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<>();
```

With this method we initiate a list of Token objects where each Token object is assumed to represent a single recognizable element of the input text.

```
    while (currentChar != '\0') {
        if (Character.isWhitespace(currentChar)) {
            skipWhitespace();
            continue;
        }

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

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

The method written above uses a loop to process each character of the input until it reaches a termination condition. Inside the loop, it categorizes each character into one of several types and creates a corresponding Token object based on the character's type.

The loop continues as long as the current character is not the null character '\0', which indicates the end of the input.

If the current character is whitespace, it calls skipWhitespace(), a method designed to advance the current position to the next non-whitespace character. It then continues to the next iteration.

If the current character is a digit, it calls integer(), a method designed to read the full integer starting from this digit, advance the current position past the integer, and return a token representing the integer.

If the current character is a letter, it calls identifier(), a method designed to read an identifier, advance the current position past the identifier, and return a token representing the identifier.

```
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;
        }

        throw new RuntimeException("Unexpected character: " + currentChar);
    }
```

The first if statement here checks if the current character is one of several operators (+, -, *, /, %, =, >, <). If so, it calls operator(), a method to create and return a token representing the operator.

The second if statement here checks for punctuation characters (., ,, ?, !, :, ;). If found, it calls punctuation(), a method to create and return a token for the punctuation.

The third if statement here checks for separator characters (), (, }, {,], [, ", '). If found, it calls separators(), a method to create and return a token representing the separator.

If a character does not fit any of the expected categories, the method throws a RuntimeException indicating an unexpected character.

```
    tokens.add(new Token(TokenType.EOF, ""));
    return tokens;
}
```

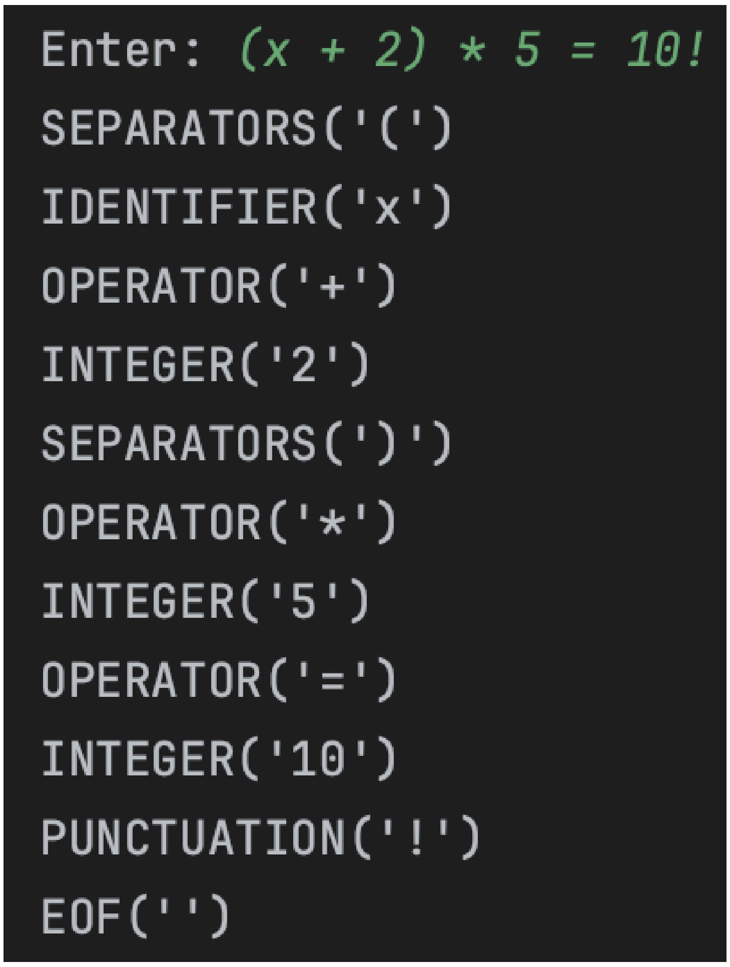
After the loop completes, the method adds a special EOF (end-of-file) token to the list. This token signifies that there are no more tokens to be read, the end of the input was reached. And at the end the method returns the list of tokens that it has created.

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

```
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.