MINISTERUL EDUCAȚIEI, CULTURII ȘI CERCETĂRII AL REPUBLICII MOLDOVA

Universitatea Tehnică a Moldovei

Facultatea Calculatoare, Informatică şi Microelectronică Departamentul Inginerie Software și Automatică

CEBAN VASILE FAF-223

Report

Laboratory work n.3

of Formal Languages and Finite Automata

Checked by:

Dumitru Crețu, *university assistant*

DISA, FCIM, UTM

Chișinău - 2024

Lexer Task

Topic: Lexer & Scanner

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.

Source Code:

```java
package Laboratory_work_3;

import java.util.List;

public class Main {
   public static void main(String[] args) {
       String codeText = """
           // This is a single-line comment
           int x = 10;    // Another comment
           float y = 3.14;
           if (x > y) {
               System.out.println("x is greater"); // This won't be tokenized
           }
           """;

       Lexer lexer = new Lexer(codeText);
       List<Token> tokens = lexer.tokenize();

       for (Token token : tokens) {
           System.out.println(token);
       }
   }
}
```

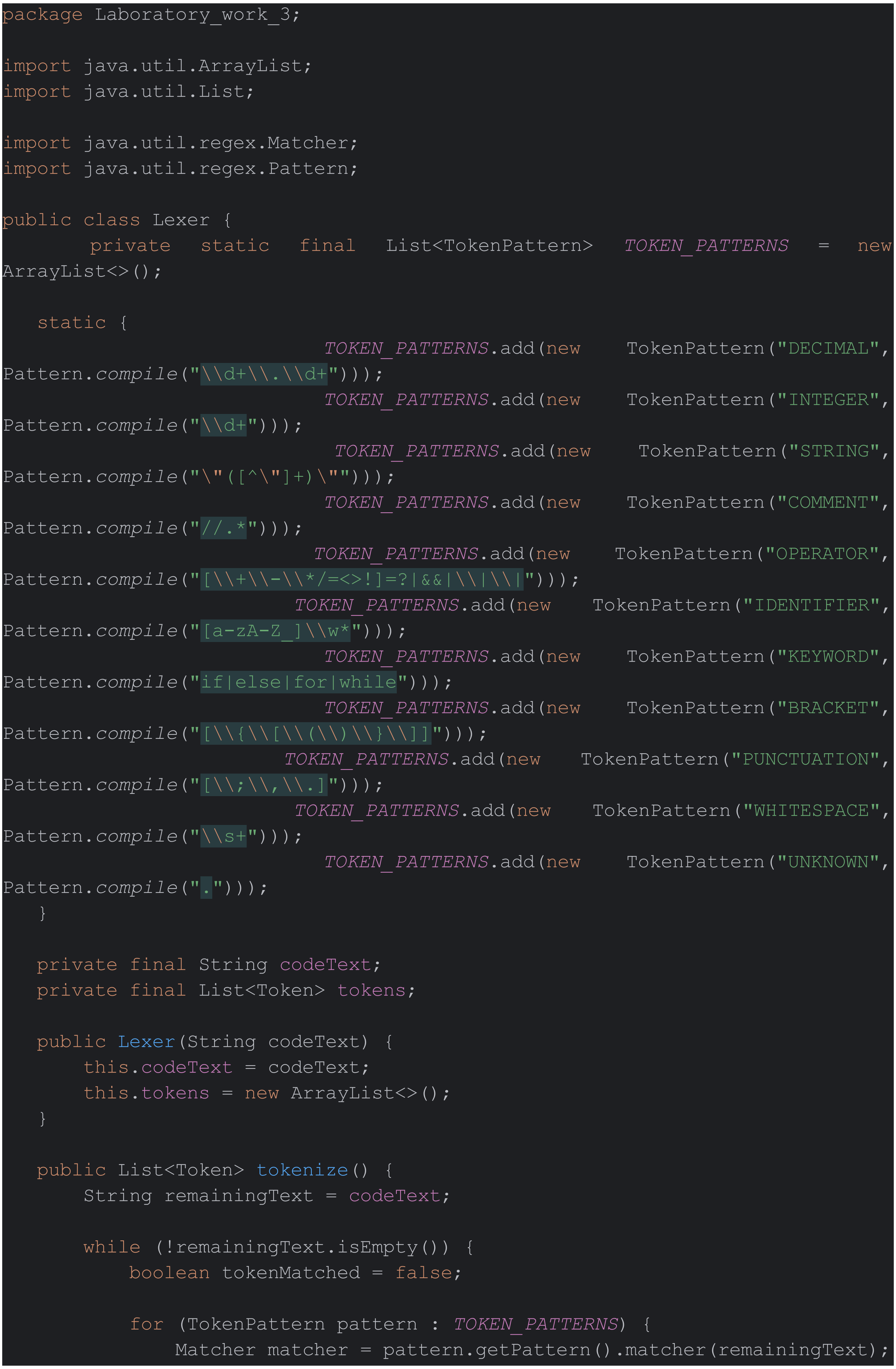
A String variable named codeText is defined that stores the example code.
A new Lexer object is created and passed codeText as a parameter.
The tokenize() method of the lexer object is called to tokenize the example code. This method will return a list of tokens.
Iterate through the list of tokens with a for loop. Each token is displayed on the console using the System.out.println() function.

```java
package Laboratory_work_3;

import java.util.regex.Pattern;

public class TokenPattern {
   private final String type;
   private final Pattern pattern;

   public TokenPattern(String type, Pattern pattern) {
       this.type = type;
       this.pattern = pattern;
   }

   public String getType() {
       return type;
   }

   public Pattern getPattern() {
       return pattern;
   }
}
```

The TokenPattern class is designed to represent a single pattern used within a lexer.
Multiple TokenPattern objects are created, each representing a different token type with the corresponding regular expression. Matching Patterns: As the lexer processes the code, it compares the code to the patterns in the given TokenPattern objects. When a match is found, a new Token object is created using the matching type and text.

```
package Laboratory_work_3;

public class Token {
    private final String type;
    private final String value;

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

    public String getType() {
        return type;
    }

    public String getValue() {
        return value;
    }

    @Override
    public String toString() {
        return "(" + type + ", " + value + ")";
    }
}
```

The lexer creates Token objects to represent the individual elements it identifies within the input code.
Further Processing: These tokens can then be used by other parts of a compiler or interpreter to understand the structure of the code.



```
package Laboratory_work_3;

import java.util.ArrayList;
import java.util.List;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class Lexer {
        private static final List<TokenPattern> TOKEN_PATTERNS = new
ArrayList<>();

   static {
                            TOKEN_PATTERNS.add(new TokenPattern("DECIMAL",
Pattern.compile("\\d+\\.\\d+")));
                            TOKEN_PATTERNS.add(new TokenPattern("INTEGER",
Pattern.compile("\\d+")));
                             TOKEN_PATTERNS.add(new TokenPattern("STRING",
Pattern.compile("\"([^\"]+)\"")));
                            TOKEN_PATTERNS.add(new TokenPattern("COMMENT",
Pattern.compile("//.*")));
                           TOKEN_PATTERNS.add(new TokenPattern("OPERATOR",
Pattern.compile("[\\+\\-\\*/=<>!]=?|&&|\\|\\|")));
                         TOKEN_PATTERNS.add(new TokenPattern("IDENTIFIER",
Pattern.compile("[a-zA-Z_]\\w*")));
                            TOKEN_PATTERNS.add(new TokenPattern("KEYWORD",
Pattern.compile("if|else|for|while")));
                            TOKEN_PATTERNS.add(new TokenPattern("BRACKET",
Pattern.compile("[\\{\\[\\(\\)\\}\\]]")));
                        TOKEN_PATTERNS.add(new TokenPattern("PUNCTUATION",
Pattern.compile("[\\;\\,\\.]")));
                         TOKEN_PATTERNS.add(new TokenPattern("WHITESPACE",
Pattern.compile("\\s+")));
                            TOKEN_PATTERNS.add(new TokenPattern("UNKNOWN",
Pattern.compile(".")));
   }

   private final String codeText;
   private final List<Token> tokens;

   public Lexer(String codeText) {
       this.codeText = codeText;
       this.tokens = new ArrayList<>();
   }

   public List<Token> tokenize() {
       String remainingText = codeText;

       while (!remainingText.isEmpty()) {
           boolean tokenMatched = false;

           for (TokenPattern pattern : TOKEN_PATTERNS) {
               Matcher matcher = pattern.getPattern().matcher(remainingText);
```

```
                if (matcher.find() && matcher.start() == 0) {
                    String value = matcher.group(0);

                              if (!pattern.getType().equals("WHITESPACE") &&
!pattern.getType().equals("COMMENT")) {
                        tokens.add(new Token(pattern.getType(), value));
                    }

                    remainingText = remainingText.substring(value.length());
                    tokenMatched = true;
                    break;
                }
            }

            if (!tokenMatched) {
                throw new RuntimeException("TokenizerError: Unknown token");
            }
        }

        return tokens;
    }
}
```

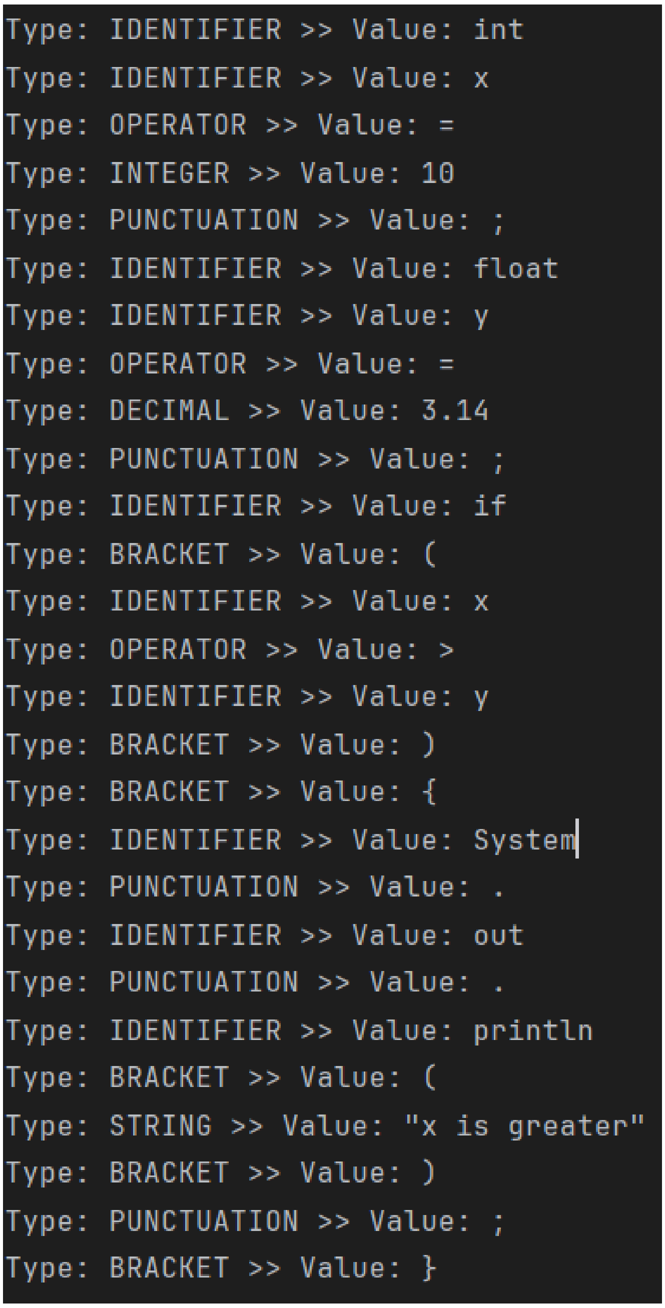
1. Import the necessary List, Matcher, and Pattern classes.
2. TOKEN_PATTERNS:
    - A List to store TokenPattern objects.
    - The static keyword means it belongs to the class itself, not specific instances.
3. Static Initializer (static { ... })
    - A block of code that runs once when the Lexer class is first loaded.
    - Inside, add TokenPattern objects, each defining a token type and its regular expression pattern.
4. Instance Fields
    - codeText: Stores the input code.
    - tokens: A list to hold the tokens as they are generated.
5. Constructor
    - Lexer(String codeText): Initializes a lexer with the code to tokenize.
6. tokenize() Method
    - Core Logic:
        - Takes the input codeText and iterates through it.
        - Attempts to match it against the patterns in TOKEN_PATTERNS.
        - Creates Token objects for successful matches and adds them to the tokens list.
        - Handles whitespace, comments, and unknown tokens.

Testing

Inputs:

```
String codeText = """
   // This is a single-line comment
   int x = 10;    // Another comment
   float y = 3.14;
   if (x > y) {
       System.out.println("x is greater"); // This is a comment
   }
   """;
```

Output:



```
Type: IDENTIFIER >> Value: int
Type: IDENTIFIER >> Value: x
Type: OPERATOR >> Value: =
Type: INTEGER >> Value: 10
Type: PUNCTUATION >> Value: ;
Type: IDENTIFIER >> Value: float
Type: IDENTIFIER >> Value: y
Type: OPERATOR >> Value: =
Type: DECIMAL >> Value: 3.14
Type: PUNCTUATION >> Value: ;
Type: IDENTIFIER >> Value: if
Type: BRACKET >> Value: (
Type: IDENTIFIER >> Value: x
Type: OPERATOR >> Value: >
Type: IDENTIFIER >> Value: y
Type: BRACKET >> Value: )
Type: BRACKET >> Value: {
Type: IDENTIFIER >> Value: System
Type: PUNCTUATION >> Value: .
Type: IDENTIFIER >> Value: out
Type: PUNCTUATION >> Value: .
Type: IDENTIFIER >> Value: println
Type: BRACKET >> Value: (
Type: STRING >> Value: "x is greater"
Type: BRACKET >> Value: )
Type: PUNCTUATION >> Value: ;
Type: BRACKET >> Value: }
```

Figure 1. Result of execution.

Conclusion

The resulting lexer demonstrates the principles of tokenization by correctly identifying various language constructs, including keywords, identifiers, operators, and more. During development, we gained a deeper understanding of regular expressions and their role in pattern matching. In the future, I plan to extend the lexer's capabilities to handle a wider range of language elements and explore integrating it into a simple parser.