Laboratory work nr.6
Parser & Building an Abstract Syntax Tree

Course: Formal Languages & Finite Automata

Student: Munteanu Ecaterina

Objectives:

1. Get familiar with parsing, what it is and how it can be programmed [1].
2. Get familiar with the concept of AST [2].
3. In addition to what has been done in the 3rd lab work do the following:
   I. In case you didn't have a type that denotes the possible types of tokens you need to:
      a. Have a type TokenType (like an enum) that can be used in the lexical analysis to categorize the tokens.
      b. Please use regular expressions to identify the type of the token.
   II. Implement the necessary data structures for an AST that could be used for the text you have processed in the 3rd lab work.
   III. Implement a simple parser program that could extract the syntactic information from the input text.

Implementation:

To do this laboratory work I first created an **ASTNode** class, which is the base of the parser.

```
abstract class ASTNode {
    abstract void printTree(int level);
    void printIndent(int level) {
        for (int i = 0; i < level; i++) {
            System.out.print("  ");
        }
    }
}
```

I also added **NumericLiteral, BinaryOperation, VariableDeclaration, VariableReference** and **PrintStatement** classes, which all implement the base abstract class, and respectively the **printTree** and **printIndent** methods. For example in NumericLiteral it looks like:

```
class NumericLiteral extends ASTNode {
    Token value;

    public NumericLiteral(Token value) {
        this.value = value;
    }

    @Override
```

```
    void printTree(int level) {
        printIndent(level);
        System.out.println("NumericLiteral: " +
value.getValue());
    }
}
```

The most important class is of course the **Parser**, which takes each term and parses it accordingly.

```
public List<ASTNode> parse() {
        List<ASTNode> statements = new ArrayList<>();
        while (currentToken().getType() != TokenType.EOF) {
            statements.add(parseStatement());
        }
        return statements;
    }
```

The **main** methods looks like this:

```
String input = "var x = 4; var y = 7; var sum = x + y;
print(sum);";
        Lexer lexer = new Lexer(input);
        List<Token> tokens = lexer.tokenize();
        Parser parser = new Parser(tokens);
        List<ASTNode> astNodes = parser.parse();

        for (ASTNode node : astNodes) {
            node.printTree(0);
        }
```

It should be able to parse simple variable declarations, addition and print function.

Conclusions:

The lab shows how to make a basic lexer, parser, and AST in Java. We learned about the main parts of a compiler and how to organize a tool for understanding languages. It's important to handle different kinds of tokens and create a parser that can handle more grammar as needed.