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

Parser & Building and Abstract Syntax Tree

of Formal Languages & Finite Automata

Checked by:

Cretu Dumitru, *university assistant*

Chișinău – 2023

1. **Theory**

A **parser** is a program or component of a program that takes input data (often in the form of text) and breaks it down into smaller components for further processing. It's commonly used in computer science and linguistics to analyze the structure of sentences or code. In natural language processing, parsers help identify the grammatical structure of sentences, such as subjects, verbs, and objects. In programming, parsers are used to analyze code syntax, ensuring it follows the rules of the programming language. Parsers play a crucial role in tasks like language translation, code compilation, and data processing by making sense of complex input data.

An **abstract syntax tree (AST)** is a hierarchical representation of the structure of code in a programming language, capturing its essential elements while abstracting away details like formatting and comments. It's commonly used in compilers and interpreters to analyze and manipulate source code. Each node in the tree represents a syntactic construct of the code, such as expressions, statements, or declarations, and the relationships between them. ASTs provide a foundation for various language processing tasks, including optimization, code generation, and static analysis. By representing code in a structured format, ASTs facilitate automated reasoning and manipulation of program structure.

2. **Objectives**

- Get familiar with parsing, what it is and how it can be programmed.
- Get familiar with the concept of AST.
- Implement the necessary data structures for an AST that could be used for the text you have processed in the 3rd lab work.
- Implement a simple parser program that could extract the syntactic information from the input text.

3. **Implementation Description**

The methods described below represents - **Abstract Syntax Tree methods**

The BinaryOperatorNode class represents a binary operation within an Abstract Syntax Tree (AST). It contains three instance variables: left, right and operator, which represent the left and right operands of the operation, and operator token itself. The constructor initializes these variables with the provided left operand, operator token,

and right operand.

```
class BinaryOperatorNode extends ASTNode {
   ASTNode left;
   ASTNode right;
   Token operator;

   BinaryOperatorNode(ASTNode left, Token operator, ASTNode right) {
       this.left = left;
       this.operator = operator;
       this.right = right;
   }
}
```

The same thing we repeat with UnaryOperatorNode class (representing the negation of a number or a variable using "-" or "!"), NumberNode class (representing the numbers), VariableNode class (representing the variable) and AssignmentNode class (representing the assignment of variable to an expression).

```
class AssignmentNode extends ASTNode {
   VariableNode variable;
   ASTNode expression;

   AssignmentNode(VariableNode variable, ASTNode expression) {
       this.variable = variable;
       this.expression = expression;
   }
}
```

The methods described below represents - **Parser**

The currentToken() method retrieves the token at the current position within a list of tokens. It first checks if the current position is within the bounds of the list. If it is, it returns the token at that position. If the current position exceeds the size of the list, it returns a special End-of-File (EOF) token to indicate the end of the token stream.

```
    Token currentToken() {
        if (currentPosition < tokens.size()) {
            return tokens.get(currentPosition);
        }
        return new Token(TokenType.EOF, "");
    }
```

The check method verifies if the type of the current token matches the specified type. If they match, it increments the current position to move to the next token in the sequence. If the types do not match, it throws a RuntimeException, indicating that the

current token is unexpected and displaying information about the token and the expected type.

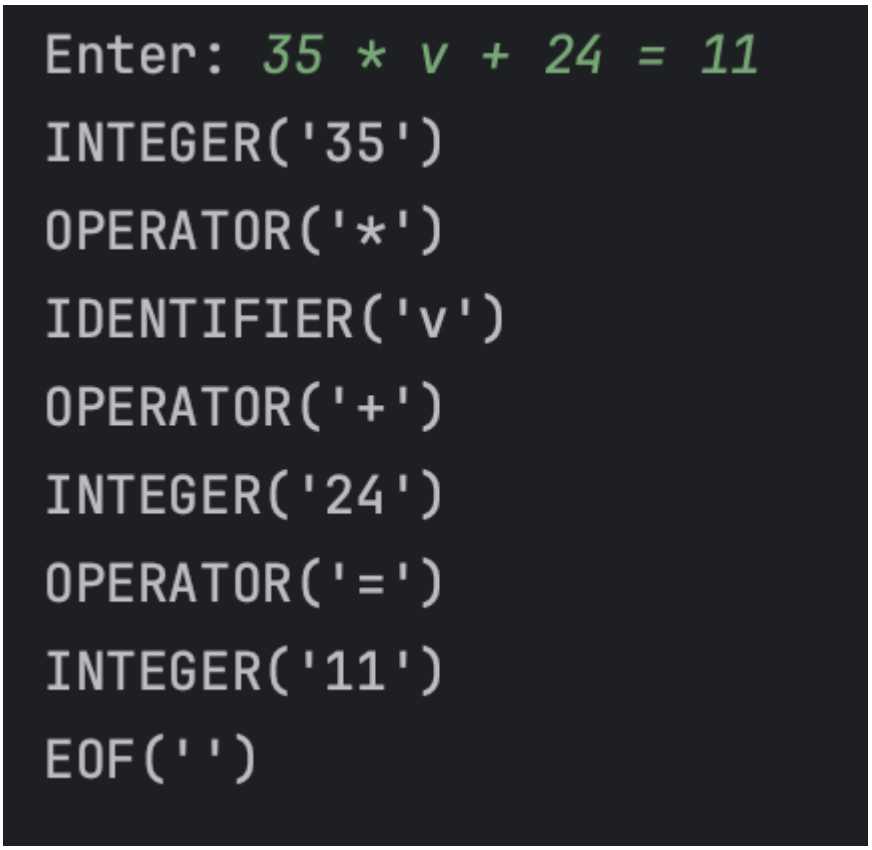
```
    void check(TokenType type) {
        if (currentToken().type == type) {
            currentPosition++;
        } else {
            throw new RuntimeException("Unexpected token: " + currentToken() + ",
expected: " + type);
        }
    }
```

The expression() method parses an expression by starting with a term and then iteratively looking for add/subtract operators. It initializes a result node with the term, then enters a loop where it checks if the current token is an operator (specifically, either '+' or '-'). If the current token is an operator, it retrieves the operator token, checks it, and then parses the next term. Finally, it constructs a BinaryOperatorNode with the previous result node, the operator token, and the newly parsed term, updating the result node. This process continues until no more add/subtract operators are found, and then it returns the resulting AST node representing the parsed expression.

```
    ASTNode expression() {
        ASTNode result = term();
        while (currentToken().type == TokenType.OPERATOR &&
(currentToken().value.equals("+") || currentToken().value.equals("-"))) {
            Token op = currentToken();
            check(TokenType.OPERATOR);
            result = new BinaryOperatorNode(result, op, term());
        }
        return result;
    }
```

We use other similar methods to parse terms, factors and assignment expressions as well.

Output:



4. Conclusion

In conclusion, I could say that the provided code comprises a lexer, parser, and related classes aimed at tokenizing and parsing input strings. The lexer processes input strings, breaking them down into tokens of various types, such as integers, operators, identifiers, and punctuation. It utilizes character-by-character scanning to identify and categorize tokens based on predefined rules. The parser, on the other hand, constructs an Abstract Syntax Tree (AST) from the tokens generated by the lexer, adhering to the syntax rules of a simple programming language. It employs recursive descent parsing to recognize and parse expressions and assignment statements. The AST nodes represent different elements of the language syntax, including binary and unary operations, numbers, variables, and assignment statements. The parser employs a combination of methods to parse expressions, terms, factors, and assignment expressions, ensuring proper adherence to the language grammar. In the end, this code demonstrates the foundational concepts of lexical analysis and syntactic parsing, crucial for building compilers, interpreters, and other language processing tools.