MEENAKSHI COLLEGE OF ENGINEERING

No 12, Vembuli Amman Koil Street, West K.K. Nagar
Chennai – 600 078

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

CS3501 – COMPILER DESIGN LABORATORY

FACULTY MANUAL

(2024-2025)

MEENAKSHI COLLEGE OF ENGINEERING

No 12, Vembuli Amman Koil Street, West K.K. Nagar
Chennai – 600 078

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

CS3501 – COMPILER DESIGN LABORATORY

FACULTY MANUAL

YEAR / SEM / SEC: III / V / (A&B)

FACULTY INCHARGES

Ms. Abitha V K Lija, AP/CSE

Ms. Madhumitha R, AP/CSE

MEENAKSHI COLLEGE OF ENGINEERING

No 12, Vembuli Amman Koil Street, West K.K. Nagar,
Chennai – 600 078

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

CS3501 COMPILER DESIGN LABORATORY

VISION STATEMENT

To become a centre of excellence in computer science with strong research and teaching environment that adapts swiftly to the real world challenges of the current era.

MISSION STATEMENT

- To provide quality education and generate new knowledge by engaging in research and by offering undergraduate, postgraduate programmes leading to professionals in diversified domain of industries, government and academia.
- To promote teaching and learning process resulting in the integration of research results and innovations by applying new technologies that leads to useful product.
- To provide ethical and value based global education by promoting activities addressing the societal needs.
- To provide placement opportunities and to motivate for higher studies.

COURSE OBJECTIVES:

- To understand intermediate code generation and run-time environment.
- To learn to implement the front-end of the compiler.
- To learn to implement code generator.
- To learn to implement code optimization.

UNIVERSITY SYLLABUS:

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.

COURSE OUTCOMES:

CO1: Understand the techniques in different phases of a compiler.
CO2: Design a lexical analyser for a sample language and learn to use the LEX tool.
CO3: Apply different parsing algorithms to develop a parser and learn to use YACC tool
CO4: Understand semantics rules (SDT), intermediate code generation and run-time environment.
CO5: Implement code generation and apply code optimization techniques.

MEENAKSHI COLLEGE OF ENGINEERING

No 12, Vembuli Amman Koil Street, West K.K. Nagar,
Chennai – 600 078

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

CS3501 COMPILER DESIGN LABORATORY

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.

ADDITIONAL EXPERIMENTS:

8. Implement error detection and recovery strategies in lexer and parser.
9. Create a simple interpreter for the language recognized by lexer and parser.

Ex. No: 1

DEVELOP A LEXICAL ANALYZER TO RECOGNIZE FEW PATTERNS IN C AND IMPLEMENTATION OF A SYMBOL TABLE.

PROBLEM STATEMENT:

To develop a lexical analyzer to recognize few patterns in C (Ex. Identifiers, Constants, Comments, Operators etc.) and Implementation of a symbol table.

ALGORITHM:

Step1: Start the program.

Step 2: Read the input string.

Step 3: Check whether the string is identifier, operator, and symbol by using the rules of identifier and keywords using lex tool using the following steps.

Step 4: If the string starts with letter followed by any number of letter or digit then display it as a identifier.

Step 5: If it is operator print it as a operator.

Step 6: If it is number print it as a number.

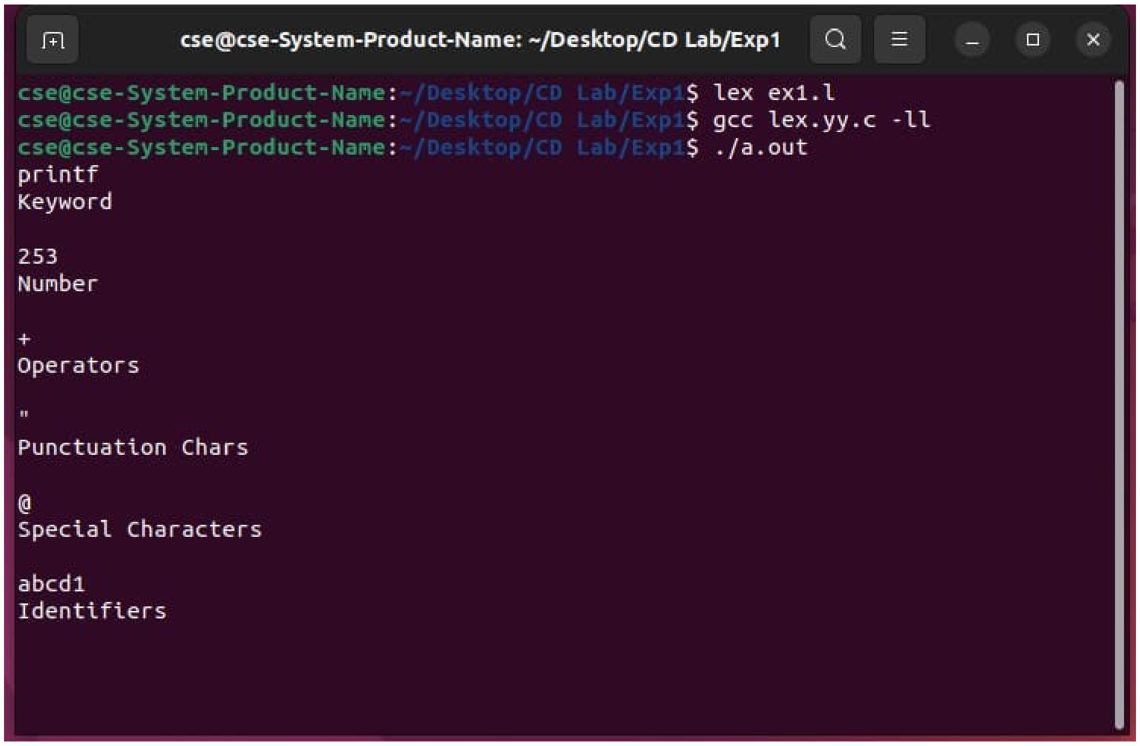
Step 7: Stop the program.

PROGRAM:

LEX PROGRAM(ex1.l):

```
%{
#include<stdio.h>
%}
%%
bool|int|float|char|if|then|else|while|for|printf|scanf|do|continue|break printf("Keyword \n");
[-+*/]+ printf("Operators \n");
[0-9]+ printf("Number \n");
[,.'"]+ printf("Punctuation Chars \n");
[&%*$@!]+ printf("Special Characters \n");
[A-Za-z0-9]+ printf("Identifiers \n");
%%
int main()
{
yylex();
}
```

OUTPUT:



```
cse@cse-System-Product-Name:~/Desktop/CD Lab/Exp1$ lex ex1.l
cse@cse-System-Product-Name:~/Desktop/CD Lab/Exp1$ gcc lex.yy.c -ll
cse@cse-System-Product-Name:~/Desktop/CD Lab/Exp1$ ./a.out
printf
Keyword

253
Number

+
Operators

"
Punctuation Chars

@
Special Characters

abcd1
Identifiers
```

RESULT:

Thus the above program for developing a lexical analyzer to recognize few patterns in C was executed and the output has been obtained successfully.

Ex.No: 2

IMPLEMENT A LEXICAL ANALYZER USING LEX TOOL

PROBLEM STATEMENT:

To write a program for implementing a Lexical analyser using LEX tool in Linux platform.

ALGORITHM:

Step 1: Lex program contains three sections: definitions, rules, and user subroutines. Each section must be separated from the others by a line containing only the delimiter, %%. The format is as follows: definitions %% rules % % user_subroutines

Step 2: In definition section, the variables make up the left column, and their definitions make up the right column. Any C statements should be enclosed in % (..)%. Identifier is defined such that the first letter of an identifier is alphabet and remaining letters are alphanumeric

Step 3: In rules section, the left column contains the pattern to be recognized in an input file to yylex(). The right column contains the C program fragment executed when that pattern is recognized. The various patterns are keywords, operators, new line character, number, string, identifier, beginning and end of block, comment statements, preprocessor directive statements etc.

Step 4: Each pattern may have a corresponding action, that is, a fragment of C source code to execute when the pattern is matched.

Step 5: When yylex() matches a string in the input stream, it copies the matched text to an external character array, yytext, before it executes any actions in the rules section.

Step 6: In user subroutine section, main routine calls yylex(). yy wrap() is used to get more input.

Step 7: The lex command uses the rules and actions contained in file to generate a program, lex.yy.c, which can be compiled with the cc command. That program can then receive input, break the input into the logical pieces defined by the rules in file, and run program fragments contained in the actions in file.

PROGRAM:

LEX PROGRAM(ex2.l):

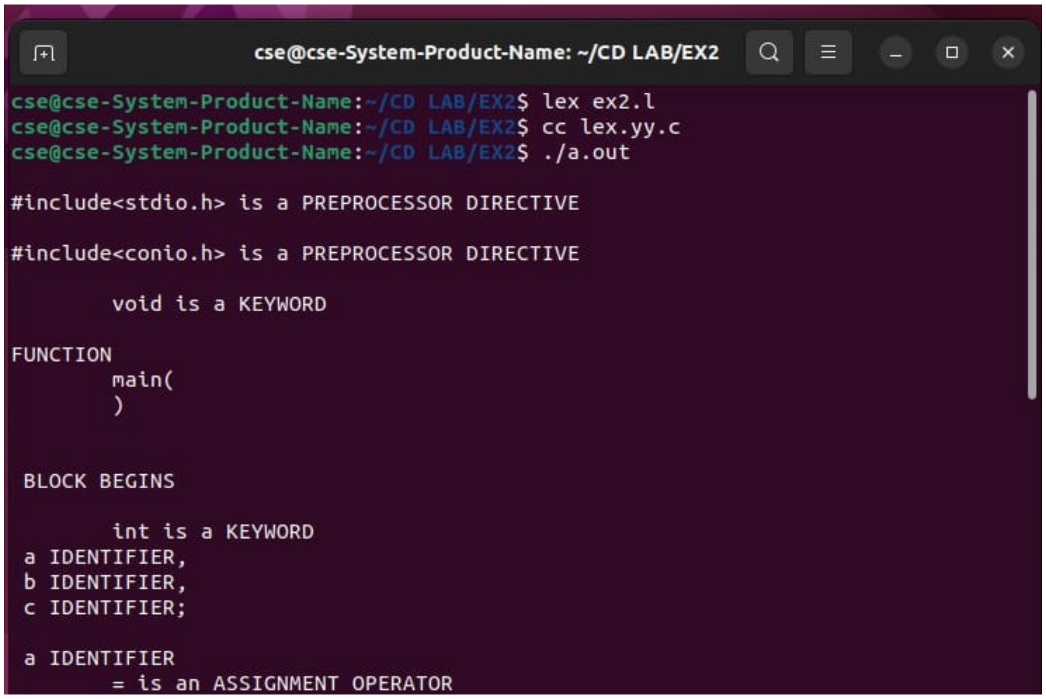
```
%{
/* program to recognize a c program */
int COMMENT=0;
int cnt=0;
%}
identifier [a-zA-Z][a-zA-Z0-9]*
%%
#.* { printf("\n%s is a PREPROCESSOR DIRECTIVE",yytext);}
int |
float |
char |
double |
while |
for |
do |
if |
break |
continue |
void |
switch |
case |
long |
struct |
const |
typedef |
return |
else |
goto {printf("\n\t%s is a KEYWORD",yytext);}
"/*" {COMMENT = 1;}
"*/" {COMMENT = 0; cnt++;}
{identifier}\( {if(!COMMENT)printf("\n\nFUNCTION\n\t%s",yytext);}
\{ {if(!COMMENT) printf("\n BLOCK BEGINS");}
\} {if(!COMMENT) printf("\n 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\t%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);
}
```

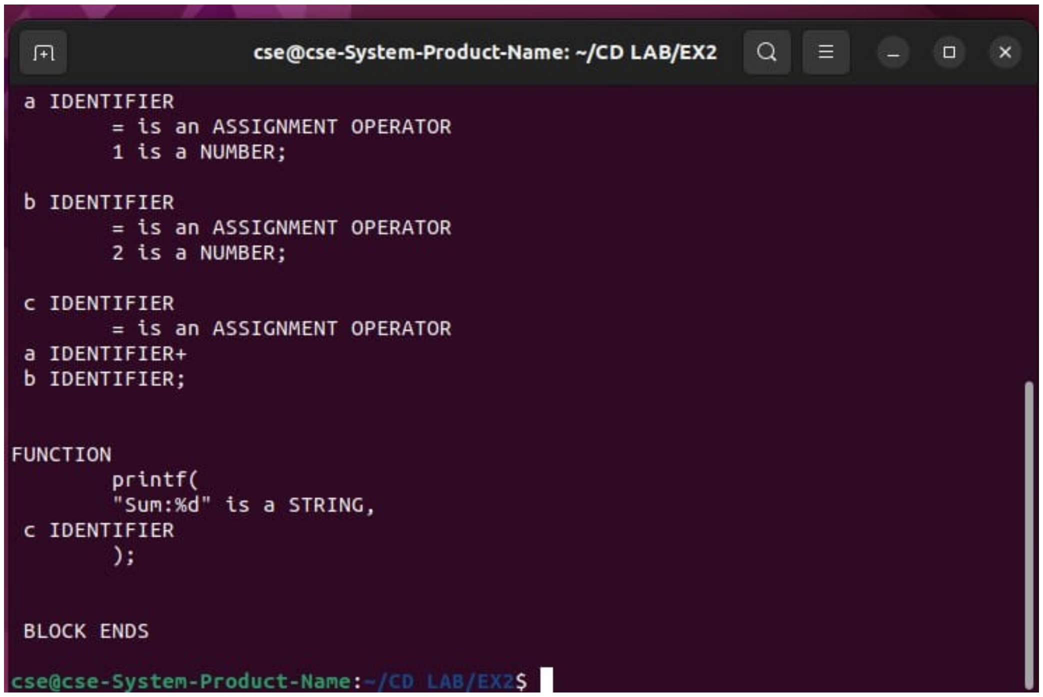
C PROGRAM(var.c):

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

OUTPUT :



```
cse@cse-System-Product-Name:~/CD LAB/EX2$ lex ex2.l
cse@cse-System-Product-Name:~/CD LAB/EX2$ cc lex.yy.c
cse@cse-System-Product-Name:~/CD LAB/EX2$ ./a.out

#include<stdio.h> is a PREPROCESSOR DIRECTIVE

#include<conio.h> is a PREPROCESSOR DIRECTIVE

        void is a KEYWORD

FUNCTION
        main(
        )


 BLOCK BEGINS

        int is a KEYWORD
 a IDENTIFIER,
 b IDENTIFIER,
 c IDENTIFIER;

 a IDENTIFIER
        = is an ASSIGNMENT OPERATOR
```



```
 a IDENTIFIER
        = is an ASSIGNMENT OPERATOR
        1 is a NUMBER;

 b IDENTIFIER
        = is an ASSIGNMENT OPERATOR
        2 is a NUMBER;

 c IDENTIFIER
        = is an ASSIGNMENT OPERATOR
 a IDENTIFIER+
 b IDENTIFIER;


FUNCTION
        printf(
        "Sum:%d" is a STRING,
 c IDENTIFIER
        );


 BLOCK ENDS

cse@cse-System-Product-Name:~/CD LAB/EX2$ 
```

RESULT:

Thus the above program for implementing a lexical analyzer using lex tool was executed and the output has been obtained success.

Ex.No: 3

GENERATE YACC SPECIFICATION FOR A FEW SYNTACTIC CATEGORIES.

3a. Program to recognize a valid arithmetic expression that uses operator +, -, * and /.

PROBLEM STATEMENT:

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

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. $$ refer to the top of the stack position while $1 for the first value, $2 for the second value in the stack.
5. Call yyparse() to initiate the parsing process.
6. yyerror() function is called when all productions in the grammar in second section doesn't match to the input statement.

PROGRAM:

LEX PROGRAM (arith.l):

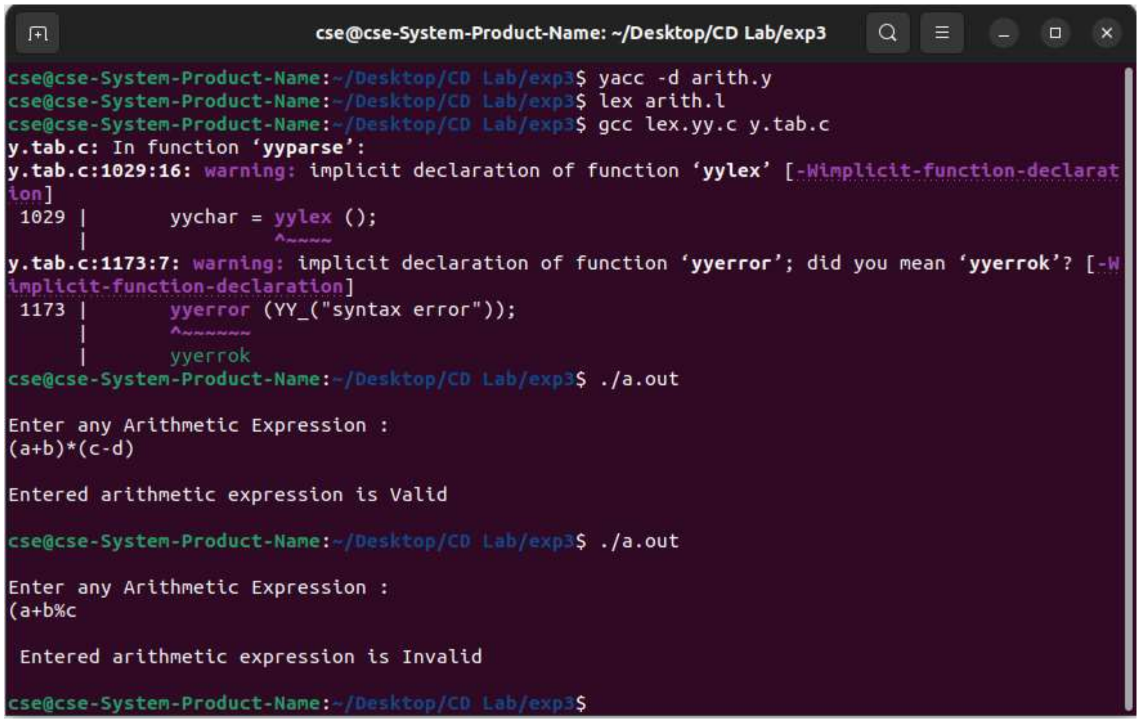
```
%{

#include<stdio.h>
#include "y.tab.h"
%}
%%
[a-zA-Z]+ return VARIABLE;
[0-9]+ return NUMBER;
[\t] ;
[\n] return 0;
. return yytext[0]; /*matched string is kept into the address pointed by pointer yytext*/
%%
int yywrap() /*Lex call yywrap() function after input is over*/
{
return 1;
}
```

YACC PROGRAM(arith.y):

```
/*Declaration Section*/
%{
#include<stdio.h>
%}
%token NUMBER
%token VARIABLE
%left '+' '-'
%left '*' '/' '%'
%left '(' ')'
/*Translation Section*/
%%
S: 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
;
%%
```

```
void main()
{
printf("\nEnter any Arithmetic Expression : \n");
yyparse();
}
int yyerror(void)
{
printf("\n Entered arithmetic expression is Invalid\n\n");
}
```

OUTPUT:



```
cse@cse-System-Product-Name:~/Desktop/CD Lab/exp3
cse@cse-System-Product-Name:~/Desktop/CD Lab/exp3$ yacc -d arith.y
cse@cse-System-Product-Name:~/Desktop/CD Lab/exp3$ lex arith.l
cse@cse-System-Product-Name:~/Desktop/CD Lab/exp3$ gcc lex.yy.c y.tab.c
y.tab.c: In function ‘yyparse’:
y.tab.c:1029:16: warning: implicit declaration of function ‘yylex’ [-Wimplicit-function-declarat
ion]
 1029 |       yychar = yylex ();
      |                ^~~~~
y.tab.c:1173:7: warning: implicit declaration of function ‘yyerror’; did you mean ‘yyerrok’? [-W
implicit-function-declaration]
 1173 |       yyerror (YY_("syntax error"));
      |       ^~~~~~~
      |       yyerrok
cse@cse-System-Product-Name:~/Desktop/CD Lab/exp3$ ./a.out

Enter any Arithmetic Expression :
(a+b)*(c-d)

Entered arithmetic expression is Valid

cse@cse-System-Product-Name:~/Desktop/CD Lab/exp3$ ./a.out

Enter any Arithmetic Expression :
(a+b%c

 Entered arithmetic expression is Invalid

cse@cse-System-Product-Name:~/Desktop/CD Lab/exp3$
```

RESULT:

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

3b. Program to recognize a valid variable which starts with a letter followed by any number of letters or digits.

PROBLEM STATEMENT:

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

ALGORITHM:

LEX

1. Declare the required header file and variable declaration with in '%{' and '%}'.
2. LEX requires regular expressions or patterns to identify token of lexemes for recognize a valid variable.
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. $$ refer to the top of the stack position while $1 for the first value, $2 for the second value in the stack.
5. Call yyparse() to initiate the parsing process.
6. yyerror() function is called when all productions in the grammar in second section doesn't match to the input statement.

PROGRAM:

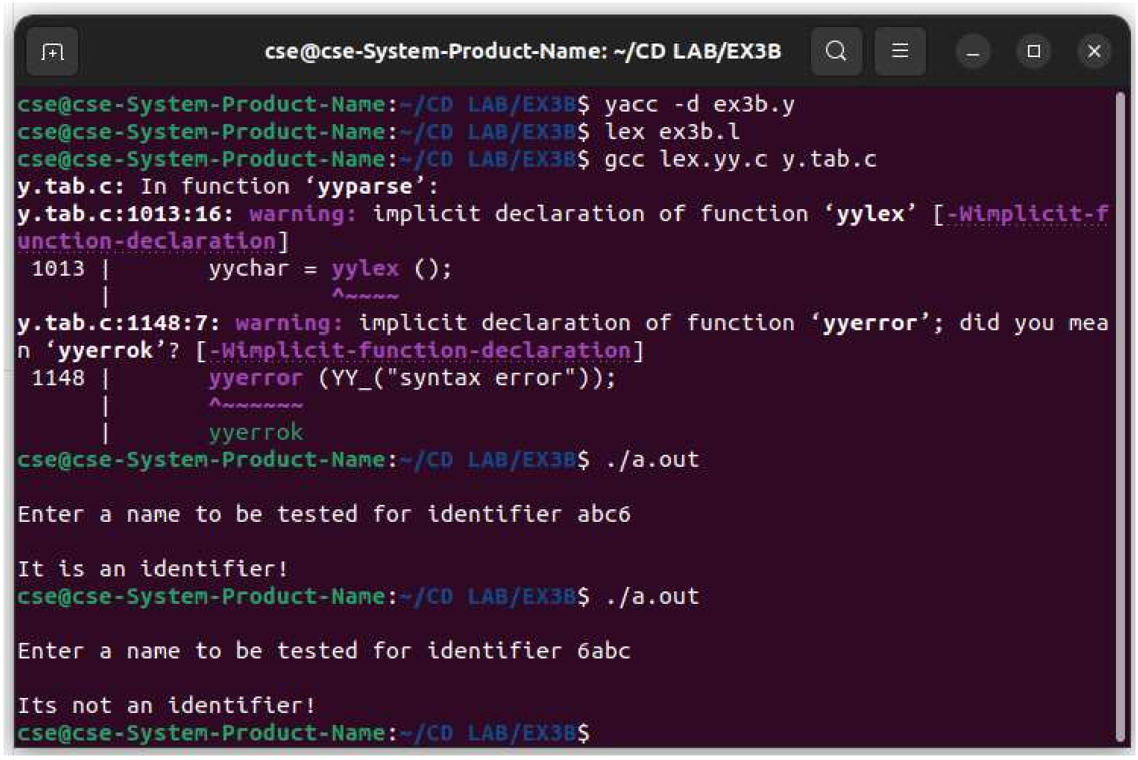
LEX PROGRAM (ex3b.l):

```
%{

   #include "y.tab.h"
%}
%%
[a-zA-Z][a-zA-Z_0-9]* return letter;
[0-9] return digit;
. return yytext[0];
\n return 0;
%%
int yywrap()
{
return 1;
}
```

YACC PROGRAM (ex3b.y):

```
%{
   #include<stdio.h>
   int valid=1;
%}
%token digit letter
%%
start : letter s
s :     letter s
      | digit s
     |      ;
%%
int yyerror()
{
   printf("\nIts not an identifier!\n");
   valid=0;
   return 0;}
int main()
{
   printf("\nEnter a name to be tested for identifier");
   yyparse();
   if(valid)
   {
      printf("\nIt is an identifier!\n");
   }
}
```

OUTPUT:



cse@cse-System-Product-Name: ~/CD LAB/EX3B

```
cse@cse-System-Product-Name:~/CD LAB/EX3B$ yacc -d ex3b.y
cse@cse-System-Product-Name:~/CD LAB/EX3B$ lex ex3b.l
cse@cse-System-Product-Name:~/CD LAB/EX3B$ gcc lex.yy.c y.tab.c
y.tab.c: In function ‘yyparse’:
y.tab.c:1013:16: warning: implicit declaration of function ‘yylex’ [-Wimplicit-f
unction-declaration]
 1013 |       yychar = yylex ();
      |                ^~~~~
y.tab.c:1148:7: warning: implicit declaration of function ‘yyerror’; did you mea
n ‘yyerrok’? [-Wimplicit-function-declaration]
 1148 |       yyerror (YY_("syntax error"));
      |       ^~~~~~~
      |       yyerrok
cse@cse-System-Product-Name:~/CD LAB/EX3B$ ./a.out

Enter a name to be tested for identifier abc6

It is an identifier!
cse@cse-System-Product-Name:~/CD LAB/EX3B$ ./a.out

Enter a name to be tested for identifier 6abc

Its not an identifier!
cse@cse-System-Product-Name:~/CD LAB/EX3B$
```

RESULT:

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

3c. Program to recognize while Loop

PROBLEM STATEMENT:

To write a lex program to recognize while loop.

ALGORITHM:

Step 1: Start the program.

Step 2: Reading an expression.

Step 3: Checking the validating of the given while loop according to the rule using yacc.

Step 4: Print the result of the given while loop.

Step 5: Stop the program.

PROGRAM:

LEX PROGRAM:

```
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];
%%
```

YACC PROGRAM:

```
%{
#include <stdio.h>
#include <stdlib.h>
#include<conio.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
        ;
%%
```

```
#include "lex.yy.c"
void main()
{
   printf("Enter the exp: ");
   yyparse();
   getch();
}
```

OUTPUT:

```
while(i<n)Input Accepted
```

RESULT:

Thus a program to recognize while loop was executed successfully.

3d. Implementation of calculator using LEX and YACC.

PROBLEM STATEMENT:

To write a program to implement Calculator using LEX and YACC.

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 cal.l.

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

Step 5: Display error is persist.

Step 6: Provide the input.

Step 7: Verify the output.

Step 8: End.

PROGRAM:

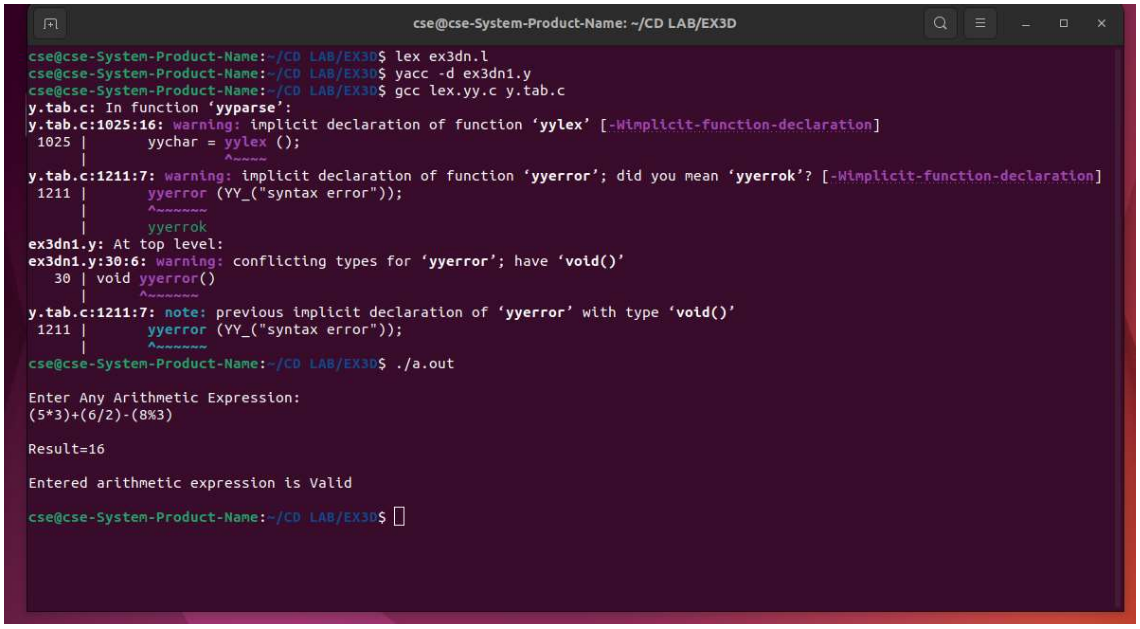
LEX PROGRAM (ex3dn.l):

```
%{
   /* Definition section */
 #include<stdio.h>
 #include "y.tab.h"
 extern int yylval;
%}

/* Rule Section */
%%
[0-9]+ {
          yylval=atoi(yytext);
          return NUMBER;
       }
[\t] ;

[\n] return 0;

. return yytext[0];

%%

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

YACC PROGRAM (ex3dn1.y):

```
%{
  #include<stdio.h>
  int flag=0;
%}
%token NUMBER
%left '+' '-'
%left '*' '/' '%'
%left '(' ')'
%%
ArithmeticExpression: E{
  printf("\nResult=%d\n", $$);
  return 0;
  };
 E:E'+'E {$$=$1+$3;}
 |E'-'E {$$=$1-$3;}
 |E'*'E {$$=$1*$3;}
 |E'/'E {$$=$1/$3;}
 |E'%'E {$$=$1%$3;}
 |'('E')' {$$=$2;}
 | NUMBER {$$=$1;}
 ;
%%
void main()
{
  printf("\nEnter Any Arithmetic Expression:\n");
  yyparse();
  if(flag==0)
  printf("\nEntered arithmetic expression is Valid\n\n");
}
void yyerror()
{
  printf("\nEntered arithmetic expression is Invalid\n\n");
  flag=1;
}
```

OUTPUT:



```
cse@cse-System-Product-Name:~/CD LAB/EX3D$ lex ex3dn.l
cse@cse-System-Product-Name:~/CD LAB/EX3D$ yacc -d ex3dn1.y
cse@cse-System-Product-Name:~/CD LAB/EX3D$ gcc lex.yy.c y.tab.c
y.tab.c: In function 'yyparse':
y.tab.c:1025:16: warning: implicit declaration of function 'yylex' [-Wimplicit-function-declaration]
 1025 |       yychar = yylex ();
      |                ^~~~~
y.tab.c:1211:7: warning: implicit declaration of function 'yyerror'; did you mean 'yyerrok'? [-Wimplicit-function-declaration]
 1211 |       yyerror (YY_("syntax error"));
      |       ^~~~~~~
      |       yyerrok
ex3dn1.y: At top level:
ex3dn1.y:30:6: warning: conflicting types for 'yyerror'; have 'void()'
   30 | void yyerror()
      |      ^~~~~~~
y.tab.c:1211:7: note: previous implicit declaration of 'yyerror' with type 'void()'
 1211 |       yyerror (YY_("syntax error"));
      |       ^~~~~~~
cse@cse-System-Product-Name:~/CD LAB/EX3D$ ./a.out

Enter Any Arithmetic Expression:
(5*3)+(6/2)-(8%3)

Result=16

Entered arithmetic expression is Valid

cse@cse-System-Product-Name:~/CD LAB/EX3D$
```

RESULT:

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

Ex.No: 4

GENERATE THREE ADDRESS CODE FOR A SIMPLE PROGRAM USING LEX AND YACC.

PROBLEM STATEMENT:

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.

PROGRAM:

LEX PROGRAM (exn4.l):

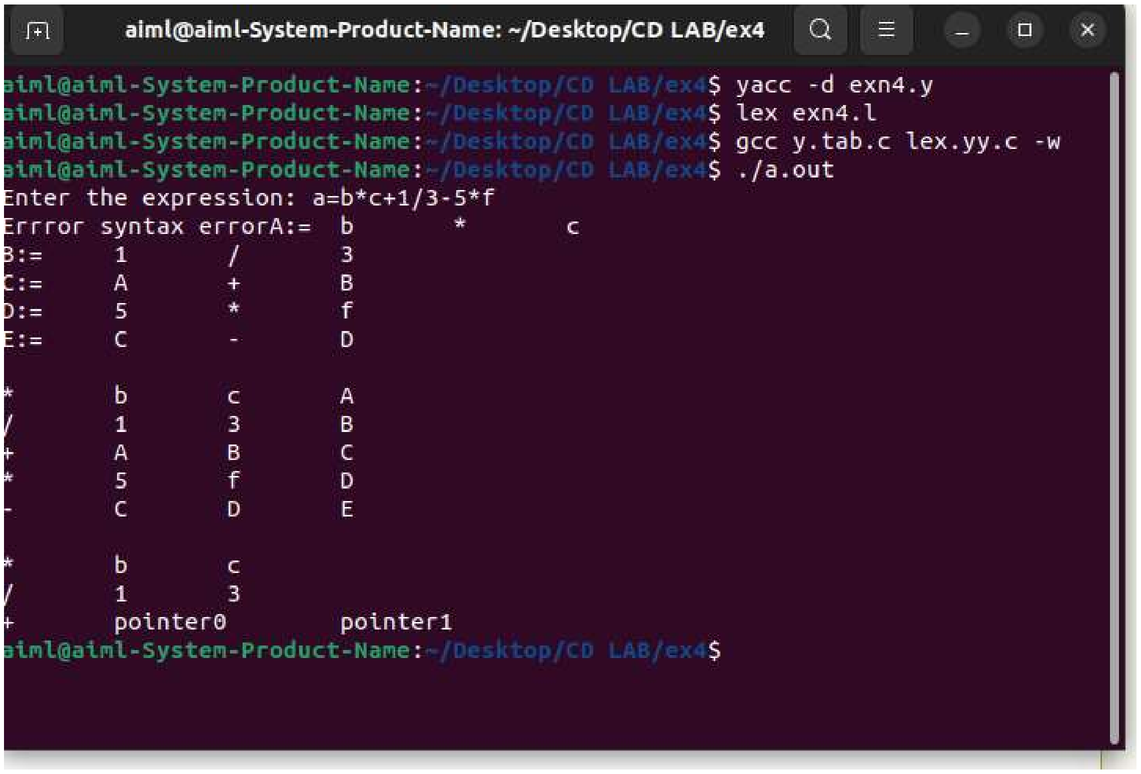
```
%{
#include"y.tab.h"
extern char yyval;
%}
%%
[0-9]+ {yylval.symbol=(char)(yytext[0]);return NUMBER;}
[a-z] {yylval.symbol= (char)(yytext[0]);return LETTER;}
. {return yytext[0];}
\n {return 0;}
%%
```

YACC PROGRAM (exn4.y):

```
%{
#include"y.tab.h"
#include<stdio.h>
char addtotable(char,char,char);
int index1=0;
char temp = 'A'-1;
struct expr{
char operand1;
char operand2;
char operator;
char result;
};
%}
%union{
char symbol;
}
%left '+' '-'
%left '/' '*'
%token <symbol> LETTER NUMBER
%type <symbol> exp
%%
statement: LETTER '=' exp ';' {addtotable((char)$1,(char)$3,'=');};
exp: exp '+' exp {$$ = addtotable((char)$1,(char)$3,'+');}
   |exp '-' exp {$$ = addtotable((char)$1,(char)$3,'-');}
   |exp '/' exp {$$ = addtotable((char)$1,(char)$3,'/');}
   |exp '*' exp {$$ = addtotable((char)$1,(char)$3,'*');}
   |'(' exp ')' {$$= (char)$2;}
   |NUMBER {$$ = (char)$1;}
   |LETTER {(char)$1;};
%%
struct expr arr[20];
void yyerror(char *s){
   printf("Errror %s",s);
```

```
}
char addtotable(char a, char b, char o){
    temp++;
    arr[index1].operand1 =a;
    arr[index1].operand2 = b;
    arr[index1].operator = o;
    arr[index1].result=temp;
    index1++;
    return temp;
}
void threeAdd(){
    int i=0;
    char temp='A';
    while(i<index1){
        printf("%c:=\t",arr[i].result);
        printf("%c\t",arr[i].operand1);
        printf("%c\t",arr[i].operator);
        printf("%c\t",arr[i].operand2);
        i++;
        temp++;
        printf("\n");
    }
}
void fouradd(){
    int i=0;
    char temp='A';
    while(i<index1){
        printf("%c\t",arr[i].operator);
        printf("%c\t",arr[i].operand1);
        printf("%c\t",arr[i].operand2);
        printf("%c",arr[i].result);
        i++;
        temp++;
        printf("\n");
    }
}
int find(char l){
    int i;
    for(i=0;i<index1;i++)
        if(arr[i].result==l) break;
    return i;
}
void triple(){
    int i=0;
    char temp='A';
    while(i<index1){
        printf("%c\t",arr[i].operator);
        if(!isupper(arr[i].operand1))
        printf("%c\t",arr[i].operand1);
        else{
            printf("pointer");
            printf("%d\t",find(arr[i].operand1));
        }
        if(!isupper(arr[i].operand2))
        printf("%c\t",arr[i].operand2);
        else{
            printf("pointer");
            printf("%d\t",find(arr[i].operand2));
        }
        i++;
        temp++;
        printf("\n");
    }
}
int yywrap(){
    return 1;
}
int main(){
    printf("Enter the expression: ");
    yyparse();
    threeAdd();
    printf("\n");
    fouradd();
    printf("\n");
    triple();
    return 0;
}
```

OUTPUT:



```
aiml@aiml-System-Product-Name:~/Desktop/CD LAB/ex4$ yacc -d exn4.y
aiml@aiml-System-Product-Name:~/Desktop/CD LAB/ex4$ lex exn4.l
aiml@aiml-System-Product-Name:~/Desktop/CD LAB/ex4$ gcc y.tab.c lex.yy.c -w
aiml@aiml-System-Product-Name:~/Desktop/CD LAB/ex4$ ./a.out
Enter the expression: a=b*c+1/3-5*f
Errror syntax errorA:=  b       *       c
B:=     1       /       3
C:=     A       +       B
D:=     5       *       f
E:=     C       -       D

*       b       c       A
/       1       3       B
+       A       B       C
*       5       f       D
-       C       D       E

*       b       c
/       1       3
+       pointer0        pointer1
aiml@aiml-System-Product-Name:~/Desktop/CD LAB/ex4$
```

RESULT:

Thus the program for generating three address code was executed and verified.

Ex.No: 5

IMPLEMENT TYPE CHECKING USING LEX AND YACC

PROBLEM STATEMENT:

To write a C program to implement type checking.

ALGORITHM:

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

Step 2: Read the expression and declaration.

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

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

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

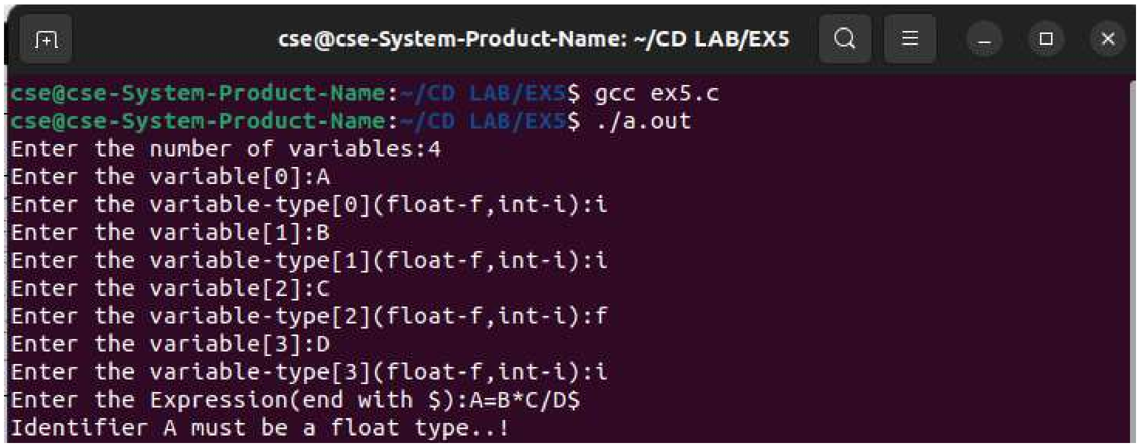
Step 6: If the both the operands’ type are matched then check for result variable. Else, print “Type mismatch”.

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

PROGRAM:

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



cse@cse-System-Product-Name: ~/CD LAB/EX5

```
cse@cse-System-Product-Name:~/CD LAB/EX5$ gcc ex5.c
cse@cse-System-Product-Name:~/CD LAB/EX5$ ./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..!
```

RESULT:

Thus the program for type checking was executed and verified.

Ex.No: 6

IMPLEMENT SIMPLE CODE OPTIMIZATION TECHNIQUES

PROBLEM STATEMENT:

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

ALGORITHM:

1. Read the un-optimized input block.
2. Identify the types of optimization.
3. Optimize the input block.
4. Print the optimized input block.
5. Execute the same with different set of un-optimized inputs and obtain the optimized input block.

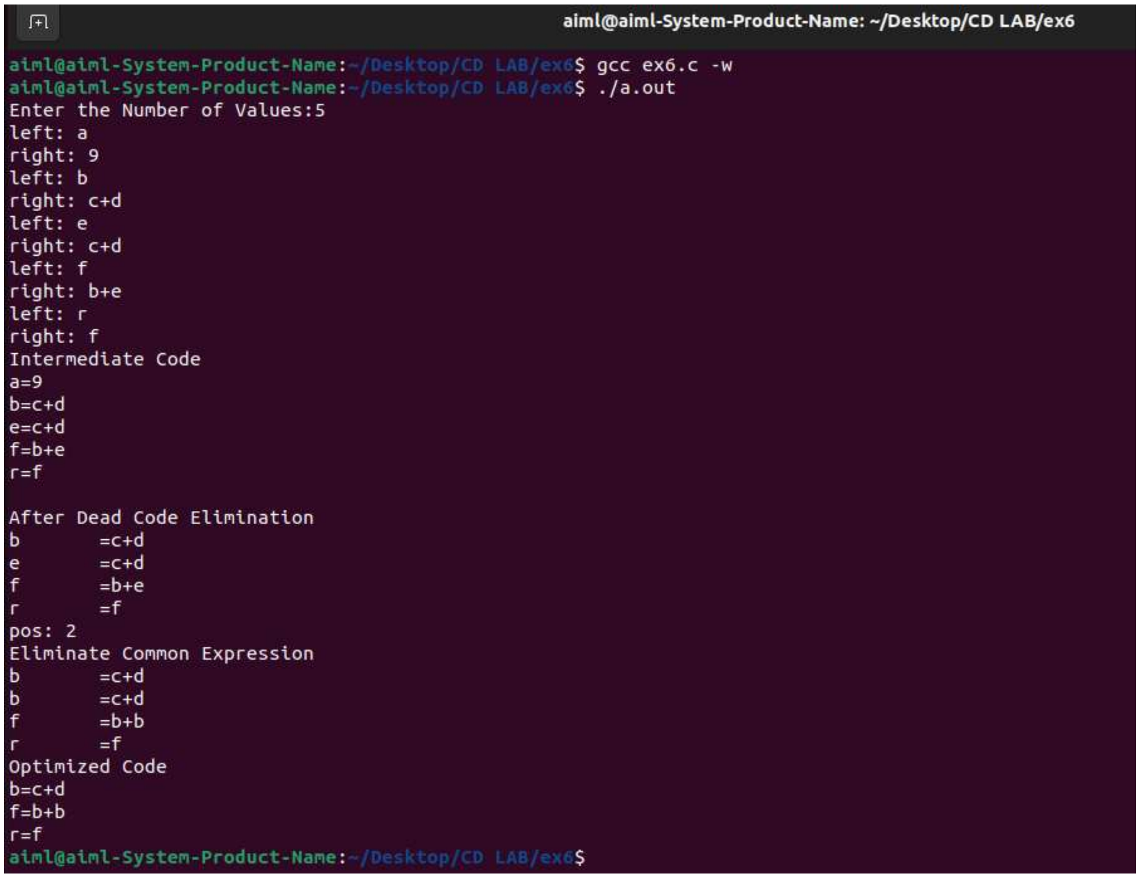
PROGRAM:

```
#include<stdio.h>
#include<string.h>
struct op
{
char l;
char r[20];
}
op[10],pr[10];
void main()
{
int a,i,k,j,n,z=0,m,q;
char *p,*l;
char temp,t;
char *tem;
printf("Enter the Number of Values:");
scanf("%d",&n);
for(i=0;i<n;i++)
{
printf("left: ");
scanf(" %c",&op[i].l);
printf("right: ");
scanf(" %s",&op[i].r);
}
printf("Intermediate Code\n") ;
for(i=0;i<n;i++)
{
printf("%c=",op[i].l);
printf("%s\n",op[i].r);
}
for(i=0;i<n-1;i++)
{
temp=op[i].l;
for(j=0;j<n;j++)
{
p=strchr(op[j].r,temp);
if(p)
{
pr[z].l=op[i].l;
strcpy(pr[z].r,op[i].
r);
z++;
}
}
}
pr[z].l=op[n-1].l;
strcpy(pr[z].r,op[n-1].r);
z++;
printf("\nAfter Dead Code Elimination\n");
```

```
for(k=0;k<z;k++)
{
printf("%c\t=",pr[k].l);
printf("%s\n",pr[k].r);
}
for(m=0;m<z;m++)
{
tem=pr[m].r;
for(j=m+1;j<z;j++)
{
p=strstr(tem,pr[j].r);
if(p)
{
t=pr[j].l;
pr[j].l=pr[m].l;
for(i=0;i<z;i++)
{
l=strchr(pr[i].r,t) ;
if(l)
{
a=l-pr[i].r;
printf("pos: %d\n",a);
pr[i].r[a]=pr[m].l;
}
}
}
}
}
printf("Eliminate Common Expression\n");
for(i=0;i<z;i++)
{
printf("%c\t=",pr[i].l);
printf("%s\n",pr[i].r);
}
for(i=0;i<z;i++)
{
for(j=i+1;j<z;j++)
{
q=strcmp(pr[i].r,pr[j].r);
if((pr[i].l==pr[j].l)&&!q)
{
pr[i].l='\0';
}
}
}
printf("Optimized Code\n");
for(i=0;i<z;i++)
{
if(pr[i].l!='\0')
{
```

```
printf("%c=",pr[i].l);
printf("%s\n",pr[i].r);
}
}
}
```

OUTPUT:



```
aiml@aiml-System-Product-Name: ~/Desktop/CD LAB/ex6
aiml@aiml-System-Product-Name:~/Desktop/CD LAB/ex6$ gcc ex6.c -w
aiml@aiml-System-Product-Name:~/Desktop/CD LAB/ex6$ ./a.out
Enter the Number of Values:5
left: a
right: 9
left: b
right: c+d
left: e
right: c+d
left: f
right: b+e
left: r
right: f
Intermediate Code
a=9
b=c+d
e=c+d
f=b+e
r=f

After Dead Code Elimination
b       =c+d
e       =c+d
f       =b+e
r       =f
pos: 2
Eliminate Common Expression
b       =c+d
b       =c+d
f       =b+b
r       =f
Optimized Code
b=c+d
f=b+b
r=f
aiml@aiml-System-Product-Name:~/Desktop/CD LAB/ex6$
```

RESULT:

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

Ex.No: 7

IMPLEMENTING THE BACK END OF THE COMPILER

PROBLEM STATEMENT:

To 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.

ALGORITHM:

Step 1: Start the program.

Step 2: Include the necessary header files.

Step 3: Declare necessary character arrays for input and output and also a structure to include it.

Step 4: Get the Intermediate Code as the input.

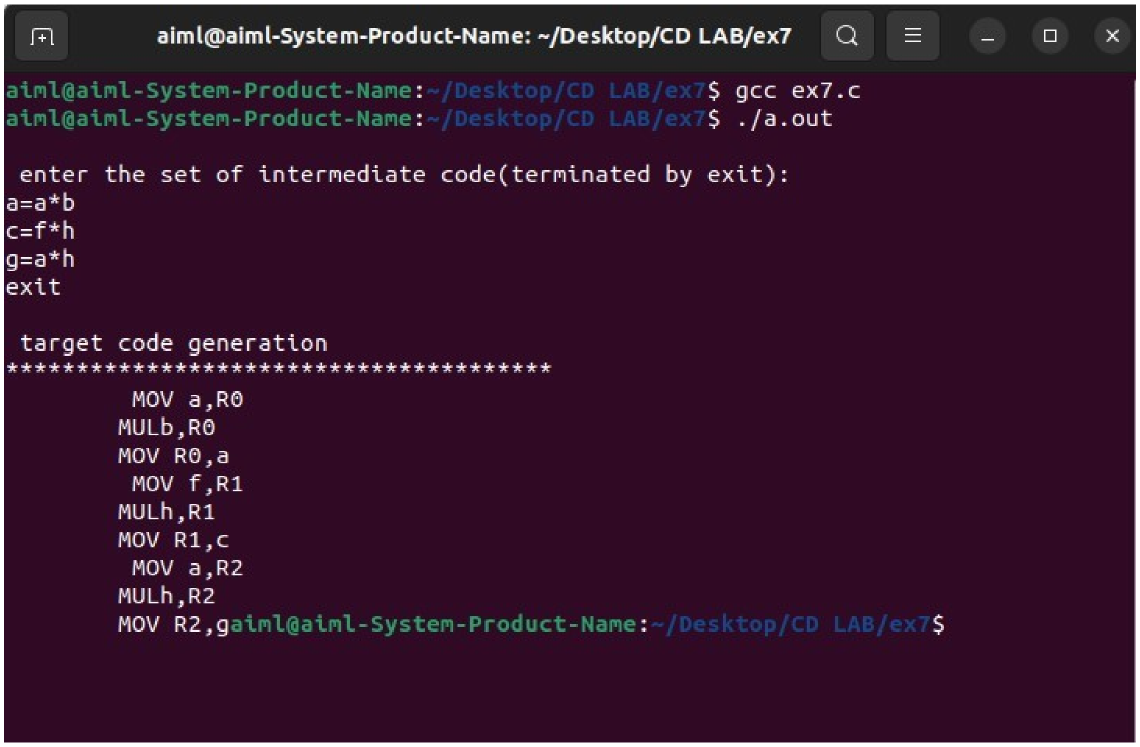
Step 5: Display the options to do the various operations and use a switch case to implement that operation.

Step 6: Terminate the program

PROGRAM:

```
#include<stdio.h>
#include<string.h>
void main()
{
char icode[10][30],str[20],opr[10];
int i=0;
printf("\n enter the set of intermediate code(terminated by exit):\n");
do
{
scanf("%s",icode[i]);
}while(strcmp(icode[i++],"exit")!=0);
printf("\n target code generation");
printf("\n*******************************************");
i=0;
do{
strcpy(str,icode[i]);
switch(str[3]){
case '+':
strcpy(opr,"ADD");
break;
case '-':
strcpy(opr,"SUB");
  break;
case '*':
strcpy(opr,"MUL");
break;
case '/':
strcpy(opr,"DIV");
break;
}
printf("\n\t MOV %c,R%d",str[2],i);
printf("\n\t%s%c,R%d",opr,str[4],i);
printf("\n\tMOV R%d,%c",i,str[0]);
}while(strcmp(icode[++i],"exit")!=0);
}
```

OUTPUT:



```
aiml@aiml-System-Product-Name:~/Desktop/CD LAB/ex7$ gcc ex7.c
aiml@aiml-System-Product-Name:~/Desktop/CD LAB/ex7$ ./a.out

 enter the set of intermediate code(terminated by exit):
a=a*b
c=f*h
g=a*h
exit

 target code generation
*******************************************
         MOV a,R0
        MULb,R0
        MOV R0,a
         MOV f,R1
        MULh,R1
        MOV R1,c
         MOV a,R2
        MULh,R2
        MOV R2,gaiml@aiml-System-Product-Name:~/Desktop/CD LAB/ex7$
```

RESULT:

Thus a program to implement the back end of the compiler was executed successfully.

Ex.No: 8

ERROR HANDLING IN LEX AND YACC

PROBLEM STATEMENT:

To implement error detection and recovery strategies in lexer and parser.

ALGORITHM:

Step 1: Define Tokens:

- Use regular expressions to define tokens for numbers, identifiers, operators, and whitespace.
- Define an error token for unrecognized characters.

Step 2: Lexical Analyzer (lexer.l):

- Create rules to match:
  - Numbers ([0-9]+)
  - Whitespace (ignore)
  - Operators ([+*/()-])
  - Identifiers ([a-zA-Z][a-zA-Z0-9]*)
  - Catch-all for errors (print an error message).
- Return the matched tokens to the parser.

Step 3: Parser (parser.y):

- Define a grammar for arithmetic expressions:
  - Create rules for addition, subtraction, multiplication, and division.
  - Handle parentheses for grouping.
- Include error handling for division by zero:
  - If the divisor is zero, print an error message.
- Capture and print the result of the evaluated expression.
- Handle identified variables.

Step 4: Error Reporting:

- Implement the yyerror function to print meaningful error messages.
- Ensure that the program continues to parse after encountering errors.

Step 5: Compile and Link:

- Use flex to generate the lexer code.
- Use bison to generate the parser code.
- Compile everything using gcc.

PROGRAM:

LEX PROGRAM (lexer.l):

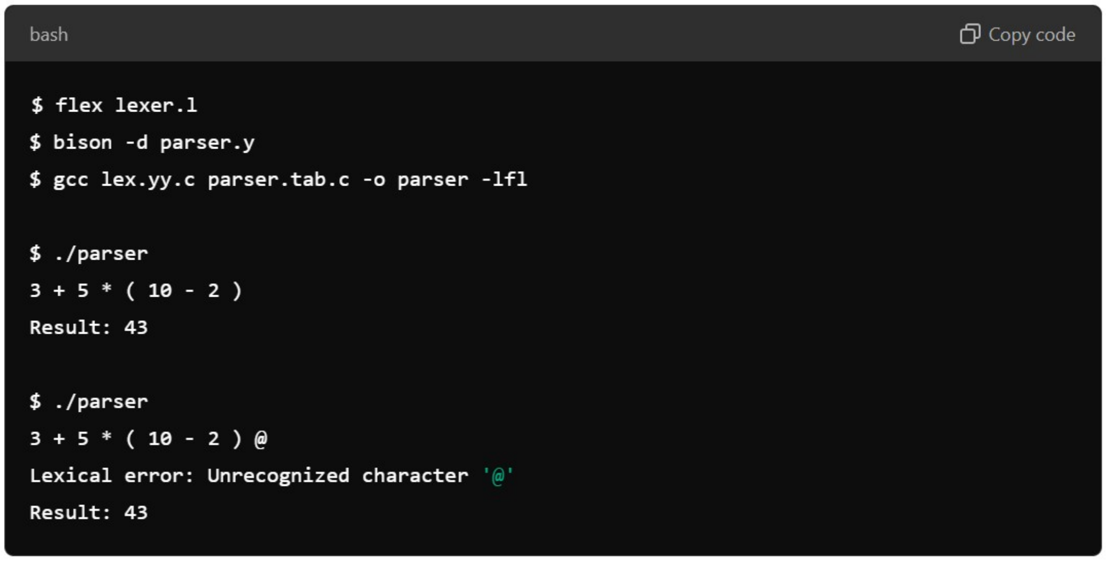
```
%{
#include "y.tab.h"
%}

%%
[0-9]+         { yylval = atoi(yytext); return NUMBER; }
[ \t\n]+      ; // Ignore whitespace
[+*/()-]      { return yytext[0]; }
[a-zA-Z][a-zA-Z0-9]* { yylval = strdup(yytext); return IDENTIFIER; }
.             { printf("Lexical error: Unrecognized character '%s'\n", yytext); }
%%
int yywrap() { return 1; }
```

Yacc Program (parser.y)

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

%union {
   int num;
   char* id;
}
```

```
%token <num> NUMBER
%token <id> IDENTIFIER
%type <num> expr

%%
program:
    expr { printf("Result: %d\n", $1); }
    ;

expr:
    expr '+' expr { $$ = $1 + $3; }
    | expr '-' expr { $$ = $1 - $3; }
    | expr '*' expr { $$ = $1 * $3; }
    | expr '/' expr {
        if ($3 == 0) {
            yyerror("Error: Division by zero");
            $$ = 0;
        } else {
            $$ = $1 / $3;
        }
    }
    | '(' expr ')' { $$ = $2; }
    | NUMBER { $$ = $1; }
    | IDENTIFIER { printf("Identified variable: %s\n", $1); $$ = 0; } // Placeholder
    ;

%%

void yyerror(const char *s) {
    fprintf(stderr, "Error: %s\n", s);
}

int main(void) {
    yyparse();
    return 0;
}
```

OUTPUT:



```bash
$ flex lexer.l
$ bison -d parser.y
$ gcc lex.yy.c parser.tab.c -o parser -lfl

$ ./parser
3 + 5 * ( 10 - 2 )
Result: 43

$ ./parser
3 + 5 * ( 10 - 2 ) @
Lexical error: Unrecognized character '@'
Result: 43
```

RESULT:

Thus the program to create a simple interpreter for the language recognized by lexer and parser has been executed successfully.

Ex.No: 9

IMPLEMENTATION OF A SIMPLE INTERPRETER

PROBLEM STATEMENT:

To create a simple interpreter for the language recognized by lexer and parser.

ALGORITHM:

Step 1: Define Data Structures:

- Create a Variable structure to store variable names and their values.
- Use an array to hold these variables.

Step 2: Function to Get Variable Value:

- Implement a function get_value that takes a variable name as input and returns its value.
- If the variable does not exist, return a default value (e.g., 0).

Step 3: Function to Set Variable Value:

- Implement a function set_value that takes a variable name and a value as input.
- Check if the variable already exists:
    - If it exists, update its value.
    - If it does not exist, add it to the variable array.

Step 4: Instruction Execution:

- Implement a function execute_instruction that takes a string instruction as input.
- Parse the instruction to extract the operation (ADD or SUB), the target variable, and two operands.
- Depending on the operation:
    - For ADD, calculate the sum and store it in the target variable.
    - For SUB, calculate the difference and store it in the target variable.
- Print the result after each operation.

Step 5: Main Function:

- In the main function, call execute_instruction with predefined instructions (e.g., "ADD x 5 3" and "SUB y x 2").

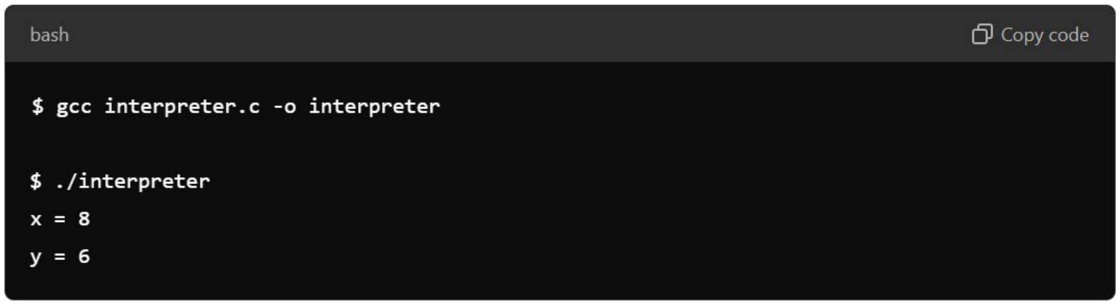
Step 6: Compile and Run:

- Compile the interpreter program using gcc.
- Execute the program to see the results of the variable assignments.

PROGRAM (interpreter.c)

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

#define MAX_VARS 100

typedef struct {
    char name[20];
    int value;
} Variable;

Variable vars[MAX_VARS];
int var_count = 0;

int get_value(char *name) {
    for (int i = 0; i < var_count; i++) {
        if (strcmp(vars[i].name, name) == 0) {
            return vars[i].value;
        }
    }
    return 0; // Default value if not found
}

void set_value(char *name, int value) {
    for (int i = 0; i < var_count; i++) {
        if (strcmp(vars[i].name, name) == 0) {
            vars[i].value = value;
            return;
        }
    }
    strcpy(vars[var_count].name, name);
    vars[var_count].value = value;
    var_count++;
}

void execute_instruction(char *instruction) {
    char op[10], var[20];
    int num1, num2;

    sscanf(instruction, "%s %s %d %d", op, var, &num1, &num2);

    if (strcmp(op, "ADD") == 0) {
        set_value(var, get_value(num1) + get_value(num2));
    } else if (strcmp(op, "SUB") == 0) {
        set_value(var, get_value(num1) - get_value(num2));
    }
    printf("%s = %d\n", var, get_value(var));
}

int main() {
    execute_instruction("ADD x 5 3");
    execute_instruction("SUB y x 2");
    return 0;
}
```

OUTPUT:


bash

```
$ gcc interpreter.c -o interpreter

$ ./interpreter
x = 8
y = 6
```

RESULT:

Thus the program to create a simple interpreter for the language recognized by lexer and parser has been executed successfully.