C++ Calculator

Version <1.0>

Revision History

|  |  |  |  |
| --- | --- | --- | --- |
| **Date** | **Version** | **Description** | **Author** |
| 11/09/2023 | 1.0 | Finished Software Architecture Doc | Arnav Jain |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

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. Architectural Representation 4

3. Architectural Goals and Constraints 4

4. Use-Case View 4

4.1 Use-Case Realizations 5

5. Logical View 5

5.1 Overview 5

5.2 Architecturally Significant Design Packages 5

6. Interface Description 5

7. Size and Performance 5

8. Quality 5

# Introduction

The Software Architecture Document for the Arithmetic Expression Evaluator serves as a comprehensive guide detailing the system's design through different architectural viewpoints. It encapsulates the core architectural decisions and provides a blueprint to understand the system's structure and behavior.

## Purpose

This document lays out the architecture of the Arithmetic Expression Evaluator, capturing its major components and the interaction between them. It serves as a reference for developers, providing insight into the system's design and ensuring consistency and quality in implementation. The document also assists new team members in understanding the existing system architecture and facilitates future maintenance and scalability.

## Scope

The Software Architecture Document covers the design and structure of the Arithmetic Expression Evaluator, a system that interprets and computes arithmetic expressions with support for various operators and numeric constants. The architecture outlined here encompasses the parsing logic, evaluation algorithm, error handling mechanisms, and user interface, detailing how each part contributes to the system as a whole.

## Definitions, Acronyms, and Abbreviations

**PEMDAS**: Parentheses, Exponents, Multiplication, Division, Addition, Subtraction (order of operations)

**RPN**: Reverse Polish Notation

**CLI**: Command-Line Interface

## References

1. Stroustrup, B. (2013). The C++ Programming Language (4th ed.). Addison-Wesley Professional.

## Overview

The remainder of the document is structured as follows: Section 2 outlines the architectural representation of the system, Section 3 discusses the architectural goals and constraints, and Section 4 provides the use-case view. Section 5 delves into the logical view, including the design's decomposition and the significant packages and modules. Section 6 describes the user interface, Section 7 addresses the system's size and performance, and Section 8 concludes with a discussion on the quality attributes of the system's architecture.

# Architectural Representation

The software architecture of the Arithmetic Expression Evaluator system is designed to provide a clear modular structure for parsing, evaluating, and processing arithmetic expressions as defined by the project's objectives. It is represented through multiple views that capture the system's static structure, dynamic behavior, and the interaction between its components. Each view is tailored to address the concerns of different stakeholders, such as developers, testers, and maintainers.

Structural View (Module View):

* Model Elements:
  + **Parser Module:** Responsible for tokenizing the input expression and checking for syntactical correctness. It handles the extraction of operators, operands, and parentheses from the input string.
  + **Evaluator Module:** Contains the logic to process the tokenized input according to operator precedence (PEMDAS rules) and parenthesis handling.
  + **Expression Tree:** A data structure used to represent the parsed expression in a hierarchical manner, with nodes for operators and leaves for operands.
  + **Operator Classes:** Encapsulate the functionality for each arithmetic operation, including addition, subtraction, multiplication, division, modulo, and exponentiation.
  + **Constant Handler:** Manages the recognition and conversion of numeric constants from the tokenized input into their numerical representation.

Behavioral View (Process View):

* Model Elements:
  + **Expression Parsing Process:** Depicts the sequence of actions from accepting user input to generating a tokenized representation of the expression.
  + **Evaluation Process:** Demonstrates the evaluation flow, particularly how the expression tree is traversed and calculated to produce the final result.
  + **Error Handling Process:** Outlines the procedures for detecting and responding to errors such as invalid input, division by zero, or unmatched parentheses.

Interaction View (Component and Connector View):

* Model Elements:
  + **User Interface Component:** Acts as the entry point for the system, through which the user inputs expressions and receives results or error messages.
  + **Parser and Evaluator Components:** These are interconnected components where the parser feeds the structured expression into the evaluator.
  + **Logger Component:** Optionally included to facilitate debugging and provide runtime information about the parsing and evaluating processes.

Deployment View:

* Model Elements:
  + **Execution Environment:** Represents the environment in which the application runs, such as the operating system and the C++ runtime library.
  + **Binary Components:** The compiled C++ binaries that are deployed to the execution environment and executed by the user.

# Architectural Goals and Constraints

The architecture of the Arithmetic Expression Evaluator is guided by a set of goals that ensure the system's functionality, reliability, maintainability, and performance while operating within certain constraints. These goals and constraints are critical in shaping the system's architecture.

Architectural Goals:

* **Functionality:** The system must correctly parse and evaluate arithmetic expressions according to the rules of precedence (PEMDAS).
* **Reliability:** The evaluator must handle errors gracefully and provide accurate results consistently.
* **Performance:** The system should evaluate expressions efficiently, with a focus on minimizing computation time for complex expressions.
* **Reusability:** Components of the architecture, such as the parser and evaluator, should be designed to be reusable in other projects or contexts.

Architectural Constraints:

* **Safety and Security:** While the project does not involve high-stakes computations, it should still ensure that no arbitrary code execution can occur through the input of expressions.
* **Use of Off-The-Shelf Products:** If the project is constrained to use certain pre-existing libraries for parsing or evaluating expressions, it must be compatible with their licenses and integration capabilities.
* **Schedule:** The project timeline may impose constraints on the depth of the architectural design, necessitating a balance between ideal design and practical, timely delivery.

These goals and constraints serve as a blueprint for the architectural decisions throughout the development process, ensuring that the final product aligns with the project's objectives and the practical realities of its implementation.

# Use-Case View

[This section lists use cases or scenarios from the use-case model if they represent some significant, central functionality of the final system, or if they have a large architectural coverage—they exercise many architectural elements or if they stress or illustrate a specific, delicate point of the architecture.]

## Use-Case Realizations

[This section illustrates how the software actually works by giving a few selected use-case (or scenario) realizations, and explains how the various design model elements contribute to their functionality. If a Use-Case Realization Document is available, refer to it in this section.]

# Logical View

The Logical View of the Arithmetic Expression Evaluator presents a design focused on a stack-based evaluation method. This design facilitates the parsing and evaluating of arithmetic expressions, emphasizing maintainability, efficiency, and clear separation of concerns. The system is decomposed into distinct subsystems and packages, each responsible for a specific aspect of the application, such as input handling, parsing, evaluation, and error management.

## Overview

The system's design model is organized into a hierarchy of packages, each encapsulating a specific layer of functionality. At the top level, we have the Input Subsystem, Parser Subsystem, Evaluator Subsystem, and Error Handling Subsystem. The Parser Subsystem transforms input expressions into a format suitable for evaluation, while the Evaluator Subsystem computes the result. The Utility Package provides support across subsystems with common functionalities like string manipulation and numerical operations.

## Architecturally Significant Design Modules or Packages

* **Description:** Handles the conversion of user input into a tokenized format and then to Reverse Polish Notation (RPN) using stacks.
* **Tokenizer Class:**
  + **Description:** Splits input strings into tokens representing numbers and operators.
  + **Responsibilities:** Tokenization, identification of tokens.
  + **Operations:** tokenize().
  + **Attributes:** input\_string, tokens.
* **RPN Converter Class:**
  + **Description:** Converts infix expressions to RPN, considering operator precedence.
  + **Responsibilities:** Managing operator stack, generating RPN output.
  + **Operations:** convertToRPN().

**Evaluator Package:**

* **Description:** Responsible for evaluating RPN expressions to produce the final calculation.
* **Evaluator Class:**
  + **Description:** Processes RPN expressions using a stack to sequentially evaluate and calculate the result.
  + **Responsibilities:** Stack management, execution of operations.
  + **Operations:** evaluateRPN().

**Utility Package:**

* **Description:** Provides auxiliary functionalities used by various parts of the system.
* **StringUtility Class:**
  + **Description:** Offers string manipulation operations useful in parsing.
  + **Operations:** split().
* **NumberUtility Class:**
  + **Description:** Handles conversions between string representations of numbers and their numeric values.
  + **Operations:** stringToNumber().

The interaction between these classes forms the backbone of the application's functionality, with the Tokenizer feeding into the RPN Converter, and the RPN Converter output being processed by the Evaluator. The Utility Package supports all these operations with necessary utility functions.

# Interface Description

The interface for the Arithmetic Expression Evaluator consists of a command-line interface (CLI) that interacts with the user through text-based prompts, inputs, and outputs. It is intentionally minimalistic to ensure ease of use and clarity.

**User Input:**

* The CLI prompts the user with a message such as "Enter an arithmetic expression: ".
* Users are expected to input a string that represents a standard arithmetic expression.
* The input may include integers and operators such as + (addition), - (subtraction), \* (multiplication), / (division), % (modulo), and ^ (exponentiation).
* Expressions can also contain parentheses ( and ) to specify the order of operations.
* Spaces in the input are allowed and are ignored during processing for user convenience.

**Screen Format:**

* The interface provides a clean prompt on the command line.
* After the user submits an expression, the interface either displays the calculated result or a descriptive error message.
* If an error is encountered (such as invalid syntax, division by zero, or unmatched parentheses), the user is informed of the nature of the error and prompted to try again.

**Valid Inputs and Outputs:**

* A valid input follows the conventional infix notation for arithmetic expressions.
* The output, if the expression is valid, is the numerical result of the evaluated expression.
* In the case of invalid input, the output is an error message that indicates the type of error detected.

**Interface Robustness:**

* The interface includes error handling to guide the user back to the input prompt after displaying an error message.
* Input validation ensures that only appropriate characters are accepted, preventing injection of unintended commands or operations.

This interface is detailed further in the User-Interface Prototype Document, which includes specific screen formats, examples of valid and invalid inputs, and the expected results or responses from the system.

# Size and Performance

[A description of the major dimensioning characteristics of the software that impact the architecture, as well as the target performance constraints.]

# Quality

The software architecture of the Arithmetic Expression Evaluator is designed to ensure the system's quality across various dimensions:

* **Extensibility:** The modular design allows new features, such as additional operators or support for floating-point numbers, to be added with minimal impact on existing code.
* **Reliability:** Error handling is built into the architecture, ensuring that inputs are validated and errors are managed effectively, contributing to the system's overall robustness.
* **Performance:** The use of efficient data structures like stacks for parsing and evaluating expressions ensures that the system performs well, even for complex expressions.
* **Security and Safety:** The architecture avoids executing arbitrary code or operations by strictly validating inputs, thus mitigating potential security risks.