<Arithmetic Expression Evaluator>

Software Requirements Specifications

Version <1.0>

Revision History

| **Date** | **Version** | **Description** | **Author** |
| --- | --- | --- | --- |
| 10/15/23 | 1.0 | Creation of SRS document | Zach Sevart, Asa Maker, Lingfeng Li |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

Table of Contents

1. Introduction 4

1.1 Purpose 4

1.2 Scope 4

1.3 Definitions, Acronyms, and Abbreviations 4

1.4 References 4

1.5 Overview 4

2. Overall Description 5

2.1 Product perspective 5

2.1.1 System Interfaces 5

2.1.2 User Interfaces 5

2.1.3 Hardware Interfaces 5

2.1.4 Software Interfaces 5

2.1.5 Communication Interfaces 5

2.1.6 Memory Constraints 5

2.1.7 Operations 5

2.2 Product functions 5

2.3 User characteristics 5

2.4 Constraints 5

2.5 Assumptions and dependencies 5

2.6 Requirements subsets 5

3. Specific Requirements 5

3.1 Functionality 5

3.1.1 <Functional Requirement One> 6

3.2 Use-Case Specifications 6

3.3 Supplementary Requirements 6

4. Classification of Functional Requirements 6

5. Appendices [6](#_qsh70q)

Software Requirements Specifications

# Introduction

Welcome to the Software Requirements Specification (SRS) for our Arithmetic Expression Evaluator. This C++ program is designed to parse and evaluate arithmetic expressions, taking into account operator precedence rules and parentheses.

The objective of this document is to outline the requirements, key features, software interfaces, and potential future changes for the software. Its content serves as a comprehensive guide for developers, project managers, and testers who will be responsible for ensuring that the software meets the outlined specifications.

This introduction sets the stage for a detailed walkthrough of the meticulous design and thoughtfully engineered components that form the blueprint for the Arithmetic Expression Evaluator.

## Purpose

The purpose of this SRS is to provide a detailed overview of the specifications and requirements for the Arithmetic Expression Evaluator software. The software is required to parse and evaluate arithmetic expressions containing the operators +, -, \*, /, %, and ^, as well as numeric constants.

## Scope

The software is a C++ program that evaluates arithmetic expressions input by the user and provides the calculated result, taking into account parentheses and operator precedence.

## Definitions, Acronyms, and Abbreviations

- SRS: Software Requirements Specification

- PEMDAS: Parentheses, Exponents, Multiplication and Division, Addition and Subtraction, to determine the order of operations.

- User - Individuals utilizing the Arithmetic Expression Evaluator application

## References

IEEE Std 830-1998, IEEE Recommended Practice for Software Requirements Specifications.

00-Project-Description, Project Description

## Overview

The project involves building a versatile arithmetic expression evaluator using C++. The program will take an arithmetic expression as input, parse it, and calculate the result according to the order of operations (PEMDAS).

# Overall Description

## Product perspective

The Arithmetic Expression Evaluator is an independent software application providing arithmetic calculation functionality.

### System Interfaces

Since this project is a command-line based application written in C++, it interacts directly with the system's terminal interface. Users will provide input and receive output through the terminal.

### User Interfaces

As a command-line application, user interactions occur entirely within a command-line interface. The user inputs arithmetic expressions, and the parsed results will be displayed in the command line.

### Hardware Interfaces

As a software application, no specific hardware interfaces are necessary. The application should run on any hardware capable of running a standard C++ environment.

### Software Interfaces

The software interfaces with a C++ compiler, such as GCC or Clang, as well as the system's command-line interface. It may make use of standard C++ libraries.

### Communication Interfaces

Being a standalone application, there's no network communication involved. The software doesn't require communication with any external or third-party applications, services, or databases.

### Memory Constraints

While exact memory usage will depend on the complexity of the arithmetic expressions provided by the user, we anticipate minimal memory usage. The primary memory requirement will be the need to store parsed expressions and intermediate results in the program's stack or heap space.

### Operations

User supplies the arithmetic expression at the command line, the system then parses the expression considering the PEMDAS rule and parenthetical expressions, evaluates the expression, handles any input, math, and syntax errors that might arise, and finally outputs the answer in the terminal. The software can be restarted for new calculations.

## Product functions

- Parse arithmetic expressions inputted by the user.

- Evaluate the parsed expression according to the order of operations.

## User characteristics

End-users are assumed to be familiar with basic arithmetic operations, operator precedence rules (PEMDAS), and the operation of command-line interfaces.

## Constraints

The software application must be developed in C++.

## Assumptions and dependencies

- Assumptions:

1. Users are acquainted with basic mathematical operations and are comfortable using command-line interfaces.

2. The program will run on an environment that can compile and run C++ code.

- Dependencies:

1. The correct operation of the program will depend on the accurate functioning of a C++ compiler and runtime environment.

2. The accurate parsing and premium functionality of the program will rely on the compatibility of the utilized C++ libraries.

## Requirements subsets

* Subset 1: Input

The application will acccept user input from a command-line interface. User will be able to input arithmetic expressions in the form of numbers, the prescribed mathematical operators (+, -, \*, /, %, ^), and parentheses.

* Subset 2: Expression Parsing

The program will be able to decode the expression entered by the user, taking note of the operator precedence and parentheses.

* Subset 3: Expression Evaluation

The software will perform the necessary calculations as per the PEMDAS rule after parsing the expression, and evaluate the result of the arithmetic expression.

* Subset 4: Result Output

The evaluated result will be displayed back to the user through the command-line interface.

* Subset 5: Error Handling

The software should be able to manage error scenarios, such as division by zero or incorrect input expressions, and provide clear and informative indicators when such errors occur.

* Subset 6: Unit Tests

To validate the correctness of the expression evaluator, unit tests will be developed to verify the function and robustness of the application.

# Specific Requirements

## Functionality

a. Expression Parsing

The system should be able to parse arithmetic expressions containing operators (+, -, \*, /, %, ^) and numeric constants.

The system should parse expressions considering operator precedence and parentheses.

b. Error Handling

The system should identify and handle invalid arithmetic expressions, providing appropriate error messages.

c. User Interface

The system should provide a user-friendly command-line interface that allows users to input arithmetic expressions and displays results.

### <Functional Requirement One>

1. Objective: Develop an Arithmetic Expression Evaluator in C++.
2. Description:The system shall parse and evaluate arithmetic expressions, considering operators and parentheses. It should handle expressions with operators (+, -, \*, /, %, ^) and numeric constants. Provide clear and informative error messages for invalid input.
3. Criteria for Success:Accurate parsing and evaluation of arithmetic expressions. Clear and informative error messages for invalid input.

## Use-Case Specifications

Actors: User

Preconditions: The user has entered an arithmetic expression.

Main Success Scenario:

The user inputs an arithmetic expression.

The system parses and calculates the value of the expression.

The system displays the calculated result.

Extensions:

2a. If the expression is invalid:

The system displays an error message.

Return to step 1.

## Supplementary Requirements

a. Performance Requirements

The system should accurately calculate the value of expressions within a few seconds.

b. Usability Requirements

The system should provide clear user guides and error messages to assist users in utilizing the expression evaluator correctly.

c. Security Requirements

If applicable, the system should safeguard the security and privacy of user data and calculation results.

# Classification of Functional Requirements

| **Functionality** | **Type** |
| --- | --- |
| Tokenization: Implement a function to tokenize the input expression.  Data Structure: Create a data structure, such as a stack or a tree, to represent the expression's structure | Expression Parsing |
| Addition: Support for the "+" operator.  Subtraction: Support for the "-" operator.  Multiplication: Support for the "\*" operator.  Division: Support for the "/" operator, ensuring no division by zero.  Modulo: Support for the "%" operator.  Exponentiation: Support for the "^" operator and potentially "\*\*" in the future. | Operator |
| Identification: Develop a mechanism to identify expressions within parentheses.  Evaluation: Ensure correct evaluation of expressions within parentheses. | Parenthesis Handling |
| Integer Recognition: Recognize integer numeric constants in the input.  Floating-Point Recognition: [Potential Future Requirement] Recognize floating-point numeric constants. | Numeric Constants |
| Input: Create a user-friendly command-line interface that allows users to enter expressions.  Output: Display the calculated results in a clear and understandable format. | User Interface |
| Invalid Expressions: Implement robust error handling for scenarios like division by zero or invalid expressions.  User Feedback: Ensure that your program provides clear and informative error messages for invalid input. | Error Handling |
| User Manual: Provide a user manual or README file explaining how to use the program, including examples.  Code Documentation: Include comments and documentation to explain the logic and functionality of the program. | Documentation and User Manual Quality |
| Unit Tests: Develop unit tests to verify the correctness of the expression evaluator.  Testing for Various Expression Scenarios: Ensure testing for valid and invalid expressions, considering various operators and parentheses usage. | Testing |
| Compliance with Project Guidelines: Ensure all code and documentation adhere to project guidelines.  Adaptability: Be prepared for future requirements, such as supporting additional operators or input types. | Compliance and Future Adaptability |
| Structure: Use object-oriented programming principles to structure your code.  Readability: Ensure code is readable and maintainable. | Code Quality |
| Evaluation Speed: Ensure expressions are evaluated and results are displayed in a timely manner. | Performance |
| [Potential Future Requirement] If user data is to be stored or transmitted, ensure secure handling and storage of this data. | Security |
| User Feedback: Ensure clear and informative feedback is provided to the user, especially in error scenarios. | Usability |
| Correctness: Ensure the evaluator provides accurate results for all valid expressions.  Consistency: Ensure consistent behavior across various types of expressions and user inputs. | Reliability |
| Explore and implement additional features or optimizations beyond the specified requirements to enhance the project. | Additional Features or Optimizations |

# Appendices

a. Project Overview

Objective: Develop a C++ program to parse and evaluate arithmetic expressions, considering operators (+, -, \*, /, %, ^), numeric constants, and parentheses.

Key Features: Expression parsing, operator support, parenthesis handling, and numeric constant recognition.

Tasks: Include expression parsing, defining operator precedence, handling parentheses, recognizing numeric constants, UI development, and error handling.

b. Valid and Invalid Expressions

Valid Expressions: Examples and explanations of expressions that adhere to mathematical and syntactical rules.

Invalid Expressions: Examples and explanations of expressions that violate mathematical or syntactical rules, such as unmatched parentheses, division by zero, incorrect operator usage, or missing operands.

c. Project Tasks and Key Features

Expression Parsing: Implementing tokenization and utilizing data structures for expression evaluation.

Operator Precedence: Defining and implementing logic to adhere to PEMDAS rules.

Parenthesis Handling: Developing mechanisms to identify and evaluate expressions within parentheses.

Numeric Constants: Recognizing and calculating with numeric constants.

User Interface: Developing a user-friendly command-line interface.

Error Handling: Implementing mechanisms to manage scenarios like division by zero or invalid expressions.

d. Grading Criteria

Correctness: Evaluation of expression (60 points).

Robustness and Error Handling: Managing invalid expressions and errors (20 points).

Code Quality: Including structure and readability (20 points).

Documentation and User Manual Quality: Providing clear and comprehensive guides (20 points).

e. Deliverables

Software Engineering Artifacts: Including a project management plan, requirements document, design document, and test cases.

C++ Program: A well-documented program that evaluates arithmetic expressions.

User Manual: A guide explaining the program usage, including examples.

f. Note

Encouragement to explore additional features or optimizations beyond the specified requirements to enhance the project.

g. Examples and Explanations

Detailed examples of valid and invalid expressions, providing results and explanations for each scenario.