



**MINISTERUL EDUCAȚIEI, CULTURII ȘI CERCETĂRII
AL REPUBLICII MOLDOVA Universitatea Tehnică a
Moldovei Facultatea Calculatoare, Informatică și
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 $|\text{tutorialspoint}| = 14$. 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 the 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 into 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. regular expression patterns are defined to match three types of tokens: identifiers (variables), numbers, and operators. `IDENTIFIER_PATTERN` matches any sequence of letters or digits starting with a letter. `NUMBER_PATTERN` matches any sequence of digits. `OPERATOR_PATTERN` matches basic arithmetic operators: `+`, `-`, `*`, and `/`.

```
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. This method tokenizes the input string. It iterates over the input string, attempting to match patterns for identifiers, numbers, and operators using regular expressions. If a match is found, a corresponding token is created and added to the list of tokens. The while loop continues until the input string is empty. 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.

```
while (!input.isEmpty()) {
    // Match identifier
    matcher = IDENTIFIER_PATTERN.matcher(input);
    if (matcher.find()) {
        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.

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

Figure 3. No match

- Finally, it returns the list of tokens.

This lexer implementation is a basic example designed to handle simple expressions with identifiers, numbers, and basic arithmetic operators. It doesn't handle more complex scenarios like whitespace, parentheses, or special characters. Also, error handling for invalid input is not included.

Regular expressions (regex) were utilized in this code to match patterns representing different types of tokens within the input string. Three regex patterns were defined:

IDENTIFIER_PATTERN: This regex matches identifiers, which are sequences of letters or digits starting with a letter.

NUMBER_PATTERN: This regex matches numbers, which are sequences of digits.

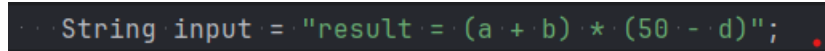
OPERATOR_PATTERN: This regex matches basic arithmetic operators: +, -, *, and /.

These regex patterns are compiled using the Pattern.compile() method from the java.util.regex package. Then, the Matcher class is used to find matches within the input string. When a match is found, the corresponding token type is determined, and a new token is created with the matched value. This process is repeated until all input

characters are processed. Overall, regex enables efficient and flexible pattern matching, facilitating the tokenization process in the lexer.

4. Screen printing of program output:

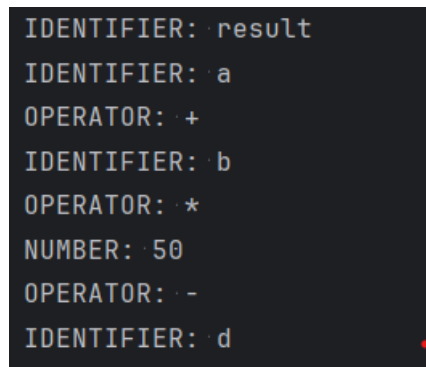
For checking if the lexer works as it should, I provided the following test case: Here, the input string is a simple arithmetic expression: "result = (a + b) * (50 - d)". The purpose of this test is to demonstrate how the lexer tokenizes this input string into individual tokens representing identifiers, operators, and numbers.



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

Figure 4. Test case

After tokenization, the code iterates over the list of tokens and prints each token's type and value:



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

Figure 5. Output

This result shows how the info string has been tokenized into individual tokens, with every symbolic's sort and relating esteem imprinted on a different line. This shows that the lexer effectively recognizes and separates the various components of the information articulation.

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/>