SQL Lexer Project Report

1. Implemented Tokens

Our lexer reads SQL-like input and converts it into a sequence of tokens that represent keywords, identifiers, literals, operators, and delimiters.

Implemented Token Types:

Token Type	Examples Detected
KEYWORD	SELECT, FROM, WHERE, INSERT, INTO, VALUES, UPDATE, SET, DELETE, CREATE, TABLE, INT, FLOAT, TEXT, AND, OR, NOT
IDENTIFIER	employees, name, salary, age, departments, id, title, budget, etc.
INTEGER	25, 30, 5000, 6000, 20
FLOAT	3.14, 12.5 (if present in input)
STRING	'Ali', 'Engineer', 'Manager', etc.
OPERATOR	=, <, >, <=, >=, !=, +, -, *, /
DELIMITER	(,), ,, ;
COMMENT	Handled both -- single-line and ## multi-line ## comments
EOF	End-of-file marker used internally to stop lexing

The lexer successfully distinguishes between keywords and identifiers using a keyword set.
Numeric values are categorized into **INTEGER** or **FLOAT** tokens.
String literals are enclosed in single quotes and checked for proper closure.

2. Error Handling

We implemented **robust error handling and recovery** so that the lexer does not stop at the first error.

Handled Error Types:

Error Type	Description	Example
Invalid Character	Detects any illegal symbol not part of SQL (e.g., @, $, ~)	@wrongSymbol = 50;
Unclosed String Literal	Missing closing quote '	('Unclosed, 22, 'Dev);
Unclosed Multi-line Comment	Missing closing ##	## This comment never ends

Recovery Strategy:

- The lexer **records the error message** instead of terminating.
- It **advances the cursor** past the problematic character and **continues lexing**.
- All errors are **stored and displayed** at the end of execution.

Example output (errors section):

```
Errors:
- Error at line 21, column 0: Unclosed comment starting at line 20, column 1
```

This ensures that the lexer can process the full input file and still produce meaningful results and a complete symbol table.

3. Challenges Faced & Fixes

Challenge	Description	Fix Implemented
Premature termination on first error	Initially, the lexer stopped when the first invalid token appeared.	Added try/except inside the lexing loop, collected errors in a list, and used lexer.advance() to continue.
Unclosed string detection	String literals without closing ' caused infinite loops.	Added explicit check and raised an error if no closing ' was found.
Multi-line comment closure	Unclosed ## ... ## comments were not detected properly.	Added line and column tracking to report start of unclosed comment.
Symbol table duplication	Same identifier appeared multiple times.	Used a dictionary to store unique symbol entries.
Whitespace and newline handling	Line/column tracking was incorrect after newlines.	Incremented line number and reset column counter in advance().

4. Sample Output (Excerpt)

```
(KEYWORD, SELECT)

(IDENTIFIER, name)
(DELIMITER, ,)
(IDENTIFIER, age)
(DELIMITER, ,)
(IDENTIFIER, salary)
(KEYWORD, FROM)
(IDENTIFIER, employees)
(KEYWORD, WHERE)
(IDENTIFIER, age)
(OPERATOR, >=)
(INTEGER, 30)
(KEYWORD, AND)
(IDENTIFIER, salary)
(OPERATOR, <)
(INTEGER, 5000)
(DELIMITER, ;)
```

Symbol Table Example:

```
Symbol Table:
name: IDENTIFIER
age: IDENTIFIER
salary: IDENTIFIER
employees: IDENTIFIER
30: INTEGER
5000: INTEGER
'Ali': STRING
25: INTEGER
'Engineer': STRING
'Sara': STRING
'Manager': STRING
6000: INTEGER
20: INTEGER
departments: IDENTIFIER
id: IDENTIFIER
title: IDENTIFIER
budget: IDENTIFIER
'Unclosed, 22, ': STRING
Dev: IDENTIFIER
```

Errors:

```
 Errors:
 - Error at line 21, column 0: Unclosed comment starting at line 20, column 1
PS E:\STUDY\UNI\Semester 7\Compiler Design\Project\sql_compiler\sql_compiler>
```

5. Conclusion

This lexer successfully:

- Recognizes essential SQL tokens.
- Handles both valid and erroneous input gracefully.
- Continues execution after errors.
- Produces a complete symbol table.

Through this project, we learned practical aspects of lexical analysis, error recovery, and building reliable compiler components.