MINISTERUL EDUCAȚIEI, CULTURII ȘI CERCETĂRII AL REPUBLICII MOLDOVA Universitatea Tehnică a Moldovei Facultatea Calculatoare, Informatică şi Microelectronică Departamentul Inginerie Software și Automatică

GLIGA DANIELA, FAF-223

Report

Laboratory work nr.3 Lexical Analysis

of Formal Languages and Finite Automata

Checked by:

Dumitru Crețu, *university assistant*

FCIM, UTM

Chișinău – 2024

1. Theory:

A Lexical Analysis is the first phase of the compiler also known as a scanner. It converts the High level input program into a sequence of Tokens. A scanner (or “lexer”) takes in the linear stream of characters and chunks them together into a series of something more akin to “words”. In programming languages, each of these words is called a token. Some tokens are single characters, like (and ,. Others may be several characters long, like numbers (123), string literals ("hi!"), and identifiers (min).

When it comes to implementing a language, the first thing needed is the ability to process a text file and recognize what it says. The traditional way to do this is to use a “lexer” (aka ‘scanner’) to break the input up into “tokens”. Each token returned by the lexer includes a token code and potentially some metadata (e.g. the numeric value of a number). First, we define the possibilities.

Any finite sequence of alphabets (characters) is called a string. Length of the string is the total number of occurrence of alphabets, e.g., the length of the string tutorialspoint is 14 and is denoted by |tutorialspoint| = 14. A string having no alphabets, i.e. a string of zero length is known as an empty string and is denoted by ε (epsilon).

Lexical analysis is an important component of the compiler design process. It plays a crucial role in the front-end of the compiler, where it is responsible for breaking down the source code into smaller, more manageable units called lexemes. These lexemes serve as the building blocks for the next phase of the compiler design process, called syntax analysis.

The main purpose of lexical analysis is to perform a preliminary analysis of the source code, checking for any lexical errors or issues before they can cause more serious problems later in the compiler design process. This includes checking for things like incorrect spelling, incorrect punctuation, and other errors that could impact the readability of the source code.

Lexical analysis is an essential component of compiler design as it acts as a filter for the source code, ensuring that only valid, well-formed code is passed along to the next phase of the compiler design process.

2. 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.

3. Implementation description:

After analysing what lexical analysis is, especially what the scanner is, I came up with he following plan of implementing a lexer. First of all, I have created an enum class with some common types of symbols which I want to transform in tokens and of course defined their pattern, using Regex library. Inside this library I have used Matcher and Pattern classes for identifying different token types within the input text.

```
private static final Pattern IDENTIFIER_PATTERN = Pattern.compile( regex: "[a-zA-Z][a-zA-Z0-9]*");
1 usage
private static final Pattern NUMBER_PATTERN = Pattern.compile( regex: "\\d+");
1 usage
private static final Pattern OPERATOR_PATTERN = Pattern.compile( regex: "[+\\-*/]");
```

Figure 1. Pattern defining

The next important step was tokenization logic. In order to do so I have created 'tokenize' method which does the follow: It takes an input string and returns a list of tokens:

- It initializes an empty list tokens to store the tokens.
- It iterates over the input string until it is empty.
  For each iteration:
- It attempts to match an identifier pattern at the beginning of the input string. If a match is found, it creates a token of type IDENTIFIER and adds it to the list of tokens. It then updates the input string by removing the matched part.

```java
while (!input.isEmpty()) {
    // Match identifier
    matcher = IDENTIFIER_PATTERN.matcher(input);
    if (matcher.lookingAt()) {
        tokens.add(new Token(TokenType.IDENTIFIER, matcher.group()));
        input = input.substring(matcher.end());
        continue;
    }
```

Figure 2. Match the pattern

- If no identifier is found, it attempts to match a number pattern in a similar manner.
- If no number is found, it attempts to match an operator pattern.
- If none of the patterns match, it skips the current character.

```java
    // If no match is found, skip the character
    input = input.substring( beginIndex: 1);
}
```

Figure 3. No match

- Finally, it returns the list of tokens.

4. Screen printing of program output:

For checking if the lexer works as it should , I provided the following test case:

```java
String input = "result = (a + b) * (50 - d)";
```

Figure 4. Test case

The output I have received is the following:

```
IDENTIFIER: result
IDENTIFIER: a
OPERATOR: +
IDENTIFIER: b
OPERATOR: *
NUMBER: 50
OPERATOR: -
IDENTIFIER: d
```

Figure 5. Output

The output seems to match the definition and conditions of what a lexer is doing, therefore, the method I have used to implement it is working.

5. Conclusions:

In conclusion, the laboratory work on implementing the lexer provided valuable insights into the fundamental process of lexical analysis in programming languages. By building the lexer, we gained a deeper understanding of how to break down input text into tokens, which serve as the building blocks for more complex language processing tasks.

Throughout the implementation process, we learned the importance of defining token types and corresponding regular expressions to accurately identify them within the input text. Another important feature we have learned is how to use Java's regex library to efficiently match patterns and extract tokens from the input string. Overall, this laboratory work provided great experience in developing a fundamental component of a language.

6. Bibliography:

1. Medium (2023, January 26). *Introduction to Lexical Analysis: What it is and How it Works | by Mitch Huang | Medium*. Medium. https://medium.com/@mitchhuang777/introduction-to-lexical-analysis-what-it-is-and-how-it-works-b25c52113405
2. G. (2022, April 26). *Introduction of Lexical Analysis – GeeksforGeeks* GeeksforGeeks. https://www.geeksforgeeks.org/introduction-of-lexical-analysis/