Arithmetic Expression Validator and Evaluator

Submitted by

SK.MD.Hussain Basha (RA2011026010070)
Murali Krishna (RA2011026010086)

Under the Guidance of

DR.J.Jeyasudha

Assistant Professor, Department of Computational Intelligence

In partial satisfaction of the requirements for the degree of

BACHELORS OF TECHNOLOGY
in
COMPUTER SCIENCE ENGINEERING

with specialization in Artificial Intelligence & Machine Learning

SCHOOL OF COMPUTING

COLLEGE OF ENGINEERING AND TECHNOLOGY

SRM INSTITUTE OF SCIENCE AND TECHNOLOGY

KATTANKULATHUR - 603203

May 2023

SRM INSTITUTION OF SCIENCE AND TECHNOLOGY
KATTANKULATHUR-603203

BONAFIDE CERTIFICATE

Certified that this Course Project Report titled **"Arithmetic Expression Validator and Evaluator"** is the bonafide work done by **SK.MD. Hussain Basha (RA2011026010070) , Murali Krishna (RA2011026010086)** who carried out under my supervision. Certified further, that to the best of my knowledge the work reported herein does not form part of any other work.

SIGNATURE
Faculty In-Charge
DR.J.Jeyasudha
Assistant Professor
Department of computational intelligence
SRM Institute of Science and Technology
Kattankulathur Campus, Chennai

HEAD OF THE DEPARTMENT
Dr. R Annie Uthra
Professor and Head ,
Department of Computational Intelligence,
SRM Institute of Science and Technology
Kattankulathur Campus, Chennai

Aim:-

The aim of the arithmetic expression validator and evaluator experiment is to develop and test a program that can accurately validate the syntax of arithmetic expressions and evaluate their numerical value. The goal is to create a software tool that can detect and report any syntax errors in an expression and compute the correct output for well-formed expressions. This experiment aims to improve the accuracy and efficiency of arithmetic calculations and reduce the likelihood of errors that can arise from manual calculations.

ABSTRACT:-

An arithmetic expression validator and evaluator is a software tool that helps to verify and calculate mathematical expressions entered by users. The tool checks the validity of the expression syntax, such as matching parentheses, correct operator usage, and operand order. It then evaluates the expression and produces the result. This tool can be useful in a wide range of applications, including education, finance, and computer programming. The validator and evaluator can be designed to handle a variety of mathematical operations, such as addition, subtraction, multiplication, and division, and more complex functions like logarithms, trigonometric functions, and exponents. With its ability to validate and evaluate arithmetic expressions, this tool can save time and prevent errors in mathematical computations quickly and accurately.

Problem Statement:

The problem addressed by the Arithmetic Expression Validator and Evaluator is the need for an efficient and accurate tool to check the validity and calculate the result of mathematical expressions entered by users. Often, users may make errors in syntax or operator usage when entering mathematical expressions, leading to incorrect results. Additionally, some expressions may be too complex for users to calculate manually, making it time-consuming to obtain the correct result.

The Validator and Evaluator tool aims to address these issues by providing a user-friendly and reliable solution that can quickly verify the validity and calculate the result of arithmetic expressions. This tool can be used in a wide range of applications, including education, finance, and computer programming, where accurate and efficient mathematical calculations are essential.

The Arithmetic Expression Validator and Evaluator aim to provide a solution that is both easy to use and accurate. By detecting errors in expression syntax and providing correct results, it can save users time and prevent mistakes that could lead to costly errors. It can handle a wide range of mathematical expressions and functions, including complex ones, making it a versatile tool for various applications. Additionally, the Validator and Evaluator can be designed to integrate with other software systems, allowing for seamless integration into existing

workflows. Overall, the problem addressed by the Arithmetic Expression Validator and Evaluator is the need for a reliable and efficient tool to handle mathematical calculations, which can improve accuracy, productivity, and efficiency in various fields.

Requirements:

LEX

The Lex tool receives at the input a set of user-defined patterns that it uses to scan the source code. Each time the source code matches one of the patterns a defined action is executed by Lex (one of the actions is that of returning the tokens).

Lex was originally developed as part of the UNIX operating system in the 1970s, and has since become a popular tool for building compilers, interpreters, and other programs that need to process input text. Lex is often used in conjunction with the yacc or bison tools, which generate parsers for the output of the lexical analyzer.

Lex is officially known as a "Lexical Analyzer". Its main job is to break up an input stream into more meaningful units, or tokens. For example, consider breaking a text file up into individual words.

More pragmatically, Lex is a tool for automatically generating a lexer (also known as a scanner).

To use Lex, you typically define a set of regular expressions that describe the different types of tokens you want to recognize in the input, and associate each regular expression with a corresponding action that should be taken when that token is recognized. Lex then generates a C or C++ program that implements a finite automaton to recognize the input text, and invokes the appropriate actions for each recognized token.

YACC

The Yacc tool receives the input of the user's grammar. Starting from this grammar it generates the C source code for the parser. Yacc invokes Lex to scan the source code and uses the tokens returned by Lex to build a syntax tree.

Yacc is often used in conjunction with the Lex tool, which generates lexical analyzers for the input text. Together, these tools provide a powerful and flexible framework for building compilers and other text processing tools.

YACC stands for Yet Another Compiler Compiler. Its GNU version is called Bison. YACC translates any grammar of a language into a parser for that language.

Grammars for YACC are described using a variant of the Backus Naur Form (BNF). A BNF grammar can be used to express context-free languages. By convention, a YACC file has the suffix .y.

Yacc generates a parser in file y.tab.c and includes file y.tab.h. Lex includes this file (y.tab.h) and uses the definitions for token values found in this file for the returned tokens.

To use Yacc, you typically define a grammar for the input text using a set of production rules, and associate each production rule with a corresponding action that should be taken when that rule is recognized. Yacc then generates a C or C++ program that implements a parser for the input text based on the specified grammar, and invokes the appropriate actions for each recognized production rule.

Based on the problem statement, some specific requirements for the Arithmetic Expression Validator and Evaluator:

1. Input: The tool should be able to accept input in the form of mathematical expressions entered by users.

2. Syntax Validation: The tool should be able to validate the syntax of the mathematical expressions to ensure that they are correctly formatted and contain no errors.

3. Operators and Operands: The tool should support a range of mathematical operators and operands, including addition, subtraction, multiplication, division, and more complex functions like logarithms, trigonometric functions, and exponents.

4. Order of Operations: The tool should follow the correct order of operations when evaluating mathematical expressions to ensure that the results are accurate.

5. Error Reporting: The tool should be able to report any errors in the expression syntax and provide suggestions for correction.

6. Evaluation: The tool should be able to evaluate mathematical expressions and produce accurate results.

7. Speed: The tool should be designed to perform evaluations quickly to provide users with results in a timely manner.

8. Customization: The tool should allow for customization of settings, such as the number of decimal places to display or the use of scientific notation.

9. Integration: The tool should be able to integrate with other software systems, such as spreadsheet software or programming languages, to allow for seamless data transfer and workflow integration.

10. User Interface: The tool should have a user-friendly interface that is easy to use and understand, including clear error messages and instructions for correcting syntax errors.

11. Security: The tool should be designed with security in mind, ensuring that user data is protected and secure.

Overall, these requirements should ensure that the Arithmetic Expression Validator and Evaluator is a reliable, accurate, and efficient tool for verifying and evaluating mathematical expressions.

WORK BREAKDOWN STRUCTURE

Work breakdown structure for developing an Arithmetic Expression Validator and Evaluator:

1. Requirements Gathering:

- Define the scope and objectives of the project.

- Identify and document the key features and functionalities required for the tool.

- Identify and document the user needs and use cases.

- Finalize the requirements and obtain approval from stakeholders.

2. Development:

- Develop the software components required for the tool, including input processing, validation, evaluation, and output generation.

- Implement the software components using the selected programming languages and frameworks.

- Write automated unit tests for each software component.

- Perform integration testing to ensure that all software components work together as expected.

3. Testing:

- Develop and execute a test plan that covers all key features and use cases.

- Test the tool for functionality, performance, usability, and security.

- Document and report all issues, bugs, and errors found during testing.

- Resolve all issues and retest the tool to ensure that it is bug-free.

4. Documentation:

- Develop user manuals, technical documentation, and other relevant documentation for the tool.

- Create help files and guides for users to understand how to use the tool.

- Develop internal documentation to assist in maintenance and support of the tool.

5. Deployment:

- Deploy the tool to the target platform, such as a web server or desktop application.

- Perform any necessary configuration or setup to ensure that the tool works as expected.

- Obtain approval from stakeholders to launch the tool.

6. Maintenance:

- Monitor the tool for issues, bugs, and errors.

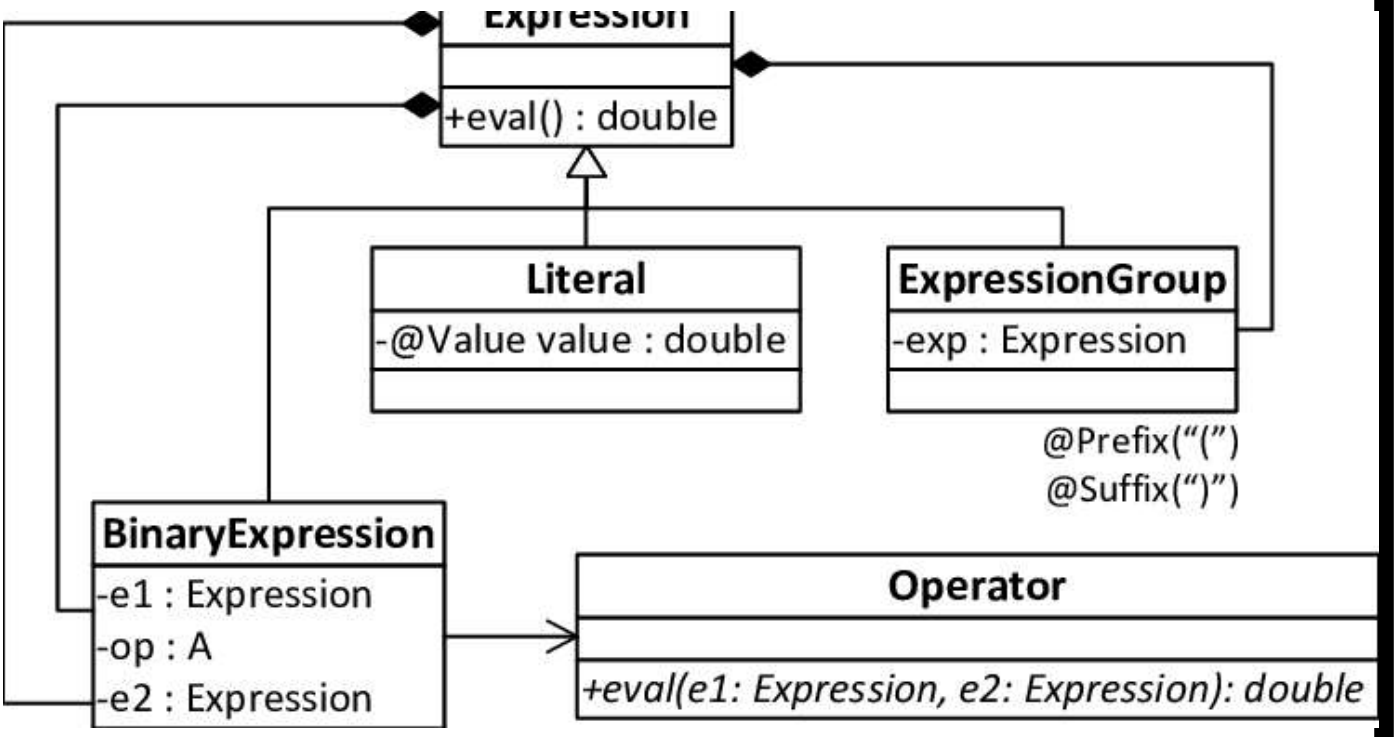
- Address and resolve any issues as they arise.

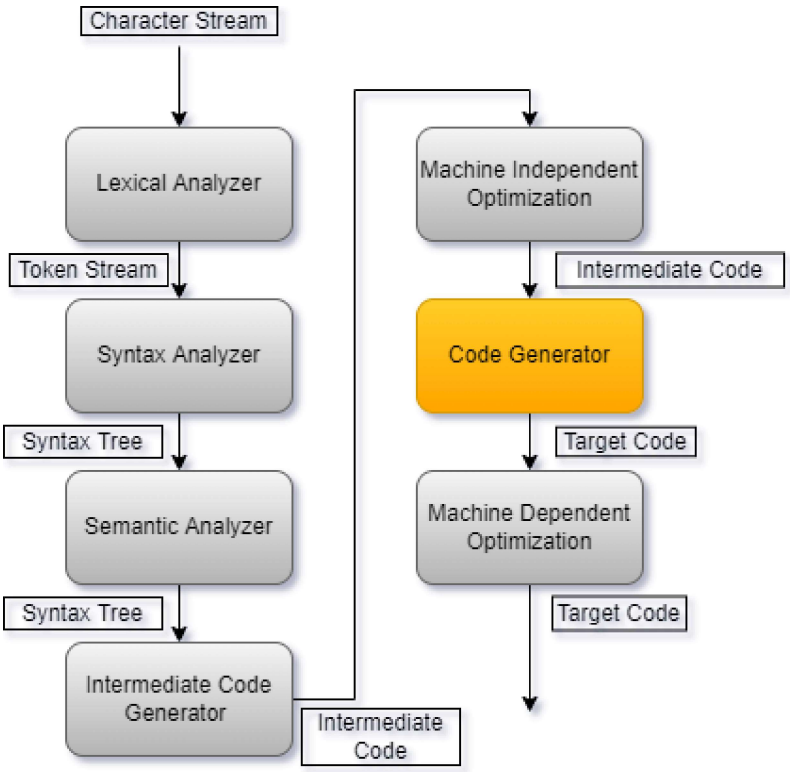
- Perform routine maintenance tasks, such as backups and updates.

- Respond to user feedback and make necessary improvements to the tool.

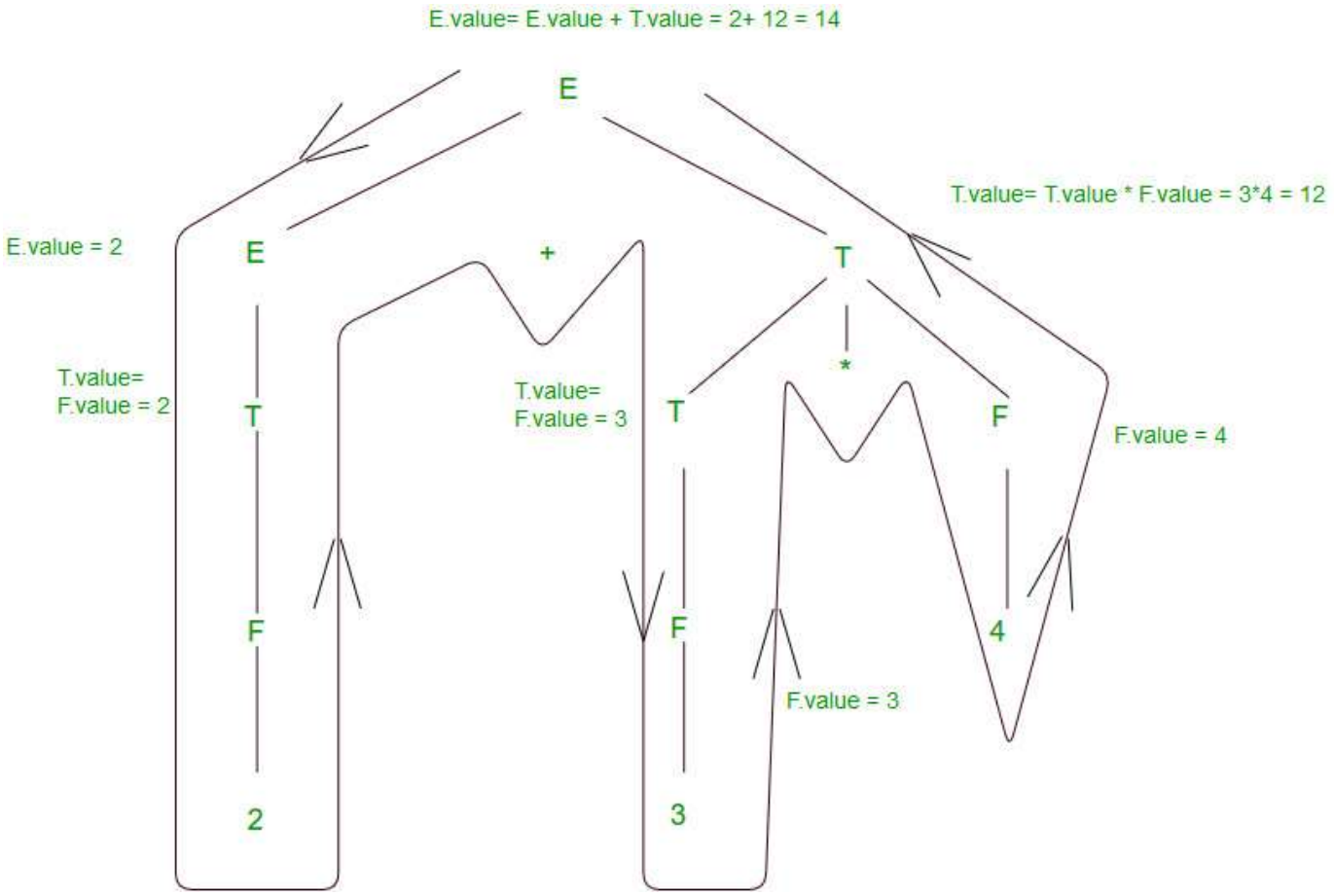
Overall, this work breakdown structure provides a comprehensive framework for developing an Arithmetic Expression Validator and Evaluator that meets the needs of its users and adheres to industry best practices.

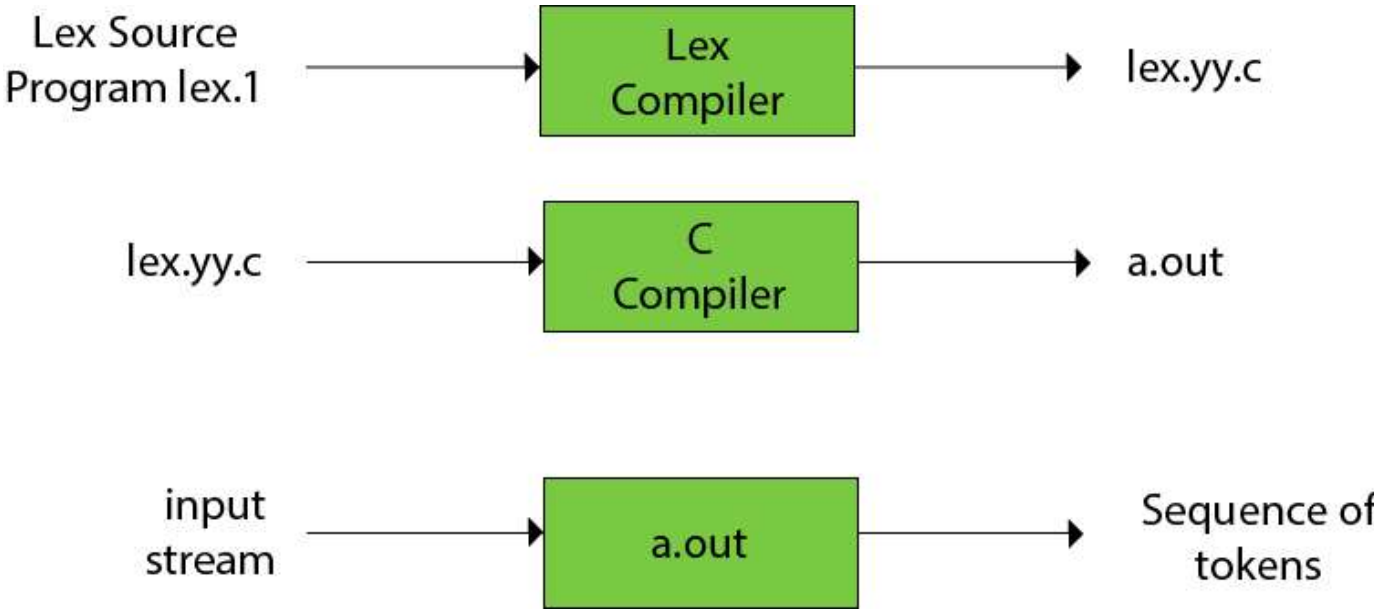
ARCHITECTURE DIAGRAM



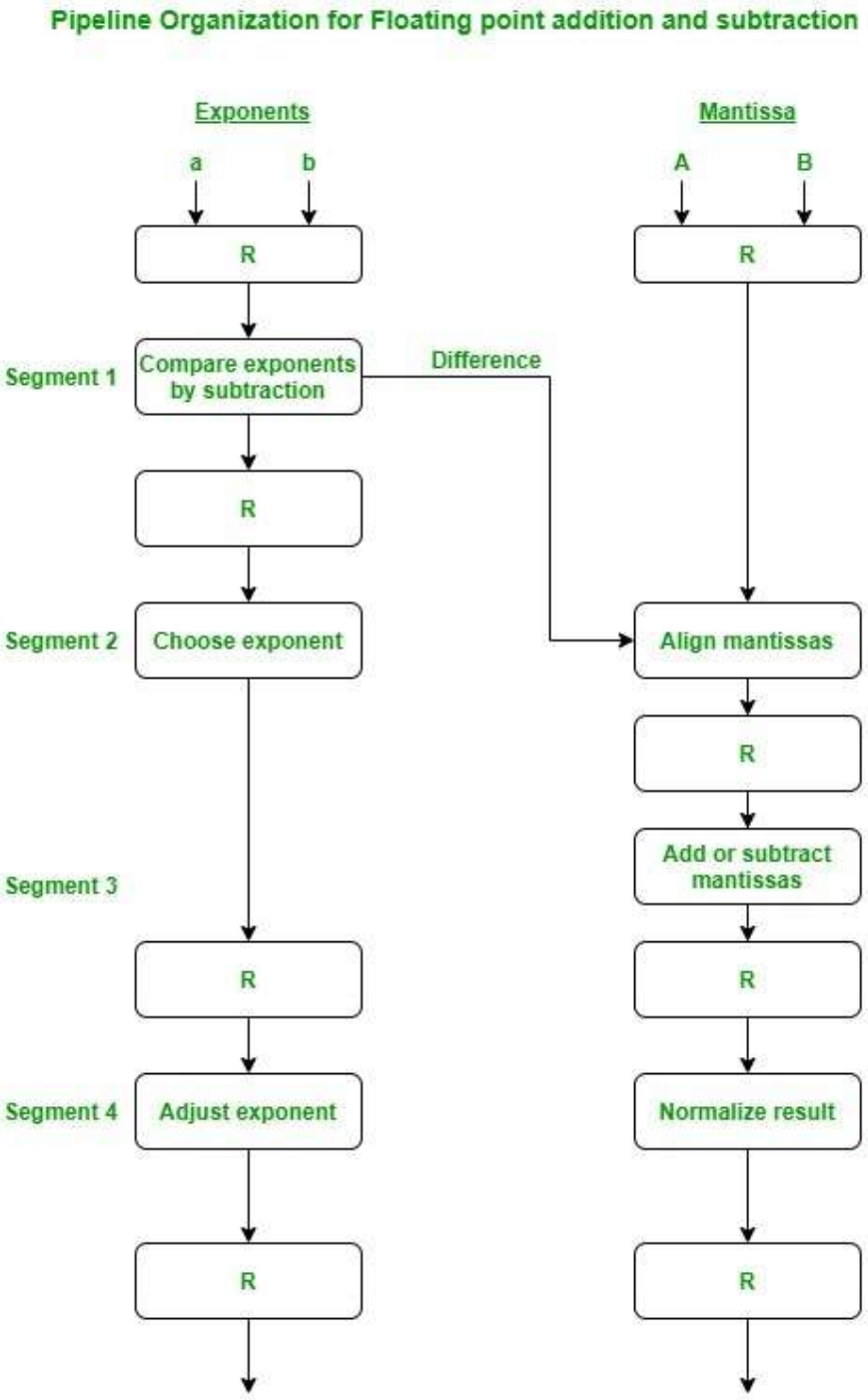


ARCHITECTURE DIAGRAM





ARCHITECTURE DIAGRAM



RISK ANALYSIS

Possible **Risk analysis** for developing an Arithmetic Expression Validator and Evaluator:

1. Technical risks:

- The complexity of the logic required for evaluating complex expressions may lead to performance issues or incorrect results.

- The implementation of the software may not meet the requirements and specifications, resulting in incorrect or unexpected behavior.

- There may be compatibility issues with different operating systems or web browsers.

2. Schedule risks:

- There may be delays in obtaining approval of requirements or design from stakeholders, leading to project schedule slippages.

- There may be scope creep or changes in requirements during development, leading to additional work and schedule slippages.

3. Resource risks:

- There may be a lack of skilled resources or expertise in the required programming languages or frameworks, leading to slower development or lower quality output.

- There may be insufficient budget or time allocated to the project, leading to compromises in quality or scope.

4. Security risks:

- The software may be vulnerable to security threats such as SQL injection, cross-site scripting, or unauthorized access.

- Sensitive data may be exposed or leaked, leading to legal or reputational issues.

5. User risks:

- The software may not meet the expectations of the users or provide the required functionality.

- Users may have difficulty using or understanding the software, leading to reduced adoption or negative feedback.

To mitigate these risks, the project team take several actions, such as conducting thorough testing and validation of the software, involving stakeholders at each stage of development, and implementing security best practices. The team can also allocate sufficient resources and budget to the project and adopt an agile approach to development, allowing for flexibility in responding to changes in requirements or scope.

Code:

expr.l:

```
%{
   #include"expr.tab.h"
   #include<math.h>
   extern int yylval;
%}
%%
[(]    {return yytext[0];}
[)]    {return yytext[0];}
[0-9]+ {yylval=atoi(yytext);return NUM;}
[+]     {return '+';}
[-]     {return '-';}
[*]     {return '*';}
[/]     {return '/';}
[\t]+   ;
" "     { }
[\n]    {return 0;}
.       {return yytext[0]; }
%%
```

expr.y:

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

%token NUM
%left '-''+'
%left '*''/'
%left '(')'

%%
start: exp {printf("\n The Value of the arithemtic expression is %d\n",$$);exit(0);}

exp:exp'+'exp {$$=$1+$3;}
   |exp'-'exp {$$=$1-$3;}
   |exp'*'exp {$$=$1*$3;}
   |exp'/'exp  {

            if($3!=0){ $$ = $1/$3; }
            else
```

```
                {
                    printf("Error: divide by Zero\n");
                    yyerror();
                    exit(0);
                }
            }

    |'('exp')' {$$=$2;}
    |NUM {$$=$1;}
    ;
%%
int main()
{
    printf("Enter the Expression:\n");
    yyparse();
        return 0;
}
void yywrap(){}
void yyerror()
{
 printf("The Given airthemetic expression is INVALID\n");
}
```

Execution:

C:\Users\abhir\OneDrive\Desktop\Arithmetic-expression-validator-and-evaluator>flex expr.l

On running the flex expr.l it creates lex.yy.c file

C:\Users\abhir\OneDrive\Desktop\Arithmetic-expression-validator-and-evaluator>biso n -d expr.y

And after running bison -d expr.y it creates expr.tab.c expr.tab.h files

C:\Users\abhir\OneDrive\Desktop\Arithmetic-expression-validator-and-evaluator>gcc lex.yy.c expr.tab.c -o hello.exe

After running the above command the input output operations will be calculated by the hello.exe file

C:\Users\abhir\OneDrive\Desktop\Arithmetic-expression-validator-and-evaluator>hello .exe

On running this command it prompts for an input and later it generates the output.

Output:

```
Enter the Expression:
5-2*6+7

 The Value of the arithemtic expression is 0
```

```
Enter the Expression:
5-6/0+7
Error: divide by Zero
The Given airthemetic expression is INVALID
```

```
Enter the Expression:
a-7+5
The Given airthemetic expression is INVALID
```

Result:

Hence, the implementation of the arithmetic expression validator and evaluator is successfully made and executed.