National University of Computer and Emerging Sciences

FAST University Islamabad Campus

BSCS-22-Semester Course Section: **E**

Subject: Compiler Construction

Assignment No. 1

Lexical Analyzer

Submitted by:	**Salman Khan**	**22I-1285**
	Arshiq Rehman	**22I-1023**

Submitted to: **Ms. Nirmal**

February 20th, 2025

1. Overall Overview and Working

The lexical analyzer reads a C++ source file (`input.cpp`), scans it character by character, and emits a stream of tokens while building a symbol table and collecting any lexical errors. It is built around multiple small DFAs (for identifiers, keywords, numbers, strings, comments, operators, delimiters, etc.), an explicit symbol table to record declarations/literals, and an error handler that tracks line numbers for precise diagnostics.

2. Language Rules

- **C++ Subset Supported**:
  - Primitive types: `int`, `float`, `double`, `char`, `string`, `bool`
  - Control keywords: `for`, `if`, `else`, `while`, `switch`, `case`
  - Constant declaration: `const`
- **Lexical conventions** follow standard C++:
  - Identifiers start with letter or underscore, followed by letters, digits, or underscores.
  - Case-sensitivity enforced (uppercase in identifiers triggers an error).
  - Character literals in single quotes, string literals in double quotes.
  - Single-line (`//…`) and multi-line (`/*…*/`) comments are skipped.

3. Data Types Usage

- The symbol table recognizes built-in types via a `Set<String> dataTypes`.
- Upon encountering a type keyword, the analyzer enters "declaration mode" (tracks `expectedType`) so that the next identifier is recorded with that type.
- Numeric literals (`NUMBER`, `DECIMAL`) are parsed into `BigDecimal`, rounded to five decimal places, and stored as `double` constants.

- Boolean literals (`true`/`false`) are stored with type `bool`.

- String and char literals are recorded with their own entries.

4. Language Understanding

- **Keywords vs. Identifiers**: Each keyword has its own DFA; only exact matches are emitted as `KEYWORD` tokens. Anything matching the identifier DFA but not a keyword is an `IDENTIFIER`.

- **Literal Recognition**: Separate DFAs for numbers, decimals, strings, and chars ensure correct lexing of each literal type.

- **Operator vs. Delimiter**: Operators (`+`, `-`, `*`, `/`, `%`, `=`, `**`) are recognized separately from delimiters (`;`, `{`, `}`, `(`, `)`, `,`).

5. Keyword Selection

- The set of C++ keywords is explicitly enumerated (`int`, `float`, …, `case`).

- On seeing `const`, the analyzer flags the next declaration as constant (`expectingConst = true`).

- Control-flow keywords are recognized and emitted but not further processed by the lexer.

6. Lexical Analyzer

- **Architecture**: A central `LexicalAnalyzer` class holds:

    1. One DFA per token category.

    2. A `SymbolTable` instance for tracking declarations and literals.

    3. An `ErrorHandler` for collecting errors with line numbers.

    4. A scope stack (`Deque<String>`) to manage nested blocks.

- **Main Loop**:

    1. Skip whitespace and update line count on `\n`.
    2. Attempt to match char literal, string literal, comments (multi then single).
    3. Call `processNextToken` to match delimiters, operators, keywords, identifiers, decimals, numbers.
    4. On match, emit a `Token`, handle it (update symbol table or scope), and advance the input pointer.
    5. On no match, report an “Invalid character” error and advance one character.

7. Lexical Analyzer Rules

- **Maximal Munch**: Each DFA simulation returns the longest accepted lexeme.
- **Priority Order**:

    1. Char literal
    2. String literal
    3. Multi-line comment
    4. Single-line comment
    5. Delimiter
    6. Operator
    7. Keyword
    8. Identifier (with special check for `true`/`false`)
    9. Decimal
    10. Number

- **Error on Uppercase in Identifiers**: Identifiers must match `[a-z_][a-z0-9_]*`; otherwise an error is flagged.

8. Pre-processing

- **Whitespace Skipping**: All spaces, tabs, newlines (tracked for error reporting) are ignored except for line counting.
- **Comment Removal**:
  - Multi-line comments: DFA recognizes `/*…*/`, counts embedded newlines to keep line numbers accurate.
  - Single-line comments: DFA recognizes `//…\n` and skips to end of line.

9. Precision of Error Identification

- The `ErrorHandler` tracks the current line.
- On any lexical violation (invalid character, uppercase identifier, unmatched `}`), it records an error message with the exact line number.
- At end of analysis, all errors are printed with their line locations.

10. Tokenizer Working

- **DFA Simulation**: The method `simulateDFA(DFA, input, startIndex)` walks the DFA as far as possible, remembering the last accepting state to implement maximal munch.
- **Token Creation**: `processNextToken` wraps the matched lexeme in a `Token(type, value)` object, which is then passed to `handleToken` for side-effects (symbol table entry, scope update, error checks).

11. Local and Global Scope Handling

- A `Deque<String> scopeStack` begins with `"global"`.

- On `{` (either operator or delimiter), a new `"block_n"` scope is pushed.

- On `}`, the top scope is popped (unless it's the global scope, in which case an unmatched-brace error is reported).

- All symbol-table entries carry the current scope, enabling later semantic analysis to distinguish local vs. global identifiers.