EECS 348

Arithmetic Expression Evaluator
Software Architecture Document

Version 2.0

Revision History

Date	Version	Description	Author
11/07/2023	1.0	Create document	Marcos Lepage
11/12/2023	2.0	Finalize document	Marcos Lepage

Table of Contents

Software Architecture Document

1. Introduction

1.1 Purpose

This document provides a holistic overview of both the system's intended architecture as well as the goals and constraints that motivated its design. To depict the software architecture in a comprehensive yet intuitive fashion, multiple architectural views are provided: each focusing on a different level of abstraction or aspect of the system. Additionally, all major architectural decisions and their significance are outlined within this document. This document is intended to be used by both project administrators and developers to guide the development process and ensure a quality product.

1.2 Scope

This Software Architecture Document provides a system architecture for the Arithmetic Expression Evaluator Project (the requirements of which are described in the complimentary Software Requirements Specification Document). The architecture information outlined in this document applies specifically to the development (i.e. the implementation) of the system as well as its subsequent maintenance.

1.3 Definitions, Acronyms, and Abbreviations

- AST: Abstract Syntax Tree
- CLI: Command Line Interface
- GUI: Guided User Interface
- ISA: Instruction Set Architecture
- OS: Operating System
- PEMDAS: Parentheses, Exponents, Multiplication and Division (from left to right), Addition and Subtraction (from left to right).
- SRS: Software Requirements Specification
- UI: User Interface

1.4 References

- Arithmetic Expression Evaluator: Software Requirements Specification Document
  Date: 10/15/2023 Organization: Arithmetic Expression Evaluator team

1.5 Overview

This Software Architecture Document contains, first, a description of the goals and constraints that motivated the design of the project's architecture. Second, an overview of the architecture and its representations as well as those representations themselves (along with detailed descriptions of the components included within the representations). Third, an outline of the software's interfaces (specifically its user interface). And finally, the quality characteristics of the described software architecture. Holistically, this document hopes to comprehensively outline the project's architecture, its characteristics/details, and the decisions that contributed to its design.

2. Architectural Representation

For the Arithmetic Expression Evaluator project, the term "software architecture" implies the macroscopic view of packages and components within the project, specifically, namespaces and classes (since the software is intended to be object-orientated). The views (i.e. representations) that will be used to describe and explain the architecture details for this project are as follows:

• **Package View**: Very high-level view that outlines the largest compositional units within the project. Represents entire libraries and programs as packages which can be interconnected.

• **Class View**: Using UML Class Diagrams, we can outline the connection between components (specifically, classes) within our software system. This view best explains the details of the architecture and the imperative qualities of the software system.

3. Architectural Goals and Constraints

• **Safety**: This software has no potential to produce hazards or inflict harm upon its users. Accordingly, little thought needs to be allocated towards safety during the design—and subsequently the development—phases of this project.

• **Security & Privacy**: The data submitted by the user into this software system has little to no privacy value. Moreover, this software system does not require access to unrelated/external user files, network/communication interfaces, or any other potentially exploitable vector. Granted that the system does not exceed the scope defined in the Software Requirements Specification, security & privacy are not of high concern.

• **Off-the-shelf product**: Desirably, this software system should be distributable as a single deliverable software package that runs as-is. For obscure compilation targets, the software system should be capable of being compiled locally.

• **Portability**: The software system should be capable of running on any operating system (Linux, MacOS, Windows, etc.) and any ISA (x86, ARM, etc.). Specifically, any platform that supports the compilation and execution of C++ programs.

• **Reuse**: Certain components of this software system should be reusable within other contexts (i.e. other systems that require either the tokenization or evaluation of an expression within a string). That being said, the system *as a whole* is not intended to be used as-is within other systems.

• **Constraints**: This software architecture must be capable of being implemented within C++ using object-orientated principles. Additionally, the architecture should be modular to allow for simultaneous development from multiple developers. To further facilitate development, the architecture should be intuitive (both in its structure and module interfaces). Ease & speed of development constrains much of the architecture since the timetable for this project is very finite.

4. Use-Case View

N/A

4.1 Use-Case Realizations

N/A

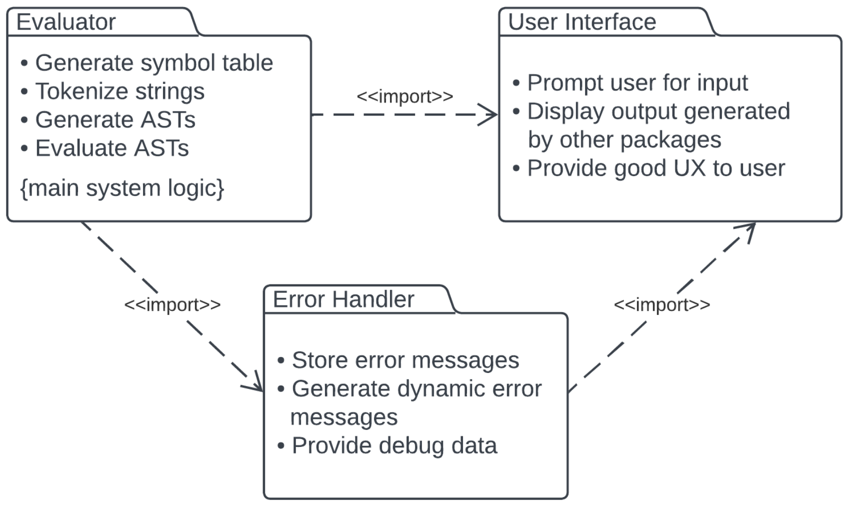
5. Logical View

5.1 Overview

This section describes the software architecture by decomposing the system into 1.) A package diagram, with each package and its responsibilities described in detail. 2.) A class diagram for the main "Evaluator" section of the codebase with descriptions of each of its components and relationships between them (note that the other packages are deemed insignificant due to their triviality). With these views, the architecture of the system should be apparent.

5.2 Architecturally Significant Design Modules or Packages

5.2.1 UML Package Diagram



Evaluation Engine Package:
• Description: The evaluation engine is responsible for the majority of the software system's functionality. The evaluation engine will be tasked with decomposing arithmetic expression strings into token vectors (by applying a pre-generated symbol table to the string). It will then be able to convert these token vectors into Abstract Syntax Trees (ASTs) for expression evaluation.

• Relationships:
- The evaluation engine is tightly linked to the error handler. The evaluation engine can call the error handle either during, tokenization, AST generation, or evaluation in order to report an error to the user.
- The evaluation engine frequently calls the user interface package in order to obtain user input and display successful evaluation results.

Error Handler Package:
• Description: The error handler package stores the error message table and is invoked to generate errors upon unsuccessful arithmetic expression parsing/evaluation.

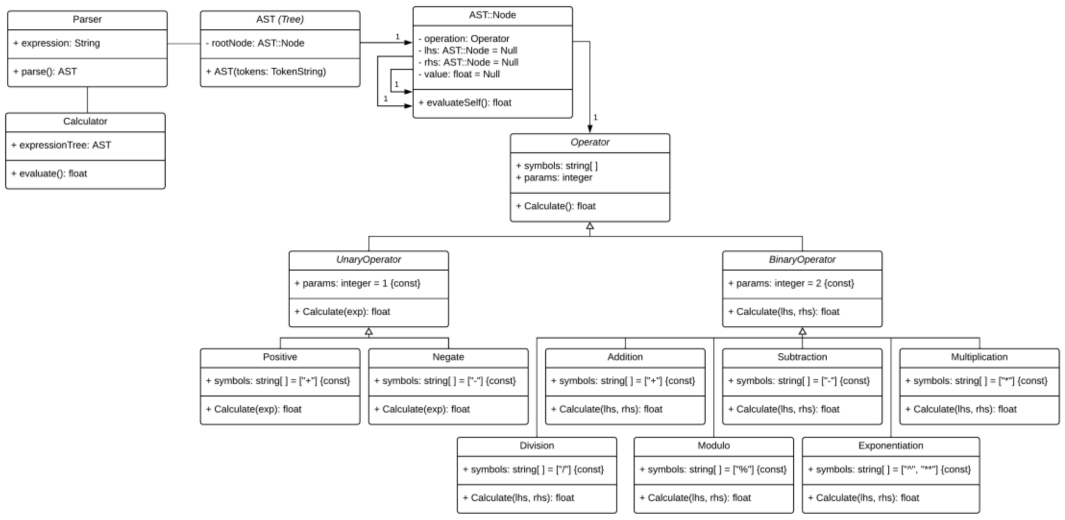
• Relationships:
- The error handler is imported by the evaluation engine to be called upon when an error is encountered. For example, a divide by zero or invalid syntax error.
- The error handler imports the user interface package to display the errors it generates.

User Interface Package:
• Description: The user interface package is responsible for collecting user input and displaying system output. It should do so in an aesthetically pleasing manner and with good UX.

• Relationships:
- The user interface package is imported by the evaluation engine to be invoked to collect user input and display successful evaluation output.
- The user interface package is imported by the error handler and is invoked to display error messages.

Note: The Error Handler & User Interface packages are considered insignificant (due to the fact that they are trivial to implement). As such, they do not warrant elaboration in the form of a class diagram.

5.2.2 ML Class Diagram (*for Evaluator*)


Parser
+ expression: String
+ parse(): AST
AST (Tree)
- rootNode: AST::Node
+ AST(tokens: TokenString)
AST::Node
- operation: Operator
- lhs: AST::Node = Null
- rhs: AST::Node = Null
- value: float = Null
+ evaluateSelf(): float
1
1
1
1
Calculator
+ expressionTree: AST
+ evaluate(): float
Operator
+ symbols: string[ ]
+ params: integer
+ Calculate(): float
UnaryOperator
+ params: integer = 1 {const}
+ Calculate(exp): float
BinaryOperator
+ params: integer = 2 {const}
+ Calculate(lhs, rhs): float
Positive
+ symbols: string[ ] = ["+"] {const}
+ Calculate(exp): float
Negate
+ symbols: string[ ] = ["-"] {const}
+ Calculate(exp): float
Addition
+ symbols: string[ ] = ["+"] {const}
+ Calculate(lhs, rhs): float
Subtraction
+ symbols: string[ ] = ["-"] {const}
+ Calculate(lhs, rhs): float
Multiplication
+ symbols: string[ ] = ["*"] {const}
+ Calculate(lhs, rhs): float
Division
+ symbols: string[ ] = ["/"] {const}
+ Calculate(lhs, rhs): float
Modulo
+ symbols: string[ ] = ["%"] {const}
+ Calculate(lhs, rhs): float
Exponentiation
+ symbols: string[ ] = ["^", "**"] {const}
+ Calculate(lhs, rhs): float

Parser Class:
• Takes an expression in the form of a string and parses it into an Abstract Syntax Tree that respects the operator precedence specified by PEMDAS.

Calculator Class:
• Takes an Abstract Syntax Tree and evaluates it recursively (using the evaluateSelf() method included in AST::Node). Returns a float containing the value of the evaluation. Alternatively, throws an error.

AST (Abstract Syntax Tree) Class:
• Tree-based data representation of an expression. Contains the root node for the tree.

AST::Node Class:
• Node for the AST Tree class. Contains an operator, left-hand side child node, and right-hand side child node. Each child node is an instance of this Node class as well.

Operator Class:
• Class that takes in a string as a symbol of the operator and an integer as a parameter. The calculator method will utilize the int to return a float as the result of the calculation.

6. Interface Description

Provides a detailed overview of the user interface for the Arithmetic Expression Evaluator, encompassing the command-line interface (CLI) and user-friendly interactions. The user interface is designed to facilitate the seamless input and output of arithmetic expressions. While the Arithmetic Expression Evaluator is primarily command-line driven, the user interface ensures a user-friendly experience with robust error handling and clarity in prompting.

- Input Format:
    - Users can input arithmetic expressions containing operators (+, -, *, /, %, ^) and numeric constants.
    - Parentheses are supported for defining precedence and grouping.
- Example Input:
    - Enter arithmetic expression: 4 * (3 + 2) % 7 -1
- Output Format:
    - The program displays the result of the evaluated expression.
- Example Output:
    - Result: 5

7. Size and Performance

N/A

8. Quality

The architecture outlined above influences the following characteristics of the software system: extensibility, maintainability, reliability, and comprehensibility. By applying the object-orientated principles of modularity and encapsulation to the architecture, we have ensured that the system will be both extensible and maintainable. In terms of extensibility, new operators (and other functionality) can be quickly appended in the form of additional class inheritances or instantiations. The separation of concerns provided by modularity enables maintainability. In practice, future developers will not need to worry about unintuitive side effects when making changes to the codebase. The outlined architecture also enables reliability. Unlike a monolith architecture, the proposed architecture has clear separation of function allowing each unit to be tested independently: reducing the chance of obscure bugs. Finally, the architecture described in this document is comprehensible. For the function it provides, it is relatively simple to understand due to the fact that it follows well-defined best practices. Altogether, the architecture specified in this document will greatly enhance the nonfunctional characteristics of the system. For further details on the requirements that this architecture satisfies, see the Software Requirements Specification document.