COMPILER DESIGN LAB

NAME: DHUNESHWARAN V

REG NO: 192324169

COURSE CODE: CSA1455

SERIAL NO: 10

# 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]);

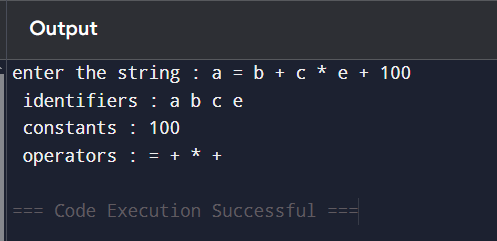
}

printf("\n operators : "); for(j=0;j<oc;j++)

{

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

}

}

# 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]=='/')

{

}

else

}

printf("\n It is a comment"); a=1;

break;

continue;

}

else

}

else

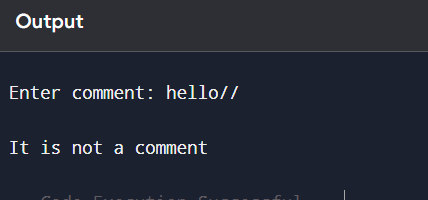
if(a==0)

printf("\n It is not a comment");

printf("\n It is not a comment");

printf("\n 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("flex\_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:** flex\_input.txt main ( )

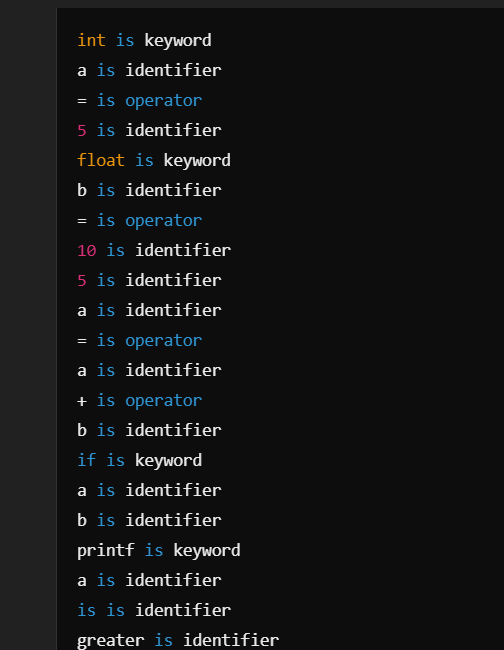
{

int a, b, c ; c = b + c;

printf ( "%d" ,c ) ;

}

**Output:**



# 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("\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");

}

}

## 

# 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];

int words = 0, lines = 0, characters = 0;

printf("Enter text (up to 100 characters, use ~ to end):\n"); scanf("%[^~]", str);

for (int i = 0; str[i] != '\0'; i++) {

if (str[i] == ' ' || str[i] == '\t') { words++;

} else if (str[i] == '\n') { lines++;

} else {

characters++;

}

}

// Check for an empty input if (characters > 0) {

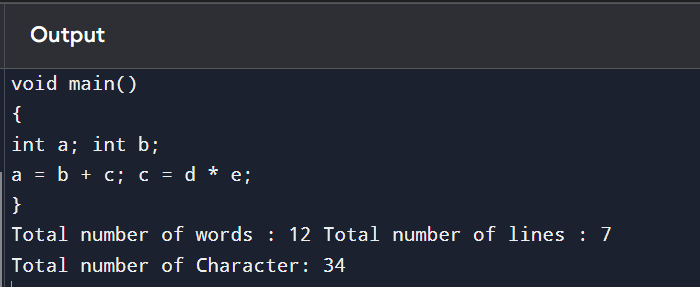
words++; // If there are characters, there is at least one word lines++; // If there are characters, there is at least one line

}

printf("Total number of words: %d\n", words); printf("Total number of lines: %d\n", lines); printf("Total number of characters: %d\n", characters);

return 0;

}



# Exp. No. 6

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

## Program:

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

int main() { char a[10];

int flag = 1, i = 1;

printf("\nEnter an identifier: "); fgets(a, sizeof(a), stdin);

if (isalpha(a[0])) {

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

if (!isdigit(a[i]) && !isalpha(a[i])) { flag = 0;

break;

} i++;

}

} else {

flag = 0;

}

if (flag == 1) {

printf("\nValid identifier\n");

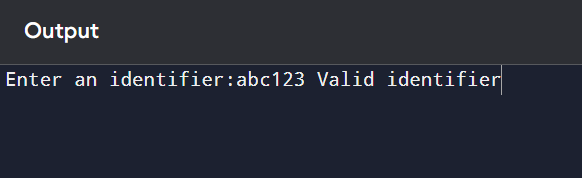
} else {

printf("\nNot a valid identifier\n");

}

return 0;

}



# Exp. No. 7

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

S → AaAb / BbBa

A → ∈

B → ∈

**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 → ε is a production, then add ε to FIRST(X). if(productionSet[i][2]=='$') addToResultSet(Result,'$');

//If X is a non-terminal, and X → Y1 Y2 … Yk

//is a production, then add a to FIRST(X)

//if for some i, a is in FIRST(Yi),

//and ε is in all of FIRST(Y1), …, FIRST(Yi-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 ε 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';

}

# 

# Exp. No. 8

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

S → AaAb / BbBa

A → ∈

B → ∈

**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]);

}

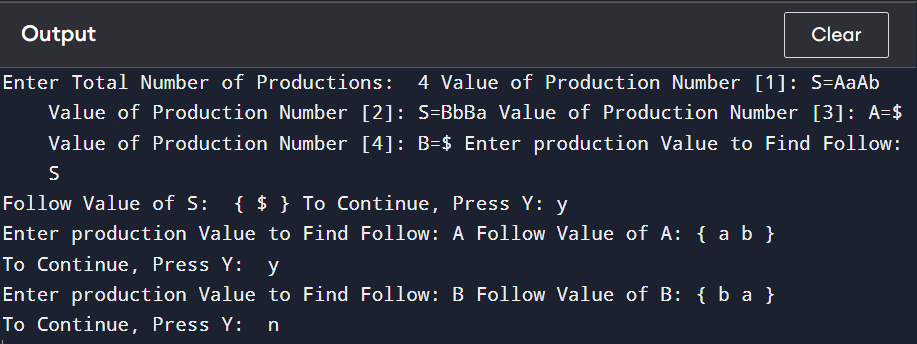
}

}

}

void Array\_Manipulation(char ch)

{



# Exp. No. 9

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

S → (L) / a L → 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");

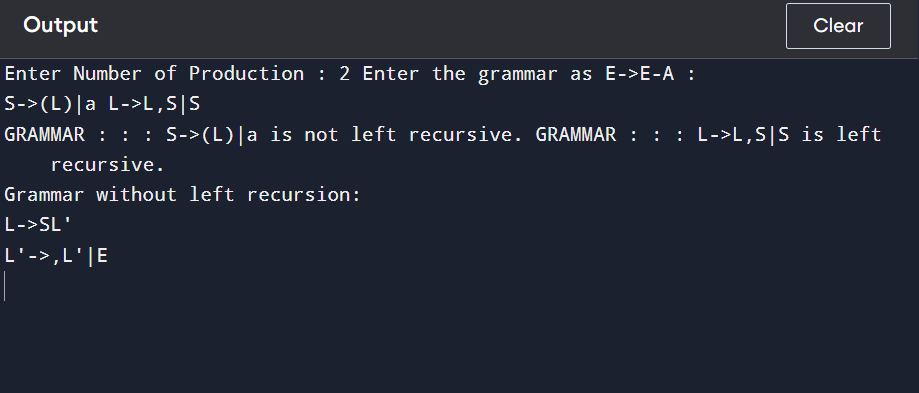
}

else

printf(" is not left recursive.\n"); index=3;

}

}



# Exp. No. 10

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

S → iEtS / iEtSeS / a E → b

**Program:** #include<stdio.h> #include<string.h>

int main()

{

char gram[20], part1[20], part2[20], modifiedGram[20], newGram[20]; int i, j = 0, k = 0, l = 0, pos;

// Input production printf("Enter Production: S->"); gets(gram);

// Extract part1 and part2

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';

// Find common prefix

for(i = 0; part1[i] == part2[i]; i++)

{

modifiedGram[k] = part1[i]; k++;

pos = i + 1;

}

// Create modified production modifiedGram[k] = 'X';

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

// Create new production

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];

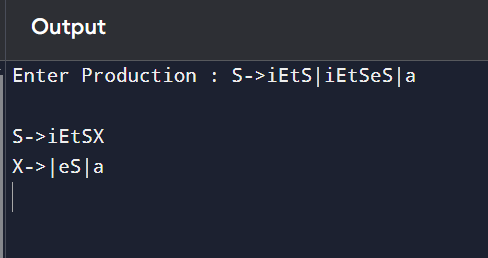
newGram[j] = '\0';

// Print the result

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

return 0;

}



# 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;

}

}

}

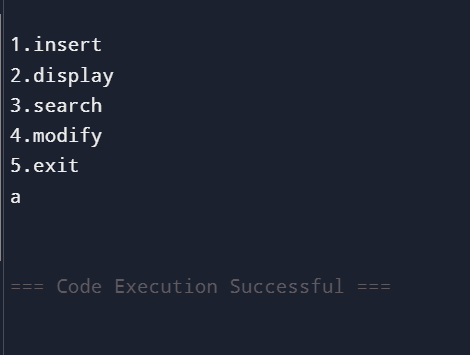
void display()

{

int i; for(i=0;i<cnt;i++)

printf("%s\t%d\n",sy[i].label,sy[i].addr);

}



# Exp. No. 12

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

E → TE’

E’ → +TE’ / ∈

T → FT’

T’ → \*FT’ / ∈

F → ( 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')

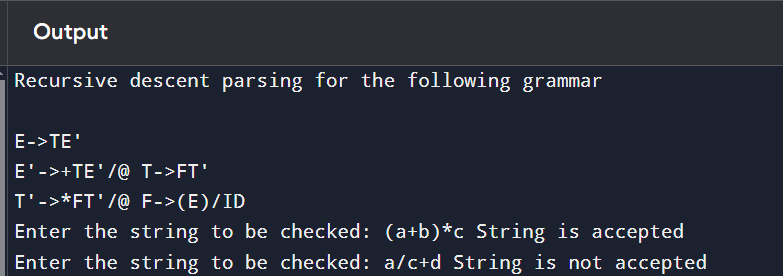
{ i++;

return(1);

}

else return(0);

}



# 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");

}

}

}

}

## 

# 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\t SHIFT 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)

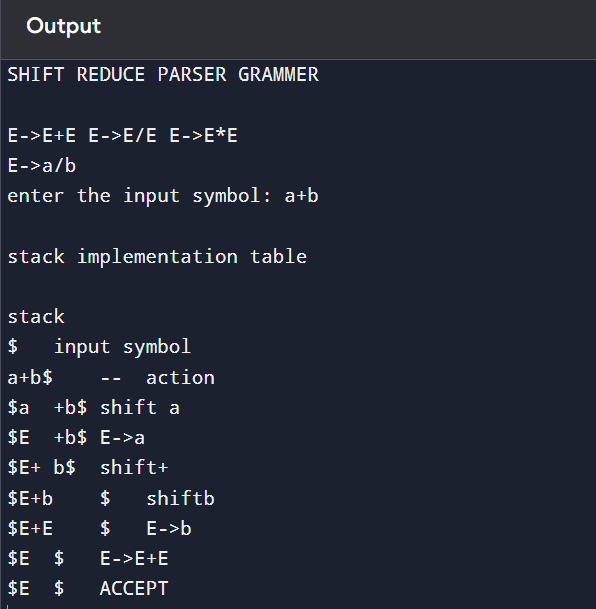
{

printf("\n%s\t\t\t%s\t\t reject",stack,ip\_sym); exit(0);

}

return;

}



# 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,l;

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);

}

}

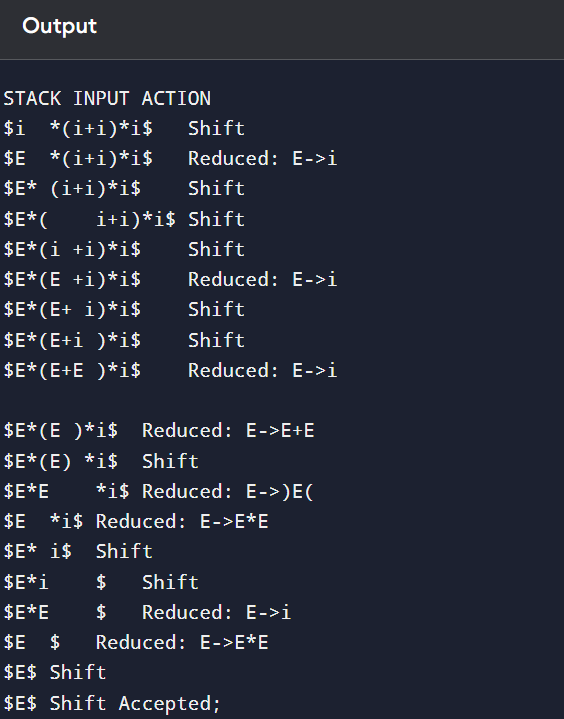
}

if(strcmp(stack,"$E$")==0) printf("\nAccepted;");

else

printf("\nNot Accepted;");

}



# 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();

}

## 

# 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("%[^~]",&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;

}

## 

# 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;

}

## 

# Exp. No. 19

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

E → E + T | T

T → T \* F | F F → ( E ) | 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[] = "EETTFF";

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);

}

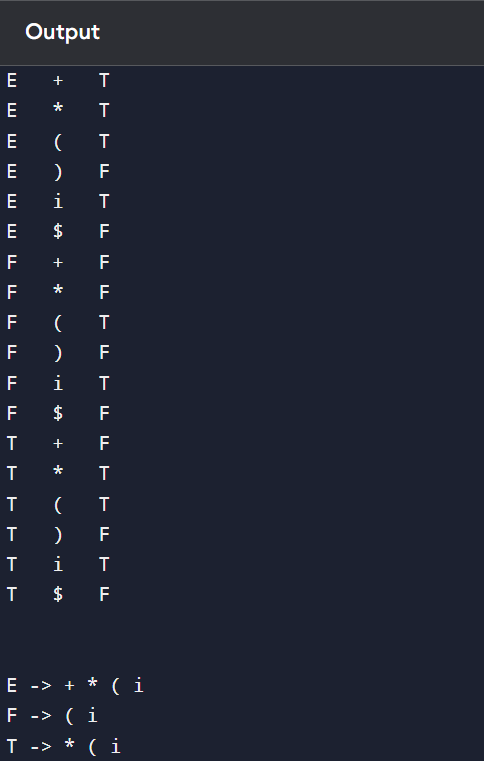
if (arr[i][2] == 'T')

printf("%c ", arr[i][1]);

}

getch();

}



# Exp. No. 20

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

**E → E + T | T**

**T → T \* F | F F → ( E ) | 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] = "EETTFF";

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;

break; // Added break to exit the loop when the match is found

}

}

}

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];

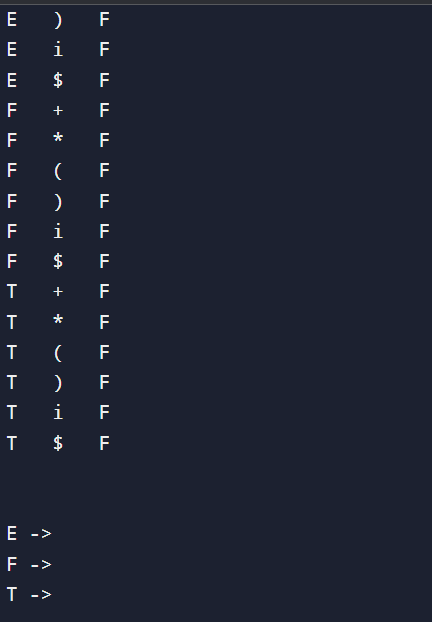
printf("\n\t%c -> ", pri);

}

if (arr[i][2] == 'T')

printf("%c ", arr[i][1]);}

}



# Exp. No. 21

## Write a LEX specification file to take input C program from a .c file and count tthe 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);

}

## 

# 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);

}

## 

# Exp. No. 23

## Write a LEX program to count the number of Macros defined and header filesincluded 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);

}

## 