**CS-424(Compiler Construction)**

**Assignment#1**

**Syed Ammar Bin Farrukh**

**Reg#2020468**

1. **Design Decisions:**

**Language Specifications**:

The scanner was designed to adhere closely to the language specifications provided for MiniLang, including data types, operators, keywords, identifiers, literals, and comments.

**Finite State Machine (FSM)**:

The scanner employs a simple FSM approach to tokenize the input file. Each token type corresponds to a specific state, and transitions between states are determined by regular expressions matching the input.

**Error Handling**:

The scanner is equipped to detect and report lexical errors, such as invalid symbols or malformed identifiers, providing informative messages to aid in debugging.

**Python Implementation**:

Python was chosen for its simplicity and readability, making it suitable for implementing a scanner efficiently.

1. **Scanner Structure:**

The scanner consists of a single Python class named Scanner. It utilizes regular expressions defined in a dictionary (TOKEN\_TYPES) to match and identify different token types. The scan() method iterates through the input file line by line, tokenizing each line and storing the tokens in a list. Lexical errors are detected and reported during scanning. The display\_tokens() method outputs the list of tokens to the console.

1. **How to Run the Program:**

* Save the provided Python code in a file named scanner.py.
* Ensure you have Python installed on your system.
* Run the program by executing the command python scanner.py.
* Enter the filename of the MiniLang source code file when prompted.
* The scanner will tokenize the input file and display the list of tokens.

1. **Test Cases:**

**Valid Input**:

Test the scanner with MiniLang source code containing valid tokens, including various data types, operators, keywords, identifiers, literals, and comments.

**Invalid Input**:

Test the scanner with input files containing invalid symbols, malformed identifiers, and other lexical errors. Ensure that appropriate error messages are displayed.

**Edge Cases**:

Include test cases with edge scenarios such as empty input files, files with single-line and multi-line comments, files with a mix of different token types in various orders.

**Example Test Cases:**

**Valid Input**:

* MiniLang code containing arithmetic expressions, variable assignments, if-else conditions, print statements, and valid identifiers/literals.
* Expected outcome: The scanner successfully tokenizes the input file without errors.

**Invalid Input**:

* MiniLang code with invalid symbols or malformed identifiers.
* Expected outcome: The scanner detects lexical errors and reports them with clear error messages, indicating the line number where the error occurred.

**Edge Cases**:

* Empty input file.
* Input file with only comments.
* Input file with single-line and multi-line comments.
* Input file with a mix of different token types in various orders.
* Expected outcome: The scanner handles these edge cases gracefully, producing the correct tokenization results and error messages when necessary.

1. **Conclusion:**

The scanner for MiniLang has been designed and implemented according to the provided specifications. It demonstrates the capabilities of lexical analysis and serves as a foundation for subsequent stages of compiler design.

By following the provided instructions, students can gain insights into lexical analysis complexities, explore tokenization processes, and develop an understanding of compiler construction principles.