MINISTERUL EDUCAȚIEI, CULTURII ȘI CERCETĂRII AL REPUBLICII MOLDOVA

Universitatea Tehnică a Moldovei

Facultatea Calculatoare, Informatică şi Microelectronică Departamentul Inginerie Software și Automatică

CEBAN VASILE FAF-223

Report

Laboratory work n.6

of Formal Languages and Finite Automata

Checked by:

Dumitru Crețu, *university assistant*

DISA, FCIM, UTM

Chișinău - 2024

Topic: Parser & Building an Abstract Syntax Tree

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:
    a. In case you didn't have a type that denotes the possible types of tokens you need to:
        i. Have a type *TokenType* (like an enum) that can be used in the lexical analysis to categorize the tokens.
        ii. Please use regular expressions to identify the type of the token.
    b. Implement the necessary data structures for an AST that could be used for the text you have processed in the 3rd lab work.
    c. Implement a simple parser program that could extract the syntactic information from the input text.

An **abstract syntax tree** (**AST**) is a data structure used in computer science to represent the structure of a program or code snippet. It is a tree representation of the abstract syntactic structure of text (often source code) written in a formal language. Each node of the tree denotes a construct occurring in the text. It is sometimes called just a **syntax tree**.

The syntax is "abstract" in the sense that it does not represent every detail appearing in the real syntax, but rather just the structural or content-related details. For instance, grouping parentheses are implicit in the tree structure, so these do not have to be represented as separate nodes. Likewise, a syntactic construct like an if-condition-then statement may be denoted by means of a single node with three branches.

This distinguishes abstract syntax trees from concrete syntax trees, traditionally designated parse trees. Parse trees are typically built by a parser during the source code translation and compiling process. Once built, additional information is added to the AST by means of subsequent processing, e.g., contextual analysis.

Abstract syntax trees are also used in program analysis and program transformation systems.

Source Code:

class BinaryOperatorExpr:

```
@Override
double evaluate() {
   return switch (operator) {
       case '+' -> left.evaluate() + right.evaluate();
       case '-' -> left.evaluate() - right.evaluate();
       case '*' -> left.evaluate() * right.evaluate();
       case '/' -> left.evaluate() / right.evaluate();
       default -> throw new RuntimeException("Unsupported operator: " +
operator);
   };
}
```

Description:

The method evaluates a mathematical expression represented by a binary tree structure. Each node of the tree represents either a numeric value or an operator.

class FunctionExpression:

```
@Override
double evaluate() {
   return switch (function) {
       case "sqrt" -> {
           if (arguments.size() != 1) {
               throw new RuntimeException("sqrt expects exactly one
argument");
           }
           yield Math.sqrt(arguments.get(0).evaluate());
       }
       case "log" -> {
           if (arguments.size() != 1) {
               throw new RuntimeException("log2 expects exactly one
argument");
           }
           yield Math.log(arguments.get(0).evaluate()) / Math.log(2);
       }
       case "lg" -> {
           if (arguments.size() != 1) {
               throw new RuntimeException("log10 expects exactly one
argument");
           }
           yield Math.log10(arguments.get(0).evaluate());
       }
       case "loge" -> {
           if (arguments.size() != 1) {
               throw new RuntimeException("loge expects exactly one
argument");
           }
           yield Math.log(arguments.get(0).evaluate());
       }
       case "pow" -> {
           if (arguments.size() != 1) {
```

```
                throw new RuntimeException("pow2 expects exactly one
argument");
            }
            yield Math.pow(2, arguments.get(0).evaluate());
        }
        default -> throw new RuntimeException("Unsupported function or wrong
number of arguments: " + function);
    };
}
```

Description:

This code snippet provides a mechanism to evaluate various mathematical functions (such as square root, logarithms with different bases, and power functions) with the corresponding number of arguments. If the function or number of arguments is incorrect, it throws a RuntimeException with an appropriate error message.

class Token:

```
public static List<Token> tokenize(String input) {
    List<Token> tokens = new ArrayList<>();
    input = input.replaceAll("\\s", "");
    int i = 0;
    while (i < input.length()) {
        char ch = input.charAt(i);
        if (Character.isDigit(ch)) {
            StringBuilder sb = new StringBuilder();
             while (i < input.length() && (Character.isDigit(input.charAt(i))
|| input.charAt(i) == '.')) {
                sb.append(input.charAt(i));
                i++;
            }
            tokens.add(new Token(TokenType.NUMBER, sb.toString()));
        } else if (Character.isLetter(ch)) {
            StringBuilder sb = new StringBuilder();
             while (i < input.length() && Character.isLetter(input.charAt(i)))
{
                sb.append(input.charAt(i));
                i++;
            }
            tokens.add(new Token(TokenType.FUNCTION, sb.toString()));
        } else {
            switch (ch) {
                case '(':
                    tokens.add(new Token(TokenType.LPAREN, "("));
                    i++;
                    break;
                case ')':
                    tokens.add(new Token(TokenType.RPAREN, ")"));
                    i++;
                    break;
                case ',':
                    tokens.add(new Token(TokenType.COMMA, ","));
```

```
                    i++;
                    break;
                default:
                                        tokens.add(new  Token(TokenType.OPERATOR,
String.valueOf(ch)));
                    i++;
                    break;
            }
        }
    }
    tokens.add(new Token(TokenType.EOF, ""));
    return tokens;
}
```

Description:

This code defines a method tokenize(String input) that takes an input string and converts it into a list of tokens. Tokens represent the smallest units of meaning in the input string, such as numbers, functions, parentheses, commas, and operators.

class Parser:

```
    private Expr expression() {
    Expr expr = term();
    while (match(TokenType.OPERATOR)) {
        char operator = tokens.get(current - 1).value.charAt(0);
        Expr right = term();
        expr = new BinaryOperatorExpr(expr, right, operator);
    }
    return expr;
}

private Expr term() {
    if (match(TokenType.NUMBER)) {
        return new NumberExpr(Double.parseDouble(previous().value));
    } else if (match(TokenType.LPAREN)) {
        Expr expr = expression();
        consume(TokenType.RPAREN, "Expect ')' after expression.");
        return expr;
    } else if (match(TokenType.FUNCTION)) {
        String functionName = previous().value;
        consume(TokenType.LPAREN, "Expect '(' after function name.");
        List<Expr> args = new ArrayList<>();
        if (!check(TokenType.RPAREN)) {
            do {
                args.add(expression());
            } while (match(TokenType.COMMA));
        }
        consume(TokenType.RPAREN, "Expect ')' after arguments.");
        return new FunctionExpr(functionName, args);
    }
    throw new RuntimeException("Unexpected token.");
}
```

```
private boolean match(TokenType type) {
    if (check(type)) {
        current++;
        return true;
    }
    return false;
}

private boolean check(TokenType type) {
    if (isAtEnd()) return false;
    return tokens.get(current).type == type;
}

private Token previous() {
    return tokens.get(current - 1);
}

private void consume(TokenType type, String message) {
    if (check(type)) {
        current++;
        return;
    }
    throw new RuntimeException(message);
}

private boolean isAtEnd() {
        return current >= tokens.size() || tokens.get(current).type ==
TokenType.EOF;
}
```

Description:

This method parses the input string character by character, identifies different types of tokens (numbers, functions, parentheses, commas, and operators), and constructs token objects accordingly. These tokens are then stored in a list and returned as the result of the method.

Testing:

Input:

```
String input = "sqrt(25) + (10 - 3) * pow(2)";
```

Output:

```
Tokens:
FUNCTION : sqrt
LPAREN : (
NUMBER : 25
RPAREN : )
OPERATOR : +
LPAREN : (
NUMBER : 10
OPERATOR : -
NUMBER : 3
RPAREN : )
OPERATOR : *
FUNCTION : pow
LPAREN : (
NUMBER : 2
RPAREN : )
EOF :
Result: 48.0
```

Figure 1. Rezultat

Conclusion

In this laboratory work, we delved into the fundamental concepts of parsing and abstract syntax trees (ASTs) in the context of programming language processing. Parsing involves analyzing the structure of source code to understand its syntactic elements, while ASTs provide a structured representation of the code's abstract syntax.

We started by understanding the process of lexical analysis using regular expressions to tokenize input text into meaningful units called tokens. These tokens serve as the building blocks for parsing.

Next, we explored the concept of parsing, which involves interpreting the syntactic structure of the tokenized code. We implemented a simple parser capable of extracting syntactic information from the input text. The parser used a recursive descent approach to recognize and process different language constructs, such as if statements and assignments.

To represent the structure of the parsed code, we introduced the concept of abstract syntax trees (ASTs). ASTs provide a hierarchical representation of code constructs, organized into nodes that correspond to language elements. We defined AST node classes such as ProgramNode, IfStatementNode, and AssignmentNode, which capture the structure of the parsed code.

By implementing parsing and AST generation, we gained insights into the inner workings of language processing tools and compilers. These concepts are foundational for understanding how programming languages are interpreted, compiled, and executed.

In conclusion, this laboratory work provided valuable hands-on experience with parsing techniques and AST representation, laying the groundwork for further exploration into language processing and compiler construction.