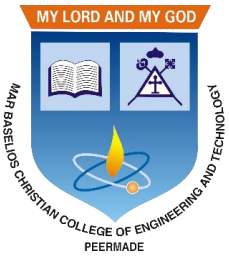

MAR BASELIOS CHRISTIAN
COLLEGE OF ENGINEERING & TECHNOLOGY
KUTTIKKANAM, PEERMADE

Department of Computer Science & Engineering

Lab Manual

COMPILER LAB
(CSL 411)

B.Tech Computer Science Seventh Semester

SL NO	PARTICULARS
1.	Institute Vision & Mission
2.	Department Vision & Mission ,PEO;PO & PSO Statements
3.	Lab Course Syllabus
COMPILER LAB PROGRAMS	
1.	Design and implement a lexical analyzer using C language
2.	Implement lexical analyzer using the tool LEX
3.	Generate a YACC specification to recognize a valid arithmetic expression that uses operators +, – , *,/ and parenthesis.
4.	Generate a YACC specification to recognize a valid identifier which starts a letter followed by any number of letters or digits.
5.	Implementation of Calculator using LEX and YACC
6.	Write a program to convert NFA with ε transition to NFA without ε transition.
7.	Design and implement a recursive descent parser for a given grammar.
8.	Construct a Shift Reduce Parser for a given language.
9.	Simulation of code optimization Techniques
10.	Implement Intermediate code generation for simple expressions.
11.	Write a program to convert NFA to DFA
12.	Implement the back end of the compiler.

COLLEGE VISION & MISSION STATEMENT

VISION

An Engineering Institute with global quality to groom competent engineers equipped to address the changing needs of society.

MISSION

Our efforts are dedicated for developing a learner centric education environment to:

- Provide value-based technical learning
- Practice real world problem solving
- Foster team work in engineering design
- Inspire innovations and R&D

COMPUTER SCIENCE AND ENGINEERING DEPARTMENT

VISION

To become an excellent department by empowering the graduates to address the challenges in the fast changing scenario, adhering to ethical values.

MISSION

- Inculcate skills to solve complex engineering problems and instill entrepreneurial capabilities.
- Equip the students to advance with the recent trends and technology through Research and Innovations in the field of computer science.
- Facilitate the development of professional behavior, ethical values and the spirit of service to the society.

SYLLABUS

1. Implementation of lexical analyzer using the tool LEX.
2. Implementation of Syntax analyzer using the tool YACC.
3. Application problems using NFA and DFA.
4. Implement Top-Down Parser.
5. Implement Bottom-up parser.
6. Simulation of code optimization Techniques.
7. Implement Intermediate code generation for simple expressions.
8. Implement the back end of the compiler.

Exp 1:

LEXICAL ANALYSIS USING C

AIM

To write a program to implement Lexical Analysis using C.

ALGORITHM

Step 1 : Start the program
Step 2 : Declare necessary variables and file pointer.
Step 3 : Initialize the necessary variables.
Step 4: Open input file in read mode
Step 5: Repeat Step 8 to Step 19 until EOF of input file.
Step 6: Read each character from input file
Step 7: If the character is '/' Then
Read the next character from input file
If the character is '/' Then
Read rest of the characters from the input file until '\n' occurs
Step 8: Else If the character is '*' Then
Read rest of the characters from the input file until adjacent '*' and '/' occurs
Step 9: Else
Undo the operation of reading a character from input file in step 7
Step 10: Read one character from input file
Step 11: If the character is an operator , then print it is an operator
Step 12: Else If the character is a special symbol, then print it is a special character.
Step 13: Else If the character is a digit Then
Store it in an array dig[] and read next character fron the input file
Step 14: Repeat the steps12.1 and 12.2 until a non digit character is reached.
12.1: Read next character from input file
12.2: Store it in dig[]
Step 15: Undo the operations in step12.1 using ungetc()
Step 16: Print it is a Number.
Step 17: Else If the character is an alphabet Then
Store it in an array str[] and read next character fron the input file
Step 18: Repeat the steps16.1 and 16.2 until a non alphanumeric or space t character is reached.
16.1: Read next character from input file
16.2: Store it in str[]
Step 19: Undo the operations in step16.1 using ungetc()
Step 20: If str is a keyword Then Print it is a Keyword.
Step 21: Else print it is an Identifier.
Step 22: Close input file
Step 23 : Stop the program

PROGRAM

```
#include<stdio.h>
#include<ctype.h>
#include<string.h>
int main()
{
FILE *input;
int l=1;
int t=0;
int j=0;
int i,flag,k;
char ch,ch1,str[20],dig[20];
input = fopen("input.c","r");
char keyword[30][30] = {"int","void","if","else","do","while"};
printf("Line no. \t Token no.\t\t Token \t\t Lexeme\n");
while(!feof(input))
{
i=0,k=0;
flag=0;
ch=fgetc(input);
if(ch=='/')
{
  ch=fgetc(input);
  if(ch=='/')
   {
        while((ch=fgetc(input))!='\n');
          if(ch=='\n')
                  l++;
             }
   else if(ch=='*')
   {
     ch=fgetc(input);
     ch1=fgetc(input);
     while(ch!='*'&&ch1!='/')
       {
          ch=ch1;
          ch1=fgetc(input);
          if(ch=='\n')
                  l++;
          }

   }
else
{
ungetc(ch,input);
}
ch=fgetc(input);
}

if( ch=='+' || ch== '-' || ch=='*' || ch=='/' ||ch=='>'||ch=='<'||ch=='='||ch=='%'||ch=='&'||ch=='|')
{
printf("%7d\t\t %7d\t\t Operator\t %7c\n",l,t,ch);
```

```
t++;
}
else if( ch==';' || ch=='{' || ch=='}' || ch=='(' || ch==')' || ch=='?' ||ch=='@' || ch=='!'
||ch=='#'||ch==','||ch=='.'||ch=="")
{
printf("%7d\t\t %7d\t\t Special symbol\t %7c\n",l,t,ch);
t++;
}
else if(isdigit(ch))
{
dig[k]=ch;
k++;
ch=fgetc(input);
while(isdigit(ch))
{
dig[k]=ch;
k++;
ch=fgetc(input);
}
ungetc(ch,input);
dig[k]='\0';
printf("%7d\t\t %7d\t\t Number\t\t %7s\n",l,t,dig);
t++;
}
else if(isalpha(ch))
{
str[i]=ch;
i++;
ch=fgetc(input);
while(isalnum(ch) && ch!=' ')
{
str[i]=ch;
i++;
ch=fgetc(input);
}
ungetc(ch,input);
str[i]='\0';
for(j=0;j<=30;j++)
{
if(strcmp(str,keyword[j])==0)
{
flag=1;
break;
}
}
if(flag==1)
{
printf("%7d\t\t %7d\t\t Keyword\t %7s\n",l,t,str);
t++;
}
else
{
printf("%7d\t\t %7d\t\t Identifier\t %7s\n",l,t,str);
```

```
t++;
}

}
else if(ch=='\n')
{
l++;
}

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

OUTPUT

input.c

```
#include<stdio.h>
void main()
{
int a,c;/*Declaration of
 variables*/
/*hi
welcome
to mbc*/
scanf("%d",&a);//for input a value
c=a+10;
//print the result
printf("Sum = %s",c);
}
```

```
administrator@administrator-Vostro-3800:~/cdlab$ gcc lex.c
administrator@administrator-Vostro-3800:~/cdlab$ ./a.out
```

Line no.	Token no.	Token	Lexeme
1	0	Special symbol	#
1	1	Identifier	include
1	2	Operator	<
1	3	Identifier	stdio
1	4	Special symbol	.
1	5	Identifier	h
1	6	Operator	>
2	7	Keyword	void
2	8	Identifier	main
2	9	Special symbol	(
2	10	Special symbol	)
3	11	Special symbol	{
4	12	Keyword	int
4	13	Identifier	a
4	14	Special symbol	,
4	15	Identifier	c
4	16	Special symbol	;
6	17	Identifier	scanf

6	18	Special symbol	(
6	19	Special symbol	"
6	20	Operator	%
6	21	Identifier	d
6	22	Special symbol	"
6	23	Special symbol	,
6	24	Operator	&
6	25	Identifier	a
6	26	Special symbol	)
6	27	Special symbol	;
6	28	Identifier	c
6	29	Operator	=
6	30	Identifier	a
6	31	Operator	+
6	32	Number	10
6	33	Special symbol	;
7	34	Identifier	printf
7	35	Special symbol	(
7	36	Special symbol	"
7	37	Identifier	Sum
7	38	Operator	=
7	39	Operator	%
7	40	Identifier	s
7	41	Special symbol	"
7	42	Special symbol	,
7	43	Identifier	c
7	44	Special symbol	)
7	45	Special symbol	;
8	46	Special symbol	}

Exp 2:

LEXICAL ANALYSER USING LEX

AIM

To implement lexical analyzer using the tool LEX

ALGORITHM

Step1: 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

Step2: 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.

Step3: 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.

Step4: Each pattern may have a corresponding action, that is, a fragment of C source code to execute when the pattern is matched.

Step5: 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.

Step6: In user subroutine section, main routine calls yylex(). yywrap() is used to get more input.

Step7: 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.

PROGRAM

```
letter [a-zA-Z]
digit[0-9]
%%

#.* {printf("\n%s is a preprocessor directive",yytext);}
{digit}+("E"("+"|"-")?{digit}+)? printf("\n%s\tis real number",yytext);
{digit}+"."{digit}+("E"("+"|"-")?{digit}+)? printf("\n%s\t is floating pt no ",yytext);
"void"|"if"|"else"|"int"|"char"|"switch"|"return"|"struct"|"do"|"while"|"void"|"for"|"float" printf("\n%s\t is
keywords",yytext);
"\a"|"\\n"|"\\b"|"\t"|"\\t"|"\b"|"\\a" printf("\n%s\tis Escape sequences",yytext);
{letter}({letter}|{digit})* printf("\n%s\tis identifier",yytext);
"&&"|"<"|">"|"<="|">="|"="|"+"|"-"|"?"|"*"|"/"|"%"|"&"|"||" printf("\n%s\toperator ",yytext);
"{"|"}"|"["|"]"|"("|")"|"#"|""|"."|"\""|"\\"|";"|"," printf("\n%s\t is a special character",yytext);
"%d"|"%s"|"%c"|"%f"|"%e" printf("\n%s\tis a format specifier",yytext);
\n
%%
int yywrap()
{
return 1;
}
int main(int argc,char *argv[])
{
yyin=fopen(argv[1],"r");
yylex();
fclose(yyin);
return 0;
}
```

OUTPUT

input1.c

```
#include<stdio.h>
void main()
{
int a,c;
scanf("%d",&a);
c=a+10;
printf("Sum = %s\n",c);
}
```

```
administrator@administrator-Vostro-3800:~/cdlab$ lex lex1.l
administrator@administrator-Vostro-3800:~/cdlab$ gcc lex.yy.c
administrator@administrator-Vostro-3800:~/cdlab$ ./a.out input1.c

#include<stdio.h>  is a preprocessor directive
void     is keywords
main    is identifier
(        is a special character
)        is a special character
{        is a special character
int      is keywords
a       is identifier
,        is a special character
c       is identifier
;        is a special character
scanf   is identifier
(        is a special character
"        is a special character
%d      is a format specifier
"        is a special character
,        is a special character
&       operator
a       is identifier
)        is a special character
;        is a special character
c       is identifier
=       operator
a       is identifier
+       operator
10      is real number
;        is a special character
printf  is identifier
(        is a special character
"        is a special character
Sum     is identifier
=       operator
%s      is a format specifier
\n      is Escape sequences
"        is a special character
,        is a special character
c       is identifier
)        is a special character
;        is a special character
}        is a special character
```

Exp 3:

TO RECOGNIZE A VALID ARITHMETIC EXPRESSION USING YACC

AIM

To generate a YACC specification to recognize a valid arithmetic expression that uses operators +, – , *,/ and parenthesis.

ALGORITHM

lex

1. In definition section include y.tab.h
2. In rules section,numbers and variables are identified by the rules
3. return that value

yacc

1. Get the input from the user and Parse it token by token.
2. First identify the valid inputs that can be given for a program.
3. The Inputs include digits and alphabets
4. Define the rules
5. Display the possible Error message in yyerror() function
6. Display the a valid message on the screen if the token is satisfied by the rules

PROGRAM

lex Program

```
%{
#include "y.tab.h"
%}
%%
[0-9]+ {return NUMBER;}
[a-zA-Z][_a-zA-Z0-9]* {return ID;}
\n {return 0;}
. {return yytext[0];}
%%
```

yacc Program

```
%{
#include<stdio.h>
#include<stdlib.h>
int yylex();

%}
%token NUMBER ID
%left '+"-"*"/'
%%
exp : exp'+'exp
|exp'-'exp
```

```
|exp'*'exp
|exp'/'exp
|'('exp')'
|NUMBER
|ID;
%%
int main(int argc,char *argv[])
{
printf("Enter the expression: ");
yyparse();
printf("\nValid Expression");
return 0;
}
int yyerror()
{
printf("\nInvalid Expression");
exit(1);
}
int yywrap()
{
return 1;
}
```

OUTPUT

```
administrator@administrator-Vostro-3800:~/cdlab$ lex arithmetic.l
administrator@administrator-Vostro-3800:~/cdlab$ yacc -d arithmetic.y
administrator@administrator-Vostro-3800:~/cdlab$ gcc lex.yy.c y.tab.c
administrator@administrator-Vostro-3800:~/cdlab$ ./a.out
Enter the expression: a+b

Valid Expressionadministrator@administrator-Vostro-3800:~/cdlab$ ./a.out
Enter the expression: (a+b)*c

Valid Expressionadministrator@administrator-Vostro-3800:~/cdlab$ ./a.out
Enter the expression: (a+b

Invalid Expressionadministrator@administrator-Vostro-3800:~/cdlab$ ./a.out
Enter the expression: (a+-b)

Invalid Expressionadministrator@administrator-Vostro-3800:~/cdlab$
```

Exp4:

TO RECOGNIZE A VALID IDENTIFIER

AIM

To generate a YACC specification to recognize a valid identifier which starts a letter followed by any number of letters or digits.

ALGORITHM:

lex

1. To recognize a valid identifier . In definition section include y.tab.h
2. In rules section,digits and alphabets are identified
3. return that value

yacc

1. Get the input from the user and Parse it token by token.
2. First identify the valid inputs that can be given for a program.
3. The Inputs include digits and alphabets
4. Define the rules
5. Display the possible Error message in yyerror() function
6. Display the a valid message on the screen if the token is satisfied by the rules

PROGRAM:

lex program

```
%{
#include"y.tab.h"
%}
%%
[0-9] { return DIGIT;}
[a-zA-Z] { return ALPHA;}
\n { return 0;}
. {return yytext[0];}
%%
```

yacc Program

```
%{
# include<stdio.h>
# include<stdlib.h>
int yylex();
%}
%token DIGIT ALPHA
%%
var:ALPHA
| var ALPHA
| var DIGIT;
```

```
%%
int main(int argc,char *argv[])
{
printf("Enter a variable name: ");
yyparse();
printf("\n Valid Variable");
return 0;
}
int yyerror()
{
printf("\n Invalid variable");
exit(1);
}
int yywrap()
{
return 1;
}
```

OUTPUT

```
administrator@administrator-Vostro-3800:~/cdlab$ lex identifier.l
administrator@administrator-Vostro-3800:~/cdlab$ yacc -d identifier.y
administrator@administrator-Vostro-3800:~/cdlab$ gcc lex.yy.c y.tab.c
administrator@administrator-Vostro-3800:~/cdlab$ ./a.out
Enter a variable name: hai

 Valid Variableadministrator@administrator-Vostro-3800:~/cdlab$ ./a.out
Enter a variable name: dad34

 Valid Variableadministrator@administrator-Vostro-3800:~/cdlab$ ./a.out
Enter a variable name: 2wer

 Invalid variable
```

Exp5:

IMPLEMENTATION OF CALCULATOR USING LEX AND YACC

AIM:
To implementation of calculator using lex and yacc

ALGORITHM:

lex

1. In definition section include stdio.h and y.tab.h and declare yylval as extern int
2. In rules section, if a numeric (0-9) convert that stored in yytext into integer and stored in yyval
3. return that value
4. write yywrap() function

yacc

1. Get the input from the user and Parse it token by token.
2. First identify the valid inputs that can be given for a program.
3. The Inputs include numbers and operators.
4. Define the precedence and the associativity of various operators like +,-,/,* ,%, (,) etc.
5. 5Write codes for displaying the result on the screen.
6. Write codes for performing various arithmetic operations.
7. Display the possible Error message that can be associated with this calculation.
8. Display the output on the screen else display the error message

PROGRAM

lex Program

```
%{
#include<stdio.h>
#include "y.tab.h"
extern int yylval;
%}
%%
[0-9]+ {
yylval=atoi(yytext);
return NUMBER;
}
[\t] ;
[\n] return 0;
. return yytext[0];
%%
int yywrap()
{
return 1;
}
```

yacc Program

```
%{
#include<stdio.h>
int yylex();
int flag=0;
%}
%token NUMBER
%left '+' '-'
%left '*' '/' '%'
%left '(' ')'
%%
ArithmeticExpression: E{
printf("\nResult=%d\n",$$);
return 0;
};
E:E'+'E {$$=$1+$3;}
|E'-'E {$$=$1-$3;}
|E'*'E {$$=$1*$3;}
|E'/'E {$$=$1/$3;}
|E'%'E {$$=$1%$3;}
|'('E')' {$$=$2;}
| NUMBER {$$=$1;}
;
%%
void main()
{
printf("\nEnter Any Arithmetic Expression which can have operations Addition, Subtraction,Multiplication, Divison, Modulus and Round brackets:\n");
yyparse();
if(flag==0)
printf("\nEntered arithmetic expression is Valid\n\n");
}
void yyerror()
{
printf("\nEntered arithmetic expression is Invalid\n\n");
flag=1;
}
```

OUTPUT

```
administrator@administrator-Vostro-3800:~/cdlab$ yacc -d calc.y
administrator@administrator-Vostro-3800:~/cdlab$ gcc lex.yy.c y.tab.c
administrator@administrator-Vostro-3800:~/cdlab$ ./a.out

Enter Any Arithmetic Expression which can have operations Addition, Subtraction,Multiplication, Divison, Modulus and Round brackets:
2+7
Result=9
Entered arithmetic expression is Valid
```

```
administrator@administrator-Vostro-3800:~/cdlab$ ./a.out

Enter Any Arithmetic Expression which can have operations Addition, Subtraction,Multiplication, Divison, Modulus and Round brackets:
(5+7)/2*(6-2)
Result=24
Entered arithmetic expression is Valid
```

Exp6:

PROGRAM TO FIND E – CLOSURE OF ALL STATES OF ANY GIVEN NFA WITH E TRANSITION.

AIM:

To find ε – closure of all states of any given nfa with ε transition

ALGORITHM:

1. Start
2. Read states,input values and transitions for NFA
3. Set the e-closure of all state is set to null(0)
4. By default e-closure of a state(eg:q0) includes that state(q0),so store it to the e-closure array
5. Repeatedly check for e-transitions and destination for each state
6. If it is found Then include in e-closure array
7. Print the e-closure of all states
8. Stop

PROGRAM

```
#include<stdio.h>
#include<stdlib.h>
int findvalue(char c)
    {
        char s[2];
        int d,i;
        s[0]=c;
        s[1]='\0';
        d=atoi(s);
         return(d);
        }


void main()
{
        int noalpha,nostate,notrans,eclos[10][20],i,j,k,d;
        char alpha[5],trans[30][3],c;
        printf("Enter the number of alphabets?\n");
        scanf("%d",&noalpha);
                printf("NOTE:- [ use letter e as epsilon]\n");
        printf("NOTE:- [e must be last character ,if it is present]\n");
        printf("\nEnter alphabets?\n");
        for(i=0;i<noalpha;i++)
        {
    getchar();
    alpha[i]=getchar();
        }
        printf("\nEnter the number of states?\n");
        scanf("%d",&nostate);
        printf("\nEnter no of transition?\n");
        scanf("%d",&notrans);
```

```
        printf("\nEnter transition?\n");
        for(i=0;i<notrans;i++)
        {
        getchar();
        trans[i][0]=getchar();
        getchar();
        trans[i][1]=getchar();
        getchar();
        trans[i][2]=getchar();
        }
         printf("\n");
        printf("e-closure of states\n");

        for(i=1;i<=nostate;i++)
        {
        for(k=0;k<20;k++)
        {
        eclos[i][k]=0;
        }
        }
        for(i=1;i<=nostate;i++)
        {
        c=0;
        eclos[i][c]=i;
        c++;
        k=i;
        for(j=0;j<notrans;j++)
                {

                d=findvalue(trans[j][0]);
                if(k==d&&trans[j][1]=='e')
                {
                  eclos[i][c]=findvalue(trans[j][2]);
                  k=eclos[i][c];
                  c++;
                }
                }

}
        for(i=1;i<=nostate;i++)
        {
        printf("e-closure of q%d -{",i);
        for(j=0;eclos[i][j]!=0;j++)
        {
        printf("q%d,",eclos[i][j]);
         }
         printf("}\n");
  }

  }
```

OUTPUT

```
administrator@administrator-Vostro-3800:~/cdlab$ gcc etrans.c
administrator@administrator-Vostro-3800:~/cdlab$ ./a.out
Enter the number of alphabets?
3
NOTE:- [ use letter e as epsilon]
NOTE:- [e must be last character ,if it is present]

Enter alphabets?
0
1
e

Enter the number of states?
3

Enter no of transition?
5

Enter transition?
1 0 1
1 e 2
2 1 2
2 e 3
3 0 3

e-closure of states
e-closure of q1 -{q1,q2,q3,}
e-closure of q2 -{q2,q3,}
e-closure of q3 -{q3,}
```

Exp7:

DESIGN AND IMPLEMENT A RECURSIVE DESCENT PARSER FOR A GIVEN GRAMMAR.

AIM:

To implement a recursive descent parser for a given grammar.

ALGORITHM:

This algorithm using Recursive procedures to implement the following Grammar.

$E \rightarrow TE'$

$E' \rightarrow +TE'$

$T \rightarrow FT'$

$T' \rightarrow *\ FT'|\varepsilon$

$F \rightarrow (E)|id$

Solution

```
Procedure E ( )                          ┐
{                                        │
        T ( );                           │  E → TE'
        E'( );                           │
}                                        ┘
Procedure E'( )                          ┐
{                                        │
        If input symbol ='+' then        │  E → + TE'
                                         │
                                         │
                                         │
                                         │
                                         │
        advance( );                      │
        T ( );                           │
        E'( );                           ┘
}
```

```
Procedure T( )
{
        F ( );
        T′( );
}
```

$T \rightarrow F\,T'$

```
Procedure T′( )
{
        If input symbol ='*' then
        advance( );
        F ( );
        T′( );
}
```

$T' \rightarrow * \; FT'$

```
Procedure F ( )
{
        If input symbol ='id' then
        advance ( );
        else if input-symbol ='(' then
        advance ( );
        E ( );
        If input-symbol = ')'
        advance ( );
        else error ( );
        else error ( );
}
```

$F \rightarrow id$

$F \rightarrow (E)$

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();
void  main()
 {
  printf("Recursive Descend parser for arithmetic expn containing + and *(eg:a+a*a)\n");
 printf("Enter an arithmetic expression : ");
 scanf("%s",input);
 E();
 if(strlen(input)==i&&error==0)
printf("\nAccepted..!!!");
else
printf("\nRejected..!!!");

 }
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]=='a')i++;
 else if(input[i]=='(')
 {
 i++;
 E();
 if(input[i]==')')
 i++;
 else
 error=1;
 }
 else
error=1;
 }
```

OUTPUT

```
administrator@administrator-Vostro-3800:~/cdlab$ gcc recursive.c
administrator@administrator-Vostro-3800:~/cdlab$ ./a.out
Recursive Descend parser for arithmetic expn containing + and *(eg:a+a*a)
Enter an arithmetic expression : a+a*a

Accepted..!!!
administrator@administrator-Vostro-3800:~/cdlab$ ./a.out
Recursive Descend parser for arithmetic expn containing + and *(eg:a+a*a)
Enter an arithmetic expression : (a+a*a

Rejected..!!!
administrator@administrator-Vostro-3800:~/cdlab$ ./a.out
Recursive Descend parser for arithmetic expn containing + and *(eg:a+a*a)
Enter an arithmetic expression : a-a+a

Rejected..!!!
administrator@administrator-Vostro-3800:~/cdlab$ ./a.out
Recursive Descend parser for arithmetic expn containing + and *(eg:a+a*a)
Enter an arithmetic expression : (a+a)*a

Accepted..!!!
administrator@administrator-Vostro-3800:~/cdlab$ ./a.out
Recursive Descend parser for arithmetic expn containing + and *(eg:a+a*a)
Enter an arithmetic expression : a+a)*a

Rejected..!!!
administrator@administrator-Vostro-3800:~/cdlab$ ./a.out
Recursive Descend parser for arithmetic expn containing + and *(eg:a+a*a)
Enter an arithmetic expression : (b*a)

Rejected..!!!
```

Exp 8:
CONSTRUCT A SHIFT REDUCE PARSER FOR A GIVEN LANGUAGE.

AIM:

To construct a Operator precedence Parser for a given language.

ALGORITHM:

The input string is w$, the initial stack is $ and a table holds precedence relations between certain terminals

Algorithm:

```
set p to point to the first symbol of w$ ;
repeat forever
  if ( $ is on top of the stack and p points to $ ) then return
  else {
     let a be the topmost terminal symbol on the stack and let b be the symbol pointed to
by p;
     if ( a <· b or a =· b ) then {          /* SHIFT */
        push b onto the stack;
        advance p to the next input symbol;
     }
     else if ( a ·> b ) then          /* REDUCE */
        repeat pop stack
        until ( the top of stack terminal is related by <· to the terminal most
        recently popped );
     else error();
  }
```

PROGRAM

```
# include<stdio.h>
              /* i   +   *   $ */
char table[4][4]={'=','>','>','>',
                  '<','>','<','>',
                  '<','>','>','>',
                  '<','<','<','='};
char op[4]={'i','+','*','$'};
char stack[50],s[50];
int top=-1,i;
void push(char c)
{
stack[++top]=c;
}
char pop()
{
return(stack[top--]);
}
int findpos(char ch)
{
int k;
for(k=0;k<4;k++)
{
```

```
if(ch==op[k])
return(k);
}
return(k);
}
void displaystack()
{
int j=0;
while(j<=top)
{
printf("%c",stack[j]);
j++;
}
printf("\t\t");
}
void displaystring()
{
int j=i;
while(s[j]!='\0')
{
printf("%c",s[j]);
j++;
}
printf("\t\t");
}
void parse()
{
char ch,ch1,opr;
int pos1,pos2;
printf("Enter the expression for operator precedence parsing(end with $) :");
scanf("%s",s);
push('$');
i=0;
while(1)
{
displaystack();
displaystring();
if(stack[top]=='$'&&s[i]=='$')
{
printf("\nValid Expression\n");
return;
}
else
{
ch=s[i];
pos2=findpos(ch);
pos1=findpos(stack[top]);
if(pos1==4||pos2==4)
{
printf("\nInvalid expression\n");
return;
}
opr=table[pos1][pos2];
```

```
if(opr=='<'||opr=='=')
{
push(ch);
i++;
printf("Shift\n");
}
else if(opr=='>')
{
ch1=pop();
pos2=findpos(ch1);
pos1=findpos(stack[top]);
while(table[pos1][pos2]!='<')
{
printf("%c ",ch1);
pos2=findpos(pop());
pos1=findpos(stack[top]);
}
printf("%c ",ch1);
printf("Reduce\n");
}
else
{
printf("\nInvalid expression\n");
return;
}
}
}
}
void main()
{
parse();
}
```

OUTPUT

```
administrator@MBC-L-127:~$ gcc oprparser.c
administrator@MBC-L-127:~$ ./a.out
Enter the expression for operator precedence parsing(end with $) :i+i*i$
$          i+i*i$      Shift
$i         +i*i$       i Reduce
$          +i*i$       Shift
$+         i*i$        Shift
$+i        *i$         i Reduce
$+         *i$         Shift
$+*        i$          Shift
$+*i       $           i Reduce
$+*        $           * Reduce
$+         $           + Reduce
$          $
Valid Expression
```

```
administrator@MBC-L-127:~$ ./a.out
Enter the expression for operator precedence parsing(end with $) :i-i
$          i-i        Shift
$i         -i
Invalid expression
administrator@MBC-L-127:~$ ./a.out
Enter the expression for operator precedence parsing(end with $) :i+i*a
$          i+i*a      Shift
$i         +i*a       i Reduce
$          +i*a       Shift
$+         i*a        Shift
$+i        *a         i Reduce
$+         *a         Shift
$+*        a
Invalid expression
```

Exp 9:

SIMULATION OF CODE OPTIMIZATION TECHNIQUES

AIM:

To perform the simulation of code optimization techniques(constant folding, Dead code elimination , common expression elimination)

ALGORITHM

1. Start
2. Read the no of arithmetic expression
3. Read the Left Hand Side(LHS) and Right Hand Side(RHS) of each expression into two structure variables(l&r)
4. Print the entered code
5. Read the RHS(structure variable r) of each each expression

   If the expression contains two constant values separated by an operator(+,-,*,/)

   Separate constant values and perform corresponding operation and store the result in structure variable r.(constant folding)
6. Print the code after constant folding and this is the input for next elimination
7. Read the LHS (structure variable l) of each expression

   If the LHS is used in RHS of other expressions, save it for next elimination

   Else discard that expression
8. Print the code after dead code elimination and this is the input for next common expression elimination
9. Read the RHS of each expression

   If RHS is repeating in other expression eliminate this

   Else save it
10. Print the code after common expression elimination
11. Stop

PROGRAM

```
#include<stdio.h>
#include<ctype.h>
#include<string.h>
#include<stdlib.h>
struct op
{
char l;
char r[20];
}op[10],pr[10];
void main()
{
int a1,b1,res;
int a,i,k,j,n,z=0,m,q,flag=0;
char opr,aa[5],bb[5],ss[5];
char *p,*l;
char temp,t;
char *tem;
printf("enter no of values");
scanf("%d",&n);
for(i=0;i<n;i++)
{
printf("left\t");
scanf(" %c",&op[i].l);
printf("right:\t");
scanf("%s",op[i].r);
}
printf("intermediate Code\n") ;
for(i=0;i<n;i++)
{
printf("%c=",op[i].l);
printf("%s\n",op[i].r);
}
for(i=0;i<n;i++)
{
k=0;
```

```
for(j=0;op[i].r[j]!='\0';j++)
{
while(isdigit(op[i].r[j])&&op[i].r[j]!='\0')
{
flag=1;
aa[k++]=op[i].r[j];
j++;
}
if((op[i].r[j]=='+'||op[i].r[j]=='-'||op[i].r[j]=='*'||op[i].r[j]=='/')&&flag==1)
{
opr=op[i].r[j];
aa[k]='\0';
j++;
}
k=0;
flag=0;
while(isdigit(op[i].r[j])&&op[i].r[j]!='\0')
{
flag=1;
bb[k++]=op[i].r[j];
j++;
}
}
if(flag==1)
{
bb[k]='\0';
a1=atoi(aa);
b1=atoi(bb);
if(opr=='+')
res=a1+b1;
if(opr=='-')
res=a1-b1;
if(opr=='*')
res=a1*b1;
if(opr=='/')
res=a1/b1;
sprintf(ss,"%d",res);
strcpy(op[i].r,ss);
```

```
 }
}
printf("intermediate Code after costant folding\n");
for(i=0;i<n;i++)
{
printf("%c=",op[i].l);
printf("%s\n",op[i].r);
}
for(i=0;i<n-1;i++)
{
temp=op[i].l;
for(j=0;j<n;j++)
{
p=strchr(op[j].r,temp);
if(p)
{
pr[z].l=op[i].l;
strcpy(pr[z].r,op[i].r);
z++ ;
break;
}} }
pr[z].l=op[n-1].l;
strcpy(pr[z].r,op[n-1].r);
z++;
printf("\nafter dead code elimination\n");
for(k=0;k<z;k++)
{

printf("%c\t=",pr[k].l);
printf("%s\n",pr[k].r);
}
//sub expression elimination
for(m=0;m<z;m++)
{
tem=pr[m].r;
for(j=m+1;j<z;j++)
{
p=strstr(tem,pr[j].r);
```

```
if(p)
{
t=pr[j].l;
pr[j].l=pr[m].l     ;
for(i=0;i<z;i++)
{
l=strchr(pr[i].r,t) ;
if(l)
{
a=l-pr[i].r;
//printf("pos: %d",a);
pr[i].r[a]=pr[m].l;
}}}}}
printf("eliminate common expression\n");
for(i=0;i<z;i++)
{
printf("%c\t=",pr[i].l);
printf("%s\n",pr[i].r);
}
// duplicate production elimination
for(i=0;i<z;i++)
{
for(j=i+1;j<z;j++)
{
q=strcmp(pr[i].r,pr[j].r);
if((pr[i].l==pr[j].l)&&!q)
{
    pr[i].l='\0';
    strcpy(pr[i].r,"\0");
 }}}
printf("optimized code\n");
for(i=0;i<z;i++)
{
if(pr[i].l!='\0')
{
printf("%c=",pr[i].l);
printf("%s\n",pr[i].r);
}}}
```

OUTPUT

```
administrator@MBC-L-127:~$ gcc codeopt.c
administrator@MBC-L-127:~$ ./a.out
enter no of values6
left    a
right:  2
left    b
right:  5+7
left    c
right:  b+d
left    e
right:  b+d
left    f
right:  c+e
left    r
right:  f
intermediate Code
a=2
b=5+7
c=b+d
e=b+d
f=c+e
r=f
intermediate Code after costant folding
a=2
b=12
c=b+d
e=b+d
f=c+e
r=f

after dead code elimination
b       =12
c       =b+d
e       =b+d
f       =c+e
r       =f
```

eliminate common expression

```
b    =12
c    =b+d
c    =b+d
f    =c+c
r    =f
```

optimized code

```
b=12
c=b+d
f=c+c
r=f
```

Exp 10:

IMPLEMENT INTERMEDIATE CODE GENERATION FOR SIMPLE EXPRESSIONS.

AIM:

To implement Intermediate code generation for simple expressions.

ALGORITHM

1. Start the program
2. Input the arithmetic expression(infix form)
3. Insert '(' to front position and ')' to the end position of infix expression
4. Scan the Infix string from left to right.
5. Initialise an empty stack.
6. If the scannned character is an operand, add it to the Postfix string.
7. If the scanned character is an Operator Then
   a. If topStack has higher precedence over the scanned character Then
      - Repeatedly Pop all the operators from the stack which are greater than or equal to in precedence than that of the scanned operator
      - Push the scanned character to stack.
   b. Else
      Push the scanned character to stack.
8. If the scanned character is an '(' Then
   push it to the stack.
9. If the scanned character is an ')' Then
   Repeatedly pop the stack and add to postfix until a '(' encountered, and discard both the parenthesis.
10. Repeat steps 6-9 till all the characters are scanned.
11. Repeat the step 12 to 13 until the NULL character is found.
12. Scan the postfix expression from left to right
13. If the scanned character is an operand Then
    push it into the stack
14. If the scanned character is an operator Then
    a) Display add, su, mul div according to the operator
    b) pop the stack to variable1
    c) pop the stack to variable2
    d) display variable2,variable1, stepvalue(stepvalue denote each step in the arithmetic expression evaluatiomn)
    e) increment the stepvalue
15. Stop

PROGRAM

```
#include<stdio.h>
#include<stdlib.h>
#include<ctype.h>
#include<string.h>
void push(char);
char pop();
void code_gen();
void in_to_pos();
int get_pre(char);
int top=0,no=1;
char stack[30],inf[50],post[50],str[10];
void main()
{
char ch;
int i=0;
inf[i++]='(';
printf("\n Enter the arithmetic expression(infix)\n");
scanf("%c",&ch);
while(ch!='\n')
{
inf[i++]=ch;
scanf("%c",&ch);
}
inf[i++]=')';
inf[i++]='\0';
in_to_pos();
printf("Postfix form: ");
puts(post);
code_gen();
}
void in_to_pos()
{
char next;
```

```
int i=0,p=0,precedence;
while(inf[i]!='\0')
{
switch(inf[i])
{
case '(':push(inf[i]);
break;
case ')':while((next=pop())!='(')
post[p++]=next;
break;
case '+':
case '-':
case '*':
case '/':
case '^':
precedence=get_pre(inf[i]);
while(precedence<=get_pre(stack[top]))
post[p++]=pop();
push(inf[i]);
break;
default:post[p++]=inf[i];
}
i++;
}
}
void code_gen()
{
int i=0,top;
char temp_op1,temp_op2;
while(post[i]!='\0')
{
if((post[i]=='+')||(post[i]=='-')||(post[i]=='*')||(post[i]=='/')||(post[i]=='^'))
{
if(post[i]=='*')
printf("mul(");
else if(post[i]=='^')
printf("pow(");
```

```
if(post[i]=='+')
printf("add(");
else if(post[i]=='-')
printf("sub(");
else if(post[i]=='/')
printf("div(");
temp_op1=pop();
temp_op2=pop();
printf("%c,%c)->%d\n",temp_op2,temp_op1,no);
sprintf(str,"%d",no);
push(str[0]);
no++;
}
else
push(post[i]);
i++;
}
}
int get_pre(char chr)
{
  switch(chr)
  {
    case '(':return 0;
    case '+':
    case '-':return 1;
    case '*':
    case '/':return 2;
    case '^':return 3;
    default :return (999);
  }
}
void push(char sym)
{
  stack[++top]=sym;
}
char pop()
{
```

```
  return (stack[top--]);
}
```

OUTPUT

```
administrator@administrator-Vostro-3800:~/cdlab$ gcc ic.c
administrator@administrator-Vostro-3800:~/cdlab$ ./a.out
 Enter the arithmetic expression(infix)
a+b-c*d/e
Postfix form: ab+cd*e/-
add(a,b)->1
mul(c,d)->2
div(2,e)->3
sub(1,3)->4
administrator@administrator-Vostro-3800:~/cdlab$ ./a.out
 Enter the arithmetic expression(infix)
(a-b)+c*d/e+f
Postfix form: ab-cd*e/+f+
sub(a,b)->1
mul(c,d)->2
div(2,e)->3
add(1,3)->4
add(4,f)→5
```

Exp 11:

PROGRAM TO CONVERT NFA TO DFA

AIM:

To convert NFA to DFA

ALGORITHM

Start

Read no of alphabets and alphabets for the transition table

Read no of states,start state and final states

Create a structure **node** that contain two members(an integer value and pointer to next node)

Create another structure **node1** that contain only one integer array as member

Declare a structure variable for **node** ,two dimensional array **transition[][]** and all values set to NULL.

Declare three structure variable **newstate** ,**tempstate and hash** for **node1** and values set to 0

Read the transitions

Store these transitions in a structure

a) Create a **node** variable temp
b) Destination of transition is stored in integer member.
c) Link part is set to transition[source][alpahabet].
d) Transition[source][alphabet] is set to temp

PROGRAM

```
#include<stdio.h>
#include<stdlib.h>
struct node
{
int st;
struct node *link;
};
struct node1
{
int nst[20];
};
void insert(int ,char, int);
int findalpha(char);
void findfinalstate(void);
```

```
int insertdfastate(struct node1);
int compare(struct node1,struct node1);
void printnewstate(struct node1);
static int set[20],nostate,noalpha,s,notransition,nofinal,start,finalstate[20],r,c,buffer[20];
int complete=-1;
char alphabet[20],ch;
static int eclosure[20][20]={0};
struct node1 hash[20];
struct node * transition[20][20]={NULL};
void main()
{
int i,j,k,m,t,n,l;
struct node *temp;
struct node1 newstate={0},tmpstate={0};
printf("Enter the number of alphabets?\n");
printf("NOTE:- [ use letter e as epsilon]\n");
printf("NOTE:- [e must be last character ,if it is present]\n");
printf("\nEnter alphabets?\n");
scanf("%d",&noalpha);
getchar();
for(i=0;i<noalpha;i++)
{
alphabet[i]=getchar();
getchar(); }
printf("Enter the number of states?\n");
scanf("%d",&nostate);
printf("Enter the start state?\n");
scanf("%d",&start);
printf("Enter the number of final states?\n");
scanf("%d",&nofinal);
printf("Enter the final states?\n");
for(i=0;i<nofinal;i++)
scanf("%d",&finalstate[i]);
printf("Enter no of transition?\n");
scanf("%d",&notransition);
printf("NOTE:- [Transition is in the format qno alphabet qno %d\n",notransition);
printf("NOTE:- [States number must be greater than zero]\n");
printf("\nEnter transition?\n");
for(i=0;i<notransition;i++)
```

```
{
scanf("%d %c%d",&r,&ch,&s);
insert(r,ch,s);
}
for(i=0;i<20;i++)
{
for(j=0;j<20;j++)
hash[i].nst[j]=0;
}
complete=-1;
i=-1;
printf("\nEquivalent DFA.....\n");
printf(".....\n");
printf("Trnsitions of DFA\n");
newstate.nst[start]=start;
insertdfastate(newstate);
while(i!=complete)
{
i++;
newstate=hash[i];
for(k=0;k<noalpha;k++)
{
c=0;
for(j=1;j<=nostate;j++)
set[j]=0;
for(j=1;j<=nostate;j++)
{
l=newstate.nst[j];
if(l!=0)
{
temp=transition[l][k];
while(temp!=NULL)
{
if(set[temp->st]==0)
{
c++;
set[temp->st]=temp->st;
}
temp=temp->link;
```

```
}}}
printf("\n");
if(c!=0)
{
for(m=1;m<=nostate;m++)
tmpstate.nst[m]=set[m];
insertdfastate(tmpstate);
printnewstate(newstate);
printf("%c\t",alphabet[k]);
printnewstate(tmpstate);
printf("\n");
}
else
{
printnewstate(newstate);
printf("%c\t", alphabet[k]);
printf("NULL\n");
}} }
printf("\nStates of DFA:\n");
for(i=0;i<complete;i++)
printnewstate(hash[i]);
printf("\n Alphabets:\n");
for(i=0;i<noalpha;i++)
printf("%c\t",alphabet[i]);
printf("\n Start State:\n");
printf("q%d",start);
printf("\nFinal states:\n");
findfinalstate();
}
int insertdfastate(struct node1 newstate)
{
int i;
for(i=0;i<=complete;i++)
{
if(compare(hash[i],newstate))
return 0;
}
complete++;
hash[complete]=newstate;
```

```
return 1;
}
int compare(struct node1 a,struct node1 b)
{
int i;
for(i=1;i<=nostate;i++)
{
if(a.nst[i]!=b.nst[i])
return 0;
}
return 1;
}
void insert(int r,char c,int s)
{
int j;
struct node *temp;
j=findalpha(c);
if(j==999)
{
printf("error\n");
exit(0);
}
temp=(struct node *) malloc(sizeof(struct node));
temp->st=s;
temp->link=transition[r][j];
transition[r][j]=temp;
}
int findalpha(char c)
{
int i;
for(i=0;i<noalpha;i++)
if(alphabet[i]==c)
return i;
return(999);
}
void findfinalstate()
{
int i,j,k,t;
for(i=0;i<=complete;i++)
```

```
{
for(j=1;j<=nostate;j++)
{
for(k=0;k<nofinal;k++)
{
if(hash[i].nst[j]==finalstate[k])
{
printnewstate(hash[i]);
printf("\t");
j=nostate;
break;
}}}}}
void printnewstate(struct node1 state)
{
int j;
printf("{");
for(j=1;j<=nostate;j++)
{
if(state.nst[j]!=0)
printf("q%d,",state.nst[j]);
}
printf("}\t");
}
```

OUTPUT

```
administrator@administrator-Vostro-3800:~/cdlab$ gcc nfatodfa.c
administrator@administrator-Vostro-3800:~/cdlab$ ./a.out
Enter the number of alphabets?
NOTE:- [ use letter e as epsilon]
NOTE:- [e must be last character ,if it is present]

Enter alphabets?
2
a
b
Enter the number of states?
4
Enter the start state?
1
```

Enter the number of final states?

2

Enter the final states?

3

4

Enter no of transition?

8

NOTE:- [Transition is in the format qno alphabet qno 8

NOTE:- [States number must be greater than zero]

Enter transition?

1 a 1

1 b 1

1 a 2

2 a 3

2 b 2

3 a 4

3 b 4

4 b 3

Equivalent DFA.....

.....

Trnsitions of DFA

{q1,} a {q1,q2,}

{q1,} b {q1,}

{q1,q2,} a {q1,q2,q3,}

{q1,q2,} b {q1,q2,}

{q1,q2,q3,} a {q1,q2,q3,q4,}

{q1,q2,q3,} b {q1,q2,q4,}

{q1,q2,q3,q4,} a {q1,q2,q3,q4,}

{q1,q2,q3,q4,} b {q1,q2,q3,q4,}

{q1,q2,q4,} a {q1,q2,q3,}

{q1,q2,q4,} b {q1,q2,q3,}

States of DFA:

{q1,} {q1,q2,} {q1,q2,q3,} {q1,q2,q3,q4,}

Alphabets:

a b

Start State:

q1

Final states:

{q1,q2,q3,} {q1,q2,q3,q4,} {q1,q2,q4,}

EXP 12

IMPLEMENT THE BACK END OF THE COMPILER.

AIM

To implement the back end of the compiler.

ALGORITHM

a) Start
b) Read the set of intermediate code in the form of quadruples and end with exit command.
c) Set i=0;
d) Repeat following steps 5 to 13 until exit command reached
e) Take the operator in each quadruple for choice
f) case '+': copy "ADD" to opr
g) case '-': copy "SUB" to opr
h) case '*': copy "MUL" to opr
i) case '/': copy "DIV" to opr
j) Print MOV operand3,Ri
k) Print opr operand3, R_i
l) Print MOV R_i,operand1
m) i++;
n) Stop

PROGRAM

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
void main()
{
char icode[10][30],str[20],opr[10];
int i=0;
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);
}
```

OUTPUT

```
administrator@MBC-L-127:~$ gcc backend.c
administrator@MBC-L-127:~$ ./a.out

 Enter the set of intermediate code (terminated by exit):
a=b+c
d=a-e
f=g*a
h=d/f
exit
```

target code generation

****

```
Mov b,R0
ADD c,R0
Mov R0,a
Mov a,R1
SUB e,R1
Mov R1,d
Mov g,R2
MUL a,R2
Mov R2,f
Mov d,R3
DIV f,R3
Mov R3,h
```