Documentation

Name: Shaheer Nashad

Reg No: 2020442

Subject: CS424

Design Decisions:

1. **Regular Expressions:** Regular expressions are used to tokenize the MiniLang source code efficiently. This approach allows for easy pattern matching of keywords, identifiers, literals, operators, and comments.
2. **Tokenization:** Each token consists of a token type and the corresponding lexeme. This design choice allows for clear identification of tokens and their respective values.
3. **Error Handling:** Lexical errors, such as invalid tokens, are detected during the scanning process and reported with detailed error messages indicating the line number and the nature of the error.
4. **Scanner Structure:** The scanner is implemented as a class in Python, encapsulating the functionality required for reading source code from a file, tokenizing it, recognizing tokens, and handling lexical errors.

Scanner Structure:

The scanner consists of two main components:

1. Token Types: A class TokenType is defined to store constants representing the different types of tokens supported by MiniLang.
2. Scanner Class: The Scanner class is responsible for scanning the MiniLang source code, tokenizing it, and handling lexical errors. It contains the following methods:
   - __init__(self, filename): Initializes the scanner with the filename of the MiniLang source code file.
   - scan(self): Scans the MiniLang source code, tokenizes it, and stores the tokens in a list.
   - get_tokens(self): Returns the list of tokens generated by the scanner.

Running the Program:

To run the MiniLang scanner program:

1. Save the provided Python code in a file, for example, minilang_scanner.py.
2. Create a MiniLang source code file with the .mini extension, for example, minilang_code.mini, and write your MiniLang code in it.
3. Open a terminal or command prompt.
4. Navigate to the directory containing the Python file and the MiniLang source code file.
5. Run the Python script using the command: python minilang_scanner.py.
6. The program will tokenize the MiniLang source code, display the token type and lexeme for each token, and report any lexical errors encountered.

Test Cases:

Test cases should cover various aspects of the MiniLang scanner's capabilities, including:

- Valid MiniLang code with different combinations of keywords, operators, identifiers, and literals.
- Comments in the MiniLang code.
- Lexical errors such as invalid tokens or malformed identifiers.

Here's an example of test cases:

1. Valid MiniLang Code:

   ```
   x = 10

   y = 5

   sum = x + y

   print sum
   ```

2. MiniLang Code with Comments:

   ```
   // This is a comment
   x = 10 // Another comment
   ```

3. Lexical Error - Invalid Token:

   ```
   z = @
   ```

4. Lexical Error - Malformed Identifier:

   ```
   123z = 10
   ```

Running the MiniLang scanner with these test cases should demonstrate its ability to tokenize valid MiniLang code, handle comments, and report lexical errors accurately.