MINISTERUL EDUCAȚIEI, CULTURII ȘI CERCETĂRII

AL REPUBLICII MOLDOVA

Universitatea Tehnică a Moldovei

Facultatea Calculatoare, Informatică şi Microelectronică

Departamentul Inginerie Software și Automatică

Grebennicova Ecaterina FAF-223

Report

Lexer & Scanner

of Formal Languages & Finite Automata

Checked by:

Cretu Dumitru, *university assistant*

Chișinău – 2023

1. Theory

The term lexer comes from lexical analysis which, in turn, represents the process of extracting lexical tokens from a string of characters. There are several alternative names for the mechanism called lexer, for example tokenizer or scanner. The lexical analysis is one of the first stages used in a compiler/interpreter when dealing with programming, markup or other types of languages. The tokens are identified based on some rules of the language and the products that the lexer gives are called lexemes. So basically the lexer is a stream of lexemes. Now in case it is not clear what's the difference between lexemes and tokens, there is a big one. The lexeme is just the byproduct of splitting based on delimiters, for example spaces, but the tokens give names or categories to each lexeme. So the tokens don't retain necessarily the actual value of the lexeme, but rather the type of it and maybe some metadata.

2. Objectives

- Understand what lexical analysis is.
- Get familiar with the inner workings of a lexer/scanner/tokenizer.
- Implement a sample lexer and show how it works.

3. Implementation Description

In this program we have implemented a sample of a lexer that converts an input string into a list of tokens based on defined token types such as integers, operators, punctuation, identifiers, and separators. It demonstrates lexical analysis by reading input from the user, processing it to identify and categorize different elements, and then printing out the resulting tokens.

The code:

```
    enum TokenType {
        INTEGER, OPERATOR, PUNCTUATION, WHITESPACE, SEPARATORS, IDENTIFIER,
EOF
    }
```

This is an enumeration that classifies different types of tokens which will be used later to separate the input parts.

```
    List<Token> tokenize() {
  List<Token> tokens = new ArrayList<>();

  while (currentChar != '\0') {
```

```java
        // Handle different character types to tokenize the input.
        if (Character.isWhitespace(currentChar)) {
            skipWhitespace();
            continue;
        }

        if (Character.isDigit(currentChar)) {
            tokens.add(integer());
            continue;
        }

        if (Character.isLetter(currentChar)) {
            tokens.add(identifier());
            continue;
        }

        if (currentChar == '+' || currentChar == '-' || currentChar == '*' ||
currentChar == '/' || currentChar == '%' || currentChar == '=' || currentChar == '>'
|| currentChar == '<') {
            tokens.add(operator());
            continue;
        }

        if (currentChar == '.' || currentChar == ',' || currentChar == '?' ||
currentChar == '!' || currentChar == ':' || currentChar == ';') {
            tokens.add(punctuation());
            continue;
        }

        if (currentChar == ')' || currentChar == '(' || currentChar == '}' ||
currentChar == '{' || currentChar == ']' || currentChar == '[' || currentChar == '"'
|| currentChar == '\'') {
            tokens.add(separators());
            continue;
        }

        // Handle unexpected characters and throw an exception
        throw new RuntimeException("Unexpected character: " + currentChar);
    }

    // Add an EOF token at the end of the token list
    tokens.add(new Token(TokenType.EOF, ""));
    return tokens;
}
```

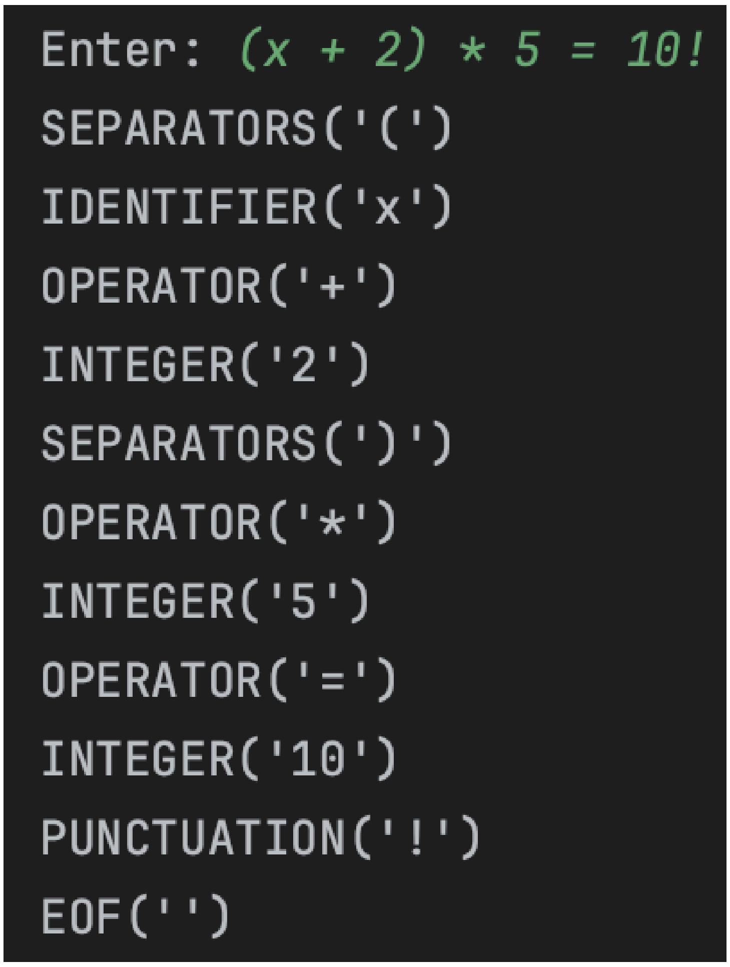
This code snippet shows the main rules that illustrate how the lexical analysis is performed. This method, along with other methods it calls, demonstrates the process of converting the input string into a sequence of tokens based on some lexical rules.

As other methods we have a method for integer, which captures a sequence of digits as an integer token.

```java
Token integer() {
    StringBuilder result = new StringBuilder();
    while (currentChar != '\0' && Character.isDigit(currentChar)) {
        result.append(currentChar);
        advance();
    }
    return new Token(TokenType.INTEGER, result.toString());
}
```

He looks for the current character to check if it is not the last character in the sequence and if it corresponds to a digit, and if true it appends to the result and using advance it moves to the next character. Same thing happens with other tokens such as identifiers, operators, whitespace etc.

Output:



```
Enter: (x + 2) * 5 = 10!
SEPARATORS('(')
IDENTIFIER('x')
OPERATOR('+')
INTEGER('2')
SEPARATORS(')')
OPERATOR('*')
INTEGER('5')
OPERATOR('=')
INTEGER('10')
PUNCTUATION('!')
EOF('')
```

4. Conclusion

In order to understand the topic better I’ve read attentively both the github tasks and the implementation tips provided for better understanding. By looking through explanations provided on the websites, it was easy to understand what the task is about and what has to be implemented. In the end, a Java class was written where a lexer was implemented that can tokenize an input string into distinct elements based on predefined rules. As I got to understand what lexical analysis represents, I managed to classify all characters/tokens in a convenient way and to control the input parsing.

A challenging aspect was managing the logic to distinguish between different types of tokens, such as identifiers and integers, which require careful consideration of the sequence. Implementation was not that hard, but not that easy at the same time. In the end all was managed in time.