

Laboratory work nr.3

Implementing a simple lexer

Course: Formal Languages & Finite Automata
Student: Schipschi Daniil

Objectives:

Understand what lexical analysis is.

Get familiar with the inner workings of a lexer/scanner/tokenizer.

Implement a sample lexer and show how it works.

Implementation tips:

You can find implementation tips on the reference.

Please take into consideration the indicated structure of the project. Find a suitable place for the lexer implementation and for the new report of course.

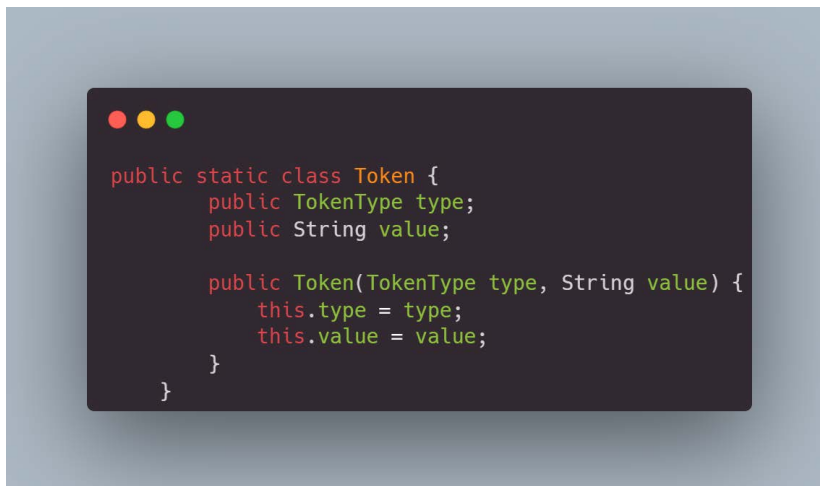
Evaluation:

The project should be located in a public repository on a git hosting service (e.g. Github, Gitlab, BitBucket etc.):

You only need to have one repository.

You can use the lexer implemented for the PBL project, if you have one. It needs to be integrated in this project and documented well.

It should contain an explanation in the form of a Markdown with the standard structure from the ../REPORT_TEMPLATE.md file.



```
public static class Token {
    public TokenType type;
    public String value;

    public Token(TokenType type, String value) {
        this.type = type;
        this.value = value;
    }
}
```

The Token class defines the basic elements recognized during lexical analysis, consisting of a type and corresponding value. It serves as the building block for parsing input, enabling structured handling of different components like numbers, operators, and functions within the lexer, thereby facilitating accurate interpretation of textual expressions.

```

public static List<Token> tokenize(String input) {
    List<Token> tokens = new ArrayList<>();
    input = input.replaceAll("\\s", "");
    int i = 0;
    while (i < input.length()) {
        char ch = input.charAt(i);
        if (Character.isDigit(ch)) {
            StringBuilder sb = new StringBuilder();
            while (i < input.length() && (Character.isDigit(input.charAt(i)) ||
input.charAt(i) == '.')) {
                sb.append(input.charAt(i));
                i++;
            }
            tokens.add(new Token(TokenType.NUMBER, sb.toString()));
        } else if (Character.isLetter(ch)) {
            StringBuilder sb = new StringBuilder();
            while (i < input.length() && Character.isLetter(input.charAt(i))) {
                sb.append(input.charAt(i));
                i++;
            }
            tokens.add(new Token(TokenType.FUNCTION, sb.toString()));
        } else {
            switch (ch) {
                case '(':
                    tokens.add(new Token(TokenType.LPAREN, "("));
                    i++;
                    break;
                case ')':
                    tokens.add(new Token(TokenType.RPAREN, ")"));
                    i++;
                    break;
                case ',':
                    tokens.add(new Token(TokenType.COMMA, ","));
                    i++;
                    break;
                default:
                    tokens.add(new Token(TokenType.OPERATOR,
String.valueOf(ch)));
                    i++;
                    break;
            }
        }
        i++;
    }
    tokens.add(new Token(TokenType.EOF, ""));

    if (!areParenthesesBalanced(tokens)) {
        throw new RuntimeException("Parentheses are not balanced.");
    }

    return tokens;
}

```

The tokenize method processes the input string character by character, generating a list of tokens representing its components. It iterates through the input string, categorizing characters into tokens based on their types such as numbers, operators, parentheses, functions, commas, or end-of-file markers. Numeric values are built character by character into strings, while letters represent function names. Parentheses, commas, and operators are directly identified and added as tokens. Finally, it ensures balanced parentheses and returns the list of tokens, crucial for subsequent stages in parsing mathematical expressions or structured text. If the parentheses are unbalanced, it throws a runtime exception.



```
private static boolean areParenthesesBalanced(List<Token> tokens) {  
    Stack<TokenType> stack = new Stack<>();  
    for (Token token : tokens) {  
        if (token.type == TokenType.LPAREN) {  
            stack.push(TokenType.LPAREN);  
        } else if (token.type == TokenType.RPAREN) {  
            if (stack.isEmpty() || stack.pop() != TokenType.LPAREN) {  
                return false;  
            }  
        }  
    }  
    return stack.isEmpty();  
}
```

The `areParenthesesBalanced` method checks if the parentheses in the list of tokens are balanced. It utilizes a stack data structure to keep track of opening parentheses encountered. During iteration through the tokens, each left parenthesis token (LPAREN) encountered is pushed onto the stack. When a right parenthesis token (RPAREN) is found, it ensures that there is a corresponding left parenthesis on top of the stack. If the stack is empty or the top element is not a left parenthesis, indicating unbalanced parentheses, it returns false. If all parentheses are matched correctly, the method returns true. This function is crucial for ensuring the validity of expressions, as unbalanced parentheses can lead to syntax errors in mathematical expressions or structured text.

CONCLUSION:

In this laboratory, we delved into the construction of a basic lexer, a fundamental component in parsing structured text. By dissecting the code, we grasped the process of tokenization, breaking down input strings into meaningful units such as numbers, operators, and functions. Through careful examination, we understood the importance of token representation encapsulated within the `Token` class, providing a structured approach for handling different elements of input. Moreover, we explored the critical role of balanced parentheses validation, ensuring the syntactic correctness of expressions. This exercise equipped us with foundational knowledge on building lexers, laying the groundwork for further exploration into the realms of parsing and interpreting textual data.