GOVERNMENT COLLEGE OF ENGINEERING

DHARMAPURI – 636 704

(Affiliated to Anna University, Chennai)

BONAFIDE CERTIFICATE

NAME :……

REG.NO :……..

DEGREE & BRANCH :……..

SUBJECT CODE & TITLE :…….

YEAR & SEM :……..

Certified that this Bonafide Record of Work done by the above student of the …….. Laboratory during the year 20 - 20 .

Staff In-Charge **Head of the Department**

Submitted for the University Practical Examination held on …….at Government College of Engineering, Dharmapuri.

Internal Examiner **External Examiner**

INDEX

EX.NO: 1	
DATE:	**IMPLEMENTATION OF SYMBOL TABLE**

AIM:

To write a program for implementing Symbol Table using C.

ALGORITHM:

Step1: Start the program for performing insert, display, delete, search and modify option in symbol table

Step2: Define the structure of the Symbol Table

Step3: Enter the choice for performing the operations in the symbol Table

Step4: If the entered choice is 1, search the symbol table for the symbol to be inserted. If the symbol is already present, it displays “Duplicate Symbol”. Else, insert the symbol and the corresponding address in the symbol table.

Step5: If the entered choice is 2, the symbols present in the symbol table are displayed.

Step6: If the entered choice is 3, the symbol to be deleted is searched in the symbol table.

Step7: If it is not found in the symbol table it displays “Label Not found”. Else, the symbol is deleted.

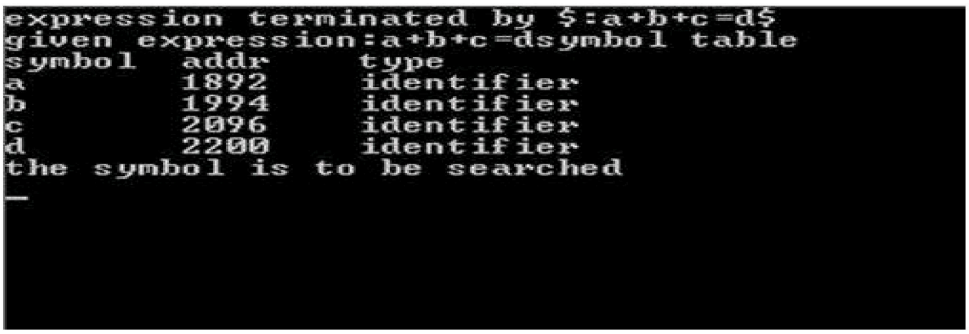
Step8: If the entered choice is 5, the symbol to be modified is searched in the symbol table.

PROGRAM :

```
#include<stdio.h>
#include<ctype.h>
#include<stdlib.h>
#include<math.h>
void main()
{
 int i=0,j=0,x=0,n;
 void *p,*add[5];
 char ch,srch,b[15],d[15],c;
 printf("Expression terminated by $:");
 while((c=getchar())!='$'){
  b[i]=c;
  i++;}
 n=i-1;
 printf("Given Expression:");
while(i<=n){
  printf("%c",b[i]);
  i++; }
 printf("\n Symbol Table\n");
 printf("Symbol \t addr \t type");
 while(j<=n){
  c=b[j];
  if(isalpha(toascii(c)){
   p=malloc(c);
   add[x]=p;
   d[x]=c;
```

```
 printf("\n%c \t %d \t identifier\n",c,p);
 x++;   j++; }
else{
 ch=c;
 if(ch=='+'||ch=='-'||ch=='*'||ch=='=') {
  p=malloc(ch);
  add[x]=p;
  d[x]=ch;
  printf("\n %c \t %d \t operator\n",ch,p);
  x++; j++;
 }}}}
```

OUTPUT:



RESULT:

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

EX.NO: 2	**IMPLEMENT LEXICAL ANALYZER USING LEX TOOL**
DATE:	

AIM:

To write a LEX program to implement the Lexical analyzer using flex tool.

ALGORITHM:

Step 1: Start the program

Step 3: Lex program consists of three parts.

Step 4: Declaration %%

Step 5: Translation rules %%

Step 6: Auxiliary procedure.

Step 7: The declaration section includes declaration of variables, main test,

Step 8: constants and regular Definitions.

Step 9: Translation rule of lex program are statements of the form

P1{action}

P2{action}

Pn{action}

Step 10: Write program in the vi editor and save it with .1 extension.

Step 11: Compile the lex program with lex compiler to produce output file as lex.yy.c.

Eg. $ lex filename.1

$gcc lex.yy.c-11

Step 12: Compile that file with C compiler and verify the output.

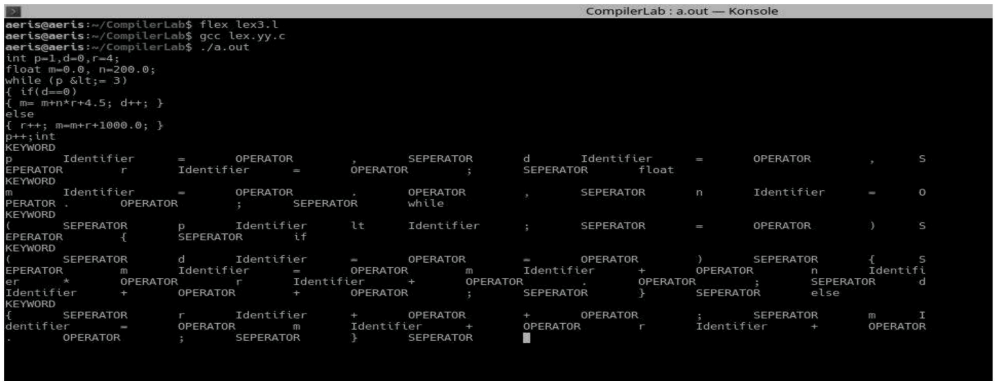
PROGRAM:

```
%{
#include<stdio.h>
%}
%%
auto |
double |
int |
long |
switch |
case |
enum |
register |
typedef |
char |
extern |
return |
union |
continue |
for |
signed |
void |
do |
if |
static |
while |
```

```
default |
goto |
sizeof |
volatile |
const |
float |
short     {ECHO; printf("\nKEYWORD\n");}

 [{};,()]    {ECHO; printf("\tSEPERATOR\t");}

 [+-/=*%]    {ECHO; printf("\tOPERATOR\t");}

 ([a-zA-Z][0-9])+|[a-zA-Z]*  {ECHO; printf("\tIdentifier\t");}
.|\n ;
%%
int yywrap()
{
return 1;
}
int main(void)
{
yylex();
return 0;
}
}
```

OUTPUT:

```
int p=1,d=0,r=4;
   float m=0.0, n=200.0;
   while (p <= 3)
   {
         if(d==0)
               { m= m+n*r+4.5; d++; }
         else
               { r++; m=m+r+1000.0; }
         p++;
   }
```



RESULT:

Thus, the program for the lexical analysis using LEX has been successfully executed and output is verified.

EX.NO: 3(a)	**GENERATE YACC SPECIFICATION**
DATE:	**ARITHMETIC EXPRESSION THAT USES OPERATOR +,-,* AND /.**

AIM:

To write a program to recognize a valid arithmetic expression that uses operator +, - , * and / using YACC tool.

ALGORITHM:

LEX

Step 1: Declare the required header file and variable declaration with in '%{' and '%}'.

Step 2: LEX requires regular expressions to identify valid arithmetic expression token of lexemes.

Step 3: LEX call yywrap() function after input is over. It should return 1 when work is done or should return 0

when more processing is required.

YACC

Step 1: Declare the required header file and variable declaration with in '%{' and '%}'.

Step 2: Define tokens in the first section and also define the associativity of the operations

Step 3: Mention the grammar productions and the action for each production.

Step 4: $$ refer to the top of the stack position while $1 for the first value, $2 for the second value in the stack.

Step 5: Call yyparse() to initiate the parsing process.

Step 6: yyerror() function is called when all productions in the grammar in second section doesn't match to the

input statement.

PROGRAM:

```
%{
#include<stdio.h>
#include "y.tab.h"
%}
%%
[a
-zA
-Z][0
-9a
-zA
-Z]* {return ID;}
[0
-9]+ {return DIG;}
[
\t]+ {;}
. {return yytext[0];} \n {return 0;}
%%
int yywrap() {
return 1;
}
//art_expr.y
%{
#include<stdio.h>
%}
%token ID DIG
%left '+"-'
```

```
%left '*'/'
%right UMINUS
%%
stmt:expn ;
expn:expn'+'expn
|expn'
-'expn
|expn'*'expn
|expn'/'expn
|'
-'expn %prec UMINUS
|'('expn')'
|DIG
|ID
26
;
%%
int main()
{
printf("Enter the Expression \n");
yyparse();
printf("valid Expression \n");
return 0;
}
int yyerror()
{
printf("Invalid Expression");
```

exit(0);

}

OUTPUT:

RESULT:

Thus the program to recognize a valid arithmetic expression that uses operator +, - , * and / using YACC tool was executed and verified successfully.

EX.NO: 3(b)	**RECOGNIZING A VALID VARIABLE**
DATE:	

AIM:

To write a program to recognize a valid variable which starts with a letter followed by any number of letters or digits using YACC tool.

ALGORITHM:

LEX

Step 1: Declare the required header file and variable declaration with in '%{' and '%}'.

Step 2: LEX requires regular expressions or patterns to identify token of lexemes for recognize a valid variable.

Step 3: Lex call yywrap() function after input is over. It should return 1 when work is done or should return 0

when more processing is required.

YACC

Step 1: Declare the required header file and variable declaration with in '%{' and '%}'.

Step 2: Define tokens in the first section and also define the associativity of the operations

Step 3: Mention the grammar productions and the action for each production.

Step 4: $$ refer to the top of the stack position while $1 for the first value, $2 for the second value in the stack.

Step 5: Call yyparse() to initiate the parsing process.

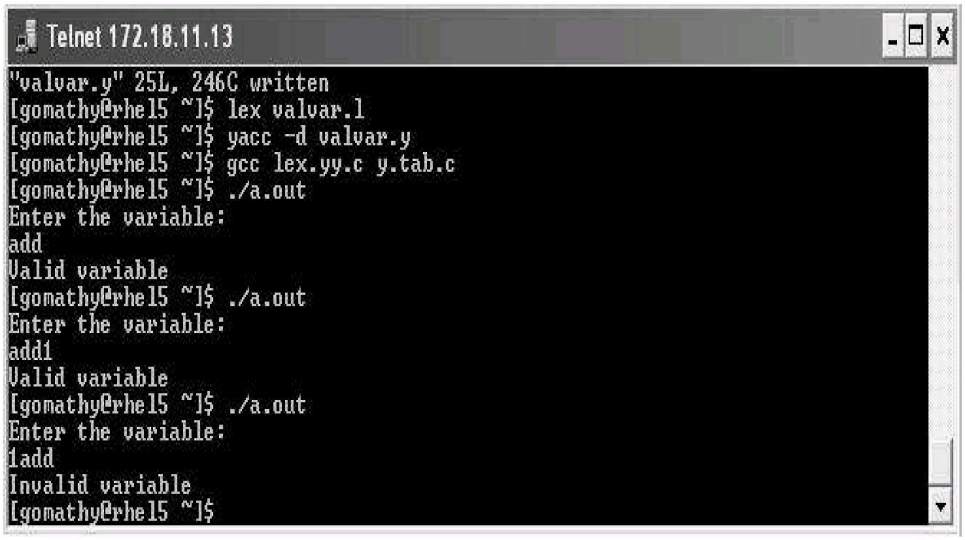
Step 6: yyerror() function is called when all productions in the grammar in second section doesn't match to the input statement

PROGRAM:

```
%{
#include "y.tab.h"
%}
%%
[a-z A-Z] {return LET;}
[0-9] {return DIG;}
. {return yytext[0];} \n {return 0;}
%%
int yywrap() {
return 1;
}
%{
#include<stdio.h>
%}
%token LET DIG
%%
variable:var ;
var:var DIG
|var LET
|LET ;
%%
int main() {
printf("Enter the variable:\n");
yyparse();
printf("Valid variable \n");
return 0;}
```

```
int yyerror()
{
printf("Invalid variable \n");
exit(0);
}
```

OUTPUT:



RESULT:

Thus, the program to recognize a valid variable which starts with a letter followed by any number of letters or digits using YACC tool was executed and verified successfully.

EX.NO: 3(c)	**RECOGNIZING A CONTROL STRUCTURE SYNTAX C LANGUAGE**
DATE:	

AIM:

To write a program to recognize a valid control structure syntax of c language using YACC tool.

ALGORITHM:

LEX

Step 1: Declare the required header file and variable declaration with in '%{' and '%}'.

Step 2: LEX requires regular expressions or patterns to identify token of lexemes for recognize a valid variable.

Step 3: Lex call yywrap() function after input is over. It should return 1 when work is done or should return 0

when more processing is required.

YACC

Step 1: Declare the required header file and variable declaration with in '%{' and '%}'.

Step 2: Define tokens in the first section and also define the associativity of the operations

Step 3: Mention the grammar productions and the action for each production.

Step 4: $$ refer to the top of the stack position while $1 for the first value, $2 for the second value in the stack.

Step 5: Call yyparse() to initiate the parsing process.

Step 6: yyerror() function is called when all productions in the grammar in second section doesn't match to the input statement

PROGRAM:

YACC PART:

CODE:(nif.y)

```
%{
#include<stdio.h>
#include<stdlib.h>
int count=0;
%}
%%
stmt: if_stmt NL {printf("No. of nested if statements=%d\n",count);exit(0);}
;
if_stmt : IF('cond')"{'if_stmt'}' {count++;}|S
;
cond: x RELOP x
;
x:ID | NUMBER
;
%%
int yyerror(char *msg){
printf("the statement is invalid\n");
exit(0);}
main(){
printf("enter the statement\n");
yyparse();}
```

LEX PART:

CODE:(nif.l)

```
%{
```

```
#include "y.tab.h"
%}
%%
"if" {return IF;}
[sS][0-9]* {return S;}
"<"|">"|"=="|"<="|">="|"!=" {return RELOP;}
[0-9]+ {return NUMBER;}
[a-z][a-zA-Z0-9_]* {return ID;}
\n {return NL;}
. {return yytext[0];}
%%
```

OUTPUT:

```
 yacc -d nif.y
lex nif.l
cc y.tab.c lex.yy.c -ly -ll
./a.out
enter the statement
if(a<b){s}
No. of nested if statements=1
 ./a.out
enter the statement
if(a>b){if(a>b){s}}
No. of nested if statements=2
```

RESULT:

Thus, the program to recognize a valid control structure syntax of c language using YACC tool was executed and verified successfully.

EX.NO: 3(d)	**IMPLEMENTATION OF CALCULATOR USING LEX AND YACC**
DATE:	

AIM:

To write a LEX and YACC program to implement calculator.

ALGORITHM:

Step 1: Start the program.

Step 2: In the declaration part of lex, includes declaration of regular definitions as digit.

Step 3: In the translation rules part of lex, specifies the pattern and its action that is to be executed whenever a

lexeme matched by pattern is found in the input in the call.

Step 4: By use of Yacc program, all the Arithmetic operations are done such as +,-,*,/.

Step 5: Display error is persist.

Step 6: Provide the input.

Step 7: Verify the output.

Step 8: End.

PROGRAM:

cal.l

```
DIGIT [0-9]+
%option noyywrap
%%
{DIGIT} { yylval=atof(yytext); return NUM;}
\n|. { return yytext[0];}
%%
```

cal.y

```
%{
#include<ctype.h>
#include<stdio.h>
#define YYSTYPE double
%}
%token NUM
%left '+' '-'
%left '*' '/'
%right UMINUS
%%
Statment:E { printf("Answer: %g \n", $$); }
```

|Statment '\n';

E : E'+'E { $$ = $1 + $3; }

| E'-'E { $$=$1-$3; }

| E'*'E { $$=$1*$3; }

| E'/'E { $$=$1/$3; }

| NUM;

%%

OUTPUT:

"cal2.y" 59L, 1186C written

[exam01@Cselab3 ~]$ lex cal2.l

[exam01@Cselab3 ~]$ yacc yaccal2.y

[exam01@Cselab3 ~]$ cc y.tab.c

[exam01@Cselab3 ~]$ ./a.out

Enter the expression:2+2

Answer: 4

RESULT:

Thus the program for implementing calculator using LEX and YACC is executed and verified.

EX.NO: 4	**GENERATE THREE ADDRESS CODE USING LEX AND YACC**
DATE:	

AIM:

To write a LEX and YACC program to generate three address code

ALGORITHM:

Step 1: Start the program.

Step 2: In the declaration part of lex, includes declaration of regular definitions as digit.

Step 3: In the translation rules part of lex, specifies the pattern and its action that is to be executed whenever a

lexeme matched by pattern is found in the input in the call.

Step 4: By use of Yacc program ,all the process to generate three address code

Step 5: Display error is persist.

Step 6: Provide the input.

Step 7: Verify the output.

PROGRAM:

lex.l

```
%{
   #include<stdio.h>
   #include"y.tab.h"
   int k=1;
%}
%%
[0-9]+ {
yylval.dval=yytext[0];
return NUM;
}
\n {return 0;}
. {return yytext[0];}
%%
void yyerror(char* str)
{
       printf("\n%s",str);
}
char *gencode(char word[],char first,char op,char second)
{
   char temp[10];
   sprintf(temp,"%d",k);
   strcat(word,temp);
   k++;
   printf("%s = %c %c %c\n",word,first,op,second);
```

```
    return word; //Returns variable name like t1,t2,t3... properly
}
int yywrap()
{
    return 1;
}
main()
{
        yyparse();
        return 0;
}
yacc.y
%{
#include<stdio.h>
int aaa;
%}
%union{
    char dval;
}
%token <dval> NUM
%type <dval> E
%left '+' '-'
%left '*' '/' '%'
%%
statement : E {printf("\nt = %c \n",$1);}
          ;
E : E '+' E{
```

```
        char word[]="t";
        char *test=gencode(word,$1,'+',$3);
        $$=test;
    }
  | E '-' E
    {
        char word[]="t";
        char *test=gencode(word,$1,'-',$3);
        $$=test;
    }
  | E '%' E
    {
        char word[]="t";
        char *test=gencode(word,$1,'%',$3);
        $$=test;
    }
  | E '*' E
    {
        char word[]="t";
        char *test=gencode(word,$1,'*',$3);
        $$=test;
    }
  | E '/' E
    {
        char word[]="t";
        char *test=gencode(word,$1,'/',$3);
        $$=test;
```

```
    }
  | '(' E ')'
    {
       $$=$2;
    }
  | NUM
    {
       $$=$1;
    }
  ;
%%
```

OUTPUT:

EXPRESION: (2+3)*5

t1= 2 + 3

t2= t1 * 5

RESULT:

Thus, the program to generate three address code using LEX and YACC has been successfully executed and output is verified.

EX.NO: 5	IMPLEMENTATION OF TYPE CHECKING
DATE:	

AIM:

To write a C program to implement type checking.

ALGORITHM:

Step 1: Start the program for type checking of given expression

Step 2: Read the expression and declaration

Step 3: Based on the declaration part define the symbol table

Step 4: Check whether the symbols present in the symbol table or not. If it is found in the symbol table it displays “Label already defined”.

Step 5: Read the data type of the operand 1, operand 2 and result in the symbol table.

Step 6: If the both the operands’ type are matched then check for result variable. Else, print “Type mismatch”.

Step 7: If all the data type are matched then displays “No type mismatch”.

PROGRAM:

```
#include<stdio.h>
#include<string.h>
#include<conio.h>
int count=1,i=0,j=0,l=0,findval=0,k=0,kflag=0;
char key[4][12]= {"int","float","char","double"};
char dstr[100][100],estr[100][100];
char token[100],resultvardt[100],arg1dt[100],arg2dt[100];
void entry();
int check(char[]);
int search(char[]);
void typecheck();
struct table {
char var[10];
char dt[10];
};
struct table tbl[20];
void main() {
clrscr();
printf("
\n IMPLEMENTATION OF TYPE CHECKING
\n");
printf("
\n DECLARATION
\n
\n");
do{
```

```
printf("
\t");
gets(dstr[i]);
i++;
} while(strcmp(dstr[i
-1],"END"));
printf("
\n EXPRESSION
\n
\n");
do{
printf("
\t");
gets(estr[l]);
l++;
}while(strcmp(estr[l
-1],"END"));
i=0;
printf("
\n SEMANTIC ANALYZER(TYPE CHECKING):
\n");
while(strcmp(dstr[i],"END")) {
entry();
printf("
\n");
i++;
}
```

```
l=0;
while(strcmp(estr[l],"END")) {
typecheck()
;
printf("
\n");
42
l++;
}
printf("
\n PRESS ENTER TO EXIT FROM TYPE CHECKING
\n");
getch();
}
void entry() {
j=0;
k=0;
memset(token,0,sizeof(token));
while(dstr[i][j]!=' ') {
token[k]=dstr[i][j];
k++;
j++;
}
kflag=check(token);
if(kflag==1) {
strcpy(tbl[count].dt,token);
k=0;
```

```
memset(token,0,strlen(token));
j++;
while(dstr[i][j]!=';') {
token[k]=dstr[i][j];
k++;
j++;
}
findval=search(token);
if(findval==0) {
strcpy(tbl[count].var,token);
}
else {
printf("The variable %s is already declared",token);
}
kflag=0;
count++;
}
else {
printf("Enter valid datatype
\n");
}
}
void typecheck() {
memset(token,0,strlen(token));
j=0;
k=0;
while(estr[l][j]!='=')
```

```
{
token[k]=estr[l][j];
k++;
j++;
}
findval=search(token);
if(findval>0) {
strcpy(resultvardt,tbl[findval].dt);
findval=0;
}
else {
printf("Undefined Variable
\n");
}
k=0;
memset(token,0,strlen(token));
j++;
while(((estr[l][j]!='+')&&(estr[l][j]!='
-')&&(estr[l][j]!='*')&&(estr[l][j]!='/')))
{
token[k]=estr[l][j];
k++;
j++;
}
findval=search(token);
if(findval>0) {
```

```
strcpy(arg1dt,tbl[findval].dt);
findval=0;
}
else {
printf("Undefined Variable
\n");
}
k=0;
memset(token,0,strlen(token));
j++;
while(estr[l][j]!=';') {
token[k]=estr[l][j];
k++;
j++;
}
findval=search(token);
if(findval>0) {
strcpy(arg2dt,tbl[findval].dt);
findval=0;
}
else {
printf("Undefined Variable
\n");
}
if(!strcmp(arg1dt,arg2dt)) {
```



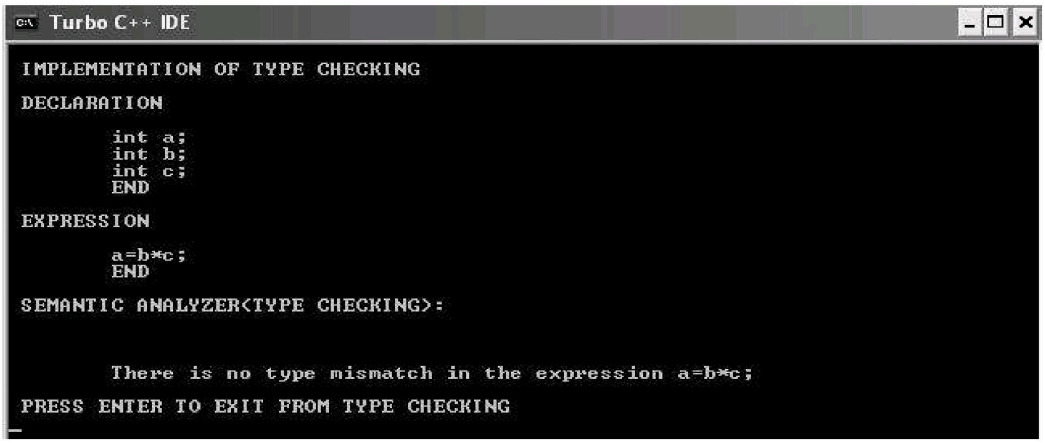
```
if(!strcmp(resultvardt,arg1dt))
```

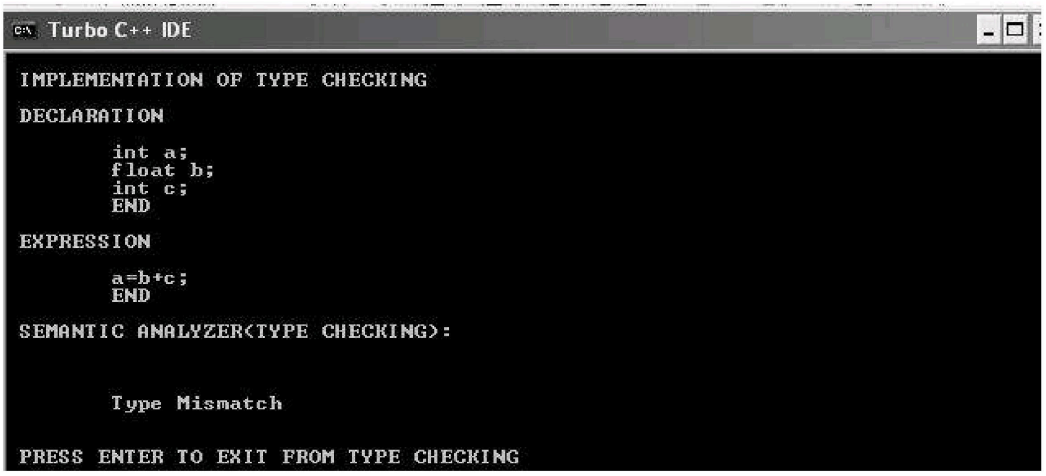
```
{
printf("\tThere is no type mismatch in the expression %s ",estr[l]);
}
else
{
printf("\tLvalue and Rvalue should be same\n");
}
}
else
{
printf("\tType Mismatch\n");
}
}
int search(char variable[])
{
int i;
for(i=1;i<=count;i++)
{
if(strcmp(tbl[i].var,variable) == 0)
{
return i;
}
}
return 0;
}
int check(char t[])
{
```

```
int in;
for(in=0;in<4;in++)
{
if(strcmp(key[in],t)==0){
return 1;}}
return 0;}
```

45

OUTPUT:





RESULT:

Thus, the program for type checking is executed and verified.

EX.NO: 6	IMPLEMENTATION OF CODE OPTMIZATION TECHNIQUES
DATE:	

AIM:

To write a C program to implement Simple Code Optimization Techniques.

ALGORITHM:

Step 1: Read the un-optimized input block.

Step 2: Identify the types of optimization.

Step 3: Optimize the input block.

Step 4: Print the optimized input block.

Step 5: Execute the same with different set of un-optimized inputs and obtain the optimized input block.

PROGRAM:

```
#include<stdio.h>
#include<conio.h>
#include<ctype.h>
void main() {
 char a[25][25],u,op1='*',op2='+',op3='/',op4='-';
 int p,q,r,l,o,ch,i=1,c,k,j,count=0;
 FILE *fi,*fo;
 // clrscr();
 printf("Enter three address code");
 printf("\nEnter the ctrl-z to complete:\n");
 fi=fopen("infile.txt","w");
 while((c=getchar())!=EOF)
 fputc(c,fi);
 fclose(fi);
 printf("\n Unoptimized input block\n");
 fi=fopen("infile.txt","r");
 while((c=fgetc(fi))!=EOF)
{
k=1;
while(c!=';'&&c!=EOF) {
a[i][k]=c;
printf("%c",a[i][k]);
k++;
c=fgetc(fi);
}
printf("\n");
```

```
i++;
}
 count=i;
 fclose(fi);
 i=1;
 printf("\n Optimized three address code");
 while(i<count)
63
 {
if(strcmp(a[i][4],op1)==0&&strcmp(a[i][5],op1)==0)
{
 printf("\n type 1 reduction in strength");
if(strcmp(a[i][6],'2')==0)
{
 for(j=1;j<=4;j++)
 printf("%c",a[i][j]);
 printf("%c",a[i][3]);
}
}
else if(isdigit(a[i][3])&&isdigit(a[i][5]))
{
printf("\n type2 constant floding");
p=a[i][3];
q=a[i][5];
if(strcmp(a[i][4],op1)==0)
 r=p*q;
if(strcmp(a[i][4],op2)==0)
```

```
 r=p+q;
if(strcmp(a[i][4],op3)==0)
 r=p/q;
if(strcmp(a[i][4],op4)==0)
 r=p-q;
for(j=1;j<=2;j++)
 printf("%c",a[i][j]);
printf("%d",r);
printf("\n");
}
else if(strcmp(a[i][5],'0')==0||strcmp(a[i][5],'1')==0)
{
 cprintf("\n type3 algebraic expression elimation");
if((strcmp(a[i][4],op1)==0&&strcmp(a[i][5],'1')==0)||(strcmp(a[i][4],op3)==0&
&strcmp(a[i][5],'1')==0))
 {
 for(j=1;j<=3;j++)
64
 printf("%c",a[i][j]);
 printf("\n"); }
 else
 printf("\n sorry cannot optimize\n");
}
else
{
 printf("\n Error input");
}
```

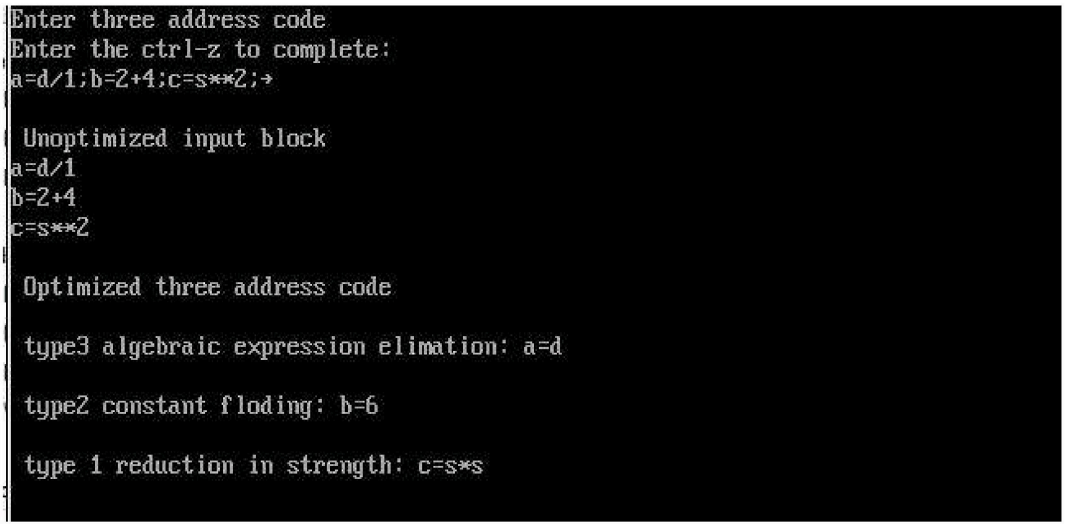
```
 i++;
 }
 getch();
}
```

65

infile.txt

a=d/1; b=2+4; c=s**2;

OUTPUT:



RESULT:

Thus the C program for implementation of Code optimization was executed successfully.

EX.NO: 7	**IMPLEMENTATION OF BACK END OF THE COMPILER**
DATE:	

AIM:

To write a ‘C’ program to generate the machine code for the given intermediate code.

ALGORITHM:

Step1: Get the input expression from the user.

Step2: The given expression is transformed into tokens.

Step3: Display the assembly code according to the operators present in the given expression.

Step4: Use the temporary registers (R0, R1) while storing the values in assembly code programs.

PROGRAM:

```
/* CODE GENERATOR */
#include<stdio.h>
#include<string.h>
int count=0,i=0,l=0;
char str[100][100];
void gen();
void main()
{
clrscr();
printf("\n CODE GENERATOR \n");
printf("\n ENTER THREE ADDRESS CODE \n\n");
do
{
printf("\t");
gets(str[i]);
i++;
} while(strcmp(str[i-1],"QUIT"));
i=0;
printf("\n ASSEMBLY LANGUAGE CODE: \n");
while(strcmp(str[i-1],"QUIT"))
{
 gen();
printf("\n");
 i++;
}
printf("\n PRESS ENTER TO EXIT FROM CODE GENERATOR\n");
```

```
getch();
}
void gen()
{
int j;
printf("\n");
for(j=strlen(str[i])-1;j>=0;j--)
{
 char reg='R';
 if(isdigit(str[i][j])||(isalpha(str[i][j]))|| str[i][j]=='+'||str[i][j]=='-
'||str[i][j]=='*'||str[i][j]=='/'||str[i][j]=='
'||str[i][j]=='|'||str[i][j]=='&'||str[i][j]==':'||str[i][j]=='=')
 {
switch(str[i][j])
{
case '+':
printf("\n\t MOV\t%c,%c%d",str[i][j-1],reg,count);
59
printf("\n\t ADD\t%c,%c%d",str[i][j+1],reg,count);
break;
case '-':
printf("\n\t MOV\t%c,%c%d",str[i][j-1],reg,count);
printf("\n\t SUB\t%c,%c%d",str[i][j+1],reg,count);
break;
case '*':
printf("\n\t MOV\t%c,%c%d",str[i][j-1],reg,count);
printf("\n\t MUL\t%c,%c%d",str[i][j+1],reg,count);
```

```
break;
case '/':
printf("\n\t MOV\t%c,%c%d",str[i][j-1],reg,count);
printf("\n\t DIV\t%c,%c%d",str[i][j+1],reg,count);
break;
case '|':
printf("\n\t MOV\t%c,%c%d",str[i][j-1],reg,count);
printf("\n\t OR\t%c,%c%d",str[i][j+1],reg,count);
break;
case '&':
printf("\n\t MOV\t%c,%c%d",str[i][j-1],reg,count);
printf("\n\t AND\t%c,%c%d",str[i][j+1],reg,count);
break;
case ':':
if(str[i][j+1]=='=')
{
 printf("\n\t MOV\t%c%d,%c",reg,count,str[i][j-1]);
 count++;
}
else
{
 printf("\n syntax error...\n");
}
break;
default:
break;
}
```

```
}
 else printf("\n Error\n");
}
}
```

60

OUTPUT:

```
CODE GENERATOR
ENTER THREE ADDRESS CODE
 A:=B+C
 D:=E/F
 QUIT
ASSEMBLY LANGUAGE CODE:
 MOV B,R0
 ADD C,R0
 MOV R0,A
 MOV E,R1
 DIV F,R1
 MOV R1,D
PRESS ENTER TO EXIT FROM CODE GENERATOR
```

RESULT:

Thus, the program for generation of Machine Code for the given intermediate code is executed and verified.