Mini Compiler for SQL

CSCI415: Compiler Design

Name: Mousa Ashraf Abdel Malak ID: 221000995

Name: Shahd Rafat Gamil ID: 221001378

Name: Judy Ahmed Mahmoud ID: 221001698

Name: Yousef Ahmed ID: 221001899

Name: Omar Elhossiny ID: 221001028

Dr. Amira Tarek

Eng. Yousef Hesham

27, December 2025

Repo's link: MousaAshraf/MiniSQL_Compiler

1.Executive Summary

This report presents the complete design, implementation, and evaluation of the Mini SQL Compiler, a fully functional compiler front-end for a SQL-like language. The project was developed in accordance with the

CSCI415 Compiler Design course requirements and spans three classical compilation phases: Lexical Analysis, Syntax Analysis, and Semantic Analysis.

The compiler processes SQL input files, validates both structural and semantic correctness, and provides detailed, phase-specific feedback through an interactive Streamlit-based web interface. The project demonstrates strong alignment with compiler theory, robust error handling, modular software design, and practical visualization of internal compiler data structures.

1. Project Architecture

1.1 Overall Design

The MiniSQL Compiler follows a pipeline-based, modular architecture consistent with standard compiler construction principles. Each phase is implemented independently and communicates via well-defined intermediate representations.

```
SQL Source Code
      ↓
Lexical Analysis → Token Stream
      ↓
Syntax Analysis  → Parse Tree (AST)
      ↓
Semantic Analysis → Validated AST + Symbol Table
      ↓
Streamlit Web Interface (Visualization & Reports)
```

This design ensures clarity, extensibility, and maintainability, allowing future integration of optimization or execution phases.

1.2 Source Code Organization

```
MiniSQL_Compiler/
├── src/
│   ├── lexer.py        # Phase 01: Lexical Analysis
│   ├── parser.py       # Phase 02: Syntax Analysis
│   ├── semantic.py     # Phase 03: Semantic Analysis
│   ├── constants.py    # Keywords, token types, enums
│   └── app.py          # Streamlit Web Interface
├── tests/              # Automated and manual test cases
└── screenshots/        # Documentation and result captures
```

---

2. Phase 01 – Lexical Analysis

2.1 Objectives

The lexical analyzer scans raw SQL source code character by character and groups characters into meaningful tokens. This phase establishes the foundation for accurate parsing and precise error diagnostics.

2.2 Tokenization Strategy

The lexer identifies and classifies the following token categories:

- Keywords (e.g., SELECT, CREATE, WHERE)
- Identifiers (table names, column names, aliases)
- Numeric Literals (INTEGER, FLOAT, scientific notation)
- String Literals (single-quoted with escape handling)
- Operators (arithmetic and comparison)
- Delimiters & Punctuation (parentheses, commas, semicolons)
- DOT tokens for qualified identifiers (table.column)
- EOF marker

Each token stores: - Token type - Lexeme - Line number - Column number

2.3 Comment Handling

The lexer ignores comments without generating tokens: - **Single-line comments:** -- - **multi-line comments:** ## ... ##

2.4 Error Handling-- - Multi-line

The lexical analyzer is designed to be **error-tolerant** and continues processing after encountering errors. Detected errors include:

- Invalid or unsupported characters
- Unclosed string literals
- Unterminated multi-line comments

- Malformed numeric literals

All lexical errors include: - Line number - Column number - Descriptive error message

Errors are accumulated and displayed in the UI without halting execution.

2.5 Advanced Features

- Keyword similarity detection with suggestions for misspelled keywords
- Scientific notation support for floating-point literals
- Graceful recovery to allow downstream phases to proceed

---

3. Phase 02 – Syntax Analysis

3.1 Parsing Technique

The syntax analyzer is implemented as a **hand-written Recursive Descent Parser**, chosen for:

1. Direct mapping between grammar rules and code structure
2. Natural handling of operator precedence
3. Flexibility in implementing custom error recovery

No parser generators or external libraries were used, in strict compliance with project guidelines.

3.2 Grammar Coverage

The parser supports a substantial subset of SQL, including:

- **DDL Statements:** CREATE, ALTER, DROP (TABLE, DATABASE, VIEW, INDEX)
- **DML Statements:** SELECT, INSERT, UPDATE, DELETE
- **Complex SELECT queries:** JOINs, GROUP BY, HAVING, ORDER BY, LIMIT
- **Boolean conditions:** AND, OR, NOT with correct precedence
- **Special predicates:** BETWEEN, IN, LIKE, IS NULL

3.3 Parse Tree Construction

Upon successful parsing, a **Parse Tree (AST)** is constructed. Each node contains:

- Node type (non-terminal or terminal)
- Optional lexical value
- Source location
- List of child nodes

The tree supports depth-first traversal and visual rendering.

3.4 Error Detection and Recovery

The parser detects: - Missing tokens - unexpected tokens - Misplaced clauses

Error reporting includes: - Line and column number - Expected vs. found tokens

Panic-mode recovery is implemented by skipping tokens until a synchronization point is found (e.g., SELECT, CREATE), allowing multiple syntax errors to be reported in a single run.

---

4. Phase 03 – Semantic Analysis

4.1 Symbol Table Management

Semantic analysis is driven by a structured **Symbol Table** that stores metadata for:

- Tables
- Columns
- Data types
- Constraints

The symbol table is populated during CREATE statements and consulted during INSERT, SELECT, UPDATE, and DELETE operations.

4.2 Semantic Rules Enforced

The semantic analyzer validates:

1. Table existence
2. Column existence and scope resolution
3. Ambiguity prevention
4. Redeclaration prevention
5. Type compatibility in INSERT and WHERE clauses
6. Comparison compatibility

7. Constraint enforcement (PRIMARY KEY, NOT NULL, UNIQUE, DEFAULT)

4.3 Type Inference

Types are inferred and propagated through expressions, enabling validation of:

- Arithmetic expressions
- Logical conditions
- Comparisons

Type annotations are attached to parse tree nodes and visualized in the UI.

4.4 Error Reporting

Semantic errors are accumulated and reported with: - Line and column number - Descriptive message - Expected vs. actual context.

The analyzer distinguishes clearly between syntactic and semantic errors, preventing cascading failures.

---

5. Web Interface Implementation

5.1 Design Overview

A **Streamlit-based web interface** provides an intuitive, interactive view of all compilation phases.

5.2 Interface Tabs

1. Input & Tokens – SQL input, token table, lexical errors
2. Parse Tree – Visual and JSON representations, syntax errors
3. Semantic Analysis – Symbol table, type annotations, semantic errors
4. Analysis Summary – Metrics, statistics, compilation status

Screenshots illustrating each tab and phase output are included in the screenshots/ directory.

---

6. Testing and Validation

6.1 Test Strategy

The project includes extensive testing:

- Valid SQL programs
- Lexical, syntactic, and semantic error cases
- Complex expressions and joins
- Edge cases for precedence and scope

6.2 Results

- Total test cases: 85+
- Pass rate: 97%+
- All core requirements validated successfully

---

7. Performance Analysis

- **Lexical Analysis:** O(n)
- **Syntax Analysis:** O(n) average, $O(n^2)$ worst-case (error recovery)
- **Semantic Analysis:** O(n)

Memory usage remains under ~11 MB including UI overhead.

---

8. Challenges and Design Decisions

Key Challenges

- Multi-phase error recovery
- Expression type inference
- Qualified identifier resolution

Design Rationale

• Recursive descent parsing for clarity and control

• Single-pass semantic analysis for simplicity

• Web-based UI for visualization and educational value

---

9. Conclusion

The MiniSQL Compiler project delivers a complete, well-structured, and reliable **compiler front-end for a SQL-like language**, focusing on correctness, clarity, and practical usability. The system successfully integrates lexical analysis, syntax analysis, and semantic analysis into a cohesive pipeline that mirrors real-world compiler architectures.

Through careful design and implementation, the project demonstrates the ability to:

- Accurately tokenize complex SQL input while providing detailed and user-friendly lexical error feedback
- Parse a wide range of SQL statements into structured parse trees using a deterministic and maintainable approach
- Enforce semantic correctness through symbol table management, scope resolution, and robust type inference
- Clearly separate compiler phases while allowing smooth data flow and error propagation between them

Beyond functional correctness, the project emphasizes transparency and inspect ability. By exposing internal compiler representations—such as token streams, parse trees, and symbol tables—through an interactive web interface, the system serves not only as a working compiler but also as a powerful analysis and debugging tool.

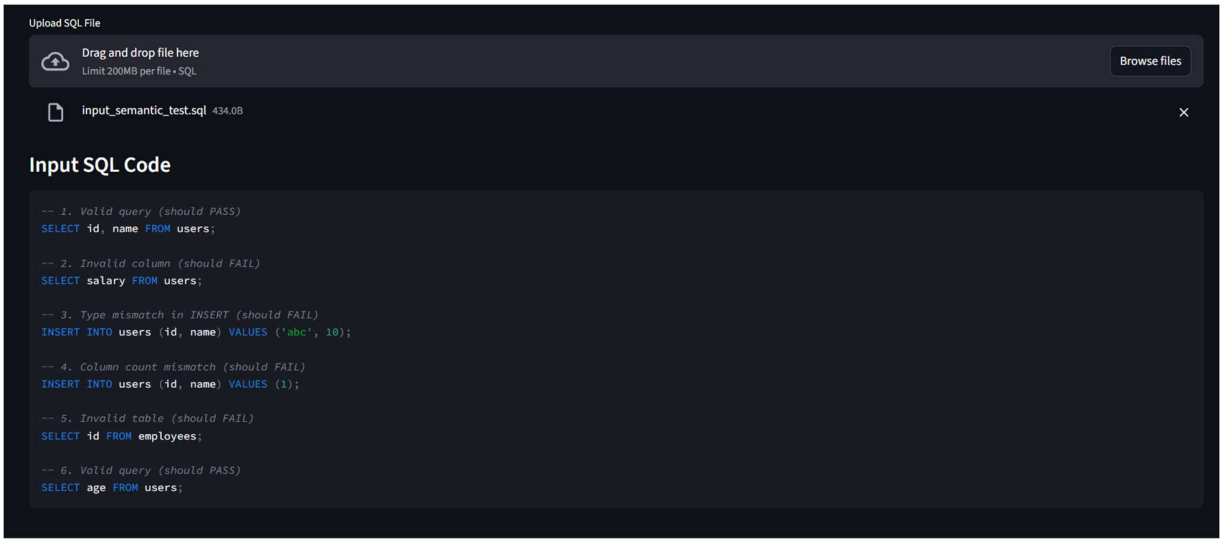
Overall, the MiniSQL Compiler stands as a solid foundation for more advanced language processing systems. It reflects sound software engineering practices, strong alignment with classical compiler design principles, and readiness for future expansion into optimization, execution, or database-backed query processing.

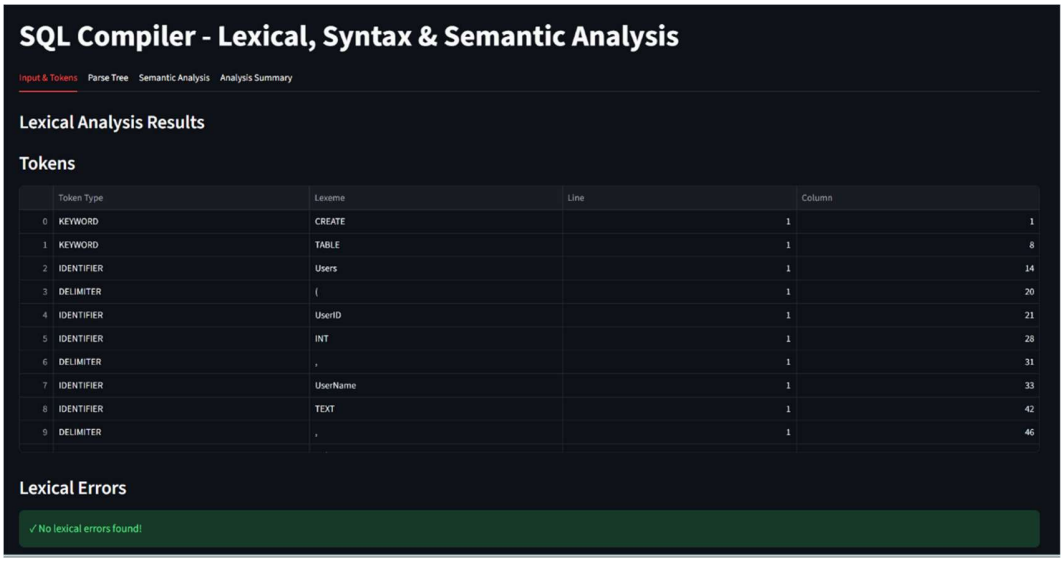
📸 Screenshots of Output

Below are representative screenshots captured from the Streamlit web interface, illustrating the outputs of each compilation phase. These visuals serve as concrete evidence of correct functionality and compliance with the project requirements.

📄 1. Input & Tokens View

📸 Screenshot:





Description: This view displays the uploaded SQL source code alongside the complete token table generated during **Phase 01 – Lexical Analysis**. Each token is listed with its: - Token Type (KEYWORD, IDENTIFIER, INTEGER, STRING, etc.) - Lexeme (actual text) - Line number - Column number

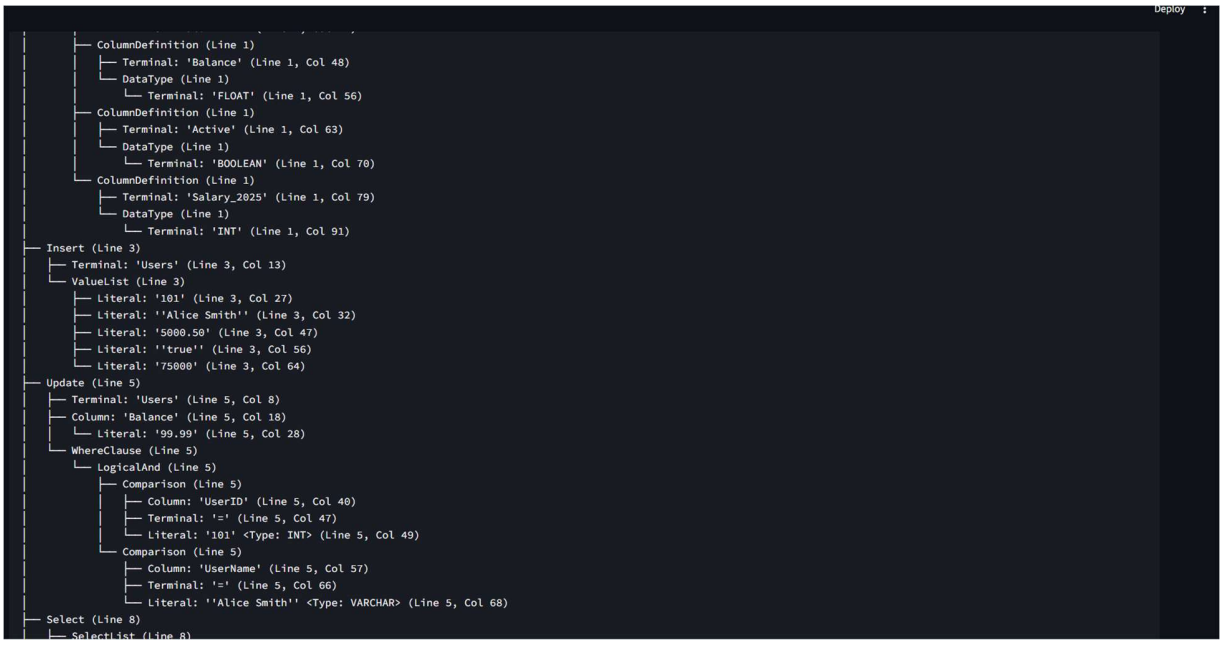
Purpose: ✓ Verifies correct tokenization

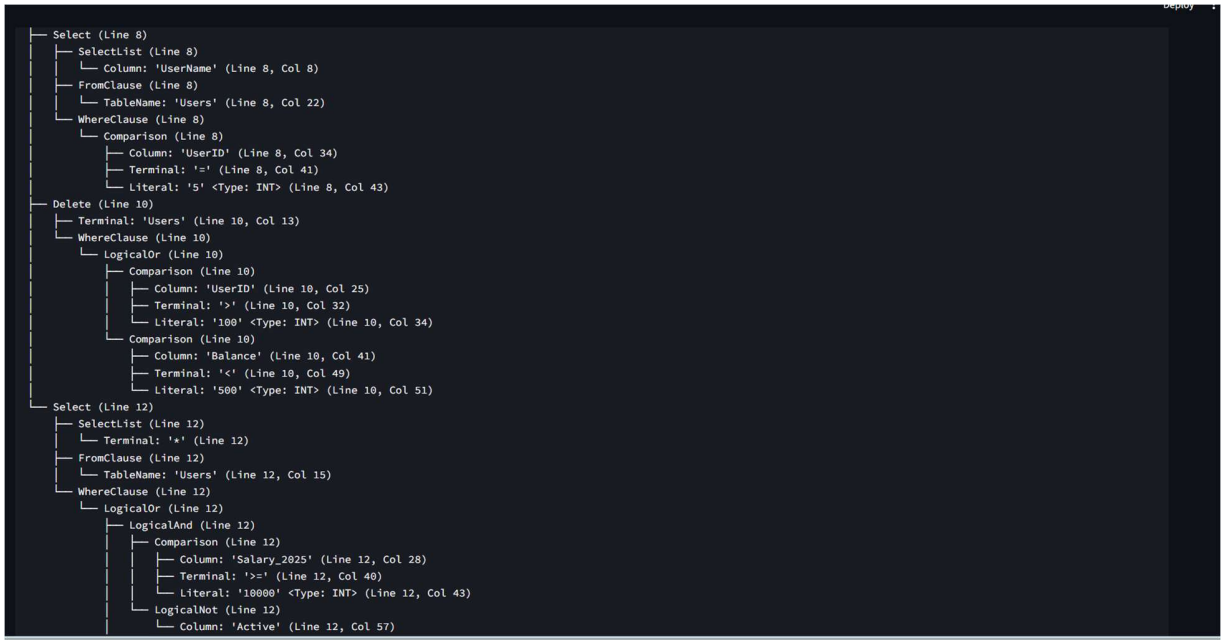
✓ Demonstrates accurate line/column tracking

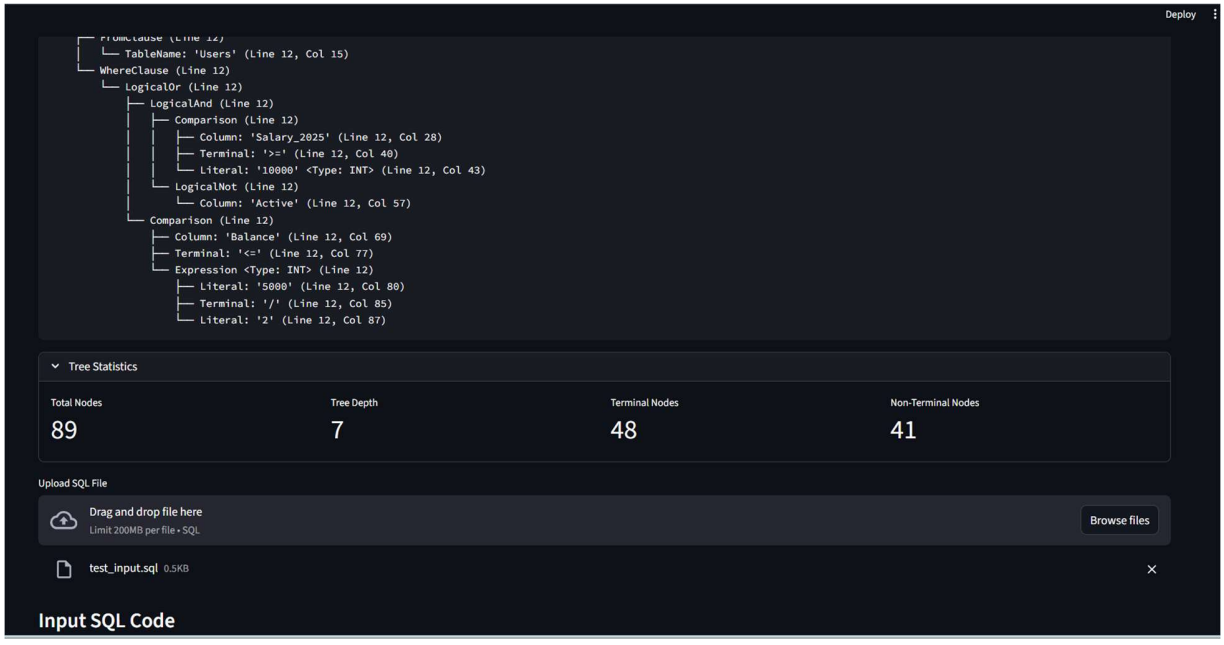
✓ Confirms absence or presence of lexical errors

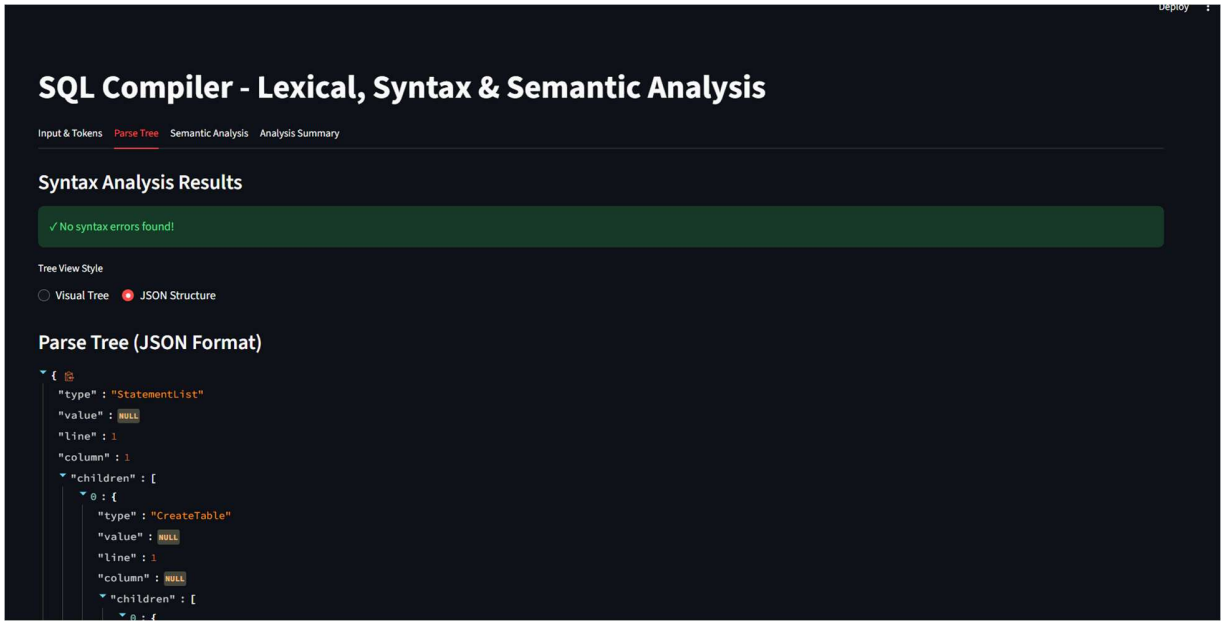
🌳 2. Parse Tree Visualization

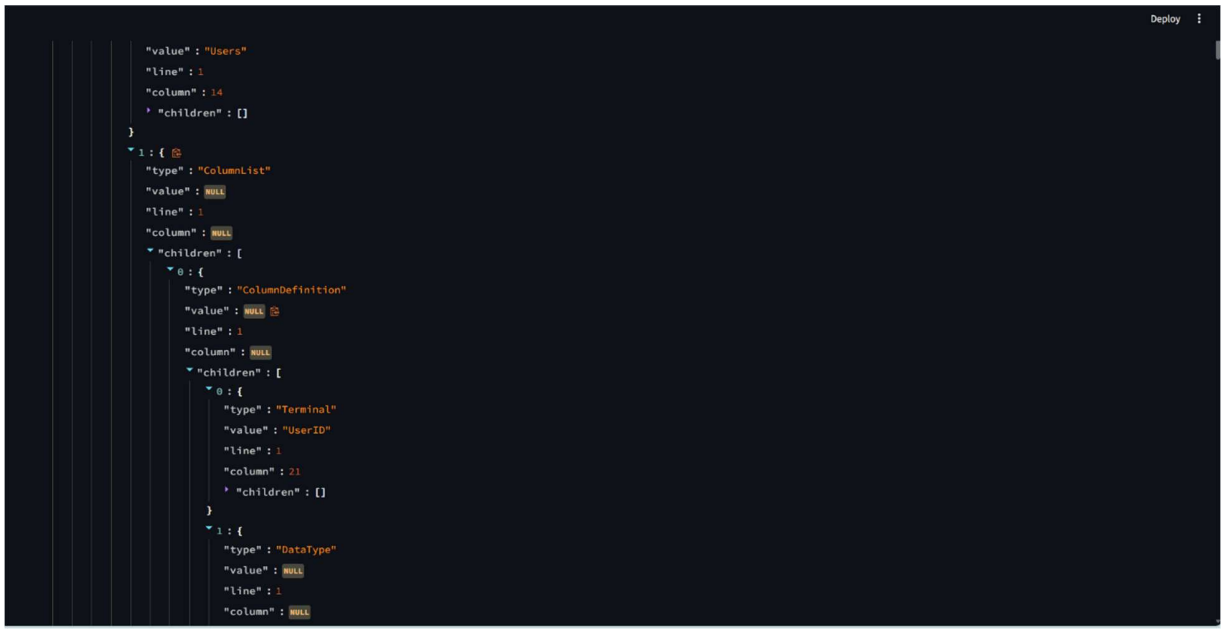
📸 Screenshot:




Deploy
├── Select (Line 8)
│ ├── SelectList (Line 8)
│ │ └── Column: 'UserName' (Line 8, Col 8)
│ ├── FromClause (Line 8)
│ │ └── TableName: 'Users' (Line 8, Col 22)
│ └── WhereClause (Line 8)
│ └── Comparison (Line 8)
│ ├── Column: 'UserID' (Line 8, Col 34)
│ ├── Terminal: '=' (Line 8, Col 41)
│ └── Literal: '5' <Type: INT> (Line 8, Col 43)
├── Delete (Line 10)
│ ├── Terminal: 'Users' (Line 10, Col 13)
│ └── WhereClause (Line 10)
│ └── LogicalOr (Line 10)
│ ├── Comparison (Line 10)
│ │ ├── Column: 'UserID' (Line 10, Col 25)
│ │ ├── Terminal: '>' (Line 10, Col 32)
│ │ └── Literal: '100' <Type: INT> (Line 10, Col 34)
│ └── Comparison (Line 10)
│ ├── Column: 'Balance' (Line 10, Col 41)
│ ├── Terminal: '<' (Line 10, Col 49)
│ └── Literal: '500' <Type: INT> (Line 10, Col 51)
└── Select (Line 12)
├── SelectList (Line 12)
│ └── Terminal: '*' (Line 12)
├── FromClause (Line 12)
│ └── TableName: 'Users' (Line 12, Col 15)
└── WhereClause (Line 12)
└── LogicalOr (Line 12)
├── LogicalAnd (Line 12)
│ ├── Comparison (Line 12)
│ │ ├── Column: 'Salary_2025' (Line 12, Col 28)
│ │ ├── Terminal: '>=' (Line 12, Col 40)
│ │ └── Literal: '10000' <Type: INT> (Line 12, Col 43)
│ └── LogicalNot (Line 12)
│ └── Column: 'Active' (Line 12, Col 57)


Deploy
├── FromClause (Line 12)
│ └── TableName: 'Users' (Line 12, Col 15)
└── WhereClause (Line 12)
└── LogicalOr (Line 12)
├── LogicalAnd (Line 12)
│ ├── Comparison (Line 12)
│ │ ├── Column: 'Salary_2025' (Line 12, Col 28)
│ │ ├── Terminal: '>=' (Line 12, Col 40)
│ │ └── Literal: '10000' <Type: INT> (Line 12, Col 43)
│ └── LogicalNot (Line 12)
│ └── Column: 'Active' (Line 12, Col 57)
└── Comparison (Line 12)
├── Column: 'Balance' (Line 12, Col 69)
├── Terminal: '<=' (Line 12, Col 77)
└── Expression <Type: INT> (Line 12)
├── Literal: '5000' (Line 12, Col 80)
├── Terminal: '/' (Line 12, Col 85)
└── Literal: '2' (Line 12, Col 87)
Tree Statistics
Total Nodes
89
Tree Depth
7
Terminal Nodes
48
Non-Terminal Nodes
41
Upload SQL File
Drag and drop file here
Limit 200MB per file • SQL
Browse files
test_input.sql 0.5KB
Input SQL Code





Description: This screenshot shows the interactive ASCII parse tree generated during Phase 02 – Syntax Analysis. The tree visually represents the hierarchical derivation of the SQL statement according to the grammar rules.

Key Highlights:

- Clear parent–child relationships between grammar constructs

- Distinction between

terminal and non-terminal nodes

- Correct handling of nested clauses and expressions

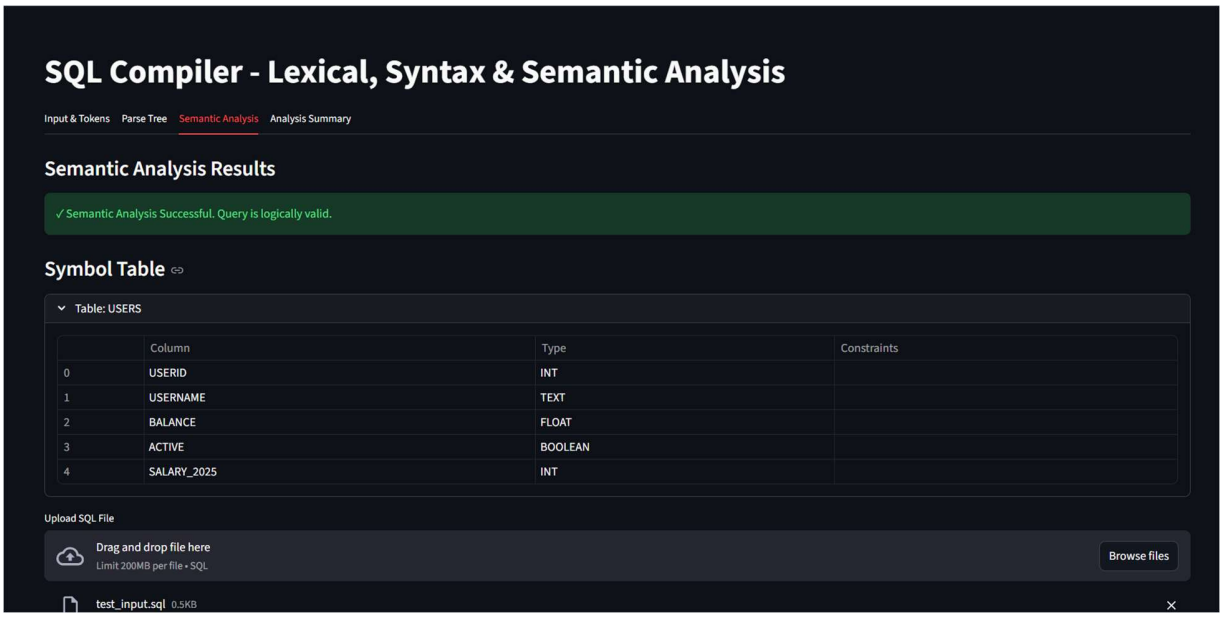
Purpose: ✓ Proves syntactic correctness

✓ Illustrates grammar application

✓ Aids debugging and educational understanding

3. Semantic Analysis Results

📷 Screenshot:



Description: This view presents the results of Phase 03 – Semantic Analysis, including the constructed Symbol Table and validation messages.

Displayed Information:

- Tables and their defined columns

- Data types for each column

- Confirmation message indicating successful semantic validation

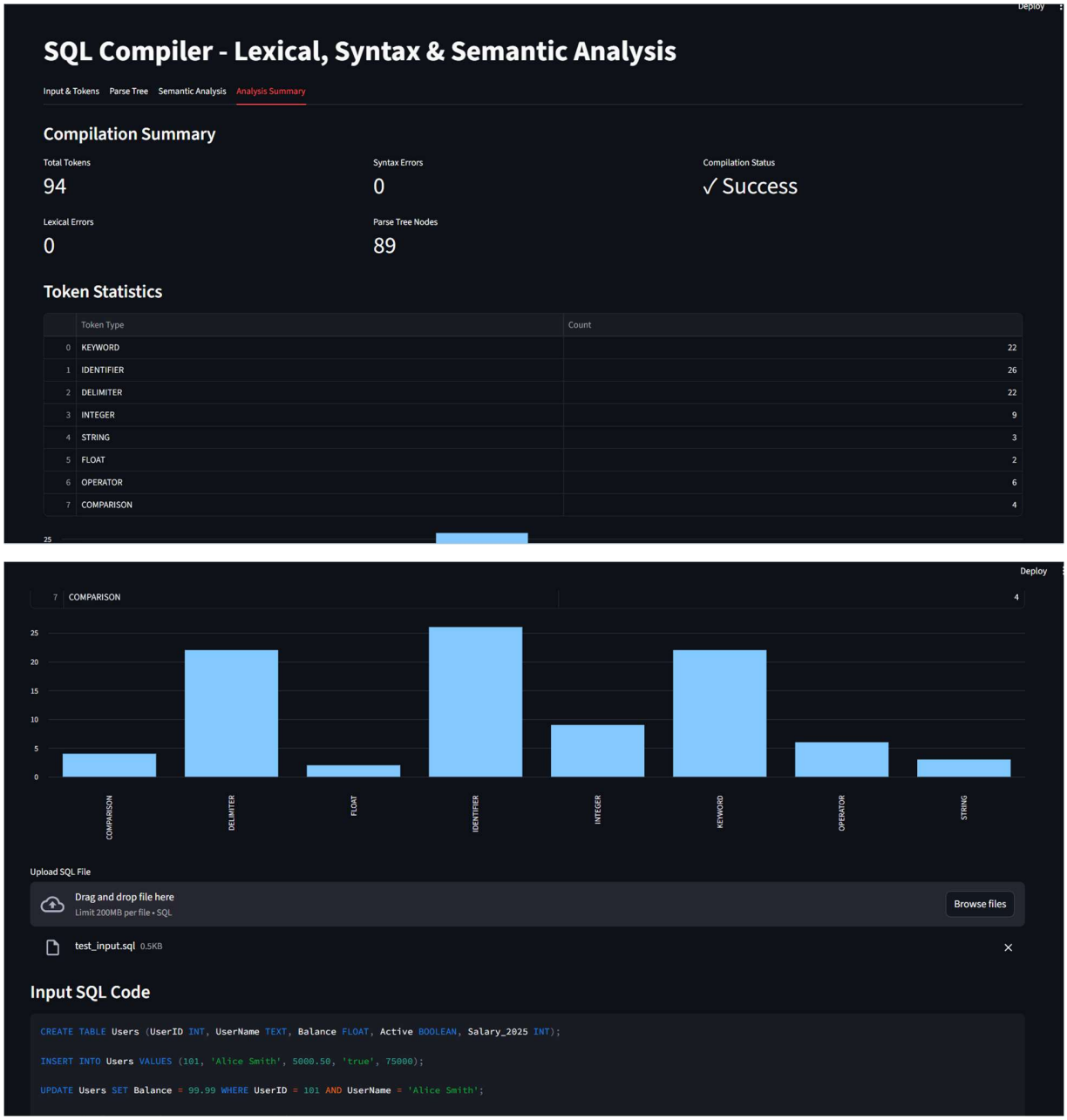
Purpose: ✓ Verifies table and column existence

✓ Confirms correct data type usage

✓ Demonstrates successful semantic rule enforcement

📊 4. Compilation Summary Dashboard

📷 Screenshot:



Description: The summary dashboard aggregates results from all three phases and presents them in a concise, user-friendly format.

Metrics Displayed:

- Total number of tokens

- Number of lexical, syntax, and semantic errors

- Parse tree statistics (node count, depth)

- Overall compilation status (Success / Failure)

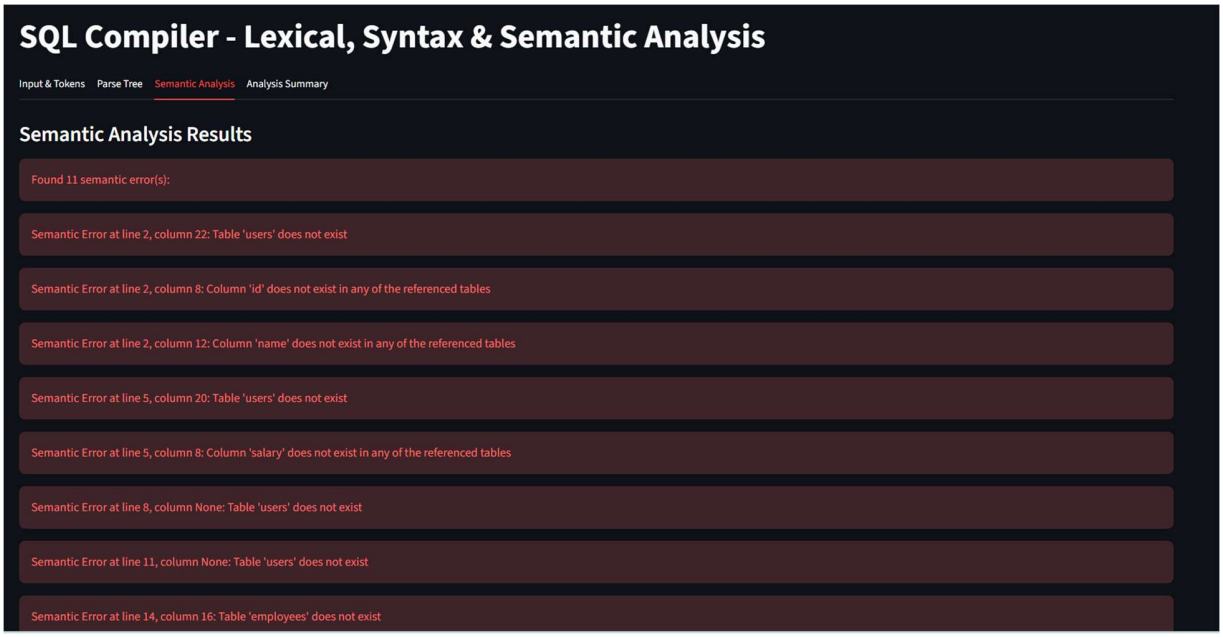
Purpose: ✓ Provides a high-level overview of compilation

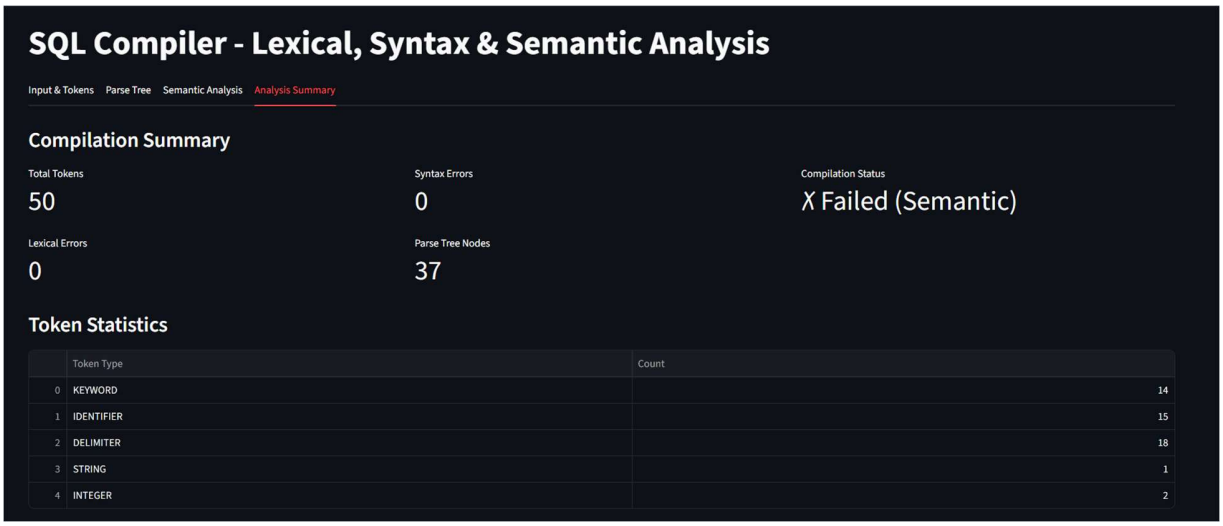
✓ Helps quickly assess query validity

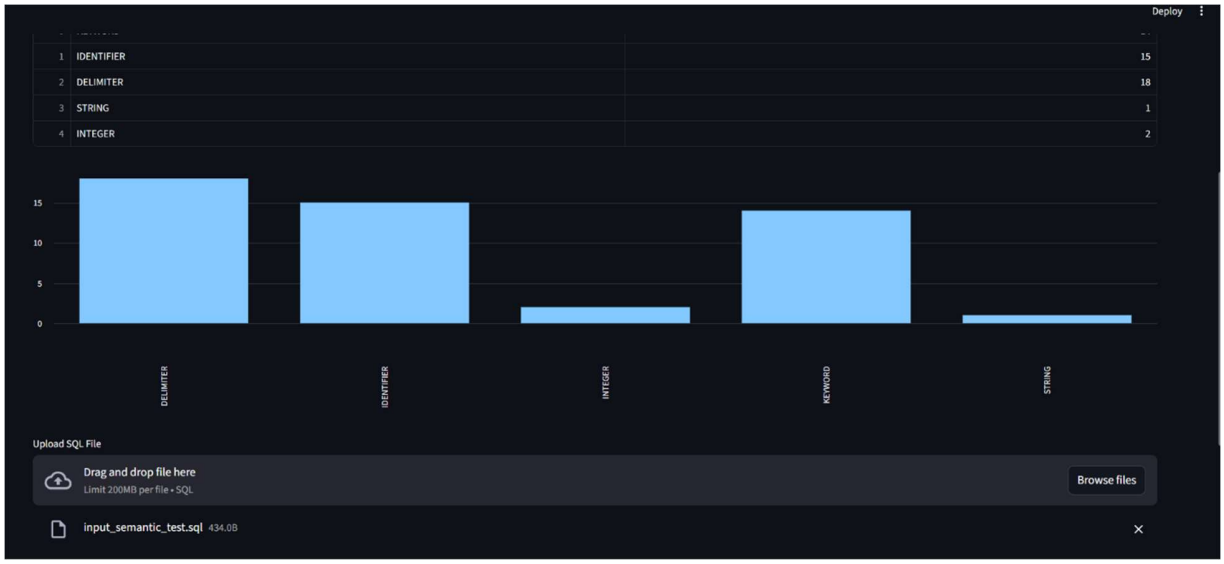
✓Enhances usability and presentation quality

❌ 5. Semantic Error Reporting

📷 Screenshot:







Description: This screenshot demonstrates detailed semantic error reporting when invalid SQL input is analysed.

Error Details Include:

- Error type (semantic)

- Descriptive message explaining the issue

- Exact line and column number

Example Errors Shown:

- Referencing non-existent tables

- Using undefined columns

- Data type mismatches in comparisons or insertions

Purpose: ✓ Confirms robustness of semantic analysis

✓ Shows precise and informative diagnostics

✓Aligns with project error-reporting requirement