Laboratory work nr.3
Lexer Scanner

Course: Formal Languages & Finite Automata

Student: Munteanu Ecaterina

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:

A lexer is the initial stage of a compiler or interpreter. It takes the source code as input and breaks it down into meaningful units called tokens. Tokens represent the smallest units of syntax in the programming language, such as keywords, identifiers, operators, literals, etc.

A token is a categorized block of text in the source code of a programming language. It is generated by the lexer and serves as input to the parser. Each token represents a lexical unit with a specific meaning in the programming language.

Defining an enumeration **TokenType** was the first step in this implementation. It will be used to represent the different types of tokens recognized by the lexer. Each token type corresponds to a specific lexical unit in the programming language, such as keywords, identifiers, operators, etc.

```
public enum TokenType {
    VAR,    // var keyword
    ID,     // Identifier
    ASSIGN, // =
    SEMI,   // ;
    OP,     // Arithmetic operator (+, -, *, /)
    LPAREN, // (
    RPAREN, // )
    PRINT,  // print keyword
    COMMA,  // ,
    STRING  // String literal
}
```

Implementing a **Token** class was the second step. It represents individual tokens generated by the lexer. Each token consists of a type (TokenType) and an optional value, which is the actual symbol or word that this token represents.

```
public class Token {
    TokenType type;
    String value;
    …
}
```

Implementing a **Lexer** class was a more voluminous step. It will help us tokenize the input source code. The lexer scans the input text character by character and generates tokens based on predefined lexical rules. The main method here is **tokenize()**:

```
public List<Token> tokenize() {
    List<Token> tokens = new ArrayList<>();
    Token token = getNextToken();
    while (token.type != TokenType.SEMI) {
        tokens.add(token);
        token = getNextToken();
    }
    tokens.add(token);
    return tokens;
}
```

It is a pretty straightforward method, analyzing one statement at a time, and stopping when a semicolon is met. The **getNextToken()** method helps us move along the statement and add new tokens to our list.

The method begins by entering a loop that iterates over each character in the input text until it encounters the end of the input (denoted by the null character **\0**). Within the loop, the method checks if the current character is whitespace. If it is, the **skipWhitespace()** method is called to move the lexer's position past any consecutive whitespace characters.

If the current character is a letter, the method assumes it's the start of either a keyword or an identifier. It calls the getIdent() method to extract the entire word, and then checks if the word matches any predefined keywords (e.g., "var", "print"). If it does, a corresponding token with the appropriate type (TokenType.VAR, TokenType.PRINT) is created. Otherwise, it's treated as an identifier and a token of type TokenType.ID is created.

If the current character is a digit, the method assumes it's the start of a number. It extracts the entire sequence of digits using a StringBuilder, creating a token of type TokenType.ID with the numeric value converted to a string.

If the current character is an operator or a special character (e.g., '=', ';', '(', ')', '+', '-', '*', '/'), the method creates a token with the corresponding type (TokenType.ASSIGN, TokenType.SEMI, TokenType.LPAREN, TokenType.RPAREN, TokenType.OP, TokenType.COMMA). For string literals enclosed in double quotes, the method extracts the characters until the closing quote and creates a token of type TokenType.STRING.

If none of the above conditions are met and the end of the input is reached, a token of type TokenType.SEMI is created to indicate the end of the statement.

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

```
        if (Character.isLetter(currentChar)) {
            String ident = getIdent();
            switch (ident) {
                case "var":
                    return new Token(TokenType.VAR, ident);
                …
        }
        …
        switch (currentChar) {
            case '=':
                advance();
                return new Token(TokenType.ASSIGN, "=");
            default:
                throw new IllegalArgumentException("Invalid
character: " + currentChar);
        }
    }
    return new Token(TokenType.SEMI, null);
```

Once the appropriate token is created, it is returned by the method, and the lexer's position is advanced to the next character in the input text. However, if the input is unknown to the method, it will throw an exception.

The **Main** class imitates a console, thus we can introduce as many statements as we want, but they will be analyzed separately, by pressing Enter to let the Lexer do its work.

The result in the console looks as follows:

```
Enter your statements (press Enter after each statement), and type 'exit' to end:
console> var x = 101;
Tokens: [VAR:'var', ID:'x', ASSIGN:'=', ID:'101', SEMI:';']

console> print(y);
Tokens: [PRINT:'print', LPAREN:'(', ID:'y', RPAREN:')', SEMI:';']

console> var z = "haha";
Tokens: [VAR:'var', ID:'z', ASSIGN:'=', STRING:'haha', SEMI:';']

console> exit

Process finished with exit code 0
```

Conclusions:

In conclusion, the laboratory work focused on implementing a lexer for a simple programming language. The lexer serves as the first step in the compilation or interpretation process, breaking down the input source code into tokens for further processing. By understanding the theoretical concepts of lexers and applying them in the implementation, I successfully wrote a lexer capable of tokenizing input text according to predefined lexical rules. This practical exercise helped me understand lexers better and will help me work better on my semester project, creating a more complex lexer for our DSL.