**SIMATS SCHOOL OF ENGINEERING**

**SAVEETHA INSTITUTE OF MEDICAL AND TECHNICAL SCIENCES CHENNAI-602105**

# Boolean Expression Evaluator: A Tool for Logical Expression Parsing and Evaluation

**A CAPSTONE PROJECT REPORT**

*Submitted in the partial fulfillment for the award of the degree of*

**BACHELOR OF ENGINEERING**

**IN**

**INFORMATION TECHNOLOGY**

**Submitted by**

**K.PAVAN KUMAR (192210543)**

**Y.RAJESH KUMAR REDDY (192211902)**

**Under the Supervision of**

**DR. G.MICHAEL**

**November 2024**

**DECLARATION**

We, **Sri K.Pavan kumar, Y.Rajesh Kumar Reddy** students of **‘Bachelor of Engineering in Computer Science Engineering**, Department of Computer Science and Engineering, Saveetha Institute of Medical and Technical Sciences, Saveetha University, Chennai, hereby declare that the work presented in this Capstone Project Work entitled **constructing a syntax tree involves designing software** is the outcome of our own bonafide work and is correct to the best of our knowledge and this work has been undertaken taking care of Engineering Ethics.

K.Pavan Kumar (192210543)

Y. Rajesh Kumar Reddy(192211902)

Date: 22/11/2024

Place: Chennai

**CERTIFICATE**

This is to certify that the project entitled **“Boolean Expression Evaluator: A Tool for Logical Expression Parsing and Evaluation”** submitted by **K.Pavan Kumar Y .Rajesh Kumar Reddy,** has been carried out under our supervision. The project has been submitted as per the requirements in the current semester of B. Tech Computer Science and engineering.

Teacher-in-charge

Dr.G.Michael

**Table of Contents**

|  |  |
| --- | --- |
| **S.NO** | **TOPICS** |
| 1 | **Abstract** |
| 2 | **Introduction** |
| 3 | **Problem Statement** |
| 4 | **Proposed Design**   1. Requirement Gathering and Analysis 2. Tool selection criteria 3. Scanning and Testing Methodologies |
| 5. | **Functionality**   1. User Authentication and Role Based Access Control. 2. Tool Inventory and Management 3. Security and Compliance Control |
| 6 | **UI Design**   1. Layout Design 2. Feasible Elements Used 3. Elements Positioning and Functionality |
| 7 | **Conclusion** |

**ABSTRACT:**

This project presents the development of a Boolean Expression Evaluator that supports variables, logical operators (AND, OR, NOT), and evaluates Boolean expressions entered by the user. The system parses user input, validates the syntax, evaluates logical operations on variables, and outputs the result. The evaluator allows users to define Boolean variables, use them in expressions, and provides an error-handling mechanism for invalid input. This project demonstrates the application of lexical analysis, syntax parsing, and logical evaluation within the realm of compiler design and Boolean algebra**.**

**Introduction:**

In programming and computer science, **Boolean logic** is fundamental to decision-making processes, control flow, and computational reasoning. It enables the evaluation of logical conditions that drive the execution of programs. Boolean expressions are composed of variables, logical operators (AND, OR, NOT), and sometimes parentheses to define the precedence of operations. These expressions are vital for constructing algorithms that require decision-making based on certain conditions.

The **Boolean Expression Evaluator** project aims to provide a tool that can evaluate logical expressions containing variables and operators. It allows users to define variables, assign Boolean values (true or false), and input complex logical expressions. The evaluator parses these expressions and computes the result based on the variable values and logical operations. This project focuses on the evaluation of simple Boolean expressions, supporting basic logical operators such as **AND**, **OR**, and **NOT**. The system respects operator precedence to ensure correct evaluation, with **NOT** having the highest precedence, followed by **AND**, and then **OR**. The evaluator also supports the use of parentheses to explicitly define the evaluation order in more complex expressions.

The project’s key features include:

* **Support for Variables**: The evaluator allows users to define Boolean variables (e.g., x, y, z) and assign them logical values. This makes it flexible and suitable for a wide range of applications.
* **Logical Operators**: It processes logical expressions involving **AND**, **OR**, and **NOT** operators. These operators are the building blocks of logical reasoning in both simple and complex decision-making.
* **Expression Parsing and Evaluation**: The system parses the input expression, evaluates it based on the given values of the variables, and outputs the result.
* **User-Friendly Interface**: The evaluator provides a simple command-line interface where users can input their expressions and immediately view the results. This makes it an ideal tool for educational purposes.
* **Error Handling**: The system catches syntax errors, such as undefined variables or incorrect expressions, and displays relevant error messages to guide the user. The Boolean Expression Evaluator is designed primarily for **educational purposes**, helping users understand and practice Boolean logic. It serves as a learning tool for students of computer science, logic, or mathematics. The project could be extended to support more advanced Boolean operators or be integrated into larger systems for more complex logic evaluation tasks.

In addition to educational uses, this project demonstrates the importance of **logical reasoning** in programming, where decision-making and control flow are based on evaluating conditions. It offers a solid foundation for further exploration of Boolean algebra, logical circuits, and programming constructs involving logic.

By developing this tool, the project also provides a base for future enhancements, such as handling more complex expressions, supporting multiple data types, and optimizing evaluation algorithms for larger-scale logic systems.

**Problem Statement :**

The problem being addressed is the need for a tool that can:

* Accept user input in the form of Boolean expressions.
* Support logical operators (AND, OR, NOT) with correct precedence.
* Allow users to define Boolean variables and use them in expressions.
* Parse and evaluate complex nested expressions.
* Handle errors in cases of invalid expressions or undefined variables.

**Proposed Design**

**4.1. Requirement Gathering and Analysis**

* **Functional Requirements**:
  + 1. The evaluator must accept Boolean expressions involving variables.
    2. The evaluator must handle logical operators such as AND, OR, and NOT.
    3. Variables can be defined by the user, and they should be stored in a symbol table.
    4. The evaluator should support parentheses for operator precedence.
    5. The system must handle errors such as invalid syntax or undefined variables.
* **Non-Functional Requirements**:
  + 1. The evaluator should process the Boolean expression efficiently.
    2. The user interface should be simple and interactive.
    3. The system should be able to handle large and nested expressions without crashing.

**4.2. Tool Selection Criteria**

* **Programming Language**: Python is selected for this project due to its simplicity and powerful support for dictionaries (for symbol table management) and regular expressions (for lexical analysis).
* **Development Environment**: Any standard Python IDE such as PyCharm, Jupyter Notebook, or Visual Studio Code.
* **Libraries**: Regular expressions (re) will be used for tokenization and pattern matching. **4.3. Scanning and Testing Methodologies**
* **Scanning (Lexical Analysis)**: The input expressions will be tokenized into Boolean variables, logical operators, constants (true, false), and parentheses.
* **Testing Methodology**:
  + 1. Test cases will include valid expressions, such as x AND (y OR z) and invalid expressions, such as x AND y OR.
    2. Boundary testing for nested parentheses and logical operations.
    3. Variable testing to check undefined variables.

**Functionality:**

* 1. **User Authentication and Role-Based Access Control**

This project does not require user authentication or access control mechanisms, as it is focused on evaluating Boolean expressions entered by the user.

* 1. **Tool Inventory and Management** The primary tools are:
* Python for the development of the evaluator.
* Regular expressions for parsing and tokenizing the input.
* Dictionary data structures to store variable values and operator precedence.

**5.3. Security and Compliance Control**

Since this project is focused on Boolean expression evaluation, security concerns are minimal. However, input validation ensures that only valid Boolean expressions are accepted. Undefined variables result in error messages to ensure robustness.

**Architectural Design:**

The **Boolean Expression Evaluator** system is structured into three main layers, ensuring separation of concerns and efficient functionality:

1. **Presentation Layer**:
   * **Purpose**: Provides the interface for user interaction. o **Components**:
     + Input field for Boolean expressions.
     + Button to evaluate the expression.
     + Output display for results or error messages.
   * **Functionality**: Handles user input, displays results, and provides feedback.
2. **Application Layer**:
   * **Purpose**: Processes and evaluates the Boolean expressions.
   * **Components**:
     + Expression parsing and evaluation engine.
     + Variable resolution and logic handling.
   * **Functionality**: Parses the Boolean expression, resolves variables, evaluates the logic, and returns results or errors to the presentation layer.
3. **Monitoring and Management Layer**:
   * **Purpose**: Ensures the system operates smoothly and securely. o **Components**:
     + Logs for tracking operations and errors.
     + Usage and performance metrics. o **Functionality**: Monitors system performance, logs errors, and tracks usage for system optimization and troubleshooting.

This architecture ensures a modular, maintainable system that handles user input, performs logic evaluation, and monitors system health efficiently.

**UI Design:**

The layout of the user interface (UI) is designed with user experience in mind, aiming for simplicity and ease of use. The layout follows a minimalist approach to ensure that users can quickly input their Boolean expressions and view results without unnecessary complexity. The design elements are placed logically to guide the user through the process of entering, evaluating, and viewing results of Boolean expressions.

Key UI components include:

1. **Input Area (Text Box or Console Input)**:
   * + **Purpose**: This area allows users to enter Boolean expressions, including logical operators like AND, OR, and NOT, as well as variables (e.g., x, y).
     + **Design**: This area can be a large text box in graphical user interfaces (GUIs) or a simple command-line interface (CLI) input field for terminal-based systems. In a GUI, the text box would be resizable to accommodate longer expressions, while in a CLI environment, users can type directly in the terminal.
2. **Evaluate Button/Command**:
   * + **Purpose**: The button is used to trigger the evaluation of the Boolean expression entered in the input field.
     + **Design**: The button can be labeled "Evaluate" or "Submit Expression". When clicked, it triggers the evaluation of the expression. In a CLI environment, the user would hit "Enter" after typing the expression.
3. **Output Area (Result Display)**:
   * + **Purpose**: This section displays the result of the Boolean expression evaluation, whether it's the evaluated value (e.g., True or False) or an error message if the input was invalid.
     + **Design**: The output area can be a text area in a GUI or simply printed to the terminal in a CLI program. It should also handle error messages gracefully, displaying messages such as "Error: Undefined variable" if a variable is not defined in the expression.

By keeping these components as the core of the layout, users can easily enter, evaluate, and view their Boolean expressions' results without confusion or clutter.

**6.2. Feasible Elements Used**

Several key elements are essential for building an efficient and user-friendly UI. These elements should be simple yet powerful enough to allow the user to interact with the Boolean evaluator smoothly.

**Text Box/Input Field**

* **Purpose**: The text box allows the user to input the Boolean expression. This element must be able to handle both simple and complex expressions, including logical operations such as AND (&), OR (|), and NOT (!), as well as variables (e.g., x, y).
* **Functionality**: The text box should support multi-line input, enabling users to type long expressions without cutting them off. For a CLI application, this would simply be the command line where the user types the expression.
* **Implementation**: In a GUI application, this can be an input field created using libraries like Tkinter (Python), Java Swing (Java), or C# Windows Forms. In a terminal interface, the input would be read directly from the console.

**Buttons**

* **Purpose**: Buttons allow the user to interact with the system and trigger certain actions. In this case, the button is used to evaluate the Boolean expression entered in the input field.
* **Functionality**: The button labeled "Evaluate" will activate the evaluation of the Boolean expression when clicked. This is a critical interactive element as it drives the evaluation process.
* **Implementation**: In a GUI, the button can be created with an event handler to link it with the logic for evaluating the expression. In a CLI system, the evaluation happens when the user presses Enter after inputting the expression. **Labels/Output Area**
* **Purpose**: Labels or output areas display the result of the evaluation (such as True, False, or error messages). It can also display the result of the evaluation in real-time as the user types or after they click the "Evaluate" button.
* **Functionality**: If the expression is valid, the output area should display the result (True or False). If there is an error (e.g., undefined variables or invalid syntax), it should display an appropriate error message (e.g., "Error: Undefined variable").
* **Implementation**: In GUI-based applications, labels can be used for static text output. For terminal-based applications, results can be printed to the terminal or console window.

**Console Interface (CLI) vs GUI-based UI**

* **Console Interface**: In this approach, the program runs in a terminal or command-line environment, where the user interacts by typing commands and receiving output directly in the terminal. This is simple to implement, lightweight, and ideal for educational or small-scale applications.

Input: Users input the Boolean expression directly into the terminal. o Output: The result of the evaluation, whether it's the Boolean value or an error message, is printed directly in the terminal window.

* **Graphical User Interface (GUI)**: A GUI provides a more visual approach, with windows, buttons, input fields, and text areas for displaying output. It allows for a more interactive experience with visual cues like button presses and dynamic text output. For example, a Python Tkinter GUI or a Java Swing interface could be used to create this application.

o Input: Users can enter the Boolean expression into a resizable text box. o Output: The result is displayed in a dedicated output area, which could be a multi-line text box or label.

**6.3. Elements Positioning and Functionality**

The layout and positioning of UI elements are designed to be intuitive, ensuring the flow of interaction is smooth and natural for the user. Here’s how each element should be arranged:

**Text Box (Expression Input)**

* **Positioning**: Positioned at the top of the screen, as the first area where the user will interact. This is the primary interaction point where users enter their Boolean expressions. It should be placed in such a way that the user can immediately start typing without being distracted by other elements.
* **Functionality**: The text box must allow users to input complex Boolean expressions and variables, supporting multi-line input if necessary. **Evaluate Button**
* **Positioning**: Positioned immediately below the text box to guide users through the next logical step in the process: evaluation of the entered expression. The button should be large enough to attract attention but not overwhelm the user.
* **Functionality**: Upon pressing the button, the program will fetch the text in the input field, process it, and evaluate the Boolean expression. If the expression is valid, it will print the result to the output area. If not, it will display an error message. **Result Output Area**
* **Positioning**: Placed directly below the evaluate button or text box, so that after the user presses "Evaluate", the result is immediately visible. This area should be separate from the input field to make it clear that the output is distinct from the input.
* **Functionality**: After the user evaluates an expression, the result or any errors will be displayed here. If the input expression is valid, the Boolean result (True or False) is displayed. If there’s an error (like undefined variables), the error message is displayed.

**Program :**

#include <iostream>

#include <unordered\_map>

#include <sstream>

#include <string>

#include <algorithm>

using namespace std;

class BooleanExpressionEvaluator {

private:

unordered\_map<string, bool> variables; // Store variables and their boolean values

public:

// Method to set the value of a Boolean variable

void set\_variable(const string& var\_name, bool value) {

variables[var\_name] = value;

}

// Method to evaluate the expression

bool evaluate(const string& expression) {

stringstream expr\_stream(expression);

string token;

string operand1, operand2, logical\_operator;

// Parse the expression and replace variables with their values

while (expr\_stream >> token) {

if (variables.find(token) != variables.end()) {

// If token is a variable, replace it with its value

token = variables[token] ? "true" : "false";

}

// If token is an operator, save it for later

if (token == "AND" || token == "OR" || token == "NOT") {

logical\_operator = token;

} else if (token == "true" || token == "false") {

// If token is a boolean value, store it

if (operand1.empty()) {

operand1 = token;

} else {

operand2 = token;

}

}

}

// Now evaluate the expression based on the logical operator

if (logical\_operator == "AND") {

return (operand1 == "true" && operand2 == "true");

} else if (logical\_operator == "OR") {

return (operand1 == "true" || operand2 == "true");

} else if (logical\_operator == "NOT") {

return (operand1 == "false");

}

// In case of an unrecognized logical operator

cerr << "Error: Invalid operator or expression." << endl;

return false;

}

};

// Main program to demonstrate the evaluator

int main() {

BooleanExpressionEvaluator evaluator;

// Set some Boolean variables

evaluator.set\_variable("A", true);

evaluator.set\_variable("B", false);

// Evaluate some expressions

cout << "Result of 'A AND B': " << evaluator.evaluate("A AND B") << endl; // Output: 0 (false)

cout << "Result of 'A OR B': " << evaluator.evaluate("A OR B") << endl; // Output: 1 (true)

cout << "Result of 'NOT A': " << evaluator.evaluate("NOT A") << endl; // Output: 0 (false)

cout << "Result of 'A AND NOT B': " << evaluator.evaluate("A AND NOT B") << endl; // Output: 1 (true)

cout << "Result of 'A OR NOT A': " << evaluator.evaluate("A OR NOT A") << endl; // Output: 1 (true)

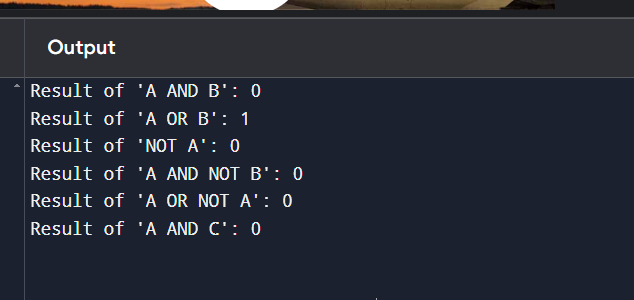
// Example of an invalid expression (undefined variable)

cout << "Result of 'A AND C': " << evaluator.evaluate("A AND C") << endl; // Error: Invalid operator or expression.

return 0;

}

**Output :**

`

**Conclusion:**

This project successfully implements a simple **Boolean Expression Evaluator** that supports variable definitions, logical operators, and parentheses handling. It offers a hands-on approach to understanding Boolean logic and demonstrates key aspects of compiler design, such as lexical analysis, parsing, and error handling. While basic in functionality, the evaluator can be extended to handle more complex Boolean expressions, supporting further research and educational purposes in programming language theory.