SQUISHY CATS

LogiCalc
Software Architecture Document

Version 1.0

Revision History

Date	Version	Description	Author
04/01/2024	0.1	Document Initialization	Olivia
04/14/2024	1.0	Finalize Version 1.0	Olivia, Sam, Aniketh

Table of Contents

Software Architecture Document

1. Introduction

This document, Software Architecture, details the architectural overview of a CLI Boolean Algebra Calculator. The program, which allows users to input different Boolean Algebra expressions, is created to fulfill the requirements of the EECS 348 course project.

1.1 Purpose

The purpose of the Software Architecture Document is to provide a low-level description of the LogiCalc system, providing insight into the structure and design of each component. This document provides a comprehensive architectural overview of the system, using two different architectural views to depict different aspects of the system. It is intended to capture and convey the significant architectural decisions made on the system.

1.2 Scope

This Software Architecture Document provides an architectural overview of the LogiCalc system. The LogiCalc system is being developed by Squishy Cats to support the computation of Boolean expressions as part of our EECS 348 course project.

1.3 Definitions, Acronyms, and Abbreviations

UML: Universal Modeling Language

IO: input/output

AST: abstract syntax tree

1.4 References

For the Software Architecture Document, the list of referenced and supporting artifacts includes:

- Team roles
- Software Development Plan
- Iteration Plans
- Software Requirements Specifications
- Test Cases
- User Manual

1.5 Overview

This Software Architecture document contains the following information:

Architectural Representation — Describes the difference between views

Architectural Goals & Constraints — Describes the architectural goals and constraints

Use-Case View — Use-case scenarios and realizations

Logical View — Describes the architecturally significant parts of the design model

Interface Description — Describes the user interfaces

Quality — Describes how the architecture contributes to all capabilities

2. Architectural Representation

This document presents the architecture as two distinct views: A Logical View, and a User-Case View. Each view describes the program flow and structure in plain English and with the use of some Universal Modeling Language (UML) diagrams.

2.1 Logical

- Tokens Module: This is responsible for tokenizing a string of characters into a list of Tokens. This is a more meaningful representation of the expression we want to evaluate.
- Parser Module: This is responsible for parsing the list of Tokens into an even more meaningful Abstract Syntax Tree. This will help us correctly evaluate expressions.
- Interpreter Module: This interprets the meaningful AST and collapses it down to a single value.
- IO Module: This handles generic user input and displays the result of the Interpreter to the user. It also handles and displays any errors that may occur within the previous three modules.

2.2 Use-Case

- System Module: This will be responsible for handling user input and displaying output.
    - IO Sub-Module: This is a part of the system that will handle input and output.
    - Evaluator Sub-Module: This will take input and evaluate it into the correct output
- Actor Module: This is the user, a human person, that will be interacting with the system.

3. Architectural Goals and Constraints

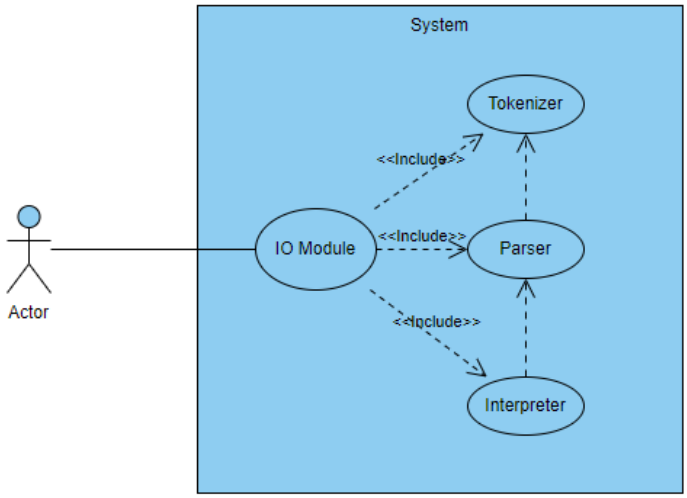
3.1 Architectural Goals

- Functionality: The system must parse and evaluate Boolean expressions in the proper order of operations.
- Reliability: The system should handle any errors that may occur and deliver a correct and consistent output.

3.2 Architectural Constraints

- Use of off-the-shelf products: The system should use the C++ programing language with its standard library of modules.
- Security: The program should not arbitrarily execute code that may be present in user input.

4. Use-Case View



Actor: User

Description: The user uses the program to evaluate a Boolean logic expression.

1. The program asks the user to input their desired Boolean logic expression
2. The user types in their desired Boolean logic expression
3. The system evaluates the expression and displays the end result to the user
    i. The program outputs an error if the expression is not a valid Boolean expression (i.e. mismatched parentheses, unknown operator/operand, empty input, not 1 operator between 2 operands, operands in invalid position, etc.)

4.1 Use-Case Realizations

Actor: User

Description: The program takes a Boolean logic expression from the user and displays the output to the user.

1) Display the prompt asking the user for a Boolean logic expression
2) Use I/O subsystem to read expression from the user
3) Give the input to the evaluation subsystem, which creates tokens from the string input
4) In the same subsystem, parse through the tokens and convert them into an AST
5) Generate the final output from the parser output (either a single “True” or “False” value)
6) Send the output to the I/O subsystem, and display the output to the user
    i. The miscellaneous subsystem is used to identify and send an error to the I/O subsystem if a parsing error occurs

5. Logical View

The logical view of the program breaks the system into two key subsystems, the Input/Output Subsystem and the Evaluation Subsystem. Additionally, there is a Utility package which assists in common classes and methods across all subsystems.

5.1 Overview

The Input/Output subsystem consists of an IO class which accepts user input, applies any normalization or basic interactions with the user. It then hands control off to the evaluation system to generate an output. The output is finally given back to the IO class and displayed to the user.

The Evaluation subsystem consists of three main parts, the tokenizer, parser, and evaluator. The tokenizer takes raw input from the IO class and transforms it into a state the parser can understand. The parser then takes the output of the tokenizer and further transforms it into a meaningful state (AST). The output of the parser is finally fed to the evaluator to collapse the input into a singular value.

5.2 Architecturally Significant Design Modules or Packages

5.2.1 IO Subsystem

Package: IO

Class - IO:

Description: Handle user input and output and any errors that may occur.

Methods: run(), check_exit()

Attributes: N/A

5.2.2 Evaluation Subsystem

Package: Tokens

Class - Token:

Description: Contains useful information about a token

Methods: Operator overloads for (==) and (!=)

Attributes: type, position, lexeme, value (optional)

Function - tokenize:

Description: Turn a string of characters into a list of Tokens with useful information

Parameters: line (string)

Returns: list<Token>

Raises: syntax error

Package: Ast

Class - Expr:

Description: Base abstract class to hold information about other AST nodes.

Methods: N/A

Attributes: N/A

Class - Literal:

Description: A literal value node (T or F)

Methods: N/A

Attributes: value (boolean)

Class - Binary:

Description: A binary operation (&, |, @ $)

Methods: N/A

Attributes: left (Expr), op (Token), right (Expr)

Class - Unary:

Description: A single operand prefix operation

Methods: N/A

Attributes: op (Token), right (Expr)

Class - Group:

Description: An expression grouped by parenthesis

Methods: N/A

Attributes: expression (Expr)

Package: Parser

Class – Parser:

Description: Parse a list of Tokens into an AST (Abstract Syntax Tree)

Methods: prev_tkn(), curr_tkn(), primary(), unary(), and(), or(), nand(), xor(), parse()

Attributes: pointer (int) = 0

Package: Interpreter

Class – Parser:

Description: Collapse an AST into a single value (evaluate the expression)

Methods: evalUnary(), evalBinary(), eval()

Attributes: N/A

5.2.3 Miscellaneous Subsystem

Package: Util

Class – ParsingError:

Description: Contain information about a parsing error, if one occurs.

Methods: N/A

Attributes: N/A

6. Interface Description

The interface will consist of a command-line interface that will ask the user to input their desired Boolean algebra expressions.

A valid input will consist of:

- "T" or "F," which stand for True or False
- One valid logical operator in between each "T" and "F" value
    - The NOT operator is excluded from this since it only goes before the operand and is optional for each operand
- Parentheses surrounding certain parts of the Boolean algebra expression (optional)
    - If parentheses are used, there must be an opening parenthesis and closing parenthesis for each set of parentheses.

If any of these requirements are not met, the program will output an error. Otherwise, the program will evaluate the expression and displays either "True" or "False" in the command-line interface, depending on the value of the expression.

7. Quality

The following quality requirements should be met within the final product:

1. The program consistently and correctly evaluates all Boolean expressions.
2. The program is portable across machines, and Unix-based operating systems.
3. The program is modularly designed such that maintenance, feature additions, and middleware can be relatively seamlessly added.
4. Be able to provide meaningful error messages.