MAHA BARATHI ENGINEERING COLLEGE

Department Of Computer Science and Engineering

LABORATORY MANUAL

Course Code : CS3501

Course Name : Compiler Design Laboratory

Year / Semester : III Year / V Semester

Regulations : R-2021

Academic year : 2024- ODD

CS8602	COMPILER DESIGN	LTPC
		3023

COURSE OBJECTIVES:

- **To learn the various phases of compiler.**
- **To learn the various parsing techniques.**
- **To understand intermediate code generation and run-time environment.**
- **To learn to implement front-end of the compiler.**
- **To learn to implement code generator.**

LIST OF EXPERIMENTS:

1. Develop a lexical analyzer to recognize a few patterns in C. (Ex. identifiers, constants, comments, operators etc.). Create a symbol table, while recognizing identifiers.

2. Implement a Lexical Analyzer using Lex Tool

3. Implement an Arithmetic Calculator using LEX and YACC

4. Generate three address code for a simple program using LEX and YACC.

5. Implement simple code optimization techniques (Constant folding, Strength reduction and Algebraic transformation)

6. Implement back-end of the compiler for which the three address code is given as input and the 8086 assembly language code is produced as output.

COURSE OUTCOMES:

At the end of this course, the students will be able to:

- Understand the different phases of compiler.
- Design a lexical analyzer for a sample language.
- Apply different parsing algorithms to develop the parsers for a given grammar.
- Understand syntax-directed translation and run-time environment.
- Learn to implement code optimization techniques and a simple code generator.
- Design and implement a scanner and a parser using LEX and YACC tools

PRACTICALS: 30 PERIODS

Table of Contents

Ex:no:1	Develop a lexical analyzer to recognize a few patterns in C. (Ex. identifiers, constants, comments,operators etc.).

AIM :

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

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 (SOURCE CODE):

```
#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

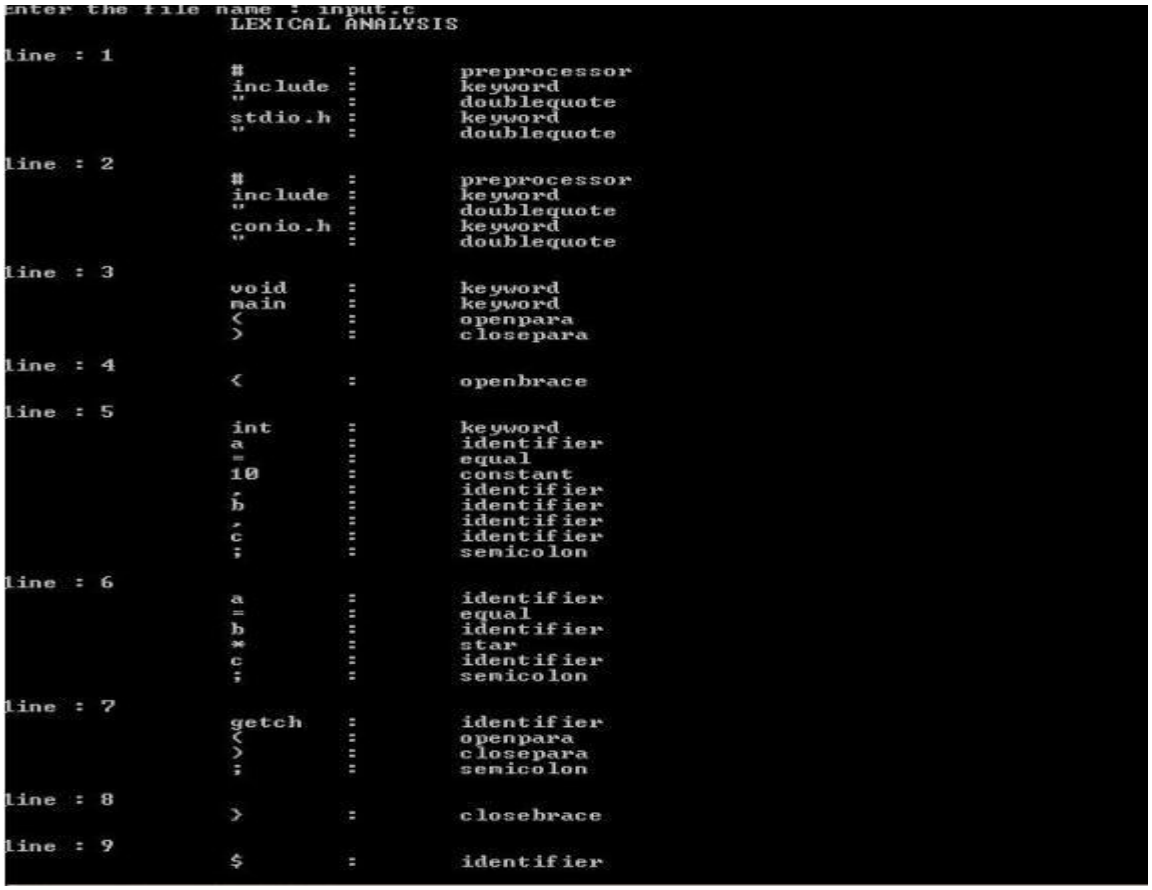
\ backslash

/ 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: 2	Create a symbol table, while recognizing identifiers.

AIM:

To write a C program to implement a 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.

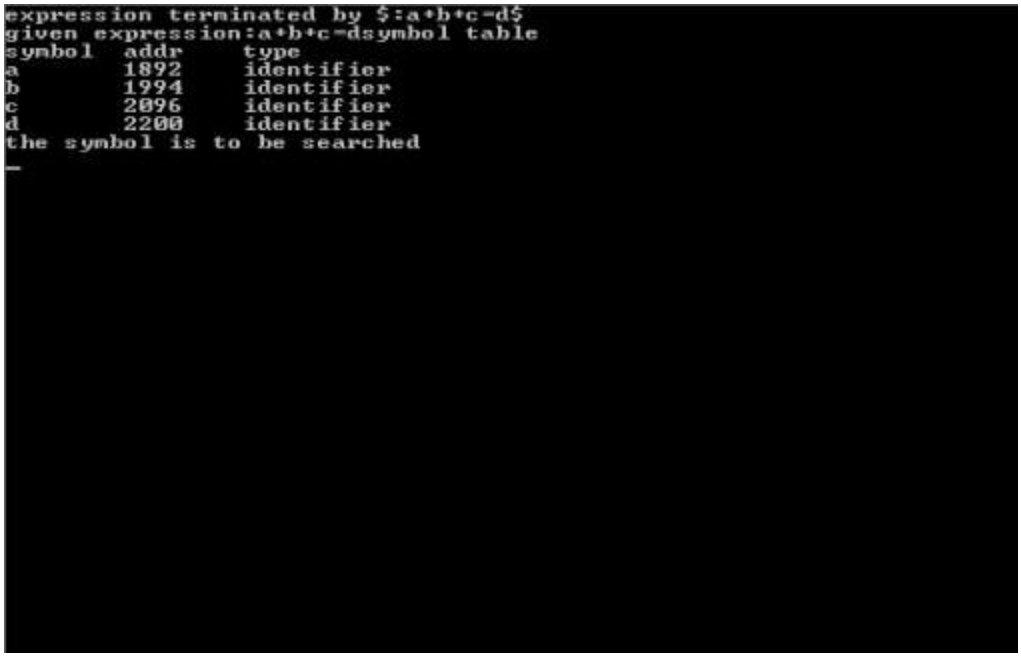
PROGRAM: (IMPLEMENTATION OF SYMBOL TABLE)

```
#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: 3	Implementation of a lexical analyzer using lex tool.

AIM:

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

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.

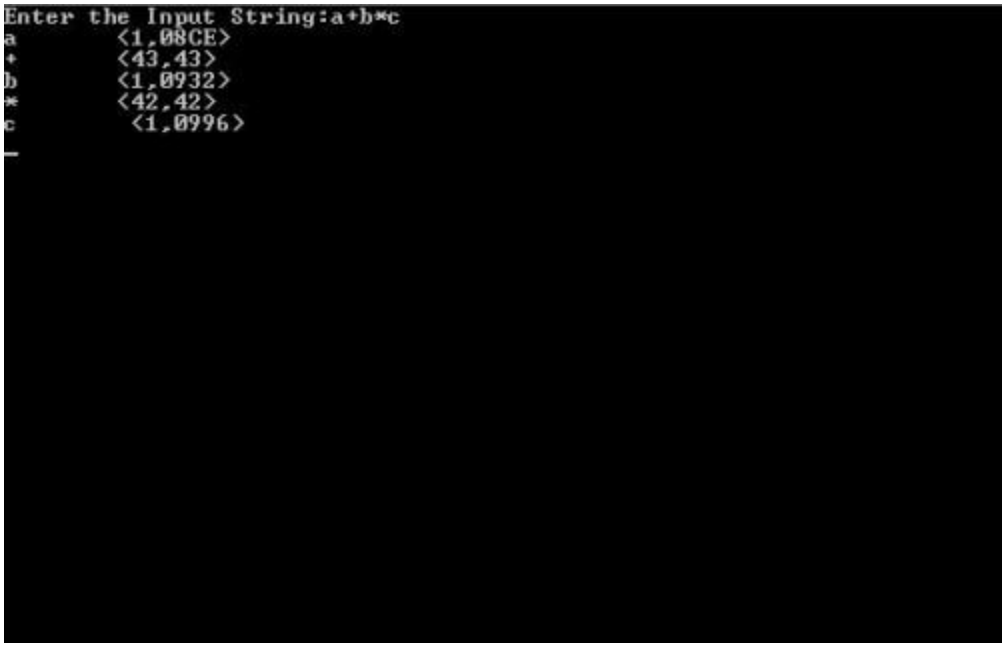
PROGRAM: (LEXICAL ANALYZER USING LEX TOOL)

```
#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: 4a	Program to recognize a valid arithmetic expression that uses operator +, -, * and /.

AIM :

To write a c program to do exercise on syntax analysis using YACC.

ALGORITHM :

1. Start the program.
2. Write the code for parser. l in the declaration port.
3. Write the code for the ‘y’ parser.
4. Also write the code for different arithmetical operations.
5. Write additional code to print the result of computation.
6. Execute and verify it.
7. Stop the program.

PROGRAM:

```
#include<stdio.h>
#include<conio.h>
void main()
{ char s[5];
clrscr();
printf("\n Enter any operator:");
gets(s);
switch(s[0])
{
case'>': if(s[1]=='=')
printf("\n Greater than or equal");
else
printf("\n Greater than");
break;
```

```
case'<': if(s[1]=='=')
printf("\n Less than or equal");
else
printf("\nLess than");
break;
case'=': if(s[1]=='=')
printf("\nEqual to");
else
printf("\nAssignment");
break;
case'!': if(s[1]=='=')
printf("\nNot Equal");
else
printf("\n Bit Not");
break;
case'&': if(s[1]=='&')
printf("\nLogical AND");
else
printf("\n Bitwise AND");
break;
case'|': if(s[1]=='|')
printf("\nLogical OR");
else
printf("\nBitwise OR");
break;
case'+': printf("\n Addition");
break;
case'-': printf("\nSubstraction");
break;
case'*': printf("\nMultiplication");
break;
case'/': printf("\nDivision");
break;
case'%': printf("Modulus");
break;
default: printf("\n Not a operator");
}
getch();
}
```

OUTPUT:

RESULT:

Thus the program for the exercise on the syntax using YACC has been executed successfully and Output is verified.

Ex:NO:4b	**Program to recognize a valid variable which starts with a letter followed by any number of letter or digits.**

AIM:

To write a Program to recognize a valid variable which starts with a letter followed by any number of letter or digits.

PROGRAM :

variable_test.l

```
%{
/* This LEX program returns the tokens for the Expression */
#include "y.tab.h"
%}
%%
"int " {return INT;}
"float" {return FLOAT;}
"double" {return DOUBLE;}
[a-zA-Z]*[0-9]*{
printf("\nIdentifier is %s",yytext);
return ID;
}
return yytext[0];
\n return 0;
int yywrap()
{
return 1;
}
```

variable_test.y

```
%{
#include
/* This YACC program is for recognising the Expression*/
%}
%token ID INT FLOAT DOUBLE
%%
D;T L
;
L:L,ID
|ID
;
```

```
T:INT
|FLOAT
|DOUBLE
;
%%
extern FILE *yyin;
main()
{
do
{
yyparse();
}while(!feof(yyin));
}
yyerror(char*s)
{
}
```

OUTPUT:

RESULT:

Thus the program for the exercise on the syntax using YACC has been executed successfully and Output is verified.

Ex:No:4c	**Implementation of calculator using lex and yacc.**

AIM:

To write a program of calculator using lex and yacc**.**

PROGRAM:

```
%{
#include<stdio.h>
int op=0,i;
float a,b;
%}
dig[0-9]+|([0-9]*)"."([0-9]+)
add "+"
sub "-"
mul"*"
div "/"
pow "^"
ln \n
%%
{dig}{digi();}
{add}{op=1;}
{sub}{op=2;}
{mul}{op=3;}
{div}{op=4;}
{pow}{op=5;}
{ln}{printf("\n the result:%f\n\n",a);}
%%
digi()
{
if(op==0)
a=atof(yytext);
else
{
b=atof(yytext);
switch(op)
{
case 1:a=a+b;
break;
case 2:a=a-b;
break;
case 3:a=a*b;
break;
case 4:a=a/b;
break;
```

```
case 5:for(i=a;b>1;b--)
a=a*i;
break;
}
op=0;
}
}
main(int argv,char *argc[])
{
yylex();
}
yywrap()
{
return 1;
}
```

OUTPUT:

```
Lex cal.l
Cc lex.yy.c-ll
a.out
4*8
The result=32
```

RESULT:

Thus the program for the exercise on the syntax using YACC has been executed Successfully and Output is verified.

Ex:No:5a	Generate YACC specification for a few syntactic categories.(Program to recognize a valid arithmetic expression that uses operator +, - , * and /.)

AIM:

To implement a program to recognize a valid arithmetic expression that uses operator +, - , * and /.

PROGRAM:

Program name:arith_id.l

```
%{
/* This LEX program returns the tokens for the expression */
#include "y.tab.h"
%}

%%
"=" {printf("\n Operator is EQUAL");}
"+" {printf("\n Operator is PLUS");}
"-" {printf("\n Operator is MINUS");}
"/" {printf("\n Operator is DIVISION");}
"*" {printf("\n Operator is MULTIPLICATION");}

[a-z A-Z]*[0-9]* {
printf("\n Identifier is %s",yytext);
return ID;
}
return yytext[0];
\n return 0;
%%

int yywrap()
{
return 1;
}
```

Program Name : arith_id.y

```
%{
#include
/* This YYAC program is for recognizing the Expression */
%}
%%
```

```
statement: A'='E
| E {
printf("\n Valid arithmetic expression");
$$ = $1;
};

E: E'+'ID
| E'-'ID
| E'*'ID
| E'/'ID
| ID
;
%%
extern FILE *yyin;
main()
{
do
{
yyparse();
}while(!feof(yyin));
}
yyerror(char*s)
{
}
```

OUTPUT:

```
[root@localhost]# lex arith_id.1
[root@localhost]# yacc –d arith_id.y
[root@localhost]# gcc lex.yy.c y.tab.c
[root@localhost]# ./a.out
x=a+b;

Identifier is x
Operator is EQUAL
Identifier is a
Operator is PLUS
Identifier is b
```

RESULT:

Thus the program of YACC specification have implemented successfully .

Ex:No:5b	**Program to recognise a valid variable which starts with a letter followed by any number of letters or digits.**

PROGRAM:

Program name: variable_test.l

```
%{
/* This LEX program returns the tokens for the Expression */
#include "y.tab.h"
%}
%%
"int " {return INT;}
"float" {return FLOAT;}
"double" {return DOUBLE;}
[a-zA-Z]*[0-9]*{
printf("\nIdentifier is %s",yytext);
return ID;
}
return yytext[0];
\n return 0;
int yywrap()
{
return 1;
}
```

Program name: variable_test.y

```
%{
#include
/* This YACC program is for recognising the Expression*/
%}
%token ID INT FLOAT DOUBLE
%%
D;T L
;
L:L,ID
|ID
;
T:INT
|FLOAT
|DOUBLE
;
%%
```

```
extern FILE *yyin;
main()
{
do
{
yyparse();
}while(!feof(yyin));
}
yyerror(char*s)
{
}
```

OUTPUT:

```
[root@localhost]# lex variable_test.I
[root@localhost]# yacc –d variable_test.y
[root@localhost]# gcc lex.yy.c y.tab.c
[root@localhost]# ./a.out
int a,b;

Identifier is a
Identifier is b[root@localhost]#
```

RESULT:

Thus the program to recognise a valid variable which starts with a letter followed by any number of letters or digits has executed successfully .

Ex:No:5c	Implement an Arithmetic Calculator using LEX and YACC.

AIM:

To write a program of arithmetic calculator using lex and yacc.

PROGRAM:

```
%{
#include<stdio.h>
int op=0,i;
float a,b;
%}
dig[0-9]+|([0-9]*)"."([0-9]+)
add "+"
sub "-"
mul"*"
div "/"
pow "^"
ln \n
%%
{dig}{digi();}
{add}{op=1;}
{sub}{op=2;}
{mul}{op=3;}
{div}{op=4;}
{pow}{op=5;}
{ln}{printf("\n the result:%f\n\n",a);}
%%
digi()
{
if(op==0)
a=atof(yytext);
else
{
b=atof(yytext);
switch(op)
{
case 1:a=a+b;
break;
case 2:a=a-b;
break;
case 3:a=a*b;
break;
case 4:a=a/b;
break;
```

```
case 5:for(i=a;b>1;b--)
a=a*i;
break;
}
op=0;
}
}
main(int argv,char *argc[])
{
yylex();
}
yywrap()
{
return 1;
}
```

OUTPUT:

```
Lex cal.l
Cc lex.yy.c-ll
a.out
4*8
The result=32
```

RESULT:

Thus the program for the exercise on the syntax using YACC has been executed Successfully and Output is verified.

Ex:No:5d	Generate three address code for a simple program using LEX and YACC.

PROGRAM:

threee.l

```
%{
#include
#include
#include “y.tab.h”
%}
%%
[0-9]+ {yylval.dval=atoi(yytext);return NUM;}
[t];
n return 0;
. {return yytext[0];}
%%
void yyerror(char *str)
{
printf(“n Invalid Character…”);
}
int main()
{
printf(“Enter Expression x => “);
yyparse();
return(0);
}
```

threee.y

```
%{
#include
int yylex(void);
char p=’A’-1;
%}
%union
{
char dval;
}
```

```
%token NUM
%left '+' '-'
%left '*' '/'
%nonassoc UMINUS
%type S
%type E
%%
S : E {printf(" x = %cn",$$);}
;
E : NUM {}
| E '+' E {p++;printf("n %c = %c + %c ",p,$1,$3);$$=p;}
| E '-' E {p++;printf("n %c = %c – %c ",p,$1,$3);$$=p;}
| E '*' E {p++;printf("n %c = %c * %c ",p,$1,$3);$$=p;}
| E '/' E {p++;printf("n %c = %c / %c ",p,$1,$3);$$=p;}
| '('E')' {$$=p;}
| '-' E %prec UMINUS {p++;printf("n %c = -%c ",p,$2);$$=p;}
;
%%
```

OUTPUT :

```
[a40@localhost ~]$ lex threee.l
[a40@localhost ~]$ yacc -d threee.y
[a40@localhost ~]$ cc lex.yy.c y.tab.c -ll
[a40@localhost ~]$ ./a.out

Enter Expression x => 1+2-3*3/1+4*5

A = 1+2
B = 3*3
C = B/1
D = A-C
E = 4*5
F = D+E
X = F
[a40@localhost ~]$ ./a.out
```

Enter Expression x => 1+2*(3+4)/5

A = 3+4
B = 2*A
C = B/5
D = 1+C
X = D
[a40@localhost ~]$ ./a.out

Enter Expression x => 1+2*(-3+-6/1)*3

A = -3
B = -6
C = B/1
D = A+C
E = 2*D
F = E*3
G = 1+F
X = G

RESULT:

Thus the program has executed successfully.

Ex:No:6	**Implement simple code optimization techniques (Constant folding, Strength reduction and Algebraic transformation)**

AIM:

To write a C program to implement simple code optimization technique.

ALGORITHM:

1. Start the program
2. Declare the variables and functions.
3. Enter the expressionand state it in the variable a, b, c.
4. Calculate the variables b & c with ‘temp’ and store it in f1 and f2.
5. If(f1=null && f2=null) then expression could not be optimized.
6. Print the results.
7. Stop the program.

PROGRAM: (SIMPLE CODE OPTIMIZATION TECHNIQUE)
Before:
Using for :

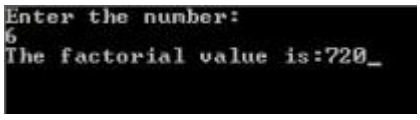
```
#include<iostream.h>
#include <conio.h>
int main()
{
int  i, n;
int fact=1;
cout<<"\nEnter a number: ";

cin>>n;
for(i=n;i>=1;i--)
fact=fact *i;
cout<<"The factoral value is: "<<fact;
getch();
return 0;
}
```

OUTPUT:

After: (SIMPLE CODE OPTIMIZATION TECHNIQUE)
Using do-while:

```
#include<iostream.h>
#include<conio.h>
void main()
{
clrscr();
int n,f;
f=1;
cout<<"Enter the number:\n";
cin>>n;
do
{
f=f*n;
n--;
}while(n>0);
cout<<"The factorial value is:"<<f;
getch();
}
```

OUTPUT:



RESULT:

Thus the Simple Code optimization technique is successfully executed

Ex:No: 7	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.

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 intermediate code has been successfully executed.