Faculty of Engineering Sciences and Technology

Shravan Asati, Supal Vasani

Discrete Mathematics

Pensieve - A Truth Table Generator

April 2025

Department of Computer Science and Engineering

Abstract

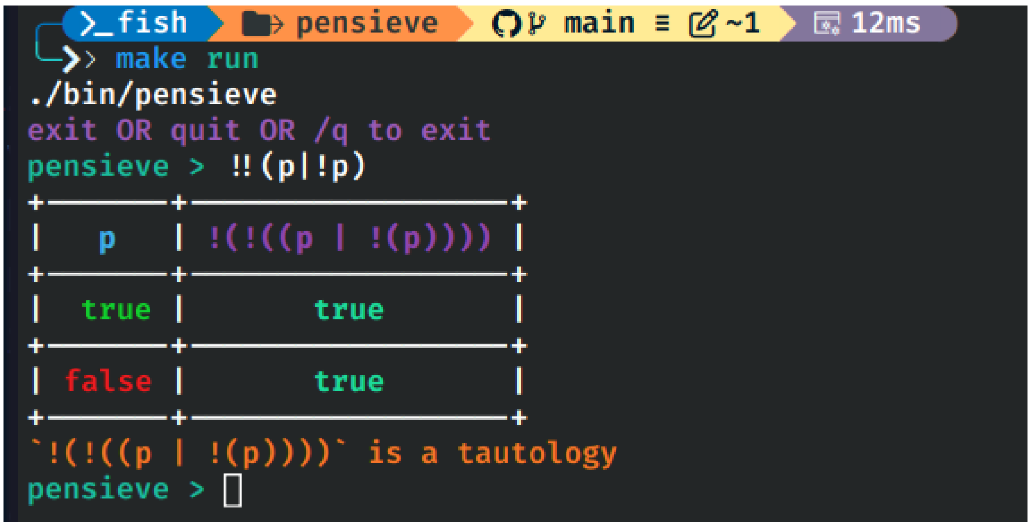
Pensieve is a command-line-based truth table generator for Boolean logic expressions, written in C++. It serves as a utility for students, educators, and developers to visualize and verify logical expressions by constructing complete truth tables.

The application accepts logical expressions using standard Boolean operators (AND, OR, NOT, XOR, implication, and biconditional) and handles any number of Boolean variables. The parser is designed to detect and report syntax errors, making it robust for educational and debugging purposes.

The primary goals of the project are to:

1. Provide a correct and efficient implementation of a truth table generator
2. Support expressions with any number of Boolean variables
3. Detect and pinpoint errors in invalid expressions
4. Render aesthetically pleasing and readable truth tables in the terminal
5. Support check for logical equivalence

The project's source code is open-sourced under the MIT License and can be found here.



```
>_fish  pensieve  main ≡ ~1  12ms
 >> make run
./bin/pensieve
exit OR quit OR /q to exit
pensieve > !!(p|!p)
+-------+-------------------+
|   p   | !(!((p | !(p)))) |
+-------+-------------------+
|  true |       true        |
+-------+-------------------+
| false |       true        |
+-------+-------------------+
`!(!((p | !(p))))` is a tautology
pensieve >
```

Figure 1: Demonstration of Pensieve in action

Acknowledgements

We want to express our sincere gratitude to our guide, Prof. Ashlesha Bhise, for their invaluable support, guidance, and encouragement throughout the development of this project.

A special thanks to all our team members for their dedication, collaboration, and continuous effort at every stage of the project.

Lastly, we would like to acknowledge the open-source community and the developers of the tools and libraries we used, without which this project would not have been possible.

- **linenoise** – Used in the Read Evaluate Print Loop (REPL), provides an interactive command line interface, manages input history, and autocompletion.
- **tabulate** – Used to render aesthetically pleasing tables.

Contents

List of Figures

Listings

Chapter 1

Introduction

In the domain of formal logic and computer science education, visualizing Boolean expressions through truth tables serves as a fundamental method for understanding logical operations and verifying the correctness of complex expressions. Despite the importance of this concept, accessible tools that provide comprehensive truth table generation with robust error handling remain surprisingly scarce. This gap in available utilities prompted the development of **Pensieve**, a specialized command-line application designed to generate, analyze, and display truth tables for Boolean expressions of arbitrary complexity.

Named after the magical device from literature that extracts and examines thoughts, Pensieve aims to make logical reasoning more transparent by transforming abstract Boolean expressions into concrete, visual representations. The application supports the complete set of standard logical operators—conjunction (AND), disjunction (OR), negation (NOT), exclusive disjunction (XOR), material implication, and biconditional operators—thereby covering the entire spectrum of propositional logic constructs used in academic and practical contexts. Additionally, Pensieve can deduce whether the given statement is a tautology or a contradiction, along with checking the logical equivalence of two statements.

1.1 Project Motivation

The motivation behind Pensieve stems from several observations regarding existing truth table generators:

- Many available tools impose limitations on expression complexity or the number of variables
- Error reporting in existing utilities often lacks specificity, hampering the debugging process
- Many available tools do not support checking the logical equivalence of two statements
- Terminal-based solutions frequently sacrifice readability for functionality
- Educational tools should allow students to experiment freely with logical expressions

These considerations guided the development of Pensieve as a tool that combines computational efficiency with educational value. By implementing Pensieve in C++, the application achieves the performance necessary to handle expressions with numerous variables while maintaining a responsive user experience.

1.2 Core Functionality

At its core, Pensieve encompasses several key components that work in concert to deliver its functionality:

1. **Expression Parser**: Pensieve has a sophisticated lexer and a shunting-yard parser that processes user input according to formal grammar rules for Boolean expressions, constructs the reverse polish notation, and identifies syntax errors with precise positional information
2. **Truth Table Generator**: An algorithm that systematically evaluates the parsed expression for all possible combinations of variable truth values, effectively generating a complete truth table
3. **Rendering Engine**: A sophisticated output formatter that presents truth tables with clear column alignment, appropriate headers, and visual separators to enhance readability in the terminal environment
4. **Error Management System**: A comprehensive error detection and reporting mechanism that provides contextual information about syntax errors, helping users identify and

correct mistakes in their expressions. This is part of the lexer.

5. **Logical Equivalence Checker**: Pensieve supports comparing multiple Boolean expressions by evaluating their truth tables and verifying if they produce identical outputs for all possible input combinations. This allows users to confirm whether two or more expressions are logically equivalent.

1.3 Educational Applications

Pensieve was designed with particular attention to educational contexts, where it can serve multiple purposes:

- As a learning aid for students studying propositional logic, discrete mathematics, or digital circuit design
- As a teaching tool for instructors demonstrating the evaluation of logical expressions
- As a verification utility for checking the correctness of manually calculated truth tables
- As a reference implementation for computer science students learning about parsing and evaluation of domain-specific languages

1.4 Syntax and Symbol Definitions

In order to parse and evaluate Boolean expressions, Pensieve uses specific single-character symbols to represent logical operators. The table below lists all the supported logical operators along with their corresponding symbols used in the application:

Operator	**Symbol Used**
AND	`&`
OR	`|`
NOT	`!`
XOR	`^`
IMPLICATION	`>`
BICONDITIONAL	`=`

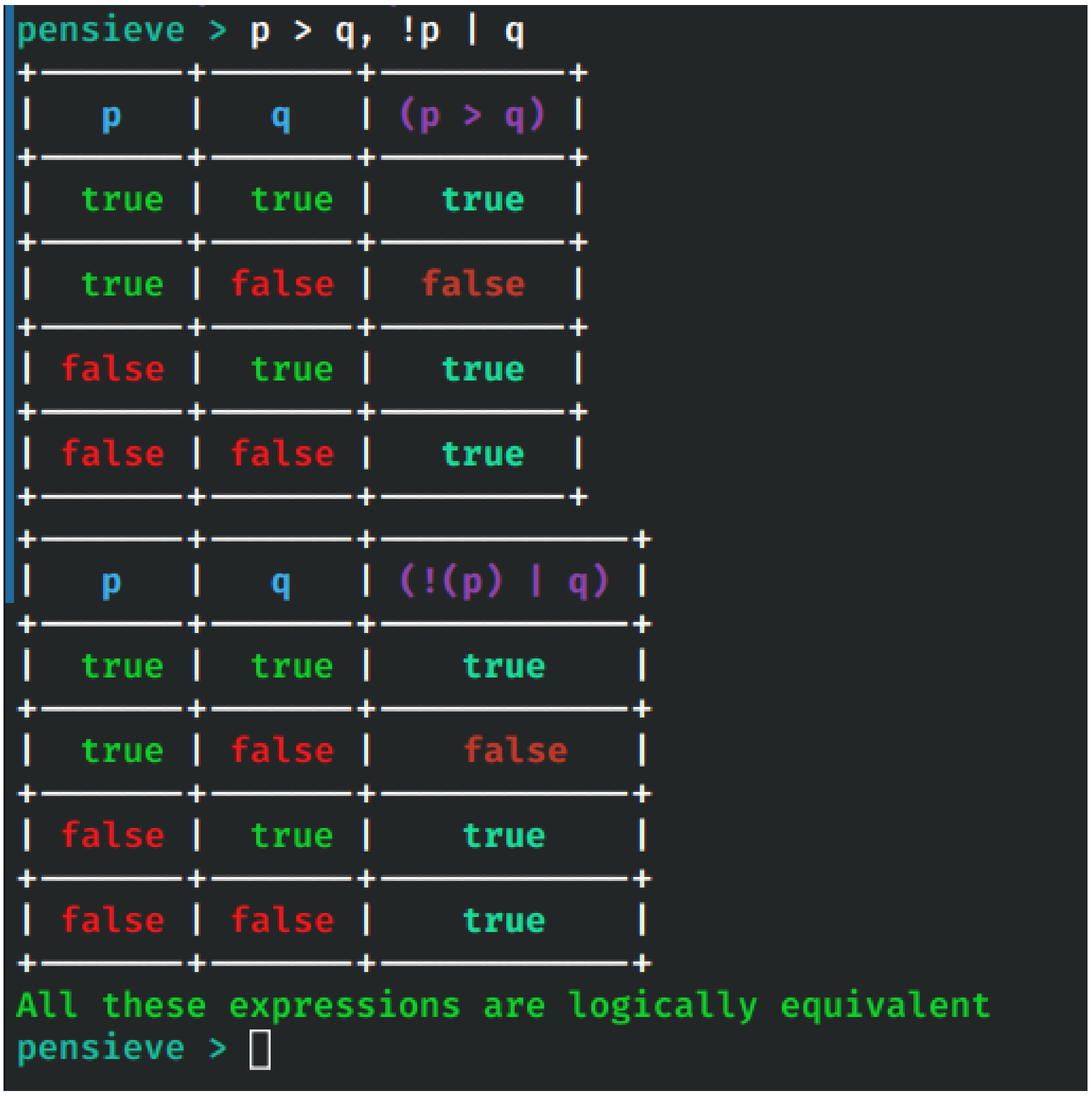
Table 1.1: Logical operators and their corresponding symbols in Pensieve

The choice of these symbols for the logical operators stems from their common use in popular programming languages.

1.5 Development Philosophy

The development of Pensieve adheres to several guiding principles:

- **Correctness**: Ensuring that the generated truth tables accurately represent the logical semantics of the input expressions
- **Robustness**: Handling edge cases gracefully and providing meaningful feedback for invalid inputs
- **Usability**: Creating an intuitive interface that requires a minimal learning curve for users familiar with Boolean logic
- **Extensibility**: Designing the codebase to accommodate future enhancements such as additional operators or output formats



```
pensieve > p > q, !p | q
+-------+-------+---------+
|   p   |   q   | (p > q) |
+-------+-------+---------+
| true  | true  |  true   |
+-------+-------+---------+
| true  | false |  false  |
+-------+-------+---------+
| false | true  |  true   |
+-------+-------+---------+
| false | false |  true   |
+-------+-------+---------+
+-------+-------+--------------+
|   p   |   q   | (!(p) | q)   |
+-------+-------+--------------+
| true  | true  |    true      |
+-------+-------+--------------+
| true  | false |    false     |
+-------+-------+--------------+
| false | true  |    true      |
+-------+-------+--------------+
| false | false |    true      |
+-------+-------+--------------+
All these expressions are logically equivalent
pensieve > 
```

Figure 1.1: Logical Equivalence Check

As illustrated in Figure 1.1, Pensieve renders truth tables with a focus on clarity and readability, making complex logical expressions more accessible and understandable. The remainder of this document explores the technical architecture, implementation challenges, and future directions for the Pensieve project.

Chapter 2

Background

2.1 Propositional Logic Fundamentals

Propositional logic forms the mathematical foundation for logical reasoning with binary truth values. This section outlines the essential concepts underlying Pensieve's implementation.

2.1.1 Core Logical Operators

The fundamental operators in propositional logic include:

- **Negation** ($\neg$): Unary operator inverting truth values; $\neg p$ is true iff p is false
- **Conjunction** ($\wedge$): Binary operator yielding true iff both operands are true
- **Disjunction** ($\vee$): Binary operator yielding true iff at least one operand is true
- **Exclusive Disjunction** ($\oplus$): Binary operator yielding true iff exactly one operand is true
- **Implication** ($\rightarrow$): Binary operator representing conditional relationships; false only when antecedent is true and consequent is false
- **Biconditional** ($\leftrightarrow$): Binary operator yielding true iff both operands have identical truth values

These operators form the syntactic foundation for Pensieve's expression parser.

2.1.2 Truth Tables and Evaluation Methods

Truth tables systematically enumerate all possible variable assignments and corresponding expression outputs. For an expression with n variables, the table contains 2^n rows representing all possible truth value combinations.

Structure and Significance

A truth table consists of:

- Input columns for each unique variable
- Intermediate columns for subexpressions (optional but instructive)
- Final column(s) showing the evaluated result of the complete expression(s)

Truth tables serve critical functions in both theory and application:

- **Complete Evaluation**: They provide exhaustive analysis of expression behavior across all possible input combinations
- **Equivalence Verification**: Two expressions are logically equivalent if and only if their truth tables have identical output columns
- **Expression Classification**: Truth tables reveal whether expressions are tautologies, contradictions, or contingencies
- **Digital Circuit Design**: They specify the behavior of combinational logic circuits
- **Algorithmic Verification**: They validate the correctness of logical algorithms and decision procedures

Evaluation Process

The systematic evaluation of truth tables follows these steps:

1. **Variable Identification**: Extract all unique variables from the expression
2. **Input Enumeration**: Generate all 2^n possible truth value combinations
3. **Expression Parsing**: Transform the infix expression into a computationally efficient form (typically RPN)
4. **Systematic Evaluation**: For each row:

   (a) Assign truth values to variables according to the current row
   (b) Evaluate subexpressions in order of operator precedence
   (c) Compute the final expression value

5. **Result Analysis**: Analyze the output column to classify the expression

Example Truth Table

For the expression $p \rightarrow (q \vee \neg p)$:

p	q	$\neg p$	$q \vee \neg p$	$p \rightarrow (q \vee \neg p)$
T	T	F	T	T
T	F	F	F	F
F	T	T	T	T
F	F	T	T	T

This example demonstrates how complex expressions are evaluated by breaking them into manageable components and applying operator semantics systematically.

2.1.3 Operator Precedence Hierarchy

Unambiguous expression evaluation requires a strictly defined precedence hierarchy:

1. Parenthetical grouping (highest precedence)
2. Negation ($\neg$)
3. Conjunction ($\wedge$)
4. Disjunction ($\vee$)
5. Exclusive disjunction ($\oplus$)
6. Implication ($\rightarrow$)
7. Biconditional ($\leftrightarrow$) (lowest precedence)

This hierarchy determines evaluation order in complex expressions, eliminating syntactic ambiguity.

2.1.4 Reverse Polish Notation and Expression Parsing

RPN Fundamentals

Reverse Polish Notation (RPN) represents expressions by placing operators after their operands. This postfix notation offers several advantages:

- Elimination of parentheses and precedence rules
- Direct stack-based evaluation
- Unambiguous representation without grouping symbols
- Efficient computational processing

Shunting-Yard Algorithm Implementation

The Shunting-Yard algorithm transforms infix expressions to RPN through:

- Token-by-token scanning of the input expression
- Stack-based operator management according to precedence rules
- Output queue construction for the resulting RPN sequence

This algorithm forms the core of Pensieve's expression parser, enabling transformation of user-provided infix expressions into computationally efficient RPN representations.

For example, converting $(p \wedge q) \vee \neg r$ to RPN:

Token	**Action**	**Output Queue**	**Operator Stack**
(	Push to stack	empty	(
p	Add to output	p	(
$\wedge$	Push to stack	p	(, $\wedge$
q	Add to output	p, q	(, $\wedge$
)	Pop until (	$p, q, \wedge$	empty
$\vee$	Push to stack	$p, q, \wedge$	$\vee$
$\neg$	Push to stack	$p, q, \wedge$	$\vee, \neg$
r	Add to output	$p, q, \wedge, r$	$\vee, \neg$
end	Pop remaining	$p, q, \wedge, r, \neg, \vee$	empty

The resulting RPN expression $p\ q\ \land\ r\ \neg\ \lor$ can be efficiently evaluated using a stack-based algorithm.

2.1.5 Expression Classification

Boolean expressions can be categorized based on their truth table characteristics:

- **Tautologies**: Expressions evaluating to true under all variable assignments
- **Contradictions**: Expressions evaluating to false under all variable assignments
- **Contingencies**: Expressions evaluating to true under some assignments and false under others

Logical equivalence between expressions occurs when they yield identical truth values across all possible variable assignments, enabling expression simplification and transformation.

2.1.6 Computational Challenges

Truth table generation presents several algorithmic and implementation challenges:

- **Exponential Complexity**: The $O(2^n)$ growth rate for n variables necessitates efficient algorithms
- **Parsing Robustness**: Requirement for accurate tokenization and syntactic analysis
- **Evaluation Correctness**: Precise implementation of operator semantics and precedence rules
- **Error Handling**: Detection and reporting of syntactic and semantic errors
- **Memory Efficiency**: Optimization for large expressions with numerous variables
- **Output Formatting**: Presentation of potentially large truth tables in human-readable form

These challenges motivate Pensieve's design decisions, including the implementation of efficient parsing algorithms, comprehensive error handling, and formatted output generation.

Chapter 3

The Tokenizer

3.1 Definitions

In Pensieve, the user inputs a logical expression stored as an input inside a variable of type **string**. This string input by itself is not useful, it needs to be converted into some internal representation (a data structure) that will enable efficient operations on it.

The input needs to be parsed and classified into various symbols called **tokens**. In the context of Pensieve, these tokens would be the logical operators and variables.

The process of converting a string input into a stream of tokens is called **tokenization**.

The program which does performs tokenization is called a **tokenizer** or a **lexer**.

3.2 Tokens

As discussed in the Background (2) chapter, Pensieve uses the shunting-yard algorithm to parse the expressions into the Reverse Polish Notation (RPN), which makes it easier to evaluate the expressions systematically. Two important parts of the systematic and orderly evaluation of operators are **precedence** and **associativity**.

Precedence determines the order of evaluation of operators, i.e., their relative priority with each other. The higher the precedence, the earlier that operator will be resolved. It is represented as a number in Pensieve.

Associativity dictates the order in which operators of the same precedence will be evaluated when they appear in the expression. Any operator can either be *left-associative* or *right-associative*. In Algebra, operators like addition, subtraction, multiplication, and division are left-associative, while exponentiation is right-associative. Left-associativity implies that the left side of the expression will be evaluated first, then the right side, and vice versa. In Pensieve, associativity is represented using a **enum class**, which are basically named constants.

Listing 3.1: Associativity enum

```
enum class Associativity { LEFT, RIGHT };
```

Next up, we also need to store the token type. It can range from variables to different logical operators.

Listing 3.2: Token types

```
enum class TokenType {
    VARIABLE,
    NEGATION_OP,
    OR_OP,
    AND_OP,
    XOR_OP,
    IMLPICATION_OP,
    BICONDITIONAL_OP,
    LPAREN,
    RPAREN
};
```

A token data structure will have to store all this information for the parser to work correctly. In Pensieve, this is how we represent a token.

Listing 3.3: Token data structure

```cpp
class Token {
protected:
    std::string value;
    TokenType tokenType;
    int precedence;
    Associativity associativity;

public:
    Token(const std::string& value, TokenType tokenType, int precedence,
          Associativity associativity);

    std::string getValue() const;
    TokenType getTokenType() const;
    int getPrecedence() const;
    Associativity getAssociativity() const;

    bool operator==(const Token& other) const;
    bool isVariable();
    bool isUnaryOperator();
    bool isParen();
};
```

This token class not only stores the above-mentioned fields, but also provides getter methods for those fields and helper methods for equality-checking and determining the token types.

It should be noted that the only unary operator in Pensieve is the **negation** operator, for which special mechanisms were programmed to support their parsing and evaluation since the Shunting-Yard algorithm does not have first-class support for them.

3.3 Evaluation Flow

Before diving into the lexer implementation, it would be prudent to be aware of all the steps involved in the truth table generation.

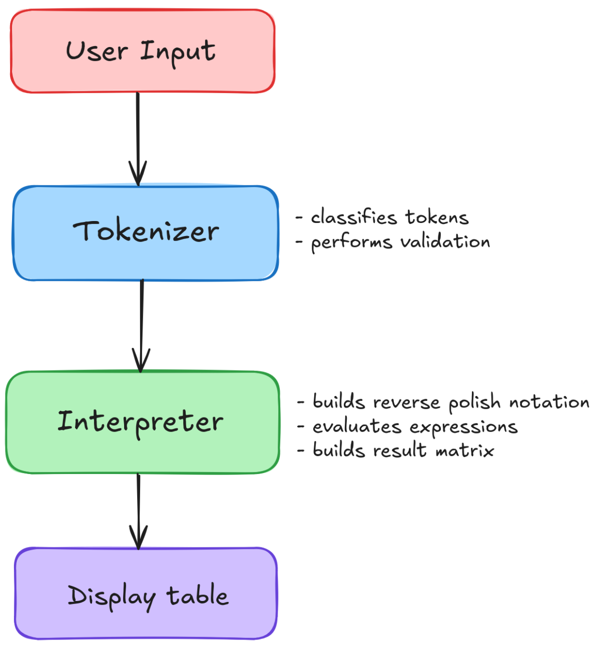


Figure 3.1: Pensieve Evaulation Flowchart

As evident from the flowchart, the input is first passed to the tokenizer, which converts the string input to a stream of tokens while validating it. The interpreter receives the tokens and converts them into RPN or Postfix notation using the Shunting-yard algorithm. The interpreter is also responsible for the result matrix (the truth table) and evaluating the expressions. Once the truth table is ready, it is displayed on the terminal with proper styling.

3.4 Lexer implementation

3.4.1 The Lexer Class Signature

Let's recall that the function of the lexer is to break down the input logical expression into tokens and perform error-checking on the expression. Let's take a look at the Lexer class signature.

Listing 3.4: The Lexer Class Signature

```cpp
class Lexer {
private:
    const std::string& infix;
    int position;
    char currentChar;
    int rank;
    std::stack<int> bracketPositions;

    void advance();
    bool isVariable(char c);
    void reportError(std::string error, int offset = -1);

public:
    Lexer(const std::string& input);
    std::vector<Token> tokenize();
};
```

The **infix** variable stores the input expression. The **position** variable points to the current index in the infix string that the lexer is scanning. The **currentChar** stores the character under examination.

rank and **bracketPositions** are used for error checking. The **reportError** method, as the name implies, is used to report errors in the expression. We'll discuss them soon.

The **advance** method increments the ‘position‘ and sets the ‘currentChar‘ variable to the current position.

isVariable is a helper method used inside the **tokenize** method to check if a character is alphabetic.

The **tokenize** method has the primary code for the lexer. It returns a ‘vector‘ of tokens. A vector in C Plus Plus (C++) is a dynamic, resizable array.

3.4.2 Token Classification

Let's break down the ‘tokenize‘ method.

Listing 3.5: Tokenize Method

```
std::vector<Token> tokens;
advance(); // not checking for exception here because empty input is
           // ignored by the input loop
bool reachedEnd = false;
while (!reachedEnd) {
    if (std::isspace(currentChar)) {
        goto next;
    }

    if (currentChar == '|') {
        tokens.push_back(OrToken());
    } else if (currentChar == '!') {
        tokens.push_back(NegationToken());
    } else if (currentChar == '&') {
        tokens.push_back(AndToken());
    } else if (currentChar == '^') {
        tokens.push_back(XorToken());
    } else if (currentChar == '>') {
        tokens.push_back(ImplicationToken());
    }
    ...
```

First of all, it runs a loop through the infix string and examines each character one by one, and adds the appropriate token to the list (vector) of tokens. For the sake of simplicity, Pensieve only uses single-character symbols for tokens.

3.4.3 Error Reporting Mechanism

The lexer has another critical function: syntax checking the logical expression. The lexer can report these errors:

1. invalid character
2. missing operand
3. missing operator

4. missing opening parentheses
5. missing closing parentheses

Let's see how those are implemented.

Pensieve includes a carefully constructed error reporting mechanism in the Lexer to ensure that syntactic mistakes in the expression are caught early and reported precisely. This mechanism is largely driven by two variables: **rank** and **bracketPositions**.

- **rank** is used to track the syntactical balance between operands and binary operators.
- **bracketPositions** is a stack used to ensure parentheses are correctly matched.

Rank is incremented when a variable (operand) is encountered, and decremented when a binary operator is encountered. The invariant maintained is that the number of operands must always be one more than the number of binary operators. Unary operators like `!` (NOT) do not affect the rank.

Listing 3.6: Rank-based error checking

```
if (isUnary) {
    // ignore unary operators in rank calculation
    goto next;
} else if (isVariable) {
    rank++;
} else if (!isParen) {
    rank--;
}

if (rank < 0) {
    reportError("missing␣operand");
    return std::vector<Token>{};
} else if (rank > 1) {
    reportError("missing␣operator");
    return std::vector<Token>{};
}
```

Any violation of this balance immediately triggers a call to the **reportError** method, which prints the original input expression, highlights the location of the error with a caret, and displays a human-readable message.

Listing 3.7: reportError Implementation

```
void Lexer::reportError(std::string error, int offset) {
    if (offset < 0) {
        offset = position;
    }
    std::cout << infix << "\n";
    std::cout << std::string(offset, '␣');
    std::cout << COLOR_RED << "^␣" << error << COLOR_RESET << std::endl;
}
```

Additionally, mismatched parentheses are handled using the **bracketPositions** stack. On encountering an opening parenthesis, its index is pushed onto the stack. On encountering a closing parenthesis, the top of the stack is popped. If a closing parenthesis is found without a matching open, or if the stack is non-empty at the end of parsing, the lexer reports an unmatched parenthesis error.

This robust mechanism ensures that malformed expressions are caught before any evaluation takes place, greatly improving the usability of the application and providing helpful feedback to the user.

Chapter 4

The Interpreter

4.1 Interpreter

The `Interpreter` class in *Pensieve* is responsible for evaluating logical expressions and generating the corresponding truth table. It takes a sequence of tokens generated by the lexer and computes the output column for the truth table by evaluating the expression across all possible combinations of input variables.

Responsibilities

- Convert infix expressions to postfix (Reverse Polish Notation).
- Extract all distinct variable tokens from the expression.
- Generate the initial truth table matrix for all combinations of input variables.
- Evaluate the logical expression for each row in the truth table.
- Format and print the final truth table using the `tabulate` library.
- Detect and display whether the expression is a tautology or a contradiction.

4.2 Postfix Conversion

The infix-to-postfix conversion is implemented using a standard stack-based Shunting Yard algorithm. It respects operator precedence and associativity while preserving the original structure of the logical expression.

Listing 4.1: Conversion from Infix to Postfix

```
void Interpreter::convertToPostfix() {
    for (auto& token : infixTokens) {
        // Handle variables and parentheses
        // Push or pop operators based on precedence and associativity
    }
    while (!operatorStack.empty()) {
        postfixTokens.push_back(operatorStack.top());
        operatorStack.pop();
    }
}
```

4.3 Variable Extraction and Matrix Generation

Once the unique variable tokens are extracted, an initial result matrix is generated. Each column corresponds to a variable, and each row to a possible input combination, covering all 2^n states.

Listing 4.2: Generating Initial Truth Table Matrix

```
void Interpreter::generateInitialMatrix() {
    int varCount = this->variableNames.size();
    int totalEntries = (int)pow(2, varCount);
    ...
    // Each column alternates between true/false in a power-of-two block pattern
}
```

4.4 Postfix Evaluation

The evaluation of the logical expression for each row of the matrix is done using a stack-based approach on the postfix tokens. Unary and binary operators are resolved accordingly.

Listing 4.3: Evaluating a Postfix Expression

```
bool Interpreter::evalPostfix(int rowIdx) {
    std::stack<bool> operands;
    ...
    for (auto& token : postfixTokens) {
        if (token.isVariable()) {
            operands.push(resultMatrix[varPos][rowIdx]);
        } else {
            bool result = resolveOperator(token, operands);
            operands.push(result);
        }
    }
    return operands.top();
}
```

The ‘resolveOperator‘ function operates on the operands and returns the result of the logical operation.

4.5 Truth Table Display

The final result is displayed in a formatted truth table. The `tabulate` library is used to generate a styled table with variable values and evaluated result per row. The result column is color-coded for clarity.

Listing 4.4: Displaying the Final Truth Table

```
std::vector<bool> Interpreter::displayResultMatrix() {
    ...
    // Add variable columns and result column
    // Format headers and cells using colors and styles
    std::cout << truthTable << std::endl;
}
```

If the result column contains all `true` values, the expression is a **tautology**. If all values are `false`, it is identified as a **contradiction**. These properties are printed in color-coded statements.

4.6 Utility Methods

Additional methods include:

- `getPostfix()`: Returns the postfix representation as a string.

- `getInfix()`: Constructs a readable infix string from the postfix tokens.
- `getVariables()`: Lists all variables in the expression.
- `evaluate()`: Public method to trigger full evaluation and table generation.

Chapter 5

The CLI

This is the entry point of the **Pensieve** logic evaluation tool. It provides a REPL interface that allows users to enter logical expressions, evaluate them, and display their truth tables. It also includes features like history, auto-completions, a debug mode, and a logical equivalence check.

5.1 Usage

5.1.1 Installation

To run the program, we first need to compile the program. To compile it, we need the source code. It can be obtained here. You can either clone the repository using the Git version control system or download the source code as a compressed zip archive and extract it.

Requirements:

1. The source code
2. A C++ compiler (preferably G++)

Most Linux systems come pre-installed with G++. On Windows, the latest compiler can be obtained from Winlibs. Ensure the compiler is from recent years, since we're using some modern C++ features not supported in older versions.

Navigate to the project directory and run:

```
make
```

The **linenoise** and **tabulate** libraries come pre-bundled with the source so we don't need to install them separately.

5.1.2 Running Pensieve

After compilation, run the executable from the terminal:

```
make run
```

This will start the interactive prompt where you can enter Boolean expressions.

5.2 Linenoise Configuration

We're employing the **linenoise** library for the line-based input with autocomplete and persistent history.

5.3 The `main()` Function

5.3.1 Setup and Greeting

The REPL is initialized with window signal handling, history file, completion callback, and styled I/O.

5.3.2 Input Loop

The loop:

1. Reads input from the user.
2. Normalizes it (lowercases and trims).
3. Parses commands:
   - `/debug`: toggles debug mode.
   - `exit`, `quit`, `/q`: exits the program.
4. If input contains comma-separated expressions, each is evaluated.

Listing 5.1: Main Loop Snippet

```
while (true) {
    char* result = linenoise(cyan("pensieve␣>␣").c_str());
    if (result == NULL) continue;
    if (*result == '\0') { free(result); break; }

    linenoiseHistoryAdd(result);
    std::string input(result);
    trim(input);
    std::transform(input.begin(), input.end(), input.begin(),
                   [](unsigned char c) { return std::tolower(c); });

    if (input == "quit" || input == "exit" || input == "/q") {
        std::cout << purple("bye") << std::endl;
        break;
    }

    if (input == "/debug") {
        debug = !debug;
        std::cout << purple("debug␣mode␣")
                  << purple((debug) ? "enabled" : "disabled") << '\n';
        continue;
    }

    ...
}
```

5.3.3 Expression Processing

Each expression is:

- Trimmed and tokenized using the Lexer.
- Passed to the Interpreter for evaluation.
- If in debug mode, the postfix form and variables are printed.
- The result (a column of booleans) is stored.

5.3.4 Equivalence Check

If multiple expressions are entered (comma-separated), their results are compared. If all vectors match, the expressions are logically equivalent.

Listing 5.2: Equivalence Checking

```
if (results.size() > 1) {
    bool same = true;
    const auto& firstResult = results[0];
    for (size_t i = 1; i < results.size(); i++) {
        if (!compareVectors(results[i], firstResult)) {
            same = false;
            break;
        }
    }

    if (same) {
        std::cout << green("All␣these␣expressions␣are␣logically␣equivalent")
                  << std::endl;
    } else {
        std::cout << red("All␣these␣expressions␣are␣NOT␣logically␣equivalent")
                  << std::endl;
    }
}
```

Chapter 6

Conclusion

Pensieve successfully addresses limitations in existing truth table generators through the implementation of a lexically robust parser and evaluation engine. This command-line utility efficiently processes Boolean expressions of arbitrary complexity while providing granular error diagnostics.

6.1 Technical Insights

Shunting Yard Algorithm Implementation The adaptation of Dijkstra's shunting yard algorithm for Boolean expression parsing yielded several key insights:

- Modification of operator precedence tables to accommodate logical primitives
- Token stream management with dual-mode operator handling (unary/binary)
- Recursive descent for parenthetical expressions with stack-based evaluation
- O(n) time complexity maintenance despite increased operator domain

Lexer and Tokenization Engineering Development of the lexical analyzer required sophisticated error-handling mechanisms:

- Implementation of token classification with positional metadata
- Context-sensitive error detection through lookahead buffer analysis
- Syntax error localization with character-precise positioning

- Token stream validation against formal grammar rules

C++ STL Utilization The project significantly enhanced STL proficiency through:

- Template-based container specialization for expression tree nodes
- Algorithm composition using functional programming paradigms
- Custom comparators for logical equivalence verification
- Stream buffer manipulation for formatted output generation

External Library Integration Third-party library incorporation provided experience in:

- Header-only dependency management with CMake integration
- Cross-platform compatibility testing for library dependencies
- Interface adaptation for library-specific data structures
- Performance profiling of external vs. custom implementations

6.2 Future Development Vectors

Expression Interface Enhancement

- Multi-character operator symbol parsing (e.g., `AND`, `OR`)
- Internationalization of operator symbols for educational contexts

Serialization Extensions

- Truth table export in CSV format using RFC 4180 compliance
- Markdown serialization with GitHub-flavored table syntax
- LaTeX-compatible truth table generation

Analytical Capabilities

- Step-by-step evaluation trace with intermediate truth values
- Boolean algebraic simplification using Quine-McCluskey algorithm
- Karnaugh map visualization for expressions with ¡= 4 variables

6.3 Summary

Pensieve demonstrates the effective application of formal language processing techniques to Boolean logic analysis. The implementation balances computational efficiency with educational utility, providing a valuable tool for both academic and practical applications in discrete mathematics and digital logic design.

The technical foundations established—particularly in lexical analysis, expression parsing, and evaluation algorithms—provide a solid platform for future enhancements. Through the systematic application of computer science principles to the domain of propositional logic, Pensieve exemplifies how specialized tools can bridge theoretical concepts with practical application.

Acronyms

C++ C Plus Plus. 15

REPL Read Evaluate Print Loop. iii, 23, 24

RPN Reverse Polish Notation. 11, 14