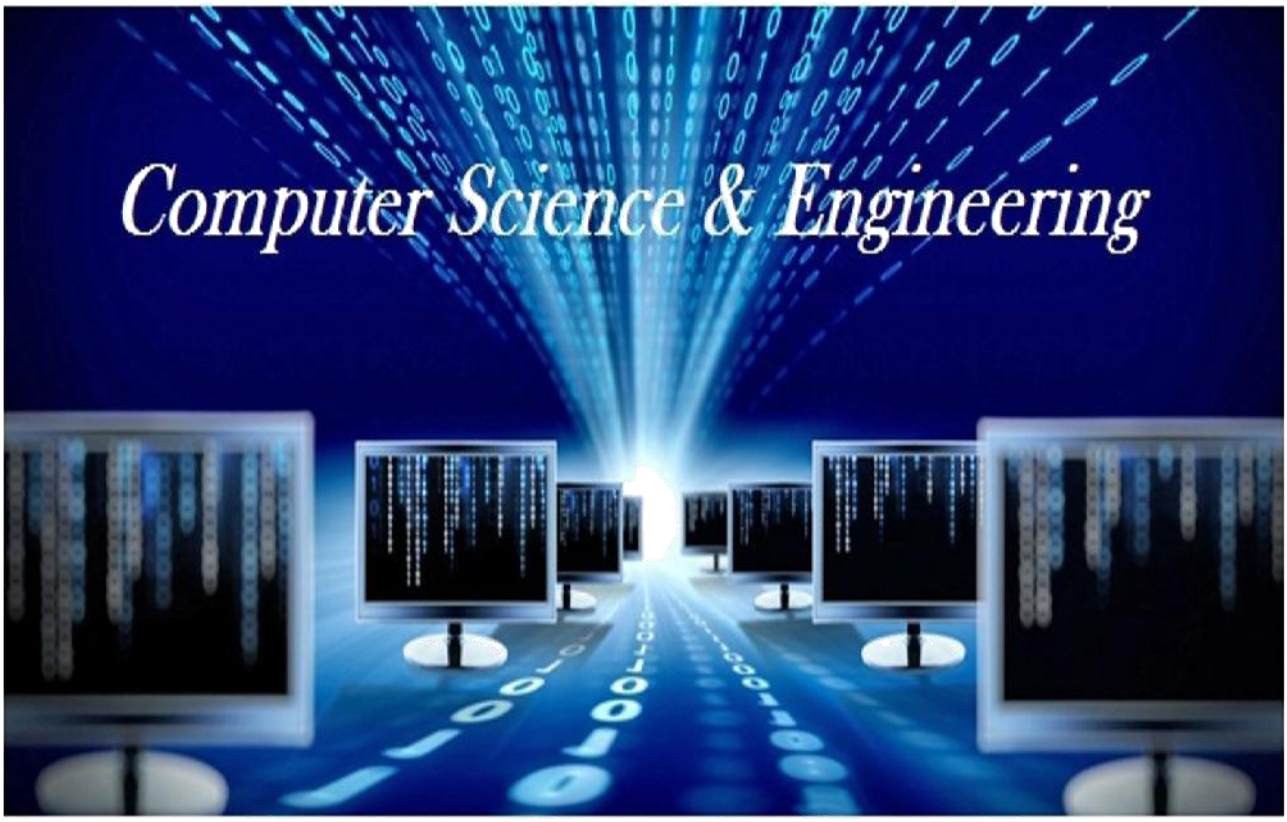
SRI VENKATESWARA COLLEGE OF ENGINEERING & TECHNOLOGY (AUTONOMOUS)
RVS NAGAR, TIRUPATI ROAD, CHITTOOR(AP) – 517127
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

LAB MANUAL

Regulation : 2017
Branch : *B.E.* – CSE
Year & Semester : III Year / II Semester

17ACS31 COMPILER LABORATORY



17ACS31 – COMPILER LABORATORY

LIST OF EXPERIMENTS:

1.Design a Lexical analyzer for the given language. The lexical analyzer should ignore redundant spaces, tabs and newlines. It should also ignore comments. Although the syntax specification states that identifiers can be arbitrarily long, you may restrict the length to some reasonable value.

2.Implement the lexical analyzer using JLex, flex or lex or other lexical analyzer generating tools.

3.Recognition of a valid variable which starts with a letter and followed by any number of letters or Digits.

4. Design Predictive parser for the given language.

5.Design LALR bottom up parser for the given language.

6. Implementation of the symbol table.

7.Implementation of type checking.

8.Implementation of Dynamic Memory Allocatation (Stack,Heap,Static)

9. Construction of a DAG (Directed Acyclic Graph)

10. Implementation of the Backend of the Compiler.

TOTAL: 45 PERIODS

S.NO	NAME OF THE EXPERIMENT	SIGNATURE	REMARKS
1	Lexical analysis recognize in c		
2	Lexical analyser using lex tools		
3	Letter followed by any number of letters or digits		
4	Design Predictive parser for the given language		
5	Design LALR bottom up parser for the given language		
6	Implementation of the symbol table.		
7	Implementation of type checking.		
8	Implementation of Dynamic Memory Allocatation		
9	Construction of a DAG		
10	Implementation of the Backend of the Compiler		

EX. NO:-1
DATE:

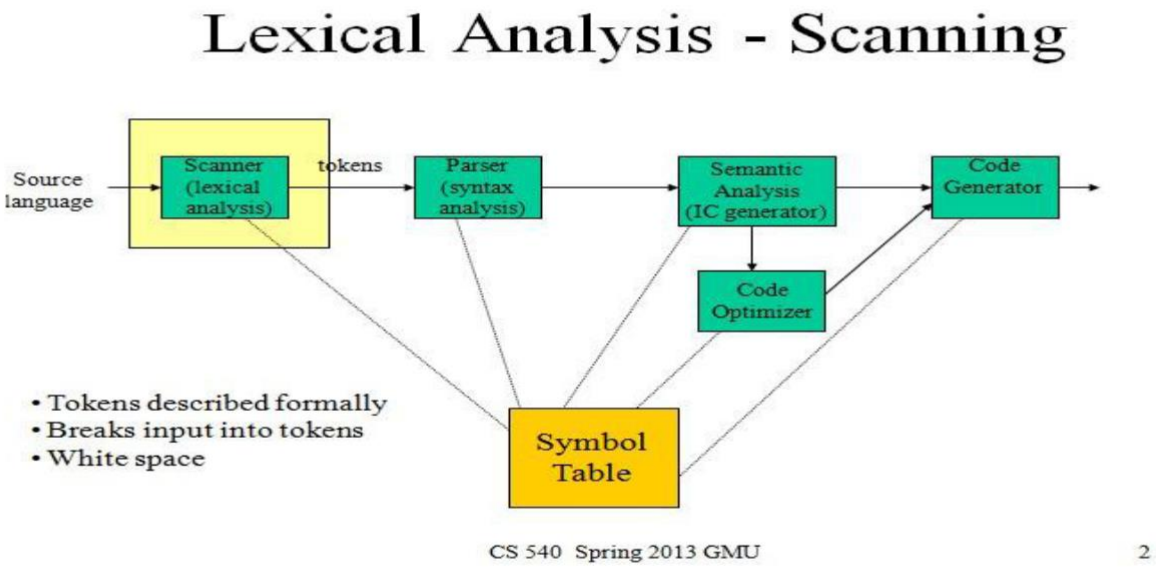
DEVELOP A LEXICAL ANALYZER TO RECOGNIZE A FEW PATTERNS IN C

AIM:

To Write a C program to develop a lexical analyzer to recognize a few patterns in C.

INTRODUCTION:

Lexical analysis is the process of converting a sequence of characters (such as in a computer program of web page) into a sequence of tokens (strings with an identified "meaning"). A program that perform lexical analysis may be called a lexer, tokenize or scanner.



TOKEN

A token is a structure representing a lexeme that explicitly indicates its categorization for the Purpose of parsing. A category of token is what in linguistics might be called a part-of-speech. Examples of token categories may include "identifier" and "integer literal", although the set of Token differ in different programming languages.

The process of forming tokens from an input stream of characters is called tokenization.

Consider this expression in the C programming language:

Sum=3 + 2;

Tokenized and represented by the following table:

Lexeme	Token category
Sum	"identifier"
=	"assignment operator"
3	"integer literal"
+	"addition operator"
2	"integer literal"
;	"end of the statement"

ALGORITHM:

1. Start the program
2. Include the header files.
3. Allocate memory for the variable by dynamic memory allocation function.
4. Use the file accessing functions to read the file.
5. Get the input file from the user.
6. Separate all the file contents as tokens and match it with the functions.
7. Define all the keywords in a separate file and name it as key.c
8. Define all the operators in a separate file and name it as open.c
9. Give the input program in a file and name it as input.c
10. Finally print the output after recognizing all the tokens.
11. Stop the program.

PROGRAM: (DEVELOP A LEXICAL ANALYZER TO RECOGNIZE A FEW PATTERNS IN C)

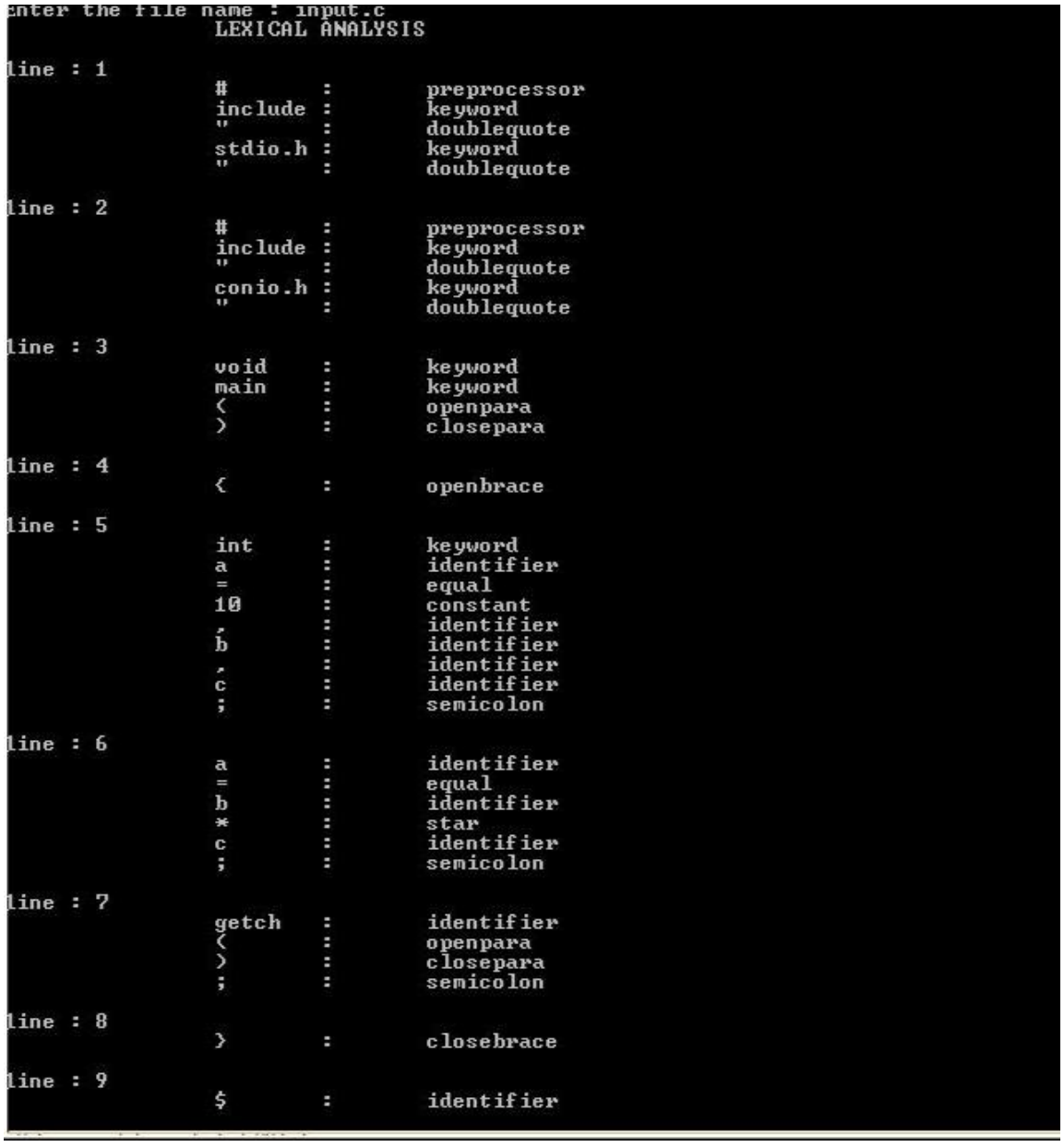
```
#include<stdio.h>
#include<conio.h>
#include<ctype.h>
#include<string.h>
 void main()
{
FILE *fi,*fo,*fop,*fk;
int flag=0,i=1;
char c,t,a[15],ch[15],file[20];
clrscr();
printf("\n Enter the File Name:");
 scanf("%s",&file);
fi=fopen(file,"r");
```

```
fo=fopen("inter.c","w");

fop=fopen("oper.c","r");
fk=fopen("key.c","r");
c=getc(fi);
while(!feof(fi))
{
if(isalpha(c)||isdigit(c)||(c=='['||c==']'||c=='.'==1))
 fputc(c,fo);
else
{
if(c=='\n')
fprintf(fo,"\t$\t");
 else fprintf(fo,"\t%c\t",c);
}
c=getc(fi);
}
fclose(fi);
 fclose(fo);
fi=fopen("inter.c","r");
printf("\n Lexical Analysis");
fscanf(fi,"%s",a);
printf("\n Line: %d\n",i++);
while(!feof(fi))
{
if(strcmp(a,"$")==0)
{
printf("\n Line: %d \n",i++);
fscanf(fi,"%s",a);
}
fscanf(fop,"%s",ch);
 while(!feof(fop))
{
if(strcmp(ch,a)==0)
{
fscanf(fop,"%s",ch);
printf("\t\t%s\t:\t%s\n",a,ch);
flag=1;
```

```
} fscanf(fop,"%s",ch);
}
rewind(fop);
 fscanf(fk,"%s",ch);
while(!feof(fk))
{
if(strcmp(ch,a)==0)
{
fscanf(fk,"%k",ch);
printf("\t\t%s\t:\tKeyword\n",a);
flag=1;
}
fscanf(fk,"%s",ch);
}
rewind(fk);
 if(flag==0)
{
 if(isdigit(a[0]))
 printf("\t\t%s\t:\tConstant\n",a);
 else
printf("\t\t%s\t:\tIdentifier\n",a);
}
flag=0;
fscanf(fi,"%s",a);
} getch();
}
Key.C:
int
void
main
 char
if
for
while
else
 printf
scanf
 FILE
```

```
Include
 stdio.h
 conio.h
iostream.h
Oper.C:
( open para
) closepara
{ openbrace
} closebrace
< lesser
> greater
" doublequote '
singlequote : colon
; semicolon
# preprocessor
= equal
== asign
% percentage
^ bitwise
& reference
* star
+ add
- sub
\ backslash
/ slash

Input.C:
#include "stdio.h"
#include "conio.h"
void main()
{
int a=10,b,c;
 a=b*c;
getch();
}
```

OUTPUT:



RESULT:

Thus the above program for developing the lexical the lexical analyzer and recognizing the few pattern s in C is executed successfully and the output is verified.

EX.NO:3
DATE:

IMPLEMENTATION OF LEXICAL ANALYZER USING LEX TOOL

AIM:

To write a program to implement the Lexical Analyzer using lex tool.

INTRODUCTION:

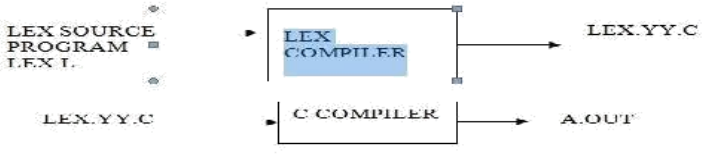
THEORY:

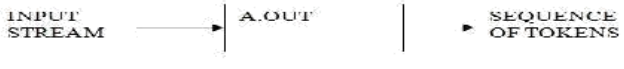
- A language for specifying lexical analyzer.
- There is a wide range of tools for construction of lexical analyzer. The majority of these tools are based on regular expressions.
- The one of the traditional tools of that kind is lex.

LEX:

- The lex is used in the manner depicted. A specification of the lexical analyzer is preferred by creating a program lex.1 in the lex language.
- Then lex.1 is run through the lex compiler to produce a 'c' program lex.yy.c.
- The program lex.yy.c consists of a tabular representation of a transition diagram constructed from the regular expression of lex.1 together with a standard routine that uses table of recognize leximes.
- Lex.yy.c is run through the 'C' compiler to produce as object program a.out, which is the lexical analyzer that transform as input stream into sequence of tokens.

LEX SOURCE:





ALGORITHM:

1. Start the program
2. Lex program consists of three parts.
3. Declaration %%
4. Translation rules %%
5. Auxiliary procedure.
6. The declaration section includes declaration of variables, main test, constants and regular
7. Definitions.
8. Translation rule of lex program are statements of the form
9. P1{action}
10. P2{action}
11. …..
12. …..
13. Pn{action}
14. Write program in the vi editor and save it with .1 extension.
15. Compile the lex program with lex compiler to produce output file as lex.yy.c.
16. Eg. $ lex filename.1
17. $gcc lex.yy.c-11
18. Compile that file with C compiler and verify the output.

PROGRAM: (LEXICAL ANALYZER USING LEX TOOL)

```
#include<stdio.h>
#include<ctype.h>
#include<conio.h>
#include<string.h>
char vars[100][100];
int vcnt;
char input[1000],c;
char token[50],tlen;
int state=0,pos=0,i=0,id;
char *getAddress(char str[])
{
for(i=0;i<vcnt;i++)
if(strcmp(str,vars[i])==0)
```

```
return vars[i];
strcpy(vars[vcnt],str);
return vars[vcnt++];
}
int isrelop(char c)
{
if(c=='+'||c=='-'||c=='*'||c=='/'||c=='%'||c=='^')
return 1;
else
return 0;
}
int main(void)
{
clrscr();
printf("Enter the Input String:");
gets(input);
do
{
c=input[pos];
putchar(c);
switch(state)
{
case 0:
if(isspace(c))
printf("\b");
if(isalpha(c))
{
token[0]=c;
tlen=1;
state=1;
}
if(isdigit(c))
state=2;
if(isrelop(c))
state=3;
if(c==';')
printf("\t<3,3>\n");
if(c=='=')
```

```
printf("\t<4,4>\n");
break;
case 1:
if(!isalnum(c))
{
token[tlen]='\o';
printf("\b\t<1,%p>\n",getAddress(token));
state=0;
pos--;
}
else
token[tlen++]=c;
break;
case 2:
if(!isdigit(c))
{
printf("\b\t<2,%p>\n",&input[pos]);
state=0;
pos--;
}
break;
case 3:
id=input[pos-1];
if(c=='=')
printf("\t<%d,%d>\n",id*10,id*10);
else{
printf("\b\t<%d,%d>\n",id,id);
pos--;
}state=0;
break;
}
pos++;
}
while(c!=0);
getch();
return 0;
}
```

OUTPUT

RESULT:

Thus the program for the exercise on lexical analysis using lex has been successfully executed and output is verified.

EX.NO:-3

PROGRAM TO RECOGNISE A VALID VARIABLE WHICH STARTS WITH A LETTER FOLLOWED BY ANY NUMBER OF LETTERS OR DIGITS

PROGRAM :

```
#include<stdio.h>

 #include<conio.h>
#include<ctype.h>

void main()

{

char a[10];  int flag, i=1;

clrscr();

printf("\n Enter a variabler:");

gets(a);

if(isalpha(a[0]))

flag=1;

else

printf("\n Not a valid variable");

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

{

if(!isdigit(a[i])&&!isalpha(a[i]))

{

flag=0;

break;

}

i++;

}

if(flag==1)
printf("\n Valid variable");
getch();
}
```

exp:-4

4.1 OBJECTIVE:

Write a C program for implementing the functionalities of predictive parser

4.2 RESOURCE:

Turbo C

4.3 PROGRAM LOGIC:

Read the input string.

By using the FIRST AND FOLLOW values.

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

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

Constructing the predictive parser table

4.4 PROCEDURE:

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

4.5 PROGRAM:

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
char prol[7][10]={"S","A","A","B","B","C","C"};
char pror[7][10]={"A","Bb","Cd","aB","@","Cc","@"};
char prod[7][10]={"S->A","A->Bb","A->Cd","B->aB","B->@","C->Cc","C-
>@"}; char first[7][10]={"abcd","ab","cd","a@","@","c@","@"};
 char follow[7][10]={"$","$","$","a$","b$","c$","d$"};
char table[5][6][10];
numr(char c)
 {
        switch(c)
        {
                case 'S': return 0;
                case 'A': return 1;
                case 'B': return 2;
                case 'C': return 3;
                case 'a': return 0;
                case 'b': return 1;
                case 'c': return 2;
                case 'd': return 3;
                case '$': return 4;
        }
```

```
        return(2);
}
void main()
{
        int i,j,k;
        clrscr();
        for(i=0;i<5;i++)
        for(j=0;j<6;j++)
                strcpy(table[i][j]," ");
        printf("\nThe following is the predictive parsing table for the following
        grammar:\n"); for(i=0;i<7;i++)
                printf("%s\n",prod[i]);
        printf("\nPredictive parsing table is\n");
        fflush(stdin);
        for(i=0;i<7;i++)
        {
                k=strlen(first[i]);
                for(j=0;j<10;j++)
                if(first[i][j]!='@')
                strcpy(table[numr(prol[i][0])+1][numr(first[i][j])+1],prod[i]);
        }
        for(i=0;i<7;i++)
        {
                if(strlen(pror[i])==1)
                {
                if(pror[i][0]=='@')
                {
                        k=strlen(follow[i]);
                        for(j=0;j<k;j++)
                        strcpy(table[numr(prol[i][0])+1][numr(follow[i][j])+1],prod[i]);
```

```
                        }
                }
        }
        strcpy(table[0][0]," ");
        strcpy(table[0][1],"a");
        strcpy(table[0][2],"b");
        strcpy(table[0][3],"c");
        strcpy(table[0][4],"d");
        strcpy(table[0][5],"$");
        strcpy(table[1][0],"S");
        strcpy(table[2][0],"A");
        strcpy(table[3][0],"B");
        strcpy(table[4][0],"C");
        printf("\n---------------------------------------------------------\n");
        for(i=0;i<5;i++)
         for(j=0;j<6;j++)
                {
                printf("%-10s",table[i][j]);
                if(j==5)
                printf("\n---------------------------------------------------------\n");
                }
        getch();
       }
```

4.6 PRE LAB QUESTIONS:

1. What is top-down parsing?
2. What are the disadvantages of brute force method?
3. What is context free grammar?
4. What is parse tree?
5. What is ambiguous grammar?
6. What are the derivation methods to generate a string for the given grammar?
7. What is the output of parse tree?

POST LAB QUESTIONS

1. What is Predictive parser?
2. How many types of analysis can we do using Parser?
3. What is Recursive Decent Parser?
4. How many types of Parsers are there?
5. What is LR Parser?

4.9 INPUT & OUTPUT:

The following is the predictive parsing table for the following grammar:

S->A
A->Bb
A->Cd
B->aB
B->@
C->Cc
C->@

Predictive parsing table is

---

a b c d $

---

1. S->AS->AS->AS->A

A A->Bb A->BbA->Cd A->Cd

B B->aB B->@ B->@ B->@

---

1. C->@C->@ C->@

EXPERIMENT-5

5.1 OBJECTIVE:

Write a program to Design LALR Bottom up Parser.

5.2 RESOURCE:

TURBO C++

5.3 PROGRAM LOGIC:

Read the input string.

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

We perform shift and reduce actions to parse the grammar.

Parsing is completed when we reach $ symbol.

5.4 PROCEDURE:

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

5.5 PROGRAM:

```
/*LALR PARSER
  E->E+T
          E->T
          T->T*F
          T->F
           F->(E)
          F->i
*/
```

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
#include<string.h>
void push(char *,int *,char);
char stacktop(char *);
void isproduct(char,char);
int ister(char);
int isnter(char);
int isstate(char);

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

```
const struct action A[12]={
                                {"sf","emp","emp","se","emp","emp"},
                                {"emp","sg","emp","emp","emp","acc"},
                                {"emp","rc","sh","emp","rc","rc"},
                                {"emp","re","re","emp","re","re"},
                                {"sf","emp","emp","se","emp","emp"},
                                {"emp","rg","rg","emp","rg","rg"},
                                {"sf","emp","emp","se","emp","emp"},
                                {"sf","emp","emp","se","emp","emp"},
                                {"emp","sg","emp","emp","sl","emp"},
                                {"emp","rb","sh","emp","rb","rb"},
                                {"emp","rb","rd","emp","rd","rd"},
                                {"emp","rf","rf","emp","rf","rf"}
                        };
struct gotol
{
  char r[3][4];
};
const struct gotol G[12]={
                                {"b","c","d"},
                                {"emp","emp","emp"},
                                {"emp","emp","emp"},
                                {"emp","emp","emp"},
                                {"i","c","d"},
                                {"emp","emp","emp"},
                                {"emp","j","d"},
                                {"emp","emp","k"},
                                {"emp","emp","emp"},
                                {"emp","emp","emp"},
                          };
char ter[6]={'i','+','*',')','(','$'};
char nter[3]={'E','T','F'};
char states[12]={'a','b','c','d','e','f','g','h','m','j','k','l'};
char stack[100];
int top=-1;
char temp[10];
struct grammar
{
```

```
 char left;
 char right[5];
};
const struct grammar rl[6]={
                                {'E',"e+T"},
                                {'E',"T"},
                                {'T',"T*F"},
                                {'T',"F"},
                                {'F',"(E)"},
                                {'F',"i"},
                              };
void main()
{
        char inp[80],x,p,dl[80],y,bl='a';
        int i=0,j,k,l,n,m,c,len;
        clrscr();
        printf(" Enter the input :");
        scanf("%s",inp);
        len=strlen(inp);
        inp[len]='$';
        inp[len+1]='\0';
        push(stack,&top,bl);
        printf("\n stack \t\t\t input");
        printt(stack,&top,inp,i);
        do
        {
        x=inp[i];
        p=stacktop(stack);

        isproduct(x,p);
        if(strcmp(temp,"emp")==0)
                error();
        if(strcmp(temp,"acc")==0)
                break;
        else
        {
                if(temp[0]=='s')
                {
                        push(stack,&top,inp[i]);
                        push(stack,&top,temp[1]);
                        i++;
```

```
                }
                else
                {
                        if(temp[0]=='r')
                        {
                                j=isstate(temp[1]);
                                strcpy(temp,rl[j-2].right);
                                dl[0]=rl[j-2].left;
                                dl[1]='\0';
                                n=strlen(temp);
                                for(k=0;k<2*n;k++)
                                        pop(stack,&top);
                                for(m=0;dl[m]!='\0';m++)
                                        push(stack,&top,dl[m]);
                                l=top;
                                y=stack[l-1];
                                isreduce(y,dl[0]);
                                for(m=0;temp[m]!='\0';m++)
                                        push(stack,&top,temp[m]);
                        }
                }
        }
        printt(stack,&top,inp,i);
 }while(inp[i]!='\0');
 if(strcmp(temp,"acc")==0)
        printf(" \n accept the input ");
 else
        printf(" \n do not accept the input ");
 getch();
}
void push(char *s,int *sp,char item)
{
 if(*sp==100)
        printf(" stack is full ");
 else
 {
        *sp=*sp+1;
```

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

```
                return i+1;
        return 0;
}
void error()
{
        printf(" error in the input ");
        exit(0);
}
void isreduce(char x,char p)
{
        int k,l;
        k=isstate(x);
        l=isnter(p);
        strcpy(temp,G[k-1].r[l-1]);
}


char pop(char *s,int *sp)
{
        char item;
        if(*sp==-1)
                printf(" stack is empty ");
        else
        {
                item=s[*sp];
                *sp=*sp-1;
        }
        return item;
}
void printt(char *t,int *p,char inp[],int i)
{
        int r;
        printf("\n");
        for(r=0;r<=*p;r++)
                rep(t,r);
        printf("\t\t\t");
        for(r=i;inp[r]!='\0';r++)
```

```
                printf("%c",inp[r]);
}
void rep(char t[],int r)
{
        char c;
        c=t[r];
        switch(c)
        {
                case 'a':  printf("0");
                        break;
                case 'b':  printf("1");
                        break;
                case 'c':  printf("2");
                        break;
                case 'd':  printf("3");
                        break;
                case 'e':  printf("4");
                        break;
                case 'f':  printf("5");
                        break;
                case 'g':  printf("6");
                        break;
                case 'h':  printf("7");
                        break;
                case 'm': printf("8");
                        break;
                case 'j':  printf("9");
                        break;
                case 'k':  printf("10");
                        break;
                case 'l':  printf("11");
                        break;
                default   :printf("%c",t[r]);
                        break;
        }
}
```

5.6 PRE-LAB QUESTIONS

== Why bottom-up parsing is also called as shift reduce parsing?
== What are the different types of bottom up parsers?
== What is mean by LR (0) items?
== Write the general form of LR(1) item?
== What is YACC?

5.7 LAB ASSIGNMENT

\{ Write a program to compute FOLLOW for the following grammar? E → TE'
E' → +TE'/î T → FT'
T' → *FT'/î F → (E)/i

\{ Write a program to construct LALR parsing table for the following grammar. S → iCtSS'
S' → eS/ î

5.8 POST-LAB QUESTIONS:

\{ What is LALR parsing?
\{ What is Shift reduced parser?
\{ What are the operations of Parser?
\{ What is the use of parsing table?
\{ What is bottom up parsing?

5.9 INPUT & OUTPUT:

Enter the input: i*i+1

Output

Stack	**input**
0	i*i+i$
0i5	*i+i$
0F3	*i+i$
0T2	*i+i$
0T2*7	i+i$
0T2*7i5	+i$
0T2*7i5F10	+i$
0T2	+i$
0E1	+i$
0E1+6	i$
0E1+6i5	$
0E1+6F3	$
0E1+6T9	$
0E1	$

accept the input*/

EX. NO: 6

DATE:

IMPLEMENTATION OF SYMBOL TABLE

AIM:

To write a C program to implement a symbol table.

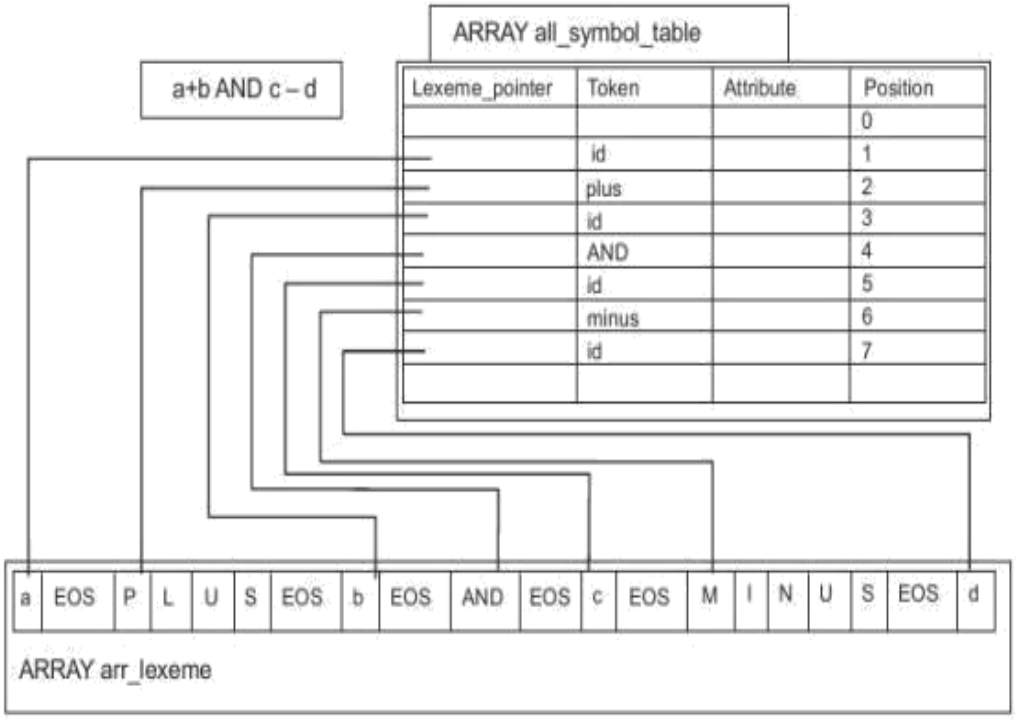
INTRODUCTION:

A Symbol table is a data structure used by a language translator such as a compiler or interpreter, where each identifier in a program's source code is associated with information relating to its declaration or appearance in the source

Possible entries in a symbol table:

- Name : a string
- Attribute:
    1. Reserved word
    2. Variable name
    3. Type Name
    4. Procedure name
    5. Constant name
- Data type
- Scope information: where it can be used.
- Storage allocation

SYMBOL TABLE



ALGORITHM:

1. Start the Program.
2. Get the input from the user with the terminating symbol '$'.
3. Allocate memory for the variable by dynamic memory allocation function.
4. If the next character of the symbol is an operator then only the memory is allocated.
5. While reading , the input symbol is inserted into symbol table along with its memory address.
6. The steps are repeated till"$"is reached.
7. To reach a variable, enter the variable to the searched and symbol table has been checked for corresponding variable, the variable along its address is displayed as result.
8. Stop the program.

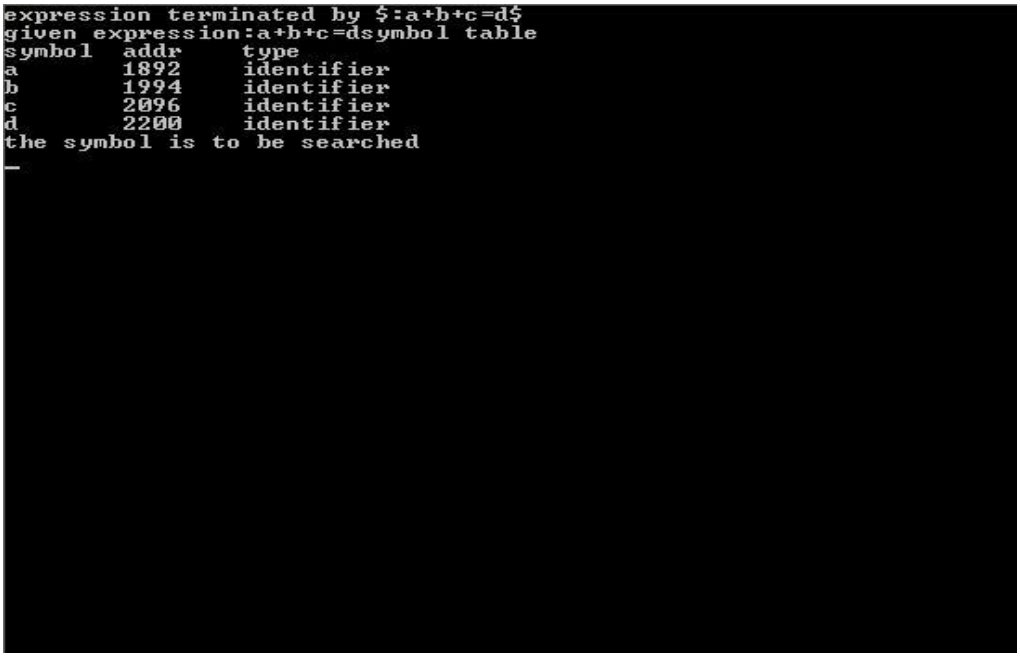
PROGRAM: (IMPLEMENTATION OF SYMBOL TABLE)

```
#include<stdio.h>
#include<conio.h>
#include<malloc.h>
#include<string.h>
#include<math.h>
#include<ctype.h>
void main()
{
int i=0,j=0,x=0,n,flag=0; void
*p,*add[15]; char ch,srch,b[15],d[15],c;
//clrscr();
printf("expression terminated by
$:"); while((c=getchar())!='$') {

b[i]=c; i++;
}
n=i-1;
printf("given expression:");
i=0;
```

```
while(i<=n)
{
printf("%c",b[i]); i++;
}
printf("symbol table\n");
 printf("symbol\taddr\ttype\n");
while(j<=n)
{
c=b[j]; if(isalpha(toascii(c)))
{
if(j==n)
{
p=malloc(c); add[x]=p;
d[x]=c;
printf("%c\t%d\tidentifier\n",c,p);
}
else
{
ch=b[j+1];
if(ch=='+'||ch=='-'||ch=='*'||ch=='=')
{
p=malloc(c);
add[x]=p;
d[x]=c;
 printf("%c\t%d\tidentifier\n",c,p);
 x++;
}
}
} j++;
}
```

```
printf("the symbol is to be searched\n");
srch=getch();
for(i=0;i<=x;i++)
{
if(srch==d[i])
{
printf("symbol found\n");
printf("%c%s%d\n",srch,"@address",add[i]);
flag=1;
}
}
if(flag==0)
printf("symbol not found\n");
//getch();
```

OUTPUT:



RESULT:

Thus the C program to implement the symbol table was executed and the output is verified.

EX.NO:7
DATE:

IMPLEMENTATION OF TYPE CHECKING

AIM:

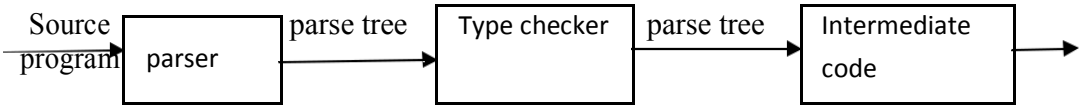
To write a C program for implementing type checking for given expression.

INTRODUCTION:

The type analysis and type checking is an important activity done in the semantic analysis phase. The need for type checking is

6. To detect the errors arising in the expression due to incompatible operand.
7. To generate intermediate code for expressions due to incompatible operand

Role of type checker



ALGORITHM:

2. Start a program.
3. Include all the header files.
4. Initialize all the functions and variables.
5. Get the expression from the user and separate into the tokens.
6. After separation, specify the identifiers, operators and number.
7. Print the output.
8. Stop the program.

PROGRAM: (TYPE CHECKING)

```
#include<stdio.h>
char str[50],opstr[75];
int f[2][9]={2,3,4,4,4,0,6,6,0,1,1,3,3,5,5,0,5,0};
int col,col1,col2;
char c;
swt()
{
   switch(c)
   {
```

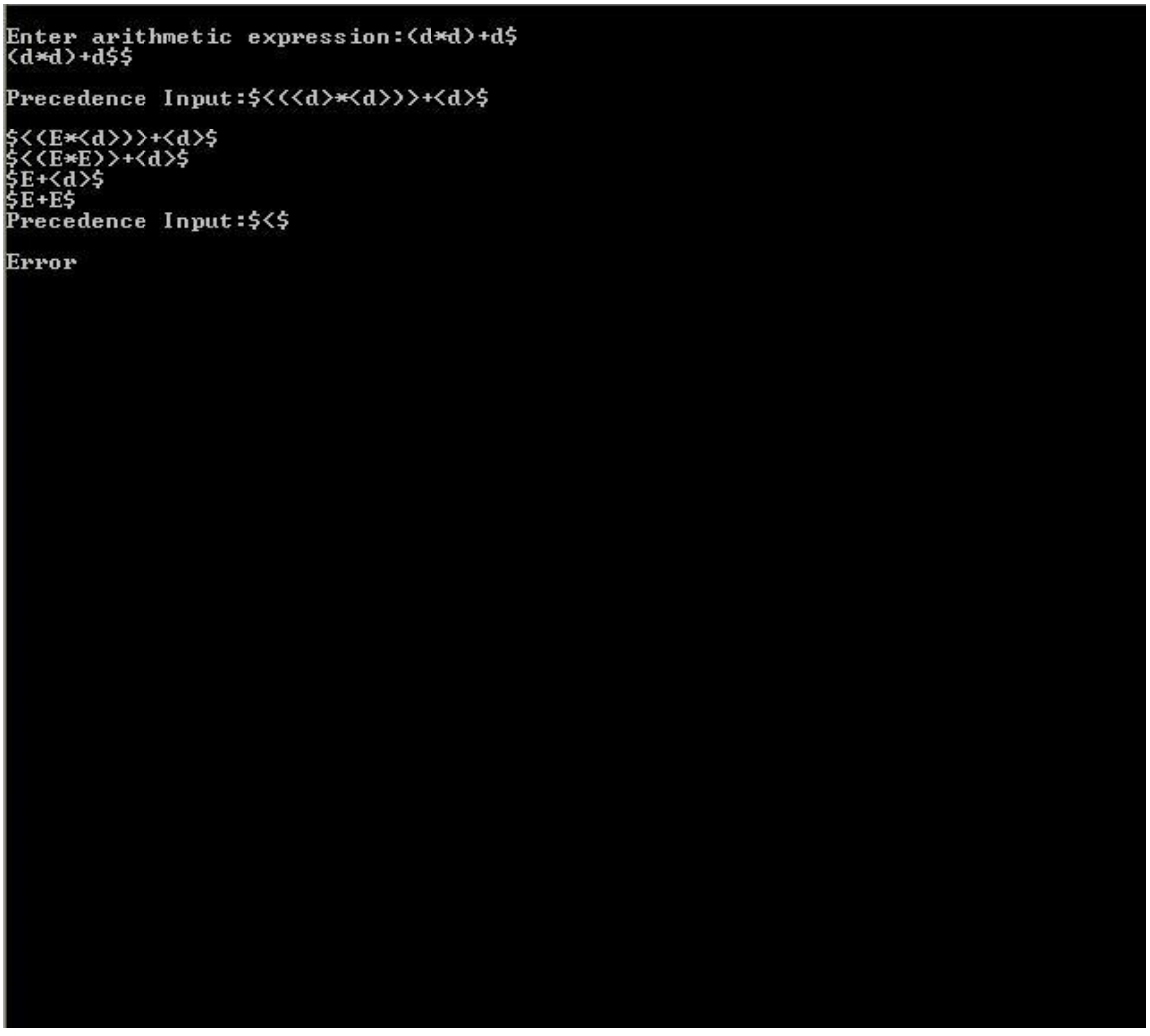
```
      case'+':col=0;break;
      case'-':col=1;break;
      case'*':col=2;break;
      case'/':col=3;break;
      case'^':col=4;break;
      case'(':col=5;break;
      case')':col=6;break;
      case'd':col=7;break;
      case'$':col=8;break;
      default:printf("\nTERMINAL MISSMATCH\n");
       exit(1);
   }
  // return 0;
}
main()
{
   int i=0,j=0,col1,cn,k=0;
   int t1=0,foundg=0;
   char temp[20];
   clrscr();
   printf("\nEnter arithmetic expression:");
   scanf("%s",&str);
   while(str[i]!='\0')
     i++;
   str[i]='$';
   str[++i]='\0';
   printf("%s\n",str);
   come:
   i=0;
   opstr[0]='$';
   j=1;
   c='$';
   swt();
   col1=col;
   c=str[i];
   swt();
   col2=col;
```

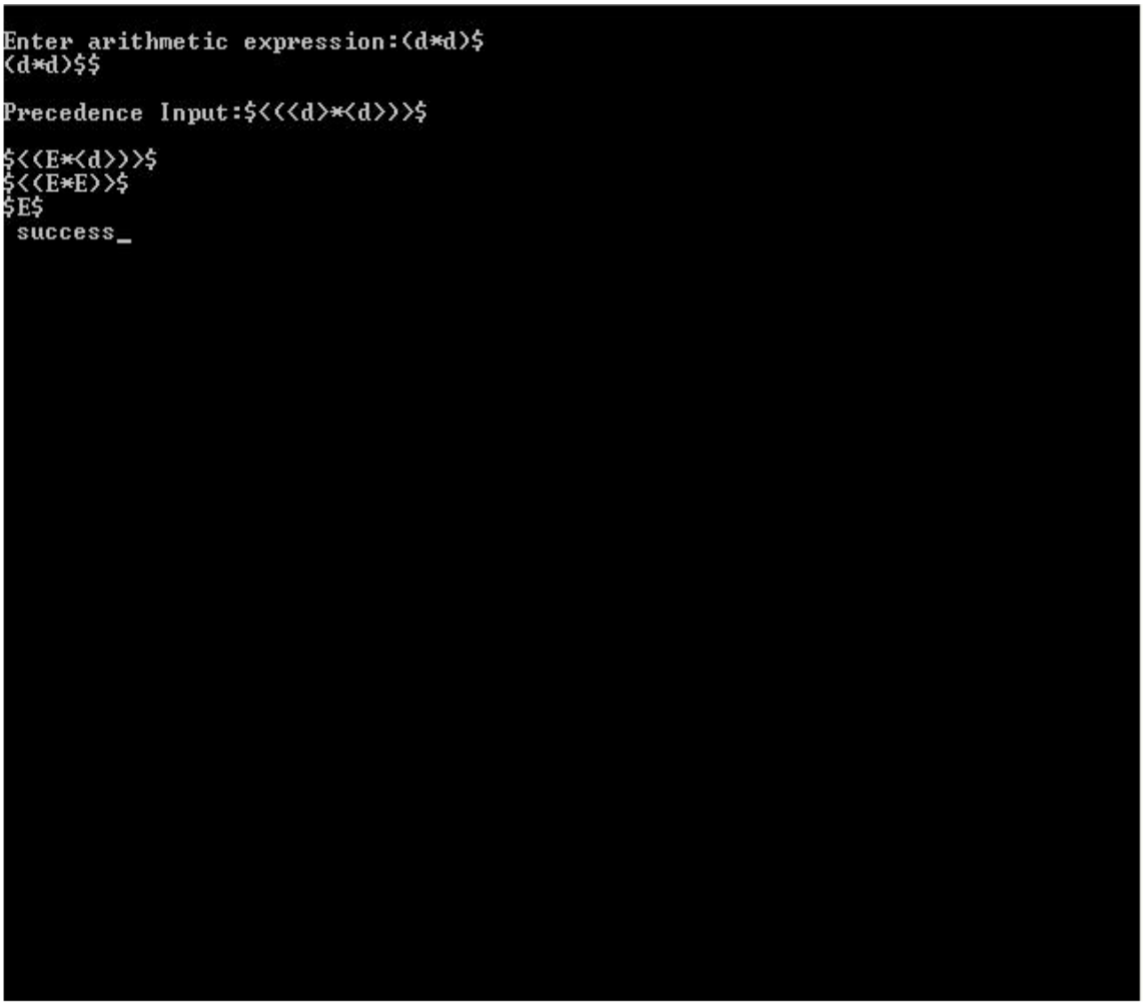
```
  if(f[1][col1]>f[2][col2])
  {
    opstr[j]='>';
    j++;
  }
else if(f[1][col1]<f[2][col2])
{
     opstr[j]='<';
     j++;
 }
 else
 {
     opstr[j]='=';j++;
 }

 while(str[i]!='$')
 {
     c=str[i];
        swt();
      col1=col;
      c=str[++i];
      swt();
      col2=col;
      opstr[j]=str[--i];
      j++;
      if(f[0][col1]>f[1][col2])
      {
         opstr[j]='>';
         j++;
      }
      else if(f[0][col1]<f[1][col2])
      {
        opstr[j]='<';
         j++;
      }
      else
   {
      opstr[j]='=';j++;
```

```
        }
        i++;
    }
    opstr[j]='$';
    opstr[++j]='\0';
    printf("\nPrecedence Input:%s\n",opstr);
    i=0;
    j=0;
    while(opstr[i]!='\0')
    {
        foundg=0;
         while(foundg!=1)
         {
           if(opstr[i]=='\0')goto redone;
           if(opstr[i]=='>')foundg=1;
           t1=i;
           i++;
        }
         if(foundg==1)
         for(i=t1;i>0;i--)
          if(opstr[i]=='<')break;
          if(i==0){printf("\nERROR\n");exit(1);}
         cn=i;
          j=0;
          i=t1+1;
         while(opstr[i]!='\0')
         {
             temp[j]=opstr[i];
            j++;i++;
           }
          temp[j]='\0';
          opstr[cn]='E';
          opstr[++cn]='\0';
          strcat(opstr,temp);
          printf("\n%s",opstr);
           i=1;
       }
         redone:k=0;
```

```
while(opstr[k]!='\0')
     {
        k++;
        if(opstr[k]=='<')
       {
  Printf("\nError");
  exit(1);
        }
  }
  if((opstr[0]=='$')&&(opstr[2]=='$'))goto sue;
  i=1
  while(opstr[i]!='\0')
  {
     c=opstr[i];
     if(c=='+'||c=='*'||c=='/'||c=='$')
     {
 temp[j]=c;j++;}
 i++;
    }
     temp[j]='\0';
     strcpy(str,temp);
     goto come;
     sue:
     printf("\n success");
     return 0;
  }
```

OUTPUT:


Enter arithmetic expression:(d*d)+d$
(d*d)+d$$
Precedence Input:$<<(d)*(d))>+(d)$
$<(E*(d))>+(d)$
$<(E*E)>+(d)$
$E+(d)$
$E+E$
Precedence Input:$<$
Error



RESULT:

Thus the program has been executed successfully and Output is verified.

EX.NO:08
DATE:

IMPLEMENT ANY ONE STORAGE ALLOCATION STRATEGIES (HEAP,STACK,STATIC)

AIM:

To write a C program for Stack to use dynamic storage allocation.

INTRODUCTION:

Storage Allocation

Runtime environment manages runtime memory requirements for the following entities:

- Code: It is known as the part of a program that does not change at runtime. Its memory requirements are at the compile time
- Procedures: Their text part is static but they are called in a random manner. That is why, stack storage is used to manage procedure calls and activations.
- Variables: Variables are known at the runtime only, unless they are global or constant. Heap memory allocation scheme is used for managing allocation and de-allocation of memory for variables in runtime.

ALGORITHM:

1. Start the program
2. Enter the expression for which intermediate code is to be generated
3. If the length of the string is greater than 3, than call the procedure to return the precedence
4. Among the operands.
5. Assign the operand to exp array and operators to the array.
6. Create the three address code using quadruples structure.
7. Reduce the no of temporary variables.
8. Continue this process until we get an output.
9. Stop the program.

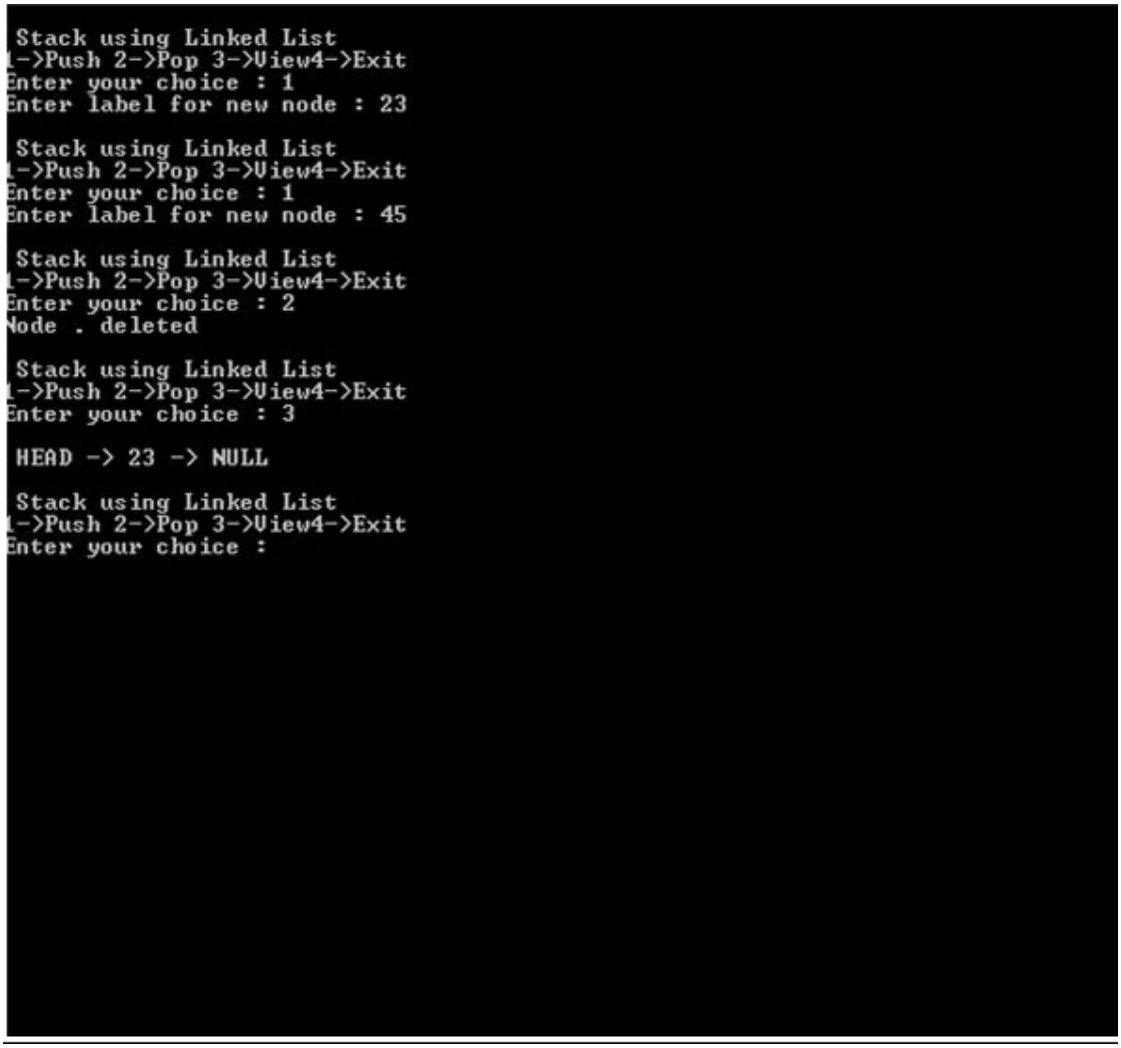
PROGRAM: (STACK TO USE DYNAMIC STORAGE ALLOCATION)

```
#include <stdio.h>

#include <conio.h>

#include <process.h>

#include <alloc.h>

struct node

{

int label;
```

```
struct node *next;
};
void main()
{
int ch = 0;
int k;
struct node *h, *temp, *head;
head = (struct node*) malloc(sizeof(struct node));
head->next = NULL;
while(1)
{
printf("\n Stack using Linked List \n");
printf("1->Push ");
printf("2->Pop ");
printf("3->View");
printf("4->Exit \n");
printf("Enter your choice : ");
scanf("%d", &ch);
switch(ch)
{
case 1:
temp=(struct node *)(malloc(sizeof(struct
node))); printf("Enter label for new node : ");
scanf("%d", &temp->label); h = head;

temp->next = h->next;
h->next = temp;
break;
case 2:
```

```
h = head->next;
head->next = h->next;
printf("Node %s deleted\n", h->label);
free(h);
break;
case 3:
printf("\n HEAD -> ");
h = head;
while(h->next != NULL)
{
h = h->next;
printf("%d -> ",h->label);
}
printf("NULL \n");
break;
case 4:
exit(0);
}
}}
```

OUTPUT:



RESULT:

Thus the program for implement storage allocation to use dynamic process for stack has been successfully executed

EX.NO:09
DATE:

CONSTRUCTION OF DAG

AIM:

To write a C program to construct of DAG(**Directed Acyclic Graph**)

INTRODUCTION:

The code optimization is required to produce an efficient target code. These are two important issues that used to be considered while applying the techniques for code optimization.

They are:

- The semantics equivalences of the source program must not be changed.
- The improvement over the program efficiency must be achieved without changing the algorithm.

ALGORITHM:

1. Start the program
2. Include all the header files
3. Check for postfix expression and construct the in order DAG representation
4. Print the output
5. Stop the program

PROGRAM: (TO CONSTRUCT OF DAG(DIRECTED ACYCLIC GRAPH))

```
#include<stdio.h>
main()
{
struct da
{
int ptr,left,right;
char label;
}dag[25];
int ptr,l,j,change,n=0,i=0,state=1,x,y,k;
char store,*input1,input[25],var;
clrscr();
for(i=0;i<25;i++)
{
```

```
dag[i].ptr=NULL;

dag[i].left=NULL;

dag[i].right=NULL;

dag[i].label=NULL;

}

printf("\n\nENTER THE

EXPRESSION\n\n"); scanf("%s",input1);

/*EX:((a*b-c))+((b-c)*d)) like this give with
paranthesis.limit is 25 char ucan change that*/

for(i=0;i<25;i++)

input[i]=NULL;

l=strlen(input1);

a:

for(i=0;input1[i]!=')';i++);

for(j=i;input1[j]!='(';j--);

for(x=j+1;x<i;x++)

if(isalpha(input1[x]))

input[n++]=input1[x];

else

if(input1[x]!='0')

store=input1[x];

input[n++]=store;

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

input1[x]='0';

if(input1[0]!='0')goto a;

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

{

dag[i].label=input[i];

dag[i].ptr=i;

if(!isalpha(input[i])&&!isdigit(input[i]))

{

dag[i].right=i-1;

ptr=i;
```

```
var=input[i-1];
if(isalpha(var))
ptr=ptr-2;
else
{
ptr=i-1;
b:
if(!isalpha(var)&&!isdigit(var))
{
ptr=dag[ptr].left;
var=input[ptr];
goto b;
}
else
ptr=ptr-1;
}
dag[i].left=ptr;
}
}
printf("\n SYNTAX TREE FOR GIVEN EXPRESSION\n\n");
printf("\n\n PTR \t\t LEFT PTR \t\t RIGHT PTR \t\t
LABEL \n\n");
for(i=0;i<n;i++)/* draw the syntax tree for the
following output with pointer value*/
printf("\n
%d\t%d\t%d\t%c\n",dag[i].ptr,dag[i].left,dag[i].right,dag[i].la
bel);
getch();
for(i=0;i<n;i++)
{
for(j=0;j<n;j++)
{
if((dag[i].label==dag[j].label&&dag[i].left==dag[j].left)&&dag[ i].right==dag[j].right)

{
```

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

{

if(dag[k].left==dag[j].ptr)dag[k].left=dag[i].ptr;

if(dag[k].right==dag[j].ptr)dag[k].right=dag[i].ptr;

}

dag[j].ptr=dag[i].ptr;

}

}

}

printf("\n DAG FOR GIVEN EXPRESSION\n\n");

 printf("\n\n PTR \t LEFT PTR \t RIGHT PTR \t LABEL \n\n");

for(i=0;i<n;i++)/*draw DAG for the following output
with pointer value*/

printf("\n %d
\t\t%d\t\t%d\t\t%c\n",dag[i].ptr,dag[i].left,dag[i].right,dag[i
].label);

getch();

}
```

OUTPUT:

```
ENTER THE EXPRESSION
((a*(b-c))+((b-c)*d))
 SYNTAX TREE FOR GIVEN EXPRESSION

 PTR             LEFT PTR              RIGHT PTR              LABEL

 0       0       0       b
 1       0       0       c
 2       0       1       -
 3       0       0       a
 4       2       3       *
 5       0       0       b
 6       0       0       c
 7       5       6       -
 8       0       0       d
 9       7       8       *
 10      4       9       +
```

RESULT:

Thus the program for implementation of DAG has been successfully executed and output is verified.

EX.NO.10
DATE:

IMPLEMENT THE BACK END OF THE COMPILER

AIM:

To implement the back end of the compiler which takes the three address code and produces the 8086 assembly language instructions that can be assembled and run using a 8086 assembler. The target assembly instructions can be simple move, add, sub, jump. Also simple addressing modes are used.

INTRODUCTION:

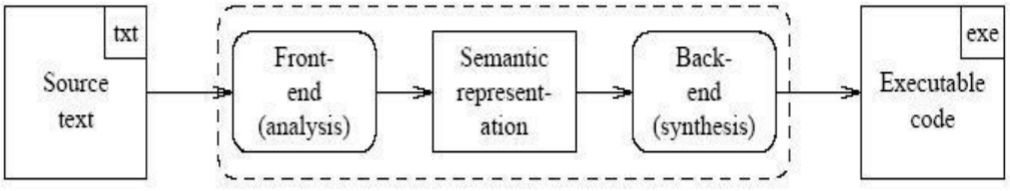
A compiler is a computer program that implements a programming language specification to "translate" programs, usually as a set of files which constitute the source code written in source language, into their equivalent machine readable instructions(the target language, often having a binary form known as object code). This translation process is called compilation.

BACK END:

- Some local optimization
- Register allocation
- Peep-hole optimization
- Code generation
- Instruction scheduling

The main phases of the back end include the following:

- Analysis: This is the gathering of program information from the intermediate representation derived from the input; data-flow analysis is used to build use-define chains, together with dependence analysis, alias analysis, pointer analysis, escape analysis etc.
- Optimization: The intermediate language representation is transformed into functionally equivalent but faster (or smaller) forms. Popular optimizations are expansion, dead, constant, propagation, loop transformation, register allocation and even automatic parallelization.
- Code generation: The transformed language is translated into the output language, usually the native machine language of the system. This involves resource and storage decisions, such as deciding which variables to fit into registers and memory and the selection and scheduling of appropriate machine instructions along with their associated modes. Debug data may also need to be generated to facilitate debugging.



ALGORITHM:

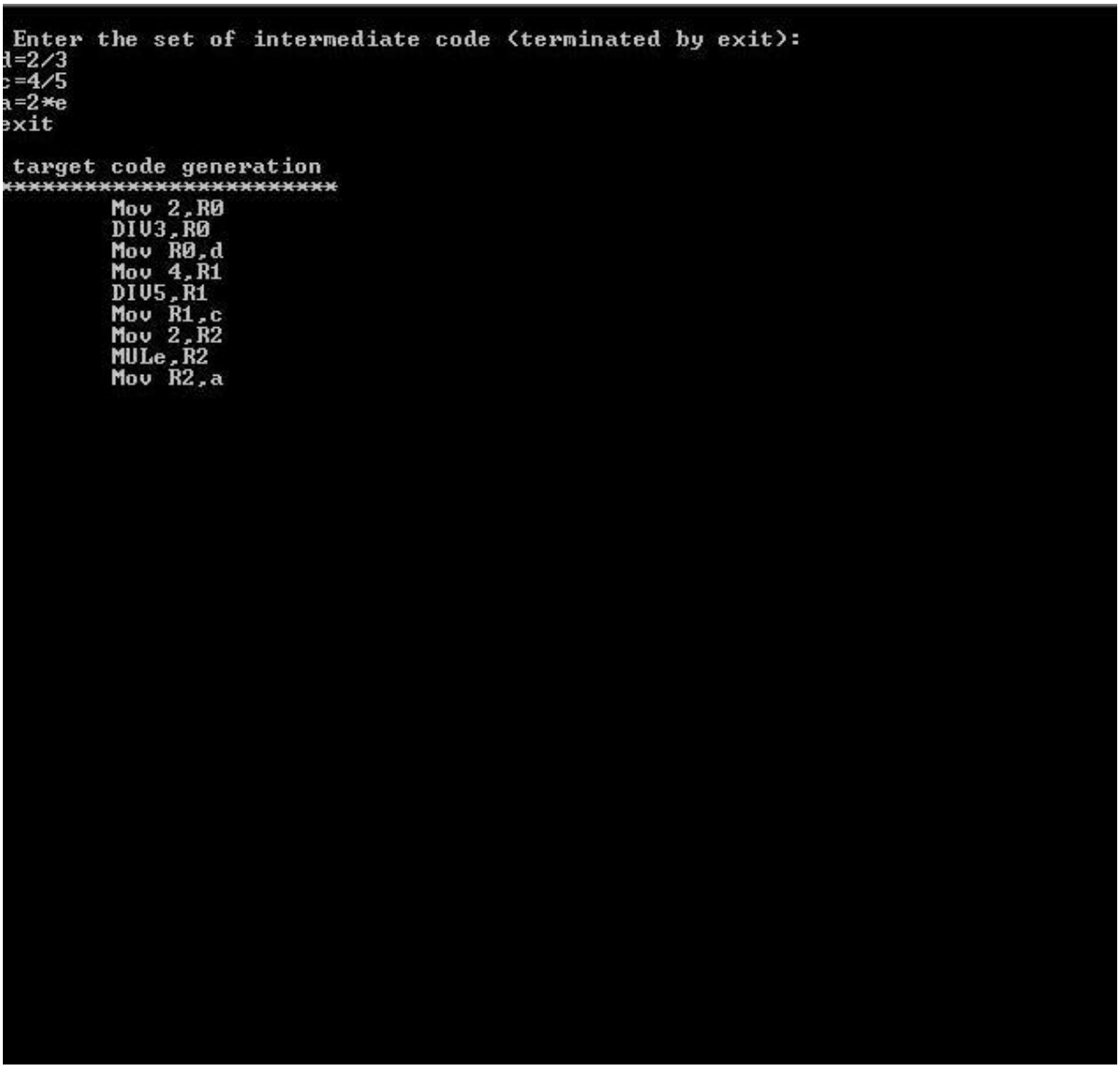
1. Start the program
2. Open the source file and store the contents as quadruples.
3. Check for operators, in quadruples, if it is an arithmetic operator generator it or if assignment operator generates it, else perform unary minus on register C.
4. Write the generated code into output definition of the file in outp.c
5. Print the output.
6. Stop the program.

PROGRAM: **(BACK END OF THE COMPILER)**

```
#include<stdio.h>

#include<stdio.h>

//#include<conio.h>

#include<string.h>

void main()

{

char icode[10][30],str[20],opr[10];

int i=0;

//clrscr();

printf("\n Enter the set of intermediate code (terminated
by exit):\n");

 do

{

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

} while(strcmp(icode[i++],"exit")!=0);

printf("\n target code generation");
```

```
printf("\n************************");
i=0;
do
{
strcpy(str,icode[i]);
switch(str[3])
{
case '+':
strcpy(opr,"ADD");
 break;
case '-':
 strcpy(opr,"SUB");
break;
case '*':
strcpy(opr,"MUL");
 break;
case '/':
strcpy(opr,"DIV");
 break;
}
printf("\n\tMov %c,R%d",str[2],i);
printf("\n\t%s%c,R%d",opr,str[4],i);
printf("\n\tMov R%d,%c",i,str[0]);
}while(strcmp(icode[++i],"exit")!=0);
//getch();
}
```

OUTPUT:



RESULT:

Thus the program was implemented to the TAC has been successfully executed.