Orange

Arithmetic Expression Evaluator
Software Requirements Specifications

Version 1.0

Revision History

Date	Version	Description	Author
11 October 2023	0.1	Initial document creation / rough draft	Adam Albee
13 October 2023	1.0	Formatting/editing/2nd pass look	Adam Albee

Table of Contents

Software Requirements Specifications

1. Introduction

1.1 Purpose

This document serves to describe and define the requirements, constraints, and general design priority of the Arithmetic Expression Evaluator (AEE) program in its entirety, including the modularization strategy of the source code.

1.2 Scope

The AEE is a self-contained command-line-interface program. It relies on nothing except the standard C++ runtime and libraries and operates using exclusively the standard text input and output streams. Due to its simplicity, the entire use-case model is a User enters a line of text and receives back either an error message or the result of evaluating the input as a mathematical expression; a formal Use-Case Diagram is thus not included.

1.3 Definitions, Acronyms, and Abbreviations

See the Project Glossary.

1.4 References

The shunting yard algorithm: wikipedia.org/wiki/Shunting_yard_algorithm

1.5 Overview

The remainder of this document is split into two parts: a specification of the external interface design and justifications thereof, followed by a specification for the internal layout of the program including a high-level overview of each module and low-level details of the interfaces between modules.

2. External Interface

2.1 Software/User Interface

2.1.1 Description

The AEE exclusively communicates using the standard text input and output streams, `std::cin` and `std::cout` respectively, as a TUI program. There is no special/alternative interface for programmatic interaction. However, care will be taken for all cases of rerouting the IO streams: all inputs will be appropriately echoed to the output as if a user hand-entered them to a terminal, including proper handling of a manually injected EOF signal, and the output formatting will be identical for identical inputs regardless of terminal or file sources *and* destinations.

2.1.2 Justification

This program is explicitly intended to be a user-interfacing TUI program, not a utility function for Unix style shell scripting. That said, it is useful for automated testing that all output formatting is identical to what the user will see when operating the program manually: in this case, a simple byte-for-byte comparison of the actual and expected output including all formatting is possible.

2.2 Input Format

Each line of input will be evaluated individually, and each line is expected to have the format of a valid mathematical expression with the following specifications:

- Whitespace will generally be ignored.
  - The exception to this is that any two adjacent numbers separated only by whitespace will be considered an invalid format (E.g. “22 + 5” is fine but “2 2 + 5” will error).
- The expression will be in a valid infix notation (E.g. 1 + 2, 5*6, 23 / 2, etc. but *not* 2 2 +, etc.).
- Brackets will be supported (both parentheses `()` and square brackets `[]`).
- The following operators will be supported:
  - Addition (a + b)
  - Unary Positive (+ a)
  - Subtraction (a - b)
  - Unary Negation (- a)
  - Multiplication (a * b)
  - Division (a / b)
  - Modulo (a % b)
  - Exponentiation (a ^ b *or* a ** b)
- Numbers will be in a decimal format, with optional separators and fractional component.
  - The valid separators are “,” and “_”. All other characters will be considered invalid.
  - The “.” character determines the fractional component. Only one is allowed, but it can be placed at the very start or very end of a number.

2.3 Runtime Constraints

2.3.1 Description

The AEE is intended to be executed inside the OS on a modern general-purpose computer including any laptop or desktop computer with access to a command-line shell interface. Considering the limited scope of the program, there are effectively no practical limitations with respect to performance or memory overhead.

2.3.2 Justification

As the program uses only the standard IO streams, there is no reason to bypass the OS layer. Additionally, there is no specific reason to target any embedded system where significant memory or performance constraints *might* be necessary. Even so, this program is small and simple enough that any embedded device capable of running a modern operating system will almost certainly be able to run the AEE program with no issues.

3. Internal Layout

3.1 Main Module

The C++ `main` function will handle all the IO using, exclusively, the standard C++ IO streams. In an infinite loop, the following steps will occur in order:

1) The user will be prompted to enter an expression, which will be stored as a string.
2) This string will be passed as an argument to the "evaluate" function.
3) The "evaluate" function will attempt to evaluate the input as an expression, and either:
    a. Error, throwing an exception with a message describing the specific error.
    b. Return a string containing the evaluated result of the expression.
4) The contents of whichever result occurs are printed to the output.

3.2 Solver Module

The solver module defines the "evaluate" function, transforming the user input argument by passing it through each of the following modules in sequence before finally returning the evaluated result. If any of the inner modules throws an error, it is ignored and allowed to propagate out to the main module.

3.3 Tokenizer

The unmodified user input string is first passed to the Tokenizer, which reads the string and generates a sequence of "tokens" to be further processed. As it doesn't actually do anything, the unary positive operator is ignored and not turned into an actual token at this step.
E.g., "+32 + -25/--5" => {NUMBER[32], ADD, NEG, NUMBER[25], DIV, NEG, NEG, NUMBER[5]}

3.3.1 Error States

The tokenizer should throw an exception whenever it encounters a character that it doesn't know how to process (E.g., a "}"), or that is in an invalid placement (E.g., a "," or "_" character outside of a number).

3.4 Shunting Yard Algorithm

The string of tokens is next rearranged using a version of Djikstra's shunting yard algorithm to transform it into the trivially computable post-fix aka "Reverse Polish" notation.
IMPLEMENTATION NOTE: Both unary negation and exponentiation are *right-associative* operators!

3.4.1 Error States

This process should throw an exception when a number token is processed immediately after another number token, when a bracket mismatch is found (e.g., "([x)]"), or when any unclosed brackets remain in the operator stack when appending it to the output queue.

3.5 Postfix Stack Computation

Lastly, the resulting RPN queue from the SYA process is computed, and the result returned to be printed!

3.5.1 Error States

This computation should throw an exception if a stack underflow occurs when processing the RPN queue, or there are multiple numbers remaining in the stack when the queue is exhausted.

4. Classification of Functional Requirements

Functionality	Type
The program will perform IO exclusively through the standard IO streams.	Essential
When the standard IO streams are rerouted, the output will look identical to as if a user had manually entered the contents of the input stream to a terminal, including all formatting and the echoing of the input expressions.	Essential
Parenthesis characters “()” can be used to control order-of-operations.	Essential
Square bracket characters “[]” can also be used to control order-of-operations. (Parentheses and square brackets cannot match with each other.)	Desired
The “+” character will be treated as the binary addition operator.	Essential
The “+” character will also be treated as the *unary* positive operator.	Desired
The “-” character will be treated as the binary subtraction operator.	Essential
The “-” character will also be treated as the *unary* negation operator.	Essential
The “*” character will be treated as the binary multiplication operator.	Essential
The “/” character will be treated as the binary division operator.	Essential
The “%” character will be treated as the binary modulo operator.	Essential
The “^” character will be treated as the binary exponentiation operator.	Essential
The “**” string will also be treated as the binary exponentiation operator.	Desired
Numbers will be treated as decimal values with an optional fractional component indicated by the “.” character.	Desired
The digit separator characters “_” and “,” will be ignored when part of numbers.	Optional