**Lexer & Scanner**

**Course: Formal Languages & Finite Automata**

**Author: Vladlen Mîțu**

**Theory:**

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 is.
2. Get familiar with the inner workings of a lexer/scanner/tokenizer.
3. Implement a sample lexer and show how it works.

Note**:** Just because too many students were showing me the same idea of lexer for a calculator, I've decided to specify requirements for such case. Try to make it at least a little more complex. Like, being able to pass integers and floats, also to be able to perform trigonometric operations (cos and sin). But it does not mean that you need to do the calculator, you can pick anything interesting you want.

**Implementation description:**

This code defines a simple lexical analyzer (lexer) for tokenizing mathematical expressions.

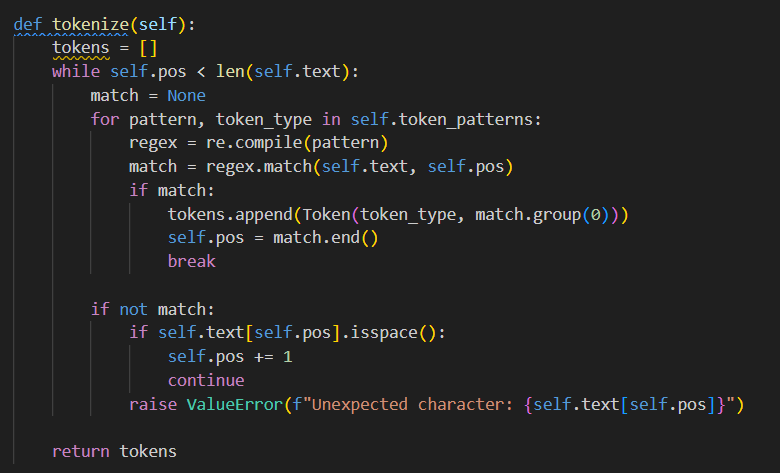
1. **TokenType Enum**: Defines the types of tokens the lexer can recognize, such as numbers, operators (+, -, \*, /), parentheses, and functions (sin, cos).
2. **Token Class**: Represents a token with a type (from TokenType) and a value (the matched text). It includes a \_\_repr\_\_ method for easy debugging and display.
3. **Lexer Class**:
   * Initialization: Takes an input string (text) and initializes the position (pos) to 0. It also defines a list of regular expression patterns (token\_patterns) to match different token types.
   * Token Patterns: Each pattern corresponds to a TokenType, such as numbers, operators, or functions.

This setup allows the lexer to process an input string and identify its components (tokens) for further processing, such as parsing or evaluation.

****

The tokenize method in the Lexer class converts an input string (self.text) into a list of tokens based on predefined patterns. Here's how it works:

1. Initialization: Starts with an empty list of tokens and processes the input string from the current position (self.pos).
2. Pattern Matching: Iterates through a list of regular expression patterns (self.token\_patterns) to find a match at the current position.
   * If a match is found, it creates a Token object with the matched value and token type, adds it to the token list, and updates the position to the end of the match.
3. Whitespace Handling: If no match is found but the current character is whitespace, it skips the whitespace and continues.
4. Error Handling: If no match is found and the character is not whitespace, it raises an error for an unexpected character.
5. Return: Returns the list of tokens after processing the entire input string.



**Conclusion:**

This code defines a Lexer class that performs lexical analysis by tokenizing mathematical expressions containing numbers, arithmetic operators, parentheses, and trigonometric functions (sin and cos). The key components include:

1. **Token Classification**: The TokenType enumeration categorizes tokens, ensuring clarity and extensibility.
2. **Token Representation**: The Token class encapsulates a token’s type and value for structured output.
3. **Lexical Analysis**: The Lexer class scans the input, applying regular expressions to extract tokens while handling spaces and invalid characters.
4. **Example Execution**: A sample input ("30 + sin(1)") demonstrates tokenization, outputting tokens in a structured format.

This implementation serves as a fundamental step in building interpreters or compilers, efficiently processing mathematical expressions and facilitating further syntactic or semantic analysis.

**References:**

1. Else Course FAF.LFA21.1
2. Lexer implementation: https://en.wikipedia.org/wiki/Lexical\_analysis
3. Lexical analysis : https://en.wikipedia.org/wiki/Lexical\_analysis