Arithmetic Expression Evaluator

Version <1.7>

Revision History

|  |  |  |  |
| --- | --- | --- | --- |
| **Date** | **Version** | **Description** | **Author** |
| 10/02/2023 | <1.0> | Update Introduction | Elizabeth Channel |
| 10/12/2023 | <1.1> | Add Functional Requirements | Chris Cooper |
| 10/13/2023 | <1.2> | Update Overall Description (2.2-2.6) | Aryan Kevat |
| 10/14/2023 | <1.3> | Added Glossary (appendices) | Vivian Lara |
| 10/14/2023 | <1.4> | Update References | Elizabeth Channel |
| 10/14/2023 | <1.5> | Added Functional Requirement Classifications | Ashley Vierling |
| 10/14/2023 | <1.6> | Added Use Case Specifications | Chris Cooper |
| 10/14/2023 | <1.7> | Update Overall Description (2.1) | Jeff Burns |

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 6

3. Specific Requirements 6

3.1 Functionality 6

3.1.1 Binary Operators 6

3.1.2 Unary Operators 7

3.1.3 Parentheses 7

3.1.4 Operator Precedence 7

3.1.5 Errors 7

3.1.6 User Input 8

3.1.7 System Output 8

3.1.8 User Interface 9

3.2 Use-Case Specifications 9

3.2.1 System Class Diagram 9

3.2.2 Main Use Case 9

3.2.3 Basic Flow 10

3.2.4 Alternate Flows 10

3.3 Supplementary Requirements 11

3.3.1 Programming Language 11

3.3.2 Design Methodology 11

4. Classification of Functional Requirements 11

5. Appendices 12

5.1 Glossary 12

# Introduction

The following document will capture the complete software requirements for the system. It will be referred to throughout the *Arithmetic Expression Evaluator* project.

## Purpose

The purpose of the Software Requirements Specifications (SRS) document is to detail the external needs of the application. It describes all nonfunctional requirements along with interfaces, constraints, and functionality. It will be referred to throughout the project to ensure organization.

## Scope

This SRS applies to the *Arithmetic Expression Evaluator* application. It is associated with the Use-Case model for the entire system and its functions. The organization of software requirements and functionality revolve around this document.

## Definitions, Acronyms, and Abbreviations

See *Glossary* in Appendices

See *Project Glossary* in Annexes of *Software Development Plan*

## References

* Software Development Plan – 9/22/2023, CAVJAC Group

## Overview

The rest of this document will capture the complete software requirements for the system. It contains many aspects of the project such as interfaces and functionality. It is organized by specification, starting general is an overview of the project and ending with specific requirements.

# Overall Description

This is a general description of the interactions and constraints of the program. Further details are provided in section 3.

## Product perspective

### System Interfaces

Connections between components will be performed using https to GitHub repository, or secure shell to cycle servers at KU, if not connected to LAN.

### User Interfaces

The program will utilize the command line interface (CLI), using a standard US QWERTY keyboard for input.

### Hardware Interfaces

Input device able to perform the function of the Standard US QWERTY keyboard.

### Software Interfaces

The program will use the Linux operating system.

### Communication Interfaces

Connections to the program will be performed using https to GitHub repository, or secure shell to cycle servers at KU, if not connected to LAN.

### Memory Constraints

There are no memory constraints currently.

### Operations

The calculator will perform the following operations: Addition, Subtraction, Multiplication, Division, Modulo, and Exponentiation in standard operating precedence.

## Product functions

The functions of Arithmetic Expression Evaluator will include the ability to parse user input as a mathematical expression. The parser will recognize common mathematical symbols and operations and follow commonly used operator precedence. The output shall contain either the simplified form, or solution, of the mathematical expression, or produce an error message corresponding to the type of issue found.

## User characteristics

The product should be able to be used by anyone with an understanding of the operators and properties of mathematics that they are inputting into the program. Specific knowledge outside of simple mathematical operations is not required. The user is assumed to know how to type in the expression using a standard keyboard.

## Constraints

The program will be developed in C++ and may use any necessary libraries in order to function. The program must not crash under any circumstances, i.e. erroneous user input should produce an error message rather than stopping the program. The program will not require any special input method or characters outside of those found on a standard US QWERTY keyboard. The program will be able to run on the EECS Cycles servers provided by the KU School of Engineering.

## Assumptions and dependencies

The major assumption made for the project is that the user understands the behavior and rules of the expressions that they input. While exact dependencies of the project are not currently known, the product will need a working C++ compiler and will use parts of the C++ standard library at compile time. If using a provided binary release, the user is assumed to be using the target platform of the binary and have any runtime dependencies installed. The target architecture, compilation dependencies, and runtime dependencies will be specified in the product documentation.

## Requirements subsets

The specific requirements of the product will be divided into three categories: functionality, use case specifications, and supplementary requirements. Functional requirements will list each specific feature and capability of the product. Use case specifications will define how a user should expect to operate with the product and the expected behaviors. Finally, supplementary requirements will detail additional non-functional requirements and constraints such as development language and paradigms.

# Specific Requirements

## Functionality

### Binary Operators

Binary operators are operators that require two operands. The operator is preceded by its first operand and proceeded by its second. Their operands shall be either constant numbers or expressions which produce numerical values when evaluated. If either of their operands lacks a numerical value, or the operator has fewer than two operands, an error exists, as defined in sections 3.1.5.2 and 3.1.5.1 respectively. All binary operators shall evaluate to a numerical value.

#### Addition

The calculator shall be able to perform addition. Addition shall be represented with the “+” symbol. It shall conform to the requirements for binary operators as defined in section 3.1.1. Its value shall equal the sum of its operands.

#### Subtraction

The calculator shall be able to perform subtraction. Subtraction shall be represented with the “-” symbol. It shall conform to the requirements for binary operators as defined in section 3.1.1. Its value shall equal the difference between its first operand and its second.

#### Multiplication

The calculator shall be able to perform multiplication. Multiplication shall be represented with the “\*” symbol. It shall conform to the requirements for binary operators as defined in section 3.1.1. Its value shall equal the product of its operands.

#### Division

The calculator shall be able to perform division. Division shall be represented with the “/” symbol. It shall conform to the requirements for binary operators as defined in section 3.1.1. Its value shall equal the quotient of its first and second operands. If its second operand is zero, an error exists, as defined in section 3.1.5.3.

#### Modulo

The calculator shall be able to perform modulo. Modulo shall be represented with the “%” symbol. It shall conform to the requirements for binary operators as defined in section 3.1.1. Its value shall equal the value of its first operand mod its second, or the remainder of its first operand divided by its second. If its second operand is zero, an error exists, as defined in section 3.1.5.4.

#### Exponentiation

The calculator shall be able to perform exponentiation. Exponentiation shall be represented with the “^” symbol. It shall conform to the requirements for binary operators as defined in section 3.1.1. Its value shall equal its first operand to the power of its second operand, or one multiplied by its first operand a number of times equal to its second operand (for positive exponents).

### Unary Operators

Unary operators are operators that require one operand. The operator precedes its operand. Its operand shall be either a constant number or an expression which produces a numerical value when evaluated. If its operand lacks a numerical value, or the operator has no operands, an error exists, as defined in sections 3.1.5.2 and 3.1.5.1 respectively.

#### Negation

The calculator shall be able to perform negation. Negation shall be represented with the “-” symbol. It shall conform to the requirements for unary operators as defined in section 3.1.2. Its value shall equal the value of its operand, multiplied by negative one.

#### Absolute Value

The calculator shall be able to perform absolute value. Absolute value shall be represented with the “+” symbol. It shall conform to the requirements for unary operators as defined in section 3.1.2. Its value shall equal the absolute value of its operand, or the square root of the square of its operand.

### Parentheses

The calculator shall be able to use parentheses to control operator execution. Parentheses consist of two symbols: “(” to ‘open’ the parentheses, and “)” to ‘close’ them. All symbols between the open and close of parentheses are said to be in the parentheses. If a “(” does not have a corresponding “)”, or vice versa, an error exists, as defined in section 3.1.5.5. “(” and “)” shall be paired as follows: the equation is evaluated left-to-right; each time a “(” is encountered, it is added to a stack; each time a “)” is encountered, it is paired with the “(” on the top of the stack, and that “(” is removed from the stack. *Note: this is merely an analogy; use of a stack is not required in implementation; any method which produces the same results is acceptable.* The numeric value of parentheses shall be equal to the numeric value of the expression contained in them. If there is not an expression in the parentheses, or if the expression in the parentheses lacks numerical value, an error exists, as defined in sections 3.1.5.1 and 3.1.5.2 respectively. Expressions within parentheses shall be executed following the precedence rules defined in section 3.1.4.

### Operator Precedence

In equations with more than one operator, operations shall be performed with the standard operator precedence, as follows.

Level 5: Parentheses

Level 4: Exponentiation

Level 3: Multiplication, division, and modulo

Level 2: Negation and absolute value

Level 1: Addition and Subtraction

Operations shall be executed from highest precedence level to lowest. In the case of a conflict, operations shall be executed left-to-right. Equations inside parentheses shall be evaluated with the highest priority; within the parentheses, the operators shall be executed in the order specified in this section, as described in section 3.1.3.

### Errors

An error shall be produced if the user-provided equation does not conform to the requirements listed herein. If an error exists, this shall be communicated to the user as defined in section 3.1.8. Each error has an error message which describes to the user why an error exists.

#### Missing Operand Error

A missing operand error exists if any operator or parentheses has fewer operands than required.

Its error message is: “{operator symbol} is missing an operand!”.

#### Operator as Operand Error

An operator as operand error exists if the operand of any operator or parentheses is a non-unary operator or a unary operator that lacks an operand, such as in the equations “1+\*1” or “(-)”.

Its error message is: “cannot use operator as operand!”.

#### Divide By Zero Error

A divide by zero error exists when the second operand of a division operator is zero (either as a numeric constant or the result of another operator).

Its error message is: “cannot divide by zero!”.

#### Modulo By Zero Error

A modulo by zero error exists when the second operand of a modulo operator is zero (either as a numeric constant or the result of another operator).

Its error message is: “cannot modulo by zero!”.

#### Negative Operand Error

A negative operand error exists if the second operand of a modulo operation is negative.

Its error message is: “{operator symbol}’s second operand cannot be negative!”

#### Unmatched Parentheses Error

An unmatched parentheses error exists if it is not possible to match all the opening parentheses in the equation to closing parentheses, and vice versa. For example, if the numbers of opening and closing parentheses are not equal, or if a closing parentheses is not preceded by an opening parentheses, such as in the case: “)(”.

Its error message is: “unmatched parentheses!”

#### Invalid Symbol Error

An invalid symbol error exists if the user-provided equation contains any characters outside the character sets defined in section 3.1.6.

Its error message is: “illegal symbol: {offending symbol}!”

#### Missing Operator Error

A missing operator exists if the user-provided equation contains expressions which evaluate to numeric constants and don’t have operators between them, such as in the cases: “1 1” and “5\*-2 4+4”

Its error message is: “illegal symbol: missing operator!”

#### Input Overflow Error

An input overflow error exists if the user provides as an operand an integer greater than 2^31-1 (the maximum value a 32-bit integer can hold).

Its error message is: “input overflow!”

### User Input

The system shall be able to take in an equation from the user. The method for acquiring this equation is defined in section 3.1.8. The equation shall consist of characters from the following sets:

Operator characters: “+-\*/^%()”

Numeric characters: “0123456789”

Spaces: “ ”

If the equation contains any characters which are not included in these character sets, an error exists, as described in section 3.1.5.6.

The characters in the equation shall be translated into operators and constant numeric values as follows:

Each operator symbol shall correspond to an operation, as defined in sections 3.1.1 - 3.1.3. When one symbol may refer to multiple operations, the operation shall be chosen in such a way that no errors exist, if possible. For example, in the case of 1--1, if both of the “-” referred to subtraction, an error would exist, while if the first referred to subtraction and the second referred to negation, an error would not exist, and thus the first shall be subtraction and the second negation.

Numeric characters shall correspond to constant numeric values. All adjacent numeric characters shall be combined to form one numeric value. Two characters are adjacent if they are not separated by any characters, including spaces. All input numeric values shall be integers.

Spaces shall not correspond to any equation elements.

If the user inputs an integer grater than 2^31-1, an error exists, as defined in section 3.1.5.9.

### System Output

The system shall have two categories of outputs: the numerical value of the user-provided equation, or an error message. These will be communicated to the user as described in section 3.1.8. If there exists an error in the user-provided equation, the system output shall be an error message. Otherwise, it shall be a numeric value.

The system shall be able to display floating point outputs with up to three decimal places of precision. The numerical output shall be a value in base ten. The system shall output the fewest characters necessary to communicate the numerical value; there shall be no preceding zeroes. Numerical output shall consist only of the digits 0-9, ‘.’, and ‘-’. The numeric output shall equal the value of the user-provided equation.

The system shall output error messages in the following format: “Error: [message]”, where the message shall be a message relevant to the error, as defined in section 3.1.5.

### User Interface

The user shall be able to interact with the calculator through a textual interface. The user shall be able to type an equation consisting of the symbols specified in section 3.1.6. The user may input symbols outside this set of symbols; however, this will result in an error, as described in section 3.1.6.

The user shall be able to evaluate the user-provided equation by pressing the Enter key. At this point, the calculator shall display to the user the numerical result of their equation, or the relevant error if an error has occurred, as defined in section 3.1.7. If the user presses the Enter key without providing an equation, the system shall provide no output.

The user shall be able to exit the calculator by pressing the window-closing button standard to the current operating system.

## Use-Case Specifications

### System Use Case Diagram

A diagram of a algorithm

Description automatically generated with medium confidence

### Main Use Case

Use Case: Evaluate User Equation

Scope: Arithmetic Expression Evaluator

Level: Summary

Context: The goal of the user is to evaluate an arithmetic expression.

Multiplicity: One user will be interacting with one Arithmetic Expression Evaluator. The user may have multiple equations they would like to evaluate, but they will be entered and evaluated sequentially.

Primary Actor: User

### Basic Flow

#### User Enters an Equation

The user types an equation into the system’s textual interface.

#### User Submits Their Equation

The user presses Enter to submit their equation.

#### System Evaluates the Equation

The system parses the user’s equation.

The system evaluates the parentheses in the equation.

The system evaluates the operators in the equation.

#### System Displays the Result

The system displays the numeric value of the equation to the user.

#### Repeat Basic Flow

Restart Basic Flow at **User Enters an Equation**.

### Alternate Flows

#### User Does Not Enter an Equation

At the **User Submits Their Equation** step of the ***Basic Flow***, if the user has not entered an equation,

1. Resume basic flow at **Repeat Basic Flow**.

#### User Closes the Program

At any step of the ***Basic Flow***, if the user presses the button to close the program,

1. Close the program.
2. End Basic Flow.

#### Invalid Symbol Error

At the **System Evaluates the Equation** step of the ***Basic Flow***, if the user-provided equation contains an illegal character,

1. Display to the user an error message in the form: “Error: illegal symbol: {offending symbol}!”
2. Resume the Basic Flow at **Repeat Basic Flow**.

#### Unmatched Parentheses Error

At the **System Evaluates the Equation** step of the ***Basic Flow***, if the user-provided equation contains unmatched parentheses,

1. Display to the user the error message: “Error: unmatched parentheses!”
2. Resume the Basic Flow at **Repeat Basic Flow**.

#### Missing Operand Error

At the **System Evaluates the Equation** step of the ***Basic Flow***, if the user-provided equation contains an operator that is missing an operand or an empty set of parentheses,

1. Display to the user an error message in the form: “Error: {offending operator} is missing an operand!”
2. Resume the Basic Flow at **Repeat Basic Flow**.

#### Operator as Operand Error

At the **System Evaluates the Equation** step of the ***Basic Flow***, if the user-provided equation contains an operator whose operand is a non-unary operator,

1. Display to the user the error message: “Error: cannot use operator as operand!”
2. Resume the Basic Flow at **Repeat Basic Flow**.

#### Divide by Zero Error

At the **System Evaluates the Equation** step of the ***Basic Flow***, if the user-provided equation contains a division operator whose second operand evaluates to zero,

1. Display to the user the error message: “Error: cannot divide by zero!”
2. Resume the Basic Flow at **Repeat Basic Flow**.

#### Negative Operand Error

At the **System Evaluates the Equation** step of the ***Basic Flow***, if the user-provided equation contains a modulo or exponentiation operator whose second operand is negative,

1. Display to the user an error message in the form: “Error: {operator symbol}’s second operand cannot be negative!”
2. Resume the Basic Flow at **Repeat Basic Flow**.

## Supplementary Requirements

### Programming Language

The calculator shall be developed using c++.

### Design Methodology

The system shall be implemented using an object-oriented approach.

# Classification of Functional Requirements

|  |  |
| --- | --- |
| **Functionality** | **Type** |
| Binary Operators | Essential |
| Addition | Essential |
| Subtraction | Essential |
| Multiplication | Essential |
| Division | Essential |
| Modulo | Essential |
| Exponentiation | Essential |
| Unary Operations | Essential |
| Negation | Essential |
| Absolute Value | Essential |
| Parentheses | Essential |
| Operator Precedence | Essential |
| Missing Operand Error | Essential |
| Operator as Operand Error | Essential |
| Divide by Zero Error | Essential |
| Negative Operand Error | Essential |
| Unmatched Parentheses Error | Essential |
| Invalid Symbol Error | Essential |
| User Input | Essential |
| System Output | Essential |
| User Interface | Essential |

# Appendices

(Not considered part of the requirements)

## Glossary

Base Ten: A method of assigning a place value to numbers (ie. Each digit in a position of a number can be an integer value from 0 to 9)

Modulo: Operation which produces the remainder of an integer division.

Operand: An object that is operated on by an operator.

OOP (Object-oriented programming): A programming model that centers software design around data (objects), rather than functions and logic.

32-bit integer (signed): Minimum values of -2,147,483,648 and a maximum value of 2,147,483,647 (inclusive).

32-bit integer (unsigned): Minimum value of 0 and a maximum value of 4,294,967,295 (inclusive).

Truncating division: Division where a fraction result is converted to an integer.

Unary: Consisting of or involving a single element or component.