ESTD. 2001

PRATHYUSHA ENGINEERING COLLEGE

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

REGULATION R2021

III YEAR - V SEMESTER

CS3501 COMPILER DESIGN LAB

PREPARED BY

Ms.V.Anithalakshmi

ASST. PROFESSOR/CSE

Table of Contents

CS3501 COMPILER DESIGN LABORATORY

CHAPTER 1

INTRODUCTION

COMPILER

Compiler is a program that reads a program written in one language – the source language – and translates it in to an equivalent program in another language – the target language.

ANALYSIS-SYNTHESIS MODEL OF COMPILATION

There are two parts to compilation: Analysis and Synthesis. The analysis part breaks up the source program into constituent pieces and creates an intermediate representation of source program. The synthesis part constructs the desired target program from the intermediate representation. Of the two parts, synthesis requires the most specialize technique.

PHASES OF A COMPILER

A compiler operates in six phases, each of which transforms the source program from one representation to another. The first three phases are forming the bulk of analysis portion of a compiler. Two other activities, symbol table management and error handling, are also interacting with the six phases of compiler. These six phases are lexical analysis, syntax analysis, semantic analysis, intermediate code generation, code optimization and code generation.

LEXICAL ANALYSIS

In compiler, lexical analysis is also called linear analysis or scanning. In lexical analysis the stream of characters making up the source program is read from left to right and grouped into tokens that are sequences of characters having a collective meaning.

SYNTAX ANALYSIS

It is also called as Hierarchical analysis or parsing. It involves grouping the tokens of the source program into grammatical phrases that are used by the compiler to synthesize output. Usually, a parse tree represents the grammatical phrases of the source program.

SEMANTIC ANALYSIS

The semantic analysis phase checks the source program for semantic errors and gathers type information for the subsequent code generation phase. It uses the hierarchical structure determined by the syntax-analysis phase to identify the operators and operands of expressions and statements. An important component of semantic analysis is type checking. Here the compiler checks that each operator has operands that are permitted by the source language specification.

SYMBOL TABLE MANAGEMENT

Symbol table is a data structure containing the record of each identifier, with fields for the attributes of the identifier. The data structure allows us to find the record for each identifier quickly and store or retrieve data from that record quickly. When the lexical analyzer detects an identifier in the source program, the identifier is entered into symbol table. The remaining phases enter information about identifiers in to the symbol table.

ERROR DETECTION

Each phase can encounter errors. The syntax and semantic analysis phases usually handle a large fraction of the errors detectable by compiler. The lexical phase can detect errors where the characters remaining in the input do not form any token of language. Errors where the token stream violates the structure rules of the language are determined by the syntax analysis phase.

INTERMEDIATE CODE GENERATION

After syntax and semantic analysis, some compilers generate an explicit intermediate representation of the source program. This intermediate representation should have two important properties: it should be easy to produce and easy to translate into target program

CODE OPTIMIZATION

The code optimization phase attempts to improve the intermediate code so that the faster running machine code will result. There are simple optimizations that significantly improve the running time of the target program without slowing down compilation too much.

CODE GENERATION

The final phase of compilation is the generation of target code, consisting normally of reloadable machine code or assembly code.

CHAPTER 2
SYLLABUS

CS3501 COMPILER DESIGN LABORATORY

L	T	P	C
0	0	3	2

OBJECTIVES:
The student should be made to:

- Be exposed to compiler writing tools.
- Learn to implement the different Phases of compiler
- Be familiar with control flow and data flow analysis
- Learn simple optimization techniques

LIST OF EXPERIMENTS:

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

2. Implement a Lexical Analyzer using LEX Tool

3. Generate YACC specification for a few syntactic categories.

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

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

   c. Program to recognize a valid control structures syntax of C language (For loop, while loop, if-else, if-else-if, switch-case, etc.).

   d. Implementation of calculator using LEX and YACC

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

5. Implement type checking using Lex and Yacc.

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

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

OUTCOMES: At the end of the course, the student should be able to

- Implement the different Phases of compiler using tools
- Analyze the control flow and data flow of a typical program
- Optimize a given program
- Generate an assembly language program equivalent to a source language program

LIST OF EQUIPMENT FOR A BATCH OF 30 STUDENTS:
Standalone desktops with C / C++ compiler and Compiler writing tools 30 Nos. (or) Server with C / C++ compiler and Compiler writing tools supporting 30 terminals or more. LEX and YACC

CHAPTER 3

HARDWARE REQUIREMENTS

Processor	Pentium IV
RAM	1GB (min)
Hard Disk	20GB(min)

SOFTWARE REQUIREMENTS

Operating System	Windows XP, Ubuntu
Software	Turbo C++, Flex Tool

CHAPTER 4

DESCRIPTION ABOUT EXPERIMENTS

The experiments are highly exposed to compiler writing tools, to implement the different phases of compiler, to be familiar with control flow and data flow analysis and to learn simple optimization techniques. It also helps the students to learn vast coverage required for designing a compiler. The experiment about lexical analyzer exemplifies the process of producing tokens, eliminating blank and comments, generating symbol table in which it stores the information about identifiers, constants encountered in the input. The lexical analyzer works in two phases, as in first phase it performs scan and in the second phase it does lexical analysis which means producing the series of tokens.

LEARNING OBJECTIVES

The objective of this lab is to teach the students about various operating systems including Windows, and Unix. Students learn about systems configuration and administration. Students learn, explore and practice technologies related to Compiler design and also gather knowledge about YACC programming language.

YACC

YET ANOTHER COMPILER-COMPILER

Yacc is a computer program for the Unix operating system. The name is an acronym for "Yet Another Compiler Compiler". It is a LALR parser generator, generating a parser, the part of a compiler that tries to make syntactic sense of the source code, specifically a LALR parser, based on an analytic grammar written in a notation similar to BNF. Yacc provides a general tool for describing the input to a computer program. The Yacc user specifies the structures of his input, together with code to be invoked as each such structure is recognized. Yacc turns such a specification into a subroutine that handles the input process; frequently, it is convenient and appropriate to have most of the flow of control in the user's application handled by this subroutine.

GENERAL PROCEDURE FOR EXECUTING THE PROGRAMS

1. Use the TURBO C editor to create a file called hello.c containing the following lines:
   ```
   #include<stdio.h>
   main()
   {
   printf("hello world");
   }
   ```
2. Save this file.At the UNIX command prompt ,invoke the gcc compiler as follows
   gcc hello.c
3. The result of this step will be an executable file called a.out. This is the default file name given be the compiler.
4. To execute the program type ./a.out
5. To compile a program and saved the compiled version in a different file name, use the option as in gcc –o hello.exe hello.c

CHAPTER 5

EXERCISES

Exercise-1(a):

Using the LEX tool, Develop a lexical analyzer to recognize a few patterns in C. (Ex. identifiers, constants, comments, operators etc.

OBJECTIVE OF THE EXPERIMENT :

To write a C program for developing a lexical analyzer to recognize few patterns.

DESCRIPTION

Lexical analysis is the process of converting a sequence of characters into a sequence of tokens, i.e. meaningful character strings. A program or function that performs lexical analysis is called a lexical analyzer, lexer, tokenizer, or scanner, though "scanner" is also used for the first stage of a lexer. A lexer is generally combined with a parser, which together analyze the syntax of programming languages, such as in compilers, but also HTML parsers in web browsers, among other examples.

Strictly speaking, a lexer is itself a kind of parser – the syntax of some programming language is divided into two pieces: the lexical syntax (token structure), which is processed by the lexer; and the phrase syntax, which is processed by the parser. The lexical syntax is usually a regular language, whose alphabet consists of the individual characters of the source code text. The phrase syntax is usually a context-free language, whose alphabet consists of the tokens produced by the lexer. While this is a common separation, alternatively, a lexer can be combined with the parser in scannerless parsing.

PROCEDURE:

1) Start a new program.
2) Declare all the required variables.
3) Get the input expression in an array.
4) Check the array using, if condition to see whether it has any keywords, constants, identifiers, operators or special characters.
5) Keywords are stored in an array initially and the keyword in the input expression can be checked by comparing the strings.
6) The alphabets other than keywords are shown as identifiers.
7) Other assignment operators can also be checked using if and else if conditions.
8) End the program.
9) Compile and run the program.

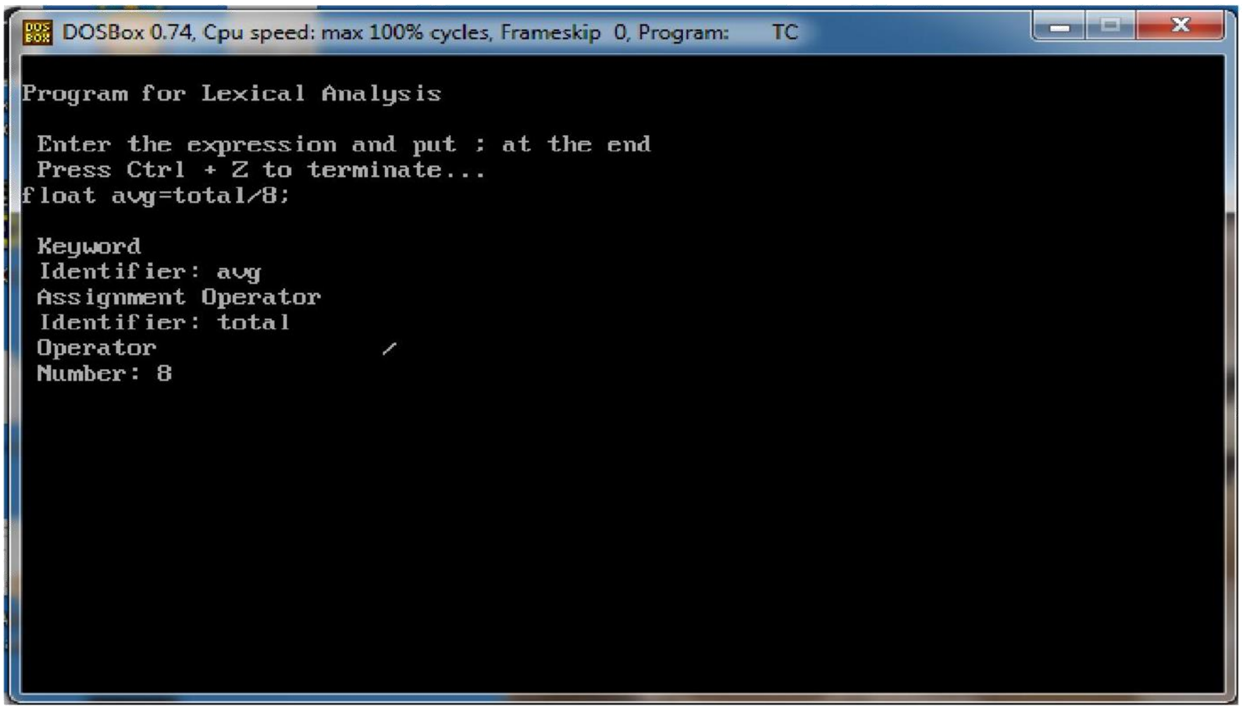
PROGRAM:

```
#include<stdio.h>
#include<conio.h>
#include<ctype.h>
#include<string.h>
#include<stdlib.h>
#define SIZE 128
#define NONE -1
#define EOS '\0'
#define NUM 256
#define KEYWORD257
#define PAREN 258
#define ID 259
#define ASSIGN 260
#define REL_OP 261
#define DONE 262
#define MAX 999
char lexemes[MAX];
char buffer[SIZE];
int lastchar = -1;
int lastentry = 0;
int tokenval=NONE;
 int lineno=1;
struct entry
{
char *lexptr;
int token;
}symtable[100];
struct entry keywords[]={"if",KEYWORD,"else",KEYWORD,"for",KEYWORD,
"int",KEYWORD,"float",KEYWORD,"double",KEYWORD,"char",KEYWORD,
"struct",KEYWORD,"return",KEYWORD,0,0};
void Error_Message(char *m)
{
printf(stderr,"line %d:%s",lineno,m);
exit(1);
}
int look_up(char s[])
{
int k;
for(k=lastentry;k>0;k--)
if(strcmp(symtable[k].lexptr,s)==0)
return k;
return 0;
}
```

```
int insert(char s[],int tok)
{
int len; len=strlen(s);
if(lastentry+1>=MAX)
  Error_Message("Symbol Table is Full");
if(lastchar+len+1>=MAX
  Error_Message("Lexemes Array is
Full");
 lastentry++;
symtable[lastentry].token=tok;
symtable[lastentry].lexptr=&lexemes[lastchar+1];
lastchar = lastchar + len + 1;
strcpy(symtable[lastentry].lexptr,s);
return lastentry;
}
void Initialize()
{
struct entry *ptr;
for(ptr=keywords;ptr->token;ptr++)
insert(ptr->lexptr,ptr->token);
}
int lexer()
{
int t;
int val,i=0;
while(1)
{
t=getchar();
if(t == ' '|| t=='\t');
else if(t=='\n')
lineno++;
else if(t == '('|| t == ')')
return PAREN;
else if(t=='<' ||t=='>' ||t=='<=' ||t=='>=' ||t ==
'!=') return REL_OP;
else if(t == '=')
return ASSIGN;
else if(isdigit(t))
{
ungetc(t,stdin);
scanf("%d",&tokenval);
return NUM;
}
else if(isalpha(t))
{
while(isalnum(t))
{
buffer[i]=t;
t=getchar();
i++;
if(i>=SIZE)
Error_Message("compiler error");
}
buffer[i]=EOS;
if(t!=EOF)
ungetc(t,stdin);
val=look_up(buffer);
if(val==0)
val=insert(buffer,ID);
tokenval=val;
return symtable[val].token;
}
else if(t==EOF)
return DONE;
else
{
tokenval=NONE;
return t;
}
}
}
void main()
{
int lookahead;
char ans;
clrscr();
printf("\nProgram for Lexical Analysis \n");
Initialize();
printf("\n Enter the expression and put ; at the end");
printf("\n Press Ctrl + Z to terminate... \n");
lookahead=lexer();
while(lookahead!=DONE)
```

```
{
if(lookahead==NUM)
printf("\n Number: %d",tokenval);
if(lookahead=='+'|| lookahead=='-'|| lookahead=='*'||
lookahead=='/')
printf("\n Operator\t\t%c",lookahead);
if(lookahead==PAREN)
printf("\n Parentesis");
if(lookahead==ID)
printf("\n Identifier: %s",symtable[tokenval].lexptr);
if(lookahead==KEYWORD)
printf("\n Keyword");
if(lookahead==ASSIGN)
printf("\n Assignment Operator");
if(lookahead==REL_OP)
printf("\n Relataional Operator");
lookahead=lexer();
}
}
```

Sample Output:


DOSBox 0.74, Cpu speed: max 100% cycles, Frameskip 0, Program: TC

```
Program for Lexical Analysis

 Enter the expression and put ; at the end
 Press Ctrl + Z to terminate...
float avg=total/8;

 Keyword
 Identifier: avg
 Assignment Operator
 Identifier: total
 Operator               /
 Number: 8
```

CONCLUSION:

Thus the C program for developing lexical analyzer to recognize a few patterns was written and executed successfully.

Exercise-1(b):

Create a symbol table, while recognizing identifiers.

OBJECTIVE OF THE EXPERIMENT :

To write a C program for implementing Symbol Table.

DESCRIPTION:

A symbol table is a data structure used by a language translator such as a compiler or interpreter, where each identifier in a program's source code is associated with information relating to its declaration or appearance in the source, such as its type, scope level and sometimes its location. A common implementation technique is to use a hash table.

A compiler may use one large symbol table for all symbols or use separated, hierarchical symbol tables for different scopes. There are also trees, linear lists and self-organizing lists which can be used to implement symbol table. It also simplifies the classification of literals in tabular format. The symbol table is accessed by most phases of a compiler, beginning with the lexical analysis to optimization.

PROCEDURE:

1) Start a new program.
2) Declare all the required variables.
3) Get the input expression in an array.
4) Check the array using, if condition to see whether it has any keywords, constants, identifiers, operators or special characters.
5) The input expression can be analyzed as alphabet or digit by using 'isalpha()' and 'isdigit()' built in function syntax.
6) Keywords are stored in an array initially and the keyword in the input expression can be checked by comparing the strings using 'strcmp()' function.
7) The alphabets other than keywords are shown as identifiers.
8) Other address constants and special characters can also be checked using if and else if conditions.
9) End the program.
10) Compile and run the program.

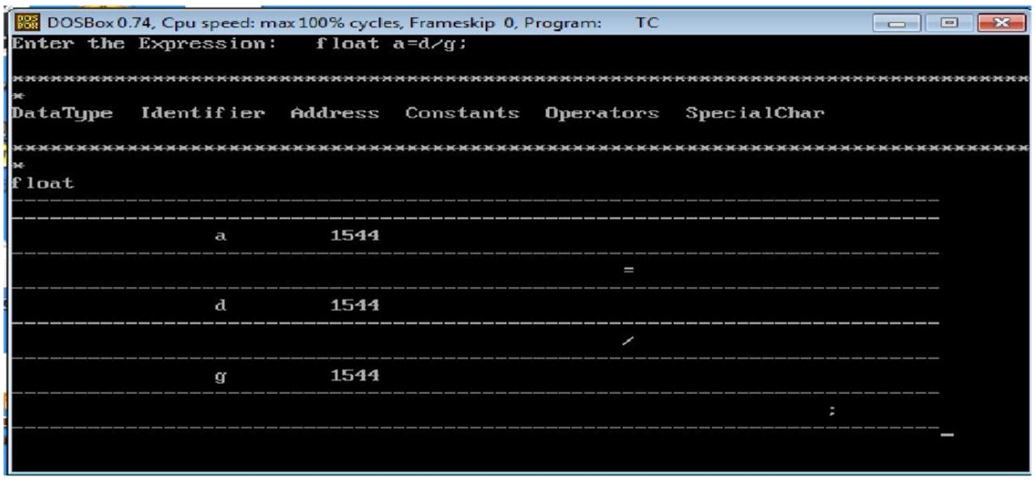
PROGRAM:

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
#include<ctype.h>
void main()
{
char in[50],dig[50],id[50];
int i=0,j=0,k,l=0;
clrscr();
printf("Enter the
Expression:\t"); gets(in);
```

```
printf("\n******************************************************************************");
printf("\nDataType Identifier Address Constants Operators SpecialChar\n");
printf("\n******************************************************************************");
while(in[i]!='\0')
{
if(isalpha(in[i]))
{
j=0;
while((isalpha(in[i]))||(isdigit(in[i])))
{
id[j]=in[i];
i++;
j++;
}
id[j]='\0';
if(strcmp(id,"char")==0||strcmp(id,"int")==0||strcmp(id,"float")==0||strcmp(id,"if"
)==0||strcmp(id,"long")==0||strcmp(id,"while")==0||strcmp(id,"do")==0||
strcmp(id,"for")==0||strcmp(id,"switch")==0||strcmp(id,"double")==0)
{
printf("\n");
for(l=0;l<j;l++)
printf("%c",id[l]);
}
else
{
printf("\t\t\t");
for(l=0;l<j;l++)
printf("%c %u",id[l]);
}
}
else if(isdigit(in[i]))
{
k=0;
while(isdigit(in[i]))
{
```

```
dig[k]=in[i];
i++;
k++;
}
printf("\n\t\t\t\t");
for(l=0;l<k;l++)
printf("%c",dig[l]);
}
else if(in[i]=='+'||in[i]=='-'||in[i]=='*'||in[i]=='/'||in[i]=='<'||in[i]=='>'||in[i]=='=')
{
printf("\t\t\t\t\t\t\t%c",in[i]);
i++;
}
else if(in[i]==';'||in[i]==':'||in[i]=='.'||in[i]=='('||in[i]==')'||in[i]=='{'||in[i]=='}')
{
printf("\t\t\t\t\t\t\t\t\t%c",in[i]);
i++;
}
else
i++;
printf("\n"); printf("---------------------------------------------------------
--------------"); getch();

}}
```

SAMPLE OUTPUT:



DOSBox 0.74, Cpu speed: max 100% cycles, Frameskip 0, Program: TC

```
Enter the Expression:   float a=d/g;

********************************************************************************
*
DataType  Identifier  Address  Constants  Operators  SpecialChar

********************************************************************************
*
float
--------------------------------------------------------------------------
--------------------------------------------------------------------------
                a         1544
--------------------------------------------------------------------------
                                                 =
--------------------------------------------------------------------------
                d         1544
--------------------------------------------------------------------------
                                                 /
--------------------------------------------------------------------------
                g         1544
--------------------------------------------------------------------------
                                                                 ;
--------------------------------------------------------------------------_
```

Result :

Thus the C program for implementing symbol table was written and executed successfully.

Exercise-2:

Implement a Lexical Analyzer using LEX Tool.

OBJECTIVE OF THE EXPERIMENT :

To write a program for implementing Lexical Analyzer using Lex Tool.

DESCRIPTION:

Lex helps write programs whose control flow is directed by instances of regular expressions in the input stream. It is well suited for editor-script type transformations and for segmenting input in preparation for a parsing routine.

Lex source is a table of regular expressions and corresponding program fragments. The table is translated to a program which reads an input stream, copying it to an output stream and partitioning the input into strings which match the given expressions. As each such string is recognized the corresponding program fragment is executed. The recognition of the expressions is performed by a deterministic finite automaton generated by Lex. The program fragments written by the user are executed in the order in which the corresponding regular expressions occur in the input stream.

The lexical analysis programs written with Lex accept ambiguous specifications and choose the longest match possible at each input point. If necessary, substantial lookahead is performed on the input, but the input stream will be backed up to the end of the current partition, so that the user has general freedom to manipulate it.

PROCEDURE:

1) Start the program.

2) Declare necessary variables and creates token representation using Regular. 3) Print the preprocessor or directives, key words by an al y s i s of the i n
put program .

4) In the program check whether there are arguments.
5) Declare a file and open it as read mode.

6) Read the file and if any taken in source program matches with RE that all returned as integer value.

7) Print the token identified using YYdex () function.

8) Stop the program.

PROGRAM:

```
%{
%}
identifier[a-zA-Z][a-zA-Z0-9]*
%%
#.* {printf("\n%s is a preprocessor directive",yytext);}
int |
float |
char |
double |
while |
do |
if |
break |
continue |
void |
switch |
return |
else |
goto {printf("\n%s is a keyword",yytext);}
{identifier}\( {printf("\n function %s",yytext);}
\{ {printf("\nblock begins");}
\} {printf("\nblock ends");}
\( {printf("\n");ECHO;}
{identifier}(\[[0-9]*\])* {printf("\n%s is an identifier",yytext);}
\".*\" {printf("\n %s is a string ",yytext);}
[0-9]+ {printf("\n%s is a number",yytext);
}
\<=  |
\>=  |
\<    |
\> |
\== {printf("\n %s is a relational operator",yytext);}
 \= | \+ | \- | \/ | \& |


% {printf("\n %s is a operator",yytext);}
. |
\n;
```

```
%%
int main(int argc,char **argv)
        {
        FILE *file;
        file=fopen("inp.c","r");
        if(!file)
        {
        printf("could not open the file!!!");
        exit(0);
        }
        yyin=file;
        yylex();
        printf("\n\n");
        return(0);
        }
        int yywrap()
        {
        return 1;
}
```

INPUT FILE:

```
        #include<stdio.h>
        void main()
        {
        int a,b,c;
        printf("enter the value for a,b");
        scanf("%d%d",&a,&b)';
        c=a+b;
        printf("the value of c:%d",&c);
}
```

OUTPUT:

```
[3cse01@localhost ~]$ lex ex3.l
[3cse01@localhost ~]$ cc lex.yy.c
[3cse01@localhost ~]$ ./a.out
#include<stdio.h> is a preprocessor directive
void is a keyword
function main(
block begins
int is a keyword
```

```
a is an identifier b
is an identifier c is
an          identifier
function printf(
"enter the value for a,b" is a string
function scanf(
"%d%d" is a string
& is an operator a
is an identifier
& is an operator
b is an identifier
c is an identifier
= is an operator
a is an identifier
+ is a operator
 b is an identifier
function printf(
"the value of c:%d" is a string
& is a operator
c is an identifier
block ends
```

Result:

Thus the program to implement the lexical analyzer using lex tool for a subset of C language was implemented and verified.

Exercise – 3(a):

Generate YACC specification for a few syntactic categories.

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

AIM:

To write a Yacc program to valid arithmetic expression.

ALGORITHM:

Step 1: Start the program to recognize a valid arithmetic expressions.
Step 2: Define the pattern for digits in lex file
Step 3: Assign yylval to return the token number as integer in lex file
Step 4: Define the patterns for operators and expressions in yacc file
Step 5: Use yywrap() to print an error using yyerror() in yacc file.
Step 6: Use yyparse() to reads a stream of value pairs in yacc file
Step 7: Display the valid arithmetic expressions
Step 8: Stop the program.

PROGRAM:

LEX PART : arithmetic.l

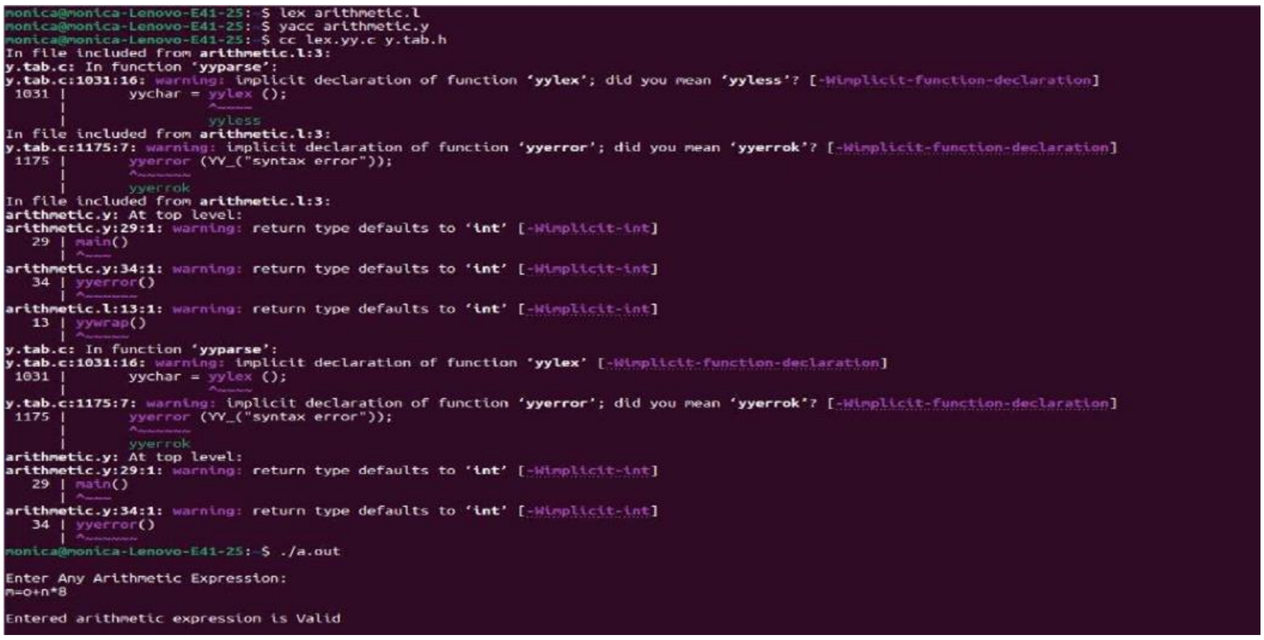
```
%{
#include<stdio.h>
#include "y.tab.h"
%}
%%
[a-zA-Z] {return VARIABLE;}
[0-9] {return NUMBER;}
[\t] ;
[\n] {return 0;}
. {return yytext[0];}
%%
yywrap()
{}
```

YACC PART : arithmetic.y

```
%{
 #include<stdio.h>
%}
%token NUMBER
%token VARIABLE
%left '+' '-'
%left '*' '/' '%'
%left '(' ')'
%%
S: VARIABLE'='E {
 printf("\nEntered arithmetic expression is Valid\n\n");
 return 0;
 }
E:E'+'E
|E'-'E
 |E'*'E
 |E'/'E
 |E'%'E
 |'('E')'
 | NUMBER
 | VARIABLE
;
%%
main()
{
 printf("\nEnter Any Arithmetic Expression:\n");
 yyparse();
}
yyerror()
{
 printf("\nEntered arithmetic expression is Invalid\n\n");
}
```

OUTPUT:

```
$:lex arithmetic.l
$:yacc arithmetic.y
$:cc lex.yy.c y.tab.h
$:./a.out
```



```
monica@monica-Lenovo-E41-25:~$ lex arithmetic.l
monica@monica-Lenovo-E41-25:~$ yacc arithmetic.y
monica@monica-Lenovo-E41-25:~$ cc lex.yy.c y.tab.h
In file included from arithmetic.l:3:
y.tab.c: In function 'yyparse':
y.tab.c:1031:16: warning: implicit declaration of function 'yylex'; did you mean 'yyless'? [-Wimplicit-function-declaration]
 1031 |       yychar = yylex ();
      |                ^~~~~
      |                yyless
In file included from arithmetic.l:3:
y.tab.c:1175:7: warning: implicit declaration of function 'yyerror'; did you mean 'yyerrok'? [-Wimplicit-function-declaration]
 1175 |       yyerror (YY_("syntax error"));
      |       ^~~~~~~
      |       yyerrok
In file included from arithmetic.l:3:
arithmetic.y: At top level:
arithmetic.y:29:1: warning: return type defaults to 'int' [-Wimplicit-int]
   29 | main()
      | ^~~~
arithmetic.y:34:1: warning: return type defaults to 'int' [-Wimplicit-int]
   34 | yyerror()
      | ^~~~~~~
arithmetic.l:13:1: warning: return type defaults to 'int' [-Wimplicit-int]
   13 | yywrap()
      | ^~~~~~
y.tab.c: In function 'yyparse':
y.tab.c:1031:16: warning: implicit declaration of function 'yylex' [-Wimplicit-function-declaration]
 1031 |       yychar = yylex ();
      |                ^~~~~
y.tab.c:1175:7: warning: implicit declaration of function 'yyerror'; did you mean 'yyerrok'? [-Wimplicit-function-declaration]
 1175 |       yyerror (YY_("syntax error"));
      |       ^~~~~~~
      |       yyerrok
arithmetic.y: At top level:
arithmetic.y:29:1: warning: return type defaults to 'int' [-Wimplicit-int]
   29 | main()
      | ^~~~
arithmetic.y:34:1: warning: return type defaults to 'int' [-Wimplicit-int]
   34 | yyerror()
      | ^~~~~~~
monica@monica-Lenovo-E41-25:~$ ./a.out

Enter Any Arithmetic Expression:
m=o+n*8

Entered arithmetic expression is Valid
```

RESULT:

Thus, the program to recognize the valid arithmetic expression is verified and executed successfully..

Exercise-3(b):

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

AIM:

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

ALGORITHM:

Step 1: Start the program to recognize a valid arithmetic expressions.

Step 2: Define the pattern for letters and digits in lex file.

Step 3: Define the pattern for identifiers in yacc file.

Step 4: Use yywrap() to print an error using yyerror() in yacc file.

Step 5: Use yyparse() to reads a stream of value pairs in yacc file.

Step 6: Print the valid identifiers using yytext().

Step 7: Display the valid identifiers. Step 8: Stop the program.

PROGRAM:

LEX PART : variable.l

```
%{
 #include "y.tab.h"
%}
%%
[a-zA-Z] { return ALPHA ;}
[0-9]+ { return NUMBER ; }
"\n" { return ENTER ;}
. { return ER; }
%%
yywrap()
{}
```

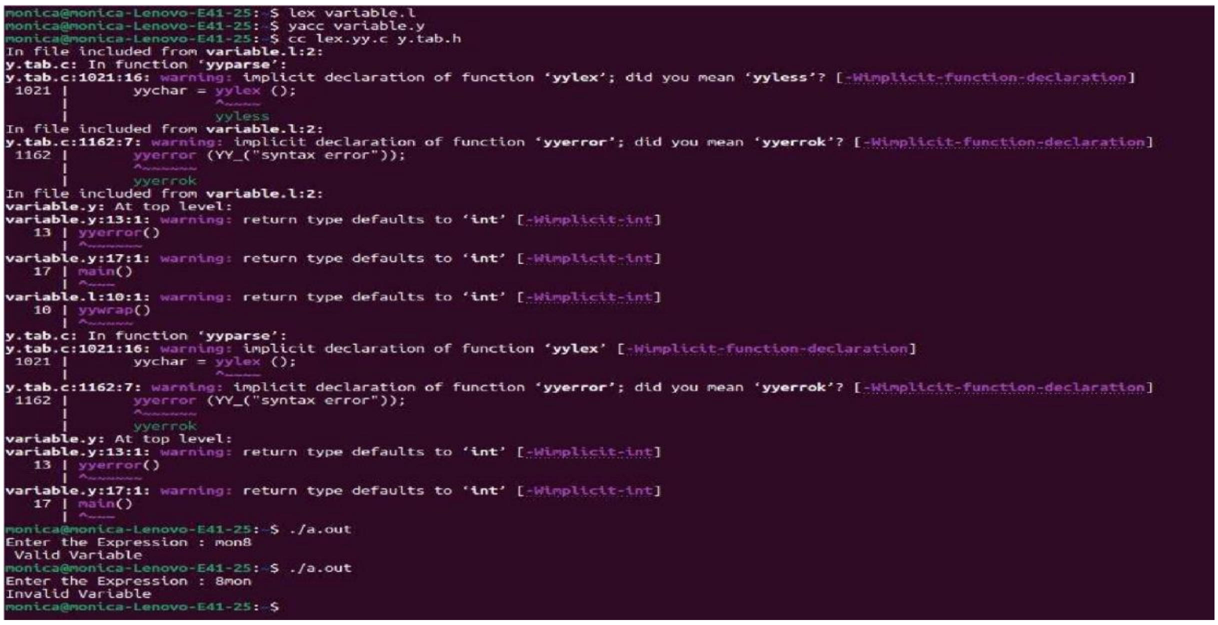
YACC PART : variable.y

```
%{
#include<stdio.h>
#include<stdlib.h>
%}
%token ALPHA NUMBER ENTER ER
%%
var : v ENTER  { printf(" Valid Variable\n"); exit(0);} v:ALPHA exp1 exp1:ALPHA exp1
```

```
|NUMBER exp1
|;
%%
yyerror()
{printf("Invalid Variable\n");}
main() { printf("Enter the Expression : "); yyparse(); }
```

OUTPUT:

```
$:lex variable.l
$:yacc variable.y
$:cc lex.yy.c y.tab.h $:./a.out
```



```
monica@monica-Lenovo-E41-25:~$ lex variable.l
monica@monica-Lenovo-E41-25:~$ yacc variable.y
monica@monica-Lenovo-E41-25:~$ cc lex.yy.c y.tab.h
In file included from variable.l:2:
y.tab.c: In function ‘yyparse’:
y.tab.c:1021:16: warning: implicit declaration of function ‘yylex’; did you mean ‘yyless’? [-Wimplicit-function-declaration]
 1021 |         yychar = yylex ();
      |                  ^~~~~
      |                  yyless
In file included from variable.l:2:
y.tab.c:1162:7: warning: implicit declaration of function ‘yyerror’; did you mean ‘yyerrok’? [-Wimplicit-function-declaration]
 1162 |       yyerror (YY_("syntax error"));
      |       ^~~~~~~
      |       yyerrok
In file included from variable.l:2:
variable.y: At top level:
variable.y:13:1: warning: return type defaults to ‘int’ [-Wimplicit-int]
   13 | yyerror()
      | ^~~~~~~
variable.y:17:1: warning: return type defaults to ‘int’ [-Wimplicit-int]
   17 | main()
      | ^~~~
variable.l:10:1: warning: return type defaults to ‘int’ [-Wimplicit-int]
   10 | yywrap()
      | ^~~~~~
y.tab.c: In function ‘yyparse’:
y.tab.c:1021:16: warning: implicit declaration of function ‘yylex’ [-Wimplicit-function-declaration]
 1021 |         yychar = yylex ();
      |                  ^~~~~
y.tab.c:1162:7: warning: implicit declaration of function ‘yyerror’; did you mean ‘yyerrok’? [-Wimplicit-function-declaration]
 1162 |       yyerror (YY_("syntax error"));
      |       ^~~~~~~
      |       yyerrok
variable.y: At top level:
variable.y:13:1: warning: return type defaults to ‘int’ [-Wimplicit-int]
   13 | yyerror()
      | ^~~~~~~
variable.y:17:1: warning: return type defaults to ‘int’ [-Wimplicit-int]
   17 | main()
      | ^~~~
monica@monica-Lenovo-E41-25:~$ ./a.out
Enter the Expression : mon8
 Valid Variable
monica@monica-Lenovo-E41-25:~$ ./a.out
Enter the Expression : 8mon
Invalid Variable
monica@monica-Lenovo-E41-25:~$
```

RESULT:

Thus, the given program to validate variable is implemented and executed successfully

Exercise-3(c):

Program to recognize a valid control structures syntax of C language (For loop, while loop, if-else, if-else-if, switch-case, etc.).

AIM:

To write a yacc program to validate control structures syntax.

ALGORITHM:

Step1: Start the program.
Step2: Read the statement.
Step4: Validating the given statement according to the rule using yacc.
Step4: Using the syntax rule print the result of the given syntax.
Step5: Stop the program.

PROGRAM: **FOR LOOP**

LEX PART : for.l

```
%{

#include<stdio.h>

#include "y.tab.h"

%}

alpha [A-Za-z]

digit [0-9]

%%

[\t \n] for
return FOR;

{digit}+    return NUM;

{alpha}({alpha}|{digit})* return ID;

"<="        return LE;

">="        return GE;

"=="        return EQ;
"!="        return NE;

"||"          return OR;

"&&"        return AND;

.           return yytext[0];

%%
```

```
yywrap()

{}
```

YACC PART : for.y

```
%{

#include <stdio.h>

#include <stdlib.h>

%}

%token ID NUM FOR LE GE EQ NE OR AND

%right '='

%left OR AND

%left '>' '<' LE GE EQ NE

%left '+' '-'

%left '*' '/'

%right UMINUS

%left '!'

%%

S       : ST {printf("Input accepted\n"); exit(0);};

ST      : FOR '(' E ';' E2 ';' E ')' DEF

        ;

DEF     : '{' BODY '}'

        | E';'

        | ST

        |

        ;

BODY  : BODY BODY

        | E ';'

        | ST

        |

        ;

E       : ID '=' E
```

```
        | E '+' E

        | E '-' E

        | E '*' E

        | E '/' E

        | E '<' E

        | E '>' E

        | E LE E

        | E GE E

        | E EQ E

        | E NE E

        | E OR E
        | E AND E
        | E '+' '+'

        | E '-' '-'

        | ID

        | NUM

        ;

E2     : E'<'E          | E'>'E
        | E LE E
       | E GE E

       | E EQ E

       | E NE E

       | E OR E

        | E AND E
       ;

%%

main() {

    printf("Enter the
expression:\n");     yyparse(); }
yyerror() {

   printf("\nEntered arithmetic expression is
Invalid\n\n"); }
```

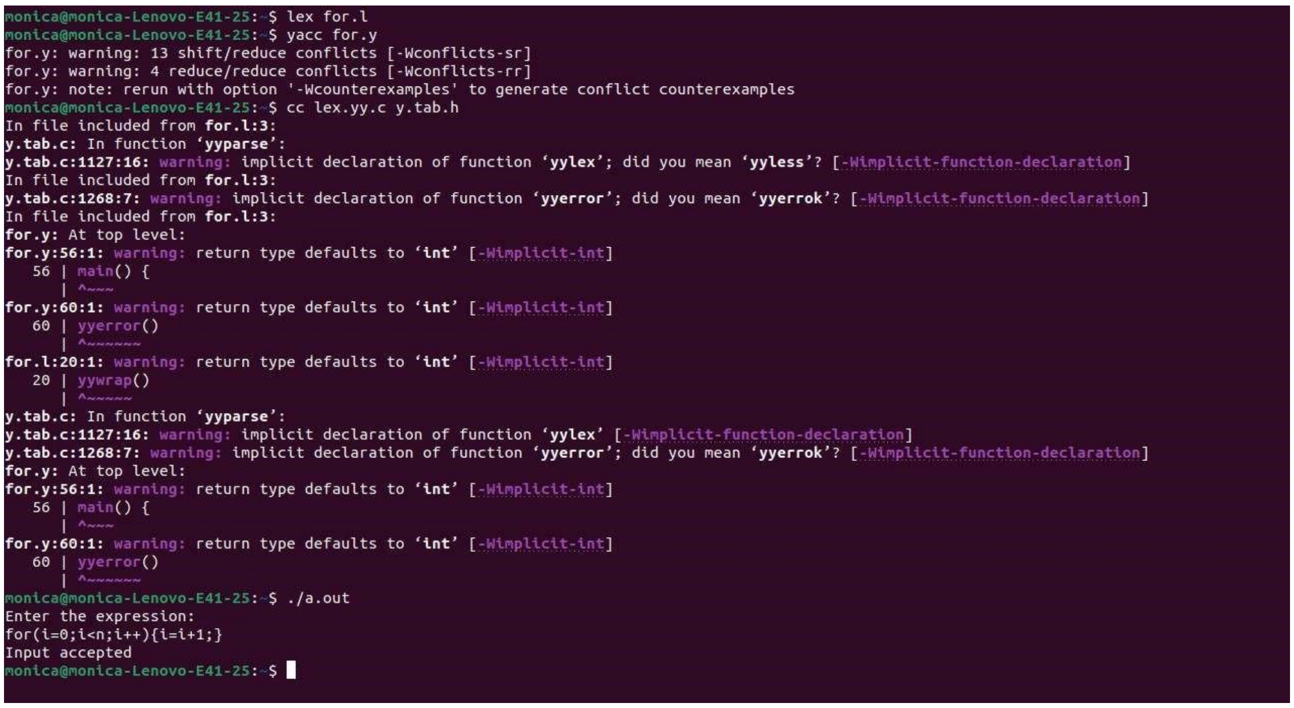
OUTPUT:

$:lex for.l

$:yacc for.y

$:cc lex.yy.c y.tab.h

$:./a.out



```
monica@monica-Lenovo-E41-25:~$ lex for.l
monica@monica-Lenovo-E41-25:~$ yacc for.y
for.y: warning: 13 shift/reduce conflicts [-Wconflicts-sr]
for.y: warning: 4 reduce/reduce conflicts [-Wconflicts-rr]
for.y: note: rerun with option '-Wcounterexamples' to generate conflict counterexamples
monica@monica-Lenovo-E41-25:~$ cc lex.yy.c y.tab.h
In file included from for.l:3:
y.tab.c: In function ‘yyparse’:
y.tab.c:1127:16: warning: implicit declaration of function ‘yylex’; did you mean ‘yyless’? [-Wimplicit-function-declaration]
In file included from for.l:3:
y.tab.c:1268:7: warning: implicit declaration of function ‘yyerror’; did you mean ‘yyerrok’? [-Wimplicit-function-declaration]
In file included from for.l:3:
for.y: At top level:
for.y:56:1: warning: return type defaults to ‘int’ [-Wimplicit-int]
   56 | main() {
      | ^~~~
for.y:60:1: warning: return type defaults to ‘int’ [-Wimplicit-int]
   60 | yyerror()
      | ^~~~~~~
for.l:20:1: warning: return type defaults to ‘int’ [-Wimplicit-int]
   20 | yywrap()
      | ^~~~~~
y.tab.c: In function ‘yyparse’:
y.tab.c:1127:16: warning: implicit declaration of function ‘yylex’ [-Wimplicit-function-declaration]
y.tab.c:1268:7: warning: implicit declaration of function ‘yyerror’; did you mean ‘yyerrok’? [-Wimplicit-function-declaration]
for.y: At top level:
for.y:56:1: warning: return type defaults to ‘int’ [-Wimplicit-int]
   56 | main() {
      | ^~~~
for.y:60:1: warning: return type defaults to ‘int’ [-Wimplicit-int]
   60 | yyerror()
      | ^~~~~~~
monica@monica-Lenovo-E41-25:~$ ./a.out
Enter the expression:
for(i=0;i<n;i++){i=i+1;}
Input accepted
monica@monica-Lenovo-E41-25:~$ 
```

PROGRAM: **WHILE LOOP**

LEX PART : while.l

```
%{
#include<stdio.h>
#include "y.tab.h"
%}
alpha [A-Za-z]
digit [0-9]
%%
[\t \n]
while           return WHILE;
{digit}+    return NUM;
{alpha}({alpha}|{digit})* return ID;
"<="        return LE;
```

```
">="        return GE;

"=="        return EQ;
"!="        return NE;

"||"          return OR;

"&&"        return AND;

.              return yytext[0];

%%

yywrap()

{}
```

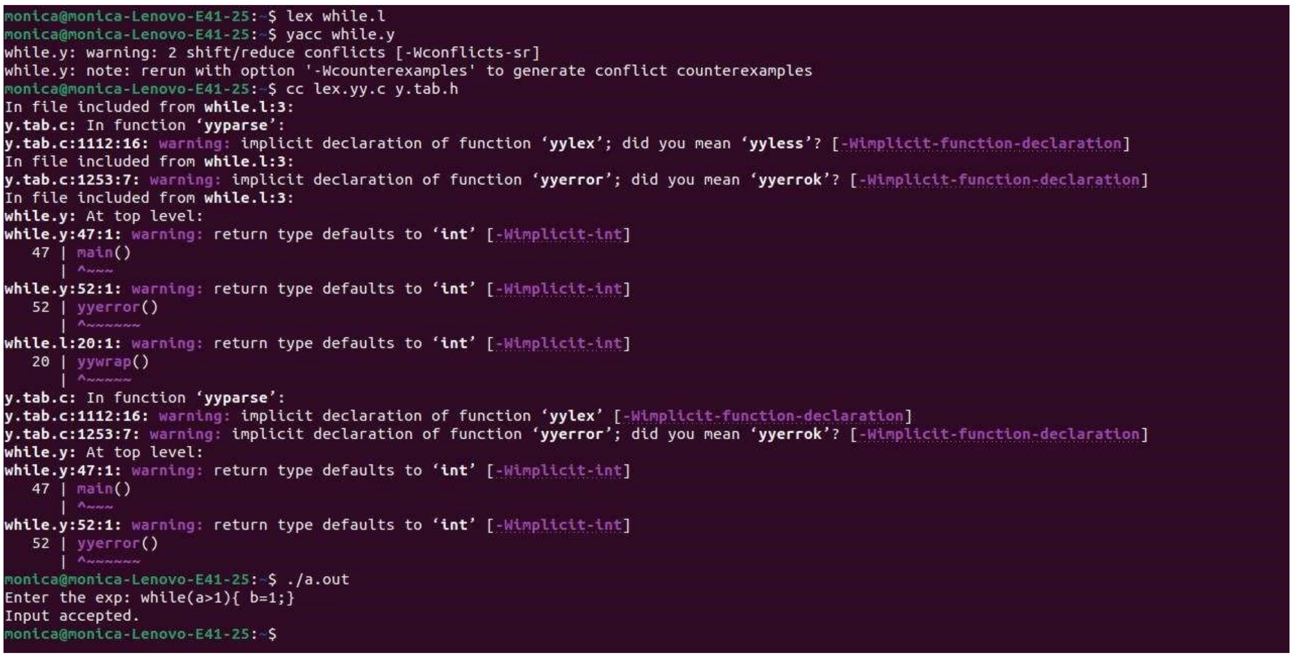
YACC PART :while.y

```
%{

#include <stdio.h>

#include <stdlib.h>

%}

%token ID NUM WHILE LE GE EQ NE OR AND

%right '='

%left AND OR

%left '<' '>' LE GE EQ NE

%left '+''-'

%left '*''/'

%right UMINUS

%left '!'

%%

S       : ST1 {printf("Input accepted.\n");exit(0);};

ST1   :   WHILE'(' E2 ')' '{' ST '}'

ST     :    ST ST

        | E';'

        ;

E      : ID'='E

        | E'+'E

        | E'-'E
        | E'*'E

        | E'/'E

        | E'<'E

        | E'>'E

        | E LE E

        | E GE E

        | E EQ E

        | E NE E

        | E OR E

        | E AND E

        | ID

        | NUM

        ;

E2    : E'<'E

        | E'>'E

        | E LE E

        | E GE E

        | E EQ E

        | E NE E

        | E OR E

        | E AND E

        | ID

        | NUM

        ;

%%

main()

{

  printf("Enter the exp: ");
yyparse();

}
yyer
```

```
ror()
{

   printf("\nEntered arithmetic expression is
Invalid\n\n"); }
```

OUTPUT:

```
$:lex while.l
$:yacc while.y
$:cc lex.yy.c y.tab.h
$:./a.out
```



```
monica@monica-Lenovo-E41-25:~$ lex while.l
monica@monica-Lenovo-E41-25:~$ yacc while.y
while.y: warning: 2 shift/reduce conflicts [-Wconflicts-sr]
while.y: note: rerun with option '-Wcounterexamples' to generate conflict counterexamples
monica@monica-Lenovo-E41-25:~$ cc lex.yy.c y.tab.h
In file included from while.l:3:
y.tab.c: In function ‘yyparse’:
y.tab.c:1112:16: warning: implicit declaration of function ‘yylex’; did you mean ‘yyless’? [-Wimplicit-function-declaration]
In file included from while.l:3:
y.tab.c:1253:7: warning: implicit declaration of function ‘yyerror’; did you mean ‘yyerrok’? [-Wimplicit-function-declaration]
In file included from while.l:3:
while.y: At top level:
while.y:47:1: warning: return type defaults to ‘int’ [-Wimplicit-int]
   47 | main()
      | ^~~~
while.y:52:1: warning: return type defaults to ‘int’ [-Wimplicit-int]
   52 | yyerror()
      | ^~~~~~~
while.l:20:1: warning: return type defaults to ‘int’ [-Wimplicit-int]
   20 | yywrap()
      | ^~~~~~
y.tab.c: In function ‘yyparse’:
y.tab.c:1112:16: warning: implicit declaration of function ‘yylex’ [-Wimplicit-function-declaration]
y.tab.c:1253:7: warning: implicit declaration of function ‘yyerror’; did you mean ‘yyerrok’? [-Wimplicit-function-declaration]
while.y: At top level:
while.y:47:1: warning: return type defaults to ‘int’ [-Wimplicit-int]
   47 | main()
      | ^~~~
while.y:52:1: warning: return type defaults to ‘int’ [-Wimplicit-int]
   52 | yyerror()
      | ^~~~~~~
monica@monica-Lenovo-E41-25:~$ ./a.out
Enter the exp: while(a>1){ b=1;}
Input accepted.
monica@monica-Lenovo-E41-25:~$
```

PROGRAM: **IF THEN ELSE**

LEX PART : if.l

```
%{

#include<stdio.h>

#include "y.tab.h"

%}

alpha [A-Za-z]

digit [0-9]

%%

[ \t\n]

if    return IF; then
return THEN; else
return ELSE;
{digit}+    return
NUM;
```

```
{alpha}({alpha}|{digit})*    return ID;

"<="    return LE;

">="    return GE;

"=="    return EQ;

"!="    return NE;

"||"    return OR;

"&&"    return AND;

.    return yytext[0];

%%

yywrap()

{}
```

YACC PART : if.y

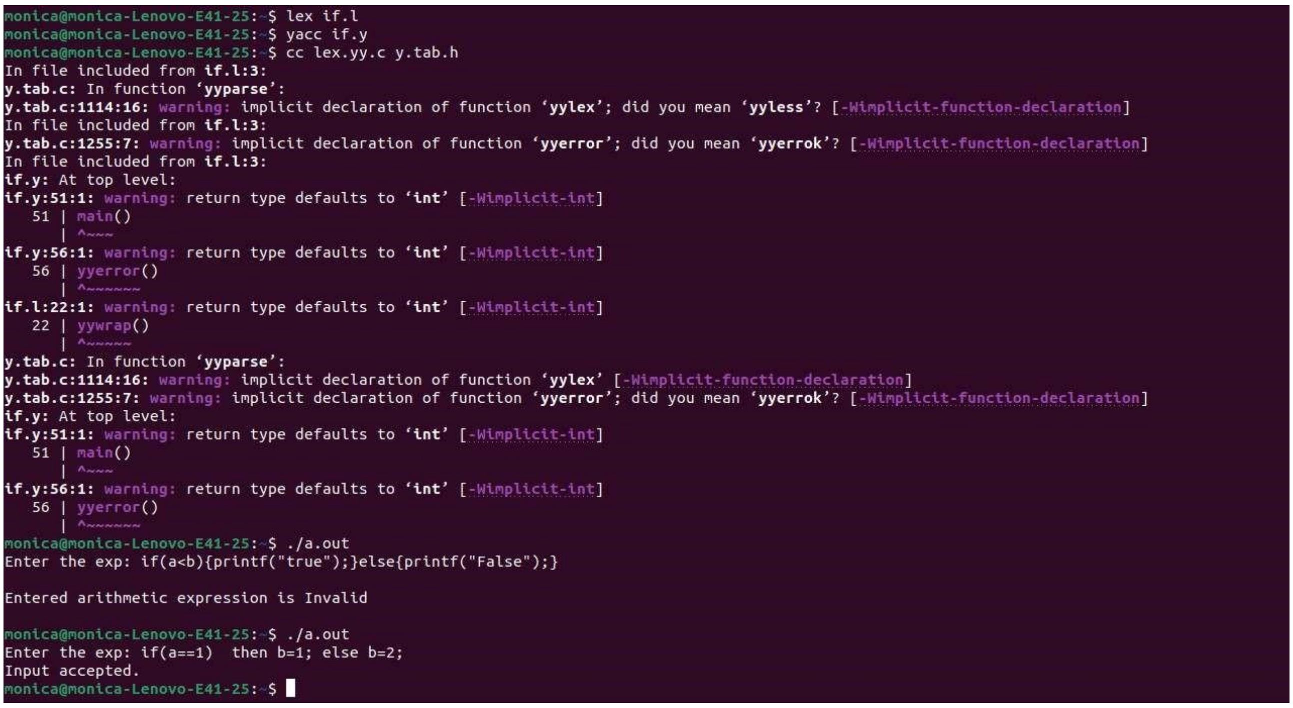
```
%{

#include <stdio.h>

#include <stdlib.h>

%}

%token ID NUM IF THEN LE GE EQ NE OR AND ELSE

%right '='

%left AND OR

%left '<' '>' LE GE EQ NE

%left '+''-'

%left '*''/'

%right UMINUS

%left '!'

%%

S     : ST {printf("Input accepted.\n");exit(0);};

ST    : IF '(' E2 ')' THEN ST1';' ELSE ST1';'

       | IF '(' E2 ')' THEN ST1';'

       ;

ST1  : ST
```

```
         | E
         ;
E    : ID'='E
     | E'+'E
     | E'-'E
     | E'*'E
     | E'/'E
     | E'<'E
     | E'>'E
     | E LE E
     | E GE E
     | E EQ E
     | E NE E
     | E OR E
     | E AND E
     | ID
     | NUM
     ;
E2   : E'<'E
     | E'>'E
     | E LE E
     | E GE E
     | E EQ E
     | E NE E
     | E OR E
     | E AND E
     | ID
     | NUM
     ;
%%
```

```
main()

{

  printf("Enter the exp: ");
yyparse();

}
yyer
ror()
{

   printf("\nEntered arithmetic expression is
Invalid\n\n"); }
```

OUTPUT:

```
$:lex if.l
$:yacc if.y
$:cc lex.yy.c y.tab.h
$:./a.out
```



```
monica@monica-Lenovo-E41-25:~$ lex if.l
monica@monica-Lenovo-E41-25:~$ yacc if.y
monica@monica-Lenovo-E41-25:~$ cc lex.yy.c y.tab.h
In file included from if.l:3:
y.tab.c: In function 'yyparse':
y.tab.c:1114:16: warning: implicit declaration of function 'yylex'; did you mean 'yyless'? [-Wimplicit-function-declaration]
In file included from if.l:3:
y.tab.c:1255:7: warning: implicit declaration of function 'yyerror'; did you mean 'yyerrok'? [-Wimplicit-function-declaration]
In file included from if.l:3:
if.y: At top level:
if.y:51:1: warning: return type defaults to 'int' [-Wimplicit-int]
   51 | main()
      | ^~~~
if.y:56:1: warning: return type defaults to 'int' [-Wimplicit-int]
   56 | yyerror()
      | ^~~~~~~
if.l:22:1: warning: return type defaults to 'int' [-Wimplicit-int]
   22 | yywrap()
      | ^~~~~~
y.tab.c: In function 'yyparse':
y.tab.c:1114:16: warning: implicit declaration of function 'yylex' [-Wimplicit-function-declaration]
y.tab.c:1255:7: warning: implicit declaration of function 'yyerror'; did you mean 'yyerrok'? [-Wimplicit-function-declaration]
if.y: At top level:
if.y:51:1: warning: return type defaults to 'int' [-Wimplicit-int]
   51 | main()
      | ^~~~
if.y:56:1: warning: return type defaults to 'int' [-Wimplicit-int]
   56 | yyerror()
      | ^~~~~~~
monica@monica-Lenovo-E41-25:~$ ./a.out
Enter the exp: if(a<b){printf("true");}else{printf("False");}

Entered arithmetic expression is Invalid

monica@monica-Lenovo-E41-25:~$ ./a.out
Enter the exp: if(a==1)  then b=1; else b=2;
Input accepted.
monica@monica-Lenovo-E41-25:~$
```

PROGRAM: **IF THEN ELSEIF THEN ELSE**

LEX PART : elseif.l

```
%{

#include<stdio.h>

#include "y.tab.h"

%}

alpha [A-Za-z]
```

```
digit [0-9]

%%

[ \t\n]

if      return  IF; then
return   THEN;   else
return   ELSE;   elseif
return           ELSEIF;
{digit}+         return
NUM;
{alpha}({alpha}|{digit})*    return ID;

"<="    return LE;

">="    return GE;

"=="    return EQ;

"!="    return NE;

"||"    return OR;

"&&"    return AND;

.    return yytext[0];

%%

yywra
p() {}
```

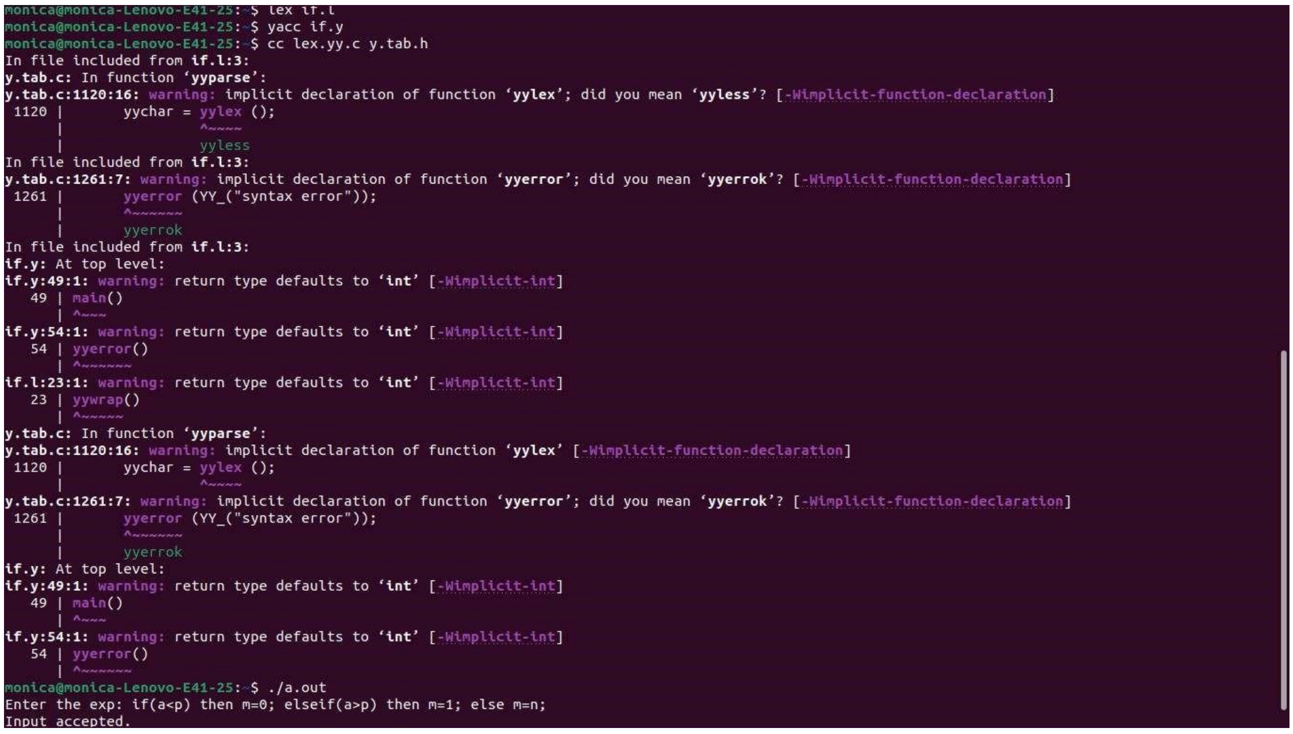
YACC PART : elseif.y

```
%{

#include <stdio.h>

#include <stdlib.h>

%}

%token ID NUM IF THEN LE GE EQ NE OR AND ELSE

%right '='

%left AND OR

%left '<' '>' LE GE EQ NE

%left '+''-'
```

```
%left '*''/'

%right UMINUS

%left '!'

%%

S      : ST {printf("Input accepted.\n");exit(0);};

ST    : IF '(' E2 ')' THEN ST1';' ELSEIF '(' E2 ')' THEN ST1';' ELSE ST1';'

        | IF '(' E2 ')' THEN ST1';'

        ;

ST1  : ST

        | E

        ;

E    : ID'='E

      | E'+'E

      | E'-'E

      | E'*'E

      | E'/'E

      | E'<'E

      | E'>'E

      | E LE E

      | E GE E

      | E EQ E

      | E NE E

      | E OR E

      | E AND E

      | ID

      | NUM

      ;

E2  : E'<'E

      | E'>'E

      | E LE E
      | E GE E

      | E EQ E

      | E NE E

      | E OR E

      | E AND E

      | ID

      | NUM

      ;

%%

main()

{

  printf("Enter the exp: ");
yyparse();

}
yyer
ror()
{

   printf("\nEntered arithmetic expression is
Invalid\n\n"); }
```

OUTPUT:

```
$:lex elseif.l
$:yacc elseif.y
$:cc lex.yy.c y.tab.h
$:./a.out
```



```
monica@monica-Lenovo-E41-25:~$ lex if.l
monica@monica-Lenovo-E41-25:~$ yacc if.y
monica@monica-Lenovo-E41-25:~$ cc lex.yy.c y.tab.h
In file included from if.l:3:
y.tab.c: In function ‘yyparse’:
y.tab.c:1120:16: warning: implicit declaration of function ‘yylex’; did you mean ‘yyless’? [-Wimplicit-function-declaration]
 1120 |       yychar = yylex ();
      |                ^~~~~
      |                yyless
In file included from if.l:3:
y.tab.c:1261:7: warning: implicit declaration of function ‘yyerror’; did you mean ‘yyerrok’? [-Wimplicit-function-declaration]
 1261 |       yyerror (YY_("syntax error"));
      |       ^~~~~~~
      |       yyerrok
In file included from if.l:3:
if.y: At top level:
if.y:49:1: warning: return type defaults to ‘int’ [-Wimplicit-int]
   49 | main()
      | ^~~~
if.y:54:1: warning: return type defaults to ‘int’ [-Wimplicit-int]
   54 | yyerror()
      | ^~~~~~~
if.l:23:1: warning: return type defaults to ‘int’ [-Wimplicit-int]
   23 | yywrap()
      | ^~~~~~
y.tab.c: In function ‘yyparse’:
y.tab.c:1120:16: warning: implicit declaration of function ‘yylex’ [-Wimplicit-function-declaration]
 1120 |       yychar = yylex ();
      |                ^~~~~
y.tab.c:1261:7: warning: implicit declaration of function ‘yyerror’; did you mean ‘yyerrok’? [-Wimplicit-function-declaration]
 1261 |       yyerror (YY_("syntax error"));
      |       ^~~~~~~
      |       yyerrok
if.y: At top level:
if.y:49:1: warning: return type defaults to ‘int’ [-Wimplicit-int]
   49 | main()
      | ^~~~
if.y:54:1: warning: return type defaults to ‘int’ [-Wimplicit-int]
   54 | yyerror()
      | ^~~~~~~
monica@monica-Lenovo-E41-25:~$ ./a.out
Enter the exp: if(a<p) then m=0; elseif(a>p) then m=1; else m=n;
Input accepted.
```

PROGRAM: **SWITCH**

LEX PART : switch.l

```
%{

#include<stdio.h>

#include "y.tab.h"

%}

alpha [A-Za-z]

digit [0-9]

%%

[ \n\t] if              return
IF; then           return
THEN; while          return
WHILE; switch        return
SWITCH; case
return CASE; default
return DEFAULT; break
return BREAK;

{digit}+      return NUM;

{alpha}({alpha}|{digit})* return ID;

"<="          return LE;

">="          return GE;
```

```
"=="          return EQ;
"!="          return NE;

"&&"          return AND;

"||"            return OR;

.             return yytext[0];

%%

yywrap()

{}
```

YACC PART : switch.y

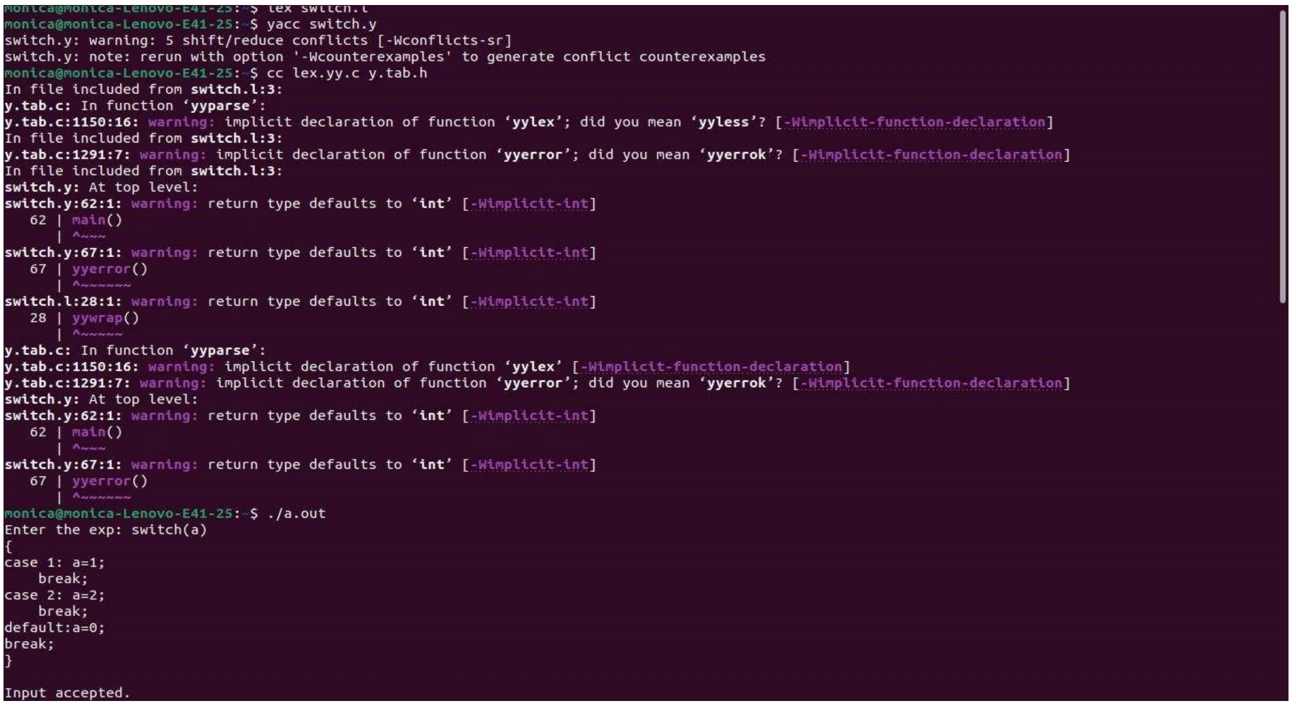
```
%{

#include<stdio.h>

#include<stdlib.h>

%}

%token ID NUM SWITCH CASE DEFAULT BREAK LE GE EQ NE AND OR IF THEN WHILE

%right '='

%left AND OR

%left '<' '>' LE GE EQ NE

%left '+''-'

%left '*''/'

%right UMINUS

%left '!'

%%

S   :   ST{printf("\nInput accepted.\n");exit(0);};

    ;

ST  :   SWITCH'('ID')''{'B'}'

    ;

B    :   C
     |   C D

    ;

C    :   C C
   |   CASE NUM':'ST1 BREAK';'
```

```
    ;

D     :    DEFAULT':'ST1 BREAK';'
     |    DEFAULT':'ST1

    ;

ST1    :    WHILE'('E2')' E';'

     |    IF'('E2')'THEN E';'

     |    ST1 ST1

     |    E';'

    ;

E2    :    E'<'E

     |    E'>'E

     |    E LE E

     |    E GE E

     |    E EQ E

     |    E NE E

     |    E AND E

     |    E OR E

    ;

E    :    ID'='E

     |    E'+'E

     |    E'-'E

     |    E'*'E

     |    E'/'E

     |    E'<'E

     |    E'>'E

     |    E LE E

     |    E GE E

     |    E EQ E

     |    E NE E

     |    E AND E
```

```
   |   E OR E

   |   ID

   |   NUM

   ;

%%

main()

{

  printf("Enter the exp: ");
yyparse();

}
yyer
ror()
{

   printf("\nEntered arithmetic expression is
Invalid\n\n");

}
```

OUTPUT:

```
$:lex switch.l
$:yacc switch.y
$:cc lex.yy.c y.tab.h
$:./a.out
```



```
monica@monica-Lenovo-E41-25:~$ lex switch.l
monica@monica-Lenovo-E41-25:~$ yacc switch.y
switch.y: warning: 5 shift/reduce conflicts [-Wconflicts-sr]
switch.y: note: rerun with option '-Wcounterexamples' to generate conflict counterexamples
monica@monica-Lenovo-E41-25:~$ cc lex.yy.c y.tab.h
In file included from switch.l:3:
y.tab.c: In function ‘yyparse’:
y.tab.c:1150:16: warning: implicit declaration of function ‘yylex’; did you mean ‘yyless’? [-Wimplicit-function-declaration]
In file included from switch.l:3:
y.tab.c:1291:7: warning: implicit declaration of function ‘yyerror’; did you mean ‘yyerrok’? [-Wimplicit-function-declaration]
In file included from switch.l:3:
switch.y: At top level:
switch.y:62:1: warning: return type defaults to ‘int’ [-Wimplicit-int]
   62 | main()
      | ^~~~
switch.y:67:1: warning: return type defaults to ‘int’ [-Wimplicit-int]
   67 | yyerror()
      | ^~~~~~~
switch.l:28:1: warning: return type defaults to ‘int’ [-Wimplicit-int]
   28 | yywrap()
      | ^~~~~~
y.tab.c: In function ‘yyparse’:
y.tab.c:1150:16: warning: implicit declaration of function ‘yylex’ [-Wimplicit-function-declaration]
y.tab.c:1291:7: warning: implicit declaration of function ‘yyerror’; did you mean ‘yyerrok’? [-Wimplicit-function-declaration]
switch.y: At top level:
switch.y:62:1: warning: return type defaults to ‘int’ [-Wimplicit-int]
   62 | main()
      | ^~~~
switch.y:67:1: warning: return type defaults to ‘int’ [-Wimplicit-int]
   67 | yyerror()
      | ^~~~~~~
monica@monica-Lenovo-E41-25:~$ ./a.out
Enter the exp: switch(a)
{
case 1: a=1;
    break;
case 2: a=2;
    break;
default:a=0;
break;
}

Input accepted.
```

RESULT:

Thus, the given program to recognize valid control structures is executed successfully.

Exercise-3(d):

Implementation of calculator using lex and yacc.

AIM:

To write a lex and yacc program to implement a 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 calc.l.
Step 4: By use of Yacc program,all the Arithmetic operations are done such as +,-,*,/. **Step 5:** Display error is persist.
Step 6: Verify the output.
Step 8: End.

PROGRAM:

LEX PART : calc.l

```
%{
        #include<stdlib.h>
        #include "y.tab.h"
        extern int yylval;
%}
%%
[0-9]+ {yylval=atoi(yytext); return NUMBER;}
[\n] return 0;
[\t];
. return yytext[0];
%%
yywrap()
{ return 0;}
```

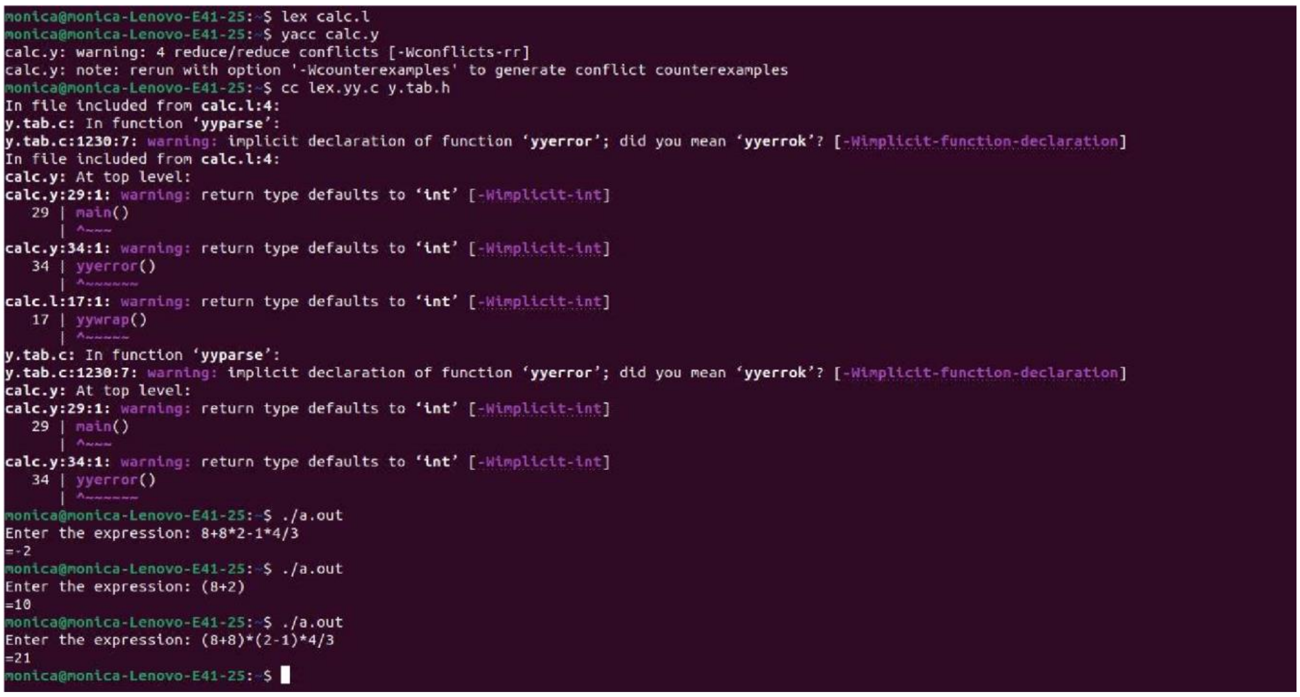
YACC PART : calc.y

```
%{

        #include<stdio.h>

        int yylex(void);

%}

%token NAME NUMBER

%left GE LE NE EQ '<' '>' '%'

%left '-' '+'

%left '*' '/'

%nonassoc UMINUS
```

```
%%
statement : NAME '=' exp
            |exp {printf("=%d\n",$1);}
            ;
exp : NUMBER {$$ == $1;}
            |exp '+' exp {$$ = $1 + $3 ;}
            |exp '-' exp {$$ = $1 - $3 ;}
            |exp '*' exp {$$ = $1 * $3 ;}
            |exp '/' exp {$$ = $1 / $3 ;}
            |exp '-' exp %prec UMINUS {$$ = -$2 ;}
            |'(' exp ')' {$$ = $2 ;}
;
%%
m
ai
n(
) {
printf("Enter the
expression: "); yyparse(); }
yyerror()
{ printf("\nError
Occured\n"); }
```

OUTPUT:

```
$:lex calc.l
$:yacc calc.y
$:cc lex.yy.c y.tab.h
$:./a.out
```



```
monica@monica-Lenovo-E41-25:~$ lex calc.l
monica@monica-Lenovo-E41-25:~$ yacc calc.y
calc.y: warning: 4 reduce/reduce conflicts [-Wconflicts-rr]
calc.y: note: rerun with option '-Wcounterexamples' to generate conflict counterexamples
monica@monica-Lenovo-E41-25:~$ cc lex.yy.c y.tab.h
In file included from calc.l:4:
y.tab.c: In function ‘yyparse’:
y.tab.c:1230:7: warning: implicit declaration of function ‘yyerror’; did you mean ‘yyerrok’? [-Wimplicit-function-declaration]
In file included from calc.l:4:
calc.y: At top level:
calc.y:29:1: warning: return type defaults to ‘int’ [-Wimplicit-int]
   29 | main()
      | ^~~~
calc.y:34:1: warning: return type defaults to ‘int’ [-Wimplicit-int]
   34 | yyerror()
      | ^~~~~~~
calc.l:17:1: warning: return type defaults to ‘int’ [-Wimplicit-int]
   17 | yywrap()
      | ^~~~~~
y.tab.c: In function ‘yyparse’:
y.tab.c:1230:7: warning: implicit declaration of function ‘yyerror’; did you mean ‘yyerrok’? [-Wimplicit-function-declaration]
calc.y: At top level:
calc.y:29:1: warning: return type defaults to ‘int’ [-Wimplicit-int]
   29 | main()
      | ^~~~
calc.y:34:1: warning: return type defaults to ‘int’ [-Wimplicit-int]
   34 | yyerror()
      | ^~~~~~~
monica@monica-Lenovo-E41-25:~$ ./a.out
Enter the expression: 8+8*2-1*4/3
=-2
monica@monica-Lenovo-E41-25:~$ ./a.out
Enter the expression: (8+2)
=10
monica@monica-Lenovo-E41-25:~$ ./a.out
Enter the expression: (8+8)*(2-1)*4/3
=21
monica@monica-Lenovo-E41-25:~$ 
```

RESULT:

Thus, the program to implement a calculator using lex tool is implemented and executed successfully.

Exercise-4:

Generate three address code for a simple program using LEX and YACC (Implementing Three Address Code)

Aim:

To generate three address code for a simple program using LEX and YACC.

Algorithm:

LEX:

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

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

lexemes.

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:

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

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

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

4. Call yyparse() to initiate the parsing process.

5. yyerror() function is called when all productions in the grammar in second section

doesn't match to the input statement.

6. Make_symtab_entry() function to make the symbol table entry.

LEX program:

```
%{

#include<stdio.h>
#include<string.h>
#include "ex4.tab.h"
%}
%%
[ \n\t]+ ;
main {return MAIN;}
int|float|double|char {strcpy(yylval.string,yytext); return TYPE; }
[a-zA-Z][a-zA-Z0-9_]* {strcpy(yylval.string,yytext);return ID; }
[0-9]+ |
[0-9]+\.[0-9]+ {
 yylval.dval=atof(yytext);
 return NUMBER;
 }
. return yytext[0];
%%
int yywrap(){
return 1;
}
```

YACC program:

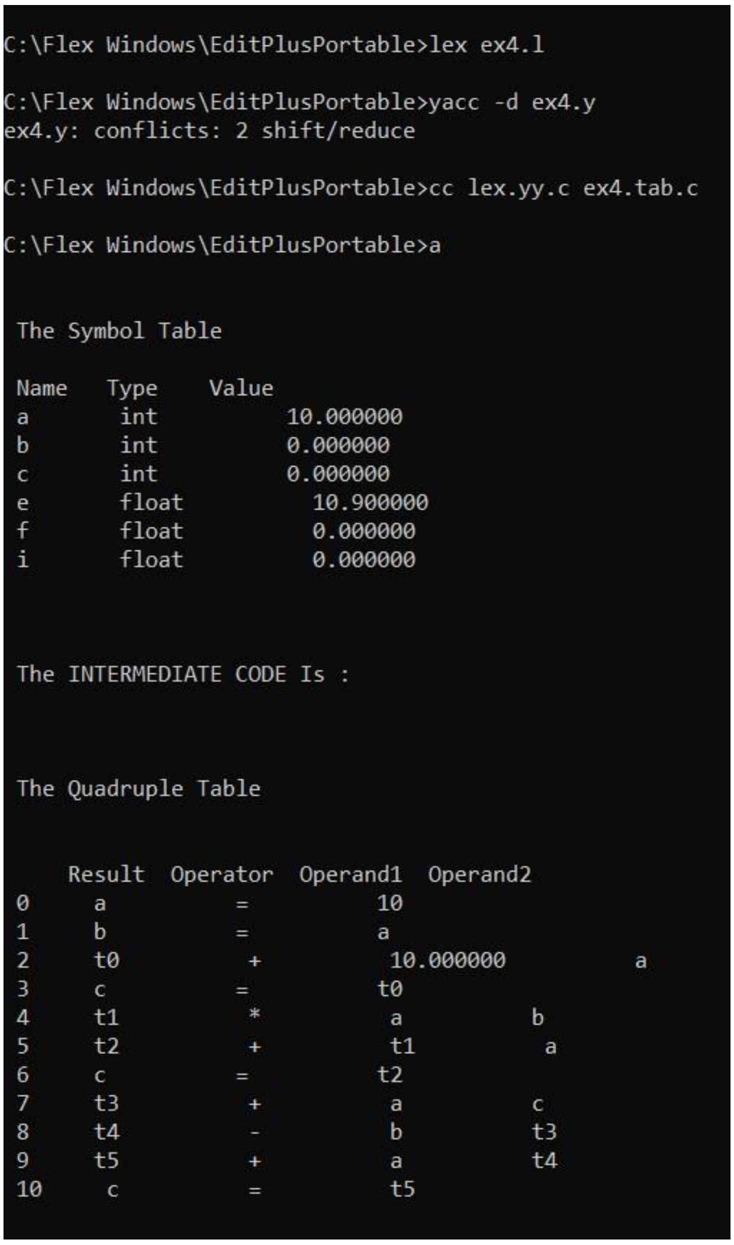
```
%{
 #include<stdio.h>
 #include<string.h>
 #include<stdlib.h>
 struct Symbol_Table
 {
 char sym_name[10];
 char sym_type[10];
 double value;
 }Sym[10];
 int sym_cnt=0;
 int Index=0;
 int temp_var=0;
 int search_symbol(char []);
 void make_symtab_entry(char [],char [],double);
 void display_sym_tab();
 void addQuadruple(char [],char [],char [],char []);
 void display_Quadruple();
 void push(char*);
 char* pop();
 struct Quadruple
 {
 char operator[5];
 char operand1[10];
 char operand2[10];
 char result[10];
 }QUAD[25];
 struct Stack
 {
 char *items[10];
 int top;
 }Stk;
%}
```

```
%union
{
 int ival;
 double dval;
 char string[10];
}
%token <dval> NUMBER
%token <string> TYPE
%token <string> ID
%type <string> varlist
%type <string> expr
%token MAIN
%left '+' '-'
%left '*' '/'
%%
program:MAIN '('')''{' body '}'
;
body: varstmt stmtlist
;
varstmt: vardecl varstmt|
;
vardecl:TYPE varlist ';'
;
varlist: varlist ',' ID { int i;
 i=search_symbol($3);
 if(i!=-1)
 printf("\n Multiple Declaration of Variable");
 else
 make_symtab_entry($3,$<string>0,0);
 }
| ID'='NUMBER {
 int i;
 i=search_symbol($1);
 if(i!=-1)
 printf("\n Multiple Declaration of Variable");
 else
 make_symtab_entry($1,$<string>0,$3);
 }
|varlist ',' ID '=' NUMBER {
 int i;
 i=search_symbol($3);
 if(i!=-1)
 printf("\n Multiple Declaration of Variable");
 else
 make_symtab_entry($3,$<string>0,$5);
 }
|ID { int i;
 i=search_symbol($1);
 if(i!=-1)
 printf("\n Multiple Declaration of Variable");
 else
 make_symtab_entry($1,$<string>0,0);
 }
;
stmtlist: stmt stmtlist|
;
stmt : ID '=' NUMBER ';' {
 int i;
 i=search_symbol($1);
 if(i==-1)
 printf("\n Undefined Variable");
 else
 {
 char temp[10];
 if(strcmp(Sym[i].sym_type,"int")==0)
 sprintf(temp,"%d",(int)$3);
 else
 snprintf(temp,10,"%f",$3);
 addQuadruple("=","",temp,$1);
 }
 }
| ID '=' ID ';'{
 int i,j;
 i=search_symbol($1);
 j=search_symbol($3);
 if(i==-1 || j==-1)
 printf("\n Undefined Variable");
 else
 addQuadruple("=","",$3,$1);
}
| ID '=' expr ';' { addQuadruple("=","",pop(),$1); }


;
expr :expr '+' expr {
 char str[5],str1[5]="t";
 sprintf(str, "%d", temp_var);
 strcat(str1,str);
 temp_var++;
 addQuadruple("+",pop(),pop(),str1);
 push(str1);
 }
|expr '-' expr {
 char str[5],str1[5]="t";
 sprintf(str, "%d", temp_var);
 strcat(str1,str);
 temp_var++;
 addQuadruple("-",pop(),pop(),str1);
 push(str1);
 }
|expr '*' expr {
 char str[5],str1[5]="t";
 sprintf(str, "%d", temp_var);
 strcat(str1,str);
 temp_var++;
 addQuadruple("*",pop(),pop(),str1);
 push(str1);
 }
|expr '/' expr {
 char str[5],str1[5]="t";
 sprintf(str, "%d", temp_var);
 strcat(str1,str);
 temp_var++;
 addQuadruple("/",pop(),pop(),str1);
 push(str1);
 }

|ID { int i;
 i=search_symbol($1);
 if(i==-1)
 printf("\n Undefined Variable");
 else
 push($1);
 }
|NUMBER { char temp[10];
 snprintf(temp,10,"%f",$1);
 push(temp);
}
;
%%
extern FILE *yyin;
```

```
int main()
{


 Stk.top = -1;
 yyin = fopen("input.txt","r");
 yyparse();
 display_sym_tab();
 printf("\n\n");
display_Quadruple();
 printf("\n\n");
 return(0);
}
int search_symbol(char sym[10])
{
 int i,flag=0;
 for(i=0;i<sym_cnt;i++)
 {
 if(strcmp(Sym[i].sym_name,sym)==0)
 {
 flag=1;
 break;
 }
 }
 if(flag==0)
 return(-1);
 else
 return(i);
}
void make_symtab_entry(char sym[10],char dtype[10],double val)
{
 strcpy(Sym[sym_cnt].sym_name,sym);
 strcpy(Sym[sym_cnt].sym_type,dtype);
 Sym[sym_cnt].value=val;
 sym_cnt++;
}
void display_sym_tab()
{
 int i;
 printf("\n\n The Symbol Table \n\n");
 printf(" Name Type Value");
 for(i=0;i<sym_cnt;i++)
 printf("\n %s %s %f",Sym[i].sym_name,Sym[i].sym_type,Sym[i].value);
}
void display_Quadruple()
{
 int i;
 printf("\n\n The INTERMEDIATE CODE Is : \n\n");
 printf("\n\n The Quadruple Table \n\n");
 printf("\n Result Operator Operand1 Operand2 ");
 for(i=0;i<Index;i++)
 printf("\n %d %s %s %s
%s",i,QUAD[i].result,QUAD[i].operator,QUAD[i].operand1,QUAD[i].operand
2);
```

```
}
int yyerror()
{
 printf("\nERROR!!\n");
 return(1);
}
void push(char *str)
{
 Stk.top++;
 Stk.items[Stk.top]=(char *)malloc(strlen(str)+1);
 strcpy(Stk.items[Stk.top],str);
}
char * pop()
{
 int i;
 if(Stk.top==-1)
 {
 printf("\nStack Empty!! \n");
 exit(0);
 }
 char *str=(char *)malloc(strlen(Stk.items[Stk.top])+1);;
strcpy(str,Stk.items[Stk.top]);
 Stk.top--;
 return(str);
}
void addQuadruple(char op[10],char op2[10],char op1[10],char res[10]){
 strcpy(QUAD[Index].operator,op);

 strcpy(QUAD[Index].operand2,op2);

 strcpy(QUAD[Index].operand1,op1);

 strcpy(QUAD[Index].result,res);

 Index++;

}
```

Output:



```
C:\Flex Windows\EditPlusPortable>lex ex4.l

C:\Flex Windows\EditPlusPortable>yacc -d ex4.y
ex4.y: conflicts: 2 shift/reduce

C:\Flex Windows\EditPlusPortable>cc lex.yy.c ex4.tab.c

C:\Flex Windows\EditPlusPortable>a


 The Symbol Table

 Name    Type     Value
 a        int          10.000000
 b        int          0.000000
 c        int          0.000000
 e        float          10.900000
 f        float          0.000000
 i        float          0.000000



 The INTERMEDIATE CODE Is :



 The Quadruple Table


     Result  Operator  Operand1  Operand2
 0     a         =         10
 1     b         =         a
 2     t0         +         10.000000         a
 3     c         =         t0
 4     t1         *         a         b
 5     t2         +         t1         a
 6     c         =         t2
 7     t3         +         a         c
 8     t4         -         b         t3
 9     t5         +         a         t4
 10     c         =         t5
```

Result:

Thus the three address code was generated successfully for a simple program using LEX and YACC.

Exercise-5:

Implement type checking using Lex and Yacc.

AIM:

To write a C program to implement type checking

ALGORITHM:

Step1: Track the global scope type information (e.g. classes and their members)

Step2: Determine the type of expressions recursively, i.e. bottom-up, passing the resulting types upwards.

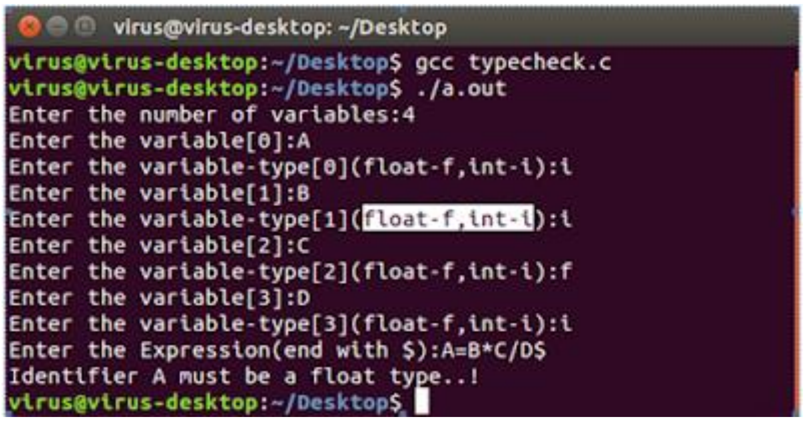
Step3: If type found correct, do the operation

Step4: Type mismatches, semantic error will be notified

PROGRAM CODE:

```
//To implement type checking

#include<stdio.h>

#include<stdlib.h>

int main()

{

int n,i,k,flag=0;

char vari[15],typ[15],b[15],c;

printf("Enter the number of variables:");

scanf(" %d",&n);

for(i=0;i<n;i++)

{

printf("Enter the variable[%d]:",i);

scanf(" %c",&vari[i]);

printf("Enter the variable-type[%d](float-f,int-i):",i);

scanf(" %c",&typ[i]);

if(typ[i]=='f')

flag=1;

}

printf("Enter the Expression(end with $):");

i=0;
getchar();
while((c=getchar())!='$')
{
b[i]=c;
i++;  }
k=i;
for(i=0;i<k;i++)
{
if(b[i]=='/')
{
flag=1;
break;  }  }
for(i=0;i<n;i++)
{
if(b[0]==vari[i])
{
if(flag==1)
{
if(typ[i]=='f')
{  printf("\nthe datatype is correctly defined..!\n");
break;  }
else
{  printf("Identifier %c must be a float type..!\n",vari[i]);
break;  }  }
else
{  printf("\nthe datatype is correctly defined..!\n");
break;  }  }
}
return 0;
}
```

OUTPUT:



```
virus@virus-desktop: ~/Desktop
virus@virus-desktop:~/Desktop$ gcc typecheck.c
virus@virus-desktop:~/Desktop$ ./a.out
Enter the number of variables:4
Enter the variable[0]:A
Enter the variable-type[0](float-f,int-i):i
Enter the variable[1]:B
Enter the variable-type[1](float-f,int-i):i
Enter the variable[2]:C
Enter the variable-type[2](float-f,int-i):f
Enter the variable[3]:D
Enter the variable-type[3](float-f,int-i):i
Enter the Expression(end with $):A=B*C/D$
Identifier A must be a float type..!
virus@virus-desktop:~/Desktop$
```

Result:

Thus the LEX and YACC program for implementing type checking was written and executed successfully.

Exercise-6:

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

OBJECTIVE OF THE EXPERIMENT :

To write a C program for implementing simple code optimization technique using constant folding.

DESCRIPTION:

Constant folding and constant propagation are related compiler optimizations used by many modern compilers. An advanced form of constant propagation known as sparse conditional constant propagation can more accurately propagate constants and simultaneously remove dead code.

PROCEDURE:

1) Start the Program
2) Get the input as a source program
3) Process the instructions inside the input file
4) Apply transformations on loops, procedure calls, and address calculations
5) Code generation will occur
6) Use registers and select appropriate instructions
7) Perform Peephole optimization

Program:

```
#include<stdio.h>
#include<string.h>
#include<conio.h>
#include<stdlib.h>
#include<ctype.h>
struct ConstFold
{
char new_Str[10];
char str[10];
}
Opt_Data[20];
void ReadInput(char Buffer[],FILE *Out_file);
```

```
int Gen_token(char str[],char
Tokens[][10]); int New_Index=0;
int main()
{
FILE *In_file,*Out_file;
char Buffer[100],ch;
int i=0;
In_file = fopen("code2.txt","r");
Out_file = fopen("output1.txt","w");
clrscr();
while(1)
{
ch =
fgetc(In_file); i=0;
while(1)
{
if(ch == '\n')
break;
Buffer[i++]=ch; ch
= fgetc(In_file);
if(ch == EOF)
break;
}//End while
if(ch ==EOF)
break;
Buffer[i]='\0';
ReadInput(Buffer, Out_file);//writing to the output file
}//End while
return 0;
}//End main
void ReadInput(char Buffer[],FILE *Out_file)
{
char temp[100],Token[10][10];
int n,i,j,flag=0;
strcpy(temp,Buffer);
n= Gen_token(temp,Token);
for(i=0;i<n;i++)
{
if(!strcmp(Token[i],"="))
{
if(isdigit(Token[i+1][0])||Token[i+1][0] == '.')
{
```

```
/*If yes then saving that number and its variable
In the Opt_Data array*/
flag=1;
strcpy(Opt_Data[New_Index].new_Str,Token[i-1]);
strcpy(Opt_Data[New_Index++].str,Token[i+1]);
}//End if
}//End if
}//End for
if(!flag)
{
for(i=0;i<New_Index;i++)
{
for(j=0;j<n;j++)
{
if(!strcmp(Opt_Data[i].new_Str,Token[j]))
strcpy(Token[j],Opt_Data[i].str);
}//End for
}//End for
}//End if
fflush(Out_file);
strcpy(temp," ");
for(i=0;i<n;i++) /*Loop to obtain complete tokens*/
{
strcat(temp,Token[i]);
if(Token[i+1][0]!=','||Token[i+1][0] !=
',') strcat(temp," ");
}//End for
strcat(temp,"\n\0");
fwrite(&temp,strlen(temp),1,Out_file);
}
/*The Gen_Token function breaks the input line into tokens*/
int Gen_token(char str[], char Token[][10])
{
int  i=0;
int  j=0;
int k=0;
while(str[k]!='\0')
{
j=0;
while(str[k] ==' '|| str[k]
=='\t') k++;
```

```
while(str[k]!=' '&& str[k]!='\0'&& str[k]!= '=' && str[k] != '/'&& str[k]!= '+' && str[k] != '-'&& str[k]!= '*'
&& str[k] != ',' && str[k]!= ';')
Token[i][j++] = str[k++]; Token[i++][j] =
'\0';
if(str[k] == '='|| str[k] == '/'|| str[k] == '+'|| str[k] == '-'|| str[k] == '*'|| str[k] == '*'|| str[k] == ','|| str[k] ==
';')
{
Token[i][0] = str[k++]; Token[i++][1] =
'\0'; }//End if
if(str[k] == '\0') break;
}//End while return i;
}
```

SAMPLE OUTPUT: Input file:
code.txt

```
#include<stdio.h>
main()
{
float pi=3.14,r,a; a = pi*r*r;
printf("a = %f",a); return 0;
}
```

OUTPUT FILE: output1.txt

```
#include<stdio.h>
 main()
 {
 float pi = 3.14, r, a ; a = 3.14 * r * r ;
 printf(" = %f", a) ; return 0 ;
```

CONCLUSION:

Thus the C program for implementing simple code optimization technique was written and executed successfully.

Exercise-7:

Implement back-end of the compiler for which the three address code is given as input and the 8086-assembly language code is produced as output. The target assembly instructions can be simple move , add, sub, jump. Also simple addressing modes are used.

OBJECTIVE OF THE EXPERIMENT:

To write a C program for implementing back end of the compiler which takes three address codes as input and produces 8086 assembly language instruction.

DESCRIPTION:

Modern processors have only a limited number of register. Although some processors, such as the x86, can perform operations directly on memory locations, we will for now assume only register operations. Some processors (e.g., the MIPS architecture) use three-address instructions. However, some processors permit only two addresses; the result overwrites one of the sources. With these assumptions, code something like the following would be produced for our example, after first assigning memory locations to id1 and id2.

```
  LD R1, id2
  ADDF R1, R1, #3.0  // add float
 RTOI R2, R1         // real to int
 ST id1, R2
```

PROCEDURE:

1) Start the program
2) Open the input file
3) Enter the intermediate code as an input to the program
4) Apply conditions for checking the keywords in the intermediate code
5) Analyze each instruction in switch case
6) After generating machine code, copy it to the output file
7) Stop the program

PROGRAM:

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
#include<string.h>
int label[20];
int no =0;
int main()
{
```

```
FILE *fp1,*fp2;
int check_label(int n);
char fname[10],op[10],ch;
char operand1[8],operand2[8],result[8];
int i=0,j=0;
clrscr();
printf("\n Enter filename of the intermediate code");
scanf("%s",&fname);
fp1=fopen(fname,"r");
fp2=fopen("target.txt","w");
if(fp1==NULL || fp2==NULL)
{
printf("\n Error Opening the
file"); getch();
exit(0);
}
while(!feof(fp1))
{
fprintf(fp2,"\n");
fscanf(fp1,"%s",op);
i++;
if(check_label(i))
{
fprintf(fp2,"\nlabel#%d:",i);
}
if(strcmp(op,"print")==0)
{
fscanf(fp1,"%s",result);
fprintf(fp2,"\n\t OUT %s",result);
}
if(strcmp(op,"goto")==0)
{
fscanf(fp1,"%s",operand2);
fprintf(fp2,"\n\t JMP label#%s",operand2);
label[no++] = atoi(operand2);
}
if(strcmp(op,"[]=")==0)
{
fscanf(fp1,"%s%s%s",operand1,operand2,result);
fprintf(fp2,"\n\tSTORE%s[%s],%s",operand1,operand2,result);
}
if(strcmp(op,"uminus")==0)
```

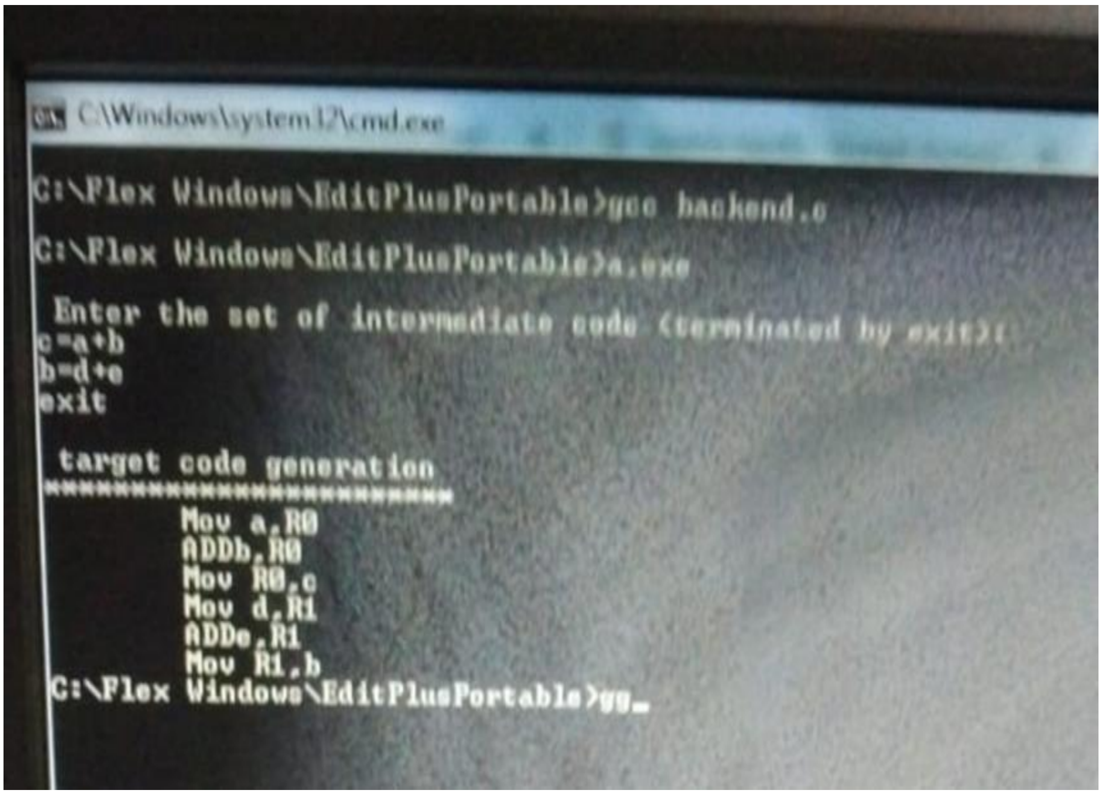
```
{
fscanf(fp1,"%s%s",operand1,result);
fprintf(fp2,"\n\t MOV R1,-%s",operand1);
fprintf(fp2,"\n\t MOV %s,R1",result);
}
switch(op[0])
{
case '*':
fscanf(fp1,"%s%s%s",operand1,operand2,result);
fprintf(fp2,"\n\t MOV R0,%s",operand1);
fprintf(fp2,"\n\t MOV R1,%s",operand2);
fprintf(fp2,"\n\t MUL R0,R1");
fprintf(fp2,"\n\t MOV %s,R0",result);
break;
case '+':
fscanf(fp1,"%s%s%s",operand1,operand2,result);
fprintf(fp2,"\n\t MOV R0,%s",operand1);
fprintf(fp2,"\n\t MOV R1,%s",operand2);
fprintf(fp2,"\n\t ADD R0,R1");
fprintf(fp2,"\n\t MOV %s,R0",result);
break;
case '-':
fscanf(fp1,"%s%s%s",operand1,operand2,result);
fprintf(fp2,"\n\t MOV R0,%s",operand1);
fprintf(fp2,"\n\t MOV R1,%s",operand2);
fprintf(fp2,"\n\t SUB R0,R1");
fprintf(fp2,"\n\t MOV %s,R0",result);
break;
case '/':
fscanf(fp1,"%s%s%s",operand1,operand2,result);
fprintf(fp2,"\n\t MOV R0,%s",operand1);
fprintf(fp2,"\n\t MOV R1,%s",operand2);
fprintf(fp2,"\n\t DIV R0,R1");
fprintf(fp2,"\n\t MOV %s,R0",result);
break;
case '=':
fscanf(fp1,"%s%s",operand1,result);
fprintf(fp2,"\n\t MOV
%s%s",result,operand1); break;
case
'>': j++;
fscanf(fp1,"%s%s%s",operand1,operand2,result);
fprintf(fp2,"\n\t JGT %s%s
label#%s",operand1,operand2,result); label[no++]=atoi(result);
break;
case '<':
fscanf(fp1,"%s%s%s",operand1,operand2,result);
fprintf(fp2,"\n\t JLT %s%s
label#%s",operand1,operand2,result); label[no++]=atoi(result);
break;
}
}
fclose(fp2);
fclose(fp1);
fp2=fopen("target.txt","r");
if(fp2==NULL)
{
printf("\n Error in opening the
file"); getch();
exit(0);
}
do
{
ch=fgetc(fp2);
printf("%c",ch);
}
while(ch!=EOF);
fclose(fp2);
getch();
return 0;
}
int check_label(int k)
{
int i;
for(i=0;i<no;i++)
{
if(k==label[i])
return 1;
}
return 0;
}
```

SAMPLE OUTPUT:
Input file: in.txt

```
[ ]=a.. i 1
*x y t1
+t1 z t2
> t2 num 6
goto 8
+x x x
+y y y
print x
=y z
print z
```

Output file: taget.txt

```
MOV R0,y
MOV R1,t1
MUL R0,R1
MOV +t1,R0
JGT t2num label#6
JMP label#8
MOV R0,x
MOV R1,x
ADD R0,R1
MOV +y,R0
OUT x
MOV printz
```



```
C:\Windows\system32\cmd.exe

C:\Flex Windows\EditPlusPortable>gcc backend.c

C:\Flex Windows\EditPlusPortable>a.exe

 Enter the set of intermediate code (terminated by exit):
c=a+b
b=d+e
exit

 target code generation
************************
        Mov a,R0
        ADDb,R0
        Mov R0,c
        Mov d,R1
        ADDe,R1
        Mov R1,b
 C:\Flex Windows\EditPlusPortable>gg_
```

CONCLUSION:

Thus the C program for implementing back end of the compiler which takes the three address code as input and produces 8086 assembly language instruction was written and executed successfully.