M.V.S.R. ENGINEERING COLLEGE

(Affiliated to Osmania University & Recognized by AICTE)
Nadergul, RangaReddy Dist.

CERTIFICATE

Department of COMPUTER SCIENCE & ENGINEERING

Certified that this is a bonafide work of lab experiments carried out by Mr/Ms.______________________________bearing Roll.No.______________________under the course of **Compiler Design** (PC631CS) Laboratory prescribed by Osmania University for B.E. **Sem-VI** (AICTE) of Computer Science & Engineering during the academic year 2020–2021.

Internal Examiner ***External Examiner***

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

VISION

- To impart technical education of the highest standards, producing technically competent confident and socially responsible engineers.

MISSION

- To impart adequate fundamental knowledge, technical and soft skills to students.
- To make learning process exciting, stimulating, and joyful.
- To create a climate conductive to excellent teaching learning process.
- To bring out creativity in students.
- To contribute to advancement of engineering & technology.
- To make positive contribution to meet societal needs.

PROGRAM EDUCATIONAL OBJECTIVES (PEOs)

The Program Educational Objectives of undergraduate program in Computer Science & Engineering are to prepare graduates who will:

1. Obtain strong fundamentals concepts, technical competency and problem-solving skills to generate innovative solutions to engineering problems.
2. Continuously enhance their skills through training, independent inquiry, professional practices and pursue higher education or research by adapting to rapidly changing technology.
3. Advance in their professional careers including increased technical, multidisciplinary approach and managerial responsibility as well as attainment of leadership positions thus making them competent professionals at global level.
4. Exhibit commitment to ethical practices, societal contributions and lifelong learning.

(A) PROGRAM OUTCOMES(POs)

At the end of the program the students (Engineering Graduates) will be able to:

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 analyse 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 modelling 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 principle and apply these to one’s own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.
12. **Lifelong 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.

(B) PROGRAM SPECIFIC OUTCOMES (PSOs)

13. **Efficient coding:** an ability to analyse a problem, design the algorithm and optimally code its solution.
14. **Software deployment:** an ability to identify & define computing requirements to test, implement and maintain a software product.

Course Objectives:

- This course is designed to provide a comprehensive knowledge of Compiler Design.
- To learn how to design compiler to translate High Level Languages to Machine Language.
- To learn different phases of compiler and how to implement them.
- To learn efficient machine Language Code Generation using the techniques of Optimization.

Course Prerequisites:

Basic knowledge of the relevant undergraduate courses of the first two years is required:

- Programming (essential concepts of imperative and object-oriented programming languages and elementary programming techniques)
- Data structures and algorithms (lists, stacks, queues, trees and associated algorithms)
- Formal languages and automata theory (regular and context-free languages, finite and pushdown automata)

Course Outcomes:

Course outcome_ number	Student will be able to
CO1	Apply the knowledge of LEX to develop a C scanner
CO2	Develop hand written top down parsers like recursive descent parser and construct first and follow sets for a given grammar.
CO3	Develop hand written bottom up parser for a given grammar.
CO4	Apply the knowledge of YACC to syntax directed translations for generating intermediate code-3 address code.
CO5	Apply the knowledge of YACC to syntax directed translations for generating intermediate code-3 address code.

M.V.S.R. ENGINEERING COLLEGE

(Affiliated to Osmania University, Hyderabad)
Nadergul(P.O.), Hyderabad-501510

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

Subject : **Compiler Design Lab- PC 631 CS** Acad. Year : **2020-21**

Class : **BE III/IV** **Sem – VI** Section :

INDEX

S.No.		Name of theProgram	Date		Pages	
			Experiment	Submission	From	To
1		Tokenizing an assignment statement				
2		<u>Sample programs using LEX</u>				
	a.	Identification of a decimal number				
	b.	Word count without using files				
	c.	Word count using files				
3		Scanner Program using LEX				
4		Elimination of Immediate Left Recursion				
5		Left factoring a given grammar				
6		<u>Top-down Parsers</u>				
	a.	Recursive Descent Parser (Grammar-1)				
	b.	Recursive Descent Parser (Grammar-2)				
7		Construction of LR(0) items				
8		Evaluation of an Expression using YACC				
9		Intermediate Code Generation using YACC (3 Address Code)				
10		Code Generation: Single and Double Address				
11		Target Code Generation				
12		Code Optimization				

1. TOKENIZING AN ASSIGNMENT STATEMENT

```
#include<stdio.h>
#include<string.h>
void main()
{
 char a[10]; int x,i;

        printf("enter exp\n"); gets(a);
        for(i=0;a[i]!='\0';i++)
        {
                if((a[i]>='A' && a[i]<='Z') || (a[i]>='a' && a[i]<='z'))
                        printf("\n %c is variable",a[i]);
                else if(a[i]=='=')
                        printf("\n %c is assignment operator ",a[i]);

                else if( a[i]=='+' || a[i]=='-' || a[i]=='*' || a[i]=='/' )
                        printf("\n %c is arithmetic operator ",a[i]);

                else if(a[i]>='0' && a[i]<='9')
                        printf("\n %c is  integer ",a[i]);
               else
                        printf("\n %c is other character",a[i]);

        }
}
```

Output:

```
$./a.out
enter exp a=b+3;

 a is variable
 = is assignment operator
  b is variable
 + is arithmetic operator
 3 is integer
 ; is other character
```

2. SAMPLE PROGRAMS USING LEX

a. Identification of a decimal number

```
//decimal.l
%{
#include<stdio.h>
%}
%%
[ \n\t];

[+-]?([0-9]+|([0-9]*\.[0-9]+)([eE][+-]?[0-9]+)?) { printf("number\n");}
. { printf("not a number\n");}
%%
main()
{
 yylex();
}
```

Output:

```
$ lex decimal.l
$ cc lex.yy.c -ll
$ ./a.out

7822.56
number

e
not a number

15.50e4
number

6
number
```

b. Word count without using files

```
//wordcount.l
%{
#include<stdio.h>
 int wc=0,lc=0,cc=0;
%}
word [^ \n\t]+
eol \n
%%
{word} {wc++;cc+=yyleng;}
{eol} {lc++;cc++;}
. {cc++;}
%%
main()
{
yylex();
printf("no. of characters:%d\nno. of words:%d\nno. of lines:%d\n",cc,wc,lc);
}
```

Output:

```
$ lex wordcount.l
$ cc lex.yy.c -ll
$ ./a.out
This is our CC lab. ^d
no. of characters:20
no. of words:5
no. of lines:1
```

c. Word count using files

```
//wordfile.l
%{
#include<stdio.h>
int wc=0,lc=0,cc=0;
%}
word [^ \n\t]+
eol \n
%%
{word} {wc++;cc+=yyleng;}
{eol} {lc++;cc++;}
. {cc++;}
%%
main(int argc,char**argv)
{
if(argc>1)
{
FILE*file;
file=fopen(argv[1],"r");
 if(!file)
{
fprintf(stderr,"could not open%s\n",argv[1]);
exit(1);
}
yyin=file;
}
yylex();
printf("no. of characters:%d\nno. of lines:%d\nno.of words:%d\n",cc,lc,wc);
return 0;
}
```

INPUT:

```
//file
This is CC lab.
This is CSE Department. ^d
```

OUTPUT:

```
$ lex wordfile.l
$ cc lex.yy.c -ll
$ ./a.out file no. of
characters:40 no. of
lines:2
no.of words:8
```

3. SCANNER PROGRAM USING LEX

```
//cscanner.l
//C SCANNER USING LEX
%{
#include<stdio.h>
int lineno=1;
%}
letter [a-zA-z]
digit [0-9]
id {letter}({letter}|{digit})*
num {digit}+
kw "int"|"char"|"float"|"printf"|"main"|"if"|"elseif"|"then"|"void"
rop ">"|"<="|">="|"<"
aop "+"|"-"|"*"|"/"
arr ({id}"["{num}"]")
pre "#include"|"#define"
 par "["|"]"|"("|")"|"{"|"}"
com ("/*"({id}|"\n")*"*/")
ws [.,;"]
%%
[\n] {lineno++;}
{ws} {printf("\nspecial symbols =%s \t\tlineno=%d",yytext,lineno);}
{rop} {printf("\nrelational operator=%s \t\tlineno=%d",yytext,lineno);}
 "=" {printf("\nassignment operator =%s \t\t lineno=%d",yytext,lineno);}
{aop} {printf("\narithmetic operator= %s \t\tlineno=%d",yytext,lineno);}
{par} {printf("\nparenthesis= %s \t\tlineno=%d",yytext,lineno);}
{pre} {printf("\npreprocessor=%s \t\tlineno=%d",yytext,lineno);}
{kw} {printf("\nkeyword=%s \t\tlineno=%d",yytext,lineno);}
{arr} {printf("\narrays=%s \t\tlineno=%d",yytext,lineno);}
{id} {printf("\nidentifier=%s \t\tlineno=%d",yytext,lineno);}
{num} {printf("\nnumber=%s \t\tlineno=%d",yytext,lineno);}
\"[^\"\n]*\" {printf("\n control string=%s \t\tlineno=%d",yytext,lineno);}
%%
int main(int argc,char* argv[])
{
if(argc>1) yyin=fopen(argv[1],"r");
else
yyin=stdin; yylex();
}
```

Output:

```
$ lex lexscanner.l
$ cc lex.yy.c -ll
$ ./a.out test.c
//test.c
#include<stdio.h>
 void main()
{
int a=20,b=10; if(a>b)
printf("%d is greater",a);
else
printf("b is greater");
}
```

```
preprocessor=#include                 lineno=1
relational operator=<                 lineno=1
identifier=stdio                      lineno=1
special symbols =.                    lineno=1
identifier=h                          lineno=1
relational operator=>                 lineno=1
keyword=void                          lineno=2
keyword=main                          lineno=2
parenthesis= (                        lineno=2
parenthesis= )                        lineno=2
parenthesis= {                        lineno=3
keyword=int                           lineno=4
identifier=a                          lineno=4
assignment operator ==                lineno=4
number=20                             lineno=4
special symbols =,                    lineno=4
identifier=b                          lineno=4
assignment operator ==                 lineno=4
number=10                             lineno=4
special symbols =;                    lineno=4
keyword=if                            lineno=5
parenthesis= (                        lineno=5
identifier=a                          lineno=5
relational operator=>                 lineno=5
identifier=b                          lineno=5
parenthesis= )                        lineno=5
keyword=printf                        lineno=6
parenthesis= (                        lineno=6
 control string="%d is greater"       lineno=6
special symbols =,                    lineno=6
identifier=a                          lineno=6
parenthesis= )                        lineno=6
special symbols =;                    lineno=6
identifier=else                       lineno=7
keyword=printf                        lineno=8
parenthesis= (                        lineno=8
 control string="b is greater"        lineno=8
parenthesis= )                        lineno=8
special symbols =;                    lineno=8
parenthesis= }                        lineno=9
```

4. ELIMINATION OF IMMEDIATE LEFT RECURSION

```
//leftrec.c
#include<stdio.h>
#include<string.h>
char alpha[20]={0};
char beta[20]={0};
char gram[30]={0};
int i=0,j=0,k=0;
void addToBeta(char);
void addToAlpha(char);
void ELR()
{
for(i=0;gram[i]!='\0';i++)
if(gram[i]=='>')
break;
for(i=i+1;gram[i]!='\0';i++)
{
if(gram[i]==gram[0])
{
for(i=i+1;gram[i]!='\0'&&gram[i]!='|';i++)
addToAlpha(gram[i]);
addToAlpha(';');
}
else
{
for(;gram[i]!='\0'&&gram[i]!='|';i++)
addToBeta(gram[i]);
addToBeta(';');
}
}
alpha[j]='\0';
beta[k]='\0';
}
void addToAlpha(char ch)
{
alpha[j]=ch;
 j++;
}
void addToBeta(char ch)
{
beta[k]=ch;
k++;
}
int main()
{
printf("\nenter the grammer:\n");
scanf("%s",gram);
ELR();
if(strlen(alpha)==0)
{
printf("\nThe grammer is not left recursive");
return 0;
}
else
{
printf("\nThe grammer after eliminating left recursion is:\n");
printf("\n%c->",gram[0]);
for(i=0;beta[i+1]!='\0';i++)
{
if(beta[i]==';')
printf("X|");
else
printf("%c",beta[i]);
}
printf("X");
printf("\nX->"); for(i=0;alpha[i+1]!='\0';i++)
{
if(alpha[i]==';')
printf("X|");
else
printf("%c",alpha[i]);
}
printf("X|%s",”Epsilon”);
return 0;
}
}
```

Output:

```
$ cc leftrec.c
$ ./a.out

enter the grammer:
T->T*F|F

The grammer after eliminating left recursion is:

T->FX
X->*FX|Epsilon
```

5. LEFT FACTORING A GIVEN GRAMMAR

```
//leftfact.c
#include<stdio.h>
#include<string.h>
char a[30];
char b[30];
char g[30];
char gr[30];
char p1[30];
char p2[30];
char gam[30];
void left()
{
 int
 i=0,j=0,k=0,l=0,m=0,len2;
 while(gr[i]!='>')
 i++;
 for(i=i+1;gr[i]!='|';i++)
 p1[j++]=gr[i];
 p1[j]='\0';
 j=0;
 for(i=i+1;gr[i]!='\0';i++) p2[j++]=gr[i];
 p2[j]='\0';
l=strlen(p2); for(i=0;p2[i]!='\0';i++)
if(p2[i]=='|')
break;
if(i<l)
{
printf("Gamma part is there\n");
j=0;
for(m=0;m<i;m++)
p2[j++]=p2[m];
p2[j]='\0';
j=0;
for(m=i+1;m<l;m++)
gam[j++]=p2[m];
gam[j]='\0';
printf("Gamma is %s\n",gam);
}
else
printf("Gamma part is not there\n");
while(p1[k]==p2[k])
 k++;
 j=0;
 for(i=0;i<k;i++)
 a[j++]=p2[i];
 a[j]='\0';
 j=0;
 for(i=k;p1[i]!='\0';i++)
 b[j++]=p1[i];
 b[j]='\0';
 j=0;
 for(i=k;p2[i]!='\0';i++)
 g[j++]=p2[i];
 g[j]='\0';
}
 int main()
{
 int len;
 printf("\n enter the grammar with atmost three productions where only the first two productions have a
common prefix :\n");
 scanf("%s",gr);
 left();
 len=strlen(gam);
 printf("left factored grammar is:\n");
 if(len)
 printf("%c->%sX|%s\n",gr[0],a,gam);
 else
 printf("%c->%sX\n",gr[0],a);
 printf("X->%s|%s\n",b,g);
 return 0;
 }
```

Output:

```
$ cc leftfact.c
$ ./a.out
enter the grammar with atmost three productions where only the first two productions have a common
prefix :
S->abA|abB|cd
Gamma part is there
Gamma is cd
left factored grammar is:
S->abX|cd
X->A|B
enter the grammar with atmost three productions where only the first two productions have a common
prefix :
S->aA|aB
Gamma part is not there
left factored grammar is:
S->aX
X->A|B
enter the grammar with atmost three productions where only the first two productions have a common
prefix :
S->aA|aB|c
Gamma part is there
Gamma is c
left factored grammar is:
S->aX|c
X->A|B
```

6. TOP DOWN PARSERS

a. Recursive descent parser (Grammar 1)

```
//recursive.c

#include<stdio.h>
void E();
void E1();
void T();
void T1();
void F();
void match(char);
int flag=1;
char ch,t;
int main()
{
printf("The grammer is\n");
printf("E-->E+T|T\nT-->T*F|F\nF-->i\n");
printf("The elimination of left recursion is needed for implementing recersive descent parser\n");
printf("The grammer after elimination of left recursion is\n");
printf("E-->TE'\nE'-->+TE'|%s\nT-->FT'\nT'-->*FT'|%s\nF-->i\n",”Epsilon”,”Epsilon”);
printf("enter input string and end the string with $\n");
scanf("%c",&ch);
E();
if((ch=='$')&&(flag!=0))
printf("successful\n");
else
printf("unsuccessful\n");
}
void match(char t)
{
if(ch==t)
scanf("%c",&ch);
else
flag=0;
}
void E()
{
T();
E1();
}
void E1()
{
if(ch=='+')
{
match('+');
T();
E1();
}
else
return;
}
void T()
{
F();
T1();
}
void T1()
{
if(ch=='*')
{
match('*');
F();
T1();
}
else
return;
}
void F()
{
match('i');
}
```

Output:

```
$ cc recursive.c
$ ./a.out

The grammer is
E-->E+T|T
T-->T*F|F
F-->i

The elimination of left recursion is needed for implementing recursive descent parser
arser
The grammer after elimination of left recursion is
E-->TE'
E'-->+TE' | Epsilon
T-->FT'
T'-->*FT' | Epsilon
F-->i

enter  input string and end the string with $
 i*i+i$
successful

$ ./a.out
The grammer is
E-->E+T|T
T-->T*F|F
F-->i
```

```
The elimination of left recursion is needed for implementing recursive descent parser
arser
The grammer after elimination of left recursion is

E-->TE'
E'-->+TE'|Epsilon
T-->FT'
T'-->*FT'|Epsilon
F-->i

enter input string and end the string with $

i$
successful

$ ./a.out

The grammer is
E-->E+T|T
T-->T*F|F
F-->i

The elimination of left recursion is needed for implementing recursive descent parser
 arser
The grammer after elimination of left recursion is

E-->TE'
E'-->+TE'|Epsilon
T-->FT'
T'-->*FT'|Epsilon
F-->i

enter input string and end the string with $

 i+i$
successful
```

b. Recursive descent parser (Grammar 2)

```
//recursive2.c
#include<stdio.h>
void E();
void T();
void match(char);
int flag=1;
char ch,t;
int main()
{
printf("The grammer is\n");
printf("E-->x+T\nT-->(E)|x\n");
printf("enter input string and end with dollar\n");
scanf("%c",&ch);
E();
if((ch=='$')&&(flag!=0))
printf("successful\n");
else
printf("unsuccessful\n");
}
void match(char t)
{
if(ch==t)
scanf("%c",&ch);
else
flag=0;
}
void E()
{
if(ch=='x') {
match('x');
match('+');
T(); }
}
void T()
{
if(ch=='(')
{
match('(');
E();
match(')');
}
else
match('x');
}
```

Output:

```
$ cc recursive2.c
$ ./a.out

The grammer is
E-->x+T
T--
>(E)|x
enter input string and end with dollar
x+(x+x)$
successful


$ ./a.out

The grammer is
T-->(E)|x
enter input string and end with dollar
(x)$
unsuccessful

$ ./a.out

The grammer is
E-->x+T
T--
>(E)|x
enter  input string and end with dollar
x+x$
successful
```

7. CONSTRUCTION OF LR(0) ITEMS

```
#include<stdio.h>
#include<string.h>
main()
{
char A[10][10],items[20][20]={};
int i,j,k,l=0,m=0,p=3,n=0,ilen[10];
char c,ch;
int flag=0,numprods,temp,temp1,q;
printf("Enter the number of productions\n");
scanf("%d",&numprods);
printf("Enter productions:\n");
for(i=0;i<=numprods;i++)
{
        gets(A[i]);
}

//Finding items

for(i=0;i<=numprods;i++)
{
        n=strlen(A[i]);
        for(j=0;j<(n-2);j++)
        {
                for(k=0;k<=strlen(A[i]);k++)
                {
                        if(p==k)
                        {
                                items[l][m]='.';
                                m++;
                                items[l][m]=A[i][k];
                                m++;
                                }
                                else
                                {
                                items[l][m++]=A[i][k];

                }
                 l++;
                p++;
                m=0;
}
p=3;
}
i=0;
printf("Items are:\n);
while(l!=i)
{
printf("%d",i);
puts(items[i]);
printf("\n");
i++;
}
}
```

Output:

```
$ ./a.out

Enter the number of productions
3
Enter  productions:

S->BC
B->a
C->b

Items are:

0)S->.BC

1)S->B.C

2)S->BC.

3)B->.a

4)B->a.

5)C->.b

6)C->b.
```

8. EVALUATION OF AN EXPRESSION USING YACC

```
//exp_ev.y
%{
#include<stdio.h>
#include<ctype.h>
 #define YYSTYPE double
%}
%token NUM
%left '+' '-'
%left '*' '/'
%right UMINUS
%%
lines:lines expr'\n' {printf("%g\n",$2);}
|lines'\n'
|/*empty*/
;
expr:expr'+'expr {$$=$1+$3;}
|expr'-'expr {$$=$1-$3;}
|expr'*'expr {$$=$1*$3;}
|expr'/'expr {$$=$1/$3;}
|'('expr')' {$$=$2;}
|'-'expr %prec UMINUS {$$=-$2;}
|NUM
;
%%

main()
{
yyparse();
}
void yyerror(char *s)
{
printf("\nerror\n");
}
yylex()
{
char c;
while((c=getchar())==' ');
if((c=='.')||(isdigit(c)))
{
ungetc(c,stdin);
scanf("%lf",&yylval);
return NUM;
}
return c;
}
```

Output:

```
$ yacc exp_ev.y
$ cc y.tab.c -ly
$ ./a.out

2*(3+7)
20
15/(5-2)
5
```

9. INTERMEDIATE CODE GENERATION USING YACC (3 ADDRESS CODE)

```
//three.y

%{
#include<stdio.h>
#include<ctype.h>
int i;
%}

%token id
%%
L:id'='E '\n' {printf("%c=%c\n",$1,$3);}
|
;

E:E'+'T {$$='p'; printf("%c=%c+%c\n",$$,$1,$3);}
|E'-'T {$$='r'; printf("%c=%c-%c\n",$$,$1,$3);}
|T
;

T:T'*'F {$$='q'; printf("%c=%c*%c\n",$$,$1,$3);}
|T'/'F {$$='s'; printf("%c=%c/%c\n",$$,$1,$3);}
|F
;

F:'-'G {$$='t'; printf("%c=-%c\n",$$,$2);}
|G
;

G:id{$$=$1;}
;
%%

#include"lex.yy.c"
main()
{
printf("Three Address Code is:\n");
yyparse();
}

void yyerror(char *s)
{
printf("\nerror!!\n");
}
```

```
//lexip.l

opr [*+-/=]
%%
[\t]. {}
[a-zA-Z]* {sscanf(yytext,"%c",&yylval);return id;}
 [\n] {return(yytext[0]);}
{opr} {return(yytext[0]);}

%%

yywrap()
{
return 1;
}
```

Output:

```
$ lex lexip.l
$ yacc three.y
$ cc y.tab.c -ly
$ ./a.out

Three  Address Code  is:

a=b/c+d*e-f*g
s=b/c
q=d*e
p=s+q
q=f*g
r=p-q
a=r
```

10. CODE GENERATION (SINGLE AND DOUBLE ADDRESS)

```
//codegen.c

#include<stdio.h>
 int main()
{
int
i=0,ch,k=3;
char a[100];
printf("\n enter a string:");
scanf("%s",a);
printf("\n1.single adress\n 2.Double Adress\n");
printf("Enter choice:");
scanf("%d",&ch);
switch(ch)
{
case 1:
printf("\nLOAD %c\n",a[2]);
while(a[i]!='\0')
{
 if(a[i+3]=='+')
printf("\nADD %c\n",a[i+4]);
else
if(a[i+3]=='-')
printf("\nSUB  %c\n",a[i+4]);
i=i+2;
}
printf("\nSTORE %c\n",a[0]);
break;
case 2:
while(a[k]!='\0')
{
if(a[k]=='+')
printf("\nADD %c %c\n",a[2],a[k+1]);
else
if(a[k]=='-')
printf("\nSUB %c  %c\n",a[2],a[k+1]);
k=k+2;
}
 if(a[1]=='=')
printf("\nMOV %c %c\n",a[0],a[2]);
break;
}
}
```

Output:

```
$ cc  codegen.c
$ ./a.out
enter a string:a=b+c-d

1.single adress
2.Double Adress
Enter choice:2

ADD b c
SUB b d
MOV a b

$ ./a.out
 enter a string:a=b+c-d

1.single adress
2.Double Adress
Enter choice:1

LOAD b
ADD c
SUB d
STORE    a
```

11. TARGET CODE GENERATION

```
//targetcode.c

 #include <stdio.h>
#define max 20
static int r=0;
int top=-1;
char st[max];
int main()
{
int h;
char ip[10];
printf("\nenter the postfix expression:");
scanf("%s",ip);
h=0;
while(ip[h]!='\0')
{
while(islower(ip[h])!=0)
{
push(ip[h]);
h++;
}
getregister();
printf("\nload %c R%d\n",st[top-1],r);
switch(ip[h])
{
case '+':
{
printf("\nadd %c R%d\n",st[top],r);
empty();
h++;
}
break;
case'-':
{
printf("\nsub %c R%d\n",st[top],r);
empty();
h++;
}
break;
case'*':
{
printf("\n mul %c R%d\n",st[top],r);
empty();
h++;
}
break;
case '/':
{
printf("\n div %c R%d\n",st[top],r);
empty();
h++;
}
break;
default:
printf("\n invalid choice");
}
}
return 0;
}
empty()
{
char t1;
pop();
pop();
printf("\n st R%d t\n",r);
t1='t';
push(t1);
}
getregister()
{
 r++;
if(r>2)
r=1;
}
push(int x)
{
if(top==max-1)
printf("\n stack full\n");
else
{
top++;
st[top]=x;
}
}
pop()
{
if(top==-1)
printf("\n stack empty\n");
else
top--;
}
```

Output:

```
$ cc targetcode.c
$ ./a.out

enter the postfix expression: abc/+
load b R1
div  c R1
st  R1 t
load  a R2
add  t R2
st  R2  t

$ ./a.out

enter the postfix expression:abcd+/*
load c R1
add d R1
st R1  t
load b  R2
div t  R2
st R2 t
load a R1
mul t  R1
st R1  t
```

12. CODE OPTIMIZATION

```
//optimize.c

#include<stdio.h>
#include<string.h>
void main()
{
char str[25][50],op[25][50],forloopparam[90],righthandparam[10][40],lefthandparam[90];
 int i=0,k=0,j=0,m=0,n=0,q=0,s=0,l=0;
int flag[10]={0},count[10]={0};
printf("\n input the loop to be optimised:\n");
gets(str[0]);
while(str[k][i++]!=';');
while(str[k][i++]!=';');
while(str[k][i]!=')')
{
if(isalpha(str[k][i]))
forloopparam[j++]=str[k][i];
i++;
}
 i=0;
strcpy(op[l++],str[0]);
gets(str[0]);
while(str[0][i++]!='{');
strcpy(op[l++],str[0]);
k=0;
while(gets(str[k])&&str[k][0]!='}')
{
while(str[k][i++]!='=');
lefthandparam[n++]=str[k][i-2];
k++;
i=0;
}
for(m=0,i=0;m<k;m++)
{
while(str[m][i++]!='=');
while(str[m][i]!=';')
{
if(isalpha(str[m][i]))
righthandparam[m][count[m]++]=str[m][i];
i++;
}
i=0;
}
for(m=0;m<k;m++)
for(s=0;s<count[m];s++)
for(i=0;i<n;i++)
{
if(righthandparam[m][s]==lefthandparam[i])
flag[m]=1;
}
for(i=0;i<k;i++)
for(q=0;q<j;q++
)
{
if(lefthandparam[i]==forloopparam[q])
flag[i]=1;
}
for(m=1;m<k;m++)
for(s=0;s<count[m];s++)
for(i=0;i<j;i++)
{
if(righthandparam[m][s]==forloopparam[i])
flag[m]=1;
}
printf("\n\noptimised loop is\n");
for(i=0;i<l;i++)
puts(op[i]);
for(i=0;i<k;i++)
if(flag[i]==1)
puts(str[i]);
puts(str[k]);
for(i=0;i<k;i++)
if(flag[i]==0)
printf("%s\t\n",str[i]);
}
```

Output:

```
$ cc optimize.c
$ ./a.out

 input the loop to be optimised:

 for(i=0;i<3;i++)
{
i=i+1;
c=a+b;
}

optimised loop is
for(i=0;i<3;i++)
{
i=i+1;
}
c=a+b;
```