



## **THAMIRABHARANI ENGINEERING COLLEGE**

(Approved by AICTE, New Delhi and Affiliated to Anna University, Chennai)  
Chathirampudukulam, Chidambaranagar - Vepemkulam Road.  
Thatchanallur, Tirunelveli 627 358, Tamil Nadu.



### **DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING**

**Course Code : CS3501**

**Course Name : COMPILER DESIGN LABORATORY**

**Year / Sem : III / V**

**Staff Name : Mrs. G. MARISUNDARI, AP/CSE**



**Prepared by**

**Approved by**



# THAMIRABHARANI ENGINEERING COLLEGE

(Approved by AICTE, New Delhi and Affiliated to Anna University, Chennai)  
Chathirampudukulam, Chidambaranagar - Vepemkulam Road.  
Thatchanallur, Tirunelveli 627 358, Tamil Nadu.



## DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

### COURSE PLAN

Revision: 1

Date: 09.08.2024

#### PART - I

<b>VISION OF TEC</b> <ul style="list-style-type: none"><li>To be a center of excellence in Engineering, exposing emerging technologies and instilling Entrepreneurial Attitude.</li></ul>	<b>MISSION OF TEC</b> <ul style="list-style-type: none"><li>Empower students through effective teaching and learning process for the development of critical thinking, effective communication and creativity.</li><li>Develop industry readiness by encouraging learning by doing, exposing current innovations and providing adequate facilities for Research.</li><li>Create the entrepreneurship desire by developing individual skills, professional ethics, moral values and societal concern.</li></ul>
<b>VISION OF DEPARTMENT</b> <ul style="list-style-type: none"><li>To create the most conducive environment for quality academic, prepare the students for a globalized technological advancement and endeavor to promote Entrepreneurial culture</li></ul>	<b>MISSION OF DEPARTMENT</b> <ul style="list-style-type: none"><li>Providing a comprehensive outcome based engineering education and a state of art infrastructure to empower students for the development of their subject proficiency, critical thinking, effective communication and creativity.</li><li>To achieve industry readiness by imparting in-depth knowledge, hands on experience and research provisions.</li><li>To inculcate professional behaviour, strong ethical values and leadership abilities in the young minds so as to instil societal concern and entrepreneurship desire.</li></ul>

**Program Educational Objectives (PEOs)****After 3-5 years of graduation, our graduates will become**

<b>PEO 1</b>	Core Competency	Apply their technical competence in computer science to solve real world problems, with technical and people leadership.
<b>PEO 2</b>	Life Long Learning	Conduct cutting edge research and develop solutions on problems of social relevance.
<b>PEO 3</b>	Professional Skills and Ethical Values	Work in a business environment, exhibiting team skills, work ethics, adaptability and lifelong learning.

**PROGRAM OUTCOMES (POs):**

<b>PO 1</b>	<b>Engineering knowledge</b>	Apply the knowledge of mathematics, science, engineering fundamentals and an engineering specialization to the solution of complex engineering problems.
<b>PO 2</b>	<b>Problem analysis</b>	Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.
<b>PO 3</b>	<b>Design/ development of solutions</b>	Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.
<b>PO 4</b>	<b>Conduct investigations of complex problems</b>	Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.
<b>PO 5</b>	<b>Modern tool usage</b>	Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.
<b>PO 6</b>	<b>The Engineer and Society</b>	Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.
<b>PO 7</b>	<b>Environment and sustainability</b>	Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.
<b>PO 8</b>	<b>Ethics</b>	Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.



## THAMIRABHARANI ENGINEERING COLLEGE

(Approved by AICTE, New Delhi and Affiliated to Anna University, Chennai)  
Chathirampudukulam, Chidambaranagar - Vepemkulam Road.  
Thatchanallur, Tirunelveli 627 358, Tamil Nadu.



<b>PO 9</b>	<b>Individual and team work</b>	Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.
<b>PO 10</b>	<b>Communication</b>	Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.
<b>PO 11</b>	<b>Project management and finance</b>	Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.
<b>PO 12</b>	<b>Life-long learning</b>	Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

### PROGRAM SPECIFIC OUTCOMES (PSOs):

After completion of Computer Science and Engineering program the student will have,

<b>PSO 1</b>	Exhibit design and programming skills to build and automate business solutions using cutting edge technologies.
<b>PSO 2</b>	Strong theoretical foundation leading to excellence and excitement towards research, to provide elegant solutions to complex problems.
<b>PSO 3</b>	Ability to work effectively with various engineering fields as a team to design, build and develop system applications.

Course Instructor

HoD



## **PART II**

### **SYLLABUS AS PER ANNA UNIVERSITY REGULATION 2021**

#### **COURSE OBJECTIVES:**

- To learn the various phases of compiler.
- To learn the various parsing techniques.
- 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.

#### **CS3501 COMPILER DESIGN LABORATORY**

**L T P C 3 0 2 4**

#### **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. Implementation of calculator using LEX and YACC
4. Program to recognize a valid control structures syntax of C language
  - a. Branching statements (if-else, if-else-if and switch-case).
  - b. Looping statements (for loop and while loop).
5. Generate three address code for a simple program using LEX and YACC.
6. Implement type checking.
7. Implement simple code optimization techniques (Constant folding, Strength reduction and Algebraic transformation)
8. 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.

**TOTAL: 30 PERIODS**

#### **COURSE OUTCOMES:**

<b>CO 1</b>	Understand the techniques in different phases of a compiler.
<b>CO 2</b>	Design a lexical analyser for a sample language and learn to use the LEX tool.
<b>CO 3</b>	Apply different parsing algorithms to develop a parser and learn to use YACC tool
<b>CO 4</b>	Understand semantics rules (SDT), intermediate code generation and run-time environment.
<b>CO 5</b>	Implement code generation and apply code optimization techniques.

#### **TEXT BOOKS:**

1. Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman, "Compilers: Principles, Techniques and Tools", Second Edition, Pearson Education, 2009.

#### **REFERENCES:**

1. Randy Allen, Ken Kennedy, Optimizing Compilers for Modern Architectures: A Dependence based Approach, Morgan Kaufmann Publishers, 2002.
2. Steven S. Muchnick, Advanced Compiler Design and Implementation, Morgan Kaufmann Publishers - Elsevier Science, India, Indian Reprint 2003.



## THAMIRABHARANI ENGINEERING COLLEGE

(Approved by AICTE, New Delhi and Affiliated to Anna University, Chennai)  
Chathirampudukulam, Chidambaranagar - Vepemkulam Road.  
Thatchanallur, Tirunelveli 627 358, Tamil Nadu.



3. Keith D Cooper and Linda Torczon, Engineering a Compilerll, Morgan Kaufmann Publishers Elsevier Science, 2004.
4. V. Raghavan, Principles of Compiler Designll, Tata McGraw Hill Education Publishers, 2010.
5. Allen I. Holub, Compiler Design in Cll, Prentice-Hall Software Series, 1993.



**PART III**  
**LESSON PLAN**

Ex. No	Date	Exercise	COs	Planned Session - 4 hrs
1.		Install and explore the core Python libraries (NumPy, SciPy, Pandas, Statsmodels, and Jupyter Notebook) used in data science. Understand their individual roles and how they integrate to support data analysis workflows.	CO1	1
2.		Read data from CSV, Excel, and web sources using Pandas. Analyze the Iris dataset to perform basic descriptive statistics and explore data using head(), describe(), and visualization commands.		1
3.		Create and manipulate Pandas Data Frames to perform indexing, slicing, filtering, merging, grouping, and generating pivot tables. Practice handling missing data and computing summary statistics.	CO3	1
4.		Analyze univariate characteristics of the Diabetes and Pima Indians datasets. Compute frequency, mean, median, mode, variance, standard deviation, skewness and Kurtosis to summarize individual variables.		1
5.		Perform bivariate and multivariate regression (linear and logistic) on Diabetes and Pima datasets. Build regression models and compare their effectiveness using error metrics and R <sup>2</sup> values across both datasets.		1
6.		Compute correlation coefficients and visualize relationships using scatter plots. Develop regression models and interpret the regression line, standard error of estimate, and R <sup>2</sup> values.		1
7.		Given a dataset with missing values and multiple features, perform data normalization and data transformation.		1
8.		Apply cross-validation to evaluate the performance of a predictive model, and use accuracy metrics such as ROC curves and precision-recall curves to assess the	CO4	2

		model's effectiveness.		
9.		Using a dataset with continuous and categorical features, apply feature selection techniques to identify the most relevant features for predicting a continuous target.	CO4	2
10.		Implement a similarity-based learning method and evaluate the model's performance by measuring errors and analyzing error surfaces to refine the model's accuracy.	CO5	2

**Course Instructor**

**HoD**





## THAMIRABHARANI ENGINEERING COLLEGE

(Approved by AICTE, New Delhi and Affiliated to Anna University, Chennai)  
Chathirampudukulam, Chidambaranagar - Vepemkulam Road.  
Thatchanallur, Tirunelveli 627 358, Tamil Nadu.



### PART IV

#### A. COURSE ASSESSMENT MATRIX

Course Outcome	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12	PSO1	PSO2	PSO3
CO 1	3	3	3	3	-	-	-	-	3		1	3	2	3	2
CO 2	3	3	3	3	3	-	-	-	3		3	2	2	1	2
CO 3	3	3	2	2	3	-	-	-	3		1	1	2	2	3
CO 4	3	2	2	1	1	-	-	-	2		2	3	1	2	1
CO 5	3	3	3	2	1	-	-	-	2		1	3	2	1	1

Competency addresses outcome:- 1 = Slightly; 2 = Moderately; 3= Substantially

**\*Explanation of table to be ascertained as separate attachment**



**THAMIRABHARANI ENGINEERING COLLEGE**

(Approved by AICTE, New Delhi and Affiliated to Anna University, Chennai)  
 Chathirampudukulam, Chidambaranagar - Vepemkulam Road.  
 Thatchanallur, Tirunelveli 627 358, Tamil Nadu.

**B. COURSE ASSESSMENT MATRIX JUSTIFICATION**

	CO1	CO2	CO3	CO4	CO5	JUSTIFICATION
PO1	3	3	3	3	3	Students will study the fundamentals of basic constructs in compiler.
PO2	3	3	3	2	3	Students able to write programs both in lex and yacc specification to verify the grammar.
PO3	3	3	2	2	3	Students able to write programs to check the compatibility of the types.
PO4	3	3	2	1	2	Students able to write programs for basic constructs with any type of problem definition.
PO5	-	3	3	1	1	Students will practice the programs Dev C++ and Flex tool.
PO6	-	-	-	-	-	
PO7	-	-	-	-	-	
PO8	-	-	-	-	-	
PO9	3	3	3	2	2	Students able to work as a team and assign tasks with timelines to complete the project.
PO10						
PO11	1	3	1	2	1	Students will able to develop a project based on compiler to handle short keywords.
PO12	3	2	1	3	3	Students will possess the proficiency to write programs for any type of problems.
PSO 1	2	2	2	1	2	Students able to write and run programs in linux platform through GCC Compiler.
PSO 2	3	1	2	2	1	Students able to write programs in validating the expressions and declarations.
PSO 3	2	2	3	1	1	Students able to design a backend of a compiler that recognize and parse grammars.

**Course Instructor****HoD**

**EX. NO: 1****DEVELOP A LEXICAL ANALYZER TO RECOGNIZE  
A FEW PATTERNS IN C****AIM**

To write a C program to develop a lexical analyzer to recognize a few patterns in C.

**ALGORITHM**

Step 1: Start the program.

Step 2: Initialize Variables

- $i = 0$ , symbolCount = 0 to track the current position and number of symbols.
- Arrays keys, func to store C keywords and standard library functions.
- SymbolTable structure to store identifiers.

Step-3: Define Helper Functions:

- addToSymbolTable(identifier):
  - Check if the identifier already exists in the symbol table.
  - If not, add it to the table and increment symbolCount.
- printSymbolTable():
  - Print the symbol table with indices and identifiers.
- keyw(p):
  - Compare the token p against the list of keywords and functions.
  - If p matches a keyword, print "keyword".
  - If p matches a standard library function, print "library function".
  - If p is a number, print "number".
  - If p is an identifier, print "identifier" and add it to the symbol table.

Step-4: Input and Open Source File:

- Prompt the user to enter the source file name.
- Open the file using fopen().

Step-5: Lexical Analysis:

- While not end of file (EOF):
  - Identify Operators:
    - If a character matches an operator, print "operator" and call keyw(str).
  - Identify Separators:
    - If a character matches a separator:
      - Handle header files (#include) and string literals.
      - Otherwise, call keyw(str) and reset i.
  - Identify Symbols:
    - If a character matches a symbol (excluding whitespace, #, <, >, newline, and tab), print "symbol".
  - Accumulate Characters:
    - If  $i \neq -1$ , accumulate characters into str.
    - Otherwise, reset i to 0.

Step-6: Print Symbol Table:

- After processing the file, call `printSymbolTable()`.

Step-7: Handle File Not Found:

- If the file cannot be opened, print an error message and exit the program.

Step 8: Stop the program.

## **PROGRAM**

```
#include<stdio.h>
#include<string.h>
#define MAX_IDENTIFIERS 100
#define MAX_IDENTIFIER_LENGTH 100
void keyw(char *p);
int i=0,symbolCount = 0;
char keys[32][10]={"auto","break","case","char","const","continue","default",
"do","double","else","enum","extern","float","for","goto","if","int","long","register","return",
"short","signed","sizeof","static","struct","switch","typedef","union",
"unsigned","void","volatile","while"};
char func[3][10]={"printf","scanf","main"};
struct SymbolTable
{
    char identifier[MAX_IDENTIFIER_LENGTH];
} symbolTable[MAX_IDENTIFIERS];
void addToSymbolTable(const char *identifier)
{
    for (int i = 0; i < symbolCount; i++)
    {
        if (strcmp(symbolTable[i].identifier, identifier) == 0)
        {
            return;
        }
    }
    strcpy(symbolTable[symbolCount].identifier, identifier);
    symbolCount++;
}
void printSymbolTable()
{
    printf("\nSymbol Table:\n");
    printf("Index\tIdentifier\n");
    for (int i = 0; i < symbolCount; i++)
    {
        printf("%d\t%s\n", i, symbolTable[i].identifier);
    }
}
main()
{
    char ch,str[25],seps[15]="\t\n,;(){}[]#\"<>",oper[]="!%^&*-.+=~|.;<>/?";
    int j;
    char fname[50];
    FILE *f1;
```

```

printf("enter file name\n");
scanf("%s",fname);
f1 = fopen(fname,"r");
if(f1!=NULL)
{
    while((ch=fgetc(f1))!=EOF)
    {
        for(j=0;j<=14;j++)
        {
            if(ch==oper[j])
            {
                printf("%c is an operator\n",ch);
                str[i]='\0';
                keyw(str);
            }
        }
        for(j=0;j<=14;j++)
        {
            if(i==1)
                break;
            if(ch==seps[j])
            {
                if(ch=='#')
                {
                    while(ch!='>')
                    {
                        printf("%c",ch);
                        ch=fgetc(f1);
                    }
                    printf("%c is a header file\n",ch);
                    i=-1;
                    break;
                }
                if(ch=="")
                {
                    do
                    {
                        ch=fgetc(f1);
                        printf("%c",ch);
                    } while(ch!="");
                    printf("\b is an argument\n");
                    i=-1;
                    break;
                }
            }
        }
    }
}

```

```

        str[i]='\0';
        keyw(str);

    }
}
for(j=0;j<=14;j++)
{
    if(ch==seps[j] && ch!=' ' && ch!='#')
    {
        if(ch!='<'&&ch!='>'&&ch!='\n'&&ch!='\t')
            printf("%c is a symbol\n",ch);
    }
}
if(i!=-1)
{
    str[i]=ch;
    i++;
}
else
    i=0;
}
printSymbolTable();
}
else
{
    printf("file not found");
    exit(0);
}
}
void keyw(char *p)
{
    int k,flag=0;
    for(k=0;k<=31;k++)
    {
        if(strcmp(keys[k],p)==0)
        {
            printf("%s is a keyword\n",p);
            flag=1;
            break;
        }
    }
    for(k=0;k<=2;k++)
    {
        if(strcmp(func[k],p)==0)

```

```

        {
            printf("%s is a library function\n",p);
            flag=1;
            break;
        }
    }
    if(flag==0)
    {
        if(isdigit(p[0]))
        {
            printf("%s is a number\n",p);
        }
        else
        {
            if(p[0]!='\0')
            {
                printf("%s is an identifier\n",p);
                addToSymbolTable(p);
            }
        }
    }
    i=-1;
}

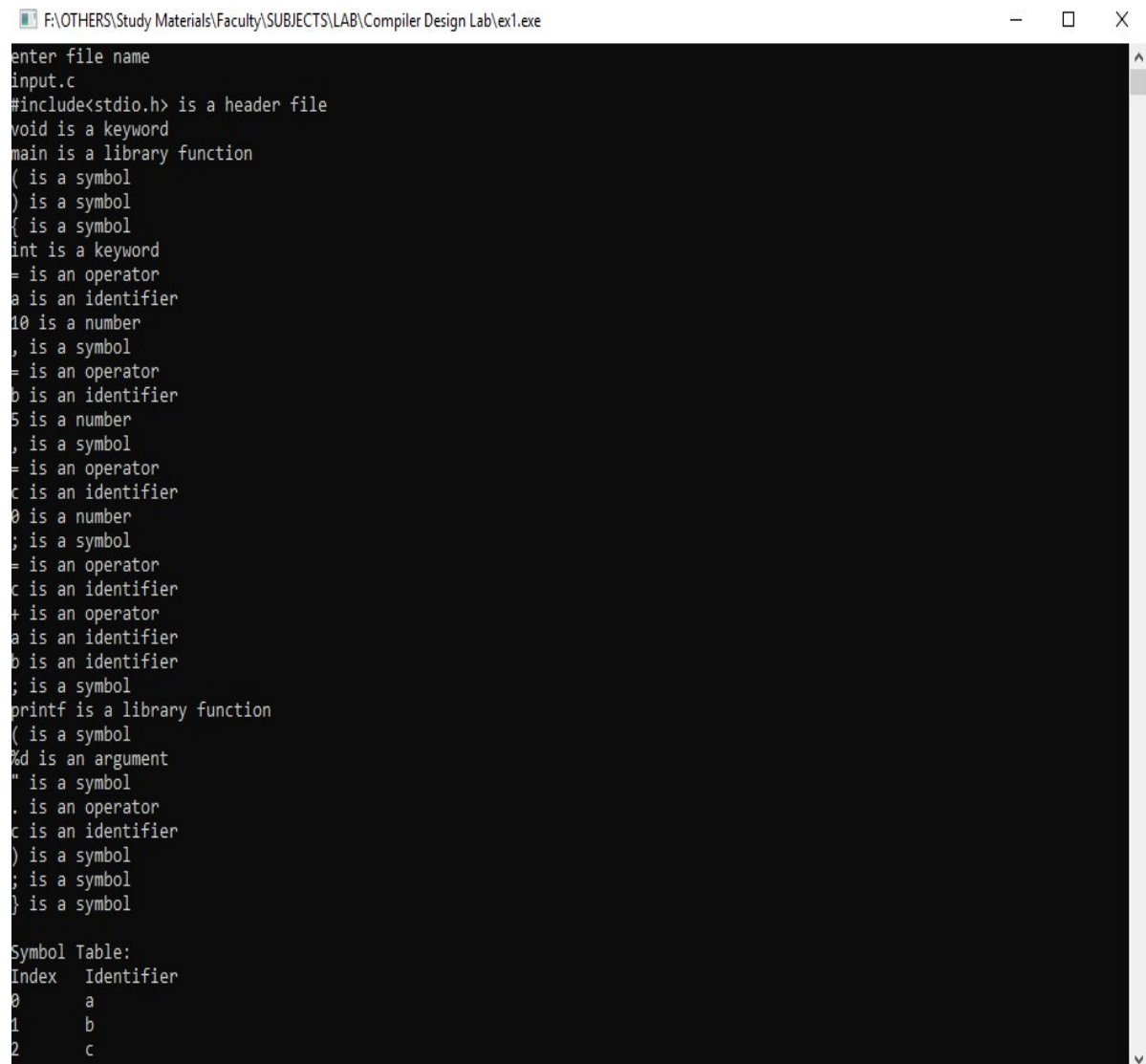
```



## INPUT: (INPUT.C)

```
#include<stdio.h>
void main()
{
    int a=10,b=5,c=0;
    c=a+b;
    printf("%d".c);
}
```

## OUTPUT



The screenshot shows a compiler window titled "F:\OTHERS\Study Materials\Faculty\SUBJECTS\LAB\Compiler Design Lab\ex1.exe". The window displays the source code for "input.c" and its symbol table. The code is as follows:

```
enter file name
input.c
#include<stdio.h> is a header file
void is a keyword
main is a library function
( is a symbol
) is a symbol
{ is a symbol
int is a keyword
= is an operator
a is an identifier
10 is a number
, is a symbol
= is an operator
b is an identifier
5 is a number
, is a symbol
= is an operator
c is an identifier
0 is a number
; is a symbol
+ is an operator
a is an identifier
b is an identifier
; is a symbol
printf is a library function
( is a symbol
%d is an argument
" is a symbol
. is an operator
c is an identifier
) is a symbol
; is a symbol
} is a symbol
```

The symbol table is displayed below the code:

Index	Identifier
0	a
1	b
2	c

## **RESULT**

Thus the above program for developing the lexical analyzer and recognizing the tokens in C was executed successfully and the output was verified.

## **EX.NO:2      IMPLEMENTATION OF LEXICAL ANALYZER USING LEX TOOL**

### **AIM**

To write a program to implement the Lexical Analyzer using lex tool.

### **ALGORITHM**

Step-1: Start the program

Step-2: Initialize Counters:

- Initialize counters e, k, c, d, i, and s to 0. These will track the number of digits, identifiers, operators, delimiters, keywords, and symbols, respectively.

Step-3: Define LEX Rules:

- Keywords: Recognize common C keywords like include, void, main, int, etc., and increment the i counter. Print the matched keyword.
- Identifiers: Match sequences starting with a lowercase letter followed by alphanumeric characters. Increment the k counter and print the identifier.
- Digits: Match sequences of digits and increment the e counter. Print the digits.
- Operators: Match common arithmetic and assignment operators, increment the c counter, and print the operator.
- Delimiters: Match characters like semicolons, parentheses, braces, commas, newlines, etc., increment the d counter, and print the delimiter.
- Symbols: Match symbols like #, <, >, and %, increment the s counter, and print the symbol.

Step-4: Open Input File:

- Open the file input.txt for reading using fopen().

Step-5: Perform Lexical Analysis:

- Call yylex() to perform lexical analysis based on the defined rules. The matched tokens are processed and counted accordingly.

Step-6: Print Summary:

- After processing, print the total counts for identifiers, symbols, digits, operators, keywords, and delimiters.

Step-7: Handle End of Input:

- Implement yywrap() to return 1, indicating the end of input for the lexer.

Step-8: Stop the program.

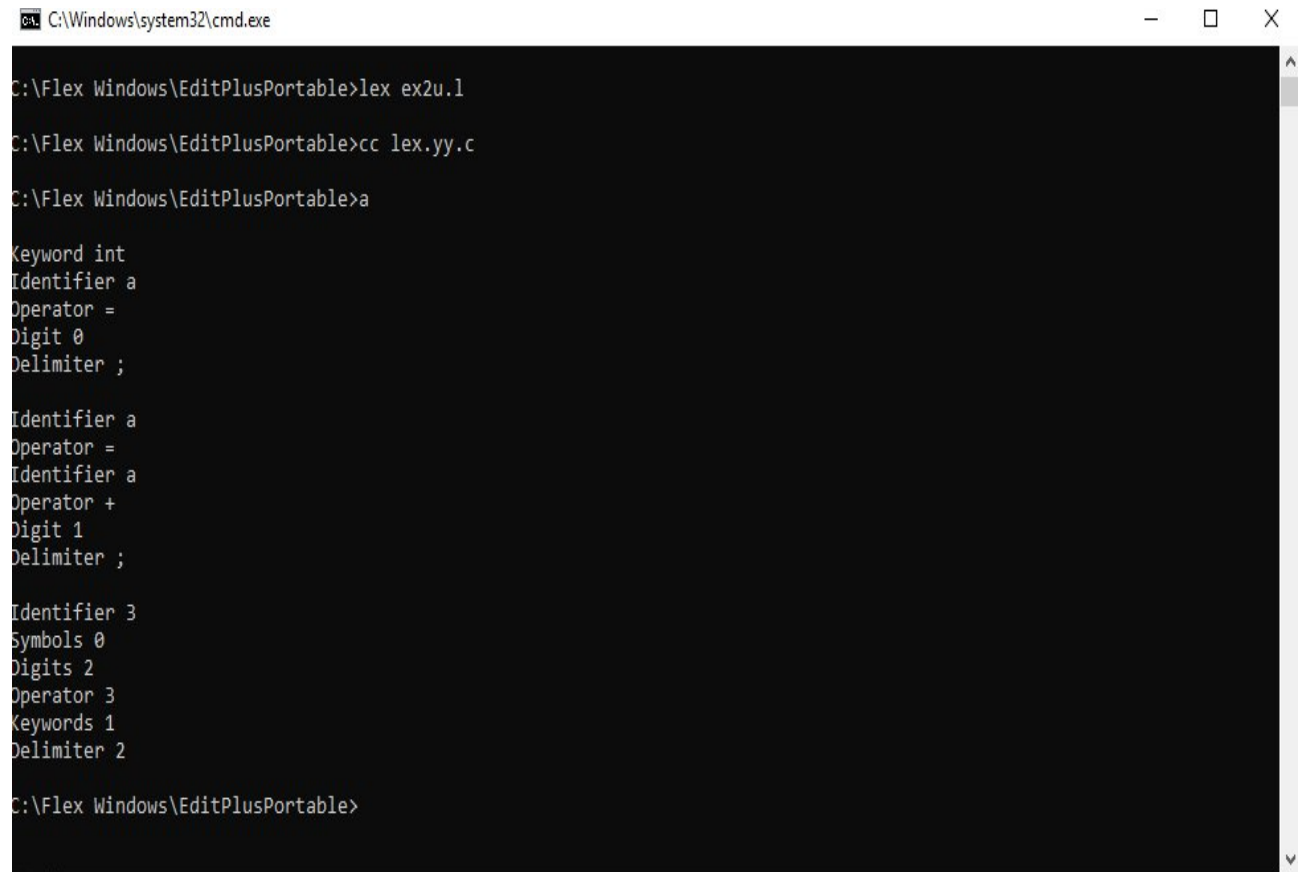
## **PROGRAM**

```
%{
#include<stdio.h>
int e,k,c,d,i,s;
}%
%%
include|void|main|int|float|double|scanf|char|printf {printf("\nKeyword %s",yytext); i++;}
[a-z][a-zA-Z0-9]* {printf("\nIdentifier %s",yytext); k++;}
[0-9]* {printf("\nDigit %s",yytext); e++;}
[+|-|*|/=]* {printf("\nOperator %s",yytext); c++;}
[;|:|(|)|{|}|}|"|'|,|\n|\\t]* {printf("\nDelimiter %s",yytext); d++;}
[#|<|>|%]* {printf("\nSymbols %s",yytext); s++;}
%%
int main(void)
{
yyin=fopen("input.txt","r");
yylex();
printf("\n\nIdentifier %d\n",k);
printf("Symbols %d\n",s);
printf("Digits %d\n",e);
printf("Operator %d\n",c);
printf("Keywords %d\n",i);
printf("Delimiter %d\n",d);
return 1;
}
int yywrap()
{
return 1;
}
```

### INPUT (input.txt)

```
int a=0;  
a=a+1;
```

### OUTPUT



```
C:\Windows\system32\cmd.exe  
C:\Flex Windows\EditPlusPortable>lex ex2u.l  
C:\Flex Windows\EditPlusPortable>cc lex.yy.c  
C:\Flex Windows\EditPlusPortable>a  
Keyword int  
Identifier a  
Operator =  
Digit 0  
Delimiter ;  
  
Identifier a  
Operator =  
Identifier a  
Operator +  
Digit 1  
Delimiter ;  
  
Identifier 3  
Symbols 0  
Digits 2  
Operator 3  
Keywords 1  
Delimiter 2  
  
C:\Flex Windows\EditPlusPortable>
```

## **RESULT**

Thus the program to implement lexical analyzer using lex was executed successfully and the output was verified.

## **EX. NO 3A    PROGRAM TO RECOGNIZE A VALID ARITHMETIC EXPRESSION THAT USES OPERATOR +, -, \* AND / USING YACC**

### **AIM**

To write a Yacc program to validate arithmetic expression.

### **ALGORITHM**

Step-1: Start the program

Step-2: Initialization:

- Include necessary header files: <ctype.h>, <stdlib.h>, <string.h>, <stdio.h>.
- Define the token type as double using #define YYSTYPE double.

Step-3: Define Tokens and Precedence:

- Declare a token num for numbers.
- Set precedence for operators:
  - + and - have left-associative precedence.
  - \* and / have left-associative precedence.

Step-3: Grammar Rules:

- st (Start Rule):
  - If an expression followed by a newline is recognized, print "VALID".
  - Handle newlines separately.
  - If an error is encountered followed by a newline, print "INVALID".
- expr (Expression Rule):
  - An expression can be a number (num).
  - An expression can be an operation between two expressions (using +, -, \*, or /).

Step-4: Main Function:

- Print a prompt asking the user to "ENTER AN EXPRESSION TO VALIDATE".
- Call yyparse() to start parsing the input.

Step-5: Lexical Analyzer (yylex):

- Skip any leading whitespace.
- If the input is a digit or a decimal point, read it as a number and return the num token.
- Return other characters (like operators) as they are.

Step-6: Error Handling:

- Implement yyerror to handle syntax errors by printing an error message.

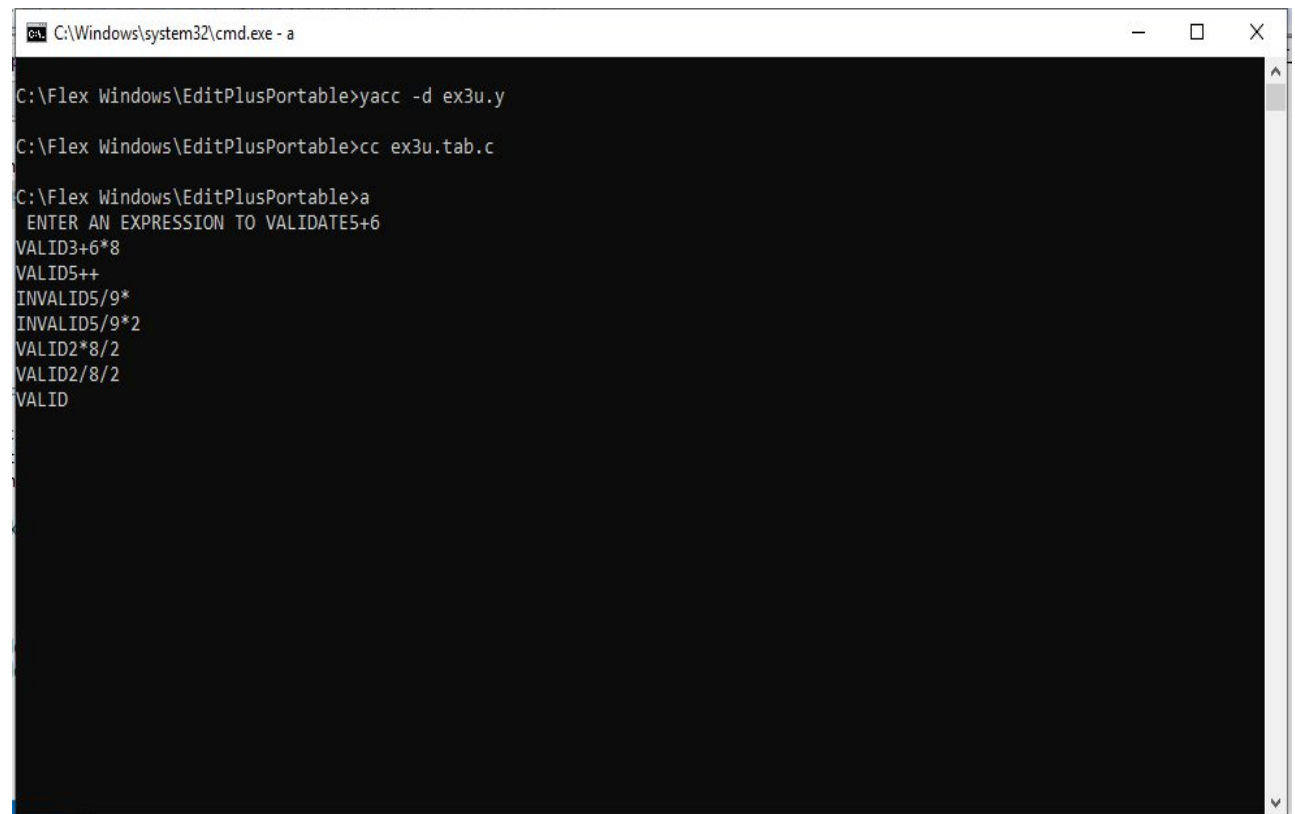
Step-7: Stop the program.

## **PROGRAM**

```
%{
#include<ctype.h>
#include<stdlib.h>
#include<string.h>
#include<stdio.h>
#define YYSTYPE double
%}
%token num
%left '+' '-'
%left '*' '/'
%%
st: st expr '\n' {printf("VALID");}
|st '\n'
|
|error '\n' {printf("INVALID");}
;
expr: num
|expr '+' expr
|expr '/' expr
|expr '-' expr
|expr '*' expr
%%
main()
{
printf(" ENTER AN EXPRESSION TO VALIDATE");
yyparse();
}
yylex()
{
int ch;
while((ch=getchar())!=' ');
if(isdigit(ch)|ch=='.')
{
ungetc(ch,stdin);
scanf("%lf",&yylval);
return num;
}
return ch;
}
yyerror(char *s)
{
printf("%S",s);
}
```



## OUTPUT



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

C:\Flex Windows\EditPlusPortable>yacc -d ex3u.y
C:\Flex Windows\EditPlusPortable>cc ex3u.tab.c
C:\Flex Windows\EditPlusPortable>a
ENTER AN EXPRESSION TO VALIDATE5+6
VALID3+6*8
VALID5++
INVALID5/9*
INVALID5/9*2
VALID2*8/2
VALID2/8/2
VALID
```

**RESULT**

Thus the program for validating arithmetic expression using yacc was executed successfully and the output was verified.

**Ex. No: 3B    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 check valid variable followed by letter or digits

**ALGORITHM**

Step-1: Start the program

Step-2: Initialization:

- Include necessary header files: <stdio.h> and <ctype.h>.
- Define tokens let for letters and dig for digits using %token.

Step-3: Grammar Rules:

- sad (Start Rule):
  - If the input starts with a letter (let) followed by a valid sequence (recl) and ends with a newline (\n), print "accepted" and return 0.
  - If the input is just a letter followed by a newline, print "accepted" and return 0.
  - If an error is encountered, print "rejected" and return 0.
- recl (Recursive Rule):
  - A valid sequence (recl) can be a letter followed by recl, a digit followed by recl, a single letter, or a single digit.

Step-4: Lexical Analyzer (yylex):

- Skip any leading spaces.
- If the input character is a letter or an underscore (\_), return the let token.
- If the input character is a digit, return the dig token.
- Return any other character as is.

Step-5: Error Handling:

- Implement yyerror to print "rejected" if an invalid sequence is encountered.

Step-6: Main Function:

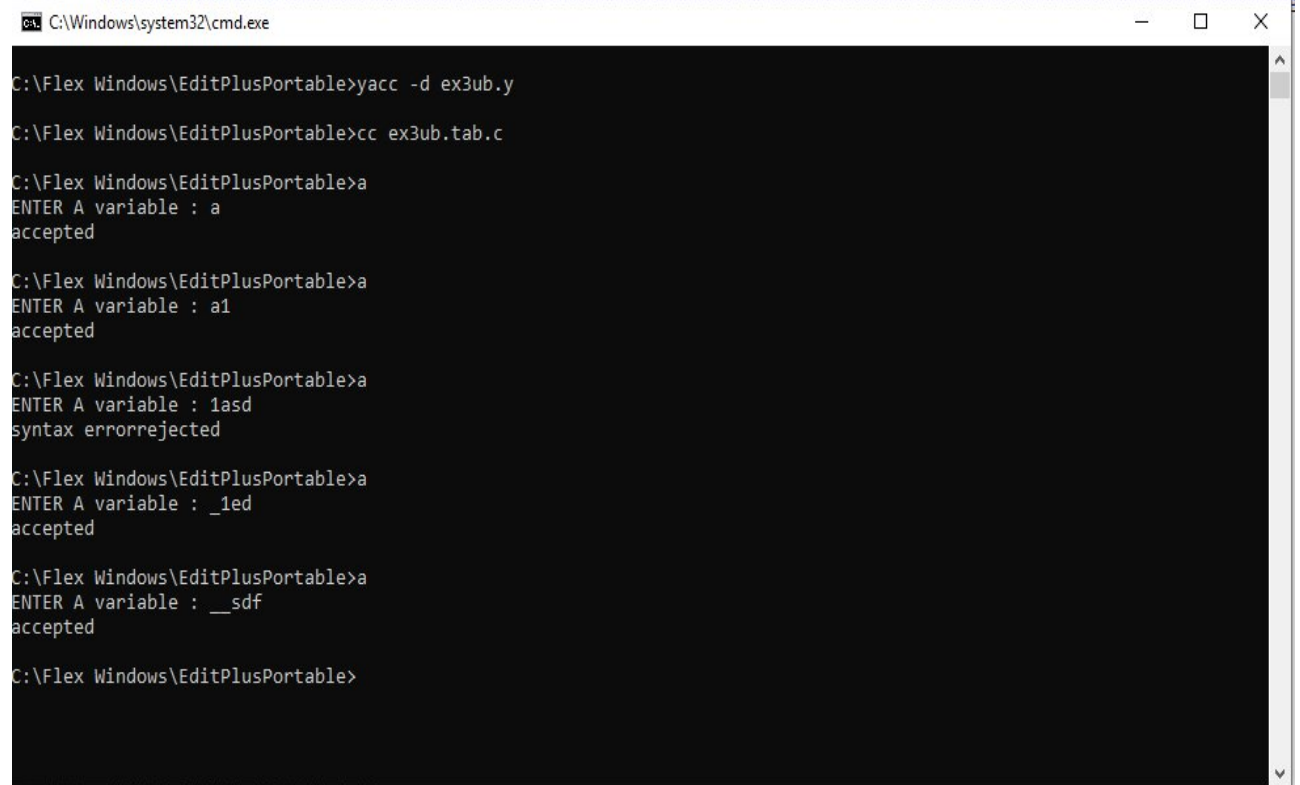
- Print a prompt asking the user to "ENTER A variable".
- Call yyparse() to start parsing the input.

Step-7: Stop the program.

## **PROGRAM**

```
%{
#include<stdio.h>
#include<ctype.h>
%}
%token let dig
%%
sad: let reclk '\n' {printf("accepted\n"); return 0;}
| let '\n' {printf("accepted\n"); return 0;}
|
|error {yyerror("rejected\n");return 0;}
;
reclk: let reclk
| dig reclk
| let
| dig
;
%%
yylex()
{
char ch;
while((ch=getchar())!=' ');
if(isalpha(ch) ||ch=='_')
return let;
if(isdigit(ch))
return dig;
return ch;
}
yyerror(char *s)
{
printf("%s",s);
}
main()
{
printf("ENTER A variable : ");
yyparse();
}
```

## OUTPUT



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

C:\Flex Windows\EditPlusPortable>yacc -d ex3ub.y

C:\Flex Windows\EditPlusPortable>cc ex3ub.tab.c

C:\Flex Windows\EditPlusPortable>a
ENTER A variable : a
accepted

C:\Flex Windows\EditPlusPortable>a
ENTER A variable : a1
accepted

C:\Flex Windows\EditPlusPortable>a
ENTER A variable : 1asd
syntax errorrejected

C:\Flex Windows\EditPlusPortable>a
ENTER A variable : _1ed
accepted

C:\Flex Windows\EditPlusPortable>a
ENTER A variable : __sdf
accepted

C:\Flex Windows\EditPlusPortable>
```

**RESULT**

Thus the program for checking letter followed by letter or digits was executed successfully and the output was verified.

## **Ex. No: 3C   IMPLEMENTATION OF CALCULATOR USING LEX AND YACC**

### **AIM**

To write a program to implement calculator using lex and yacc

### **ALGORITHM**

#### **Lexical Analysis (cal.l):**

Step-1: Start the program.

Step-2: Initialization:

- Include necessary headers and declare external variables.

Step-3: Token Definitions:

- Identify numeric sequences [0-9]+ and convert them to integer tokens (NUMBER).
- Ignore whitespace ([\t]).
- Return newline ([\n]) and any other characters as is.

Step-4: End of Input:

- Implement yywrap() to indicate the end of input.

Step-5: Stop the program.

#### **Parsing and Evaluation (cal.y):**

Step-1: Start the program.

Step-2: Initialization:

- Include necessary headers and declare a flag for error checking.

Step-3: Token Declarations:

- Declare NUMBER token and precedence rules for arithmetic operators.

Step-4: Grammar Rules:

- Define ArithmeticExpression which will evaluate to the result of an arithmetic expression.
- Define the rules for handling arithmetic operations: addition, subtraction, multiplication, division, modulus, and parentheses.

Sep-5: Main Function:

- Prompt the user to enter an arithmetic expression.
- Call yyparse() to parse and evaluate the expression.
- Print a success message if the expression is valid.

Step-6: Error Handling:

- Implement yyerror() to handle invalid expressions by setting the error flag and printing an error message.

Step-7: Stop the program.

## **PROGRAM**

cal.l

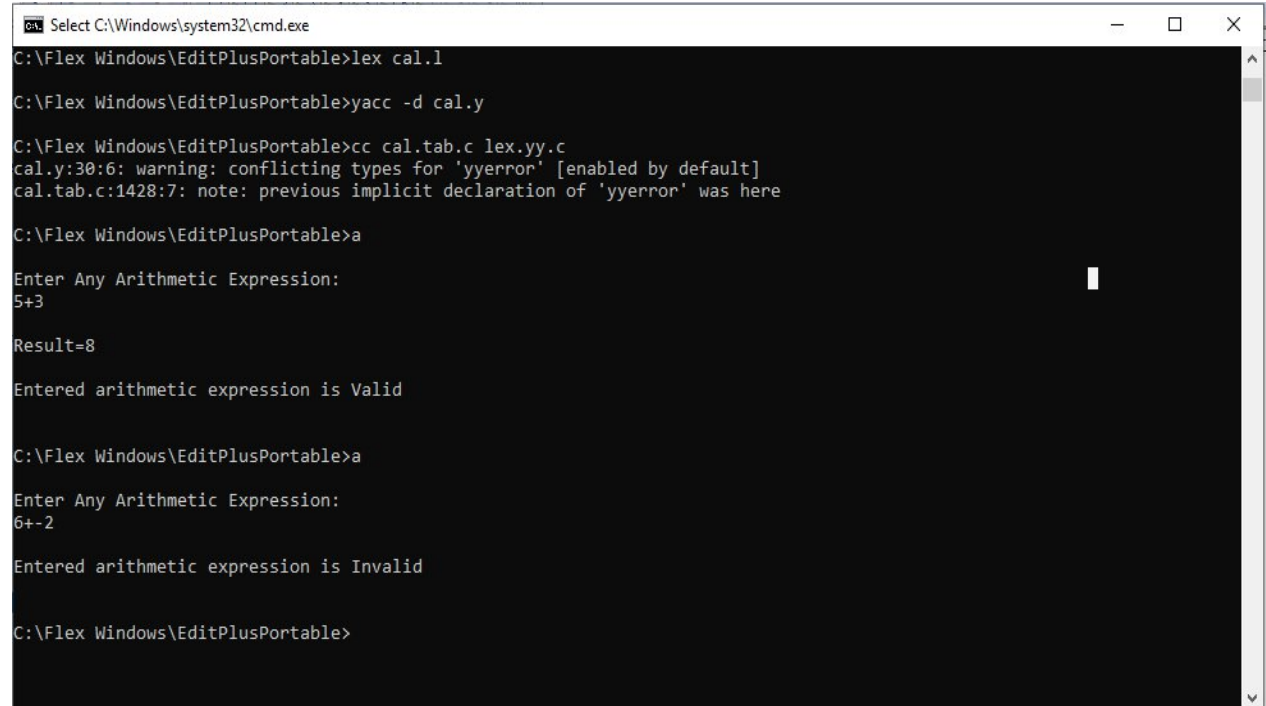
```
%{
#include<stdio.h>
#include "cal.tab.h"
extern int yylval;
%}
%%
[0-9]+ {
    yylval=atoi(yytext);
    return NUMBER;
}
[\t] ;
[\n] return 0;
. return yytext[0];
%%
int yywrap()
{
    return 1;
}
```



ex3d.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



```
Select C:\Windows\system32\cmd.exe
C:\Flex Windows\EditPlusPortable>lex cal.l
C:\Flex Windows\EditPlusPortable>yacc -d cal.y
C:\Flex Windows\EditPlusPortable>cc cal.tab.c lex.yy.c
cal.y:30:6: warning: conflicting types for 'yyerror' [enabled by default]
cal.tab.c:1428:7: note: previous implicit declaration of 'yyerror' was here
C:\Flex Windows\EditPlusPortable>a
Enter Any Arithmetic Expression:
5+3
Result=8
Entered arithmetic expression is Valid

C:\Flex Windows\EditPlusPortable>a
Enter Any Arithmetic Expression:
6+-2
Entered arithmetic expression is Invalid

C:\Flex Windows\EditPlusPortable>
```

**RESULT**

Thus the program for implementing calculator using lex and yacc was executed successfully and the output was verified.

**AIM**

To write a C program to recognize branching statements (if-else, if-else-if, switch-case).

**ALGORITHM**

Step-1: Start the program.

Step-2: Input Handling

- Read the input string containing the control structure code.
- Remove any trailing newline characters.

Step-3: Validation for if, if-else, and if-else-if Structures:

- Validating if Structure:
  1. Check if the input contains if( followed by a closing ) and { } braces.
  2. If all components are present, return true for a valid if structure.
- Validating if-else Structure:
  1. Check if the input contains if( followed by { } and an else block with { } braces.
  2. If all components are present, return true for a valid if-else structure.
- Validating if-else-if Structure:
  1. Check if the input contains if( followed by { }, an else if( block with { }, and an else block with { }.
  2. If all components are present, return true for a valid if-else-if structure.

Step-4: Validation for switch-case Structure:

- Validating switch-case:
  1. Check if the input contains switch( followed by { }.
  2. Ensure at least one case label with a valid numeric or character value and a corresponding break; statement is present.
  3. Optionally, check for a default: label.
  4. Ensure all components are correctly placed.
  5. If all components are present and valid, return true for a valid switch-case structure.

Step-5: Final Decision:

- Based on the checks, print the corresponding message:
  - If valid if, if-else, if-else-if, or switch-case structure is found, print the respective validation message.
  - If none match, print that the input is not a valid control structure.

Step-6: Stop the program.

## **PROGRAM**

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

bool isValidIf(const char *str)
{
    if(strstr(str,"if")!=NULL&&strstr(str,"")!=NULL)
    {
        if(strstr(str,"{")!=NULL&&strstr(str,"}")!=NULL)
            return 1;
    }
    return 0;
}

bool isValidIfElse(const char *str)
{
    if(strstr(str,"if")!=NULL&&strstr(str,"{")!=NULL)
    {
        if(strstr(str,"else")!=NULL&&strstr(str,"}")!=NULL)
        {
            if(strstr(str,"}")!=NULL)
                return 1;
        }
    }
    return 0;
}

bool isValidIfElseIf(const char *str)
{
    if(strstr(str,"if")!=NULL&&strstr(str,"{")!=NULL)
    {
        if(strstr(str,"else if")!=NULL &&strstr(str,"{")!=NULL)
        {
            if(strstr(str,"else") !=NULL &&strstr(str,"{") !=NULL)
            {
                if(strstr(str,"}")!=NULL)
                    return 1;
            }
        }
    }
    return 0;
}

bool isValidCaseValue(const char *str)
{
    while (*str == ' ' || *str == '\t') str++;
    if (!isdigit(*str) && *str != "\"" && *str != "")
```

```

        {
            return false;
        }
        return true;
    }
bool isValidSwitchCase(const char *str)
{
    const char *switchPtr = strstr(str, "switch(");
    const char *casePtr = strstr(str, "case");
    const char *breakPtr = strstr(str, "break;");
    const char *defaultPtr = strstr(str, "default:");
    if (switchPtr != NULL && strchr(str, '{') != NULL)
    {
        if(strchr(str, '}') != NULL)
        {
            if (casePtr == NULL || breakPtr == NULL)
            {
                return false;
            }
            while (casePtr != NULL)
            {
                casePtr += 4;
                if (!isValidCaseValue(casePtr))
                {
                    return false;
                }
                if (strstr(casePtr, ".") == NULL)
                {
                    return false;
                }
                casePtr = strstr(casePtr, "case");
            }
            return true;
        }
    }
    return false;;
}

int main()
{
    char input[256];
    printf("Enter a control structure code to check its validity: ");
    fgets(input, sizeof(input), stdin);
    input[strcspn(input, "\n")] = '\0';
    if (isValidIfElseIf(input))

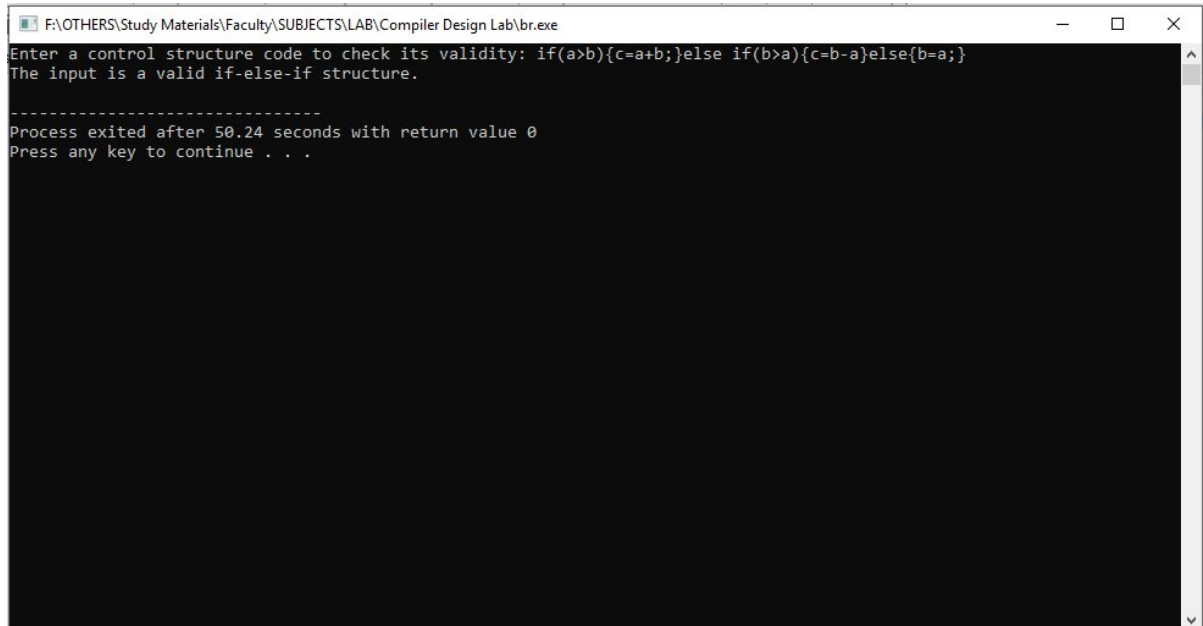
```

```

    {
        printf("The input is a valid if-else-if structure.\n");
    }
    else if (isValidIfElse(input))
    {
        printf("The input is a valid if-else structure.\n");
    }
    else if (isValidIf(input))
    {
        printf("The input is a valid if structure.\n");
    }
    else if (isValidSwitchCase(input))
    {
        printf("The input is a valid switch-case structure.\n");
    }
    else
    {
        printf("The input is not a valid control structure.\n");
    }
    return 0;
}

```

## OUTPUT



```
F:\OTHERS\Study Materials\Faculty\SUBJECTS\LAB\Compiler Design Lab\br.exe
Enter a control structure code to check its validity: if(a>b){c=a+b;}else if(b>a){c=b-a}else{b=a;}
The input is a valid if-else-if structure.

-----
Process exited after 50.24 seconds with return value 0
Press any key to continue . . .
```



## **RESULT**

Thus the above to recognize if,if-else,if-else if-else and switch-case was executed successfully and the output was verified.

**AIM**

To write a C program to recognize looping statements (for and while).

**ALGORITHM**

Step-1: Start the program.

Step-2: Declare Functions:

- trim\_whitespace(char \*str): Trims leading and trailing whitespace from the input string.
- is\_valid\_for\_loop(char \*input): Validates if the input string represents a correctly formatted for loop.
- is\_valid\_while\_loop(char \*input): Validates if the input string represents a correctly formatted while loop.
- recognize\_control\_structure(char \*input): Determines if the input string is a valid for or while loop, and prints the appropriate message.

Step-3: Main Function:

- Declare a char array input to store user input.
- Prompt the user to "Enter a control structure to validate".
- Use fgets to capture the user input and store it in input.
- Remove the newline character from the end of the input string.
- Call recognize\_control\_structure(input).

Step-4: Function recognize\_control\_structure:

- Call is\_valid\_for\_loop(input):
  - If it returns 1, print "Recognized: valid for loop".
- Else, call is\_valid\_while\_loop(input):
  - If it returns 1, print "Recognized: valid while loop".
- Else, print "No valid control structure recognized."

Step-5: Function is\_valid\_for\_loop:

- Call trim\_whitespace(input) to remove leading and trailing spaces.
- Check if the input starts with "for":
  - If true, search for the closing parenthesis ).
  - Count the number of semicolons ; between "for(" and the closing parenthesis:
    - If there are exactly 2 semicolons:
      - Check if there is an opening { and closing } brace after the closing parenthesis:
        - If both are present, return 1 (indicating a valid for loop).
- If any condition fails, return 0.

Step-6: Function is\_valid\_while\_loop:

- Call trim\_whitespace(input) to remove leading and trailing spaces.

- Check if the input starts with "while(":
  - If true, search for the closing parenthesis ).
  - Check if there is an opening { and closing } brace after the closing parenthesis:
    - If both are present, return 1 (indicating a valid while loop).
- If any condition fails, return 0.

Step-7: Function trim\_whitespace:

- Remove leading spaces by advancing the pointer until the first non-space character is found.
- Remove trailing spaces by moving a pointer from the end of the string back to the first non-space character.
- Null-terminate the string after the last non-space character.

Step-8: Stop the program.

## **PROGRAM**

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>
void trim_whitespace(char *str)
{
    char *end;
    while (isspace((unsigned char)*str))
        str++;
    if (*str == 0)
    {
        return;
    }
    end = str + strlen(str) - 1;
    while (end > str && isspace((unsigned char)*end))
        end--;
    *(end + 1) = '\0';
}
int is_valid_for_loop(char *input)
{
    trim_whitespace(input);
    if (strncmp(input, "for(", 4) == 0)
    {
        char *closing_paren = strchr(input, ')');
        if (closing_paren != NULL)
        {
            int semicolon_count = 0;
            for (char *p = input + 4; p < closing_paren; p++)
            {
                if (*p == ';')
                    semicolon_count++;
            }
            if (semicolon_count == 2)
            {
                char *opening_brace = strchr(closing_paren, '{');
                if (opening_brace != NULL && strchr(opening_brace, '}') != NULL)
                {
                    return 1;
                }
            }
        }
    }
    return 0;
}
```

```

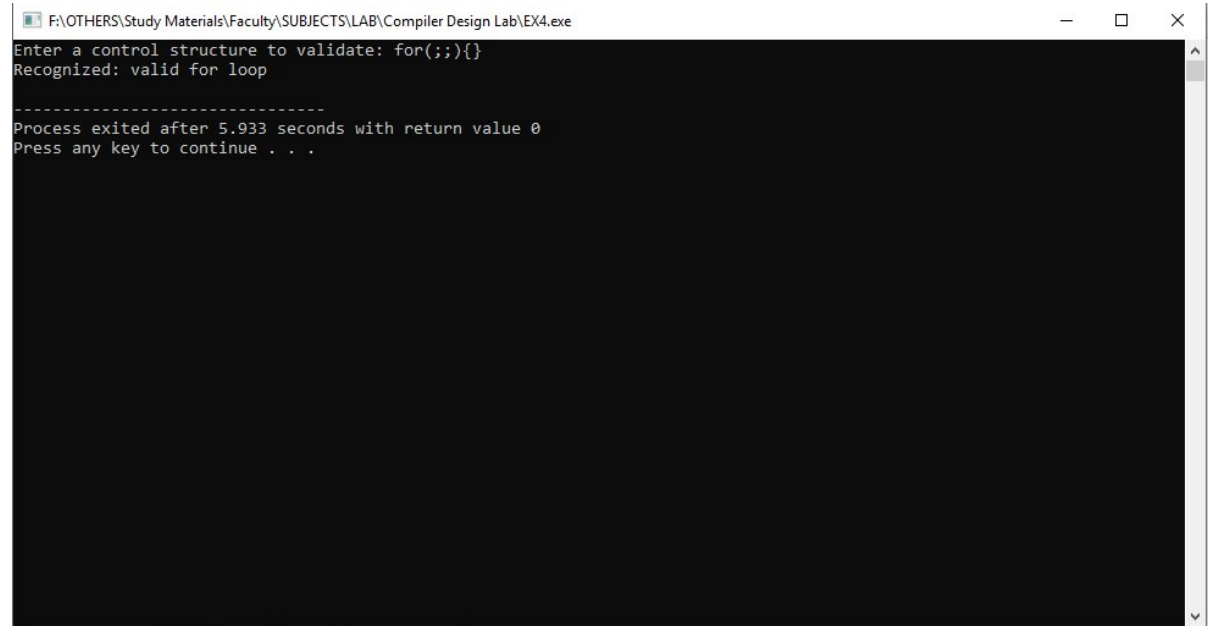
int is_valid_while_loop(char *input)
{
    trim_whitespace(input);
    if (strncmp(input, "while(", 6) == 0)
    {
        char *closing_paren = strchr(input, ');');
        if (closing_paren != NULL)
        {
            char *opening_brace = strchr(closing_paren, '{');
            if (opening_brace != NULL && strchr(opening_brace, '}') != NULL)
            {
                return 1;
            }
        }
    }
    return 0;
}

void recognize_control_structure(char *input)
{
    if (is_valid_for_loop(input))
    {
        printf("Recognized: valid for loop\n");
    }
    else if (is_valid_while_loop(input))
    {
        printf("Recognized: valid while loop\n");
    }
    else
    {
        printf("No valid control structure recognized.\n");
    }
}

int main()
{
    char input[256];
    printf("Enter a control structure to validate: ");
    fgets(input, sizeof(input), stdin);
    input[strcspn(input, "\n")] = 0;
    recognize_control_structure(input);
    return 0;
}

```

## OUTPUT



```
F:\OTHERS\Study Materials\Faculty\SUBJECTS\LAB\Compiler Design Lab\EX4.exe
Enter a control structure to validate: for(;;){}
Recognized: valid for loop

-----
Process exited after 5.933 seconds with return value 0
Press any key to continue . . .
```

**RESULT**

Thus the above to recognize for and while was executed successfully and the output was verified.

## **EX. NO: 5      GENERATE THREE ADDRESS CODE FOR A SIMPLE PROGRAM USING LEX AND YACC**

### **AIM**

To write a program to Generate Three Address Code using lex and yacc

### **ALGORITHM**

#### **three.l**

Step-1: Start the program.

Step-2: Initialization:

- Include the header file "three.tab.h" for token definitions.

Step-3: Token Recognition:

- Whitespace: Skip over spaces and tabs ([ \t]+).
- Numbers:
  - Match sequences of digits ([0-9]+).
  - Convert the matched text to an integer using atoi(yytext).
  - Store the integer in yylval and return the NUM token.
- Operators:
  - Match and return the following tokens:
    - "+" → ADD
    - "-" → SUB
    - "\*" → MUL
    - "/" → DIV
- End of Line:
  - Match a newline character ("\n").
  - Return the EOL token.
- Other Characters:
  - Return any other character as is.

Step-4: Wrap-Up Function (yywrap):

- Return 1 to indicate the end of input.

Step-5: Stop the program.



### **three.y**

Step-1: Start the program.

Step-2: Initialization:

- Include standard I/O and `stdlib.h` for printing and conversion.
- Define `temp_var = 0` to keep track of temporary variable numbers.
- Declare helper functions: `generate_op` for generating intermediate code and `yyerror` for error handling.

Step-3: Parsing Rules:

- Program (program):
  - Allows parsing of multiple statement lines, each followed by an EOL.
- Statement (statement):
  - Parses an `expr` and prints the result as an assignment to a temporary variable (`tX`).
- Expression (`expr`):
  - Handles addition (`expr ADD term`) and subtraction (`expr SUB term`).
  - Calls `generate_op` to generate intermediate code.
  - Returns the result of a single term if no operation is needed.
- Term (`term`):
  - Handles multiplication (`term MUL factor`) and division (`term DIV factor`).
  - Calls `generate_op` similarly to `expr`.
  - Returns the result of a single factor if no operation is needed.
- Factor (`factor`):
  - Returns a `NUM` directly.
  - Evaluates expressions within parentheses (`(' expr ')`).

Step-4: Intermediate Code Generation (`generate_op`):

- Generate and print intermediate code in the format `tX = tY op tZ`.
- Increment `temp_var` and return the new temporary variable number.

Step-5: Main Function:

- Call `yyparse()` to start the parsing and code generation process.

Step-6: Error Handling:

- Implement `yyerror` to print any parsing errors encountered.

Step-7: Stop the program.

## **PROGRAM**

three.l

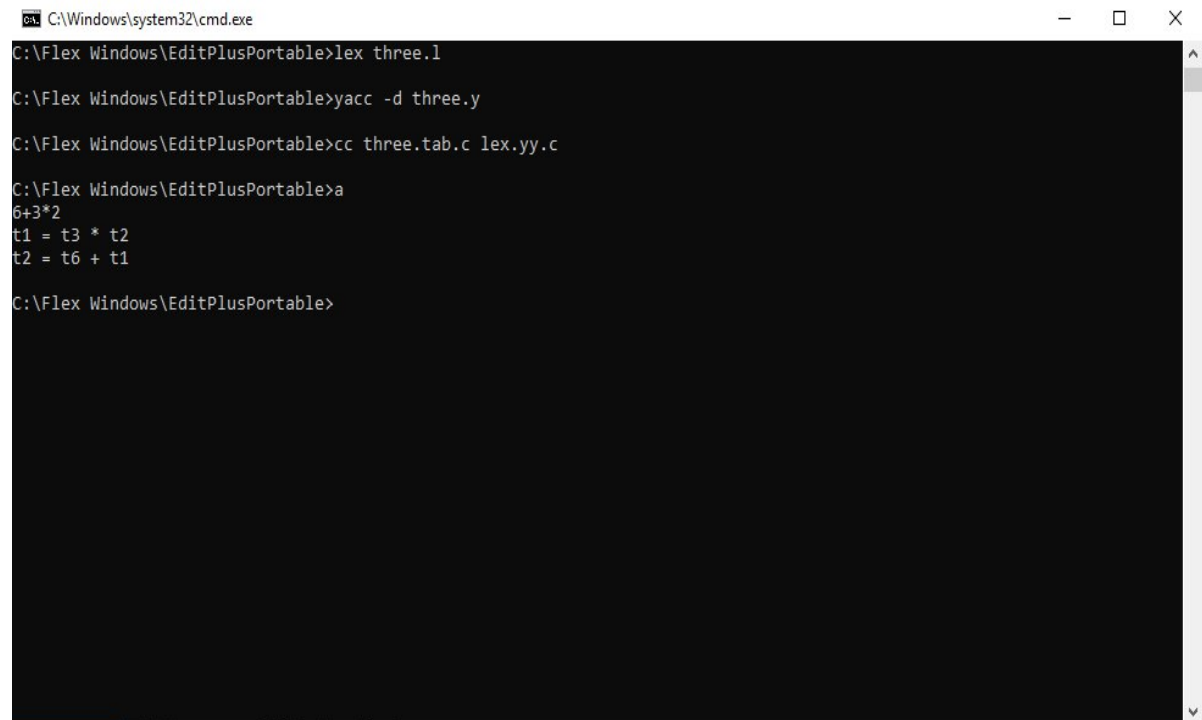
```
%{
#include "three.tab.h"
%}
%%
[ \t]+;
[0-9]+ {yyval = atoi(yytext); return NUM;}
"+"   {return ADD;}
"_"   {return SUB;}
"*"   {return MUL;}
"/"   {return DIV;}
"\n"  {return EOL;}
.     {return yytext[0];}
%%
int yywrap()
{
    return 1;
}
```

```

three.y
%{
#include <stdio.h>
#include <stdlib.h>
int temp_var = 0;
int generate_op(int left, int right, char op);
void yyerror(const char* s);
}%
%token NUM ADD SUB MUL DIV EOL
%left ADD SUB
%left MUL DIV
%%
program:
    | program statement EOL
    ;
statement:
    expr { printf("t%d = %s\n", ++temp_var, $1); }
    ;
expr:
    expr ADD term { $$ = generate_op($1, $3, '+'); }
    | expr SUB term { $$ = generate_op($1, $3, '-'); }
    | term { $$ = $1; }
    ;
term:
    term MUL factor { $$ = generate_op($1, $3, '*'); }
    | term DIV factor { $$ = generate_op($1, $3, '/'); }
    | factor { $$ = $1; }
    ;
factor:
    NUM { $$ = $1; }
    | '(' expr ')' { $$ = $2; }
    ;
%%
#include <stdlib.h>
int generate_op(int left, int right, char op)
{
    printf("t%d = t%d %c t%d\n", ++temp_var, left, op, right);
    return temp_var;
}
int main()
{
    yyparse();
    return 0;
}
void yyerror(const char* s)
{
    fprintf(stderr, "%s\n", s);
}

```

## OUTPUT



```
C:\Windows\system32\cmd.exe
C:\Flex Windows\EditPlusPortable>lex three.l
C:\Flex Windows\EditPlusPortable>yacc -d three.y
C:\Flex Windows\EditPlusPortable>cc three.tab.c lex.yy.c
C:\Flex Windows\EditPlusPortable>a
6+3*2
t1 = t3 * t2
t2 = t6 + t1
C:\Flex Windows\EditPlusPortable>
```

**RESULT**

Thus the program for implementing three address code using lex and yacc was executed successfully and the output was verified.

## EX. NO 6

## IMPLEMENTATION OF TYPE CHECKING

### AIM

To write a C program to implement type checking.

### ALGORITHM

1. Start the program.
2. Prompt the user to input the number of variables, n.
3. Input Variable Information:
  - Create arrays vari and typ to store variable names and their types.
  - Use a loop to input the variable names and their types for each of the n variables.
  - For each variable i:
    - Prompt the user to enter the variable name and store it in vari[i].
    - Prompt the user to enter the variable type (either 'f' for float or 'i' for int) and store it in typ[i].
    - If the variable type is 'f', set a flag to 1 to indicate that a float type is present.
4. Input the Expression:
  - Prompt the user to enter an expression, terminated by '\$', and store it in b.
5. Check for Division (/):
  - Use a loop to scan the characters in the expression stored in the b array.
  - If a '/' character is found in the expression, set the flag to 1.
6. Check Expression Type:
  - Use a loop to iterate over the variables you previously input.
  - For each variable i:
    - Check if the first character of the expression (b[0]) matches any of the variable names in the vari array.
    - If a match is found:
      - Check the value of the flag. If flag is 1 (indicating a float type in the expression):
        - Check if the variable's type (typ[i]) is 'f'. If it is, the expression is considered correctly typed.
        - If the variable type is not 'f', indicate a type mismatch for the variable.
      - If flag is not set, it means no division was found in the expression, and it is considered correctly typed.
    - If no match is found, continue checking the next variable.
7. Output the Result:
  - Output messages based on the results of the checks:
    - If the expression is correctly typed, print a message indicating so.
    - If the expression has a type mismatch, print an error message.
8. Stop the Program.

## **PROGRAM**

```
#include<stdio.h>
#include<stdlib.h>
void 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;
}
}
}
}

```



## OUTPUT

```
F:\OTHERS\Study Materials\Faculty\SUBJECTS\LAB\Compiler Design Lab\typecheck.exe
Enter the number of variables:3
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):i
Enter the Expression(end with $):a=b/c$
Identifier a must be a float type..!

-----
Process exited after 32.63 seconds with return value 37
Press any key to continue . . .
```

```
F:\OTHERS\Study Materials\Faculty\SUBJECTS\LAB\Compiler Design Lab\typecheck.exe
Enter the number of variables:3
Enter the variable[0]:a
Enter the variable-type[0](float-f,int-i):f
Enter the variable[1]:b
Enter the variable-type[1](float-f,int-i):f
Enter the variable[2]:c
Enter the variable-type[2](float-f,int-i):f
Enter the Expression(end with $):a=b/c
$
the datatype is correctly defined..!

-----
Process exited after 17.81 seconds with return value 0
Press any key to continue . . .
```

**RESULT**

The above program to implement type checking was executed successfully and the output was verified.

## **Ex. No: 7 IMPLEMENTATION OF SIMPLE CODE OPTIMIZATION TECHNIQUES**

### **AIM**

To write a C program to implement Constant folding, Strength reduction and Algebraic transformation.

### **ALGORITHM**

Step-1: Start the program.

Step-2: Input Reading:

- Prompt the user to input the first operand a.
- Prompt the user to input the second operand b.
- Prompt the user to input the operator op (+, -, \*, /).

Step-3: Constant Folding:

- Based on the operator op, perform the corresponding arithmetic operation on a and b.
- If the operator is /, check if b is not zero before division to avoid division by zero error.
- Print the result of the constant folding operation.

Step-4: Strength Reduction:

- If the operator is \* and b is 2, 4, or 8, replace multiplication with a left shift (<<).
- If the operator is / and b is 2, 4, or 8, replace division with a right shift (>>).
- For other values of b, perform the normal arithmetic operation.
- Print the result of the strength reduction operation.

Step-5: Algebraic Transformation:

- Simplify expressions based on algebraic identities:
  - If the operator is \*, return 0 if either operand is 0, or return the other operand if one is 1.
  - If the operator is +, return the other operand if one is 0.
  - If the operator is -, return the other operand if one is 0, or return 0 if both operands are equal.
- For other cases, perform the normal arithmetic operation.
- Print the result of the algebraic transformation.

Step-6: Stop the program.

## **PROGRAM**

```
#include <stdio.h>
#include <stdlib.h>
int constant_folding(int a, int b, char op)
{
    switch(op)
    {
        case '+':
            return a + b;
        case '-':
            return a - b;
        case '*':
            return a * b;
        case '/':
            if(b != 0)
                return a / b;
            else
            {
                printf("Division by zero error!\n");
                exit(1);
            }
        default:
            printf("Invalid operator!\n");
            exit(1);
    }
}

int strength_reduction(int a, int b, char op)
{
    switch(op)
    {
        case '*':
            if(b == 2)
                return a << 1; // a * 2 becomes a << 1
            if(b == 4)
                return a << 2; // a * 4 becomes a << 2
            if(b == 8)
                return a << 3; // a * 8 becomes a << 3
            return a * b;
        case '/':
            if(b == 2)
                return a >> 1; // a / 2 becomes a >> 1
            if(b == 4)
                return a >> 2; // a / 4 becomes a >> 2
            if(b == 8)
```

```

        return a >> 3; // a / 8 becomes a >> 3
    return a / b;
default:
    printf("Strength reduction not applicable\n");
    exit(1);
}
}
int algebraic_transformation(int a, int b, char op)
{
    switch(op)
    {
        case '*':
            if(a == 0 || b == 0)
                return 0;        // a * 0 = 0
            if(a == 1)
                return b;        // 1 * b = b
            if(b == 1)
                return a;        // a * 1 = a
            return a * b;
        case '+':
            if(a == 0)
                return b;        // 0 + b = b
            if(b == 0)
                return a;        // a + 0 = a
            return a + b;
        case '-':
            if(b == 0)
                return a;        // a - 0 = a
            if(a == b)
                return 0;        // a - a = 0
            return a - b;
        default:
            printf("Algebraic transformation not applicable\n");
            exit(1);
    }
}
int main()
{
    int a, b, result;
    char op;
    printf("Enter first operand (integer): ");
    scanf("%d", &a);
    printf("Enter second operand (integer): ");
    scanf("%d", &b);

```

```
printf("Enter operator (+, -, *, /): ");
scanf(" %c", &op);
printf("\nConstant Folding Result:\n");
result = constant_folding(a, b, op);
printf("%d %c %d = %d\n", a, op, b, result);
printf("\nStrength Reduction Result:\n");
result = strength_reduction(a, b, op);
printf("%d %c %d = %d\n", a, op, b, result);
printf("\nAlgebraic Transformation Result:\n");
result = algebraic_transformation(a, b, op);
printf("%d %c %d = %d\n", a, op, b, result);
return 0;
}
```

## OUTPUT

```
F:\OTHERS\Study Materials\Faculty\SUBJECTS\LAB\Compiler Design Lab\EX6.exe
Enter first operand (integer): 5
Enter second operand (integer): 3
Enter operator (+, -, *, /): *

Constant Folding Result:
5 * 3 = 15

Strength Reduction Result:
5 * 3 = 15

Algebraic Transformation Result:
5 * 3 = 15

-----
Process exited after 8.58 seconds with return value 0
Press any key to continue . . .
```

```
F:\OTHERS\Study Materials\Faculty\SUBJECTS\LAB\Compiler Design Lab\EX6.exe
Enter first operand (integer): 5
Enter second operand (integer): 6
Enter operator (+, -, *, /): +

Constant Folding Result:
5 + 6 = 11

Strength Reduction Result:
Strength reduction not applicable

-----
Process exited after 11.13 seconds with return value 1
Press any key to continue . . .
```

**RESULT**

Thus the program for implementing constant folding, strength reduction and algebraic transformation was executed successfully and the output was verified.



## **Ex.No: 8      IMPLEMENTATION OF THE BACK END OF THE COMPILER**

### **AIM**

To write a C program for implementing backend of a compiler.

### **ALGORITHM**

Step-1: Start the program.

Step-2: Input Initialization:

- Declare a 2D character array inp to store the input statements.
- Declare integer variables n (number of statements), i, j, len, and reg (register index, initialized to 1).

Step-3: Input Reading:

- Prompt the user to enter the number of statements (n).
- Read each statement into the inp array.

Step-4: Processing Each Statement:

- Loop through each statement (i from 0 to n-1):
  - Determine the length of the statement (len).
  - Loop through each character in the statement starting from index 2 (j from 2 to len-1):
    - Check for Operand (Variable):
      - If the character is a lowercase letter (a to z), print the corresponding LOAD instruction and increment reg.
    - Check for Operator and Generate Instruction:
      - If j points to an operator (+, -, \*, /) and the preceding character is an =:
        - Depending on the operator, print the corresponding operation instruction (ADD, SUB, MUL, DIV) using the last two registers.
        - Print the STORE instruction to store the result in the left-hand side variable of the assignment.
    - Break the loop after processing the operator and generating instructions.

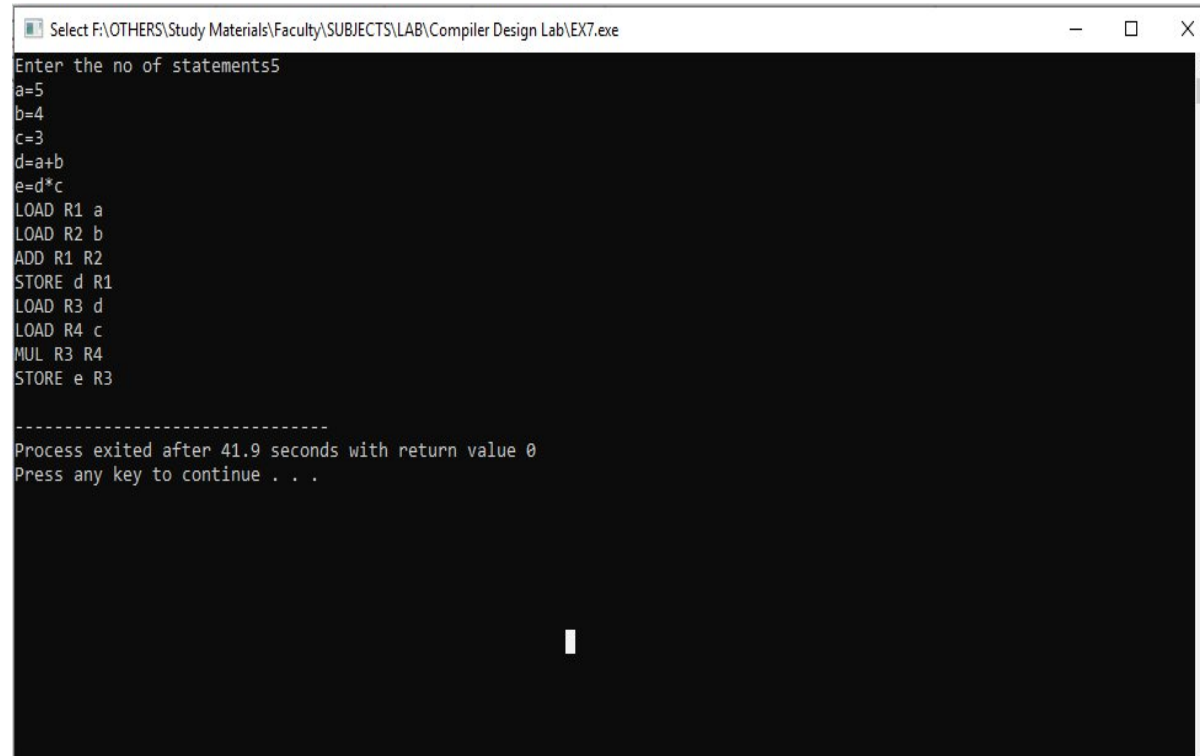
Step-5: Stop the program.

## **PROGRAM**

```
#include<stdio.h>
#include<string.h>
int main()
{
    char inp[100][100];
    int n,i,j,len;
    int reg = 1;
    printf("Enter the no of statements");
    scanf("%d",&n);
    for(i = 0; i < n; i++)
        scanf("%s",&inp[i]);
    for(i = 0; i < n; i++)
    {
        len = strlen(inp[i]);
        for(j=2; j < len; j++)
        {
            if(inp[i][j] >= 97 && inp[i][j] <= 122)
            {
                printf("LOAD R%d %c \n",reg++,inp[i][j]);
            }
            if(j == len-1 && inp[i][len-j] == '=')
            {
                j=3;
                if(inp[i][j] == '+')
                {
                    printf("ADD R%d R%d\n",reg-2,reg-1);
                    printf("STORE %c R%d\n",inp[i][0],reg-2);
                }
                else if(inp[i][j]=='-')
                {
                    printf("SUB R%d R%d\n",reg-2,reg-1);
                    printf("STORE %c R%d\n",inp[i][0],reg-2);
                }
                else if(inp[i][j]=='*')
                {
                    printf("MUL R%d R%d\n",reg-2,reg-1);
                    printf("STORE %c R%d\n",inp[i][0],reg-2);
                }
                else if(inp[i][j]=='/')
                {
                    printf("DIV R%d R%d\n",reg-2,reg-1);
                    printf("STORE %c R%d\n",inp[i][0],reg-2);
                }
            }
        }
    }
}
```

```
        break;
    }
}
return 0;
}
```

## OUTPUT



```
Select F:\OTHERS\Study Materials\Faculty\SUBJECTS\LAB\Compiler Design Lab\EX7.exe
Enter the no of statements5
a=5
b=4
c=3
d=a+b
e=d*c
LOAD R1 a
LOAD R2 b
ADD R1 R2
STORE d R1
LOAD R3 d
LOAD R4 c
MUL R3 R4
STORE e R3

-----
Process exited after 41.9 seconds with return value 0
Press any key to continue . . .
```

## **RESULT**

Thus, the C program for implementing backend of the compiler was executed successfully and the output was verified.