EXCEL ENGINEERING COLLEGE
(Autonomous)
KOMARAPALAYAM – 637303

DEPARTMENT OF COMPUTER SCIENCE ENGINEERING

23CS504 - COMPILER DESIGN LABORATORY

USER MANUAL

EXCEL ENGINEERING COLLEGE
(Autonomous)
KOMARAPALAYAM

Vision and Mission Statements of Institute

Vision

To create competitive human resource in the fields of engineering for the benefit of society to meet global challenges.

MISSION

- To provide a conducive ambience for better learning and to bring creativityin the students
- To develop sustainable environment for innovative learning to serve theneedy
- To meet global demands for excellence in technical education
- To train young minds with values, culture, integrity, innovation andleadership

EXCEL ENGINEERING COLLEGE
(Autonomous)
KOMARAPALAYAM

DEPARTMENT OF CSE

Vision of Department

To create better quality technical engineers in computer science and engineering with ethically strong values which cater local and global needs of the society.

Mission of Department

- **M1**: To upgrade the student skills and creativity through infrastructure development by enriching the curriculum, laboratory development in state-of-the-art technologies and industrial collaboration**.**
- **M2:** To provide quality education in Computer science and Engineering through Innovation teaching learning process.
- **M3:** To facilitate research and entrepreneurship development activities to meet global and societal needs.
- **M4:** To inculcate leadership quality with ethics and moral values.

EXCEL ENGINEERING COLLEGE
(Autonomous)
KOMARAPALAYAM

DEPARTMENT OF CSE

Programme Educational Objectives (PEOs)

- Demonstrate professional growth and success in various fields of computer science and engineering by applying their technical knowledge and skills to solve real-world problems

- Work collaboratively in interdisciplinary teams, demonstrating strong communication skills, leadership abilities and ethical responsibility in their professional field

- Engage in lifelong learning to remain Current in their profession to acquire new skills and adapt recent technological changes

Programme Outcomes

1. **Engineering knowledge**: Apply the knowledge of mathematics, science, engineering fundamentals and an engineering specialization to the solution of complex engineering problems.

2. **Problem analysis**: Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.

3. **Design/development of solutions**: Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.

4. **Conduct investigations of complex problems**: Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.

5. **Modern tool usage**: Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.

6. **The engineer and society**: Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.

7. **Environment and sustainability**: Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.

8. **Ethics:** Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.

9. **Individual and team work**: Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.

10. **Communication**: Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.

11. **Project management and finance**: Demonstrate knowledge and understanding of the engineering and management principles and apply these to one‘s own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.

12. **Life-long learning**: Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

EXCEL ENGINEERING COLLEGE
(Autonomous)
KOMARAPALAYAM

DEPARTMENT OF CSE

Programme Specific Programme Outcomes

1. **Real World Knowledge**: Apply the knowledge in solving the real world problems and cope up with the issues and challenges of Information and Computing Technologies.

2. **Technical Skill Development**: Play a vital role as individual and member of a team to develop software solutions by the application of systematic approach and appropriate models with suitable Languages, Tools, Integrated Development Environments, Patterns, Frameworks and Architecture

3. **Innovating Attitude:** Adapt to emerging Information and Communication Technologies (ICT) to innovate ideas and solutions to existing / Novel problems and build the challenging attitude in realizing them as contribution to the world

23CS504	COMPILER DESIGN LABORATORY	L	T	P	C
		0	0	2	1
Nature of Course	Professional Core				
Pre requisites	Programming in C, Operating System, Theory of Computation				

Course Objectives

The course is intended to

1. Make use of LEX tool to recognize tokens.
2. Generate a parser for the given grammar.
3. Exploit YACC tool to perform syntax analysis.
4. Implement back-end of the compiler to generate Assembly code.
5. Implement code optimization Techniques for Intermediate Code.

Course Outcomes

On successful completion of the course, students will be able to

CO. No.	Course Outcome	Bloom's Level
CO1	Implement a symbol table for managing identifiers in a compiler.	Apply
CO2	Apply LEX tool to recognize tokens in the given source program	Apply
CO3	Design a parser for the given grammar	Apply
CO4	Make use of YACC tool to perform syntax analysis	Apply
CO5	Generate Assembly code for the given three address code	Apply
CO6	Apply code optimization techniques for intermediate code	Apply

List of Exercises

S. No	List of Exercises	CO Mapping	RBT
1	Implementation of Symbol Table	CO1	Apply
2	Implementation of Lexical Analyzer to recognize a few patterns in C (Identifiers, constants, Comments, Operators, etc.)	CO2	Apply
3	Implementation of Lexical Analyzer using LEX Tool	CO2	Apply
4	Implement an Arithmetic Calculator using LEX and Yacc	CO4	Apply
5	Implementation of Shift Reduce Parsing	CO3	Apply
6	Develop programs to identify left recursions and left factors and eliminate them from the grammar given	CO3	Apply
7	Implement Operator Precedence Parsing	CO3	Apply
8	Implementation of Type Checking	CO3	Apply
9	Generate three address codes for a simple program	CO5	Apply
10	Generate the Machine Codes for the given code segment using C Language. Give the Quadruple representation of the given code segment as input for your program.M=n+k*h , n=g/k-m,p=m/n,v=m-k*p	CO5	Apply
11	Implement back-end of the compiler for which the three-address code is given as input and the assembly language code is produced as output.	CO5	Apply
12	Implementation of simple code optimization techniques	CO6	Apply

TOTAL: 45 Periods

Mapping of Course Outcomes (CO) with Programme Outcomes (PO) Programme Specific Outcomes (PSO)															
COs	Pos												PSOs		
	1	2	3	4	5	6	7	8	9	10	11	12	1	2	3
1	3	3	3	3	-	-	-	-	-	-	-	-	2	3	-
2	3	3	3	3	3	-	-	-	-	-	-	-	2	3	-
3	3	3	3	3	-	-	-	-	-	-	-	-	2	3	-
4	3	3	3	3	3	-	-	-	-	-	-	-	2	3	-
5	3	3	3	3	-	-	-	-	-	-	-	-	2	3	-
	3	High				2	Medium					1	Low		

Summative assessment based on Continuous and End Semester Examination			
Bloom's Level	**Rubric based Continuous Assessment [40 marks]**	**Preparatory Examination[20 Marks]**	**End Semester Examination [40 marks]**
Remember			
Understand			
Apply			
Analyze			
Evaluate			
Create			

LIST OF EXPERIMENTS

S. No	List of Exercises	CO Mapping	RBT
CYCLE-1			
1	Implementation of Symbol Table	CO1	Apply
2	Implementation of Lexical Analyzer to recognize a few patterns in C (Identifiers, constants, Comments, Operators, etc.)	CO2	Apply
3	Implementation of Lexical Analyzer using LEX Tool	CO2	Apply
4	Implement an Arithmetic Calculator using LEX and Yacc	CO4	Apply
5	Implementation of Shift Reduce Parsing	CO3	Apply
6	Develop programs to identify left recursions and left factors and eliminate them from the grammar given	CO3	Apply
CYCLE-2			
7	Implement Operator Precedence Parsing	CO3	Apply
8	Implementation of Type Checking	CO3	Apply
9	Generate three address codes for a simple program	CO5	Apply
10	Generate the Machine Codes for the given code segment using C Language. Give the Quadruple representation of the given code segment as input for your program. M=n+k*h , n=g/k-m,p=m/n,v=m-k*p	CO5	Apply
11	Implement back-end of the compiler for which the three-address code is given as input and the assembly language code is produced as output.	CO5	Apply
12	Implementation of simple code optimization techniques	CO6	Apply

-

<u>**INDEX**</u>

Ex. No. 1	IMPLEMENTATION OF SYMBOL TABLE

AIM

To write a C program to implement a symbol table.

ALGORITHM:

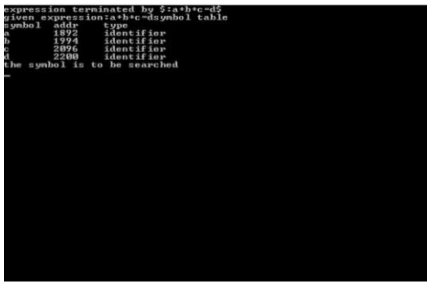
1. Start the Program.
2. Get the input from the user with the terminating symbol '$'.
3. Allocate memory for the variable by dynamic memory allocation function.
4. If the next character of the symbol is an operator then only the memory is allocated.
5. While reading , the input symbol is inserted into symbol table along with its memory address.
6. The steps are repeated till"$"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:

```
#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:



```
expression terminated by $:a+b*c-d$
given expression:a+b*c-dsymbol table
symbol  addr    type
a       1892    identifier
b       1994    identifier
c       2096    identifier
d       2200    identifier
the symbol is to be searched
_
```

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

Ex. No. 2	Develop a lexical analyzer to recognize a few patterns in C. (Ex.identifiers, constants, Comments, operators etc.). Create a symbol table, while recognizing identifiers

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
```
#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");
#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:

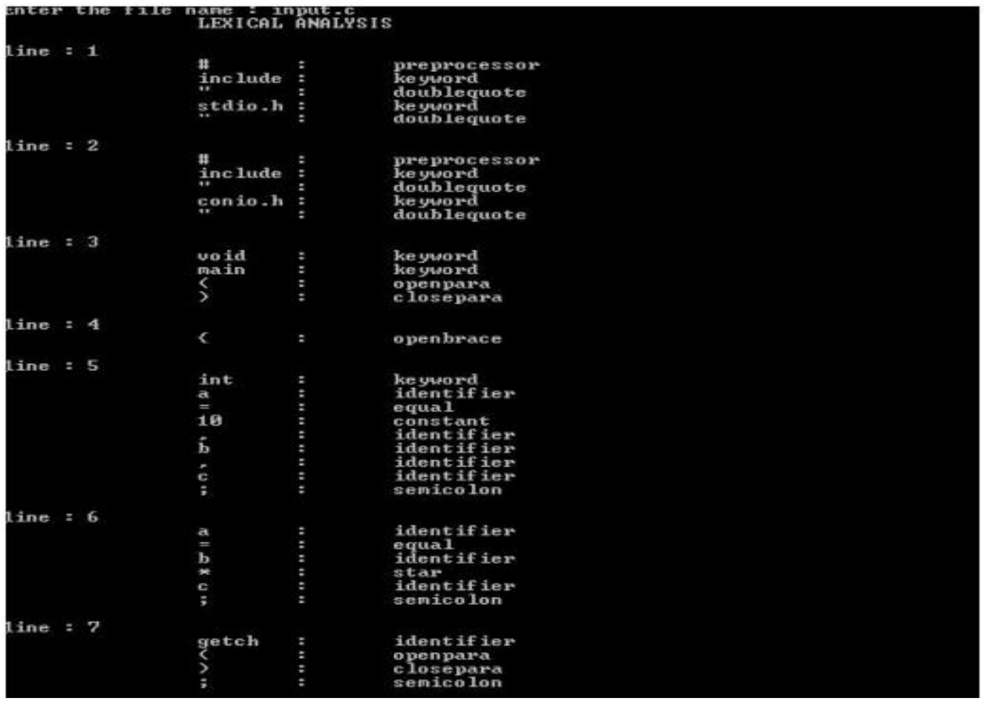
```
( open para
) closepara
{ openbrace
} closebrace
< lesser
> greater
" doublequote ' singlequote
: colon
```

; 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:



```
Enter the file name : input.c
                    LEXICAL ANALYSIS

line : 1
                    #        :       preprocessor
                    include  :       keyword
                    "        :       doublequote
                    stdio.h  :       keyword
                    "        :       doublequote

line : 2
                    #        :       preprocessor
                    include  :       keyword
                    "        :       doublequote
                    conio.h  :       keyword
                    "        :       doublequote

line : 3
                    void     :       keyword
                    main     :       keyword
                    (        :       openpara
                    )        :       closepara

line : 4
                    {        :       openbrace

line : 5
                    int      :       keyword
                    a        :       identifier
                    =        :       equal
                    10       :       constant
                    ,        :       identifier
                    b        :       identifier
                    ,        :       identifier
                    c        :       identifier
                    ;        :       semicolon

line : 6
                    a        :       identifier
                    =        :       equal
                    b        :       identifier
                    *        :       star
                    c        :       identifier
                    ;        :       semicolon

line : 7
                    getch    :       identifier
                    (        :       openpara
                    )        :       closepara
                    ;        :       semicolon
```

Result:

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

Ex. No. 3	Implementation of Lexical Analyzer using LEX Tool

AIM

To implement a **Lexical Analyzer** using the **LEX tool** that reads a source program and identifies tokens such as **keywords**, **identifiers**, **numbers**, and **operators**

ALGORITHM

1. Start the program.
2. Define regular expressions for tokens:
   - Keywords
   - Identifiers
   - Numbers
   - Operators
3. In the rules section, write actions for each regular expression.
4. Use yylex() to start lexical analysis.
5. For each matched token:
   - Print the token type and value.
6. Ignore whitespaces and newlines.
7. Stop when input ends (yywrap() returns 1).
8. End the program

PROGRAM:

```
%{
#include <stdio.h>
%}
// Definitions
KEYWORD     int|float|if|else|while|return
IDENTIFIER  [a-zA-Z_][a-zA-Z0-9_]*
NUMBER      [0-9]+(\.[0-9]+)?
OPERATOR    \+|\-|\*|\/|\=|\<|\>|\<=|\>=|\==
%%
// Rules
{KEYWORD}     { printf("Keyword: %s\n", yytext); }
{IDENTIFIER}  { printf("Identifier: %s\n", yytext); }
{NUMBER}      { printf("Number: %s\n", yytext); }
{OPERATOR}    { printf("Operator: %s\n", yytext); }
[ \t\n]+      { /* Ignore whitespace */ }
.             { printf("Unknown token: %s\n", yytext); }
%%
// Main function
int main() {
    printf("Enter the code (Ctrl+D to end input):\n");
    yylex();
    return 0;
}
// Function to signal end of input
int yywrap() {
    return 1;
}
```

Steps to Compile and Run

1. Save the above code in a file named lexer.l.
2. Open terminal and run:

```
lex lexer.l                  # Generates lex.yy.c
gcc lex.yy.c -ll -o lexer   # Compiles the code
./lexer               # Run the program
```

Sample Input (entered after running ./lexer):

```
int num = 25;
float value = 3.14;
if (num > 10) return value;
```

Sample Output:

```
Keyword: int
Identifier: num
Operator: =
Number: 25
Operator: ;
Keyword: float
Identifier: value
Operator: =
Number: 3.14
Operator: ;
Keyword: if
Operator: (
Identifier: num
Operator: >
Number: 10
Operator: )
Keyword: return
Identifier: value
Operator: ;
```

Output:

```
Keyword: int
Identifier: main
Unknown token: (
Unknown token: )
Unknown token: {
Identifier: printf
Unknown token: (
Unknown token: "
Unknown token: E
Unknown token: n
Unknown token: t
Unknown token: e
Unknown token: r
Unknown token: 
Unknown token: t
Unknown token: h
Unknown token: e
Unknown token: 
Unknown token: c
Unknown token: o
Unknown token: d
Unknown token: e
Unknown token: 
Unknown token: (
Unknown token: C
Unknown token: t
Unknown token: r
Unknown token: l
Unknown token: +
Unknown token: D
Unknown token: 
Unknown token: t
Unknown token: o
Unknown token: 
Unknown token: e
Unknown token: n
Unknown token: d
Unknown token: 
Unknown token: i
Unknown token: n
Unknown token: p
Unknown token: u
Unknown token: t
Unknown token: )
Unknown token: :
Unknown token: \
Unknown token: n
Unknown token: "
Unknown token: )
Unknown token: ;
Identifier: yylex
Unknown token: (
Unknown token: )
Unknown token: ;
Keyword: return
Number: 0
Unknown token: }
Operator: /
Operator: /
Identifier: Function
Identifier: to
Identifier: signal
Identifier: end
Identifier: of
Identifier: input
Keyword: int
Identifier: yywrap
Unknown token: (
Unknown token: )
Unknown token: {
Keyword: return
Number: 1
Unknown token: }
```

Result
The program successfully implements a Lexical Analyzer using the LEX Tool. It identifies and classifies different tokens in the input such as keywords, identifiers, numbers, and operators.

Ex. No. 4	Implement an Arithmetic Calculator using LEX and Yacc

AIM
To design and implement a simple arithmetic expression calculator using LEX and YACC that supports basic operations: addition, subtraction, multiplication, division, and parentheses

ALGORITHM

1. Lexical Analysis (LEX) Identify tokens: numbers, operators (+, -, *, /), parentheses (,), and newline (\n). Return token types to the YACC parser.
2. Syntax Analysis (YACC) Define grammar rules for arithmetic expressions. Use operator precedence and Associativity. Evaluate the expression during parsing (semantic actions).
3. Execution Compile using lex and yacc. Input arithmetic expressions. Output the computed result.
Exit on EOF (Ctrl+D or Ctrl+Z depending on OS).

PROGRAM
. calc.l – LEX Program

```
%{
#include "y.tab.h"
#include <stdio.h>
%}

%%
[0-9]+      { yylval = atoi(yytext); return NUMBER; }
[+\-*/]     { return yytext[0]; }
[\n]        { return '\n'; }
[()]        { return yytext[0]; }
[ \t]       { /* Ignore whitespace */ }
.           { printf("Invalid character: %s\n", yytext); }
%%

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

2.calc.y – YACC Program

```
%{
#include <stdio.h>
#include <stdlib.h>
%}
```

```
%token NUMBER
%left '+' '-'
%left '*' '/'
%right UMINUS

%%
input:
    /* empty */
  | input line
;

line:
    '\n'
  | expr '\n'   { printf("Result = %d\n", $1); }
;

expr:
    NUMBER              { $$ = $1; }
  | expr '+' expr       { $$ = $1 + $3; }
  | expr '-' expr       { $$ = $1 - $3; }
  | expr '*' expr       { $$ = $1 * $3; }
  | expr '/' expr       {
                          if ($3 == 0) {
                              printf("Error: Division by zero\n");
                              exit(1);
                          }
                          $$ = $1 / $3;
                        }
  | '-' expr %prec UMINUS { $$ = -$2; }
  | '(' expr ')'        { $$ = $2; }
;
%%

int main() {
    printf("Enter arithmetic expressions (Ctrl+D to exit):\n");
    yyparse();
    return 0;
}

int yyerror(const char *s) {
    fprintf(stderr, "Parse error: %s\n", s);
    return 1;
}
```

How to Compile and Run

```
yacc -d calc.y           # Generates y.tab.c and y.tab.h
lex calc.l               # Generates lex.yy.c
gcc y.tab.c lex.yy.c -o calc -ll   # Compile and link LEX/YACC
./calc                   # Run the calculator
```

OUTPUT:

Sample Input and Output

```
5 + 3
7 * (2 + 3)
10 / 2 - 1
-4 + 6
10 / 0
```

Output:

```
Result = 8
Result = 35
Result = 4
Result = 2
Error: Division by zero
```

Result:

The arithmetic calculator was successfully implemented using LEX and YACC. It correctly evaluates expressions with operator precedence, parentheses, and unary minus. Errors such as division by zero are also handled.

Ex. No. 5	Implementation of Shift Reduce Parsing

AIM

To write a C program to implement Shift Reduce Parsing.

ALGORITHM:

1. Get a string W and the grammar G as input.
2. Push the starting symbol of production, otherwise an error indication.
3. Compare the stack element with the given input string.
4. If the symbol is available in the production of its right side, then reduce it to non terminal on the left side of the production.
5. Repeat the steps 3 and 4 , until the stack and the input reach $.
6. Otherwise display it as error.
7. Stop the process.

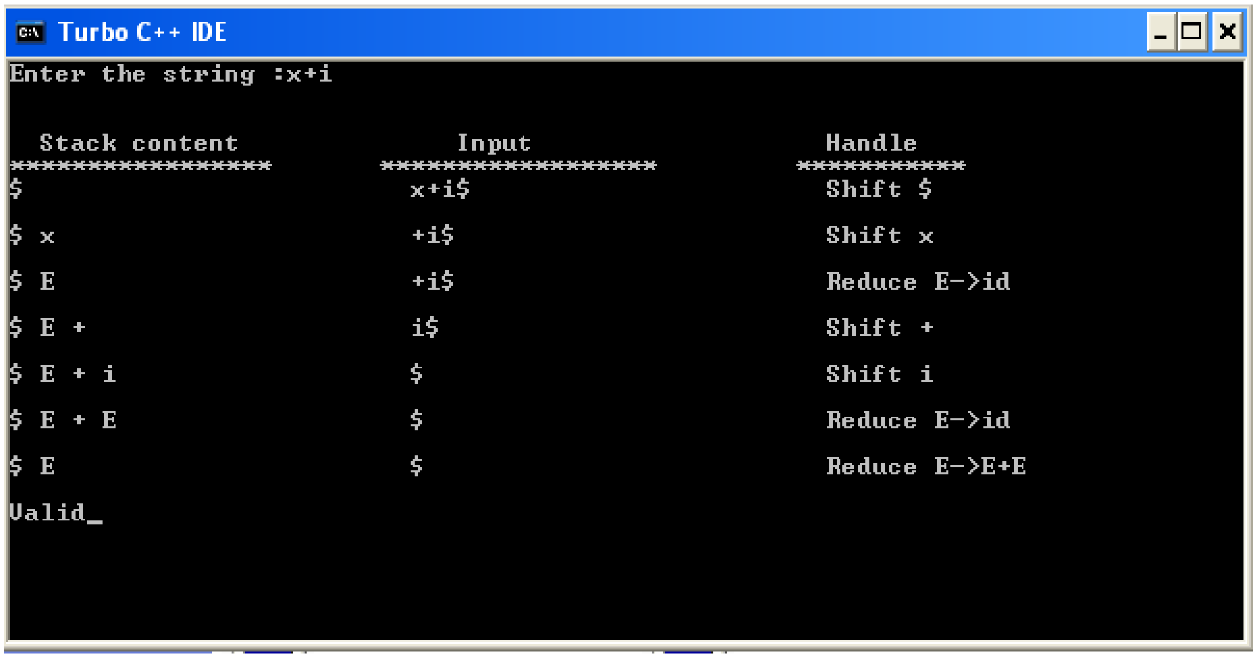
PROGRAM:

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
#include<string.h>
#define V1 27
#define V2 54

char str[4],popedchar,prod[5][4]={"E+E","E-E","E*E","E/E","id"};
static char *ptr;
int flag=0,mm=4,posx1=V1,posx2=V2,posy1=6,posy2=6;

struct stack
{
char x[15];
int top;
}s;

void settop()
{
s.top=0;
s.x[0]='$';
}

void push(char ch)      {
```

```
if(s.top<15)
   {
   s.top++;
   s.x[s.top]=ch;
   }
   }

   char pop()
   {
   if(s.top>=0)
   return s.x[s.top--];
   return '~';
   }

   void display()
   {
   int y=0;
   while(y<=s.top)
   {
   printf("%c ",s.x[y]);
   y++;
   }
   gotoxy(posx1,posy1);
   printf("%s",ptr);
   posy1+=2;

   gotoxy(posx2,posy2);

   if(flag==0)
           printf("Shift %c\n\n",s.x[s.top]);
   else
           printf("Reduce E->%s\n\n",prod[mm]);
   posy2+=2;
   }

   int comp()
   {
   int i;
   for(i=0;i<4;i++)
           if(strcmp(str,prod[i])==0)
   return i;
   return -1;
   }

   void main()
   {
   char ipstring[20];
```

```
int length;
clrscr();
settop();
printf("Enter the string :");
gets(ipstring);
ptr=ipstring;
length=strlen(ptr);
ptr[length++]='$';
ptr[length]='\0';
printf("\n\n  Stack content\t\t     Input\t\t     Handle\n");
printf("****************\t******************\t    ***********\n");
display();
push(*ptr);
ptr++;

while(*ptr!='\0')
{
        display();
        popedchar=pop();
                if(popedchar>96&&popedchar<122)
                {
                        push('E');
                        flag=1;
                        mm=4;
                        display();
                 if(s.top>=3 && (s.x[s.top]=='E'||')'))
                  {
                        str[2]=pop();
                        str[1]=pop();
                        str[0]=pop();
                        str[3]='\0';
                        mm=comp();

                  if(mm!=-1)
                   {      push('E');              flag=1;           display();          }
                  else
                   {      push(str[0]);    push(str[1]);    push(str[2]);                 }
                  }
                }
        else if(popedchar=='('||'+'||'-'||'*'||'/')
        {
          push(popedchar);

        }
        push(*ptr);
        flag=0;
        ptr++;
} //End of while loop
if(s.top==2 && s.x[s.top-1]=='E')     printf("Valid");
else     printf("Invalid expression");
getch();
}
```

OUTPUT:



Turbo C++ IDE

```
Enter the string :x+i

  Stack content              Input                  Handle
*****************        ******************          ***********
$                          x+i$                     Shift $

$ x                        +i$                      Shift x

$ E                        +i$                      Reduce E->id

$ E +                      i$                       Shift +

$ E + i                    $                        Shift i

$ E + E                    $                        Reduce E->id

$ E                        $                        Reduce E->E+E

Valid_
```

RESULT

Thus the Shift Reduce Parsing implemented and output verified successfully

Ex. No. 6	Develop programs to identify left recursions and left factors and eliminate them from the grammar given

AIM

To eliminate left recursion and left factoring to produce equivalent grammar without ambiguity or redundancy

ALGORITHM

1. Detect and Eliminate Immediate Left Recursion
    - For each non-terminal A:
       Partition productions into:
        - Left recursive: start with A
        - Non-left recursive: don't start with A
    - If left recursive productions exist:
        - Create new non-terminal A'
        - For each non-left recursive production β, output: A -> β A'
        - For each left recursive production Aα, output: A' -> α A'
        - Add A' -> ε (empty production)
2. Detect and Eliminate Left Factoring
    - For each non-terminal A:
        - Find the longest common prefix among the alternatives
    - If a common prefix exists:
        - Factor out the prefix α
        - Create new non-terminal A'
        - Replace productions with:
            - A -> α A'
            - A' -> β1 | β2

PROGRAM

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#define MAX_PROD 20
#define MAX_LEN 100
```

```
char nonTerminal;
char productions[MAX_PROD][MAX_LEN];
int prodCount = 0;

// Utility function to find common prefix of two strings
int commonPrefixLength(char *a, char *b) {
    int i = 0;
    while (a[i] && b[i] && a[i] == b[i])
        i++;
    return i;
}

// Input grammar from user
void inputGrammar() {
    printf("Enter non-terminal: ");
    scanf(" %c", &nonTerminal);
    printf("Enter number of productions for %c: ", nonTerminal);
    scanf("%d", &prodCount);

    printf("Enter productions (RHS only, space separated):\n");
    for (int i = 0; i < prodCount; i++) {
        printf("%c -> ", nonTerminal);
        scanf(" %s", productions[i]);
    }
}

// Function to eliminate immediate left recursion
void eliminateLeftRecursion() {
    char alpha[MAX_PROD][MAX_LEN];
    char beta[MAX_PROD][MAX_LEN];
    int alphaCount = 0, betaCount = 0;

    for (int i = 0; i < prodCount; i++) {
        if (productions[i][0] == nonTerminal) {
            // Left recursive production A -> A α
            strcpy(alpha[alphaCount++], productions[i] + 1); // remove leading nonTerminal
        } else {
            strcpy(beta[betaCount++], productions[i]);
        }
    }

    if (alphaCount == 0) {
        printf("\nNo left recursion detected for %c.\n", nonTerminal);
        printf("Grammar:\n");
        for (int i = 0; i < prodCount; i++) {
            printf("%c -> %s\n", nonTerminal, productions[i]);
        }
        return;
    }

    printf("\nLeft recursion detected for %c.\n", nonTerminal);
    printf("Eliminating left recursion...\n");

    printf("\nNew productions:\n");
    // A -> beta A'
    for (int i = 0; i < betaCount; i++) {
        printf("%c -> %s%c'\n", nonTerminal, beta[i], nonTerminal);
    }
    // A' -> alpha A' | ε
    printf("%c' -> ", nonTerminal);
    for (int i = 0; i < alphaCount; i++) {
        printf("%s%c' ", alpha[i], nonTerminal);
        if (i != alphaCount -1)
            printf("| ");
    }
    printf("| ε\n");
}

// Function to perform left factoring
void leftFactoring() {
    // Find longest common prefix among productions
    if (prodCount < 2) {
        printf("\nNot enough productions for left factoring.\n");
        return;
    }

    int prefixLen = strlen(productions[0]);
    for (int i = 1; i < prodCount; i++) {
        int curPrefix = commonPrefixLength(productions[0], productions[i]);
        if (curPrefix < prefixLen)
            prefixLen = curPrefix;
    }

    if (prefixLen == 0) {
        printf("\nNo common prefix found for left factoring.\n");
        return;
    }

    printf("\nCommon prefix found: ");
    for (int i = 0; i < prefixLen; i++)
        printf("%c", productions[0][i]);
    printf("\n");
    printf("\nAfter left factoring:\n");
    printf("%c -> ", nonTerminal);
    for (int i = 0; i < prefixLen; i++)
        printf("%c", productions[0][i]);
    printf("%c'\n", nonTerminal);

    printf("%c' -> ", nonTerminal);
    for (int i = 0; i < prodCount; i++) {
        if ((int)strlen(productions[i]) == prefixLen)
            printf("ε");
        else
            printf("%s", productions[i] + prefixLen);

        if (i != prodCount - 1)
            printf(" | ");
    }
    printf("\n");
}

int main() {
    inputGrammar();

    eliminateLeftRecursion();

    leftFactoring();

    return 0;
}
```

OUTPUT:

```
        Enter non-terminal: A
        Enter number of productions for A: 3
        A -> Aa
        A -> b
        A -> Ac
        Left recursion detected for A.
        Eliminating left recursion...
        New productions:
        A -> bA'
        A' -> aA' | cA' | ε
Common prefix found: A
After left factoring:
        A -> A A'
        A' -> a | c | ε
```

RESULT:

Thus the identification of left recursions and left factors and elimination them from the given grammar has been implemented and output verified successfully

Ex. No. 7	Implement Operator Precedence Parsing

AIM

To implement an **Operator Precedence Parser** in C that:

- Parses expressions based on precedence and associativity of operators.
- Determines whether the input expression is syntactically correct.
- Uses a predefined **precedence table** for parsing.

ALGORITHM

Operator Precedence Parsing Basics

- Works only for grammars that do **not contain ε-productions or left recursion**.
- Uses **precedence relations** between terminals:
  - < (shift)
  - > (reduce)
  - = (equal)
- Maintains a **stack** and **input buffer**.
- Performs **shift or reduce** actions based on the **precedence table**

Algorithm

1. Initialize **stack** with $ (bottom marker).
2. Read the **input expression** and append $ at the end.
3. Repeat until stack and input buffer reduce to E:
   - Let a = topmost terminal on stack.
   - Let b = current input symbol.
   - Use precedence table to find relation between a and b.
   - If relation is < or =, then **shift** b to stack.
   - If relation is >, then **reduce** stack top to E.
   - If no valid relation, report **syntax error**.
4. Accept the string if only E remains

PROGRAM

```
#include <stdio.h>
#include <string.h>

#define MAX 100

char stack[MAX];
```

```
int top = -1;

char precedenceTable[4][4] = {
   // +    *    i    $
   {'>', '<', '<', '>'}, // +
   {'>', '>', '<', '>'}, // *
   {'>', '>', ' ', '>'}, // i
   {'<', '<', '<', 'A'}  // $
};

int getIndex(char symbol) {
   switch(symbol) {
      case '+': return 0;
      case '*': return 1;
      case 'i': return 2;
      case '$': return 3;
      default: return -1;
   }
}

void push(char c) {
   stack[++top] = c;
}

void pop() {
   if (top >= 0) top--;
}

void printStackAndInput(char* input, int ip) {
   for (int i = 0; i <= top; i++) printf("%c", stack[i]);
   printf("\t");
   printf("%s\n", &input[ip]);
}

int main() {
   char input[MAX];
   int ip = 0;

   printf("Enter expression (use i for id): ");
   scanf("%s", input);
   strcat(input, "$");

   push('$');

   printf("\nStack\tInput\n");
   printf("-----\t-----\n");
```

```
while (1) {
    printStackAndInput(input, ip);

    char topTerminal;
    // Find topmost terminal in stack
    int i = top;
    while (i >= 0 && getIndex(stack[i]) == -1) i--;
    topTerminal = stack[i];

    char current = input[ip];

    int row = getIndex(topTerminal);
    int col = getIndex(current);

    if (row == -1 || col == -1) {
        printf("Error: Invalid symbol encountered.\n");
        break;
    }

    char relation = precedenceTable[row][col];

    if (relation == '<' || relation == '=') {
        // Shift
        push(input[ip++]);
    }
    else if (relation == '>') {
        // Reduce
        if (stack[top] == 'i') {
            stack[top] = 'E'; // Replace id with E
        } else if (stack[top] == 'E' && (stack[top-1] == '+' || stack[top-1] == '*') && stack[top-2] == 'E') {
            // Reduce E + E or E * E to E
            top -= 2;
            stack[top] = 'E';
        } else {
            printf("Error: Invalid reduction.\n");
            break;
        }
    }
    else if (relation == 'A' && stack[top] == 'E' && stack[top-1] == '$' && input[ip] == '$') {
        printf("Accepted ✅\n");
        break;
    }
    else {
        printf("Error: Invalid expression.\n");
        break;
    }
}
```

```
    return 0;
}
```

OUTPUT:

Sample Input
i+i*i

```
Output:

Enter expression (use i for id): i+i*i

Stack     Input
-----     -----
$ i+i*i$
$i        +i*i$
$E        +i*i$
$E+       i*i$
$E+i      i*i$
$E+E      *i$
$E+E*     i$
$E+E*i    $
$E+E*E    $
$E+E      $
$E        $
Accepted ✅
```

RESULT:

Thus the Operator Precedence Parsing program has been implemented and output verified successfully

Ex. No. 8	Implementation of Type Checking

AIM

To implement a **Type Checking** mechanism in C that verifies **operand types** in expressions or assignments and ensures **type compatibility** during compile-time.

ALGORITHM

1. **Input** a list of variable declarations along with their types.
2. **Store** them in a symbol table (array of structs).
3. **Input expressions or assignments**, such as a = b + c.
4. For each operation:
   - Lookup the type of operands (b, c)
   - Ensure operands have the same or compatible types.
   - Set the result variable (a) to the resulting type.
5. If types are incompatible:
   - Report a **type mismatch error**.
6. Output the result of each operation and whether the expression is type-correct.

PROGRAM

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#define MAX 100

typedef struct {
    char var[20];
    char type[10];  // int, float, char, etc.
} Symbol;

Symbol table[MAX];
int symbolCount = 0;

// Function to add variable to symbol table
void addSymbol(char *var, char *type) {
    strcpy(table[symbolCount].var, var);
    strcpy(table[symbolCount].type, type);
    symbolCount++;
}
```

```
// Function to get type of a variable
char* getType(char *var) {
    for (int i = 0; i < symbolCount; i++) {
        if (strcmp(table[i].var, var) == 0)
            return table[i].type;
    }
    return NULL; // variable not declared
}

// Check type compatibility for binary operation
int checkTypeCompatibility(char *type1, char *type2) {
    if (strcmp(type1, type2) == 0)
        return 1;
    if ((strcmp(type1, "int") == 0 && strcmp(type2, "float") == 0) ||
        (strcmp(type1, "float") == 0 && strcmp(type2, "int") == 0))
        return 1; // allow int-float compatibility
    return 0;
}

int main() {
    int n;
    char var[20], type[10];

    printf("Enter number of variable declarations: ");
    scanf("%d", &n);
    printf("Enter variable name and type (e.g. a int):\n");
    for (int i = 0; i < n; i++) {
        scanf("%s %s", var, type);
        addSymbol(var, type);
    }

    int m;
    char res[20], op1[20], op2[20], operator[5];
    printf("\nEnter number of expressions to type check: ");
    scanf("%d", &m);
    printf("Enter expressions in format: result = operand1 operator operand2\n");

    for (int i = 0; i < m; i++) {
        scanf("%s = %s %s %s", res, op1, operator, op2);

        char *type1 = getType(op1);
        char *type2 = getType(op2);

        if (!type1 || !type2) {
            printf("Error: Undeclared variable used in expression.\n");
            continue;
        }
        printf("\nChecking: %s = %s %s %s\n", res, op1, operator, op2);
        printf("Operand1 type: %s, Operand2 type: %s\n", type1, type2);

        if (checkTypeCompatibility(type1, type2)) {
            char *resultType = (strcmp(type1, "float") == 0 || strcmp(type2, "float") == 0) ? "float" : "int";
            printf("Type Check Passed ✅. Result Type: %s\n", resultType);
            addSymbol(res, resultType); // Add result to symbol table
        } else {
            printf("❌Type Mismatch Error: Cannot operate on %s and %s\n", type1, type2);
        }
    }

    return 0;
}
```

OUTPUT:

Sample Input

```
Enter number of variable declarations: 4
a int
b float
c int
d float

Enter number of expressions to type check: 3
x = a + c
y = b + d
z = a + d
```

Sample Output

```
Checking: x = a + c
  Operand1 type: int, Operand2 type: int
  Type Check Passed ✅. Result Type: int

Checking: y = b + d
  Operand1 type: float, Operand2 type: float
  Type Check Passed ✅. Result Type: float

Checking: z = a + d
  Operand1 type: int, Operand2 type: float
  Type Check Passed ✅. Result Type: float
```

RESULT:

Thus the type checking program has been implemented and output verified successfully

Ex. No. 9	Write a c program to Generate three address codes for a simple program

AIM:

To Generate three address code for simple program

ALGORITHM:

1. Start the program
2. Include the header files and declaring the necessary variables datatypes.
3. Get the file name.
4. Opening the two different types of files using fopen() method.
5. Using switch case statement for issue moving and multiplying the variables
6. Read the file from the user
7. Print the result and then program is destroyed.
8. Stop the program

PROGRAM:

```
#include<stdio.h>
#include<string.h>
void pm();
void plus();
void div();
int i,ch,j,l,addr=100;
char ex[10], exp[10] ,exp1[10],exp2[10],id1[5],op[5],id2[5];
void main()
{
clrscr();
while(1)
{
printf("\n1.assignment\n2.arithmetic\n3.relational\n4.Exit\nEnter the choice:");
scanf("%d",&ch);
switch(ch)
{
case 1:
printf("\nEnter the expression with assignment operator:");
scanf("%s",exp);
l=strlen(exp);
exp2[0]='\0';
i=0;
while(exp[i]!='=')
{
i++;
}
strncat(exp2,exp,i);
strrev(exp);
exp1[0]='\0';
strncat(exp1,exp,l-(i+1));
strrev(exp1);
printf("Three address code:\ntemp=%s\n%s=temp\n",exp1,exp2);
```

```
break;

case 2:
printf("\nEnter the expression with arithmetic operator:");
scanf("%s",ex);
strcpy(exp,ex);
l=strlen(exp);
exp1[0]='\0';

for(i=0;i<l;i++)
{
if(exp[i]=='+'||exp[i]=='-')
{
if(exp[i+2]=='/'||exp[i+2]=='*')
{
pm();
break;
}
else
{
plus();
break;
}
}
else if(exp[i]=='/'||exp[i]=='*')
{
div();
break;
}
}
break;
case 3:
printf("Enter the expression with relational operator");
scanf("%s%s%s",&id1,&op,&id2);
if(((strcmp(op,"<")==0)||(strcmp(op,">")==0)||(strcmp(op,"<=")==0)||(strcmp(op,">=")==0)||(strcmp(op,"=="
)==0)||(strcmp(op,"!=")==0))==0)
printf("Expression is error");
else
{
printf("\n%d\tif %s%s%s goto %d",addr,id1,op,id2,addr+3);
addr++;
printf("\n%d\t T:=0",addr);
addr++;
printf("\n%d\t goto %d",addr,addr+2);
addr++;
printf("\n%d\t T:=1",addr);
}
```

```
break;
case 4:
exit(0);
}
}
}
void pm()
{
strrev(exp);
j=l-i-1;
strncat(exp1,exp,j);
strrev(exp1);
printf("Three address code:\ntemp=%s\ntemp1=%c%ctemp\n",exp1,exp[j+1],exp[j]);
}
void div()
{
strncat(exp1,exp,i+2);
printf("Three address code:\ntemp=%s\ntemp1=temp%c%c\n",exp1,exp[i+2],exp[i+3]);
}
void plus()
{
strncat(exp1,exp,i+2);
printf("Three address code:\ntemp=%s\ntemp1=temp%c%c\n",exp1,exp[i+2],exp[i+3]);
}
```

OUTPUT:

```
1. assignment
2. arithmetic
3. relational
4. Exit
Enter the choice:1
Enter the expression with assignment operator:
a=b
Three address code:
temp=b
a=temp
1.assignment
2.arithmetic
3.relational
4.Exit
Enter the choice:2
Enter the expression with arithmetic operator:
a+b-c
Three address code:
temp=a+b
temp1=temp-c
1.assignment
2.arithmetic
```

```
3.relational
4.Exit
Enter the choice:2
Enter the expression with arithmetic operator:
a-b/c
Three address code:
temp=b/c
temp1=a-temp
1.assignment
2.arithmetic
3.relational
4.Exit
Enter the choice:2
Enter the expression with arithmetic operator:
a*b-c
Three address code:
temp=a*b
temp1=temp-c
1.assignment
2.arithmetic
3.relational
4.Exit
Enter the choice:2
Enter the expression with arithmetic operator:a/b*c
Three address code:
temp=a/b
temp1=temp*c
1.assignment
2.arithmetic
3.relational
4.Exit
Enter the choice:3
Enter the expression with relational operator
a
<=
b
100 if a<=b goto 103
101 T:=0
102 goto 104
103 T:=1
1.assignment
2.arithmetic
3.relational
4.Exit
Enter the choice:4
```

RESULT:

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

Ex. No. 10	Generate the Machine Codes for the given code segment using C Language. Give the Quadruple representation of the given code segment as input for your program. M=n+k*h , n=g/k-m,p=m/n,v=m-k*p

AIM

To write a C program that takes a **quadruple (3-address code)** representation of an expression and **generates machine code** (intermediate-level assembly-like instructions) for it.

ALGORITHM

- **Start with the 3-address code** (quadruples) representation of each statement.
- For each quadruple:
    - Load the arguments into registers (if not already).
    - Perform the operation.
    - Store the result in a variable or temporary.
- Keep a symbol table or register map (for simplicity, we just use a temp counter).
- Print pseudo machine instructions like:

PROGRAM

```
#include <stdio.h>
#include <string.h>

#define MAX 100

typedef struct {
    char op[5];
    char arg1[10];
    char arg2[10];
    char result[10];
} Quad;

int main() {
    int n, i;
    Quad quads[MAX];

    printf("Enter number of quadruples: ");
    scanf("%d", &n);
    printf("Enter quadruples in format: op arg1 arg2 result\n");
    for (i = 0; i < n; i++) {
        scanf("%s %s %s %s", quads[i].op, quads[i].arg1, quads[i].arg2, quads[i].result);
    }

    printf("\nGenerated Machine Code:\n\n");

    for (i = 0; i < n; i++) {
        printf("// %s = %s %s %s\n", quads[i].result, quads[i].arg1, quads[i].op, quads[i].arg2);

        // Load first operand into R1
        printf("MOV R1, %s\n", quads[i].arg1);

        // Apply operation with second operand
        if (strcmp(quads[i].op, "+") == 0) {
            printf("ADD R1, %s\n", quads[i].arg2);
        } else if (strcmp(quads[i].op, "-") == 0) {
            printf("SUB R1, %s\n", quads[i].arg2);
        } else if (strcmp(quads[i].op, "*") == 0) {
            printf("MUL R1, %s\n", quads[i].arg2);
        } else if (strcmp(quads[i].op, "/") == 0) {
            printf("DIV R1, %s\n", quads[i].arg2);
        }

        // Store result
        printf("MOV %s, R1\n\n", quads[i].result);
    }

    return 0;
}
```

OUTPUT:

Sample Input

```
Enter number of quadruples: 7
* k h t1
+ n t1 M
/ g k t2
- t2 m n
/ m n p
* k p t3
- m t3 v
```

Output: Generated Machine Code

Generated Machine Code:

```
// t1 = k * h
MOV R1, k
MUL R1, h
MOV t1, R1

// M = n + t1
MOV R1, n
ADD R1, t1
MOV M, R1

// t2 = g / k
MOV R1, g
DIV R1, k
MOV t2, R1

// n = t2 - m
MOV R1, t2
SUB R1, m
MOV n, R1

// p = m / n
MOV R1, m
DIV R1, n
MOV p, R1

// t3 = k * p
MOV R1, k
MUL R1, p
MOV t3, R1

// v = m - t3
MOV R1, m
SUB R1, t3
MOV v, R1
```

Result:

Thus the Machine Codes for the Quadruple representation program has been created & executed successfully.

Ex. No. 11	**Write a c program Implement back-end of the compiler for which the three-address code is given as input and the assembly language code is produced as output**

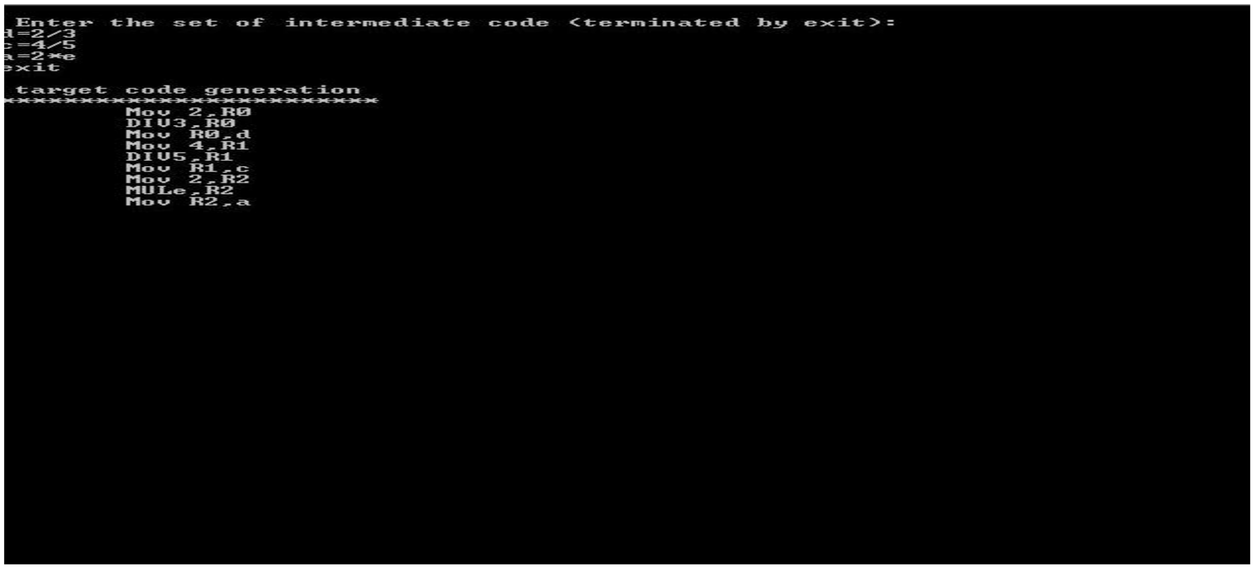
AIM:
To implement the back end of the compiler which takes the three address code and produces the 8086 assembly language instructions

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:



```
 Enter the set of intermediate code (terminated by exit):
d=2/3
c=4/5
a=2*e
exit

 target code generation
***************************
         Mov  2,R0
         DIV3,R0
         Mov  R0,d
         Mov  4,R1
         DIV5,R1
         Mov  R1,c
         Mov  2,R2
         MULe,R2
         Mov  R2,a
```

Result:

Thus the program was implemented for back-end of the compiler for which the three-address code is given as input and the assembly language code has been generated

Ex. No. 12	**Write a c program 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:

The code generation algorithm takes as input a sequence of three – address statements constituting a basic block. For each three – address statement of the form x := y op z we perform the following actions:

1. Invoke a function getreg to determine the location L where the result of the computation y op z should be stored. L will usually be a register, but it could also be a memory location. We shall describe getreg shortly.
2. Consult the address descriptor for y to , (one of) the current location(s) of y. prefer the register′determine y if the value of y is currently both in memory and a register. If′for y , L′the value of y is not already in L, generate the instruction MOV y to place a copy of y in L.

, L′3. Generate the instruction OP z is a current location of z. Again, prefer a register to a′where z memory location if z is in both. Update the address descriptor of x to indicate that x is in location L. If L is a register, update its descriptor to indicate that it contains the value of x, and remove x from all other register descriptors.

4. If the current values of y and/or z have no next users, are not live on exit from the block, and are in register descriptor to indicate that, after execution of x := y op z, those registers no longer will contain y and/or z, respectively

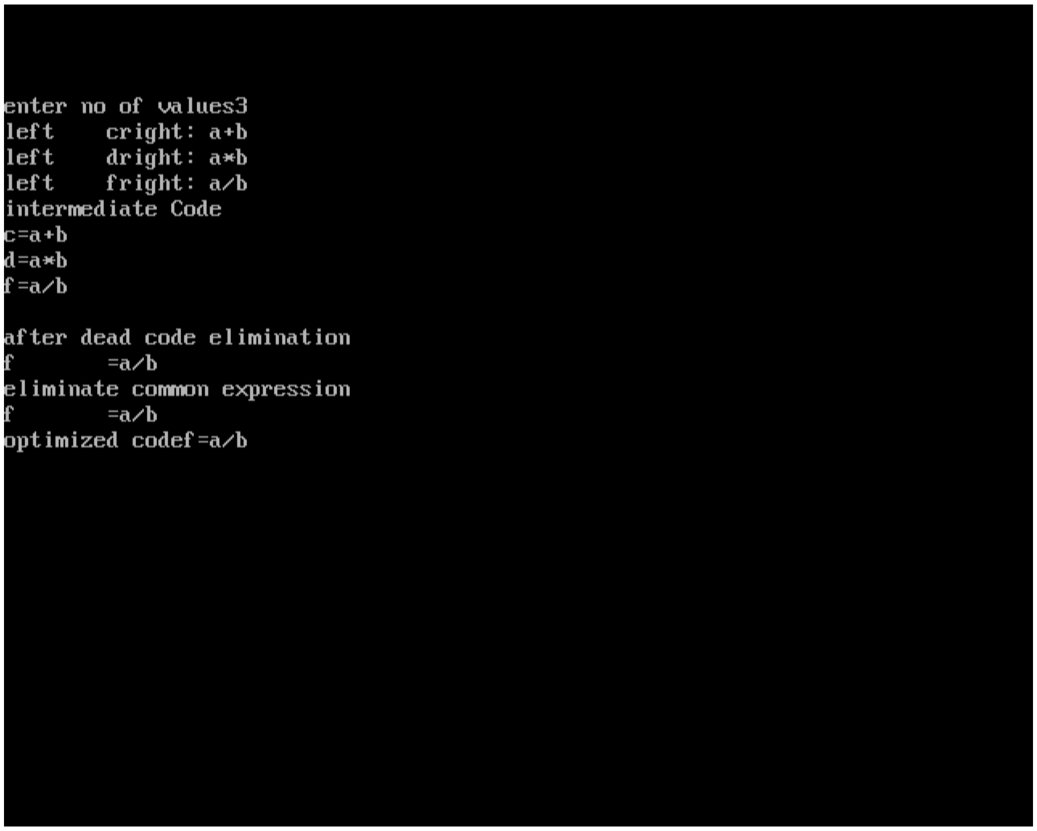
PROGRAM:

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
struct op
{
char l;
char r[20];
}op[10],pr[10];

void main()
{
int a,i,k,j,n,z=0,m,q;

char *p,*l;
char temp,t;
char *tem;
clrscr();
```

```
printf("enter no of values");
scanf("%d",&n);
for(i=0;i<n;i++)
{
printf("left\t");
op[i].l=getche();
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-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++ ;

}} }
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");
for(i=0;i<z;i++)
{
if(pr[i].l!='\0')
{
printf("%c=",pr[i].l);
printf("%s\n",pr[i].r);
}
}
getch();
}
```

OUTPUT:

Thus the program was implemented to the code optimization technique has been successfully executed



```
enter no of values3
left    cright: a+b
left    dright: a*b
left    fright: a/b
intermediate Code
c=a+b
d=a*b
f=a/b

after dead code elimination
f       =a/b
eliminate common expression
f       =a/b
optimized codef=a/b
```

Result:

Thus the program was implemented to the code optimization technique has been successfully executed