



আন্তর্জাতিক ইসলামী বিশ্ববিদ্যালয় চট্টগ্রাম
الجامعة الإسلامية العالمية شيتا خونج
International Islamic University Chittagong

Department of Computer Science and Engineering
Compiler Project Report

Course Code: CSE-3528

Course Title: Compiler Lab

Project Title: Mini SQL Parser and Validator

Submitted To: Mr. Rayhanuzzaman

Assistant Lecturer, Department of CSE, IIUC

Submitted By:

Team No: 04

Team Members:

ID No.	Name
C231053	Mohammed Sami Hasan
C231068	Hasnain Kabir Nabil
C231070	Md. Nasim UL Haque

Semester: 5th

Section: 5BM

Submission Date: 08/08/2025

Remarks:

Project title: Mini SQL Parser and Validator

1. Introduction

The Mini SQL Parser project is a compiler-based application developed using C++ that focuses on parsing and analyzing basic SQL statements. It simulates the behavior of a simple database query processor and provides a hands-on understanding of compiler construction principles. The parser can process a subset of SQL operations such as SELECT, INSERT, UPDATE, and DELETE, and it performs key tasks including lexical analysis, syntax parsing, semantic validation, and symbol table management.

This project serves as an educational tool, demonstrating how structured queries are interpreted and validated in compiler design. By implementing features such as token generation, recursive descent parsing, and symbol tracking, the project replicates the initial stages of SQL query execution found in real-world database systems. The parser is particularly relevant for students studying compiler theory or database systems, offering a practical illustration of how compilers handle structured language inputs.

2. Objective

The primary objective of the Mini SQL Parser project is to design and implement a simplified SQL parsing system that demonstrates core principles of compiler construction. This includes the development of components such as a lexical analyzer, syntax parser, semantic validator, and symbol table manager, all within the context of SQL query processing.

Specifically, the project aims to:

- Implement a tokenizer capable of recognizing and categorizing SQL tokens such as keywords, operators, identifiers, and literals.
- Design a recursive descent parser that can validate the syntactic structure of various SQL statements.
- Construct an internal representation, such as an abstract syntax tree (AST), for parsed SQL queries.
- Manage symbol table entries for tables and columns, enabling semantic checks and schema validation.
- Provide informative error reporting to assist with debugging and to enhance understanding of compilation errors.
- Support multiple SQL statement types with logical operators and conditional clauses for a more realistic simulation.

This project not only meets academic goals for learning compiler design techniques but also lays the groundwork for developing more advanced database processing tools in the future.

3. Methodology

The Mini SQL Parser project adopts a modular, multi-phase compilation approach to process SQL statements. Each phase corresponds to a core component of a traditional compiler, allowing structured query input to be systematically analyzed and validated.

Lexical Analysis (Tokenization):

The first phase involves breaking down the raw SQL input into meaningful tokens using a custom tokenizer. This module scans the input character by character and uses pattern matching to identify SQL keywords, identifiers, operators, literals, and delimiters. A finite state machine-like logic is employed to categorize tokens accurately, ensuring robust detection of invalid characters or malformed inputs. The tokenizer supports over 25 distinct token types and provides output in a tabular format for verification.

Syntax Analysis (Parsing):

In this phase, the stream of tokens is parsed using a recursive descent parsing approach. Each type of SQL statement (SELECT, INSERT, UPDATE, DELETE) is handled by a specific parsing function that enforces SQL grammar rules. Look-ahead mechanisms are used to decide the parsing path, and syntactic structures are validated accordingly. This stage also constructs internal representations, such as an Abstract Syntax Tree (AST), for further semantic processing.

Semantic Analysis:

Semantic validation ensures that the parsed SQL statement makes logical sense. This involves checking the existence of referenced tables and columns, validating data types, and ensuring that operators are used correctly. The parser performs type checks for literals and expressions and validates column references within the context of the declared schema.

Symbol Table Management:

The central part of the system is the symbol table, which tracks metadata about tables and columns used throughout the program. Implemented using hash maps for efficient lookup, the symbol table supports case-insensitive identifiers, detects duplicates, and stores schema information dynamically as new tables or columns are parsed. It also provides an export feature for external review.

By structuring the project into these distinct yet interconnected phases, the methodology ensures clarity, maintainability, and a close resemblance to how professional compilers and interpreter's function.

4. Tools and Technologies Used

The development of the Mini SQL Parser project involved a set of modern programming tools and technologies, carefully chosen for their compatibility with compiler construction principles and performance in handling system-level programming tasks.

Programming Language:

- **C++ (C++17 standard):**

The core implementation is written in C++ using features from the C++17 standard, such as smart pointers, enhanced regular expressions, and unordered containers. C++ was chosen for its strong performance, memory control, and suitability for building custom parsers and data structures.

Development Environment:

- **Visual Studio Code:**

The project was developed in Visual Studio Code, a lightweight and highly customizable code editor, with support for syntax highlighting, debugging, and terminal integration for building and testing the project.

Compiler:

- **GCC (GNU Compiler Collection) / MinGW (on Windows):**

The project was compiled using the g++ compiler with C++17 support. The use of command-line compilation ensures better understanding of compiler flags and build processes.

Standard Libraries and Modules:

- `<regex>`: Used for pattern matching during tokenization.
- `<memory>`: Utilized for managing dynamic memory with smart pointers.
- `<unordered_map>` and `<unordered_set>`: Employed for efficient implementation of the symbol table.
- `<vector>`, `<string>`, and other STL containers: Used for managing tokens, symbols, and syntax trees.

Testing and Debugging Tools:

- **Manual Test Cases:**

The system was tested using a series of predefined SQL queries saved in test files. These queries cover various supported statements and help verify the correctness of each component.

5. Project Features

The Mini SQL Parser provides several essential features that demonstrate the foundational stages of SQL query processing. These features are designed to mimic the behavior of a basic database engine, focusing on parsing, validation, and symbol tracking, without involving actual data storage or execution.

SQL Statement Support:

The parser supports a limited but meaningful subset of SQL, including the following operations:

- **SELECT** statements with support for:
 - Selecting specific columns or all columns using *
 - Single table queries via the FROM clause
 - Conditional filtering using the WHERE clause
 - Logical expressions with AND and OR operators
 - Comparison operators (=, <, >, <=, >=, <>)
- **INSERT** statements for:
 - Specifying target tables and columns
 - Inserting values with type validation (e.g., string or numeric)
 - Matching column-value pairs
- **UPDATE** statements enabling:
 - Use of the SET clause for assigning new values to columns
 - Conditional updates using WHERE
 - Multiple columns in a single update
- **DELETE** statements for:
 - Deleting rows based on conditions in the WHERE clause
 - Support for logical expressions in conditions

Tokenization and Lexical Analysis:

The tokenizer efficiently converts SQL statements into a list of categorized tokens. It can identify over 25 token types, including:

- SQL keywords
- Identifiers (table/column names)
- Numeric and string literals
- Operators and delimiters

Recursive Descent Parsing

The parser uses recursive descent techniques to validate the structure of SQL statements. Each supported SQL operation has a dedicated parsing function, ensuring clear separation of concerns and adherence to SQL grammar.

Symbol Table Management

The parser includes a dynamic symbol table that stores metadata about tables and columns. It offers:

- Fast lookup using hash maps
- Case-insensitive identifier support
- Duplicate declaration detection
- Export capability to display the current schema information

Abstract Syntax Tree (AST) Representation

Although simplified, the parser constructs internal representations of statements that mimic an AST, allowing future extension for query optimization or execution. These features collectively offer a well-rounded simulation of SQL parsing, suitable for academic demonstration of compiler frontend design.

6. Data Flow Diagram

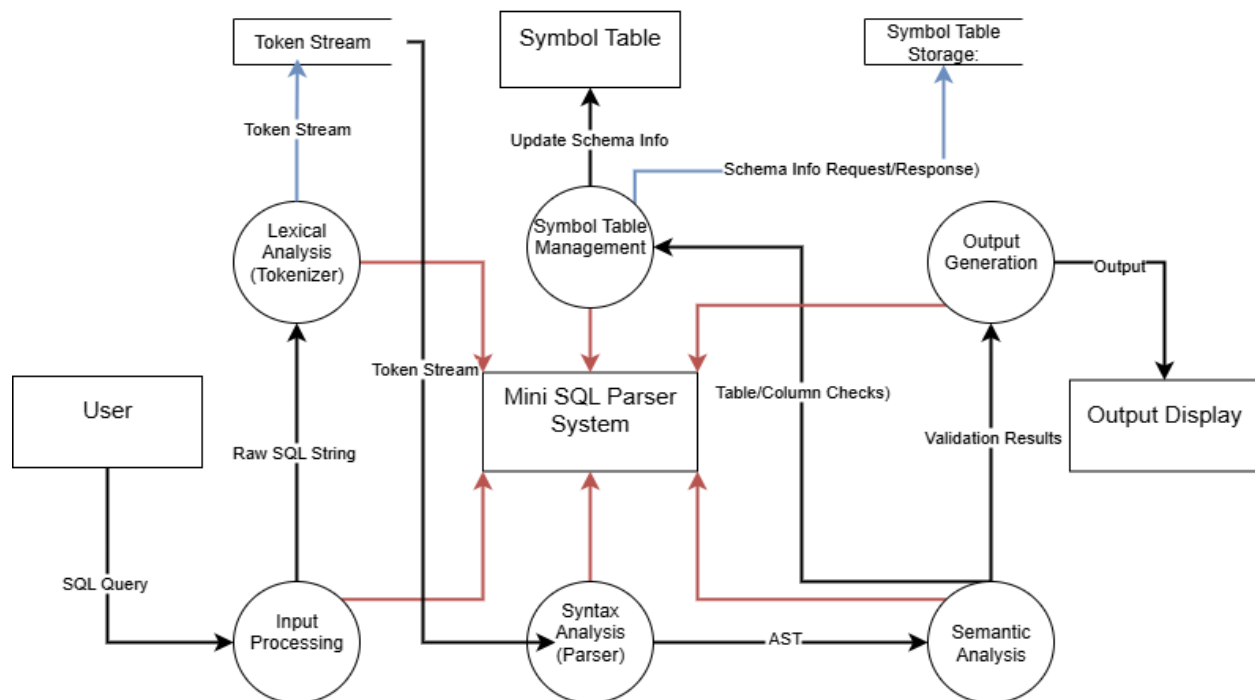


Figure: DFD of the mini-SQL parser system

7. Working Mechanism

The Mini SQL Parser operates through a well-defined sequence of stages that reflect the essential phases of a compiler. These stages begin with raw input provided by the user and end with structured output or error messages. Each component of the system interacts in a pipeline fashion to simulate how real-world SQL query processors work.

Step 1: Input Acquisition

The system starts by accepting SQL statements from the user through the console interface. Users can input a wide range of queries, such as SELECT, INSERT, UPDATE, and DELETE commands.

Step 2: Lexical Analysis (Tokenization)

Once the input is received, the tokenizer module performs lexical analysis. It reads the input character by character, matches patterns using regular expressions, and converts the raw SQL into a stream of tokens. These tokens are classified into categories such as keywords, identifiers, operators, literals, and delimiters. The generated token list is used in the next phase for syntax analysis.

Step 3: Syntax Analysis (Parsing)

The parser receives the token stream and verifies that the sequence of tokens conforms to SQL grammar rules. This phase uses a recursive descent parsing approach, where different functions handle different SQL statements. If the structure is valid, the parser constructs an internal representation such as an Abstract Syntax Tree (AST), which captures the hierarchical structure of the query.

Step 4: Semantic Analysis

Following successful parsing, the system performs semantic checks. This includes validating whether the referenced tables and columns exist in the symbol table, checking data types, and ensuring logical consistency in conditions and expressions. If all checks pass, the system proceeds; otherwise, appropriate semantic errors are reported.

Step 5: Symbol Table Management

The symbol table acts as a repository of schema information (e.g., table names, column names, and their attributes). During parsing, new tables or columns may be added to the symbol table. It also allows the parser to verify the validity of identifiers during semantic analysis.

Step 6: Output Generation

After successful analysis, the parser generates structured output summarizing the parsed SQL statement. This output includes token information, structural breakdown, or symbol table

exports. In case of any lexical, syntax, or semantic errors, detailed error messages are displayed, highlighting the nature and location of the problem.

This flow of execution ensures a clear and educational simulation of how SQL queries are processed in the frontend of a database management system, making it a practical demonstration of compiler design concepts.

8. Input & Output

```
=====
How many SQL statements do you want to execute? 2
Please provide SQL Statement (or type EXPORT to save symbol table):
SELECT name, age FROM users WHERE age > 30 AND salary < 5000;
=====
```

Tokens:

Token	Type
SELECT	SELECT
name	IDENTIFIER
,	COMMA
age	IDENTIFIER
FROM	FROM
users	IDENTIFIER
WHERE	WHERE
age	IDENTIFIER
>	GREATER
30	NUMBER
AND	AND
salary	IDENTIFIER
<	LESS
5000	NUMBER
;	SEMICOLON

Parsed Statement (AST):

Type: SELECT

Columns:

- name
- age

Table: users

WHERE: age > 30 AND salary < 5000

WHERE (Tree):

Condition: age > 30
LogicalOp: AND
Condition: salary < 5000

WHERE (Flat): age > 30 AND salary < 5000

Symbol Table:

Table Name: users

Columns:

- age
 - name
-

Please provide SQL Statement (or type EXPORT to save symbol table):

EXPORT

Enter filename to export output: test01

Output exported to test01

Please provide SQL Statement (or type EXPORT to save symbol table):

INSERT INTO employees (id, name, salary, department, is_active) VALUES
(101, 'John Doe', 75000.50, 'Engineering', true);

=====

Tokens:

Token	Type
INSERT	INSERT
INTO	INTO
employees	IDENTIFIER
(LPAREN
id	IDENTIFIER
,	COMMA
name	IDENTIFIER
,	COMMA
salary	IDENTIFIER
,	COMMA
department	IDENTIFIER
,	COMMA
is_active	IDENTIFIER
)	RPAREN
VALUES	VALUES
(LPAREN

101	NUMBER	
,	COMMA	
John Doe	STRING	
,	COMMA	
75000.50	NUMBER	
,	COMMA	
Engineering	STRING	
,	COMMA	
true	BOOLEAN	
)	RPAREN	
;	SEMICOLON	

+-----+

Parsed **Statement** (AST):

Type: INSERT
Table: employees
Columns:
- id
- name
- salary
- department
- is_active
Values:
- 101
- 'John Doe'
- 75000.50
- 'Engineering'
- true

Symbol **Table**:

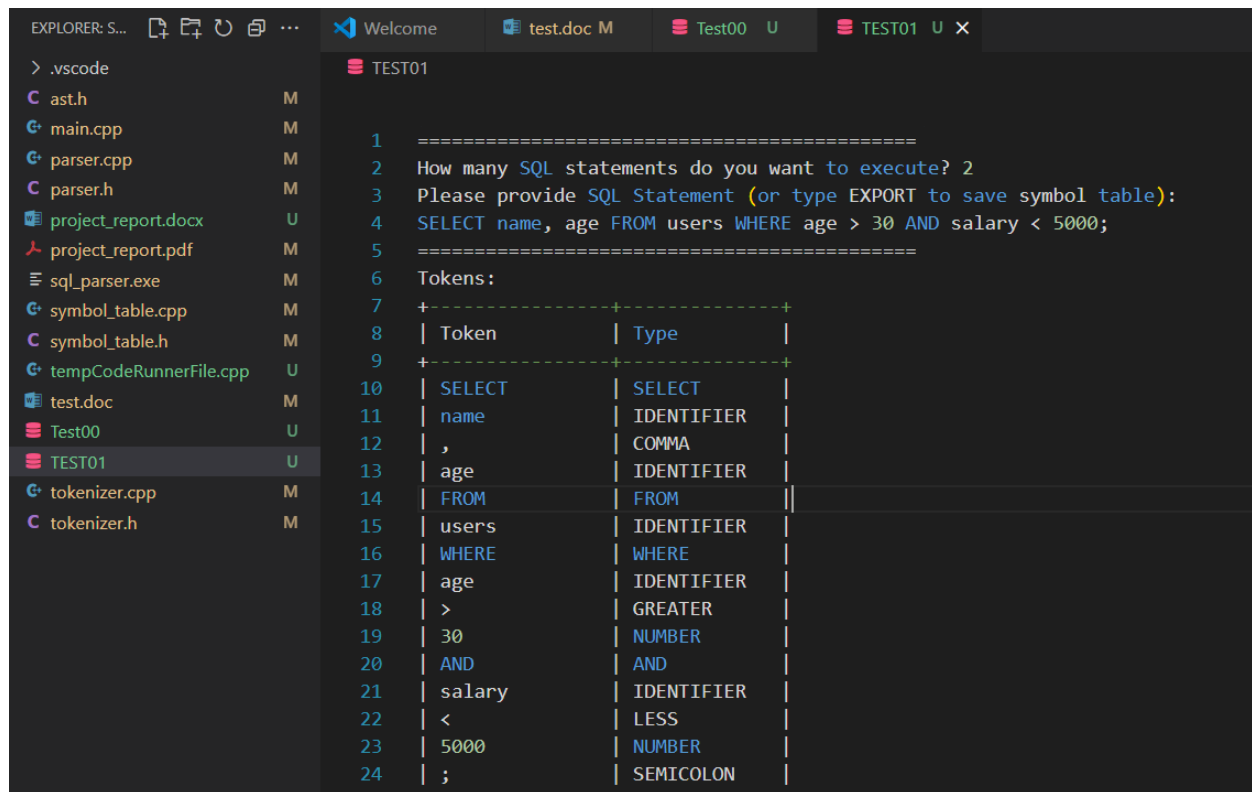
Table Name: employees
Columns:
- is_active
- id
- name
- salary
- department

Table Name: users

Columns:

- age
- name

Files Export in the System storage:



```
1 =====
2 How many SQL statements do you want to execute? 2
3 Please provide SQL Statement (or type EXPORT to save symbol table):
4 SELECT name, age FROM users WHERE age > 30 AND salary < 5000;
5 =====
6 Tokens:
7 +-----+-----+
8 | Token      | Type      |
9 +-----+-----+
10 | SELECT     | SELECT    |
11 | name       | IDENTIFIER|
12 | ,          | COMMA     |
13 | age        | IDENTIFIER|
14 | FROM       | FROM      |
15 | users      | IDENTIFIER|
16 | WHERE      | WHERE     |
17 | age        | IDENTIFIER|
18 | >          | GREATER   |
19 | 30         | NUMBER    |
20 | AND        | AND       |
21 | salary     | IDENTIFIER|
22 | <          | LESS      |
23 | 5000       | NUMBER    |
24 | ;          | SEMICOLON |
```

9. Error Handling

Effective error handling is a critical component of any compiler or parser system. In the Mini SQL Parser project, error detection and reporting are integrated into each major stage of query processing: lexical analysis, syntax parsing, and semantic validation. The system is designed to detect a variety of errors and report them with clarity to assist users in debugging their SQL inputs.

1. Lexical Errors

These errors occur during tokenization when the system encounters characters or patterns that do not conform to any recognized token types. Examples include:

- Invalid symbols (e.g., @, # in identifiers)
- Unterminated string literals
- Malformed numeric values

The tokenizer captures these issues early and outputs messages indicating the invalid token and its position in the input.

2. Syntax Errors

Syntax errors arise when the sequence of tokens does not match the expected grammar for a valid SQL statement. Common issues include:

- Missing keywords such as FROM, SET, or WHERE
- Incorrect ordering of clauses
- Mismatched parentheses
- Unexpected token types in specific positions

The parser identifies these errors during recursive descent parsing and reports them along with the problematic token and its surrounding context.

3. Semantic Errors

These errors involve logical mistakes that are not caught by syntax rules but are discovered during the semantic analysis phase. Examples include:

- Referencing non-existent tables or columns
- Assigning a string value to a numeric column
- Using incompatible data types in expressions
- Attempting to insert values into undefined columns

Semantic validation is performed using the symbol table, and violations are reported with detailed descriptions to guide correction.

4. Error Reporting Mechanism

Across all stages, the system provides structured and descriptive error messages, which may include:

- The line or token number (if applicable)
- The type of error (lexical, syntax, semantic)
- A brief explanation of what went wrong
- The specific token or clause causing the issue

This consistent error reporting enhances the educational value of the parser and aids users in identifying and correcting input errors efficiently.

10. Limitations

While the Mini SQL Parser successfully demonstrates fundamental principles of compiler design and SQL parsing, it is intentionally limited in scope to maintain simplicity and educational focus. The following are some of the known limitations of the system:

- **Limited SQL Syntax Support**

The parser only supports a subset of SQL operations — specifically SELECT, INSERT, UPDATE, and DELETE statements. More advanced SQL features such as JOIN, GROUP BY, ORDER BY, HAVING, UNION, subqueries, and functions are not implemented.

- **No Query Execution or Data Storage**

The system simulates the parsing and validation of SQL queries but does not connect to an actual database or execute queries. There is no mechanism to store or retrieve real data, as the focus is solely on analysis, not execution.

- **Basic Error Handling Scope**

Although the system handles lexical, syntax, and semantic errors, the granularity and recovery mechanisms are limited. The parser does not attempt to recover from errors to continue parsing subsequent statements and instead halts processing on the first encountered issue.

- **Lack of Full AST Implementation**

While internal representations of parsed statements are constructed, a fully structured Abstract Syntax Tree (AST) with support for tree traversal or optimization is not included in the current version.

- **Static Input and Output Format**

All SQL input is provided via the console or test files, and outputs are text-based without a GUI or advanced visualization. This limits the user experience and may make debugging more difficult in large inputs.

- **No User-Defined Data Types or Constraints**

The system does not support data definition language (DDL) features beyond basic CREATE TABLE, and lacks support for defining custom data types, constraints (e.g., PRIMARY KEY, NOT NULL), or relationships (e.g., foreign keys).

These limitations are acceptable given the educational goals of the project, but they also highlight areas where future enhancements can be made.

11. Future Improvements

The Mini SQL Parser lays the groundwork for a functional SQL query analyzer, yet there are several enhancements that could significantly increase its capability, usability, and real-world applicability. Future improvements may include the following:

Expanded SQL Grammar Support

To handle more complex queries, the parser can be extended to support:

- Joins (INNER JOIN, LEFT JOIN, etc.)

- Subqueries in SELECT, WHERE, and FROM clauses
- Aggregate Functions like COUNT(), SUM(), AVG()
- Grouping and Sorting (GROUP BY, ORDER BY)
- Aliases and Nested Queries

This would make the parser more compliant with standard SQL and suitable for broader educational and testing use cases.

Improved Error Recovery

Current error handling stops parsing at the first error. Future versions could be implemented:

- Error recovery strategies (e.g., panic-mode recovery)
- Continued parsing after non-critical errors
- Suggestions for corrections in error messages

GUI or Web Interface

A graphical or web-based interface could make the tool more user-friendly by:

- Allowing users to input and run queries visually
- Displaying token streams, parse trees, and symbol tables
- Highlighting errors in real-time

Advanced Symbol Table Functionality

Enhancing the symbol table to include:

- Data types, constraints, and scopes
- Multiple table support
- Schema validation

This would improve semantic checks and better simulate a real database environment.

Modular Codebase and Documentation

- Better code modularization to allow plug-and-play of new grammar rules
- In-code and external documentation to aid future contributors and maintainers

These improvements would transform the Mini SQL Parser into a more complete and powerful educational tool or even a functional lightweight SQL engine.

12. Conclusion

The Mini SQL Parser project successfully demonstrates fundamental concepts of compiler construction, particularly in the context of parsing a simplified subset of SQL queries. Through the implementation of modules for tokenization, parsing, and symbol table management, the project provides a practical, hands-on approach to understanding how high-level languages are processed and validated by compilers or interpreters.

By focusing on core SQL operations such as SELECT, INSERT, UPDATE, and DELETE, and enabling conditional logic through WHERE clauses and logical operators, the system effectively simulates the front-end processing of a database query engine. The project not only reinforces theoretical concepts from compiler design, such as lexical analysis and syntax validation, but also emphasizes the importance of semantic analysis using a symbol table.

While limited in its current scope, the project lays a strong foundation for future enhancements, including integration with backend databases, support for more advanced SQL features, and the development of user-friendly interfaces. Overall, this project serves as a meaningful academic exercise that bridges the gap between compiler theory and real-world application in database systems.

13. Team Management

The Mini SQL Parser project was developed by a team of three dedicated members, each contributing crucial expertise to different parts of the system. The collaborative effort ensured a comprehensive and well-integrated solution.

- **Mohammed Sami Hasan (Team Leader):** As the team leader, Sami coordinated the overall project development, managed task assignments, and ensured timely progress. He also actively contributed to the design and implementation of core parser components and supervised integration and testing phases.
- **Hasnain Kabir Nabil:** Hasnain was primarily responsible for developing the Tokenizer module, focusing on lexical analysis and accurate token generation. His work provided the essential foundation for parsing by ensuring precise and efficient tokenization of SQL statements.
- **Md. Nasim UL Haque:** Nasim took charge of the Symbol Table management and Semantic Analysis components. His efforts enabled robust validation of SQL statements through schema management and semantic checks, greatly enhancing the parser's reliability.

The team utilized GitHub for version control, enabling seamless collaboration and effective conflict resolution. Regular meetings facilitated knowledge sharing and alignment on project goals. Documentation and report writing were completed collaboratively, ensuring clarity and consistency across all deliverables.

This well-organized teamwork and clear distribution of responsibilities were key factors in successfully delivering a functional and educational mini-SQL parser.

14. References

The following references were used to guide the development, structure, and understanding of the Mini SQL Parser project:

- **Books & Academic Resources**
 - Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2006). *Compilers: Principles, Techniques, and Tools* (2nd Edition) also known as the Dragon Book.
- **Online Tutorials & Documentation**
 - GeeksforGeeks - Compiler Design Tutorials
 - cplusplus.com - C++ Standard Library Documentation
 - [Stack Overflow](#) — For troubleshooting specific issues and code snippets.
- **Tooling & Platforms**
 - GNU Compiler Collection (G++)
 - Visual Studio Code
 - [GitHub](#) — For version control and collaborative development.
- **AI Assistance**
 - **ChatGPT (OpenAI, GPT-4o):** Assisted in the planning, structuring, and writing of the project report. Provided explanations for compiler concepts, helped format content clearly, and suggested improvements based on project details.
- **Project-Specific Materials**
 - Lab instructions, teacher guidelines, and lecture notes provided during the compiler course.

15. Code Sample

1. Parsing a **SELECT** statement (from **parser.cpp**):

```
SelectStatement parseSelect(const std::vector<Token>& tokens, SymbolTable&
symTable) {
    index = 0;
    SelectStatement stmt;
    expect(tokens, TokenType::SELECT);
    while (tokens[index].type == TokenType::IDENTIFIER ||
tokens[index].type == TokenType::STAR) {
        stmt.columns.push_back(tokens[index++].value);
        if (tokens[index].type == TokenType::COMMA) index++;
        else break;
    }
    expect(tokens, TokenType::FROM);
    stmt.table = expect(tokens, TokenType::IDENTIFIER).value;
    // Add table and columns to symbol table
    symTable.addTable(stmt.table);
    for (const auto& col : stmt.columns) {
        symTable.addColumn(stmt.table, col);
    }
    if (tokens[index].type == TokenType::WHERE) {
        ++index;
        stmt.whereClause = parseConditionExpression(tokens,
index).release();
    } else {
        stmt.whereClause = nullptr;
    }
    if (tokens[index].type == TokenType::SEMICOLON)
        ++index;
    return stmt;
}
```

2. Symbol Table Class (from **symbol_table.h** and **symbol_table.cpp**):

```
// symbol_table.h
class SymbolTable {
public:
    void addTable(const std::string& tableName);
    void addColumn(const std::string& tableName, const std::string&
columnName);
    bool hasTable(const std::string& tableName) const;
```

```

        std::vector<std::string> getColumns(const std::string& tableName)
const;
        void print(std::ostream& out = std::cout) const;
private:
        std::unordered_map<std::string, std::unordered_set<std::string>>
tableColumns;
};
// symbol_table.cpp
void SymbolTable::addTable(const std::string& tableName) {
    if (tableColumns.find(tableName) == tableColumns.end()) {
        tableColumns[tableName] = std::unordered_set<std::string>();
    }
}

void SymbolTable::addColumn(const std::string& tableName, const
std::string& columnName) {
    addTable(tableName);
    tableColumns[tableName].insert(columnName);
}

bool SymbolTable::hasTable(const std::string& tableName) const {
    return tableColumns.find(tableName) != tableColumns.end();
}

std::vector<std::string> SymbolTable::getColumns(const std::string&
tableName) const {
    std::vector<std::string> columns;
    auto it = tableColumns.find(tableName);
    if (it != tableColumns.end()) {
        for (const auto& col : it->second) {
            columns.push_back(col);
        }
    }
    return columns;
}

void SymbolTable::print(std::ostream& out) const {
    out << "Symbol Table:\n";
    out << "-----\n";
    for (const auto& pair : tableColumns) {
        out << "Table Name: " << pair.first << "\n";
        out << "Columns:\n";
        if (pair.second.size() == 1 && pair.second.count("*") == 1) {

```

```

        out << "  ALL (*)\n";
    } else {
        for (const auto& col : pair.second) {
            out << "    - " << col << "\n";
        }
    }
    out << "-----\n";
}
}

```

3. Tokenizer Function (from **tokenizer.cpp**):

```

vector<Token> tokenize(const string& input) {
    vector<Token> tokens;
    size_t i = 0;
    while (i < input.length()) {
        char c = input[i];
        if (isspace(c)) {
            ++i;
            continue;
        }
        if (isalpha(c)) {
            string word;
            while (i < input.length() && (isalnum(input[i]) || input[i] ==
'_' || input[i] == '-')) {
                word += input[i++];
                TokenType type = keywordToToken(word);
                tokens.push_back({ type, word });
            }
        } else if (isdigit(c)) {
            string number;
            bool dotEncountered = false;
            while (i < input.length() && (isdigit(input[i]) || (input[i] ==
'.' && !dotEncountered))) {
                if (input[i] == '.') {
                    dotEncountered = true;
                }
                number += input[i++];
            }
            tokens.push_back({ TokenType::NUMBER, number });
        } else if (c == '\\') {
            string str;
            ++i; // skip opening quote
            while (i < input.length() && input[i] != '\\') {

```

```

        str += input[i++];
    }
    ++i; // skip closing quote if present
    tokens.push_back({ TokenType::STRING, str });
} else if (c == ',') {
    tokens.push_back({ TokenType::COMMA, "," }); ++i;
} // ... more token cases ...
else {
    tokens.push_back({ TokenType::UNKNOWN, string(1, c) }); ++i;
}
}
tokens.push_back({ TokenType::END, "" });
return tokens;
}

```

4. AST Structures (from **ast.h**):

```

struct ValueWithType {
    std::string value;
    TokenType type;
};

struct ConditionExpression {
    ValueWithType left;
    std::string op;
    ValueWithType right;
    std::string logicalOp; // "AND", "OR", or empty
    std::unique_ptr<ConditionExpression> next;
};

struct SelectStatement {
    std::vector<std::string> columns;
    std::string table;
    ConditionExpression* whereClause;
};

```