Topic: Lexer & Scanner

Course: Formal Languages & Finite Automata

Author: Moraru Gabriel

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 [1] 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:

```
# Token types
TOKEN_TYPES = {
    'NUMBER': r'\d+\.\d+|\d+',  # Matches integers and floats
    'PLUS': r'\+',
    'MINUS': r'\-',
    'MULTIPLY': r'\*',
    'DIVIDE': r'/',
    'LEFTPAREN': r'\(',
    'RIGHTPAREN': r'\)',
    'SIN': r'sin',
    'COS': r'cos',
}
```

What It Does:

- This dictionary maps **token names** (like NUMBER, PLUS, SIN) to their **regular expression patterns**.
- Example:
    - NUMBER: Matches integers (5) and floats (5.5).
    - SIN: Matches "sin" in expressions like "sin(30)".

```
Example: ''.join(['ab', 'pq', 'rs']) -> 'ab.pq.rs'
TOKEN_REGEX = '|'.join(f'(?P<{name}>{pattern})' for name, pattern in TOKEN_TYPES.items())
```

What It Does:

- Constructs a single **regular expression** from all token patterns.
- Uses **named capturing groups** (?P<NAME>) to extract matched tokens.

```
class Token:
    def __init__(self, type, value):
        self.type = type
        self.value = value

    def __repr__(self):
        return f"Token({self.type}, {self.value})\n"
```

What It Does:

Represents **a single token** with:

- type: The category of the token (NUMBER, PLUS, etc.).
- value: The actual value from the expression (5.5, sin, +, etc.).

__repr__: Defines how tokens are printed.

```
class Lexer:                    (parameter) text: Any
    def __init__(self, text):
        self.text = text
        self.tokens = []

    def tokenize(self):
        for match in re.finditer(TOKEN_REGEX, self.text):
            token_type = match.lastgroup
            value = match.group()
            if token_type == 'NUMBER':
                value = float(value) if '.' in value else int(value)
            self.tokens.append(Token(token_type, value))
        return self.tokens
```

What It Does:

- Stores:
    - tokens: A list to store extracted tokens.
- Uses **regular expressions** to find tokens in the input text.
- Extracts:
    - token_type: The token category (NUMBER, PLUS, etc.).
- Stores tokens in self.tokens and returns them.

```
if __name__ == "__main__":
    expression = "56 - 2 + sin(50)/cos(50)"
    lexer = Lexer(expression)
    tokens = lexer.tokenize()
    print(f"The input is: {expression}" )
    print(tokens)
```

What It Does:

- **Defines an expression**: "56 - 2 + sin(50)/cos(50)".
- **Creates a Lexer object** and tokenizes the expression.

Output example:

```
The input is: 56 - 2 + sin(50)/cos(50)
[Token(NUMBER, 56), Token(MINUS, -), Token(NUMBER, 2), Token(PLUS, +), Token(SIN, sin), Token(LEFTPAREN, (), Token(NUMBER, 50),
Token(RIGHTPAREN, )), Token(DIVIDE, /), Token(COS, cos), Token(LEFTPAREN, (), Token(NUMBER, 50), Token(RIGHTPAREN, ))]
gabrielmoraru@Morarus-MacBook-Pro Downloads %
```

Conclusion:

This lab provided hands-on experience in lexical analysis by implementing a Python lexer to tokenize mathematical expressions. Using regular expressions, we identified numbers, operators, and functions, showcasing how raw text converts into meaningful tokens. Defining clear tokenization rules ensured accuracy, while Python's `re` module enabled efficient pattern matching. Future improvements could expand functionality and integrate the lexer into a full interpreter or compiler. Overall, this project reinforced key concepts in language processing, essential for building programming languages and computational tools.

References:

1.Else Course FAF.LFA21.1

2. [1] A sample of a lexer implementation

3. [2] Lexical analysis