Phase 02 Report: Syntax Analysis

a. Formal Grammar (EBNF Notation)

The syntax analyzer validates SQL queries based on the following grammar written in Extended Backus–Naur Form (EBNF):

- **Non-Terminals** appear in *CamelCase*.
- **Terminals** appear in **UPPERCASE** or as **"literals"**.
- { ... } means repetition (zero or more).
- [...] means optional parts.

```
Query         ::= Statement { Statement }
Statement     ::= (CreateStmt | InsertStmt | SelectStmt | UpdateStmt | DeleteStmt) ";"

# DDL: Create Table
CreateStmt    ::= "CREATE" "TABLE" IDENTIFIER "(" ColumnDefList ")"
ColumnDefList::= ColumnDef { "," ColumnDef }
ColumnDef     ::= IDENTIFIER DataType
DataType      ::= "INT" | "FLOAT" | "TEXT"

# DML: Insert
InsertStmt    ::= "INSERT" "INTO" IDENTIFIER "VALUES" "(" ValueList ")"
ValueList     ::= Value { "," Value }
Value         ::= STRING | INTEGER | FLOAT

# DML: Select
SelectStmt    ::= "SELECT" SelectList "FROM" IDENTIFIER [WhereClause]
SelectList    ::= "*" | ColumnList
ColumnList    ::= IDENTIFIER { "," IDENTIFIER }

# DML: Update
UpdateStmt    ::= "UPDATE" IDENTIFIER "SET" IDENTIFIER "=" Value [WhereClause]

# DML: Delete
DeleteStmt    ::= "DELETE" "FROM" IDENTIFIER [WhereClause]

# Conditional Logic
WhereClause   ::= "WHERE" Condition
Condition     ::= Term { "OR" Term }
Term          ::= Factor { "AND" Factor }
Factor        ::= "NOT" Factor | "(" Condition ")" | Comparison
Comparison    ::= Operand OPERATOR Operand
Operand       ::= IDENTIFIER | Value
```

b. Parsing Technique Implemented

Parsing Method: Recursive Descent Parsing

Reasons for Choosing It:

1. **Direct Mapping of Grammar to Code:**

Each grammar rule corresponds to one parsing function (e.g., `parse_select_stmt()`), improving readability and debugging.

2. **No External Tools Required:**

The project forbids parser generators (e.g., Yacc, ANTLR). Recursive Descent can be fully handwritten using plain Python.

3. **Easy to Extend:**

It allows custom error handling, clear tree construction, and integrates well with later phases such as Semantic Analysis.

c. Structure of the Generated Parse Tree

The parse tree is built using the `ParseNode` class defined in `parser.py`.

Class Structure

- **ParseNode**
  - **name:** Node label (e.g., “SelectStmt”, “WhereClause”).
  - **value:** Token value for leaf nodes; None for rule nodes.
  - **children:** A list of nested `ParseNode` objects forming the hierarchy.

Example Tree (for the query `SELECT name FROM students;`)

- ParseNode("Query")
  - ParseNode("Statement")
    - ParseNode("SelectStmt")
      - ParseNode("KEYWORD", "SELECT")
      - ParseNode("IDENTIFIER", "name")
      - ParseNode("KEYWORD", "FROM")
      - ParseNode("IDENTIFIER", "students")
    - ParseNode("SEMICOLON", ";")

d. Syntax Error Detection and Recovery

The parser includes robust error-handling mechanisms.

Error Detection

Errors are detected when:

- The `match()` function encounters an unexpected token.
- A grammar rule expects a specific token (e.g., `FROM`) but finds another (e.g., `INSERT`).

When this happens, a `SyntaxError` is raised with:

- Line number
- Column number
- A descriptive message (e.g., *“Expected ';' but found 'INSERT'”*)

Panic Mode Recovery

To continue parsing after an error, the parser uses **Panic Mode Recovery**:

1. The error is caught and added to an error list.
2. `panic_mode()` is called.
3. The parser skips tokens until it finds a synchronizing symbol — in this case, the semicolon `;`.
4. Parsing resumes from the next statement instead of terminating the program.

Github Repo : https://github.com/MohamedElsadany56/MiniSQLCompiler