ANNA UNIVERSITY

REGIONAL CAMPUS

COIMBATORE – 641 046

CS3501 - COMPILER LABORATORY

RECORD NOTE BOOK

ANNA UNIVERSITY

REGIONAL CAMPUS

COIMBATORE – 641 046

BONAFIDE CERTIFICATE

Register No | |

This is to certify that this is the bonafide record of the work done by Mr/Ms _____ in **CS3501 – COMPILER LABORATORY,** ANNA UNIVERSITY REGIONAL CAMPUS COIMBATORE during FIFTH SEMESTER in the academic year 2023-2024.

Staff in Charge **Head of the Department**

Submitted for the practical examination held on _____

Internal Examiner **External Examiner**

CONTENTS

Ex. No.	Exp.Date	Name of the Experiment	Page No.	Sign
1				
2				
3				
4				
5				
6				
7				
8				
9				
10				

Ex.No:1A **Develop a Lexical Analyzer to Recognize Patterns in C**
Date :

AIM:

To develop Lexical analyzer to recognize 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.

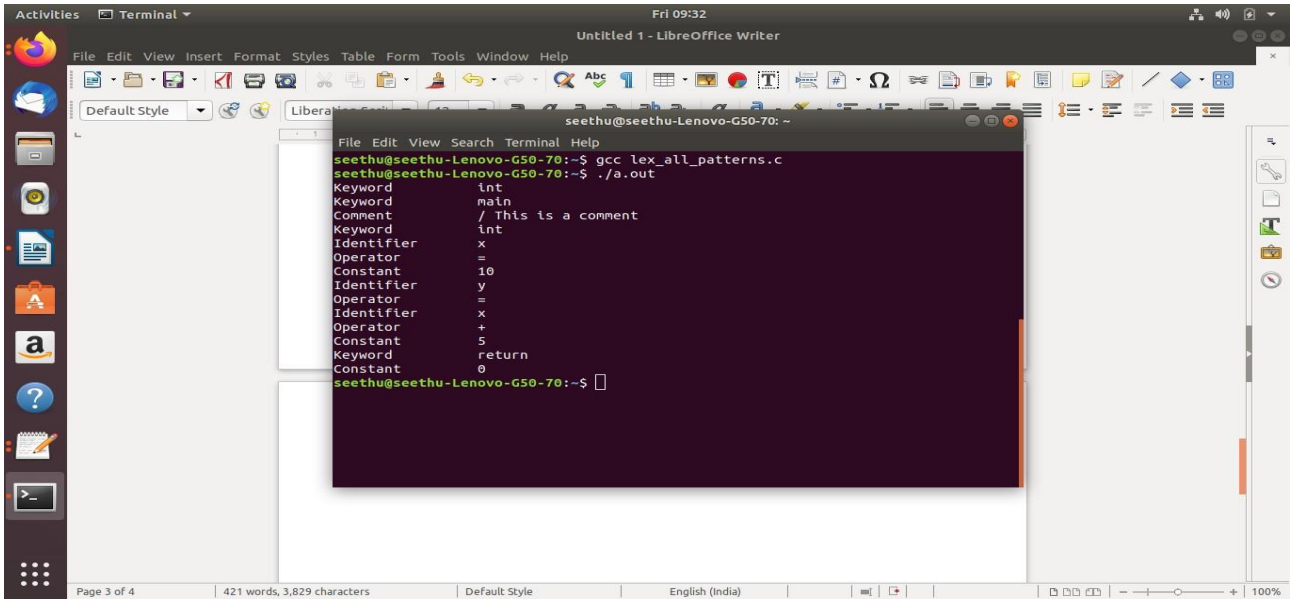
PROGRAM:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#define MAX_TOKEN_LENGTH 100
// Token types
enum TokenType {
 KEYWORD,
 IDENTIFIER,
 CONSTANT,
 COMMENT,
 OPERATOR,
 SPECIAL_SYMBOL
};
// Token structure
struct Token {
 enum TokenType type;
 char value[MAX_TOKEN_LENGTH];
};
// Function to print tokens
```

```
void printToken(struct Token token) {
printf("%s \t %s\n", (token.type==0) ? "Keyword": (token.type==1) ? "Identifier" : (token.type==2)
? "Constant" :
 (token.type==3) ? "Comment" : (token.type=4) ? "Operator" : "Special_symbol",
token.value);
}
// Function to perform lexical analysis
void lexer(char *input_code) {
 struct Token token;
 char *delimiters = " \t\n"; // Whitespace characters
 while (*input_code != '\0') {
 // Skip whitespace
 if (strchr(delimiters, *input_code) != NULL) {
 input_code++;
 continue;
 }
 // Check for keywords
 if (isalpha(*input_code) || *input_code == '_') {
 token.type = IDENTIFIER;
 int i = 0;
 while (isalnum(*input_code) || *input_code == '_') {
 token.value[i++] = *input_code++;
 }
 token.value[i] = '\0';
 // Check if the identifier is a keyword
 if (strcmp(token.value, "int") == 0 ||
 strcmp(token.value, "float") == 0 ||
 strcmp(token.value, "char") == 0 ||
 strcmp(token.value, "if") == 0 ||
 strcmp(token.value, "else") == 0 ||
 strcmp(token.value, "while") == 0 ||
 strcmp(token.value, "for") == 0 ||
 strcmp(token.value, "return") == 0 ||
 strcmp(token.value, "main") == 0) {
 token.type = KEYWORD;
 }
 printToken(token);
 continue;
 }
 // Check for constants (integers)
 if (isdigit(*input_code)) {
 token.type = CONSTANT;  int i = 0;
 while (isdigit(*input_code)) {
token.value[i++] = *input_code++;
```

```
}
token.value[i] = '\0';
printToken(token);
continue;
}
// Check for comments
if (*input_code == '/') {
input_code++;
if (*input_code == '/') {
token.type = COMMENT;
int i = 0;
while (*input_code != '\n' && *input_code != '\0') {
token.value[i++] = *input_code++;
}
token.value[i] = '\0';
printToken(token);
continue;
} else if (*input_code == '*') {
token.type = COMMENT;
int i = 0;
while (!(input_code[0] == '*' && input_code[1] == '/') && *input_code != '\0') {
token.value[i++] = *input_code++;
}
token.value[i] = '\0';
printToken(token);
input_code += 2; // Skip the closing */
continue;
}
}
// Check for operators
if (*input_code == '+' || *input_code == '-' || *input_code == '*' || *input_code == '/' ||
*input_code == '=' || *input_code == '<' || *input_code == '>') {
token.type = OPERATOR;
token.value[0] = *input_code++;
token.value[1] = '\0';
printToken(token);
continue;
}
// Move to the next character if none of the patterns match
input_code++;
}}
```

OUTPUT:



RESULT:

Ex.No 1B **Create the Symbol Table**
Date:

AIM:

To write a C program to implement a symbol table.

ALGORITHM:

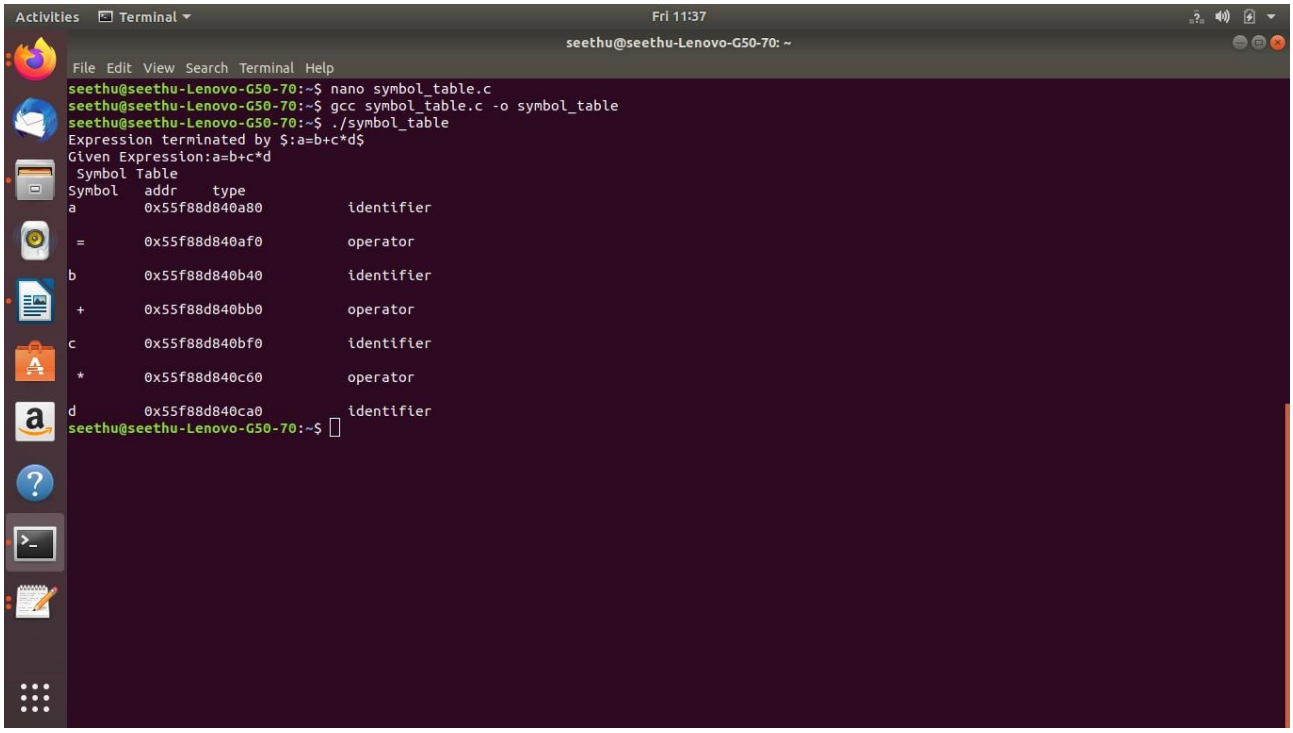
1. Start the Program.
2. Get the input from the user with the terminating symbol „$“.
3. Allocate memory for the variable by dynamic memory allocation function.
4. If the next character of the symbol is an operator then only the memory is allocated.
5. While reading, the input symbol is inserted into symbol table along with its memory address.
6. The steps are repeated till”$”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<ctype.h>
#include<stdlib.h>
#include<string.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:");
i=0;
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 %p \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 %p \t operator\n",ch,p);
 x++;
 j++;
}
}
}
}
```

OUTPUT:



RESULT:

Ex.No 2 **Implementation of Lexical Analyzer using LEX Tool**

Date:

AIM:

To Implement a Lexical Analyzer using Lex tool.

ALGORITHM:

1. Start the program
2. Lex program consists of three parts.
3. Declaration %%
4. Translation rules %%
5. Auxiliary procedure.
6. The declaration section includes declaration of variables, main test, constants and regular
7. Definitions.
8. Translation rule of lex program are statements of the form
9. P1{action}
10.P2{action}
11…..
12…..
13.Pn{action}
14.Write program in the vi editor and save it with .1 extension.
15.Compile the lex program with lex compiler to produce output file as lex.yy.c.
16.Eg. $ lex filename.1
17. $gcc lex.yy.c-11
18.Compile that file with C compiler and verify the output.

PROGRAM:

```
//Implementation of Lexical Analyzer using Lex tool
%{
int COMMENT=0;
%}
identifier [a-zA-Z][a-zA-Z0-9]*
%%
#.* {printf("\n%s is a preprocessor directive",yytext);}
int |
float |
char |
double |
while |
for |
struct |
typedef |
```

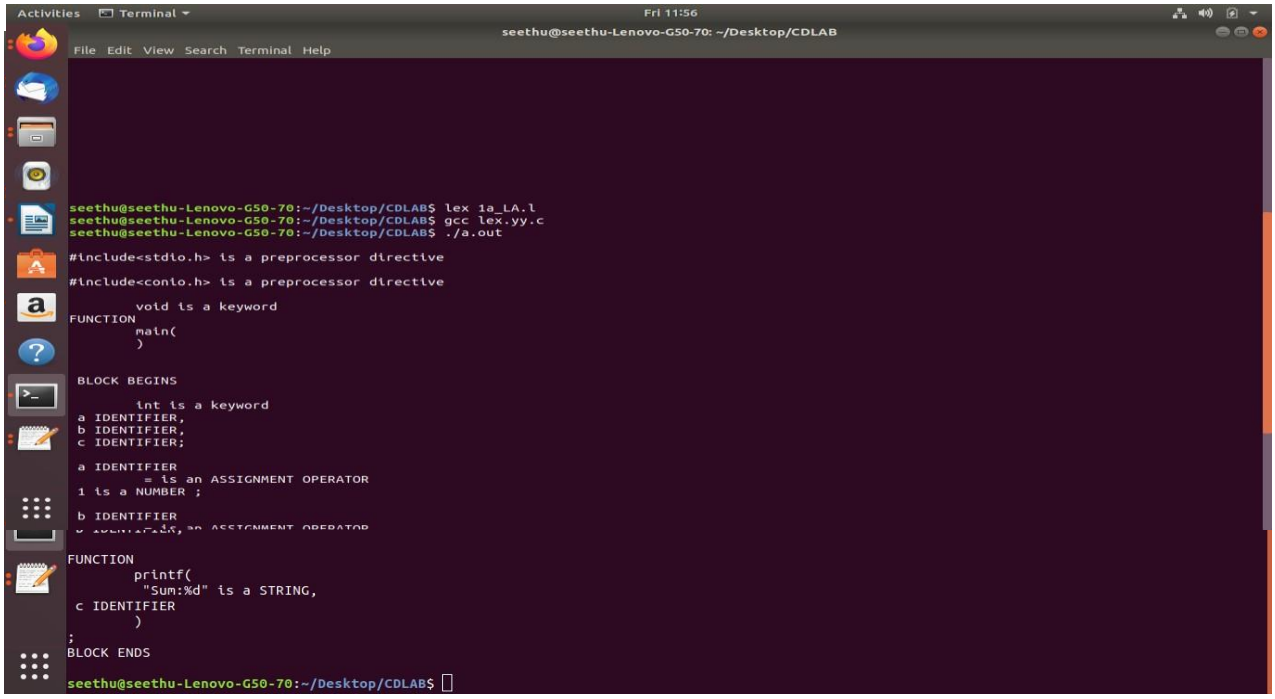
```
do |
if |
break |
continue |
void |
switch |
return |
else |
goto {printf("\n\t%s is a keyword",yytext);}
"/*" {COMMENT=1;}{printf("\n\t %s is a COMMENT",yytext);}
{identifier}\( {if(!COMMENT)printf("\nFUNCTION \n\t%s",yytext);}
\{ {if(!COMMENT)printf("\n BLOCK BEGINS");}
\} {if(!COMMENT)printf("BLOCK ENDS ");}
{identifier}(\[[0-9]*\])? {if(!COMMENT) printf("\n %s IDENTIFIER",yytext);}
\".*\" {if(!COMMENT)printf("\n\t %s is a STRING",yytext);}
[0-9]+ {if(!COMMENT) printf("\n %s is a NUMBER ",yytext);}
\)(\:)? {if(!COMMENT)printf("\n\t");ECHO;printf("\n");}
\( ECHO;
= {if(!COMMENT)printf("\n\t %s is an ASSIGNMENT OPERATOR",yytext);}
\<= |
\>= |
\< |
== |
\> {if(!COMMENT) printf("\n\t%s is a RELATIONAL OPERATOR",yytext);}
%%
int main(int argc, char **argv)
{
FILE *file;
file=fopen("var.c","r");
if(!file)
{
printf("could not open the file");
exit(0);
}
yyin=file;
yylex();
printf("\n");
return(0);
}
int yywrap()
{
return(1);
}
```

Input File

var.c

```
#include<stdio.h>
#include<conio.h>
void main()
{
int a,b,c;
a=1;
b=2;
c=a+b;
printf("Sum:%d",c);
}
```

OUTPUT:



RESULT:

Ex.No :3A **Check Valid Arithmetic Expression or Not**
Date:

AIM:

To write a C program that validates whether the given arithmetic expression is valid or not.

ALGORITHM:

1.Start the program
2.Write the code for lex file 3a.l
3.write the code for the yacc file 3a.y
4.Execute both lex and yacc file
5.The user is prompted to enter an expression
6.The expression is tokenized by the user,producing a sequence of tokens
7.The parser processes the token based on grammar rules and builds a parse tree.
8.If the expression is valid “Valid expression is printed” otherwise invalid expression is printed.
9.Stop the program.

PROGRAM:

lex file 3a.l

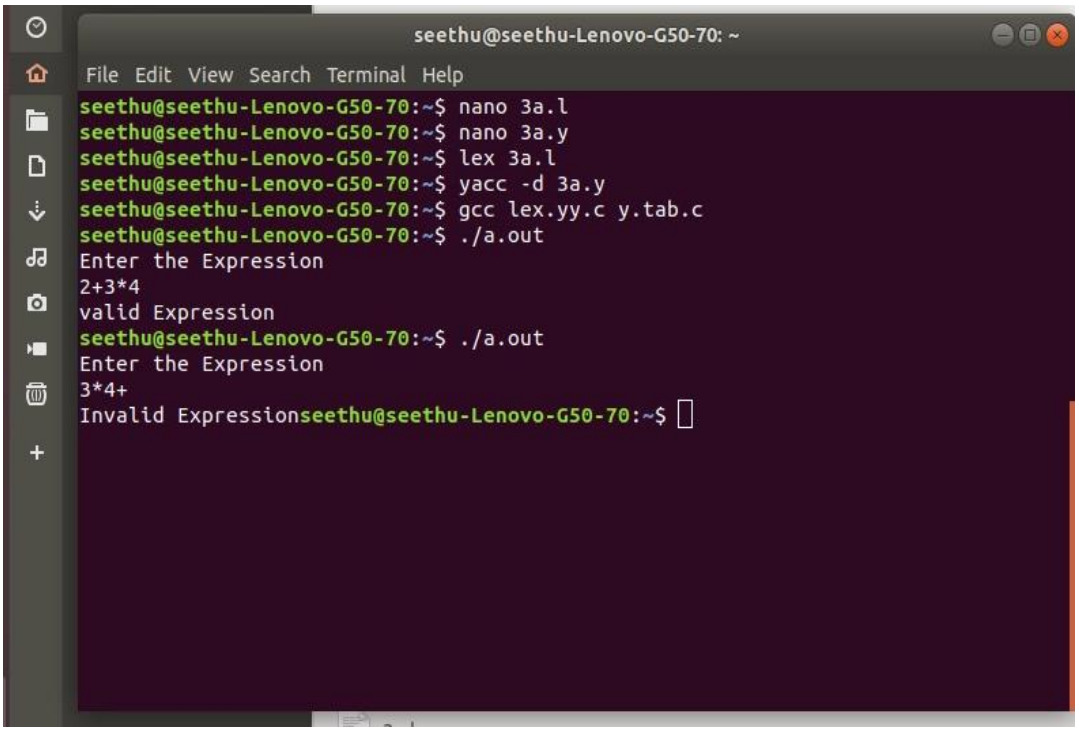
```
%{
#include<stdio.h>
#include "y.tab.h"
%}
%%
[a-zA-Z][0-9a-zA-Z]* {return ID;}
[0-9]+ {return DIG;}
[ \t]+ {;}
. {return yytext[0];}
\n {return 0;}
%%
int yywrap()
{
return 1;
}
```

yacc file 3a.y

```
%{
#include <stdio.h>
#include <stdlib.h>
int yylex(void);
void yyerror(char const *);
%}
```

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

OUTPUT:



RESULT:

Ex.No: 3B **Check the Variable is Valid or Not**
Date:

AIM:
To write a program to implement whether the given variable is valid or not.

ALGORITHM:
1.Start the program.
2.In the main function, prompt the user to enter a string .
3.Call the yyparse function to parse the input string.
4.Define the grammar rules in the yyparse function.
5.If the parsing is successful print valid.
6.If the are syntax error, the yyerror function is called to handle the syntax error and it prints invalid.
7.Stop the program.

PROGRAM:

var.l

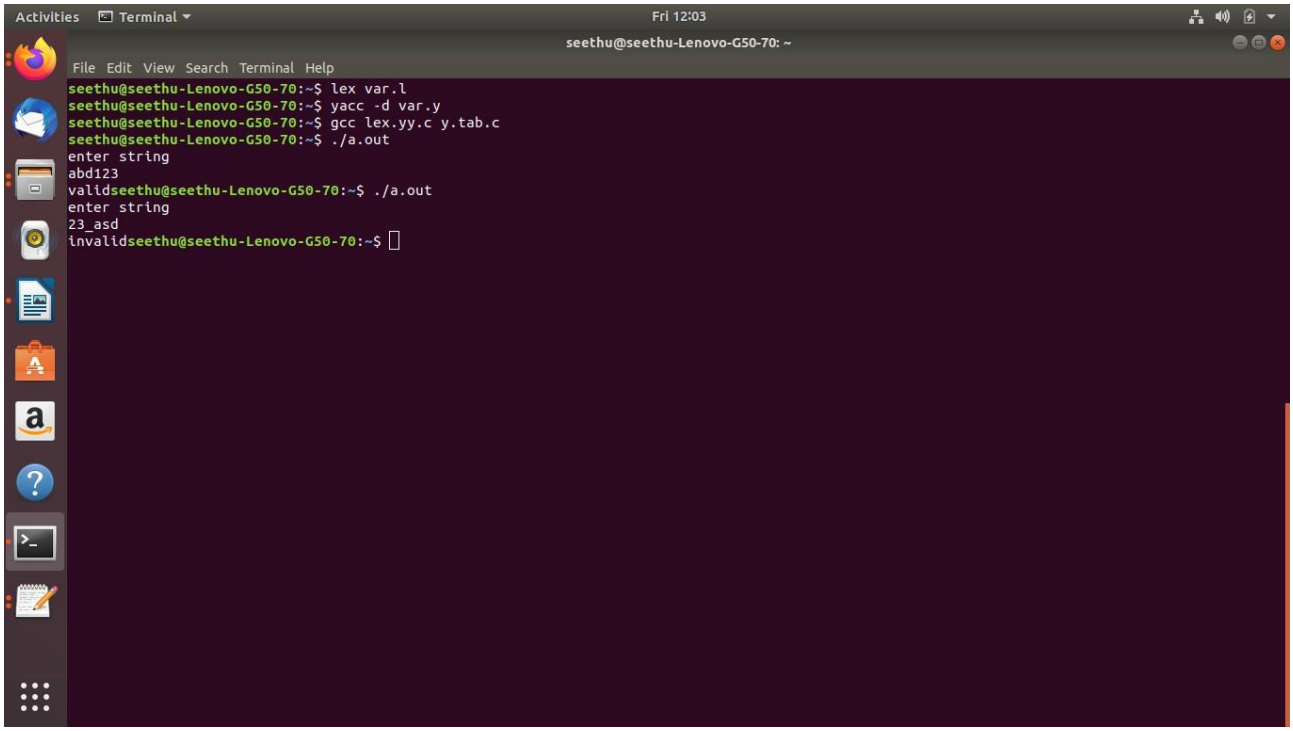
```
%{
#include "y.tab.h"
%}
%%
[0-9]+ {return DIGIT;}
[a-zA-Z]+ {return LETTER;}
[ \t] {;}
\n { return 0;}
. {return yytext[0];}
%%
int yywrap() {
 // Return 1 to indicate the end of input
 return 1;
}
```

var.y

```
%{
```

```
#include<stdio.h>
#include<stdlib.h>
int yylex(void);
void yyerror(char const *);
%}
%token DIGIT LETTER
%%
stmt:A
 ;
A: LETTER B
;
B: LETTER B
| DIGIT B
| LETTER
| DIGIT
;
%%
void main(){
printf("enter string \n");
yyparse();
printf("valid");
exit(0);
}
void yyerror(const char *s)
{
printf("invalid");
exit(0);
}
```

OUTPUT:



RESULT:

Ex.No: 3C **Implement the Calculator Using LEX and YACC**
DATE:

AIM:

To write a program that implements the calculator using lex and yacc.

ALGORITHM:

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

PROGRAM:

calc.l

```
%{
#include <stdio.h>
#include "y.tab.h"
int c;
extern int yylval;
%}
%%
" " ;
[a-z] {
 c = yytext[0];
 yylval = c - 'a';
 return(LETTER);
 }
[0-9] {
 c = yytext[0];
 yylval = c - '0';
 return(DIGIT);
 }
[^a-z0-9\b] {
 c = yytext[0];
 return(c);
 }
```

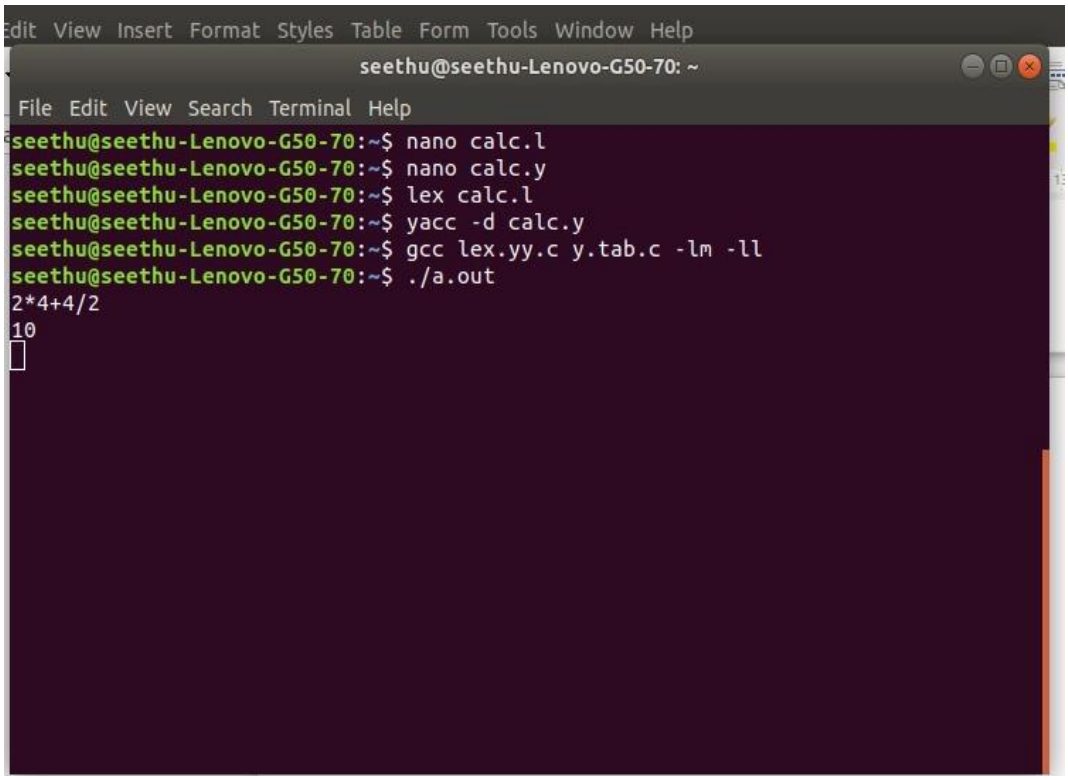
calc.y

```
%{
#include <stdio.h>
int yylex(void);
void yyerror(char const *);
int regs[26];
int base;
%}
%start list
%token DIGIT LETTER
%left '|'
%left '&'
%left '+' '-'
%left '*' '/' '%'
%left UMINUS /*supplies precedence for unary minus */
%% /* beginning of rules section */
list: /*empty */
 |
 list stat '\n'
 |
 list error '\n'
 {
 yyerrok;
 }
 ;
stat: expr
 {
 printf("%d\n",$1);
 }
 |
 LETTER '=' expr
 {
 regs[$1] = $3;
 }
 ;
expr: '(' expr ')'
 {
 $$ = $2;
 }
 |
 expr '*' expr
 {
 $$ = $1 * $3;
```

```
}
|
expr '/' expr
{
$$ = $1 / $3;
}
|
expr '%' expr
{
$$ = $1 % $3;
}
|
expr '+' expr
{
$$ = $1 + $3;
}
|
expr '-' expr
{
$$ = $1 - $3;
}
|
expr '&' expr
{
$$ = $1 & $3;
}
|
expr '|' expr
{
$$ = $1 | $3;
}
|
'-' expr %prec UMINUS
{
$$ = -$2;
}
|
LETTER
{
$$ = regs[$1];
}
|
number
;
```

```
number: DIGIT
 {
 $$ = $1;
 base = ($1==0) ? 8 : 10;
 } |
 number DIGIT
 {
 $$ = base * $1 + $2;
 }
 ;
%%
int main()
{
return(yyparse());
}
void yyerror(const char *s)
{
 fprintf(stderr, "%s\n",s);
}
int yywrap()
{
 return(1);
}
```

OUTPUT:



RESULT:

Ex.No:4 **Generate Three Address Code Using LEX and YACC**
Date:

AIM:

To write a C program to generate three address code for simple program using lex and yacc.

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:

three.l

```
%{
#include "y.tab.h"
%}
%%
[0-9]+? {yylval.sym=(char)yytext[0]; return NUMBER;}
[a-zA-Z]+? {yylval.sym=(char)yytext[0];return LETTER;}
\n {return 0;}
. {return yytext[0];}
%%
int yywrap()
{
return 1;
}
```

three.y

```
%{
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
#include<ctype.h>
int yylex(void);
void yyerror(char const *);
void ThreeAddressCode();
void triple();
void qudraple();
char AddToTable(char ,char, char);
int ind=0;//count number of lines
```

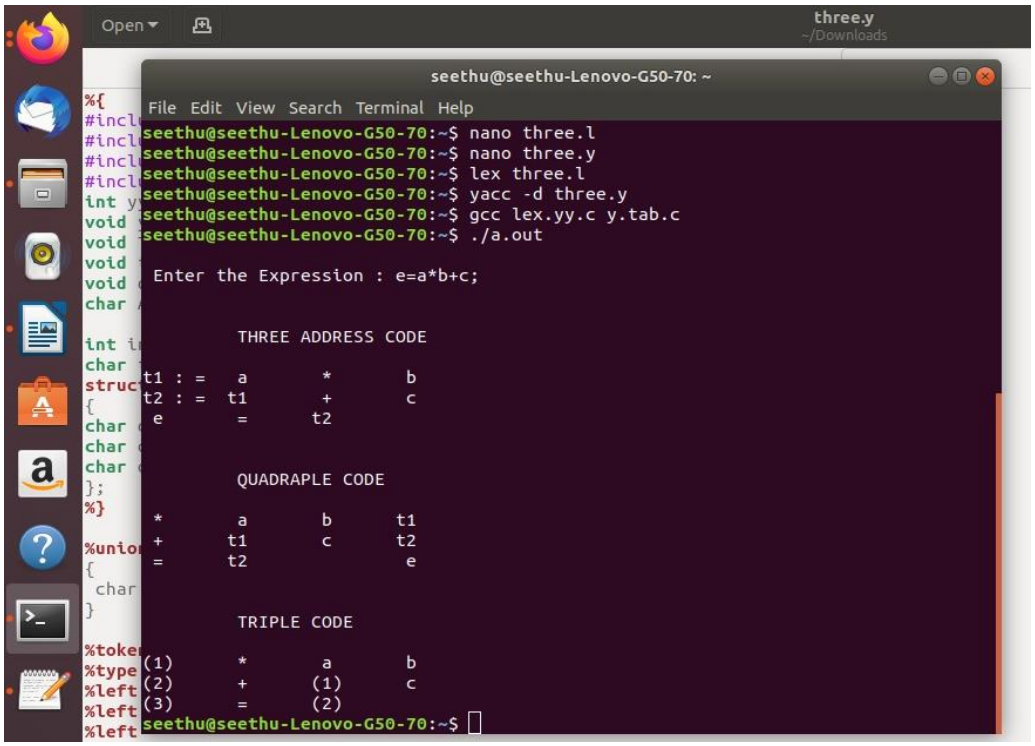
```
char temp = '1';//for t1,t2,t3.....
struct incod
{
char opd1;
char opd2;
char opr;
};
%}
%union
{
char sym;
}
%token <sym> LETTER NUMBER
%type <sym> expr
%left '+'
%left '*'/'
%left '-'
%%
statement: LETTER '=' expr ';' {AddToTable((char)$1,(char)$3,'=');}
| expr ';'
;
expr:
expr '+' expr {$$ = AddToTable((char)$1,(char)$3,'+');}
| expr '-' expr {$$ = AddToTable((char)$1,(char)$3,'-');}
| expr '*' expr {$$ = AddToTable((char)$1,(char)$3,'*');}
| expr '/' expr {$$ = AddToTable((char)$1,(char)$3,'/');}
| '(' expr ')' {$$ = (char)$2;}
| NUMBER {$$ = (char)$1;}
| LETTER {$$ = (char)$1;}
|'-' expr {$$ = AddToTable((char)$2,(char)'\t','-');}
;
%%
void yyerror(const char *s)
{
printf("%s",s);
exit(0);
}
struct incod code[20];
char AddToTable(char opd1,char opd2,char opr)
{
code[ind].opd1=opd1;
code[ind].opd2=opd2;
code[ind].opr=opr;
ind++;
```

```
return temp++;
}
void ThreeAddressCode()
{
int cnt = 0;
char temp = '1';
printf("\n\n\t THREE ADDRESS CODE\n\n");
while(cnt<ind)
{
 if(code[cnt].opr != '=')
 printf("t%c : = \t",temp++);
 if(isalpha(code[cnt].opd1))
 printf(" %c\t",code[cnt].opd1);
 else if(code[cnt].opd1 >='1' && code[cnt].opd1 <='9')
 printf("t%c\t",code[cnt].opd1);
 printf(" %c\t",code[cnt].opr);
 if(isalpha(code[cnt].opd2))
 printf(" %c\n",code[cnt].opd2);
 else if(code[cnt].opd2 >='1' && code[cnt].opd2 <='9')
 printf("t%c\n",code[cnt].opd2);
 cnt++;
}
}
void quadraple()
{
int cnt = 0;
char temp = '1';
printf("\n\n\t QUADRAPLE CODE\n\n");
while(cnt<ind)
{
 printf(" %c\t",code[cnt].opr);
 if(code[cnt].opr == '=')
 {
 if(isalpha(code[cnt].opd2))
 printf(" %c\t \t",code[cnt].opd2);
 else if(code[cnt].opd2 >='1' && code[cnt].opd2 <='9')
 printf("t%c\t \t",code[cnt].opd2);
 printf(" %c\n",code[cnt].opd1);
 cnt++;
 continue;
 }
 if(isalpha(code[cnt].opd1))
 printf(" %c\t",code[cnt].opd1);
 else if(code[cnt].opd1 >='1' && code[cnt].opd1 <='9')
```

```
 printf("t%c\t",code[cnt].opd1);
 if(isalpha(code[cnt].opd2))
 printf(" %c\t",code[cnt].opd2);
 else if(code[cnt].opd2 >='1' && code[cnt].opd2 <='9')
 printf("t%c\t",code[cnt].opd2);
 else printf(" %c",code[cnt].opd2);
 printf("t%c\n",temp++);
 cnt++;
}
}
void triple()
{
int cnt=0;
char temp='1';
printf("\n\n\t TRIPLE CODE\n\n");
 while(cnt<ind)
{
 printf("(%c) \t",temp);
 printf(" %c\t",code[cnt].opr);
 if(code[cnt].opr == '=')
 {
 if(isalpha(code[cnt].opd2))
 printf(" %c \t \t",code[cnt].opd2);
 else if(code[cnt].opd2 >='1' && code[cnt].opd2 <='9')
 printf("(%c)\n",code[cnt].opd2);
 cnt++;
 temp++;
 continue;
 }
 if(isalpha(code[cnt].opd1))
 printf(" %c \t",code[cnt].opd1);
 else if(code[cnt].opd1 >='1' && code[cnt].opd1 <='9')
 printf("(%c)\t",code[cnt].opd1);
 if(isalpha(code[cnt].opd2))
 printf(" %c \n",code[cnt].opd2);
 else if(code[cnt].opd2 >='1' && code[cnt].opd2 <='9')
 printf("(%c)\n",code[cnt].opd2);
 else printf(" %c\n",code[cnt].opd2);
 cnt++;
 temp++;
}
}
int main()
{
```

```
printf("\n Enter the Expression : ");
yyparse();
ThreeAddressCode();
quadraple();
triple();
}
```

OUTPUT:



RESULT:

Ex.No: 5 **Implementation of Type Checking**
Date:

AIM:

To write a C program that implements type checking.

ALGORITHM:

1.Start the program
2.Initialize the variable
3.Declare the function
4.The entry() function is declared to update the symbol table
5.The typecheck() function is created to check for the type mismatches
6.The search() function is declared to search for a variable in symbol table
7.The check() function is created to check if a token is valid or not
8.Call the entry() and typecheck() functions
9.Print the result of typechecking
10.Stop the program

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
{
```

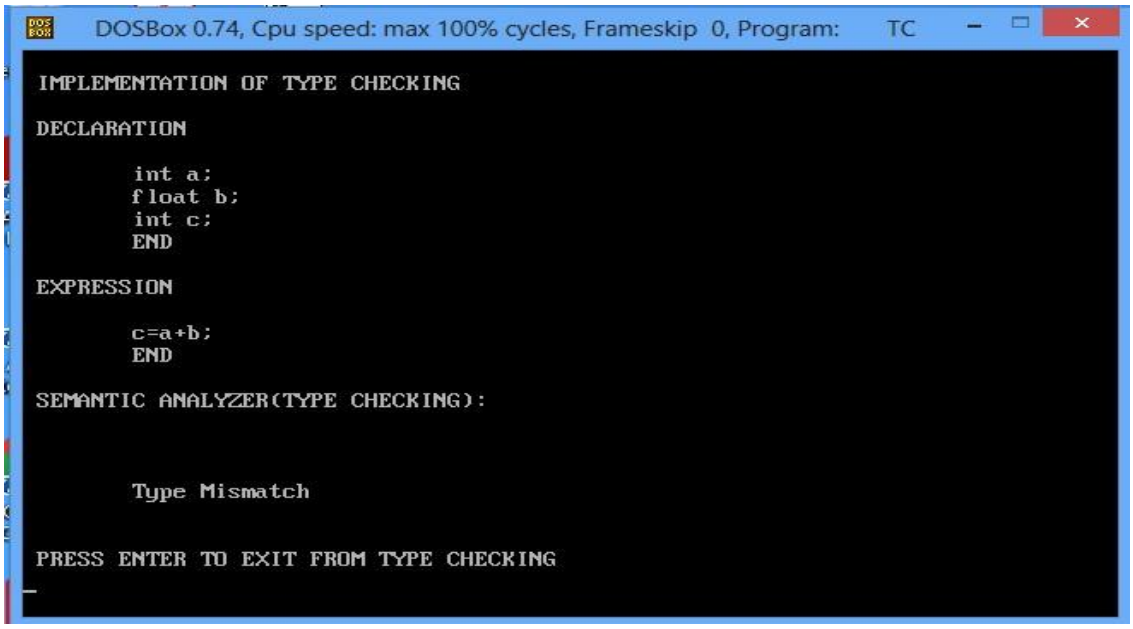
```
printf("\t");
gets(dstr[i]);
i++;
} while(strcmp(dstr[i-1],"END"));
printf("\n EXPRESSION \n\n");
do
{
printf("\t");
gets(estr[l]);
l++;
}while(strcmp(estr[l-1],"END"));
i=0;
printf("\n SEMANTIC ANALYZER(TYPE CHECKING): \n");
while(strcmp(dstr[i],"END"))
{
entry();
printf("\n");
i++;
}
l=0;
while(strcmp(estr[l],"END"))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

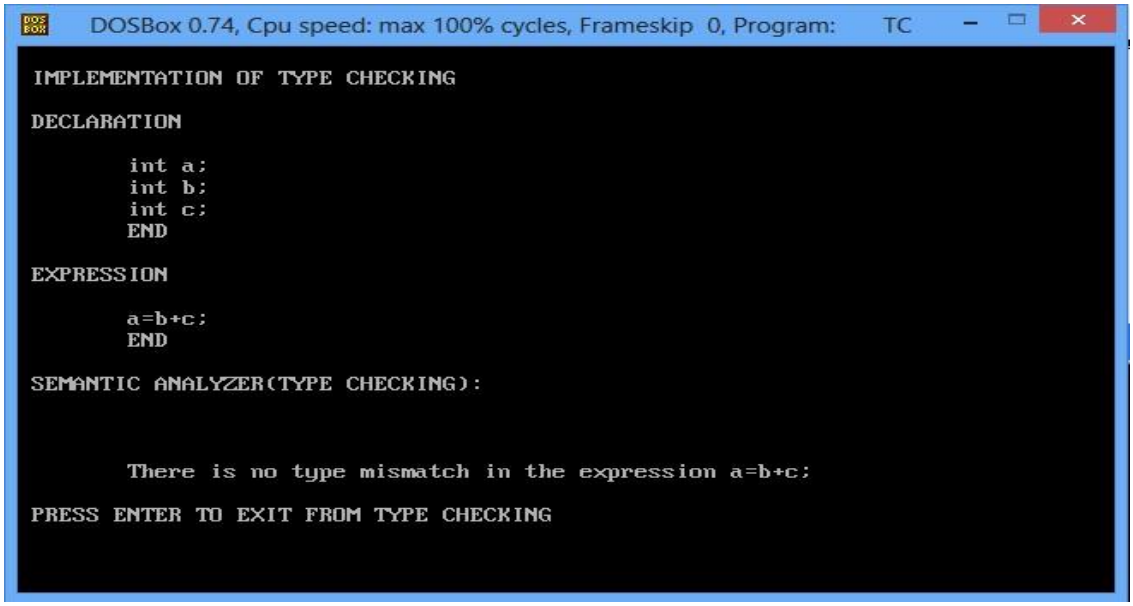
```
k=0;
memset(token,0,strlen(token));
j++;
while(dstr[i][j]!=';')
{
token[k]=dstr[i][j];
k++;
j++;
}
findval=search(token);
if(findval==0)
{
strcpy(tbl[count].var,token);
}
else
{
printf("The variable %s is already declared",token);
}
kflag=0;
count++;
}
else
{
printf("Enter valid datatype\n");
}
}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

OUTPUT:





RESULT:

Ex.No: 6 **Implementation of Simple Code Optimization Techniques**
Date:

AIM:

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

ALGORITHM:

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

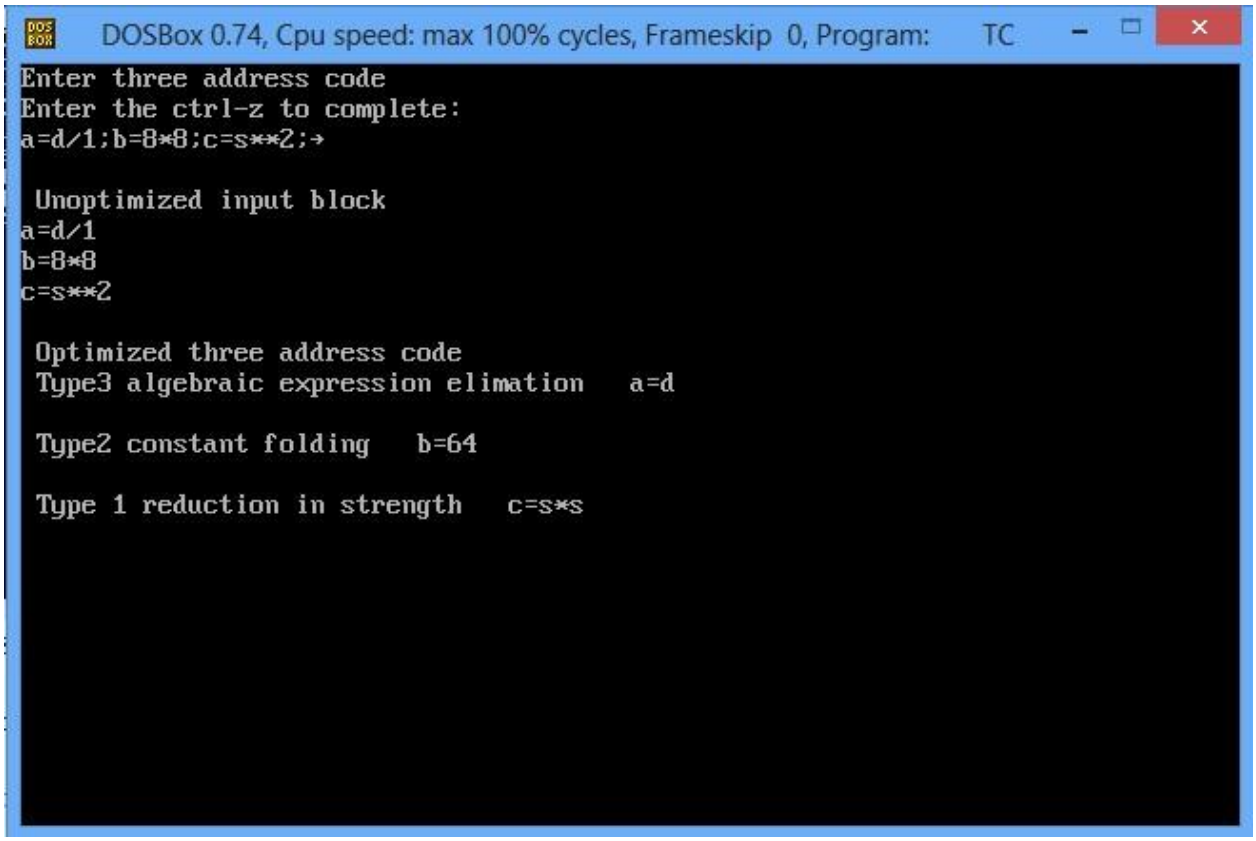
PROGRAM:

```
#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=1,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)
{
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 folding ");
p=a[i][2];
q=a[i][4];
if(strcmp(a[i][3],op1)==0)
r=p*q;
if(strcmp(a[i][3],op2)==0)
r=p+q;
if(strcmp(a[i][3],op3)==0)
r=p/q;
if(strcmp(a[i][3],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)
{
printf("\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++)
```

```
printf("%c",a[i][j]);
printf("\n");
}
else
printf("\n sorry cannot optimize\n");
}
else
{
printf("\n Error input");
}
i++;
}
getch();
}
```

OUTPUT:



RESULT:

Ex.No:7 **Implement the Back End of Compiler**
Date:

AIM:

To implement the back end of the compiler which takes the three address code and produces the 8086 assembly language instructions that can be assembled and run using a 8086 assembler. The target assembly instructions can be simple move, add, sub, jump. Also simple addressing modes are used.

ALGORITHM:

1.Start the program
2.Open the source file and store the contents as quadruples.
3.Check for operators, in quadruples, if it is an arithmetic operator generator it or if assignment operator generates it, else perform unary minus on register C.
4.Write the generated code into output definition of the file in outp.c
5.Print the output.
6.Stop the program.

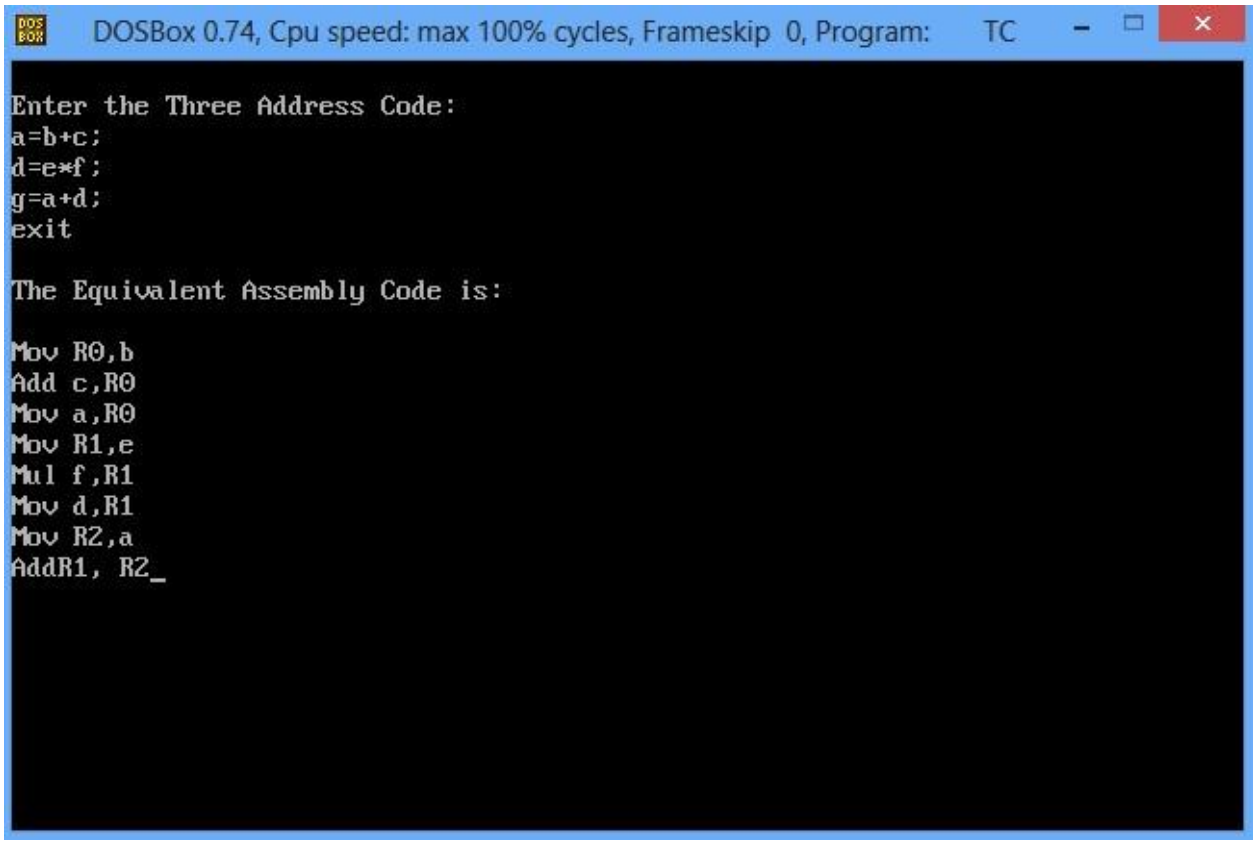
PROGRAM:

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
#include<ctype.h>
#include<graphics.h>
typedef struct
{
char var[10];
int alive;
}
regist;
regist preg[10];
void substring(char exp[],int st,int end)
{
int i,j=0;
char dup[10]="";
for(i=st;i<end;i++)
dup[j++]=exp[i];
dup[j]='0';
strcpy(exp,dup);
}
int getregister(char var[])
```

```
{
int i;
for(i=0;i<10;i++)
{
if(preg[i].alive==0)
{
strcpy(preg[i].var,var);
break;
}
}
return(i);
}
void getvar(char exp[],char v[])
{
int i,j=0;
char var[10]="";
for(i=0;exp[i]!='\0';i++)
if(isalpha(exp[i]))
var[j++]=exp[i];
else
break;
strcpy(v,var);
}
void main()
{
char basic[10][10],var[10][10],fstr[10],op;
int i,j,k,reg,vc,flag=0;
clrscr();
printf("\nEnter the Three Address Code:\n");
for(i=0;;i++)
{
gets(basic[i]);
if(strcmp(basic[i],"exit")==0)
break;
}
printf("\nThe Equivalent Assembly Code is:\n");
for(j=0;j<i;j++)
{
getvar(basic[j],var[vc++]);
strcpy(fstr,var[vc-1]);
substring(basic[j],strlen(var[vc-1])+1,strlen(basic[j]));
getvar(basic[j],var[vc++]);
reg=getregister(var[vc-1]);
if(preg[reg].alive==0)
```

```
{
printf("\nMov R%d,%s",reg,var[vc-1]);
preg[reg].alive=1;
}
op=basic[j][strlen(var[vc-1])];
substring(basic[j],strlen(var[vc-1])+1,strlen(basic[j]));
getvar(basic[j],var[vc++]);
switch(op)
{
case '+': printf("\nAdd"); break;
case '-': printf("\nSub"); break;
case '*': printf("\nMul"); break;
case '/': printf("\nDiv"); break;
}
flag=1;
for(k=0;k<=reg;k++)
{
if(strcmp(preg[k].var,var[vc-1])==0)
{
printf("R%d, R%d",k,reg);
preg[k].alive=0;
flag=0;
break;
}
}
if(flag)
{
printf(" %s,R%d",var[vc-1],reg);
printf("\nMov %s,R%d",fstr,reg);
}
strcpy(preg[reg].var,var[vc-3]);
getch();}
}
```

OUTPUT:



RESULT: