Experiment no-2

AIM:

To implement a Symbol table with functions to create, insert, modify, search and displaying C language.

ALGORITHM:

1. Start the program
2. Define the structure of the symbol table
3. Enter the choice for performing the operations in the symbol table
4. If choice is 1, search symbol table for the symbol to be inserted. If the symbol is already present display "Duplicate Symbol", else insert symbol and corresponding address in the symbol table
5. If choice is 2, symbols present in the symbols table are displayed
6. If choice is 3, symbol to be deleted is searched in the symbol table, if found deletes else Displays "Not Found".
7. If choice is 5, the symbol to be modified is searched in the symbol table. The label or address or both can be modified.

PROGRAM:

```
/* C Program to implement SYMBOL TABLE */
#include<stdio.h>
#include<conio.h>
#include<alloc.h>
#include<string.h>
#define null 0
int size=0;
void insert();
void del();
int search(char lab[]);
void modify();
void display();
struct symbtab
{
char label[10];
int addr;
struct symbtab *next;
};
struct symbtab *first,*last;
void main()

{
 int op;
int y;
char la[10];
clrscr();
do
{
printf("\n SYMBOL TABLEIMPLEMENTATION\n");
printf("1.INSERT\n");
printf("2.DISPLAY\n");
printf("3.DELETE\n");
printf("4.SEARCH\n");
printf("5.MODIFY\n");
printf("6.END\n");
printf("\nEnter your option: ");
scanf("%d",&op);
```

```
switch(op)
{
case 1:
insert();
display();
break;
case 2:
display();
break;
case 3:
del();
display();
break;
case 4:
printf("Enter the label to be searched: ");
scanf("%s",la);
y=search(la);
if(y==1)
printf("The label is already in the symbol table\n");
elseprintf("The label is not found in the symbol table!\n");
break;
case 5:modify();
display();
break;
case 6:
break;
} }
while(op<6);
getch();
}
void insert()
{
 int n;
char l[10];
printf("Enter the label: ");
scanf("%s",l);n=search(l);
if(n==1)
printf("The label is already in the symbol table. Duplicate cannot be inserted\n");
else
{
struct symbtab *p;
p=malloc(sizeof(struct symbtab));
strcpy(p->label,l);
printf("Enter the address: ");
scanf("%d",&p->addr);
p->next=null;
if(size==0)
{
first=p;
last=p;
}
Else
{
 last->next=p;
last=p;
}
```

```
size++;
}}
void display()
{
 int i;
struct symbtab *p;
p=first;
printf("LABEL\tADDRESS\n");
for(i=0;i<size;i++)
{
printf("%s\t%d\n",p->label,p->addr);
p=p->next;
}}
 int search(char lab[])
{
int i,
flag=0;
struct symbtab *p;
p=first;for(i=0;i<size;i++)
{
 if(strcmp(p->label,lab)==0
{
flag=1;
}
p=p->next;
}
return flag;
}
void modify()
{
char l[10],nl[10];
int add,choice,i,s;
struct symbtab *p;
p=first;
printf("What do you want to modify?\n");
printf("1.Only the label\n");
printf("2.Only the address of a particular label\n");
printf("3.Both the label and address\n");
printf("Enter your choice: ");
scanf("%d",&choice);
switch(choice)
{
case 1:
printf("Enter the old label\n");
scanf("%s",l);
printf("Enter the new label: ");
scanf("%s",nl);
s=search(l);
if(s==0)
printf("\nNo such label");
else
{for(i=0;i<size;i++)
{
 if(strcmp(p->label,l)==0)
{
strcpy(p->label,nl);
```

```
}
p=p->next;
}}
break;
case 2:
printf("Enter the label whose address is to be modified: ");
scanf("%s",l);
printf("Enter the new address:");
scanf("%d",&add);
s=search(l);
if(s==0)
printf("\nNo such label");
else
{
for(i=0;i<size;i++)
{
 if(strcmp(p->label,l)==0)
{p->addr=add;}p=p->next;
}}
break;
case 3:
printf("Enter the old label: ");
scanf("%s",l);
printf("Enter the new label: ");
scanf("%s",nl);
printf("Enter the new address: ");
scanf("%d",&add);
s=search(l);
if(s==0)
printf("\nNo such label");
else
{
for(i=0;i<size;i++)
{ if(strcmp(p->label,l)==0)
{strcpy(p->label,nl);
p->addr=add;}p=p->next;
}}
break;
}}
void del()
{
 int a;
char l[10];
struct symbtab *p,*q;
p=first;
printf("Enter the label to be deleted: ");
scanf("%s",l);
a=search(l);
if(a==0)
printf("Label not found\n");
else{ if(strcmp(first->label,l)==0)first=first->next;
else if(strcmp(last->label,l)==0)
{q=p->next;
while(strcmp(q->label,l)!=0){p=p->next;
q=q->next;
 }
```

```
p->next=null;
last=p;
}
else{q=p->next;
while(strcmp(q->label,l)!=0)
{
p=p->next;
q=q->next;
}
p->next=q->next;
}
size--;
}}
```

Experiment no-3

IMPLEMENTATION OF SINGLE PASS ASSEMBLER

```
#include<stdio.h>
#include<string.h>
#include<ctype.h>
#include<conio.h>
struct symtab
{
char symbol[20];
int addr;
char ch;
}
stab[20];
void main()
{
char cc[10],opcode[25],label[25],operand[25],object[10];
int s,entry=0,flag=0,c=0,i,op,locctr,code,length;
char mne[20][20]={"LDA","STA","ADD","COMP","J","JLT"};
char opval[20][20]={"1A","1C","2B","1E","33","L9"};
FILE *fp1,*fp2;
fp1=fopen("sinput.txt","r");
fp2=fopen("sobj.txt","w");
fscanf(fp1,"%s%s",&label,&opcode);
if(strcmp(opcode,"START")==0)
fscanf(fp1,"%d",&locctr);
s=locctr;
fscanf(fp1,"%s%s%s",&label,&opcode,&operand);
while(!feof(fp1)) //strcmp(opcode,"END")!=0)
{
if(strcmp(opcode,"RESW")==0){op=atoi(operand);
locctr=locctr+(op*3);
}
else if(strcmp(opcode,"RESB")==0)
{
op=atoi(operand);

locctr=locctr+op;
}
Else
 if(strcmp(opcode,"BYTE")==0){op=strlen(operand);
locctr=locctr+op;
}
else if(strcmp(opcode,"WORD")==0)
{
locctr=locctr+3;
code=atoi(operand);
}
elselocctr=locctr+3;
fscanf(fp1,"%s%s%s",&label,&opcode,&operand);
}
length=locctr-s;
rewind(fp1);
//----------------------------------
fscanf(fp1,"%s%s",&label,&opcode);
printf("%s%s",label,opcode);
```

```
getch();
if(strcmp(opcode,"START")==0)
fscanf(fp1,"%d",&locctr);
s=locctr;
fprintf(fp2,"H^%s^00%d^00%d",label,locctr,length);
fprintf(fp2,"\nT^%d^1E",locctr);
fscanf(fp1,"%s%s%s",&label,&opcode,&operand);
while(!feof(fp1))
{
strcpy(object,"");
flag=0;
if(strcmp(label,"-")!=0)
{
entry=0;for(i=0;i<c;i++)

{
if(strcmp(stab[i].symbol,label)==0 && stab[i].ch=='*')
{
fprintf(fp2,"\nT^%d^02^%d",stab[i].addr,locctr);
stab[c].addr=locctr;stab[c].ch='d';
entry=1;
fprintf(fp2,"\nT^%d^1E",locctr);
}}
if(entry==0)
{
strcpy(stab[c].symbol,label);
stab[c].addr=locctr;stab[c].ch='d';
}
c++;
}
entry=0;
for(i=0;i<7;i++)
{
if(strcmp(opcode,mne[i])==0)
{
strcpy(object,opval[i]);
entry=1;break;
}}
if(entry==1)
{
for(i=0;i<c;i++)
{
if(strcmp(operand,stab[i].symbol)==0){code=stab[i].addr;flag=1;
break;
}}
if(flag==0)
{
code=0;
strcpy(stab[c].symbol,operand);
stab[c].addr=locctr+1;
stab[c].ch='*';
c++;
}}
if(strcmp(opcode,"RESW")==0)
{op=atoi(operand);locctr=locctr+(op*3);
}
```

```
Else
 if(strcmp(opcode,"RESB")==0)
{op=atoi(operand);locctr=locctr+op;
}
Else
 if(strcmp(opcode,"BYTE")==0)
{op=strlen(operand);
locctr=locctr+op;
}
else if(strcmp(opcode,"WORD")==0)
{locctr=locctr+3;
code=atoi(operand);
}
Else
locctr=locctr+3;
if(strcmp(opcode,"RESW")!=0 && strcmp(opcode,"RESB")!=0)
{
if(strcmp(opcode,"BYTE")==0)
{
fprintf(fp2,"^%s",object);
for(i=2;i<op-1;i++)
fprintf(fp2,"%d",toascii(operand[i]));
}
Else
fprintf(fp2,"^%s%d",object,code);
}
fscanf(fp1,"%s%s%s",&label,&opcode,&operand);
}
fprintf(fp2,"\nE^%d",s);
fcloseall();
}
```

Experiment no-4

Implementation of pass 2 compiler

Source code program in c pass two of pass two assembler

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
#include<stdlib.h>
void main()
{
char a[10],ad[10],label[10],opcode[10],operand[10],mnemonic[10],symbol[10];
int i,address,code,add,len,actual_len;
FILE *fp1,*fp2,*fp3,*fp4;
clrscr();
fp1=fopen("assmlist.dat","w");
fp2=fopen("symtab.dat","r");
fp3=fopen("intermediate.dat","r");
fp4=fopen("optab.dat","r");
fscanf(fp3,"%s%s%s",label,opcode,operand);
if(strcmp(opcode,"START")==0)
{
fprintf(fp1,"\t%s\t%s\t%s\n",label,opcode,operand);
fscanf(fp3,"%d%s%s%s",&address,label,opcode,operand);
}
while(strcmp(opcode,"END")!=0)
{
if(strcmp(opcode,"BYTE")==0)
{
fprintf(fp1,"%d\t%s\t%s\t%s\t",address,label,opcode,operand);
len=strlen(operand);
actual_len=len-3;
for(i=2;i<(actual_len+2);i++)
{
itoa(operand[i],ad,16);
fprintf(fp1,"%s",ad);
}
fprintf(fp1,"\n");
}
else if(strcmp(opcode,"WORD")==0)
{
len=strlen(operand);
itoa(atoi(operand),a,10);
fprintf(fp1,"%d\t%s\t%s\t%s\t00000%s\n",address,label,opcode,operand,a);
}
else if((strcmp(opcode,"RESB")==0)(strcmp(opcode,"RESW")==0))
{
fprintf(fp1,"%d\t%s\t%s\t%s\n",address,label,opcode,operand);
}
else
{
rewind(fp4);
fscanf(fp4,"%s%d",mnemonic,&code);
while(strcmp(opcode,mnemonic)!=0)
fscanf(fp4,"%s%d",mnemonic,&code);
if(strcmp(operand,"**")==0)
{
fprintf(fp1,"%d\t%s\t%s\t%s\t%d0000\n",address,label,opcode,operand,code);
```

```
}
else
{
rewind(fp2);
fscanf(fp2,"%s%d",symbol,&add);
while(strcmp(operand,symbol)!=0)
{
fscanf(fp2,"%s%d",symbol,&add);
}
fprintf(fp1,"%d\t%s\t%s\t%s\t%d%d\n",address,label,opcode,operand,code,add);
}
}
fscanf(fp3,"%d%s%s%s",&address,label,opcode,operand);
}
fprintf(fp1,"%d\t%s\t%s\t%s\n",address,label,opcode,operand);
printf("Finished");
fclose(fp1);
fclose(fp2);
fclose(fp3);
fclose(fp4);
getch();
}
```

INPUT FILES

INTERMEDIATE.DAT

```
** START 2000
2000 ** LDA FIVE
2003 ** STA ALPHA
2006 ** LDCH CHARZ
2009 ** STCH C1
2012 ALPHA RESW 1
2015 FIVE WORD 5
2018 CHARZ BYTE C'EOF'
2019 C1 RESB 1
2020 ** END **
```

OPTAB.DAT

```
LDA 33
STA 44
LDCH 53
STCH 57
END *
```

SYMTAB.DAT

```
ALPHA 2012
FIVE 2015
CHARZ 2018
C1 2019
```

Experiment no-5

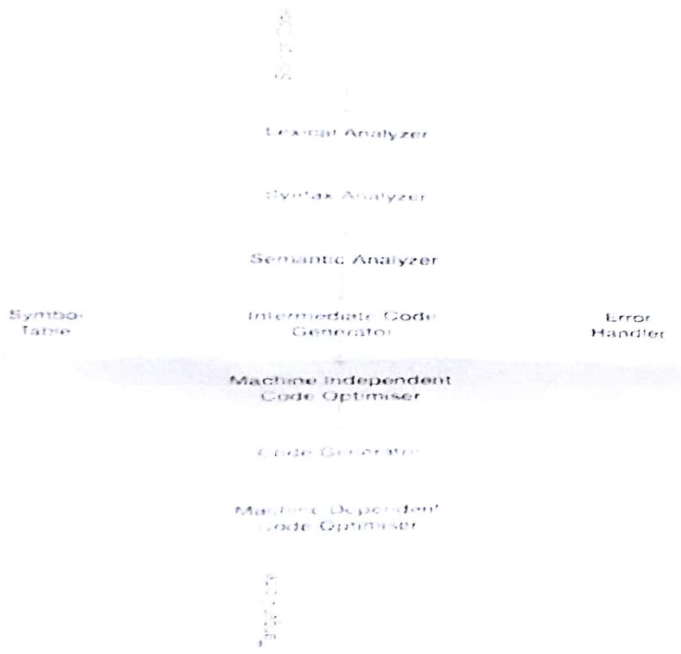
Aim- study of phases of compiler.

compiler is a program that converts a source program written in high level programming language into target program which is machine understandable language. The most common reason for converting source code is to create an executable program. Generally, the target program is an executable program that can be used by the user to process the input and produce the related output. The compiler works with two prime features of the language syntax and semantics. If the compiled program can run on a computer whose CPU or operating system is different from the one on which compiler runs, the compiler is known as a cross-compiler. More generally, compilers are the specific type of translator. A program that translates from a low level language to a higher level language is a decompiler.

PHASES OF COMPILER

Each phase transforms the source program from one representation into another representation. They communicate with error handlers and symbol table. But in this customized compiler, we are going through only necessary three phases of the compiler i.e.Lexical Phase,Syntax Phase and
Target Code Generation.

There are six phases of compiler



Phase I
(Lexical Analyzer) - This phase reads the source code as a stream of characters and converts it into meaningful lexemes. A token describes a pattern of characters having same meaning in the source program (such as identifiers, operators, keywords, and numbers).

Phase II
(Syntax Analyzer) – This phase generates the syntax tree according to the already known CFG. A syntax analyzer is also called a parser. A parse tree describes the syntactic structure.

Phase III

(Semantic Analyzer) – This phase checks the source program for semantic errors and collects the type of information for the code generation. The main functionality is type checking.

Phase IV
(ICG) – ICG stands for Intermediate Code Generator. After semantic analysis intermediate code is generated, which is in between high level language and machine language. These codes are generally machine architecture independent, but the level of intermediates code is close to the level of machine codes.

Phase V
(Code Optimizer) – This phase removes unnecessary code lines, and arranges the sequence of statements in order to speed up the program execution without wasting resources (CPU, Memory).

Phase VI
(Code Generation) – In this phase the code generator takes the optimized representation of intermediate code and converts it into machine understandable code (Target Code).

Experiment no=6

write a program for dividing the given input program into lexemes.

ALGORITHM:

Step 1: start

Step 2: Read the file and open the file as read mode.

Step 3: Read the string for tokens identifiers, variables.

Step 4: take parenthesis also a token.

Step 5: parse the string

Step 6: stop

Program

```
#include<stdio.h>

#include<conio.h>

#include<string.h>

main()

{

int i,j,k,p,c;

char s[120],r[100];

char par[6]={'(',')','{','}','[',']'};

char sym[9]={',',';',':','.','<','>','?','S','='};

char key[9][10]={"main","if","else","switch","void","do","while","for","return"}; char
dat[4][10]={"int","float","char","double"}; char opr[5]={'*','+','-','/','^'};

FILE *fp;

clrscr();

printf("\n\n\t enter the file name");

scanf("%s",s);

fp=fopen(s,"r");

c=0;

do

{ fscanf(fp,"%s",r); getch(); for(i=0;i<6;i++) if(strchr(r,par[i])!=NULL)

printf("\n paranthesis :%c",par[i]); for(i=0;i<9;i++) if(strchr(r,sym[i])!=NULL) printf("\n symbol :%c",sym[i]);
for(i=0;i<9;i++) if(strstr(r,key[i])!=NULL) printf("\n keyword :%s",key[i]); for(i=0;i<4;i++)
if((strstr(r,dat[i])&&(!strstr(r,"printf")))!=NULL){
printf("\n data type :%s",dat[i]); fscanf(fp,"%s",r);
```

```
 printf("\n identifiers :%s",r);

 }

 for(i=0;i<5;i++)

 if(strchr(r,opr[i])!=NULL) printf("\n operator :%c",opr[i]); p=c;
 c=ftell(fp);

 }

 while(p!=c);

 return 0; }
```

Experiment no-7

AIM: write a program for implementing a Lexical analyzer using LEX tool in Linux platform.

ALGORITHM.

1. Lex program contains three sections: definitions, rules, and user subroutines. Each section must be separated from the others by a line containing only the delimiter, %%.

   The format is as follows:

   definitions

   %%

   rules

   %%

   user_subroutines

2. In definition section, the variables make up the left column, and their definitions make up the right column. Any C statements should be enclosed in %{..}%. Identifier is defined such that the first letter of an identifier is alphabet and remaining letters are alphanumeric.
3. In rules section, the left column contains the pattern to be recognized in an input file to yylex(). The right column contains the C program fragment executed when that pattern is recognized. The various patterns are keywords, operators, new line character, number, string, identifier, beginning and end of block, comment statements, preprocessor directive statements etc.
4. Each pattern may have a corresponding action, that is, a fragment of C source code to execute when the pattern is matched.
5. When yylex() matches a string in the input stream, it copies the matched text to an external character array, yytext, before it executes any actions in the rules section.
6. In user subroutine section, main routine calls yylex(). yywrap() is used to get more input.
7. The lex command uses the rules and actions contained in file to generate a program, lex.yy.c, which can be compiled with the cc command. That program can then receive input, break the input into the logical pieces defined by the rules in file, and run program fragments contained in the actions in file.

```
%{
#include<stdio.h>
 #include<string.h>
int i;
%}
%%
[a-z A-Z]*
 { for(i=0;i<=yyleng;i++)
{
if((yytext[i]=='a')&&(yytext[i+1]=='b')&&(yytext[i+2]=='c'))
{
yytext[i]='A';
yytext[i+1]='B';
yytext[i+2]='C';
}
}
printf("%s",yytext);
}
[\t]* return;
.* {ECHO;}
\n {printf("%s",yytext);}
%%
main()
{
yylex();
}
int yywrap()
{
return 1;
}
```

OUTPUT:

```
[CSE@localhost ~]$ lex lex1.l
 [CSE@localhost ~]$ cc lex.yy.c
 [CSE@localhost ~]$. /a.out

abc ABC
```

```
%{
#include<stdio.h>
 #include<string.h>
int i;
%}
%%
[a-z A-Z]*
 { for(i=0;i<=yyleng;i++)
{
if((yytext[i]=='a')&&(yytext[i+1]=='b')&&(yytext[i+2]=='c'))
{
yytext[i]='A';
yytext[i+1]='B';
yytext[i+2]='C';
}
}
printf("%s",yytext);
}
[\t]* return;
.* {ECHO;}
\n {printf("%s",yytext);}
%%
main()
{
yylex();
}
int yywrap()
{
return 1;
}
```

OUTPUT:

```
[CSE@localhost ~]$ lex lex1.l
 [CSE@localhost ~]$ cc lex.yy.c
 [CSE@localhost ~]$. /a.out

abc ABC
```

Exp No: 8

AIM: Write a program to compute FIRST function.

ALGORITHM:

Step 1: Start

Step 2: Declare FILE *fp

Step 3: open the file in.txt in write mode

Step 4: Read the Productions

Step 5: Compute Follow function

Step 6: stop the productions.

```
/* FIRST FUNCTION*/

#include<stdio.h>

#include<string.h>

char res[10]={" "},first[10];

int l=0,count=0,j,term=0;

FILE *fp1;

void main()

{

FILE *fp;

int i=0,k=0,n,a[10],set,tem[10]={0},t;

char ch,s;

clrscr();

printf("Enter the productions..\n");

fp=fopen("input.txt","w");

while((ch=getchar())!=EOF)

putc(ch,fp);

fclose(fp);

fp=fopen("input.txt","r");

/*calculation of production starting variable*/

while(!(feof(fp)))

{
```

```
ch=fgetc(fp);
if(feof(fp))
break;
first[l++]=ch;
count++;
a[i++]=count;
while(ch!='\n')
{
count++;
ch=fgetc(fp);
}
count++;
}
rewind(fp);
n=l;
clrscr();
j=0;
l=0;
while(!(feof(fp)))
{
ch=fgetc(fp);
if(feof(fp))
break;
while(ch!='\n')
{
ch=fgetc(fp);
if(count==1)
{
if(((ch>='a')&&(ch<='z'))||ch=='+'||ch=='-'||ch=='*'||ch=='/'||ch=='^'||ch==')'||ch=='('||(ch=='^')||ch=='#')
{
```

```
if(term!=1||ch!='#')
unione(ch,j++);
if((term==1)&&(ch=='#'))
term=2;
count=0;
}
else
{
tem[++k]=ftell(fp);
set=1;
}
}
if(ch=='>'||ch=='l')
{
count=1;
}
if(set==1)
{
for(i=0;i<n;i++)
{
fseek(fp,a[i]-1,0);
s=fgetc(fp);
if(s==ch)
{
term=1;
break;
}
}
count=0;
```

```
set=0;
}
}
if(tem[k]!=0){
t=tem[k]-1;
tem[k--]=0;
fseek(fp,t,0);
if(term==2)
count=1;
fgetc(fp);
ch=fgetc(fp);
if((k==0&&term==2)&&(ch=='\n'||ch=='l'))
unione('#',j++);
fseek(fp,t,0);
}
else
j=print();
}
getch();
}
unione(char ch,int j)
{
int i;
for(i=0;i<j;i++)
if(res[i]==ch)
return;
res[++j]=ch;
}
print()
```

```
 {
 static int k=0;
 if(k==0)
 {
 fp1=fopen("output.txt","w");
 k=1;
 }
 printf("first[%c]==",first[l]);
 fputc(first[l++],fp1);
 for(j=0;res[j]!='\0';j++)
 {
 printf("%c",res[j]);
 fputc(res[j],fp1);
 }
 printf("\n");
 for(j=0;res[j]!='\0';j++)
 res[j]=' ';
 count=0;
 term=0;
```

```
fputc('\n',fp1);

return 0;

}
```

FOLLOW FUNCTION Experiment no-9

AIM: Write a program to compute FOLLOW function.

ALGORITHM:

Step 1: Start

Step 2: Declare FILE *fp

Step 3: open the file in.txt in write mode

Step 4: Read the Productions

Step 5: Compute Follow function

Step 6: stop the productions.

/*computation of FOLLOW(A) */

```
#include<stdio.h>
#include<conio.h>
char ch,first[10],stack[10];
int i,j,k;
void main()
{
FILE *fp;
clrscr();
fp=fopen("in txt","w");
printf("Enter the productions..\n");
while(((ch=getchar())!='@'))
putc(ch,fp);
fclose(fp);
fp=fopen("in txt","r");
i=0;
while(!(feof(fp)))
{
ch=fgetc(fp);
if(feof(fp))
break;
```

```
first[i++]=ch;
while(ch!='\n')
ch=fgetc(fp);
}
rewind(fp);
i=0;j=0;
while(first[i]!='\0')
{
ch=getc(fp);
if(ch==first[i])
stack[j]='$';
else
while(!(feof(fp)))
{
while(ch!='>')
ch=getc(fp);
while(ch!=first[i])
{
if(feof(fp))goto down;
ch=fgetc(fp);
}
ch=fgetc(fp);
stack[j]=ch;
down : j++;
}
print();
i++;
}
```

```
getch();
}
print()
{
printf("FOLLOW[%c]={",first[i]);
for(k=0;stack[k]!='\0';k++)
printf("%c",stack[k]);
printf("}\n");
return 0;
```

OPERATOR PRECEDENCE PARSING Exp No: 10

AIM: Write a program to implement operator precedence parsing.

ALGORITHM:

Step 1: start.

Step 2: Declare the prototypes for functions.

Step 3: Enter the String like id*id+id

Step 4: Read the string and analyze tokens, identifiers, variables.

Step 5: Display the operator precedence table.

Step 6: stop.

PROGRAM:

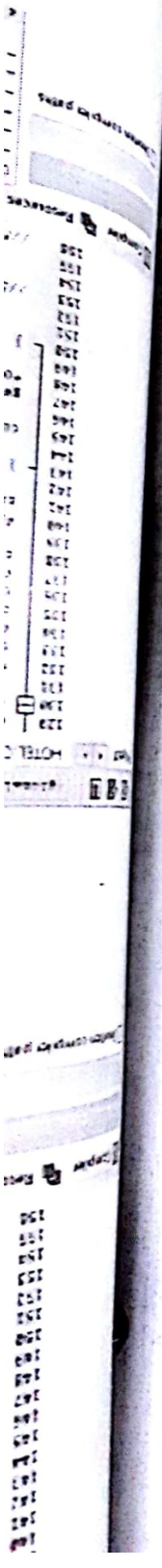
```
//OPERATOR PRECEDENCE PARSER

#include<stdio.h>

#include<conio.h>

#include<string.h>

char *p;

e();

t();

main()

{

int i,j=0,n,b[10],k=0;

char a[10],c[10];

clrscr();

printf("Enter the string\n");

scanf("%s",a);

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

{

switch(a[i])

{

case '+' :

case '-' :
```

```
c[j]=a[i];
b[j]=1;
j++;
break;
case '*' :
case '/' :
c[j]=a[i];
b[j]=2;
j++;
break;
case '^' :
c[j]=a[i];
b[j]=3;
j++;
break;
default :
if(k==0)
{
k=1;
c[j]=a[i];b[j]=4;
j++;
}
}
}
c[j]='$';
b[j]=0;
j++;
printf("\n\n");
printf("\n\t----------");
```

```
printf("\n\n");
for(i=0;i<j;i++)
printf("\t%c",c[i]);
printf("\t");
for(i=0;i<j;i++)
{
printf("\n\t-------------------------------");
printf("\n\n%c",c[i]);
for(n=0;n<j;n++)
{
if(b[i]<b[n])
printf("\t<");
if(b[i]>b[n])
printf("\t>");
if(b[i]==b[n])
printf("\t=");
}
printf("\t");
}
printf("\n\t-----------------------------------");
p=a;
if(e())
printf("\n\n String parsed");
else
printf("\n String not parsed");
getch();
return 0;
}
int e()
```

```
{
if(*p=='i')
{
p++;
t();
t();
}
```

```
else
return 0;
}
int t()
{
if(*p==NULL)
return 1;
else
if(*p=='+'||*p=='*')
{
p++;
if(*p=='i')
{
p++;
}
else
{
return 0;
}
}
else
return 0;
}
```



RECURSIVE DECENDENT PARSING Exp No: 11

AIM: To write a program on recursive descendent parsing.

ALGORITHM:

Step 1: start.

Step 2: Declare the prototype functions E() , EP(),T(), TP(),F()

Step 3: Read the string to be parsed.

Step 4: Check the productions

Step 5: Compare the terminals and Non-terminals

Step 6: Read the parse string.

Step 7: stop the production

PROGRAM:

```
//RECURSIVE DECENT PARSER

#include<stdio.h>

#include<conio.h>

char *p;

E();

EP();

T();

F();

TP();

int main()

{

clrscr();

printf("The grammer is\n");

printf("E->E+T/T\n");

printf("T->T*F/F\n");

printf("F->(E)/id\n");

printf("\n Enter the string to be parsed:");

scanf("%s",p);
```

```
    if(E())
    printf("\nString parsed");
    else
    printf("\nString not parsed");
    getch();
    return 0;
    }
    E()
    {
    T();
    if(EP())
    return 1;
    else
    return 0;
    }
    EP()
    {
    if(*p=='+')
    {
    p++;
    T();
    EP();}
    else if(*p=='\0')
    return 1;
    else
    return 0;
    }
    T()
```

```
{
F();
if(TP())
return 1;
else
return 0;
}
TP()
{
if(*p=='*')
{
p++;
F();
TP();
}
else if(*p=='\0')
return 0;
}
F()
{
if(*p=='i')
p++;
if(*p=='d')
p++;
else if(*p=='(')
{
p++;
F();
```

```
F();
}
else if(*p==')')
p++;
else
return 0;
}
```