University of Kansas EECS 348

Arithmetic Expression Evaluator in C++ Software Requirements Specifications

Version 2.0

Revision History

Date	Version	Description	Author
10/1/2024	1.0	Sections 1 and 2 filled out	Lillian, Eliza, Tyler, Johney, Max
10/2/2024	1.2	Sections 2.1.3 and 2.3	Eliza
10/7/2024	1.3	Sections 1.1 and 2.1.2	Daniel
10/7/2024	1.3	Sections 2.4 and 2.5	Johney
10/7/2024	1.4	Sections 1 and 2.1.1	Tyler
10/8/24	1.5	Drafted parts of section 3.1	Eliza
10/10/24	1.6	Section 2 and 3	Eliza
10/11/24	1.7	Added to section 3.1	Tyler
10/11/24	1.8	Added to section 3.1.7	Max
10/15/24	1.9	Added to sections 3.2.2 and 3.2.4	Lillian
10/16/24	1.10	Added to section 3.2.3	Max
10/16/24	1.11	Filled out 3.1 and added to 3.2.6	Tyler
10/17/24	1.12	Filling out section 3.3 as a team	Lillian, Eliza, Tyler, Johney, Max
10/17/24	2.0	Adding to section 3.1.7; final revisions made	Max

Table of Contents

Software Requirements Specifications

1. Introduction

This Software Requirements Specification document provides an overview of the requirements for the Arithmetic Expression Evaluator. Its purpose is to document all requirements identified during the requirements engineering phase of the AEE project. Additionally, this document serves as a reference for the development team.

The scope of this SRS document includes all necessary requirements for the AEE, including functional and non-functional requirements, as well as any constraints. It also contains all information needed to interpret these requirements, including definitions of acronyms and a list of references used in the document.

1.1 Purpose

The Software Requirements Specification defines the general outline for the program by defining the requirements and structure of the AEE. Functional requirements specify what the project will include, non-functional requirements will detail how this will be included and other functionalities, and constraints give limitations on how functionalities of the Arithmetic Expression Evaluator will be created. This will ensure that all factors are covered to create a feature-complete product in a comprehensive manner.

1.2 Scope

The Software Requirements Specification applies to the Arithmetic Expression Evaluator in C++ (AEE) as detailed in the Software Development Plan document. The AEE will be capable of parsing arithmetic expressions, including handling parentheses and invalid input, and will be implemented into a larger compiler. The AEE is associated with the Use-Case models outlined in Section 3.2. The SRS document is only for use by the AEE project team and the EECS 348 instructors.

1.3 Definitions, Acronyms, and Abbreviations

AEE- Arithmetic Expression Evaluator in C++; the name of the project outlined in the SRS and SDP documents

EECS- the department of Electrical Engineering and Computer Science at the University of Kansas

EECS 348- class number for Software Engineering I at the University of Kansas

GCC- GNU Compiler Collection; C++ compiler used to compile all code written for the AEE

int- integer

KU- University of Kansas

SDP- Software Development Plan document

SRS- Software Requirements Specifications document

UPEDU- Unified Process for Education; software development process that streamlines development

VCS- version control system

1.4 References

Guiochet, J. (2016). *Use case textual description template*. Retrieved from https://www.researchgate.net/figure/Use-case-textual-description-template_fig2_290219988.

Saiedian, Hossein. (2024). *A brief introduction to UML use case modeling* [PowerPoint slides]. EECS 348 Fall 2024 Canvas.

Saiedian, Hossein. (2024). *Software requirements engineering* [PowerPoint slides]. EECS 348 Fall 2024 Canvas.

Saiedian, Hossein. (2024). *Striving for Successful Team Projects* [PowerPoint slides]. EECS 348 Fall 2024 Canvas.

1.5 Overview

Overall Description - This section discusses the general factors that will affect the product and its requirements. The section doesn’t state specific requirements, rather, it provides background for those requirements, and makes them easier to understand. This includes Product Perspective and Functions, User Characteristics, Constraints, and Assumptions & Dependencies.

Specific Requirements - This section includes an in depth look at the software requirements, and what the team needs to accomplish to meet the necessary needs. It covers Functionality, Use-Case Specifications, and Supplementary Requirements.

Classification of Functional Requirements - This section lists all functional requirements and orders them by type or order of appearance in the document.

Appendices - Includes any additional information.

2. Overall Description

Requirements give a shape to the project and define what the project will and will not consist of, as well as how those requirements will be met. Project requirements consist of user and software interfaces, constraints on memory, main functions of the project, user's attributes, constraints, and assumptions.

2.1 Product perspective

2.1.1 User Interfaces

The user interface for the AEE will consist of a command-line interface that prompts the user to enter a math expression. This interface will continuously ask for input until the user indicates they would like to terminate the program. Once an expression has been entered, the interface will display the result in the terminal. Additionally, the interface will be structured in a manner that is easily readable by the user. This includes precise and informative option menus, clear output, and helpful feedback for invalid input.

2.1.2 Software Interfaces

The program will consist only of a command-line interface as mentioned previously, with outputs being displayed to the larger program's interface in a comprehensive manner so that the external program does not have to handle the outputs as errors, unless the external software gives an invalid input. The larger program will call the AEE within its interface and the AEE will perform all the needed operations the larger program needs.

2.1.3 Memory Constraints

The *Arithmetic Expression Evaluator* will only hold the memory of values for the time that it is run; when the call for the AEE ends it loses memory of the values it was working with. In short terms the AEE has volatile memory. This means that the user of this project will be responsible for handling memory after the AEE returns. The AEE also has constraints in memory based on the hardware it is run on. If the hardware only has enough space and registers to handle a certain number of bits, then that number is the limit.

2.2 Product Functions

The AEE will be able to handle valid as well as invalid mathematical expressions. It will take in a string input consisting of numbers and operators [+,-,*,/,%,**,(,)], converting the input into integers and floats, and producing the appropriate mathematical output. Since the AEE will be able to handle invalid input without crashing, the program must also be able to identify nonsensical mathematical equations (eg. 7/0, 34+*80) as well as handle said input without error. The AEE will also be incorporated into a larger compiler program, so it must be able to communicate with the compiler and deliver output in an usable form. Incorporation into the compiler may involve removing certain functionality, such as requiring user input from the terminal.

2.3 User Characteristics

The user of this project will be a larger software or an individual. The first scenario is more likely where the software will pass expressions made up of operators and parentheses to the *Arithmetic Expression Evaluator*. This software will also take the return of the AEE, which will be integers. Specific characteristics of the larger software are unknown other than what it will give and what it expects as a return. In the latter case where the project is used by an individual human, the human will be more likely to make mistakes or give an indefinite input that could cause an error, which the project will need to handle. This individual might also not understand the parameters of what the AEE can handle which is limited by the hardware it is run on and how many bits that hardware can store.

2.4 *Constraints*

The Arithmetic Expression Evaluator (AEE) must be developed in C++ to align with the project's requirements and to ensure compatibility with the larger compiler system. The evaluator is constrained to operate within a command-line interface, limiting interactions to text-based input and output. This ensures compatibility with the larger compiler system and minimizes resource demands. The AEE will identify and report any invalid inputs, but it won't handle errors beyond reporting them. This keeps the error handling simple and focused, allowing the AEE to run smoothly without needing extra resources. The AEE will only store data temporarily for each calculation. Once it gives the result, any further memory handling will be managed by the larger system. This helps keep the AEE lightweight and focused on one calculation at a time. The evaluator is restricted to using the C++ int and float data types to handle numerical operations. This constraint provides a consistent framework for calculations, simplifying the code and reducing the risk of overflow in complex expressions. Team members will limit the scope of operations to basic arithmetic operators (+, -, *, /, %, and **). This constraint aligns with the project's requirements, enabling clear and predictable functionality. The AEE needs to complete calculations quickly, working with just one task at a time. This keeps the project simple and ensures it runs well, even on basic systems. The AEE will process and evaluate expressions but won't directly interact with users beyond the command line. Any additional user interface tasks will be managed by the larger system, allowing the AEE to stay focused on its core function. By following these constraints, the team can streamline the development process, manage resources effectively, and meet the AEE's requirements. These guidelines support decision-making and keep the project aligned with its goals.

2.5 *Assumptions and dependencies*

The team has established a shared understanding of the requirements for the Arithmetic Expression Evaluator (AEE). Below are the assumptions that will guide the development process, as well as the dependencies that must be accounted for to ensure the project's success.

Assumptions:
The AEE will accurately parse and evaluate arithmetic expressions that include basic operators (+, -, *, /, %, and **). Team members assume that handling these operators will cover the main functional requirements outlined. Team members assume input will generally follow standard arithmetic syntax. However, the project will include error handling to detect and respond to unexpected input, which ensures robustness. Memory management is a priority, so team members assume that the C++ int and float data types will be sufficient for the project's needs. Team members won't be accommodating calculations that require handling of very large or complex numbers outside the standard C++ capabilities and memory space. The output of the AEE will be designed to smoothly integrate with a larger compiler system. Team members will work to ensure compatibility through a specific interface, as detailed in the requirements.

Dependencies:
The project relies on an appropriate C++ runtime environment and compiler that can handle standard C++ operations. This is essential for both development and execution. The project depends on the C++ Standard Library for essential features such as string manipulation and math operations, eliminating the need for external libraries.To meet the integration requirements, the AEE will be designed to interact with the larger compiler system, which will manage any extended error handling, logging, or memory management tasks beyond the AEE's scope. Team members assume the larger compiler system will handle user input beyond the AEE's direct requirements, so the project will be developed with a clear boundary for where the AEE's role ends and the compiler system's role begins.

By following these steps and clearly understanding these assumptions and dependencies, team members can effectively meet the project requirements and deliver a well-functioning Arithmetic Expression Evaluator. Team members will regularly review progress to ensure alignment with these assumptions and adjust as needed based on any new insights or requirements that may arise.

3. Specific Requirements

Here, the requirements for the Arithmetic Expression Evaluator are specified for the team and users to reference. Each requirement will be described in great detail to cover all possible use cases and all possible scenarios. Use case modeling will be able to visually display these requirements in a digestible format, following the requirements detailed below. The following sections will describe the functional, non-functional, and constraint requirements, as well as all use cases, for the AEE in quality detail.

3.1 Functionality

This requirements section of the SRS outlines the specific tasks that the AEE needs to perform to satisfy user needs. Additionally, this section provides a list of functional requirements as well as an overview of the project use cases. Each requirement is written in natural language and clearly conveys the user expectations for the AEE. This section of the document also serves as a reference for the AEE development team. When entering the design phase of the AEE project, team members will frequently reference the requirements listed in this section to ensure the project stays aligned with its goals. The requirements section also lays the groundwork for the testing phase of the project, as the development team will use these requirements to check if the software performs as expected.

The goal of listing the requirements in a clear, concise manner is to avoid any confusion that might arise during the development of the AEE. By avoiding misunderstandings, less time will be spent interpreting the requirements, and more time can be spent ensuring user satisfaction.

The AEE will evaluate expressions according to standard mathematical precedence rules. The order of precedence is as follows: Parenthese(), Exponentiation **, Multiplication *, Division /, and Modulus %, Addition + and Subtraction -. The precedence ensures that multiplication, division, and modulus are evaluated before addition and subtraction, unless grouped differently by parentheses. This hierarchy allows for predictable and accurate calculations according to standard arithmetical principles.

3.1.1 Parenthesis

Opening parenthesis will be indicated by the "(" string, and closing parenthesis will be indicated by the ")" string. Each parenthesis must have an opening or closing match, meaning there must be an equivalent amount of opening and closing parenthesis.
Each parenthesis will be required to have a matching pair. An array will keep track of pairings: every time a parenthesis "(" is detected, one will be added to the array, and every time an ")" parenthesis is detected one will be deleted from the array.

3.1.2 Exponentiation

Exponentiation will be indicated by the string "**". The integer before the exponentiation symbol will be the base of the exponent and the integer preceding the symbol will be the power that the base is raised to. There will be a check for three special cases of power values:
Negative power: if the power is negative, then the value of the base will become equal to its reciprocal to the power of that base; the base and its power will be put in fraction form over one: $a^{-x} \rightarrow \frac{1}{a^x}$. This will be done by first calculating that a^x is without the negative, and then 1 will be divided by that result.
Zero power: if the power is negative then the result of the expression will result in 1.
Fraction power: if the power is any number between 0 and 1, then the base will be put to the power of the numerator of that fraction and subsequently will be put to the root of the denominator, meaning it will be run through the root function: $a^{\frac{x}{y}} \rightarrow \sqrt[y]{a^x}$

3.1.3 Multiplication and Division

Multiplication will be indicated by the “*” string. The integer before the multiplication symbol will be the first operand and the integer preceding the symbol will be the second operand. If there is another multiplication symbol after the second operand (making it also the first operand in another evaluation), the Arithmetic Expression Evaluator will evaluate the current first and second operand and then the result of that will become the first operand for the next multiplication.
Division will be handled similarly but will instead be indicated by the “/” string, with the integer before it being the first operator and the integer after it being the second operator.

3.1.4 Modulus

Modulus will be indicated by the “%” string. The integer before the modulus symbol will be the dividend and the integer after the modulus symbol will be the divisor. Modulus returns the remainder of the dividend and divisor. Since division is involved in this operation, division, multiplication, and subtraction will be supporting functions for the modulus operation. First, the dividend will be divided by the divisor, and that value will be floored to get an integer. Then, the divisor will be multiplied by that result. That value will then be subtracted from the dividend to get the remainder.

3.1.5 Addition and Subtraction

Addition will be indicated by the “+” string and subtraction will be indicated by the “-” string. Addition and subtraction will be evaluated according to the mathematical order of operations, meaning they will be performed after exponents and multiplication and division, unless the operation is inside of parentheses. When inside of parentheses, addition and subtraction take precedence over all other operations. The Arithmetic Expression Evaluator will interpret negative numbers as single operands when they appear at the beginning of an expression or immediately following another operator. For example, in the expression -5 + 3, -5 is treated as a single operand, not as the result of 0 - 5.Addition and subtraction are left-associative, meaning that they are evaluated from left to right. For instance, in the expression 5 - 3 + 2, the evaluator first calculates (5 - 3), then adds 2 to the result.

3.1.6 Invalid Input

Before performing any operations, the program will read the input for any errors in the input. Errors may include extra parentheses, mixing mathematical operations (e.g. 1+*4), and non-math related strings. The program will print that an error has occurred and specify what the error is, and will allow the user to input a new string. Additionally, the AEE will check for basic mathematical calculation errors such as dividing by 0; undefined exponentiation (0 ** -2), non-integer modulus (7.5%2), and negative square roots. The AEE will also include error handling for empty inputs, notifying users when no expression has been entered and prompting them to provide a valid one. Lastly, the AEE will inform the user about any sort of invalid operators used within an expression (2^2).

3.1.7 Incorporation Into Larger System

The program as a whole will be usable within a larger compiler. The AEE will be able to deliver inputs from arithmetic expressions and output them to the compiler. This includes any addition, subtraction, multiplication, division, exponentials, and use of parentheses. Any syntactical errors or invalid inputs in a mathematical expression will be handled by the AEE and let the user know if there are any errors.

3.1.8 User Input From Terminal

The terminal will prompt the user for input until they enter an expression. After the expression has been entered, the AEE will check the input for any possible errors. If any errors are found, the AEE will reject the input and request new input from the user. The AEE will provide detailed feedback based on the specific error detected. For formatting errors, the AEE will include an example of correct input in the error message. Additionally, the AEE will give the user the option to quit the program by entering the “q” key.

3.2 Use-Case Specifications

3.2.1 Parenthesis:

Use Case Name: Parenthesis
Actors: Human user
Preconditions: User must have a valid set of matching parenthesis in their math expression input
Normal flow:
1. AEE checks for validity of mathematical expression containing parenthesis
2. AEE checks which statements have increased precedence due to parenthesis and they are called first
3. Use case performs the operation with increased precedence inside the parenthesis

Exceptional flow:
1. AEE finds that expression including “()” is syntactically incorrect; Invalid input use case is called

Postconditions: Further evaluation of the expression is done by other use cases
Non-functional requirements: Code written for this use case must be done in C++

3.2.2 Exponentiation

Use Case Name: Exponentiation
Actors: Human user
Preconditions: User must have inputted the “**” symbol into the terminal
Normal flow:
1. AEE checks for validity of the mathematical expression including “**”
2. Any use cases present that take precedence over exponentiation (parentheses) are called first
3. Use case raises whatever expression precedes the “**” to the power of whatever succeeds the “**”

Exceptional flow:
1. AEE finds that the mathematical expression including “**” is syntactically incorrect; Invalid Input use case is called

Postconditions: Further evaluation of the expression is done by other use cases
Non-functional requirements: Code written for this use case must be done in C++

3.2.3 Multiplication and Division:

Use Case Name: Multiplication and Division
Actors: Human user
Preconditions: User must have inputted either the “*” symbol for multiplication, or the “/” symbol for division into the terminal.
Normal flow:
1. AEE checks for validity of the mathematical expression including “*” or “/”
2. Any use cases present that take precedence over multiplication and division (parentheses and exponentiation) are called first.
3. Use case multiplies what precedes the “*” by whatever succeeds the “*”, OR use case divides what precedes the “/” by whatever succeeds the “/”.

Exceptional flow:
1. AEE find that the mathematical expression including “*” or “/” is syntactically incorrect; Invalid Input use case is called.

Postconditions: Further evaluation of the expression is done by other use cases
Non-functional requirements: Code written for this use case must be done in C++

3.2.4 Modulus:

Use Case Name: Modulus
Actors: Human user
Preconditions: User must have inputted the “%” symbol into the terminal

Normal flow:

1. AEE checks for validity of the mathematical expression including “%”
2. Any use cases present that take precedence over modulus (parentheses and exponentiation) are called first
3. Use case takes the modulus of whatever precedes the “%” modulus whatever succeeds the “%”

Exceptional flow:

1. AEE finds that the mathematical expression including “%” is syntactically incorrect; Invalid Input use case is called

Postconditions: Further evaluation of the expression is done by other use cases

Non-functional requirements: Code written for this use case must be done in C++

3.2.5 Addition and Subtraction:

Use Case Name: Addition and Subtraction

Actors: Human or system user

Preconditions: The user or external program has inputted an expression containing either the “+” symbol for addition or the “-” symbol for subtraction.

Normal flow:

1. The Arithmetic Expression Evaluator (AEE) reads the input expression, checking for any instances of the "+" or "-" operators.
2. The AEE evaluates any expressions with higher precedence first (such as parentheses, exponentiation, multiplication, division, and modulus).
3. Once higher-precedence operations are resolved, the AEE processes addition and subtraction from left to right.
4. The AEE performs addition by summing the integer or float values on either side of the "+" operator.
5. For subtraction, the AEE subtracts the value on the right side of the "-" operator from the value on the left.
6. The AEE continues through the expression, performing addition and subtraction in order until the final result is calculated.
7. The AEE outputs the result of the expression, either displaying it to the user or returning it to the larger program.

Exceptional flow:

1. Invalid Syntax: If the AEE detects invalid syntax, such as consecutive operators (e.g., "5 + - 3") or an incomplete expression (e.g., "10 +"), it will trigger the Invalid Input use case.
2. Non-Numeric Input: If non-numeric characters (other than valid operators or parentheses) are present (e.g., "7 + x - 2"), the AEE will stop evaluation and return an error message specifying the invalid input.
3. Overflow or Underflow: If an addition or subtraction operation results in a value beyond the limits of the C++ int or float data types, the AEE will display an overflow or underflow error message.
4. Precision Loss: If the operation involves very large or very small values that could lead to a loss of precision, the AEE will notify the user of potential inaccuracies in the result.
5. Unmatched Parentheses: If parentheses are unmatched in the expression (i.e., "(5 + 3 - 1"), the AEE will signal an error and prompt the user to correct it.
6. Error Messages: After any of these errors are detected, the AEE will display an error message indicating the nature of the issue, allowing the user to re-enter a corrected expression.

Postconditions: Result of the addition/subtraction replaces the part of the string where the operands were

Non-functional requirements: Code for this use case must be written in C++

3.2.6 Invalid Input:

Use Case Name: Invalid input
Actors: Addition, Division, Exponentiation, Modulus, Multiplication, and Subtraction
Preconditions: Invalid input is called from a math operation use case
Normal flow:

8. The AEE checks which mathematical use case has called invalid input
9. An error message is generated based on which error is detected
10. The AEE attempts to parse the remaining expression for additional errors
11. If no further input errors are detected, the AEE displays the single error message to the user.

Exceptional flow:

7. Additional errors are found in the input
8. The AEE generates additional error messages to be displayed to the user

Postconditions: The AEE displays the input menu and prompts the user to enter new input
Non-functional requirements: Code for this use case must be written in C++

3.2.7 Input:

Use Case Name: Input
Actors: Human or system user
Preconditions: User gave any input through terminal
Normal flow:

1. AEE takes input as string of characters and stores it
2. Stored input will be checked for invalidity, which will trigger the Invalid Input use case if true
3. If check for invalid input comes back false then proceed with normal evaluation, which starts with identifying priority use cases. Prioritized use cases highest to lowest: Parenthesis, Exponentiation, Multiplication and Division, Modulus, Addition and Subtraction. If any use case is identified it will be called the evaluate, then the AEE can proceed with the next priority.

Exceptional flow:

1. Stored input is check for Invalid Input and this comes back as true
2. AEE gives an error message that is displayed in terminal that describes what the error is

Postconditions: Full evaluation is completed and all input is processed.
Non-functional requirements: Code written for this use case must be done in C++

3.3 Supplementary Requirements

Nonfunctional requirements are requirements that do not specify how the product will fill out the functional requirements that focus on efficiency, user friendliness, maintenance, and reliability; anything that's not a user feature. The nonfunctional requirements for the *Arithmetic Expression Evaluator* are listed below.

Constraints are a section of requirements that restrict the design of the project, including available time, resources, cost of project, and completion deadlines. Constraints often overlap heavily and are a subset of nonfunctional requirements. Most constraints are defined in the EECS 348 Canvas course, under project requirements. The constraints for the *Arithmetic Expression Evaluator* are listed below.

Nonfunctional requirements and constraints for the project include:

- Compatibility with the C++ language
- All code for the AEE must be written in C++
- Compatibility with GCC; all code must be compiled with GCC
- Results must be produced within the timeline outlined in the SDP
- Each Use Case outlined in Section 3.2 must have its own function and/or file
- Final project must run on the KU Linux servers without error
- All code must be pushed to a single Github repository; git must be used as the VCS

4. Classification of Functional Requirements

Functionality	Type
Parenthesis Use Case Section 3.2.1	Essential
Exponentiation Use case Section 3.2.2	Essential
Multiplication and Division Use case Section 3.2.3	Essential
Modulus Use case Section 3.2.4	Essential
Addition and Subtraction Use case Section 3.2.5	Essential
Invalid Input Use case Section 3.2.6	Essential
Input Use case Section 3.2.7	Essential
User has a simplified format to give and receive input and output.	Desirable
Finish AEE project within time constraints specified in Project Plan Document	Desirable
If the user is a human, give options for making the output decorative	Optional

5. Appendices

In the appendices section, the team will outline the essential tools, coding standards, and strategies needed to ensure a successful development process for the Arithmetic Expression Evaluator (AEE). Team members will standardize our development environment by using a specific C++ compiler, a preferred IDE (i.e. Visual Studio Code), and version control through Git, enabling effective collaboration and version tracking. Coding protocol, including consistent naming and thorough documentation, will be established to maintain clarity and readability across our codebase. Testing will be conducted at multiple levels, including unit and integration testing, to verify the functionality and robustness of each feature. To keep the project on schedule, a timeline will be followed for design completion, prototype, beta (pre-release demo), and final releases (finished product). Weekly meetings will provide a platform for discussing progress and addressing challenges, while each team member will have designated tasks and responsibilities. Lastly, a list will be compiled of resources and references that have informed our development, ensuring the project is grounded in reliable information and methodologies.