Ministerul Educaţiei și Cercetării al Republicii Moldova Universitatea Tehnică a Moldovei

Facultatea Calculatoare, Informatică și Microelectronică

Laboratory work 3:

Lexer & Scanner

Elaborated:

st. gr. FAF-223 Cravcenco Dmitrii

Verified:

asist. univ. Dumitru Cretu

Chişinău - 2024

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

Firstly I designed my basic token types, that my lexer will be able to recognize.

```
enum TokenType {
   TYPE,
   VARIABLE,
   EQUALS,
   VALUE,
   SEMICOLON
}
```

Then I implemented a class that represents a token, which is a pair of TokenType and its value. The value is the actual string that was recognized by the lexer.

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

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

   ...
}
```

Then, as the main part, I implemented lex() method, that is responsible for tokenizing the input string. The method takes a string as input and returns a list of tokens. I also define some valid types of data and regex patterns for the tokens. Then I split the input string into lines and iterate over each line to tokenize it. I also check if the structure of the code is valid, and if not, I throw an exception.

```
public static List<Token> lex(String code) {
      List<Token> tokens = new ArrayList<>();

      List<String> validTypes = List.of("Barcelona", "RealMadrid", "Chelsea");

      // Define regex patterns for the tokens
      Pattern variablePattern = Pattern.compile("\\b[a-z][a-zA-Z]*\\b");
      Pattern equalsPattern = Pattern.compile("=");
      Pattern valuePattern = Pattern.compile("\\b\\d+\\b");
      Pattern semicolonPattern = Pattern.compile(";");

      String[] lines = code.split("(?<=;)");

      int lineNumber = 1;
```

```
        for (String line : lines) {
            List<Token> lineTokens = new ArrayList<>();
            Matcher matcher = getCombinedPattern(validTypes).matcher(line);

            while (matcher.find()) {
                String match = matcher.group();

                if (validTypes.contains(match)) {
                    lineTokens.add(new Token(TokenType.TYPE, match));
                } else if
                ...
                ...
                else {
                    throw new RuntimeException("Unrecognized token '" + match + "' in line " +
lineNumber);
                }
            }

            if (!isValidStructure(lineTokens)) {
                String missingPart = getMissingPart(lineTokens);
                throw new RuntimeException("Error: Invalid code structure in line " + lineNumber +
                        ". " + missingPart + " in line: " + line);
            }

            tokens.addAll(lineTokens);
            lineNumber++;
        }

        return tokens;
    }
```

In lex() method I also use some helper methods, such as getCombinedPattern(), isValidStructure(), getMissingPart(). The first one is responsible for combining the regex patterns for the tokens.

```
private static Pattern getCombinedPattern(List<String> validTypes) {
        String validTypesRegex = String.join("|", validTypes);
        String combinedRegex = "\\b(" + validTypesRegex + ")\\b|[A-Za-z][a-zA-Z]*|=|\\b\\d+\\b|;";
        return Pattern.compile(combinedRegex);
    }
```

The second one is responsible for checking if the structure of the code is valid. I defined a rule that the code should have the following structure: type varName = varValue;

```
private static boolean isValidStructure(List<Token> tokens) {
        // Check if the token sequence represents a valid structure: type varName = varValue;
        return tokens.size() == 5 &&
                tokens.get(0).getType() == TokenType.TYPE &&
                tokens.get(1).getType() == TokenType.VARIABLE &&
                tokens.get(2).getType() == TokenType.EQUALS &&
                tokens.get(3).getType() == TokenType.VALUE &&
```

```
            tokens.get(4).getType() == TokenType.SEMICOLON;
    }
```

The last one is responsible for getting the missing part of the code, if the structure is invalid. It checks every token for two things: if it is in the correct place and if it is missing and returns a text for error message.

```
private static String getMissingPart(List<Token> tokens) {
        StringBuilder missingPart = new StringBuilder(" Token(s) missing: ");
        StringBuilder incorrectPlaceTokens = new StringBuilder(" Token(s) in incorrect place: ");

        if ((!tokens.isEmpty() && tokens.get(0).getType() != TokenType.TYPE)) {
            incorrectPlaceTokens.append("TYPE ");
        }
        if (!Arrays.toString(tokens.stream().map(Token::getType).toArray()).contains("TYPE")) {
            missingPart.append("TYPE ");
        }

        ...


        String result = missingPart.toString().endsWith(": ") ? "" : missingPart + ".";
        result += incorrectPlaceTokens.toString().endsWith(": ") ? "" : incorrectPlaceTokens + ".";
        return result;
    }
```

Conclusions and Results

In this laboratory, I delved into the realm of lexical analysis, gaining a deeper understanding of its significance and the mechanics of a lexer, scanner, or tokenizer. Here's a breakdown of my key takeaways and accomplishments:

- Lexical Analysis Understanding

Through this laboratory, I learned the fundamental concept of lexical analysis, which involves breaking down a sequence of characters into meaningful tokens for further processing.

- Inner Workings of a Lexer

My exploration led me to explore the intricate workings of a lexer, also referred to as a scanner or tokenizer.

- Lexer Implementation

To gain a deeper understanding, I implemented the task of crafting a custom lexer. This task involved analyzing input code, identifying its components based on predefined rules, and emitting corresponding tokens. Through this exercise, I gained insight into the process of code analysis and tokenization.

I gained valuable insights into the underlying mechanisms of lexical analysis. This hands-on experience provided me with a deeper appreciation for the intricacies involved in programming language processing, from l exical analysis to syntactic parsing.