i; for(i=0;i<cnt;i++)
        {
            if(strcmp(sy[i].label,s)==0)
                flag=1;
        }
        return flag;
    }
    void modify()
    {
        int val,ad,i;
        char lab[10];
        printf("enter the labe:");
        scanf("%s",lab);
        val=search(lab);
        if(val==0)
            printf("no such symbol");
        else
        {
            printf("label is found \n");
            printf("enter the address");
            scanf("%d",&ad);
            for(i=0;i<cnt;i++)
            {
                if(strcmp(sy[i].label,lab)==0)

```

```

        sy[i].addr=ad;
    }
}
void display()
{
    int i;
    for(i=0;i<cnt;i++)
        printf("%s\t%d\n",sy[i].label,sy[i].addr);
}

```

Output:

```

1.insert
2.display
3.search
4.modify
5.exit
1
enter the label a
enter the address 100

```

```

1.insert
2.display
3.search
4.modify
5.exit
2
a    100

```

```

1.insert
2.display
3.search
4.modify
5.exit
3

```


enter the label a

label is found

1.insert

2.display

3.search

4.modify

5.exit

4

enter the labe: a

label is found

enter the address 200

1.insert

2.display

3.search

4.modify

5.exit

2

a 200

1.insert

2.display

3.search

4.modify

5.exit

5

Exp. No. 12

Write a C program to construct recursive descent parsing for the given grammar

$E \rightarrow TE'$

$E' \rightarrow +TE' / \epsilon$

$T \rightarrow FT'$

$T' \rightarrow *FT' / \epsilon$

$F \rightarrow (E) / id$

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);
}
```

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

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

String is not accepted

Exp. No. 13

Write a C program to implement either Top Down parsing technique or Bottom Up Parsing technique to check whether the given input string is satisfying the grammar or not.

Program:

```
#include<stdio.h>  
#include<conio.h>  
#include<string.h>  
int main() {  
    char string[50];  
    int flag,count=0;  
    printf("The grammar is: S->aS, S->Sb, S->ab\n");  
    printf("Enter the string to be checked:\n");  
    gets(string);  
    if(string[0]=='a') {  
        flag=0;  
        for (count=1;string[count-1]!='\0';count++) {  
            if(string[count]=='b') {  
                flag=1;  
                continue;  
            } else if((flag==1)&&(string[count]=='a')) {
```

```

        printf("The string does not belong to the specified
grammar");
        break;
    } else if(string[count]=='a')
        continue; else if((flag==1)&&(string[count]!='\0')) {
        printf("String not accepted.....!!!!");
        break;
    } else {
        printf("String accepted");
    }
}
}
}
}

```

Output:

The grammar is: $S \rightarrow aS$, $S \rightarrow Sb$, $S \rightarrow ab$

Enter the string to be checked:

abb

String accepted

Exp. No. 14

Implement the concept of Shift reduce parsing in C Programming.

Program:

```

#include<stdio.h>
#include<stdlib.h>
#include<conio.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(); int main()
{
    //clrscr();
    printf("\n\t\tSHIFT REDUCE PARSER\n"); printf("\n GRAMMER\n");
    printf("\n E->E+E\n E->E/E"); printf("\n E->E*E\n E->a/b"); printf("\n enter the
input symbol:\t"); gets(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$\t\t\t%s",stack,ip_sym,act); strcpy(act,"shift");
temp[0]=ip_sym[ip_ptr]; temp[1]='\0'; strcat(act,temp); check();
st_ptr++;
}
st_ptr++; check();
}
void check()
{
int flag=0; temp2[0]=stack[st_ptr]; temp2[1]='\0';
if((!strcmpi(temp2,"a"))||(!strcmpi(temp2,"b")))
{
stack[st_ptr]='E'; if(!strcmpi(temp2,"a"))
printf("\n %s\t\t%s$\t\t\tE->a",stack,ip_sym); else
printf("\n %s\t\t%s$\t\t\tE->b",stack,ip_sym); flag=1;
}
if((!strcmpi(temp2,"+"))||(!strcmpi(temp2,"*"))||(!strcmpi(temp2,"/")))
{
flag=1;
}
if((!strcmpi(stack,"E+E"))||(!strcmpi(stack,"E\E"))||(!strcmpi(stack,"E*E")))
{
strcpy(stack,"E"); st_ptr=0; if(!strcmpi(stack,"E+E"))
printf("\n %s\t\t%s$\t\t\tE->E+E",stack,ip_sym); else
if(!strcmpi(stack,"E\E"))
printf("\n %s\t\t%s$\t\t\tE->E\E",stack,ip_sym); else
if(!strcmpi(stack,"E*E"))
printf("\n %s\t\t%s$\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(!strcmpi(stack,"E")&&ip_ptr==len)
{
printf("\n $%s\t\t%s$\t\t\tACCEPT",stack,ip_sym); getch();
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

$E \rightarrow E + E$

$E \rightarrow E / E$

$E \rightarrow E * E$

$E \rightarrow a / b$

enter the input symbol: a+b

stack implementation table

stack	input symbol	action
\$	a+b\$	--
\$a	+b\$	shift a
\$E	+b\$	$E \rightarrow a$
\$E+	b\$	shift+
\$E+b	\$	shiftb
\$E+E	\$	$E \rightarrow b$
\$E	\$	$E \rightarrow E + E$
\$E	\$	ACCEPT

Exp. No. 15

Write a C Program to implement the operator precedence parsing.

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"};
//(E) becomes )E( when pushed to stack

int top=0,|;
char prec[9][9]={

    /*input*/

    /*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);
        }
    }
}

if(strcmp(stack,"$E$")==0)
    printf("\nAccepted;");
else
    printf("\nNot Accepted;");
}

```

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
 \$\$ Shift
 \$\$ Shift
 Accepted;

Exp. No. 16

Write a C Program to Generate the Three address code representation for the given input statement.

Program:

```

#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
#include<string.h>
struct three
{
char data[10],temp[7];
}s[30];
int 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();
}

```

Output:

Input: sum.txt

out = in1 + in2 + in3 - in4

Output: out.txt

t1=in1+in2

t2=t1+in3

t3=t2-in4

out=t3

Exp. No. 17

Write a C program for implementing a Lexical Analyzer to Scan and Count the number of characters, words, and lines in a file.

Program:

```
#include <stdio.h>
int main()
{
    char str[100]; //input string with size 100

    int words=0, newline=0, characters=0; // counter variables

    scanf("%[^\n]", &str); //scanf formatting

    for(int i=0; str[i]!='\0'; i++)
    {
        if(str[i] == ' ')
        {
            words++;
        }
        else if(str[i] == '\n')
        {
            newline++;
            words++; //since with every next line new words start. corner case 1
        }
        else if(str[i] != ' ' && str[i] != '\n'){
            characters++;
        }
    }
    if(characters > 0) //Corner case 2,3.
    {
        words++;
        newline++;
    }
    printf("Total number of words : %d\n", words);
```

```

printf("Total number of lines : %d\n",newline);
printf("Total number of characters : %d\n",characters);
return 0;
}

```

Output:

```
void main()
```

```
{
```

```
int a;
```

```
int b;
```

```
a = b + c;
```

```
c = d * e;
```

```
}
```

Total number of words : 18

Total number of lines : 7

Exp. No. 18

Write a C program to implement the back end of the compiler.

Program:

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

```
#include<conio.h>
```

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

```
int main()
```

```
{
```

```
    int n,i,j;
```

```
    char a[50][50];
```

```
    printf("enter the no: intermediate code:");
```

```
    scanf("%d",&n);
```

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

```
    {
```

```
        printf("enter the 3 address code:%d:",i+1);
```

```
        for(j=0;j<6;j++)
```

```
        {
```

```
            scanf("%c",&a[i][j]);
```

```
        }
```

```
    }
```

```
    printf("the generated code is:");
```

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



```

{
    printf("\n mov %c,R%d",a[i][3],i);
    if(a[i][4]=='-')
    {
        printf("\n sub %c,R%d",a[i][5],i);
    }
    if(a[i][4]=='+')
    {
        printf("\n add %c,R%d",a[i][5],i);
    }
    if(a[i][4]=='*')
    {
        printf("\n mul %c,R%d",a[i][5],i);
    }
    if(a[i][4]=='/')
    {
        printf("\n div %c,R%d",a[i][5],i);
    }
    printf("\n mov R%d,%c",i,a[i][1]);
    printf("\n");
}
return 0;
}

```

Output:

enter the no: intermediate code:2

enter the 3 address code:1:a=b+c

enter the 3 address code:2:d=n*d

the generated code is:

mov b,R0

add c,R0

mov R0,a

mov n,R1

mul d,R1

mov R1,d

Exp. No. 19

Write a C program to compute LEADING() – operator precedence parser for the given grammar

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid \text{id}$

Program:

```
#include<conio.h>
```

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

```
char arr[18][3]={{'E', '+', 'F'},{'E', '*', 'F'},{'E', '(', 'F'}, {'E', ')', 'F'},{'E', 'i', 'F'},{'E', '$', 'F'},
{'F', '+', 'F'},{'F', '*', 'F'},{'F', '(', 'F'},{'F', ')', 'F'},{'F', 'i', 'F'},{'F', '$', 'F'}, {'T', '+', 'F'},
{'T', '*', 'F'}, {'T', '(', 'F'},{'T', ')', 'F'},{'T', 'i', 'F'},{'T', '$', 'F'}};
```

```
char prod[] = "EETTF";
```

```
char res[6][3]={{'E', '+', 'T'}, {'T', '\0'}, {'T', '*', 'F'}, {'F', '\0'}, {'(', 'E', ')'}, {'i', '\0'}};
```

```
char stack [5][2];
```

```
int top = -1;
```

```
void install(char pro, char re) {
```

```
    int i;
```

```
    for (i = 0; i < 18; ++i) {
```

```
        if (arr[i][0] == pro && arr[i][1] == re) {
```

```
            arr[i][2] = 'T';
```

```
            break;
```

```
        }
```

```
    }
```

```
    ++top;
```

```
    stack[top][0] = pro;
```

```
    stack[top][1] = re;
```

```
}
```

```
int main() {
```

```
    int i = 0, j;
```

```
    char pro, re, pri = ' ';
```

```
    for (i = 0; i < 6; ++i) {
```

```
        for (j = 0; j < 3 && res[i][j] != '\0'; ++j) {
```

```

        if (res[i][j] == '+' || res[i][j] == '*' || res[i][j] == '(' || res[i][j] == ')' || res[i][j] ==
'i' || res[i][j] == '$') {
            install(prod[i], res[i][j]);
            break;
        }
    }
}
while (top >= 0) {
    pro = stack[top][0];
    re = stack[top][1];
    --top;
    for (i = 0; i < 6; ++i) {
        if (res[i][0] == pro && res[i][0] != prod[i]) {
            install(prod[i], re);
        }
    }
}
for (i = 0; i < 18; ++i) {
    printf("\n\t");
    for (j = 0; j < 3; ++j)
        printf("%c\t", arr[i][j]);
}
getch();
printf("\n\n");
for (i = 0; i < 18; ++i) {
    if (pri != arr[i][0]) {
        pri = arr[i][0];
        printf("\n\t%c -> ", pri);
    }
    if (arr[i][2] == 'T')
        printf("%c ", arr[i][1]);
}
getch();
}

```

Output:

```
E  +  T
E  *  T
E  (  T
E  )  F
E  i  T
E  $  F
F  +  F
F  *  F
F  (  T
F  )  F
F  i  T
F  $  F
T  +  F
T  *  T
T  (  T
T  )  F
T  i  T
T  $  F
```

$E \rightarrow + * (i$

$F \rightarrow (i$

$T \rightarrow * (i$

Exp. No. 20

Write a C program to compute TRAILING() – operator precedence parser for the given grammar

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid id$

Program:

```
#include<conio.h>
```

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

```
char arr[18][3] ={{'E', '+', 'F'}, {'E', '*', 'F'}, {'E', '(', 'F'}, {'E', ')', 'F'}, {'E', 'i', 'F'},
{'E', '$', 'F'}, {'F', '+', 'F'}, {'F', '*', 'F'}, {'F', '(', 'F'}, {'F', ')', 'F'}, {'F', 'i', 'F'},
{'F', '$', 'F'}, {'T', '+', 'F'}, {'T', '*', 'F'}, {'T', '(', 'F'}, {'T', ')', 'F'}, {'T', 'i', 'F'},
```

```

    {'T', '$', 'F'},
};
char prod[6] = "EETTF";
char res[6][3] = { {'E', '+', 'T'}, {'T', '\0', '\0'}, {'T', '*', 'F'}, {'F', '\0', '\0'}, {'(', 'E', ')'}, {'i', '\0', '\0'} };
char stack[5][2];
int top = -1;

void install(char pro, char re) {
    int i;
    for (i = 0; i < 18; ++i) {
        if (arr[i][0] == pro && arr[i][1] == re) {
            }
        }
    ++top;
    arr[i][2] = 'T';

    stack[top][0] = pro;
    stack[top][1] = re;
}

int main() {
    int i = 0, j;
    char pro, re, pri = ' ';

    for (i = 0; i < 6; ++i) {
        for (j = 2; j >= 0; --j) {

            if (res[i][j] == '+' || res[i][j] == '*' || res[i][j] == '(' || res[i][j] == ')' || res[i][j] == 'i' || res[i][j] == '$') {
                install(prod[i], res[i][j]);
                break;
            } else if (res[i][j] == 'E' || res[i][j] == 'F' || res[i][j] == 'T') {
                if (res[i][j - 1] == '+' || res[i][j - 1] == '*' || res[i][j - 1] == '(' || res[i][j - 1] == ')' || res[i][j - 1] == 'i' || res[i][j - 1] == '$') {
                    install(prod[i], res[i][j - 1]);
                    break;
                }
            }
        }
    }
}

```

```

    }
}
}

while (top >= 0) {
    pro = stack[top][0];
    re = stack[top][1];
    --top;
    for (i = 0; i < 6; ++i) {
        for (j = 2; j >= 0; --j) {
            if (res[i][0] == pro && res[i][0] != prod[i]) {
                install(prod[i], re);
                break;
            } else if (res[i][0] != '\0') break;
        }
    }
}

for (i = 0; i < 18; ++i) {
    printf("\n\t");
    for (j = 0; j < 3; ++j)
        printf("%c\t", arr[i][j]);
}

printf("\n\n");
for (i = 0; i < 18; ++i) {
    if (pri != arr[i][0]) {
        pri = arr[i][0];
        printf("\n\t%c -> ", pri);
    }
    if (arr[i][2] == 'T')
        printf("%c ", arr[i][1]);
}
}

```

Output:

```

E   +   F
E   *   F
E   (   F
E   )   F
E   i   F
E   $   F

```

```

F   +   F
F   *   F
F   (   F
F   )   F
F   i   F
F   $   F
T   +   F
T   *   F
T   (   F
T   )   F
T   i   F
T   $   F

```

E ->

F ->

T ->

Exp. No. 21

Write a LEX specification file to take input C program from a .c file and count the number of characters, number of lines & number of words.

Input Source Program: (sample.c)

```

#include <stdio.h>
int main()
{
    int number1, number2, sum;
    printf("Enter two integers: ");
    scanf("%d %d", &number1, &number2);
    sum = number1 + number2;
    printf("%d + %d = %d", number1, number2, sum);
    return 0;
}

```

Program: (count_lines.l)

```

%{
int nchar, nword, nline;
}%
%%
\n { nline++; nchar++; }
[^ \t\n]+ { nword++; nchar += yyleng; }
. { nchar++; }
%%

```

```

int yywrap(void) {
return 1;
}
int main(int argc, char *argv[]) {
yyin = fopen(argv[1], "r");
yylex();
printf("Number of characters = %d\n", nchar);
printf("Number of words = %d\n", nword);
printf("Number of lines = %d\n", nline);
fclose(yyin);
}

```

Output:

G:\lex>flex count_line.l

G:\lex>gcc lex.yy.c

G:\lex>a.exe sample.c

Number of characters = 233

Number of words = 33

Number of lines = 10

G:\lex>

Exp. No. 22

Write a LEX program to print all the constants in the given C source program file.

Input Source Program: (sample.c)

```

#define P 314
#include<stdio.h>
#include<conio.h>
void main()
{

    int a,b,c = 30;
    printf("hello");
}

```

Program: (countconstants.l)

digit [0-9]

%{

int cons=0;

%}

%%


```

{digit}+ { cons++; printf("%s is a constant\n", yytext); }
.\n {}
%%
int yywrap(void) {
return 1; }
int main(void)
{
FILE *f;
char file[10];
printf("Enter File Name : ");
scanf("%s",file);
f = fopen(file,"r");
yyin = f;
yylex();
printf("Number of Constants : %d\n", cons);
fclose(yyin);
}

```

Output:

G:\lex>flex countconstants.l

G:\lex>gcc lex.yy.c

G:\lex>a.exe

Enter File Name : sample.c

314 is a constant

30 is a constant

Number of Constants : 2

G:\lex>

Exp. No. 23

Write a LEX program to count the number of Macros defined and header files included in the C program.

Input Source Program: (sample.c)

```

#define PI 3.14
#include<stdio.h>
#include<conio.h>
void main()
{

```

```
int a,b,c = 30;
printf("hello");
}
```

Program: (count_macro.l)

```
%{
int nmacro, nheader;
%}
%%
^#define { nmacro++; }
^#include { nheader++; }
.|\\n { }
%%
int yywrap(void) {
return 1;
}
int main(int argc, char *argv[]) {
yyin = fopen(argv[1], "r");
yylex();
printf("Number of macros defined = %d\\n", nmacro);
printf("Number of header files included = %d\\n", nheader);
fclose(yyin);
}
```

Output:

G:\lex>flex count_macro.l

G:\lex>gcc lex.yy.c

G:\lex>a.exe sample.c

Number of macros defined = 1

Number of header files included = 2

G:\lex>

Exp. No. 24

Write a LEX program to print all HTML tags in the input file.

Input Source Program: (sample.html)

```
<html>
<body>
<h1>My First Heading</h1>
<p>My first paragraph.</p>
</body>
</html>
```

Program: (html.l)

```
%{
int tags;
%}
%%
"<"[^>]*> { tags++; printf("%s \n", yytext); }
.|\\n { }
%%
int yywrap(void) {
return 1; }
int main(void)
{
FILE *f;
char file[10];
printf("Enter File Name : ");
scanf("%s",file);
f = fopen(file,"r");
yyin = f;
yylex();
printf("\n Number of html tags: %d",tags);
fclose(yyin);
}
```

Output:

```
G:\lex>flex html.l
```

```
G:\lex>gcc lex.yy.c
```

```
G:\lex>a.exe
```

```
Enter File Name : sample.html
```

```
<html>
<body>
<h1>
</h1>
<p>
</p>
</body>
</html>
```

Number of html tags: 8

G:\lex>

Exp. No. 25

Write a LEX program which adds line numbers to the given C program file and display the same in the standard output.

Input Source Program: (sample.c)

```
#define PI 3.14
#include<stdio.h>
#include<conio.h>
void main()
{

    int a,b,c = 30;

    printf("hello");
}
```

Program: (addlinenos.l)

```
%{
int yylineno;
}%
%%
^(.*)\n printf("%4d\t%s", ++yylineno, yytext);
%%
int yywrap(void) {
return 1;
}
int main(int argc, char *argv[]) {
yyin = fopen(argv[1], "r");
yylex();
fclose(yyin);
}
```

Output:

```
G:\lex>flex addlinenos.l
```

```
G:\lex>gcc lex.yy.c
```

```
G:\lex>a.exe sample.c
```

```
1  #define PI 3.14
2  #include<stdio.h>
3  #include<conio.h>
4  void main()
5  {
6  int a,b,c = 30;
7  printf("hello");
8  }
9
```

```
G:\lex>
```

Exp. No. 26

Write a LEX program to count the number of comment lines in a given C program and eliminate them and write into another file.

Input Source File: (input.c)

```
#include<stdio.h>
int main()
{
    int a,b,c; /*variable declaration*/
    printf("enter two numbers");
    scanf("%d %d",&a,&b);
    c=a+b;//adding two numbers
    printf("sum is %d",c);
    return 0;
}
```

Program: (comment.l)

```
%{
int com=0;
%}
%s COMMENT
%%
"/*" {BEGIN COMMENT;}
```

```

<COMMENT>"*/" {BEGIN 0; com++;}
<COMMENT>\n {com++;}
<COMMENT>. {;}
\\V.* {; com++;}
.|\\n {fprintf(yyout,"%s",yytext);}
%%
void main(int argc, char *argv[])
{
if(argc!=3)
{
printf("usage : a.exe input.c output.c\n");
exit(0);
}
yyin=fopen(argv[1],"r");
yyout=fopen(argv[2],"w");
yylex();
printf("\n number of comments are = %d\n",com);
}
int yywrap()
{
return 1;
}

```

Output:

G:\lex>flex comment.l

G:\lex>gcc lex.yy.c

G:\lex>a.exe input.c
usage : a.exe input.c output.c

G:\lex>a.exe input.c output.c

number of comments are = 2

G:\lex>

Output File: (output.c)

```
include<stdio.h>
int main()
{
int a,b,c;
printf("enter two numbers");
scanf("%d %d",&a,&b);
c=a+b;
printf("sum is %d",c);
return 0;
}
```

Exp. No. 27

Write a LEX program to identify the capital words from the given input.

Program: (capital.l)

```
%%
[A-Z]+[\t\n ] { printf("%s is a capital word\n",yytext); }
. ;
%%

int main( )
{
    printf("Enter String :\n");
    yylex();
}
int yywrap( )
{
    return 1;
}
```

Output:

G:\lex>flex capital.l

G:\lex>gcc lex.yy.c

G:\lex>a.exe

Enter String :

CAPITAL of INDIA is DELHI
CAPITAL is a capital word
INDIA is a capital word
DELHI
is a capital word

G:\lex>

Exp. No. 28

Write a LEX Program to check the email address is valid or not.

Program: (email_valid.l)

```
%{  
int flag=0;  
%}  
%%  
[a-z . 0-9]+@[a-z]+".com"|" .in" { flag=1; }  
%%  
int main()  
{  
yylex();  
if(flag==1)  
printf("Accepted");  
else  
printf("Not Accepted");  
}  
int yywrap()  
{ return 1;  
}
```

Output:

G:\lex>flex email_valid.l

G:\lex>gcc lex.yy.c

G:\lex>a.exe
sse123@gmail.com

Accepted

G:\lex>

Exp. No. 29

Write a LEX Program to convert the substring abc to ABC from the given input string

Program: (substring.l)

```
%{
int i;
%}
%%
[a-z A-Z]* { for(i=0;i<=yyleng;i++)
    { if((yytext[i]=='a')&&(yytext[i+1]=='b')&&(yytext[i+2]=='c'))
        { yytext[i]='A';
          yytext[i+1]='B';
          yytext[i+2]='C';
        }
    }
    printf("%s",yytext);
}

[\\t]* return 1;
.* {ECHO;}
\\n {printf("%s",yytext);}
%%
int main()
{
    yylex();
}
int yywrap()
{
    return 1;
}
```

Output:

G:\lex>flex substring.l

G:\lex>gcc lex.yy.c

G:\lex>a.exe

abcdefghijkl
ABCdefghABCijkla

G:\lex>

Exp. No. 30

Implement a LEX program to check whether the mobile number is valid or not.

Program: (mobile.l)

```
%%  
[1-9][0-9]{9} {printf("\nMobile Number Valid\n");}  
.+ {printf("\nMobile Number Invalid\n");}  
%%  
int main()  
{  
    printf("\nEnter Mobile Number : ");  
    yylex();  
    printf("\n");  
    return 0;  
}  
int yywrap()  
{ }
```

Output:

G:\lex>flex mobile.l

G:\lex>gcc lex.yy.c

G:\lex>a.exe

Enter Mobile Number : 7856453489

Mobile Number Valid

G:\lex>

Exp. No. 31

Implement Lexical Analyzer using FLEX (Fast Lexical Analyzer). The program should separate the tokens in the given C program and display with appropriate caption.

Input Source Program: (sample.c)

```
#include<stdio.h>

void main()
{
    int a,b,c = 30;

    printf("hello");
}
```

Program: (token.l)

```
digit [0-9]
letter [A-Za-z]
%{
int count_id,count_key;
%}
%%
(stdio.h|conio.h) { printf("%s is a standard library\n",yytext); }
(include|void|main|printf|int) { printf("%s is a keyword\n",yytext); count_key++; }
{letter}({letter}|{digit})* { printf("%s is a identifier\n", yytext); count_id++; }
{digit}+ { printf("%s is a number\n", yytext); }
\"(\\.|[^\"])*\" { printf("%s is a string literal\n", yytext); }
.|\\n { }
%%
int yywrap(void) {
return 1;
}
int main(int argc, char *argv[]) {
yyin = fopen(argv[1], "r");
yylex();
printf("number of identifiers = %d\n", count_id);
printf("number of keywords = %d\n", count_key);
fclose(yyin);
}
```

Output:

G:\lex>flex token.l

```
G:\lex>gcc lex.yy.c
```

```
G:\lex>a.exe sample.c
```

include is a keyword

stdio.h is a standard library

void is a keyword

main is a keyword

int is a keyword

a is a identifier

b is a identifier

c is a identifier

30 is a number

printf is a keyword

"hello" is a string literal

number of identifiers = 3

number of keywords = 5

```
G:\lex>
```

Exp. No. 32

Write a LEX program to count the number of vowels in the given sentence.

Program: (vowels.l)

```
%{
```

```
    int vow_count=0;
```

```
    int const_count =0;
```

```
%}
```

```
%%
```

```
[aeiouAEIOU] {vow_count++;}
```

```
[a-zA-Z] {const_count++;}
```

```
%%
```

```
int yywrap(){}
```

```
int main()
```

```
{
```

```
    printf("Enter the string of vowels and consonants:");
```

```
    yylex();
```

```
    printf("Number of vowels are: %d\n", vow_count);
```

```
    printf("Number of consonants are: %d\n", const_count);
```

```
    return 0;
```

```
}
```

Output:

```
G:\lex>flex vowels.l
```

```
G:\lex>gcc lex.yy.c
```

```
G:\lex>a.exe
```

Enter the string of vowels and consonants: Vowel sounds allow the air to flow freely, causing the chin to drop noticeably, whilst consonant sounds are produced by restricting the air flow

```
      ,      ,  
Number of vowels are: 42  
Number of consonants are: 77  
^C  
G:\lex>
```

Exp. No. 33

Write a LEX program to count the number of vowels in the given sentence.

(Refer the program and output of experiment 32, both are same)

Exp. No. 34

Write a LEX program to separate the keywords and identifiers.

(Refer the program and output of experiment 31, both are same)

Exp. No. 35

Write a LEX program to recognise numbers and words in a statement.

Program: (numbers_words.l)

```
%%  
[\t ]+ ;  
[0-9]+|[0-9]*\.[0-9]+ { printf("\n%s is NUMBER", yytext);}  
#.* { printf("\n%s is COMMENT", yytext);}  
[a-zA-Z]+ { printf("\n%s is WORD", yytext);}  
\n { ECHO;}  
%%  
int main()  
{  
    while( yylex());  
}
```

```

}

int yywrap( )
{
    return 1;
}

```

Output:

G:\lex>flex numbers_words.l

G:\lex>gcc lex.yy.c

G:\lex>a.exe

Variables A and B contains 10 and 20 respectively

Variables is WORD

A is WORD

and is WORD

B is WORD

contains is WORD

10 is NUMBER

and is WORD

20 is NUMBER

respectively is WORD

G:\lex>

Exp. No. 36

Write a LEX program to identify and count positive and negative numbers.

Program: (positive_neg_nums.l)

```

%{
int positive_no = 0, negative_no = 0;
%}
%%
^[-][0-9]+ {negative_no++;
                printf("negative number = %s\n",
                    yytext);} // negative number

```

```

[0-9]+ {positive_no++;
        printf("positive number = %s\n",
               yytext);} // positive number
%%
int yywrap(){
int main()
{
yylex();
printf ("number of positive numbers = %d,"
        "number of negative numbers = %d\n",
        positive_no, negative_no);

return 0;
}

```

Output:

G:\lex>flex positive_neg_nums.l

G:\lex>gcc lex.yy.c

G:\lex>a.exe

-10

negative number = -10

20

positive number = 20

number of positive numbers = 1,number of negative numbers = 1

G:\lex>

Exp. No. 37

Write a LEX program to validate the URL.

Program: (url.l)

```

%%
((http)|(ftp))s?:\\/[a-zA-Z0-9](.[a-z])+(.[a-zA-Z0-9+=?]*)* {printf("\nURL Valid\n");}

```

```
.+ {printf("\nURL Invalid\n");}
```

```
%%
```

```
void main()
```

```
{
```

```
    printf("\nEnter URL : ");
```

```
    yylex();
```

```
    printf("\n");
```

```
}
```

```
int yywrap()
```

```
{
```

```
}
```

Output:

```
G:\lex>flex url.l
```

```
G:\lex>gcc lex.yy.c
```

```
G:\lex>a.exe
```

```
Enter URL : https:\\www.sse.in
```

```
URL Invalid
```

```
https://www.sse.in
```

```
URL Valid
```

```
G:\lex>
```

Exp. No. 38

Write a LEX program to validate DOB of students.

Program: (dob.l)

```
%%
```

```
(([0-1-9])|([1-2][0-9])|(3[0-1]))V((0[1-9])|(1[0-2]))V(19[0-9]{2}|2[0-9]{3})
```

```
printf("Valid DoB");
```

```
.* printf("Invalid DoB");
```



```
%%
```

```
int main()
{
    yylex();
    return 0;
}
int yywrap()
{}
```

Output:

```
G:\lex>flex dob.l
```

```
G:\lex>gcc lex.yy.c
```

```
G:\lex>a.exe
```

```
26/07/1995
```

```
Valid DoB
```

```
13\2\96
```

```
Invalid DoB
```

```
G:\lex>
```

Exp. No. 39

Write a LEX program to check whether the given input is digit or not.

Program: (digit_or_not.l)

```
%%
```

```
[0-9]+ {printf("\nValid digit \n");}
```

```
. * printf("\nInvalid digit\n");
```

```
%%
```

```
int yywrap(){}
```

```
int main()
```

```
{
```

```
    yylex();
```

```
    return 0;
```

```
}
```

Output:

G:\lex>flex digit_or_not.l

G:\lex>gcc lex.yy.c

G:\lex>a.exe

23

Valid digit

h56

Invalid digit

G:\lex>

Exp. No. 40

Write a LEX program to implement basic mathematical operations.

Program: (cal.l)

```
%{
#undef yywrap
#define yywrap() 1
int f1=0,f2=0;
char oper;
float op1=0,op2=0,ans=0;
void eval();
%}

DIGIT [0-9]
NUM {DIGIT}+(\.{DIGIT})?
OP [*/+ -]

%%

{NUM} {
    if(f1==0)
    {
        op1=atof(yytext);
        f1=1;
    }
}
```

```

        else if(f2==-1)
        {
            op2=atof(yytext);
            f2=1;
        }

        if((f1==1) && (f2==1))
        {
            eval();
            f1=0;
            f2=0;
        }
    }
}

```

```

{OP} {

    oper=(char) *yytext;
    f2=-1;
}

```

```

[\n] {

    if(f1==1 && f2==1)
    {
        eval;
        f1=0;
        f2=0;
    }
}

```

```

%%

```

```

int main()
{
    yylex();
}

```

```

void eval()
{
    switch(oper)
    {
        case '+':
            ans=op1+op2;
            break;

        case '-':
            ans=op1-op2;
            break;

        case '*':
            ans=op1*op2;
            break;

        case '/':
            if(op2==0)
            {
                printf("ERROR");
                return;
            }
            else
            {
                ans=op1/op2;
            }
            break;
        default:
            printf("operation not available");
            break;
    }
    printf("The answer is = %lf",ans);
}

```

Output:

G:\lex>flex cal.l

G:\lex>gcc lex.yy.c

G:\lex>a.exe

20 + 30

The answer is = 50.000000

25 * 5

The answer is = 125.000000

G:\lex>