**Ex. No. :1 IMPLEMENTATION OF LEXICAL ANALYZER IN “C”**

**AIM**

To write a C program to implement a lexical analyzer.

**ALGORITHM**

**Step 1:** Input the string from the file.

**Step 2:** Check if the first character of the string is an alphabet.

1. If yes, then extract the string of alphabets in an array.
2. Else, go to step 4.

**Step 3:** Compare the array with the predefined set of keywords and check for a match.

1. If a match is found, then print “keyword”.
2. Else print “Identifier”.

**Step 4:** If the character is not an alphabet, compare it with the set of operator and check

for a match.

1. If yes, print the operator.
2. Else print “error”.

**PROGRAM**

#include<stdio.h>

#include<ctype.h>

#include<string.h>

void keyw(char \*p);

int i=0,id=0,kw=0,num=0,op=0;

char keys[40][15]={"auto","break","case","char","const","continue","default",

"do","double","else","enum","extern","float","for","goto",

"if","int","long","register","return","short","signed","main","getch","clrscr",

"sizeof","static","struct","switch","typedef","union","printf",

"unsigned","void","volatile","while"};

main()

{

char ch,str[25],seps[15]=" \t\n,;(){}[]#\"<>";

char oper[]="!%^&\*-+=~|.<>/?";

int j;

char fname[50];

FILE \*f1;

clrscr();

printf("enter file path (drive:\\fold\\filename)\n");

scanf("%s",fname);

f1 = fopen(fname,"r");

if(f1==NULL)

{

printf("file not found");

getch();

exit(0);

}

while((ch=fgetc(f1))!=EOF)

{

for(j=0;j<=14;j++)

{

if(ch==oper[j])

{

printf("%c is an operator\n",ch);

op++;

str[i]='\0';

keyw(str);

}

}

for(j=0;j<=14;j++)

{

if(i==-1)

break;

if(ch==seps[j])

{

if(ch=='#')

{

while(ch!='>')

{

printf("%c",ch);

ch=fgetc(f1);

}

printf("%c is a header file\n",ch);

i=-1;

break;

}

if(ch=='"')

{ do

{

ch=fgetc(f1);

printf("%c",ch);

}while(ch!='"');

printf("\b is an argument\n");

i=-1;

break;

}

str[i]='\0';

keyw(str);

}

}

if(i!=-1)

{

str[i]=ch;

i++;

}

else

i=0;

}

printf("Keywords: %d\nIdentifiers: %d\nOperators: %d\nNumbers: %d\n",kw,id,op,num);

getch();

}

void keyw(char \*p)

{

int k,flag=0;

for(k=0;k<=31;k++)

{

if(strcmp(keys[k],p)==0)

{

printf("%s is a keyword\n",p);

kw++;

flag=1;

break;

}

}

if(flag==0)

{

if(isdigit(p[0]))

{

printf("%s is a number\n",p);

num++;

}

else

{

if(p[0]!='\0')

{

printf("%s is an identifier\n",p);

id++;

}

}

}

i=-1;

}

**INPUT FILE**

**V1.C**

main()

{

printf("HELLO");

getch();

return(0);

}

**OUTPUT**

enter file path (drive:\fold\filename)

V1.C

main is a keyword

printf is a keyword

HELLO is an argument

getch is a keyword

return is a keyword

0 is a number

Keywords: 4

Identifiers: 0

Operators: 0

Numbers: 1

**RESULT**

Thus above C program to implement Lexical Analyzer was successfully executed and the output was verified.

**EX. NO: 2 IMPLEMENTATION OF LEXICAL ANALYSER USING LEX TOOL**

**AIM**

To write a ‘C’ program for implementing the Lexical Analyzer using LEX Tool.

**ALGORITHM**

**Step 1:** Start the Program

Step 2: Create a Lex Specification file using the Echo,YYLEX, YYTEXT, YYWRAPAP, YYIN

**Step 3:** The Echo() function is used to emit the input as it is.

**Step 4:** The YYLEX() function encounters scanning the source program.

**Step 5:** The YYTEXT() function is used to store the Null Terminated string.

**Step 6:** The YYWRAP() function returns zero when the scanner continues scanning and returns

‘1’ when the end of file is encountered.

**Step 7:** The YYIN() function is used to store the input source program.

**Step 8:** The main function first calls the yylex and yywrap() function and executes.

**Step 9:** Print the Preprocessor Directive ,Keyword,function as in the source program.

**Step10:** Stop the Program.

**PROGRAM**

%{

%}

%%

[\t]+

#.\*{printf("\n%s is a Preprocessor directive",yytext);}

void|

main"()"|

int|

float|

char|

printf|

scanf|

if|

while|

for|

do|

getch ()|

return |

else {printf("\n%s is a Keyword",yytext);}

\{ {printf("\n%s BLOCK BEGINS HERE",yytext);}

\}{printf("\n%s BLOCK ENDS HERE",yytext);}

\(|

\){printf("\n %s is a symbol",yytext);}

\+|

\-|

\\*|

\/|

\<|

\>|

\!=|

\==|

\={printf("\n %s is an operator",yytext);}

[a-zA-Z][a-zA-Z0-9]\* {printf("\n%s is a variable",yytext);}

\".\*\"{printf("\n%s is a string",yytext);}

[0-9]\* {printf("\n%s is a constant",yytext);}

\/\\*.\*\\*\/|

\/\/.\* {printf("\n%s is a comment",yytext);}

%%

int main(int argc,char \*\*argv)

{

FILE \*fp=fopen(argv[1],"r");

yyin=fp;

yylex();

return 0;

}

int yywrap()

{

return 0;

}

**Input File**

**add.c**

#include<stdio.h>

#include<conio.h>

void main()

{

clrscr();

int a=5,b=7,c;

c=a+b;

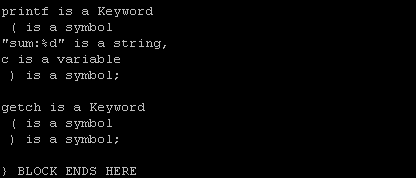
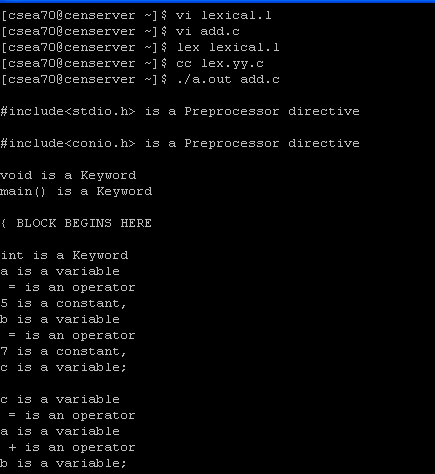
printf(“%d sum:”);

scanf(“%d”,&c);

getch();

}

**OUTPUT**



**RESULT**

Thus the above Program for implementing Lexical Analyzer using LEX tool was executed and the output was verified successfully.

**Ex.No. :3 IMPLEMENTATION OF THE RECURSIVE DECENT PARSER**

**AIM**

To write a C program to implement a recursive descent parser.

**ALGORITHM**

**Step 1:** Get the input expression.

**Step 2:** Construct the parser tree virtually for the following grammar:

E->TE’

E’->+TE’| €

T->FT’

T’->\*FT’ | €

F->I | (E)

Where each non-terminal has a defined function corresponding it.

**Step 3:** Call the function corresponding to the start symbol.

**Step 4:** For each non-terminal on the LHS of the production do,

1. If the RHS has a non-terminal, then call the function corresponding to it.
2. If the RHS has a terminal, then check if it matches with the input, else print error.

**Step 5:** Repeat the previous step until the given input string is completely parsed or an error is encountered.

**Step 6**: Stop the program

**PROGRAM**

#include<stdio.h>

#include<string.h>

char input[10];

int i=0,error=0;

void E();

void T();

void Eprime();

void Tprime();

void F();

main()

{

clrscr();

printf("\n\tRecursive Decent Parser\n\t-----------------------");

printf("\n\t Grammar Without Left Recursion");

printf("\n\t\t E->TE'\n\t\tE'->+TE'|e\n\t\tT->FT'");

printf("\n\t\tT'->FT'|e\n\t\tF->(E)|j");

printf("\n\t Enter an Arithmetic Expression:");

gets(input);

E();

if(strlen(input)==i&&error==0)

printf("\n\t The Given string is Accepted...!!!");

else

printf("\n\t The Given string is Rejected...!!!");

getch();

return (0);

}

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(input[i]=='j')

i++;

else if(input[i]=='(')

{

i++;

E();

if(input[i]==')')

i++;

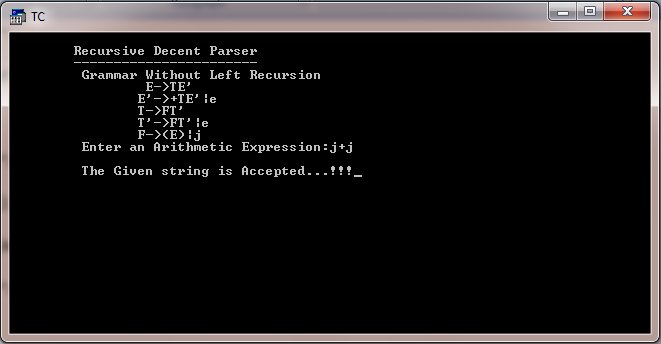
}

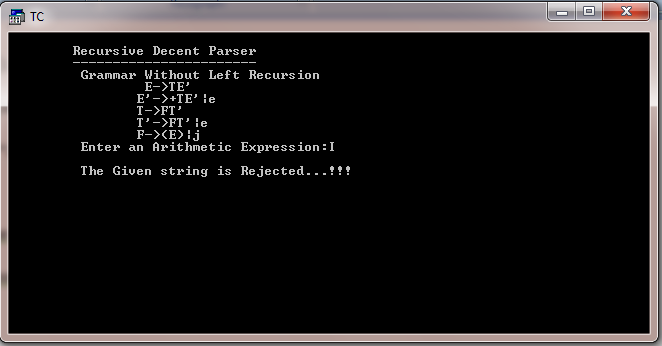
else

error=1;

}

**OUTPUT**

****

****

**RESULT**

Thus the above C-program to implement the recursive descent parser was executed and the output was verified.

**Ex. No. 4 : IMPLEMENTATION OF A PARSER USING LEX & YACC TOOL**

**AIM**

To write a LEX and YACC program to implement parser.

**ALGORITHM**

**Step 1: In parser.l give the LEX specification.**

1. In the declaration section given the regular expression for an identifier and number.
2. Identify the tokens as keywords or variables or numbers or data types or relational operators and return them to y.tab.h
3. Skip the white spaces.
4. Increment the line number for every new line.

**Step 2: In parser.y give the syntax specification**

1. Define tokens as specified in sample.l and the arithmetic operators as left associative.
2. Specify the productions i.e. the syntax for a program block, code, statement, condition. For example, the syntax for program is PROGRAM: MAIN BLOCK; which means the program starts with main, followed by a block.
3. Give the auxiliary code and initialize the parsing using yyparse().
4. Keep track of the error number to check if the program is error free or not. If error =0 then the program is error free else it has errors.
5. Print the line number having the error and error no. is incremented for every error using yyerror().

**Parse.l**

%{

#include<stdlib.h>

#include "y.tab.h"

void yyerror(char \*);

%}

%%

[0-9] { printf("%s\tNumber \n",yytext); return INTEGER;}

[-+/\*] {printf("%s\tOperator \n",yytext); return \*yytext;}

[\n] return \*yytext;

[\t];

yyerror("Invalid character");

%%

**Parse.y**

%{

#include<stdio.h>

void yyerror(char \*);

%}

%token '+' '\*' '-' '/'

%token INTEGER

%%

S:

E '\n' { printf("\n Accepted"); exit(1);}

;

E:

E'+'T

|E'-'T

|T

;

T:

T '\*' F|

T'/'F

|F

;

F:

'('E')'

| INTEGER

;

%%

void yyerror(char \*s)

{

printf("%s - Invalid String \n",s);

exit(1);

}

int main()

{

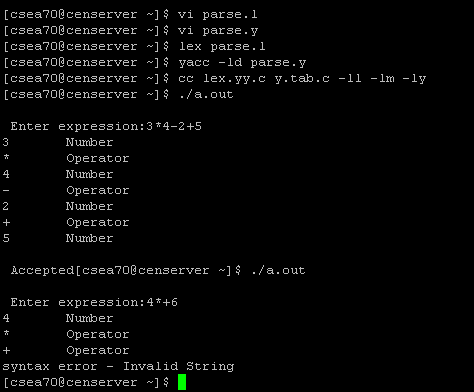
printf("\n Enter expression:");

yyparse();

return 0;

}

**OUTPUT**



**RESULT**

Thus the above program to implement parser using LEX and YACC was executed and the output was verified.

**Ex. No. 5 : IMPLEMENTATION OF CALCULATOR USING YACC**

**AIM**

To write a yacc program for performing calculator functions like addition and multiplication.

**ALGORITHM**

**Step 1**: In the declaration section, include the header files stdlio.h and ctype.h

**Step 2**: Define the tokens.

**Step 3:** Define the precedence and left associatively of the operators.

**Step 4:** In the rule section, give the productions for program, statement, expr. For example, expr may be INTEGER, VARIABLE, EXPR’+’EXPR, EXPR’-‘EXPR, EXPR’\*’EXPR, EXPR’/’EXPR, ‘(‘EXPR’)’.

**Step 5:** The evaluated value of the arithmetic expression is then stored in $$.

**PROGRAM**

%{

#include<stdio.h>

#include<ctype.h>

%}

%token DIGIT

%%

lines:expr'\n'{printf("%d\n",$1);}

;

expr:expr'+'term{$$=$1+$3;}

|term

;

|term:term'\*'factor{$$=$1+$3;}

|factor

;

factor:'('expr')'{$$=$2;}

|DIGIT

;

%%

yylex()

{

int c;

c=getchar();

if(isdigit(c))

{

yylval=c='0';

return DIGIT;

}

return c;

}

int main()

{

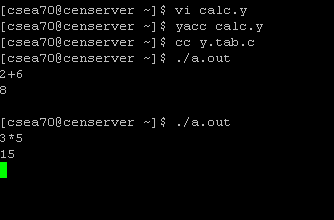
yyparse();

}

yyerror()

{}

**OUTPUT**



**RESULT**

Thus the above Yacc program to perform calculator functions like addition and multiplication was implemented and output was verified.

**Ex. No. 6 : IMPLEMENTATION OF INTERMEDIATE CODE GENERATION**

**(FRONT END OF A COMPILER)**

**AIM**

To write a ‘C’ program for implement the intermediate code for the given expression.

**ALGORITHM**

**Step 1:** Start the program

**Step 2:** Input the expression

**Step 3:** Checks the expression for its validation.

**Step 4:** If it is invalid, return the error message, otherwise, for each computation store the result

in the three address statement. (store it in temporary variable say t1,t2,t3 etc.,)

**Step 5:** Assign the final temporary value to the variable in which the result has to be stored.

**Step 12:** Stop the Program.

**PROGRAM**

#include<stdio.h>

#include<ctype.h>

#include<stdlib.h>

#include<conio.h>

#include<string.h>

void small();

void done(int );

int p[5]={0,1,2,3,4},c=1,i,k,l,m,pi;

char sw[5]={'=','-','+','/','\*'},j[20],a[5],b[5],ch[2];

void main()

{

clrscr();

printf("Enter the expression:");

scanf("%s",j);

printf("\n\n\tThe Intermediate code is:\n");

small();

}

void done(int i)

{

a[0]='\0';b[0]='\0';

if(!isdigit(j[i+2]) && !isdigit(j[i-2]))

{

a[0]=j[i-1];

b[0]=j[i+1];

}

if(isdigit(j[i+2]))

{

a[0]=j[i-1];

b[0]='t';

b[1]=j[i+2];

}

if(isdigit(j[i-2]))

{ b[0]=j[i+1];

a[0]='t';

a[1]=j[i-2];

b[1]='\0';

}

if(isdigit(j[i+2]) && isdigit(j[i-2]))

{ a[0]='t';

b[0]='t';

a[1]=j[i-2];

b[1]=j[i+2];

itoa(c,ch,10);

j[i+2]=j[i-2]=ch[0];

}

if(j[i]=='\*')

printf("\tt%d=%s\*%s\n",c,a,b);

if(j[i]=='/')

printf("\tt%d=%s/%s\n",c,a,b);

if(j[i]=='+')

printf("\tt%d=%s+%s\n",c,a,b);

if(j[i]=='-')

printf("\tt%d=%s-%s\n",c,a,b);

if(j[i]=='=')

printf("\t%c=t%d",j[i-1],--c);

itoa(c,ch,10);

j[i]=ch[0];

c++;

small();

}

void small()

{ pi=0;l=0;

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

{ for(m=0;m<5;m++)

if(j[i]==sw[m])

if(pi<=p[m])

{

pi=p[m];

l=1;

k=i;

} }

if(l==1)

done(k);

else

{

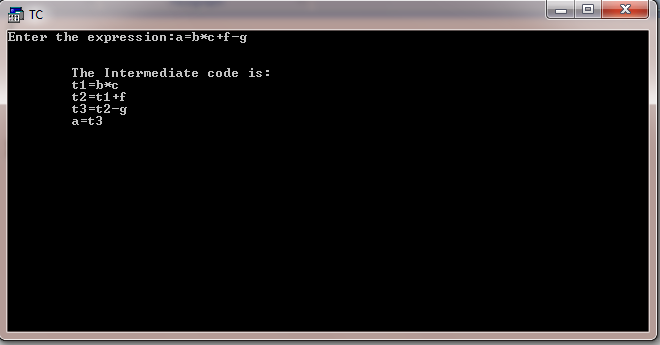
getch();

exit (0);

}

}

**OUTPUT**

****

**RESULT**

Thus the above C program to implement the intermediate code for the expression was executed and output was verified.

**Ex. No. : 7 IMPLEMENTATION OF ELIMINATION OF LEFT RECURSION**

**AIM**

To write a c program to eliminate left recursion for given production.

**ALGORITHM**

**Step 1:** Start the program**.**

**Step 2:** Get the production from the user.

**Step 3:** For the production of the form,

*A → Aα | β*

*means,it has left recursion.*

**Step 4:** To eliminate left recursion ,the above production can *be replaced by non-left recursive productions,*

*A->βA’*

*A’-> αA’ |e*

**Step 5:** print the productions after eliminating left recursion.

**Step 6:** Stop the program.

**PROGRAM**

#include<stdio.h>

#include<conio.h>

#include<string.h>

void main()

{

char str[25];

int i=0;

clrscr();

printf("------------------Elimination of Left Recursive-------------------\n");

printf("\nEnter the production:");

scanf("%s",str);

printf("\nProductions after eliminating Left Recursion:\n");

if(str[0]==str[3])

{

printf("%c->%c%c"\n\n",str[0],str[strlen(str)-1],str[0]);

printf("%c'->",str[0]);

i=4;

while(str[i]!='/')

printf("%c",str[i++]);

printf("%c'|E\n\n",str[0]);

}

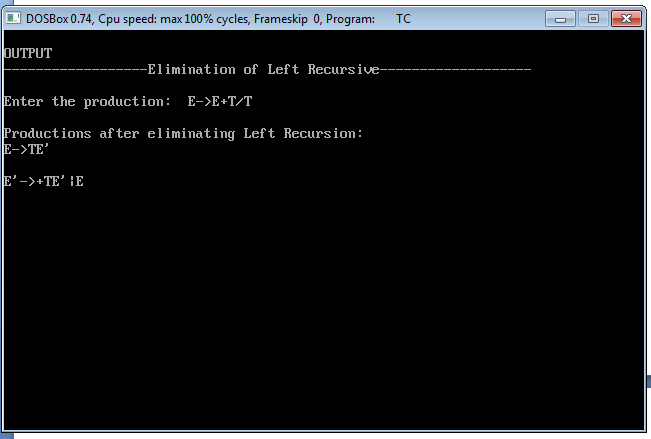
else

printf("%s",str);

getch();

}

**OUTPUT**



**RESULT**

Thus the above C program to eliminate the left recursion in the given production was executed and the output was verified.

**Ex. No. 8 : IMPLEMENTATION OF BACKEND OF A COMPILER**

**(CODE GENERATOR)**

**AIM**

To write a c program to implement the back end of a compiler(code generator)for the given three address code.

**ALGORITHM**

**Step1:**Start the program.

**Step2**:Get the three address code from the user.

**Step3**:Check the length of the given input.

**Step4**:If the given length is greater than 5,then return the error message as invalid expression.

**Step5**:Else,check the operator(+,-,\*,/)and print the corressponding block of mnemonic code of the operator.

**Step6:**Stop the program.

**PROGRAM**

#include<stdio.h>

#include<conio.h>

Void main()

{

Char s[50],i=0;

Clrscr();

Printf(“enter the three address code:”);

Scanf(“%s”,s);

If(strlen(s)==5)

{

While(s[i]!=’\0’)

{

If(s[i]==’/’)

{

Printf(“MOVF%c ,R1\n”,s[i-1]);

Printf(“MOVF%c ,R2\n”,s[i+1]);

Printf(“DIVF%c R1,R2\n”);

}

i++;

}

i=0;

while(s[i]!=’\0’)

{

If(s[i]==’\*’)

{

Printf(“MOVF%c ,R1\n”,s[i-1]);

Printf(“MOVF%c ,R2\n”,s[i+1]);

Printf(“MULF%c R1,R2\n”);

}

i++;

}

i=0;

while(s[i]!=’\0’)

{

If(s[i]==’+’)

{

Printf(“MOVF%c ,R1\n”,s[i-1]);

Printf(“MOVF%c ,R2\n”,s[i+1]);

Printf(“ADDF%c R1,R2\n”);

}

i++;

}

i=0;

while(s[i]!=’\0’)

{

If(s[i]==’-’)

{

Printf(“MOVF%c ,R1\n”,s[i-1]);

Printf(“MOVF%c ,R2\n”,s[i+1]);

Printf(“SUBF%c R1,R2\n”);

}

i++;

}

i=0;

printf(“MOVF R2,%c\n”,s[0]);

}

Else

Printf(“\n Invalid Expression”);

getch();

}

**OUTPUT**









**RESULT**

Thus, the above C program to implement the back end of the compiler was executed and the output was verified.

**Ex.No. 9 : IMPLEMENTATION OF DIRECTED ACYCLIC GRAPH (DAG)**

**AIM**

To write a C-program to implement a Directed Acyclic Graph(DAG).

**ALGORITHM**

**Step 1:** Start the program

**Step 2:** Get the expression from the user.

**Step 3:** Convert the expression in to postfix expression

**Step 4:** Set interior nodes representing operator for the expression.

**Step 5:** Set children representing the operands for the expression.

**Step 6:** Stop the program.

**PROGRAM**

#include<stdio.h>

#include<string.h>

struct dag

{

int no,a1,a2;

char t,c;

}

d[25];

char infix[30],postfix[30],s[30],s1[25],symb,topsymb;

int l1,l2,pos1,pos2,top,top1,k,m=0,t1[25][25];

char pre[5]={'/','\*','+','-'};

void intopos();

void construction();

void display();

void push(char);

char pop();

void push1(int);

int pre1(char,char);

void main()

{

clrscr();

printf("\n Enter the expression:");

scanf("%s",infix);

intopos();

construction();

display();

getch();

}

void intopos()

{

l1=strlen(infix);

pos1=pos2=top=0;

while(pos1<l1)

{

symb=infix[pos1];

if(symb=='a'||symb=='b'||symb=='c'||symb=='d'||symb=='e'||symb=='f')

{

postfix[pos2]=symb;

pos2++;

}

else

{

while(top>0&&pre1(s[top-1],symb))

{

topsymb=pop();

postfix[pos2]=topsymb;

pos2++;

}

if(top==0||symb!=')')

push(symb);

else

topsymb=pop();

}

pos1++;

}

while(top>0)

{

topsymb=pop();

postfix[pos2]=topsymb;

pos2++;

}

l2=strlen(postfix);

}

int pre1(char y,char z)

{

int i,j,temp,temp1;

if(y=='('||z=='(')

return 0;

if(z==')')

return 1;

if(y==')')

{

printf("error");

}

for(i=0;i<3;i++)

if(pre[i]==y)

temp=i;

for(j=0;j<3;j++)

if(pre[j]==z)

temp1=j;

if(((temp==0)&&(temp1==1))||((temp1==0)&&(temp==1)))

return 1;

else if(((temp==2)&&(temp1==3))||((temp1==2)&&(temp==3)))

return 1;

else

{

if(temp>temp1)

return 1;

else if(temp==temp1)

return 1;

else

return 0;

}

}

void construction()

{

int i=0,j,flag,flag1,temp,temp1;

while(i<l2)

{

flag=0;

if((postfix[i]=='a')||(postfix[i]=='b')||(postfix[i]=='c')||(postfix[i]=='d')||(postfix[i]=='e')||(postfix[i]=='f'))

{

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

if(d[j].c==postfix[i])

{

flag=1;

break;

}

if(flag!=1)

{

d[k].no=k+1;

d[k].t='v';

d[k].c=postfix[i];

push1(k+1);

k++;

}

else

push1(j+1);

}

else

{

temp=pop1();

temp1=pop1();

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

if((d[j].t==postfix[i])&&(d[j].a1==temp)&&(d[j].a2==temp1))

{

flag=1;

break;

}

if(flag!=1)

{

d[k].no=k+1;

d[k].t=postfix[i];

d[k].a1=temp;

d[k].a2=temp1;

push1(k+1);

k++;

}

else

push1(j+1);

}

i++;

}

}void push(char item)

{

s[top]=item;

top++;

}

char pop()

{

char item1;

top--;

item1=s[top];

return item1;

}

void push1(int item)

{

s1[top1]=item;

top1++;

}

int pop1()

{

int item;

top1--;

item=s1[top1];

return item;

}

void display()

{

int i;

printf("\n DAG\n");

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

{

if(d[i].t=='v')

printf("\n%d\t%c\t%c\n",d[i].no,d[i].t,d[i].c);

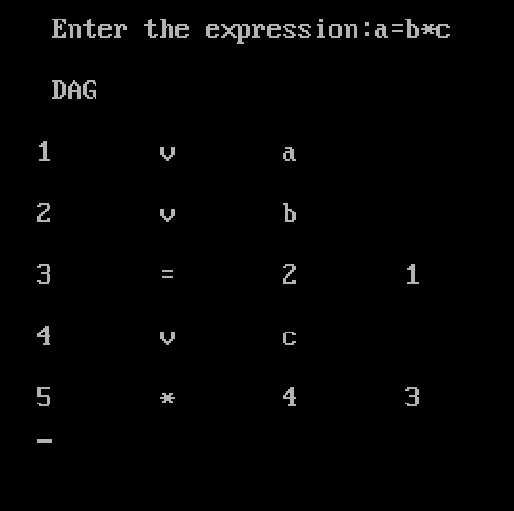
else

printf("\n%d\t%c\t%d\t%d\n",d[i].no,d[i].t,d[i].a1,d[i].a2);

}

}

**OUTPUT**



**RESULT**

Thus the above C program to implement a Directed Acyclic Graph(DAG) was executed and the output was verified.

**Ex.No.10 : CONSTRUCTION OF LR PARSING TABLE**

**AIM**  
  
    To write a C program to implement simple LR Parsing algorithm.  
  
**ALGORITHM**

•    Input: An input string w and an LR parsing table with functions action and goto for a grammar G.  
•    Output: If w is in L(G), a bottom – up parse for w; otherwise an error indication.  
•    Method: Initially, the parser has s0 on its stack, s0 is the initial state, and w$ in the input buffer. The parser then executes the program until accept or error action is encountered.  
         set ip to point to the first symbol of w$;  
        repeat forever begin  
    let s be the state on the top of the stack and  
        a the symbol pointed to by ip;  
    if action [s, a] = shift s′ then begin  
        push a then s′ on the top of the stack;  
        advance ip to the next input symbol  
    end  
    else if action [s, a] = reduce A → β then begin  
        pop 2\*|β| symbols off the stack;  
        let s′ be the state now on top of the stack;  
        push A then goto [s′, A] on top of the stack;  
        output the production A → β  
    end  
    else if action [s, a] = accept then  
        return  
    else error()  
         end

**PROGRAM**

#include<stdio.h>

#include<conio.h>

#include<ctype.h>

#include<string.h>

struct LR

{

int no;

char items[10][10];

}

I[15];

int tab2[15][15]={0,0};

char tab1[15][15],prod[10][10],T[10],NT[10],follow[10][10],startsym;

int n,noitems,not,nont,nof[10];

void getprod(void) ,t\_nt(void),createtab(void),getlritem(void) ,print(void);

void getfollow(void);

int is(char,char[],int);

void main()

{

clrscr();

getprod();

t\_nt();

getfollow();

getlritem();

createtab();

print();

getch();

}

void getlritem(void)

{

int i,j;

printf("Enter the number of LR items:\n");

scanf("%d",&noitems);

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

{

printf("\nNumber of items in I[%d]:",i);

scanf("%d",&I[i].no);

printf("\nEnter the items:\n");

for(j=0;j<I[i].no;j++)

scanf("%s",I[i].items[j]);

}

}

void getprod(void)

{

int i;

printf("Enter the number of productions:");

scanf("%d",&n);

printf("\nEnter the productions:");

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

scanf("%s",prod[i]);

startsym=prod[0][0];

}

void t\_nt(void)

{

int i,j;

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

if(is(prod[i][0],NT,nont)<0)

{

NT[nont]=prod[i][0];

nont++;

}

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

for(j=3;j<strlen(prod[i]);j++)

if(is(prod[i][j],NT,nont)<0)

if(is(prod[i][j],T,not)<0)

{

T[not]=prod[i][j];

not++;

}

T[not]='$';

not++;

printf("\nNT:");

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

printf("%c \t ",NT[i]);

printf("\nT:");

for(i=0;i<not;i++)

printf("%c\t",T[i]);

}

int is(char c,char A[],int x)

{

int y;

for(y=0;y<x;y++)

if(A[y]==c)

return y;

return -1;

}

void getfollow(void)

{

int i,j;

char ch;

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

{

printf("Enter the number of element in follow(%c)",NT[i]);

scanf("%d",&nof[i]);

printf("Enter the follow of %c",NT[i]);

for(j=0;j<nof[i];j++)

{

ch=getch();

follow[i][j]=ch;

printf("%c",follow[i][j]);

}}}

void createtab(void)

{int i,j,k,l,m,p,q,r,y,z,tid,ntid,tno,ntno,len;

char ch1,ch2,ch3,ch4,tempstr[15];

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

for(j=0;j<I[i].no;j++)

{

len=strlen(I[i].items[j]);

for(k=0;k<len;k++)

{

ch1=I[i].items[j][k];

ch2=I[i].items[j][k+1];

tid=is(ch2,T,not);

ntid=is(ch2,NT,nont);

if((ch1=='.')&&(tid>0))

{

for(l=0;l<noitems;l++)

for(m=0;m<I[l].no;m++)

for(p=0;p<strlen(I[l].items[m]);p++)

{

ch3=I[l].items[m][k];

ch4=I[l].items[m][k+1];

if((ch3==ch2) && (ch4==ch1))

{

tab1[i][tid]='s';

tab2[i][tid]=l;

goto lab;

}}}

else if(I[i].items[j][len-1]=='.')

{

if(I[i].items[j][len-2]==startsym)

tab1[i][not-1]='A';

else

{

ntno=is(I[i].items[j][0],NT,nont);

for(q=0;q<nof[ntno];q++)

{

ch3=follow[ntno][q];

tno=is(ch3,T,not);

tab1[i][tno]='R';

z=0;

for(y=0;y<=len;y++)

if(I[i].items[j][y]!='.')

{

tempstr[z]=I[i].items[j][y];

z++;

}

for(r=0;r<n;r++)

if(strcmp(tempstr,prod[r])==0)

break;

tab2[i][tno]=r+1;

}}}

if((ch1=='.')&&(ntid>=0))

{

for(l=0;l<noitems;l++)

for(m=0;m<I[l].no;m++)

for(p=0;p<strlen(I[l].items[m]);p++)

{

ch3=I[l].items[m][k];

ch4=I[l].items[m][k+1];

if((ch3==ch2) && (ch4==ch1))

if(tab2[i][not+ntid]==0)

{

tab2[i][not+ntid]=l;

goto lab;

}}}

lab:;

}}}

void print(void)

{

int i,j,k,l=not+nont;

printf("\nLR Table:\n");

for(i=0;i<not;i++)

printf("\t%c",T[i]);

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

printf("\t%c",NT[j]);

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

{printf("\n%d\t",i);

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

printf("%c%d",tab1[i][j],tab2[i][j]);

}

}

**OUTPUT**

Enter the number of productions:6

Enter the productions:

E->E+T

E->T

E->T\*F

T->F

F->(E)

F->a

NT:E T F

T :+ \* ( ) a $

Enter the number of elements in follow(E):3

Enter the follow(E):+ ) $

Enter the number of elements in follow(T):4

Enter the follow(T):+ \* ) $

Enter the number of elements in follow(F):4

Enter the follow(E):+ \* ) $

Enter the number of LR items:12

Number of items in I[0]: 7

Enter the items:

S->.E

E->.E+T

E->.T

E->.T\*F

T->.F

F->.(E)

F->.a

Number of items in I[1]: 2

Enter the items:

S->E.

E->E.+T

Number of items in I[2]: 2

Enter the items:

E->T.

T->T.\*F

Number of items in I[3]: 1

Enter the items:

T->F.

Number of items in I[4]: 7

Enter the items:

F->(.E)

E->.E+T

E->.T

E->.T\*F

T->.F

F->.(E)

F->.a

Number of items in I[5]: 1

Enter the items:

F->a

Number of items in I[6]: 5

Enter the items:

E->E+.T

T->.T\*F

T->.F

F->.(E)

F->.a

Number of items in I[7]:3

Enter the items:

T->T\*F

F->.(E)

F->.a

Number of items in I[8]:2

Enter the items:

F->(E.)

E->E.+T

Number of items in I[9]:2

Enter the items:

E->E+T.

T->T.\*F

Number of items in I[10]:1

Enter the items:

T->T\*F

Number of items in I[11]:1

Enter the items:

F->(E)

LR Table:

+ \* ( ) a $ E T F

0 0 0 s4 0 s5 0 1 2 3

1 s6 0 0 0 0A 0 0 0 0

2 R2 s7 R2 R2

3 R4 R4 R4 R4

4 s4 s5 8 2 3

5 r6 r6 r6 r6

6 s4 s5 9 3

7 s4 s5 10

8 s6 s11

9 R1 s7 R1 R1

10 r3 r3 r3 r3

11 r5 r5 r5 r5

**RESULT**

Thus the above C program to implement a LR Parsing table was executed and the output was verified.