Truthy

Boolean Logic Simulator in C++
Software Architecture Document

Version 1.0

Revision History

Date	Version	Description	Author
11/4/24	1.0	Filled in required sections	Evan, Nathan, Mitchel, Jay, Darshil, Ian
13/4/24	1.0	Removed placeholders	Evan

Table of Contents

Software Architecture Document

1. Introduction

1.1 Purpose

The Software Architecture Document records the structure and implementation of the boolean logic simulator. The document introduces the intent and scope of the document. It then provides an overview of the boolean logic simulator's architecture, logic, design model, interface, and quality. In each section descriptions and reasoning is given for the chosen designs and decisions. The Software Architecture Document is intended to be viewed and utilized by Truthy team members and stakeholders. It serves as a reference for the implementation stage of the project and as a guide for evaluation by stakeholders afterwards.

1.2 Scope

The Software Architecture Document is informed by the Software Requirements Specification and Project Description. The design choices and intended functionality are described to be in accordance with the requirements and decisions made in these previous documents. This document is intended as a reference and guide for the implementation of the boolean logic simulator. The description of the designs will also be referenced during the creation of the Test Cases and User Manual documents.

1.3 Definitions, Acronyms, and Abbreviations

Boolean Logic: A branch of mathematics that deals only in boolean values: True and False

Tokenization: The digital representation of a real thing, in this case a boolean expression. Composed of tokens, that are recognizable and usable by the system.

Parsing: Breaking down input to fit a structure that a program can understand and process.

Permutation: A possible variation from a set of possible outcomes.

Abstract Syntax Tree: A tree data structure that will hold and represent a given expression. The operands are held in the leaf nodes and the operators are held in the non-leaf child nodes.

Graphical User Interface(GUI): The visual output of a computer program.

Software Requirement Specification(SRS): A document that outlines the requirements of a software engineering project

1.4 References

Title	Source	Publishing Organization	Last Updated Date
EECS 348 Project Description	KU	KU	2/17/2024
Software Requirements Specifications	Software Requirements Specification	Truthy	3/21/2024
User Manual	Has not been created	Truthy	Has not been created
Project Plans	Project Plan	Truthy	2/25/2024
Test Cases	Has not been created	Truthy	Has not been created

1.5 Overview

The later sections of this document describe the overarching architecture of the boolean logic calculator. Section 2 provides this overview along with descriptions of the goals and constraints of the architecture. The document then provides the design model of the architecture in section 3. This model lays out the packages of the design, how they interlink, and their purpose within the larger architecture. Diagrams and descriptions are provided for each relevant package. In the following section the user interface of the boolean logic calculator is detailed. Included are descriptions of the interface screen, valid inputs, and valid outputs. The final section provides detail into how the design model handles different aspects of the quality of the boolean logic calculator's functionality.

2. Architectural Representation

The projects aims to implement logical operations (AND, OR, NOT, NAND, XOR), implement a parser that correctly processes user input, implement the ability to evaluate the entire expression of a boolean operation to the correct True or False value, implement an error handler that can correctly handle any issue found, develop an accurate set of test cases that can test various complex combinations of logic circuits to determine the accuracy of the Boolean logic evaluator, ensure code quality and readability, and document the various steps in the software engineering process.

3. Architectural Goals and Constraints

3.1 Architectural Goals

- Implement Logical Operations: Develop the ability to perform logical operations such as AND, OR, NOT, NAND, XOR.
- Implement a parser: Create a parser that can accurately handle user input, including parentheses and boolean algebra.
- Implement Expression Evaluation: Develop the capability to evaluate entire expressions of boolean operations into a correct True or False value.
- Implement an Error Handler: Design an error handler that can correctly address any issues encountered.
- Develop Test Cases: Create a comprehensive set of test cases that can assess the correctness of the boolean logic evaluator by testing various complex logic circuit combinations.

3.2 Architectural Constraints

- Safety, Security, Privacy: These elements are crucial to the architecture design.
- Portability and Reusability: The design should consider the portability and reusability of the code.
- Design and Implementation Strategy: A clear strategy is essential to achieve the project's goals.
- Development Tools and Team Structure: Proper development tools and an effective team structure are necessary for the project's success.
- Schedule Management: It is vital to manage the schedule meticulously to ensure timely completion of all tasks.
- Legacy Code: Compatibility with existing code must be considered.

4. Logical View

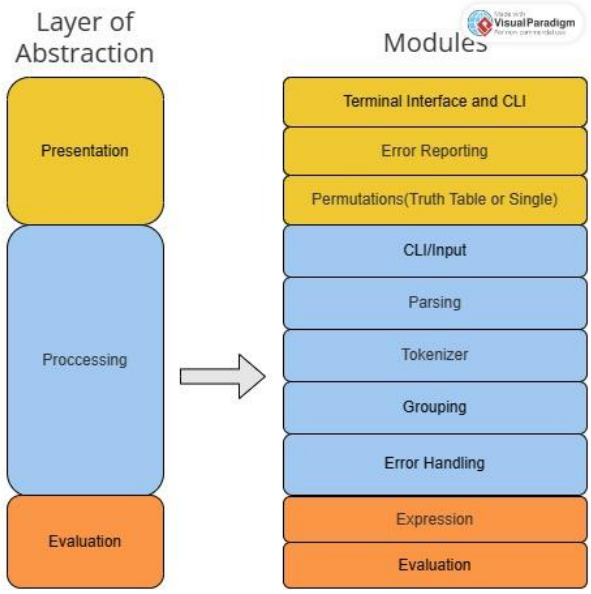
4.1 Overview

The boolean logic simulator software will be composed of 3 layers Presentation, Processing, and Evaluation.

The Presentation layer contains the GUI modules: Terminal Interface, Error Reporting, and Permutation (Truth Table generation). This layer is responsible for everything that a user would see and interact with.

The Processing layer contains everything to do with processing the user input from the CLI into a usable boolean expression. It contains the Input, Parsing, Tokenizer, Grouping, and Error Handling. This layer is the majority of the software because it can take in a complex input, and reduce it into simple parts for easier computation.

The Evaluation layer is responsible for evaluating the correct result. This will contain the evaluation module, and will use the existing expression class from the grouping module to compute the boolean expression. The result will be returned to various GUI modules depending on the situation (I.E. it will go back to the top layer.)

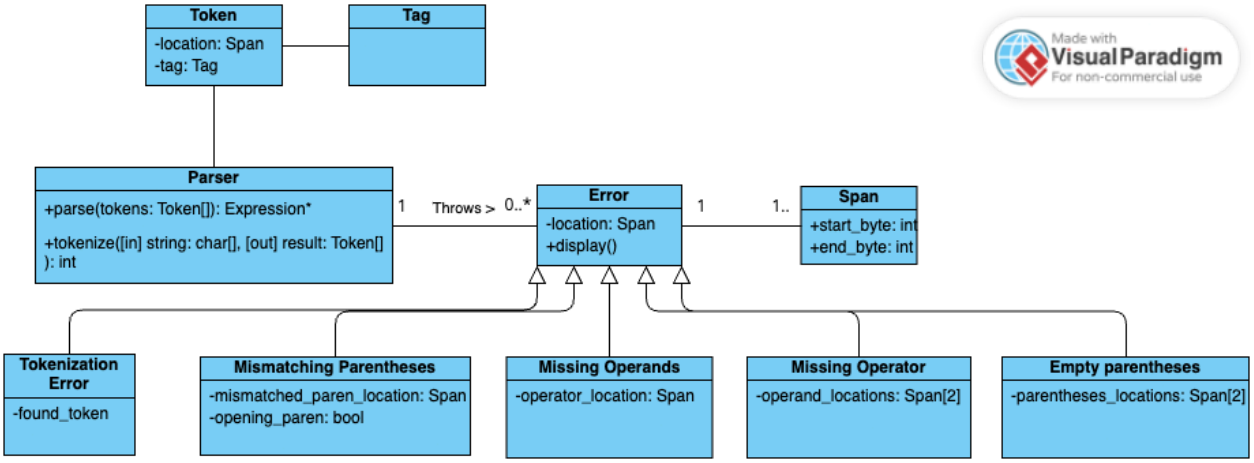


4.2 Architecturally Significant Design Modules or Packages

4.2.1 Input
Description: Gets the user input from the CLI. This includes any variables and the expression itself.

4.2.2 Parsing
Description: Handles parsing text into an expression tree representation
Major packages: Tokenizer, Grouping
Major Classes:

- Parser: The parser handles tokenizing and grouping tokens



4.2.2.1 Tokenization

Description: Turn text into a series of tokens. If an invalid string is found, it is reported with an error

Major Classes:

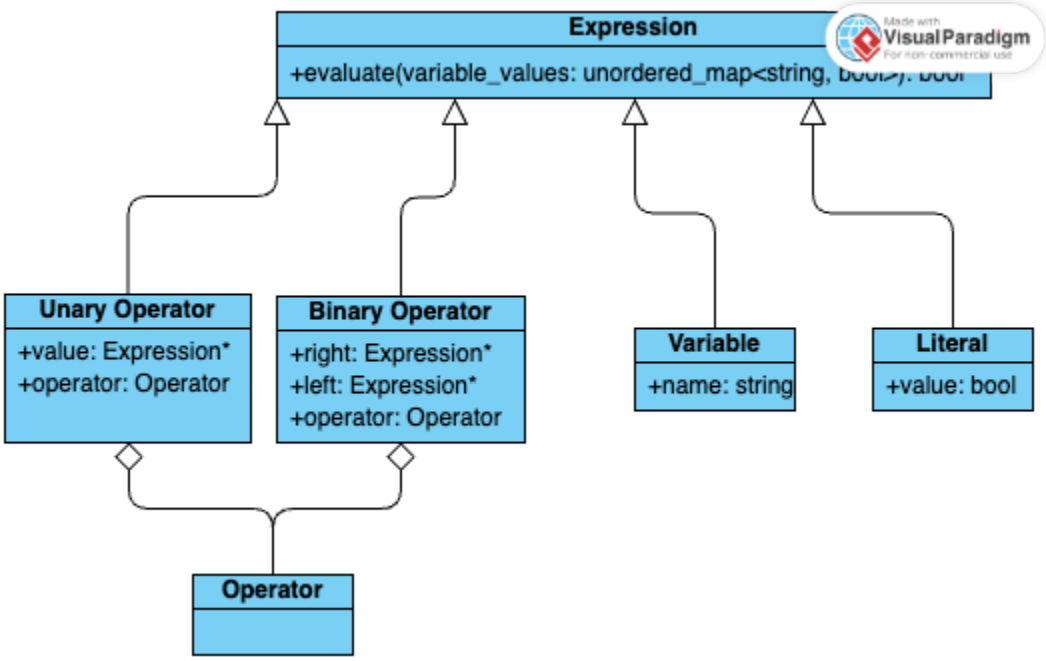
- Token: A token represents a location with a tag
- Tag: A tag is an enum that represents the type of the string in the expression. The possible tags are boolean, variable, opening parentheses, closing parentheses and each operator.

4.2.2.2 Grouping

Description: Group tokens into a tree of expressions.

Major Classes:

- Expression: Represents any expression tree which may contain child expressions
- Operator: An enum with all operator types truthy supports



4.2.2.3 Error Handling

Description: Contains the structure for any errors that tokenization and parsing may throw

Major Classes:

- Error: The error type contains the location of the error along with a method to display the error. It has subtypes for tokenization, parentheses and operand specific errors

- Span: A span represents a location in the input string. It is tracked to provide more helpful error messages to the user if the program exits early

4.2.3 Evaluation

Description: Evaluate the abstract syntax tree along with any values to fill in for variables

Major Classes:

- Expression: (also described in 5.2.2.2) Represents an expression tree that may be recursively evaluated. The expression is evaluated with a hashmap of key value pairs for the values of variables in the expression

4.2.3.1 Permutation

Description: Generates each permutation of variables in the expression which may be used by the terminal interface to generate a truth table

4.2.4 Error Reporting

Description: Display a description of an error when one is handled. These errors include, missing operand, unrecognized symbol, parenthesis unclosed or mismatched , variable being defined in terms of itself, an empty expression, double operator, variables not assigned, inconsistent characters, operator after an operand, and invalid characters.

4.2.5 Terminal Interface

Description: Allows for commands, methods, and expressions to be inputted which results in valid outputs being displayed. The methods allow for variables to be set to truth values, which allows them to be valid operands. The expressions include operators, parentheses, and operands with at least one operator and two operands (one if the operator is not(!)).

5. Interface Description

5.1 Major Entity Interfaces and Screen Formats

We are using a command line interface to enter valid inputs and to display valid outputs. The screen format will fit on any size screen and will depend on the window size. The window may be shrunk but it will display in the default best way for that window size.

5.2 Valid Inputs

A valid input includes two operands and an operator, but only one if the operator is not (!). Each operator must be between two operands, or just before an operand if the operator is not. Some valid inputs include: T&F , (T | F), ((F @ T) $ (T | (F & F))) & (T & (T @ (!T)))

5.3 Valid Outputs

There are only 2 valid outputs, True and False, that result from the expression evaluated from a valid input. If the input is invalid, it will be handled and the correct error will be displayed. The above examples of inputs would have the following outputs respectively: False, True, False.

6. Size and Performance

This section outlines the critical size and performance characteristics of the Boolean Logic Simulator that significantly influence its architectural design, as well as the specific performance targets that the system aims to meet. These aspects are pivotal in ensuring the simulator is both efficient and effective in its function.

Dimensioning Characteristics

Code Size: The simulator is designed to be lightweight, with a focus on minimizing the codebase to improve maintainability and reduce complexity. The use of C++ allows for compact and efficient code, especially with the use of templates and inline functions for operations that need to be fast, such as logical evaluations and expression parsing.

Data Structures: The choice of data structures is crucial for managing the internal representation of Boolean expressions and circuits. Efficient structures such as binary trees for expression parsing and linked structures for representing circuits ensure that operations are performed optimally. These structures are chosen to facilitate quick traversal and modification, which are essential for the real-time evaluation of complex expressions.

Memory Usage: Memory efficiency is a key architectural consideration, especially given the potential complexity of the logic circuits the simulator needs to handle. The software is designed to use memory judiciously, with structures that expand dynamically but only as needed to keep the footprint small. This is critical for ensuring the simulator can run on machines with limited resources, typical in educational environments.

Performance Constraints

Response Time: The simulator is expected to evaluate Boolean expressions and generate truth tables in real-time. The target response time for the evaluation of any single expression is less than one second for typical use cases, ensuring a seamless user experience.

Concurrency: In scenarios where the simulator is used in a classroom setting with multiple instances running concurrently, the architecture should ensure that performance does not degrade. This involves efficient management of CPU and memory resources.

Robustness: The system must consistently perform under various conditions without errors. This includes handling incorrect inputs gracefully and maintaining performance stability even with complex or non-optimized Boolean expressions.

Architectural Implications

To meet these size and performance constraints, the architecture employs several strategies:

Optimized Algorithms: Algorithms for parsing and evaluating expressions are optimized for speed and efficiency, using best practices in computational logic and data manipulation.

Resource Management: Effective resource management mechanisms are in place to ensure that memory and processing power are allocated dynamically based on the current load, enhancing the system's ability to perform reliably across different hardware configurations.

7. Quality

This section discusses how the software architecture of the Boolean Logic Simulator enhances various non-functional attributes of the system, such as extensibility, reliability, portability, and additional critical factors like security and privacy. These characteristics are crucial in ensuring that the simulator not only meets functional requirements but also adheres to broader standards of software quality and user expectations.

Extensibility
The software architecture is designed with modularity in mind, using object-oriented programming principles to encapsulate different functionalities like parsing, evaluation, and user interface management. This approach allows

for easy expansion and modification of features. For instance, adding new logical operators or more complex gate simulations can be accomplished without altering the core components of the system. This modular design ensures that future enhancements, such as support for more complex digital circuits or additional logical operations, can be integrated seamlessly.

Reliability
Reliability is achieved through a robust error handling framework and comprehensive unit testing. The simulator includes detailed exception handling that anticipates potential user errors, such as invalid syntax or unsupported operations, ensuring the system remains stable under various conditions. Each module is accompanied by a suite of unit tests that verify both common and edge case scenarios, guaranteeing that each component functions correctly across updates and modifications. This systematic testing approach minimizes bugs and maintains the integrity of evaluations.

Portability
The application is developed in C++, which is widely supported across different operating systems including Windows, macOS, and Linux. The use of standard C++ libraries and minimal dependency on external libraries enhances the simulator's portability, allowing it to run on a broad range of computer hardware without requiring specific adaptations. This cross-platform compatibility is essential for educational software, ensuring that all students, regardless of their operating system, can access and utilize the simulator.

Additional Factors
Performance: Efficiency in processing and memory usage is critical, especially for handling complex circuits and large expressions. The architecture is optimized for performance, ensuring quick response times and minimal latency in evaluations.
Usability: The design of the CLI focuses on user experience, with clear instructions, feedback mechanisms, and error messages that guide the user through correct usage and help them understand Boolean logic more effectively.
Overall, the architecture of the Boolean Logic Simulator is crafted to ensure that the system is not only functional but also robust, extendable, user-friendly, and secure. These qualities make it a reliable and valuable tool for educational purposes.