# 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. A lexer, short for lexical analyzer, is a fundamental component of a compiler or interpreter. Its primary role is to break down the input source code into smaller meaningful units called tokens. These tokens serve as the basic building blocks for subsequent stages of compilation or interpretation.

Here’s an overview of the theory behind lexers:

1. **Tokenization**: The lexer processes the input source code character by character, identifying and categorizing sequences of characters into tokens. Each token represents a distinct lexical unit in the programming language, such as keywords, identifiers, literals, operators, and punctuation symbols.

2. **Regular Expressions**: Lexers typically employ regular expressions to define patterns that match different token types. These patterns describe the syntactic structure of tokens, allowing the lexer to recognize and extract them from the input stream efficiently.

3. **Finite Automata**: Underlying many lexer implementations is the concept of finite automata. Finite automata provide a formal model for recognizing patterns in strings. Lexers often use finite automata to efficiently process input characters and transition between different states based on the encountered symbols.

4. **Token Types**: Tokens are classified into different types based on their syntactic role within the programming language. Common token types include identifiers, keywords, literals (such as numbers and strings), operators, punctuation symbols, and special symbols (such as parentheses and brackets).

5. **Whitespace and Comments**: Lexers typically handle whitespace characters (such as spaces, tabs, and newline characters) and comments separately from other tokens. They may choose to ignore whitespace or include it as a distinct token depending on the language’s syntax requirements.

6. **Error Handling**: Lexers must also handle invalid input gracefully. When encountering unrecognized characters or sequences that do not match any defined token pattern, the lexer may generate error tokens or raise exceptions to indicate lexical errors in the source code.

7. **Output**: Once the lexer has processed the entire input source code, it generates a stream of tokens representing the lexical structure of the program. This token stream serves as input for subsequent stages of compilation or interpretation, such as parsing and semantic analysis.

Overall, the lexer plays a crucial role in the compilation or interpretation process by converting raw source code into a structured representation that can be further analyzed and processed by subsequent components of the compiler or interpreter.

## 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:**

### Part 1: TokenType Enum

```
# Define different types of tokens
class TokenType(Enum):
    NUMBER = 0        # Represents numbers
    OPERATOR = 1      # Represents arithmetic operators (+, -, *, /)
    LEFT_P = 2        # Represents left parenthesis '('
    RIGHT_P = 3       # Represents right parenthesis ')'
    SPACE = 4         # Represents spaces
    ERROR = 5         # Represents errors
```

- This part creates a list of categories for the different parts of the input expression, like numbers, operators, parentheses, spaces, and errors.
- Each category is assigned a number for easy identification.

### Part 2: Token Class

```
# Define a class to represent individual tokens
class Token:
    def __init__(self, type: TokenType, value: str, position: int):
        self.type = type           # Type of token
        self.value = value         # Value of token
        self.position = position   # Position of token in the input string

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

- This part defines a structure for each token found during the analysis of the input.
- Each token has a type (e.g., number, operator), a value (e.g., the actual number or operator symbol), and a position in the input string where it was found.

### Part 3: ArithmeticLexer Class

```
# Define a class to analyze arithmetic expressions
class ArithmeticLexer:
    def __init__(self, ignore_whitespace: bool = False):
        self.ignore_whitespace = ignore_whitespace

    # Tokenize method
```

- This part defines a tool to understand arithmetic expressions.
- It’s prepared to skip spaces during analysis if requested.

## Part 4: Tokenize Method and Helper Methods

```
    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

        # Iterate through each character in the input string
        for c in input:
            if c.isspace():
                # If whitespace should be ignored, skip to the next charac
                if self.ignore_whitespace:
                    current_position += 1
                    continue
            elif c in ['+', '-', '*', '/']:
```

- This part contains the main function that dissects the input string into smaller meaningful parts.
- There are also some additional functions to help with different aspects of this process, like managing numbers, creating tokens, and dealing with errors.

## Part 5: Example Usage

```
# Example of how to use the lexer
lexer = ArithmeticLexer()                                # Create a lexer
tokens = lexer.tokenize("3 + 4 * (2 - 1)")               # Tokenize an ex
for token in tokens:                                     # Iterate over t
    print(token)                                         # Print each tok
```

- This part demonstrates how to use the lexer in practice.
- It begins by creating a lexer instance.
- Then, it tokenizes a sample arithmetic expression.
- Finally, it displays the resulting tokens.

### 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.