Topic: Lexer & Scanner

Course: Formal Languages & Finite Automata

### Author: Ungureanu Vlad

Overview

The term lexer comes from lexical analysis which, in turn, represents the process of extracting lexical tokens from a string of characters. There are several alternative names for the mechanism called lexer, for example tokenizer or scanner. The lexical analysis is one of the first stages used in a compiler/interpreter when dealing with programming, markup or other types of languages. The tokens are identified based on some rules of the language and the products that the lexer gives are called lexemes. So basically the lexer is a stream of lexemes. Now in case it is not clear what's the difference between lexemes and tokens, there is a big one. The lexeme is just the byproduct of splitting based on delimiters, for example spaces, but the tokens give names or categories to each lexeme. So the tokens don't retain necessarily the actual value of the lexeme, but rather the type of it and maybe some metadata.

Objectives:

1. Understand what lexical analysis [1] is.
2. Get familiar with the inner workings of a lexer/scanner/tokenizer.
3. Implement a sample lexer and show how it works.

Implementation description

The provided code implements a lexer for arithmetic expressions. A lexer breaks down expressions into tokens, such as numbers and operators. This lexer helps analyze and process arithmetic expressions, demonstrating key concepts in language processing and object-oriented design.

Code Example:

```
class Grammar:
    from enum import Enum
from typing import List

# Enum defining different types of tokens
class TokenType(Enum):
    NUMBER = 0        # Represents numeric values
    OPERATOR = 1      # Represents arithmetic operators (+, -, *, /)
    LEFT_PAREN = 2    # Represents left parenthesis '('
    RIGHT_PAREN = 3   # Represents right parenthesis ')'
    WHITESPACE = 4    # Represents whitespace characters
    ERROR = 5         # Represents an error token
```

```
# Class representing a token with its type, value, and position in the input string
class Token:
    def __init__(self, type: TokenType, value: str, position: int):
        self.type = type
        self.value = value
        self.position = position

    def __str__(self):
        return f"[{self.type}: {self.value}, position: {self.position}]"

# Class responsible for lexing arithmetic expressions, breaking them down into tokens
class ArithmeticLexer:
    def __init__(self, ignore_whitespace: bool = False):
        self.ignore_whitespace = ignore_whitespace

    def tokenize(self, input: str) -> List[Token]:
        tokens = []  # List to store tokens
        current_token = ''  # String to build the current token
        current_position = 0  # Current position in the input string
        left_paren_count = 0  # Count of left parentheses
        right_paren_count = 0  # Count of right parentheses
        last_digit_starting_position = 0  # Starting position of the last digit encountered

        # Iterate through each character in the input string
        for c in input:
            if c.isspace():
                # If whitespace should be ignored, skip to the next character
                if self.ignore_whitespace:
                    current_position += 1
                    continue
                # Add the current number token if any, and add the whitespace token
                self.add_number_if_needed(current_token, tokens, last_digit_starting_positio
                tokens.append(Token(TokenType.WHITESPACE, c, current_position))
            elif c in ['+', '-', '*', '/']:
                # Add the current number token if any, and add the operator token
                self.add_number_if_needed(current_token, tokens, last_digit_starting_positio
                tokens.append(Token(TokenType.OPERATOR, c, current_position))
            elif c.isdigit():
                # If the current token is empty, update the starting position of the last d
                if not current_token:
                    last_digit_starting_position = current_position
                # Append the digit to the current token
                current_token += c
            elif c == '(':
                # Add the current number token if any, and add the left parenthesis token
                self.add_number_if_needed(current_token, tokens, last_digit_starting_positio
                tokens.append(Token(TokenType.LEFT_PAREN, c, current_position))
                left_paren_count += 1
            elif c == ')':
                # Add the current number token if any, and add the right parenthesis token
                self.add_number_if_needed(current_token, tokens, last_digit_starting_positio
                right_paren_count += 1
                # Check if there are more right parentheses than left parentheses
                if right_paren_count > left_paren_count:
                    return self.invalid_token_error(c, current_position)
                tokens.append(Token(TokenType.RIGHT_PAREN, c, current_position))
            else:
                return self.invalid_token_error(c, current_position)
            current_position += 1  # Move to the next position in the input string

        # Check if the number of left parentheses matches the number of right parentheses
        if left_paren_count != right_paren_count:
            return self.invalid_token_error("Mismatched parentheses", len(input))

        # Add the current number token if any
        self.add_number_if_needed(current_token, tokens, last_digit_starting_position)

        return tokens  # Return the list of tokens

    # Method to add the current number token to the list if it's not empty
    def add_number_if_needed(self, current_token, tokens, last_digit_starting_position):
        if current_token:
            tokens.append(self.create_token(current_token, last_digit_starting_position))
            current_token = ''  # Clear the current token

    # Method to create a token from the provided string and position
    def create_token(self, token_string, position):
        if token_string.isdigit():
            return Token(TokenType.NUMBER, token_string, position)
        else:
            return Token(TokenType.ERROR, f"Invalid expression: {token_string} at position

    # Method to create an error token with the provided message and position
    def invalid_token_error(self, token, position):
        return [Token(TokenType.ERROR, f"Invalid expression: {token} at position {position}

# Example usage
lexer = ArithmeticLexer()
tokens = lexer.tokenize("3 + 4 * (2 - 1)")
for token in tokens:
    print(token)
```

For some test

```python
from arithmetic_lexer import ArithmeticLexer

lexer = ArithmeticLexer()
expressions = [
    "3 + 4 * (2 - 1)",
    "10 / (5 - 3) + 2",
    "1 + 2 + 3 + 4 + 5",
    "10 / 0"
]

for expression in expressions:
    tokens = lexer.tokenize(expression)
    print(f"Expression: {expression}")
    print("Tokens:")
    for token in tokens:
        print(token)
    print()
```

Conclusion

The Python code for an arithmetic expression lexer demonstrates several key concepts in language processing and programming:

1. **Tokenization:** The lexer breaks down arithmetic expressions into tokens, such as numbers, operators, and parentheses, facilitating further analysis and processing.

2. **Error Handling:** The implementation includes error detection and reporting, generating error tokens for issues like mismatched parentheses or unexpected characters.

3. **Object-Oriented Design:** The code uses classes and enums to organize and represent tokens and lexer functionality, showcasing the benefits of structured and modular design.

4. **Flexibility:** The lexer offers flexibility by allowing the option to ignore whitespace, enhancing its usability in different contexts.

5. **Readability and Maintainability:** The code is designed to be clear, concise, and easy to understand, following best practices in writing clean and maintainable code.

6. **Testing and Validation:** An example usage demonstrates how to create an instance of the lexer and tokenize an arithmetic expression, highlighting the importance of testing and validating the lexer's functionality.

Overall, the code provides a practical implementation of a lexer for arithmetic expressions, illustrating fundamental concepts in language processing and programming.