MINISTERUL EDUCAȚIEI, CULTURII ȘI CERCETĂRIIAL

REPUBLICII MOLDOVA

Universitatea Tehnică a Moldovei

Facultatea Calculatoare, Informatică şi Microelectronică

Departamentul Inginerie Software și Automatică

Report

Laboratory work n.6

Formal Languages &Finite Automata

Author: *Țapu Pavel*

Chișinău – 2023

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 Description

In this laboratory work, we explored the concepts of parsing and Abstract Syntax Trees (ASTs) by implementing a lexical analyzer (lexer) and a parser in Java. The primary objective was to tokenize an arithmetic expression, parse the tokens into an AST, and demonstrate the process through a simple arithmetic expression: "6 + 2 * (5 - 3)".

We started by defining the possible types of tokens using an enumeration TokenType.java. This enum included basic arithmetic operators (PLUS, MINUS, TIMES, DIVIDE), parentheses (LPAREN, RPAREN), integers (INTEGER), and an end-of-file marker (EOF). The Token.java class represented individual tokens, containing fields for the token type and its value.

```
public enum TokenType {  18 usages  new *
    INTEGER, PLUS, MINUS, TIMES, DIVIDE, LPAREN, RPAREN, EOF
}
```

The core of our lexical analysis was implemented in Lexer.java. Using regular expressions, the lexer scanned the input string and identified the various tokens based on the patterns defined for each token type. The lexer iterated through the input string, matched patterns, and generated a list of tokens, appending an EOF token at the end to signify the end of the input.

```
while (matcher.find()) {
    if (matcher.group( name: "INTEGER") != null) {
        tokens.add(new Token(TokenType.INTEGER, Integer.parseInt(matcher.group( name: "INTEGER"))));
    } else if (matcher.group( name: "PLUS") != null) {
        tokens.add(new Token(TokenType.PLUS, matcher.group( name: "PLUS")));
    } else if (matcher.group( name: "MINUS") != null) {
        tokens.add(new Token(TokenType.MINUS, matcher.group( name: "MINUS")));
    } else if (matcher.group( name: "TIMES") != null) {
        tokens.add(new Token(TokenType.TIMES, matcher.group( name: "TIMES")));
    } else if (matcher.group( name: "DIVIDE") != null) {
        tokens.add(new Token(TokenType.DIVIDE, matcher.group( name: "DIVIDE")));
    } else if (matcher.group( name: "LPAREN") != null) {
        tokens.add(new Token(TokenType.LPAREN, matcher.group( name: "LPAREN")));
    } else if (matcher.group( name: "RPAREN") != null) {
        tokens.add(new Token(TokenType.RPAREN, matcher.group( name: "RPAREN")));
    }
}
```

The next step involved constructing the AST, for which we created abstract classes in AST.java. The AST class served as the base, with BinOp and Num subclasses representing binary operations and numeric values, respectively. BinOp nodes contained references to their left and right operands and the operation token, while Num nodes stored numeric values.

```
BinOp(AST left, Token op, AST right) {
    this.left = left;
    this.op = op;
    this.right = right;
}
```

We then implemented the parser in Parser.java, responsible for converting the list of tokens into an AST. The parser followed a recursive descent parsing strategy, beginning with the expression() method that handled addition and subtraction operations. The term() method dealt with multiplication and division, while the factor() method handled numeric values and expressions within parentheses. The parser used a currentToken to keep track of the next token to process, advancing through the token list as it built the AST nodes.

Finally, in Main.java, we brought together the lexer and parser to demonstrate the entire process. The main method created a lexer instance, tokenized the input expression, and then created a parser instance to parse the tokens into an AST. We included methods to print the tokens and the AST structure, providing a clear visualization of the tokenization and parsing stages.

```java
public static void main(String[] args) {  new *
    String text = "6 + 2 * (5 - 3)";

    Lexer lexer = new Lexer(text);
    lexer.tokenize();

    Parser parser = new Parser(lexer);
    AST ast = parser.parse();

    System.out.println("Tokens:");
    for (Token token : lexer.getTokens()) {
        System.out.println(token);
    }

    System.out.println("\nAST:");
    printAST(ast, level: 0);
}
```

Overall, this laboratory work provided hands-on experience with lexical analysis, parsing, and AST construction, illustrating the essential steps involved in processing arithmetic expressions in a programming language. Through this implementation, we gained a deeper understanding of how compilers and interpreters handle source code, breaking it down into meaningful components and organizing them into structured representations for further processing.

Results

The output provided illustrates the detailed process of tokenization and parsing applied to the arithmetic expression 6 + 2 * (5 - 3). It is divided into two main sections: the list of tokens generated by the lexer and the abstract syntax tree (AST) produced by the parser.

In the tokenization phase, the lexer processes the input string and identifies each component as a specific token. The identified tokens are as follows: Token(INTEGER, 6), Token(PLUS, +), Token(INTEGER, 2), Token(TIMES, *), Token(LPAREN, (), Token(INTEGER, 5), Token(MINUS, -), Token(INTEGER, 3), Token(RPAREN,)), and Token(EOF, null). Each token accurately represents elements of the arithmetic expression, categorizing integers, operators, and parentheses appropriately. This precise tokenization allows the parser to correctly interpret the syntactic structure of the expression.

```
Tokens:
Token(INTEGER, 6)
Token(PLUS, +)
Token(INTEGER, 2)
Token(TIMES, *)
Token(LPAREN, ()
Token(INTEGER, 5)
Token(MINUS, -)
Token(INTEGER, 3)
Token(RPAREN, ))
Token(EOF, null)

AST:
BinOp:
  Left:
    Num: 6
  Op: +
  Right:
    BinOp:
      Left:
        Num: 2
      Op: *
      Right:
        BinOp:
          Left:
            Num: 5
          Op: -
          Right:
            Num: 3
```

The parsing phase transforms the tokens into an abstract syntax tree (AST), representing the hierarchical structure of the arithmetic expression. The root of the AST is a BinOp node representing the addition operation. Its left child is a Num node with the value 6, and its right child is another BinOp node representing the multiplication operation. The multiplication node's left child is a Num node with the value 2, and its right child is a BinOp node representing the subtraction operation within parentheses. This subtraction node has two children: a Num node with the value 5 on the left and a Num node with the value 3 on the right.

The AST clearly reflects the correct order of operations, adhering to arithmetic rules and the precedence of operators. The expression within the parentheses (5 - 3) is parsed first, forming the rightmost subtree. The result of this operation is then used in the multiplication 2 * (5 - 3), forming the next level of the tree. Finally, the result of 2 * 2 is used in the addition 6 + (2 * 2), forming the root level of the tree. This hierarchical representation ensures that the expression is evaluated in the correct order, demonstrating the effectiveness of the parser in interpreting and prioritizing operations accurately.

Conclusion

In this laboratory work, we delved into the foundational concepts of lexical analysis and parsing, gaining practical experience by implementing a lexer and a parser in Java. Through this exercise, we successfully tokenized an arithmetic expression and constructed an abstract syntax tree (AST) to represent its hierarchical structure. This process illuminated the critical steps involved in breaking down a source code into tokens and organizing them into a structured representation for further processing. The lexer efficiently identified and categorized tokens using regular expressions, while the parser accurately interpreted the syntactic structure, adhering to arithmetic rules and operator precedence. By visualizing both the tokens and the AST, we demonstrated a clear understanding of how compilers and interpreters process and evaluate expressions. This work not only reinforced theoretical knowledge of parsing and ASTs but also provided valuable hands-on experience, enhancing our ability to implement and understand these essential components of programming language processing.