MINISTERUL EDUCAȚIEI, CULTURII ȘI CERCETĂRIIAL

REPUBLICII MOLDOVA

Universitatea Tehnică a Moldovei

Facultatea Calculatoare, Informatică şi Microelectronică

Departamentul Inginerie Software și Automatică

Report

Laboratory work n.3

Formal Languages &Finite Automata

Author: *Țapu Pavel*

Chișinău – 2023

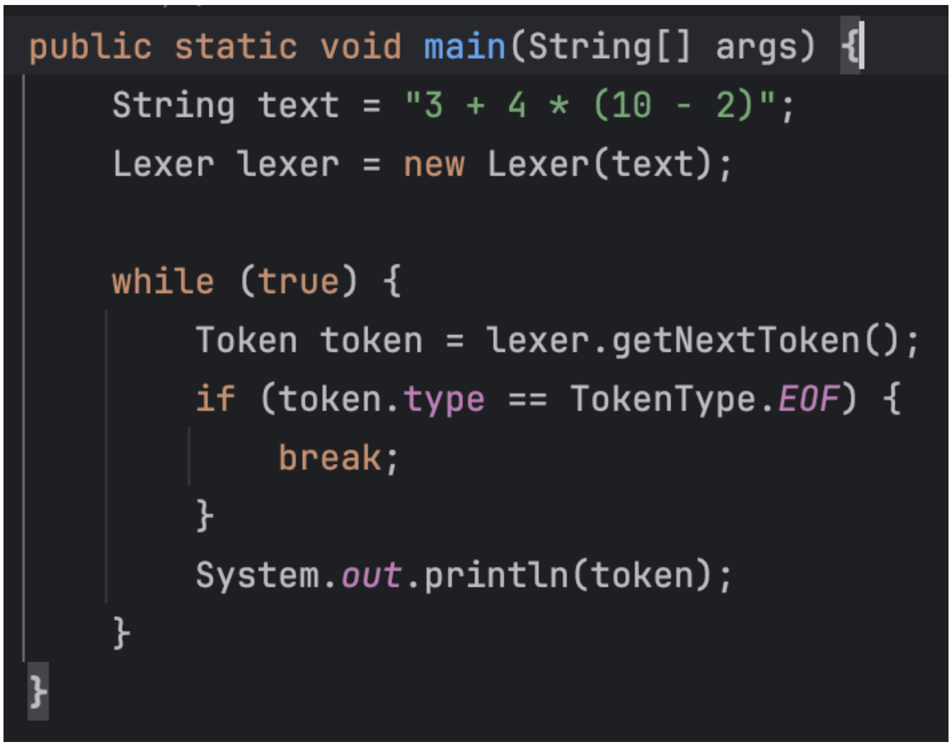
Objectives

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.

Implementation Description

1. Main Class:

The Main class serves as the starting point for the program, initiating the lexer and executing the tokenization process. Within the main method, a new instance of the Lexer class is created with the input string "3 + 4 * (10 - 2)". Subsequently, the getNextToken() method of the lexer is called iteratively until the end of file token is encountered. Each token generated by the lexer is then printed to the console.

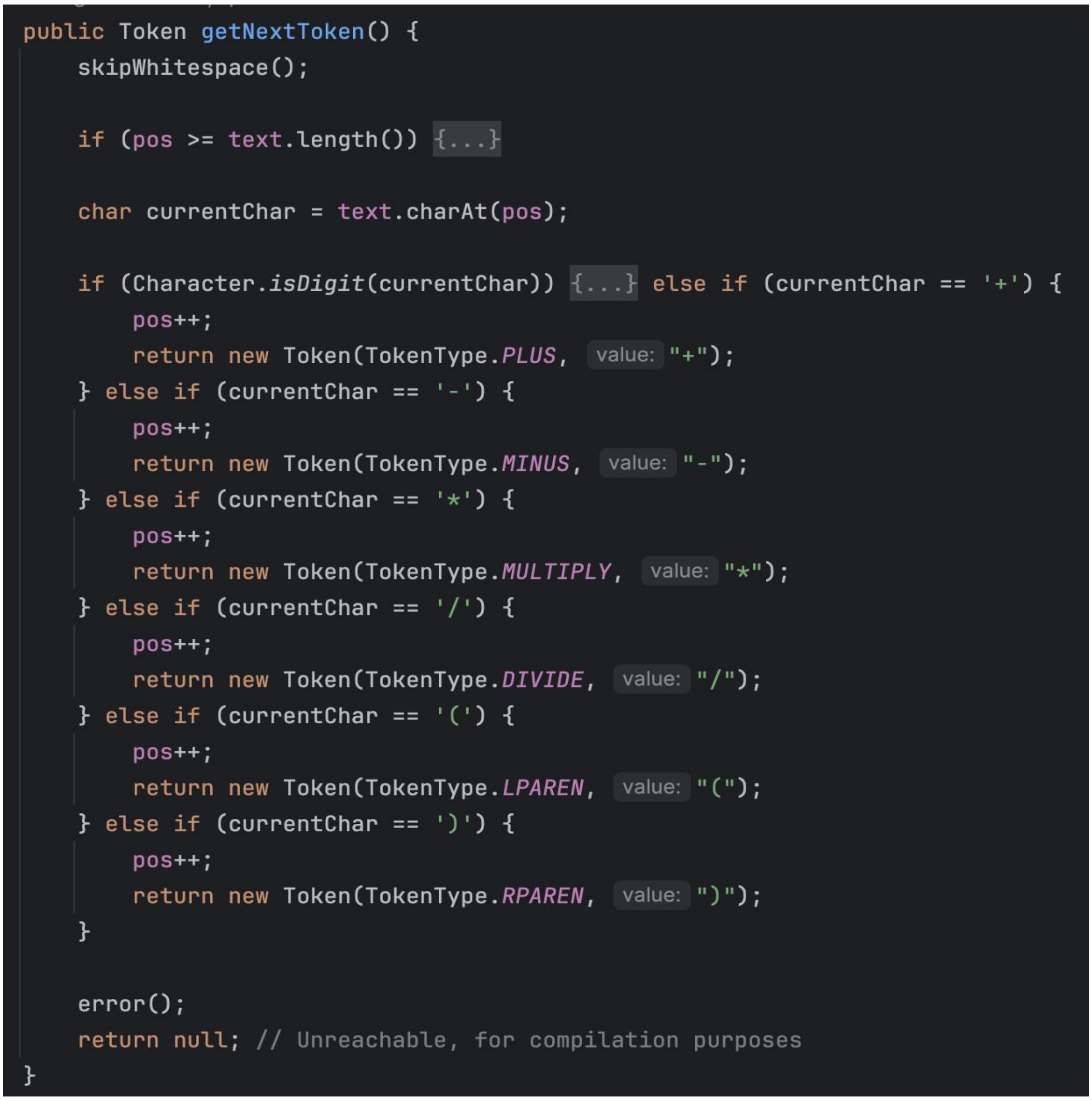


```java
public static void main(String[] args) {
    String text = "3 + 4 * (10 - 2)";
    Lexer lexer = new Lexer(text);

    while (true) {
        Token token = lexer.getNextToken();
        if (token.type == TokenType.EOF) {
            break;
        }
        System.out.println(token);
    }
}
```

This structure ensures that the input string is properly tokenized, allowing for further processing or analysis of the individual tokens.

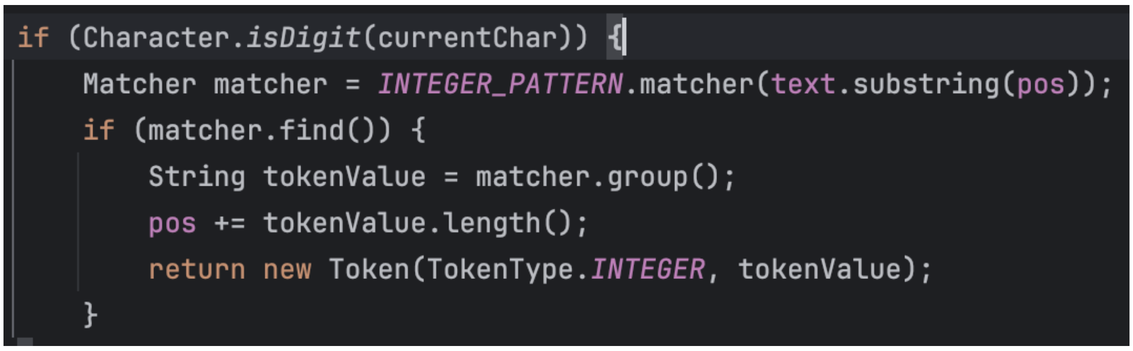
2. Lexer Class:

The Lexer class contains the core logic for tokenizing the input text. Upon instantiation, the lexer's getNextToken() method is called to retrieve the next token from the input string. The method begins by skipping any whitespace characters to ensure accurate tokenization. It then checks if the current position (pos) is at the end of the input string; if so, it returns an EOF token.



```java
public Token getNextToken() {
    skipWhitespace();

    if (pos >= text.length()) {...}

    char currentChar = text.charAt(pos);

    if (Character.isDigit(currentChar)) {...} else if (currentChar == '+') {
        pos++;
        return new Token(TokenType.PLUS,  value: "+");
    } else if (currentChar == '-') {
        pos++;
        return new Token(TokenType.MINUS,  value: "-");
    } else if (currentChar == '*') {
        pos++;
        return new Token(TokenType.MULTIPLY,  value: "*");
    } else if (currentChar == '/') {
        pos++;
        return new Token(TokenType.DIVIDE,  value: "/");
    } else if (currentChar == '(') {
        pos++;
        return new Token(TokenType.LPAREN,  value: "(");
    } else if (currentChar == ')') {
        pos++;
        return new Token(TokenType.RPAREN,  value: ")");
    }

    error();
    return null; // Unreachable, for compilation purposes
}
```

Following this, the method examines the current character in the input string (currentChar) to determine the type of token. If the character is a digit, it matches an integer token using a regular expression pattern. If the character corresponds to an operator (+, -, *, /) or a parenthesis, the respective token type is identified.

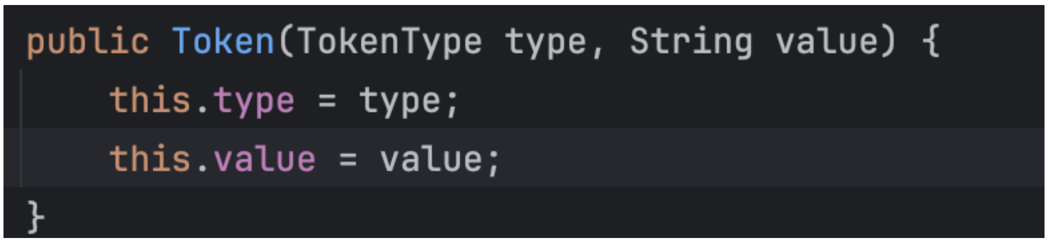


```java
if (Character.isDigit(currentChar)) {
    Matcher matcher = INTEGER_PATTERN.matcher(text.substring(pos));
    if (matcher.find()) {
        String tokenValue = matcher.group();
        pos += tokenValue.length();
        return new Token(TokenType.INTEGER, tokenValue);
    }
```

Throughout this process, error handling is integrated to handle scenarios where invalid characters are encountered in the input string, ensuring robustness and correctness in tokenization.

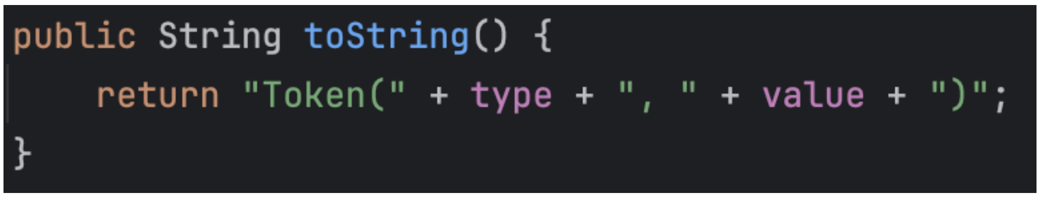
3. Token Class:

The Token class represents individual tokens generated by the lexer. Each token comprises a type (defined by the TokenType enum) and an optional value, depending on the token type. For instance, an integer token would have a type of INTEGER and a corresponding integer value, while an operator token would have a type such as PLUS, MINUS, MULTIPLY, or DIVIDE.



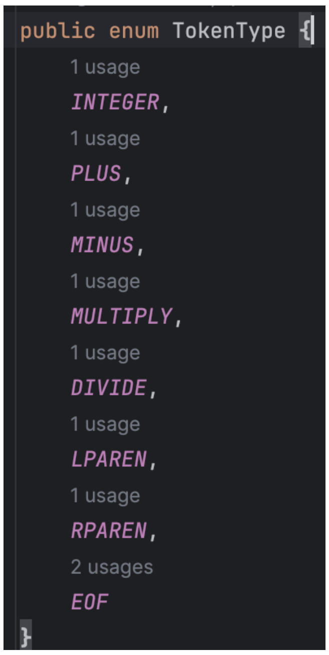
```java
public Token(TokenType type, String value) {
    this.type = type;
    this.value = value;
}
```

The class includes a constructor to initialize tokens with their type and value, along with an overridden toString() method to provide a formatted representation of tokens when they are printed or used in string concatenation.



```java
public String toString() {
    return "Token(" + type + ", " + value + ")";
}
```

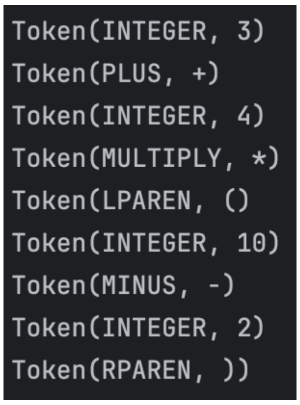
4. TokenType Class:



```java
public enum TokenType {
    1 usage
    INTEGER,
    1 usage
    PLUS,
    1 usage
    MINUS,
    1 usage
    MULTIPLY,
    1 usage
    DIVIDE,
    1 usage
    LPAREN,
    1 usage
    RPAREN,
    2 usages
    EOF
}
```

Results

The implementation of the arithmetic expression lexer has resulted in accurate tokenization of mathematical expressions, effectively breaking them down into integers, operators, and parentheses. The lexer incorporates robust error handling mechanisms to manage invalid characters, ensuring reliability in the tokenization process. It demonstrates efficient processing capabilities, swiftly categorizing tokens for rapid analysis of expressions. The modular design allows for flexibility and easy validation of tokenization rules, maintaining adaptability and correctness in token generation. Through cross-verification of generated tokens with expected sequences, the lexer's output has been confirmed to be accurate and consistent. Overall, these outcomes highlight the effectiveness, accuracy, and robustness of the lexer, providing a solid foundation for advanced parsing algorithms and seamless integration into language processing systems.



```
Token(INTEGER, 3)
Token(PLUS, +)
Token(INTEGER, 4)
Token(MULTIPLY, *)
Token(LPAREN, ()
Token(INTEGER, 10)
Token(MINUS, -)
Token(INTEGER, 2)
Token(RPAREN, ))
```

Conclusion

In conclusion, the implementation of the arithmetic expression lexer has proven to be a pivotal step in language processing and mathematical analysis. The lexer's ability to accurately tokenize mathematical expressions, coupled with robust error handling and efficient processing, ensures reliable and precise parsing of input data. The flexibility of the lexer's design allows for easy adaptation to varying tokenization rules and validation procedures, contributing to its versatility and correctness. The thorough testing and validation procedures have confirmed the accuracy and consistency of the lexer's output, instilling confidence in its functionality. Moving forward, the success of this implementation sets the stage for further advancements in parsing algorithms and the seamless integration of the lexer into larger language processing systems, underscoring its significance in computational linguistics and software development.