EECS 348

Arithmetic Expression Evaluator
Software Requirements Specifications

Version 1.2

Revision History

Date	Version	Description	Author
10/1/2023	1.0	Create document	Marcos Lepage
10/11/2023	1.1	Update document	Marcos Lepage
10/15/2023	1.2	Finalize document	Marcos Lepage

Table of Contents

Software Requirements Specifications

1. Introduction

1.1 Purpose

The purpose of this Software Requirements Specification (SRS) is to express the meaning of the functional and non-functional requirements of an Arithmetic Expression Evaluator project. This document serves as a guide for the development team to design, implement, and test the software.

1.2 Scope

This SRS applies to the development of a C++ program capable of parsing and evaluating arithmetic expressions containing operators +, -, *, /, %, and ^. The software must also support numeric constants and handle expressions within parentheses.

1.3 Definitions, Acronyms, and Abbreviations

- CLI: Command Line Interface
- GUI: Guided User Interface
- ISA: Instruction Set Architecture
- SRS: Software Requirements Specification
- PEMDAS: Parentheses, Exponents, Multiplication and Division (from left to right), Addition and Subtraction (from left to right).
- UI: User Interface
- UX: User Experience

1.4 References

No external references are required for this document.

1.5 Overview

This document is organized into sections that describe the purpose, scope, and requirements of the Arithmetic Expression Evaluator project. The "Overall Description" section provides a high-level overview of the project, and the "Specific Requirements" section outlines the functional and non-functional requirements in detail.

2. Overall Description

2.1 Product perspective

2.1.1 System Interfaces – N/A

2.1.2 User Interfaces

The intended user interface for this program is a CLI (command line interface). CLIs significantly reduce development time by being contained entirely within the terminal (often the same terminal that spawned the process), allowing the usage of standard input and output interfaces. For technically inclined individuals, CLIs are a common (if not expected) means of communicating with a system. However, for more casual users an aesthetic GUI is more desirable. For this reason, the development team retains the option to create a GUI for the project, should the time constraints allow for it.

2.1.3 Hardware Interfaces – N/A

2.1.4 Software Interfaces – N/A

2.1.5 Communication Interfaces – N/A

2.1.6 Memory Constraints – N/A

2.1.7 Operations

The software will perform operations related to parsing, evaluating arithmetic expressions, and handling error conditions.

2.2 Product functions

The primary function of the software is to parse and evaluate arithmetic expressions according to the order of operations (PEMDAS).

2.3 User characteristics – N/A

2.4 Constraints

The software must be developed in C++ and follow object-oriented programming principles.

2.5 Assumptions and Dependencies

The software assumes that input expressions will be valid and adhere to arithmetic rules. Dependencies include the C++ development environment and standard libraries.

2.6 Requirements subsets – N/A

3. Specific Requirements

3.1 Functionality

3.1.1 Correctly Evaluate User-Provided Arithmetic Expressions:

For every and all arithmetic expressions entered into the program by any means supported, the system shall evaluate said expression and return a numerical result which represents this evaluation. The evaluation value must always be correct (following the rules of both algebra and arithmetic, as well as those rules specified in other functional requirements).

3.1.2 Provide UI for User Expression Input and Evaluation Output:

The system shall provide a user interface (at the minimum a CLI) to enable casual users to interact with the system. This UI will, first, provide a means for users to enter expressions into the program. Second, provide a method of submitting these expressions for evaluation. And third, provide a means of immediately viewing the evaluation of the submitted expression.

3.1.3 Support the "+" Operator for Addition:

The system shall designate the "+" symbol as the addition operator. If the "+" symbol has both a left- and right-hand operand, then it will be recognized as a binary addition operator (adding the two operands), with an operator precedence of **1**. If the "+" symbol only has a right-hand operand, then it will be recognized as a unary addition operator (which effectively does nothing), with an operator precedence of **4**. However, having a left-hand operand without a right-hand operand is invalid and will yield an error.

3.1.4 Support the "-" Operator for Subtraction:

The system shall designate the "**-**" symbol as the subtraction operator. If the "**-**" symbol has both a left and right-hand operand, then it will be recognized as a binary subtraction operator (subtracting the right-hand operand from the left-hand operand), with an operator precedence of **1**. If the "**-**" symbol only has a right-hand operand, then it will be recognized as a unary subtraction operator (which negates its right-hand operand), with an operator precedence of **4**. However, having a left-hand operand without a right-hand operand is invalid and will yield an error.

3.1.5 Support the "" Operator for Multiplication:*

The system shall designate the "*" symbol as the multiplication operator. Whenever evaluating an arithmetic expression, the system will recognize and evaluate multiplication operations, giving them a precedence of **2**. Multiplication operations must have two operands, failure to compile will yield an error.

3.1.6 Support the "/" Operator for Division:

The system shall designate the "/" symbol as the division operator. Whenever evaluating an arithmetic expression, the system will recognize and evaluate division operations, giving them a precedence of **2**. Division operations must have two operands, failure to compile will yield an error. The left-hand operand will be the dividend, and the right-hand operand will be the divisor.

3.1.7 Support the "%" Operator for Modulo Division:

The system shall designate the "%" symbol as the modulo operator. Whenever evaluating an arithmetic expression, the system will recognize and evaluate modulo operations, giving them a precedence of **2**. Modulo operations must have two operands, failure to compile will yield an error. The left-hand operand will be the dividend, and the right-hand operand will be the divisor. Modulo returns the remainder.

3.1.8 Support the "^" Operator for Exponentiation:

The system shall designate the "^" symbol as the exponentiation operator. Whenever evaluating an arithmetic expression, the system will recognize and evaluate exponentiation operations, giving them a precedence of **3**. Multiplication operations must have two operands, failure to compile will yield an error. The left-hand operand will be the base, and the right-hand operand will be the exponent.

3.1.9 Support for Composite Arithmetic Expressions of Arbitrary Length:

The system will successfully evaluate arithmetic expressions regardless of if they are composite (i.e., expressions which consist of multiple operators). There shall be no upper bound (besides memory consumption) which limits the length of a composite function. The system shall be abstract enough to handle composite expressions including (but not limited to) those which are: heavily sequentially chained, or deeply nested.

3.1.10 Parenthesis Handling for Expression Grouping.

The system shall designate "(", and ")" (parenthesis) as the grouping operators. Any expression (i.e., the single operand) present within parentheses will have evaluation priority. Specifically, the grouping operators will have an operator precedence of **5**, which is the highest precedence of any operator. For every open parenthesis, there must be a complementary closing parathesis (and vice versa). Failure to comply with this rule will result in an error.

3.1.11 Integer Constants/Literals Recognition.

The system shall be capable of recognizing numeric literals (i.e., constants) within arithmetic expressions. The system will at least provide support for integers yet may be later expanded to support all real numbers (see 3.1.16). Operators will perform their operations on either these numeric literals, or expressions consisting of numeric literals and operators.

3.1.12 Respect for Operator Precedence During Evaluation.

The system shall recognize operator precedence during expression evaluation. Operators with lower precedence will be evaluated after those with higher precedence. Operators with the same precedence will be evaluated in the order in which they appear within the expression, left-to-right. Simply put, the system will obey the precedence laws of arithmetic & algebra (commonly referred to as PEMDAS).

3.1.13 Error Handling for Invalid and Otherwise Crash-Inducing Expressions:

It is recognized that not all arithmetic expressions are evaluable. Thus, the system shall possess handling for invalid or otherwise crash-inducing expressions. This includes, at minimum, dodging crashes by refusing to evaluate expressions which are not evaluable. Moreover, when an error is encountered, the system should produce a message that explains the *general nature* of the error to the user (in hopes that the user may be able to identify and fix their error). Errors that prevent the evaluation of an expression include (but are not limited to): divide by zero, unclosed grouping operator, dangling closed grouping operator, invalid numeric literal, invalid/unrecognized operator.

3.1.14 Incorporate Brackets for Grouping Arithmetic Expressions (in Addition to Parentheses):

The system shall (desirably) incorporate brackets as a subsequent grouping operator. Identically to parenthesis, brackets increase the evaluation priority of the expression they contain and themselves possess the maximum operator precedence of **5**. Separate from parathesis, brackets can only be closed by other brackets (and vice versa). The implementation of brackets helps the user disambiguate long expressions.

*3.1.15 Designate the "**" Operator for Exponential Operations:*

The system shall (desirably) designate the "**" symbol, in addition to the previously described "^" symbol (see 3.1.8), as the exponentiation operator. The "**" symbol will possess parsing precedence over the "*" multiplication symbol, to prevent expression parsing errors. The functionality of "**" will be identical to that of the "^" symbol, see 3.1.8.

3.1.16 Designate the Use of Floating-Point (Real) Numbers:

The system shall (desirably) recognize floating-point numeric literals (i.e., real numbers) as they appear within expressions. Hence reserving the "." symbol for floating-point literals. Floating-point numbers shall exhibit all the same behaviors as integer literals (described in 3.1.11). All operators defined earlier within the specification shall be capable of operating on floating-point values. It is permissible for real numbers to be subject to an arbitrary (but not excessively lossy) precision limitation (due to hardware limitations). Thus, irrational numbers and numbers with repeating decimal values are allowed to be truncated: both in their internal representations and in their output form.

3.1.17 Provide Debugging Messages to Help the User Identify Errors in their Expressions:

The system shall (optionally) provide thorough debugging messages upon the submission of an invalid expression. It should be noted that this functional requirement is an extension of the functionality described in 3.1.13. The functional requirement 3.1.13 stated that the system must provide *general* feedback in the case of an error. However, this requirement specifies that the feedback on those same errors must be specific. Similar to a compiler, the output messages must directly identify the character position at which the error occurred. Additionally, the messages should provide sensible feedback on how to repair the error. Finally, the message should include the general explanation provided by 3.1.13.

3.2 Use-Case Specifications

Arithmetic Expression Evaluation Use Case:

• Initialize Program: The user executes program binaries from their terminal.
• Enter Expression: The user types their arithmetic expression into the CLI prompt.
• Evaluate Expression: The user hits "enter" on their keyboard, submitting the expression.
• View Evaluation: The system displays the evaluated result for the user to view in the CLI.
• Repeat Use Case: Prompt the user for another expression, looping back to "Enter Expression".

Arithmetic Expression Evaluation Alternate Flow:

Handle user enters "exit" into the CLI prompt.

At the "Evaluate Expression" step of the Arithmetic Expression Evaluation use case, if the user input was the string "exit",

• Exit: Immediately exit program and end use case.

Handle user enters and invalid / unevaluatable expression into the CLI prompt.

At the "Evaluate Expression" step of the Arithmetic Expression Evaluation use case, if the program fails (or will fail to) evaluate the provided expression,

- Display Error: Display the error that invalidated the user input.
- Repeat Use Case: Prompt the user for another expression, looping back to "Enter Expression".

3.3 Supplementary Requirements

Development Constraints:

- The software must be implemented in the C++ programming language.
- The software must be structured using object-orientated programming principles.
- The software must possess unit tests to evaluate the functionality of the software during development.
- The program must be executable on common consumer ISAs (e.g., x86, ARM) as well as operating systems (e.g., Windows, MacOS, Linux).

Non-functional requirements:

- The software must be performant, evaluating common arithmetic expressions in under a millisecond.
- The software must not include any memory leaks, nor allocate unwarranted quantities of memory.
- The software must be stable, never crashing regardless of user input.
- The software must be written in a readable and maintainable way (i.e., follow programming guidelines).
- The software must be scalable, which includes using standard and intuitive interfaces for all software components including those only exposed internally.
- The software may optionally (yet desirably) have an intuitive and aesthetically pleasing user interface.

4. Classification of Functional Requirements

Functionality	Type
Correctly evaluate user-provided arithmetic expressions.	Essential
Provide UI for user expression input and evaluation output.	Essential
Support the "+" operator for addition.	Essential
Support the "-" operator for subtraction.	Essential
Support the "*" operator for multiplication.	Essential
Support the "/" operator for division.	Essential
Support the "%" operator for modulo division.	Essential
Support the "^" operator for exponentiation.	Essential
Support for composite arithmetic expressions of arbitrary length.	Essential
Parenthesis handling for expression grouping.	Essential
Integer constants/literals recognition.	Essential
Respect for operator precedence during evaluation.	Essential
Error handling for invalid and otherwise crash-inducing expressions (e.g. divide by 0)	Essential
Incorporate [] for grouping arithmetic expressions in addition to parentheses.	Desirable
Designate the "**" operator for exponential operations.	Desirable
Designate the use of floating-point (real) numbers.	Desirable
Provide debugging messages to help the user identify errors in their expressions.	Optional

5. Appendices

No appendices are included in this document.