MINISTERUL EDUCAȚIEI, CULTURII ȘI CERCETĂRII AL REPUBLICII MOLDOVA Universitatea Tehnică a Moldovei Facultatea Calculatoare, Informatică şi Microelectronică Departamentul Inginerie Software și Automatică

GLIGA DANIELA, FAF-223

Report

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

of Formal Languages and Finite Automata

Checked by:

Dumitru Crețu, *university assistant*

FCIM, UTM

Chișinău – 2024

1. Theory:

Parsing, syntax analysis, or syntactic analysis is the process of analyzing a string of symbols, either in natural language, computer languages or data structures, conforming to the rules of a formal grammar. The term parsing comes from Latin pars (orationis), meaning part (of speech).[1]
The term has slightly different meanings in different branches of linguistics and computer science. Traditional sentence parsing is often performed as a method of understanding the exact meaning of a sentence or word, sometimes with the aid of devices such as sentence diagrams. It usually emphasizes the importance of grammatical divisions such as subject and predicate.
Within computational linguistics the term is used to refer to the formal analysis by a computer of a sentence or other string of words into its constituents, resulting in a parse tree showing their syntactic relation to each other, which may also contain semantic information.[citation needed] Some parsing algorithms generate a parse forest or list of parse trees from a string that is syntactically ambiguous.[2]
The term is also used in psycholinguistics when describing language comprehension. In this context, parsing refers to the way that human beings analyze a sentence or phrase (in spoken language or text) "in terms of grammatical constituents, identifying the parts of speech, syntactic relations, etc."[1] This term is especially common when discussing which linguistic cues help speakers interpret garden-path sentences.
Within computer science, the term is used in the analysis of computer languages, referring to the syntactic analysis of the input code into its component parts in order to facilitate the writing of compilers and interpreters. The term may also be used to describe a split or separation.

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.

2. Objectives:

Get familiar with parsing, what it is and how it can be programmed [1].
Get familiar with the concept of AST [2].
In addition to what has been done in the 3rd lab work do the following:
In case you didn't have a type that denotes the possible types of tokens you need to:
Have a type TokenType (like an enum) that can be used in the lexical analysis to categorize the tokens.
Please use regular expressions to identify the type of the token.
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.

Implementation description:

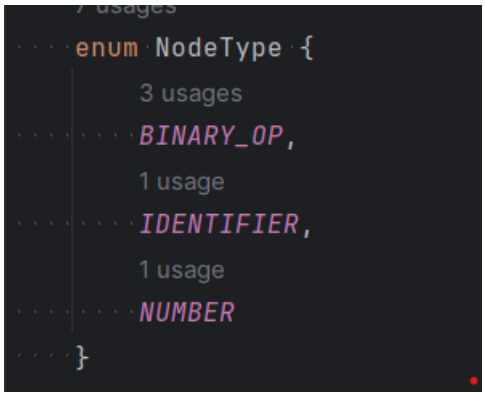
The implementation of I created defines a simple lexer and parser for a basic mathematical expression language. Let's break down the functionality step by step:

Besides the patterns I have defined in the 3rd laboratory work, I have implemented the token type method which enumerate the token types:

```
enum TokenType {
    2 usages
    IDENTIFIER,
    3 usages
    NUMBER,
    3 usages
    OPERATOR,
    2 usages
    LPAREN,  // Left Parenthesis "("
    2 usages
    RPAREN   // Right Parenthesis ")"
}
```

IDENTIFIER_PATTERN: This pattern matches identifiers, which start with a letter (uppercase or lowercase) followed by zero or more letters or digits.
NUMBER_PATTERN: This pattern matches numeric literals consisting of one or more digits.
OPERATOR_PATTERN: This pattern matches basic arithmetic operators such as +, -, *, and /.
PAREN_PATTERN: This pattern matches parentheses, specifically the characters (and).

The same was done to implement the AST functionalitites. Firstly the node types were identified:



This class serves as the building block for constructing the AST during the parsing process. Each node in the AST represents a component of the parsed expression, whether it's an operator, operand, or any other structural element. By organizing nodes in a tree structure, the AST represents the syntactic structure of the expression, making it easier to evaluate or manipulate programmatically.
This Parser class is designed to handle the parsing of tokens generated by the lexer and construct an Abstract Syntax Tree (AST) representing a mathematical expression. The Parser class is used to convert a sequence of tokens (generated by the lexer) representing a mathematical expression into a structured AST. It uses recursive descent parsing techniques, breaking down complex expressions into simpler components based on grammar rules. The resulting AST can then be used for various purposes such as evaluation, optimization, or code generation in a compiler or interpreter for mathematical expressions.

The method expression: starts by initializing a Node variable left by calling the term() method. This step parses the leftmost term of the expression. The method enters a while loop that continues as long as there are more tokens to process (currentTokenIndex < tokens.size()). It retrieves the next token from the token list using tokens.get(currentTokenIndex) and stores it in the token variable. It checks if the current token is an operator (TokenType.OPERATOR) and if the operator is either addition (+) or subtraction (-). If the condition is met (token.value.equals("+") || token.value.equals("-")), it indicates that the expression contains an addition or subtraction operation. If the current token is an addition or subtraction operator, the parser advances to the next token (currentTokenIndex++) and parses the next term on the right side of the operator by calling term() again. This gives us the right operand of the binary operation. After parsing the right operand, it creates a new Node (opNode) representing the binary operation. The opNode is initialized with the type NodeType.BINARY_OP and the value of the operator (token.value). The left node becomes the left child of the operator node (opNode.left = left), representing the expression's left subtree. The right operand node becomes the right child of the operator node (opNode.right = right), representing the expression's right subtree. Finally, the left node is updated to be the opNode, effectively moving the parsing pointer to the next level up in the AST hierarchy. This process continues until there are no more operators to process or until the end of the expression is reached. Once the loop completes, the method returns the left node, which now represents the entire parsed expression subtree for the current context.

```
private Node expression() {
    Node left = term();
    while (currentTokenIndex < tokens.size()) {
        Token token = tokens.get(currentTokenIndex);
        if (token.type == TokenType.OPERATOR && (token.value.equals("+") || token.value.equals("-"))) {
            currentTokenIndex++;
            Node right = term();
            Node opNode = new Node(NodeType.BINARY_OP, token.value);
            opNode.left = left;
            opNode.right = right;
            left = opNode;
        } else {
            break;
        }
    }
    return left;
}
```

The term() method parses a term in an expression, starting with the leftmost factor obtained by calling the factor() method initially. It enters a while loop to continue parsing as long as there are more tokens to process (currentTokenIndex < tokens.size()). Within the loop, it checks if the current token is an operator (* or /) using token.type == TokenType.OPERATOR. If the current token is an operator, it advances to the next token (currentTokenIndex++) and parses the next factor on the right side of the operator by calling factor() again. It constructs a new Node representing the binary operation (* or /) and updates the left node to be this operator node, creating the AST structure. The method then returns the resulting left node, which represents the parsed term subtree with all multiplication and division operations appropriately organized in the AST. The factor() method, on the other hand, parses individual factors such as numbers, identifiers, or expressions enclosed in parentheses, recursively calling expression() when encountering parentheses for nested expressions.

```
private Node term() {
    Node left = factor();
    while (currentTokenIndex < tokens.size()) {
        Token token = tokens.get(currentTokenIndex);
        if (token.type == TokenType.OPERATOR && (token.value.equals("*") || token.value.equals("/"))) {
            currentTokenIndex++;
            Node right = factor();
            Node opNode = new Node(NodeType.BINARY_OP, token.value);
            opNode.left = left;
            opNode.right = right;
            left = opNode;
        } else {
            break;
        }
    }
    return left;
}
```

This term() method is responsible for parsing a term within a mathematical expression, such as a multiplication or division operation. Here's a breakdown of its functionality:

It starts by parsing the leftmost factor of the term by calling the factor() method and storing the result in the left node. Using a while loop, it continues parsing as long as there are more tokens in the input (currentTokenIndex < tokens.size()). Within the loop, it checks if the current token is an operator (* or /) using the token's type and value (token.type == TokenType.OPERATOR). If the token is indeed a multiplication or division operator, it advances to the next token (currentTokenIndex++) and parses the next factor on the right side of the operator by calling factor() again. It constructs a new Node representing the binary operation (* or /) with the left and right operands and updates the left node to represent this operation, continuing the loop until no more operators are found, and finally returns the resulting AST subtree representing the parsed term.

```
private Node factor() {
    Token token = tokens.get(currentTokenIndex++);
    if (token.type == TokenType.NUMBER || token.type == TokenType.IDENTIFIER) {
        return new Node(token.type == TokenType.NUMBER ? NodeType.NUMBER : NodeType.IDENTIFIER, token.value);
    } else if (token.type == TokenType.LPAREN) {
        Node exprNode = expression();
        // Expecting a closing parenthesis
        if (currentTokenIndex < tokens.size() && tokens.get(currentTokenIndex).type == TokenType.RPAREN) {
            currentTokenIndex++; // Consume the ')'
            return exprNode;
        } else {
            throw new IllegalArgumentException("Mismatched parentheses");
        }
    } else {
        throw new IllegalArgumentException("Unexpected token: " + token.value);
    }
}
```

The printAST function recursively prints the contents of an Abstract Syntax Tree (AST) starting from the provided node. It distinguishes between binary operators and operands (identifiers or numbers), printing their respective values. This function is useful for visualizing the structure of the parsed AST, aiding in understanding how expressions are represented hierarchically.

```
    private static void printAST(Node node) {
        if (node == null) return;
        if (node.type == NodeType.BINARY_OP) {
            System.out.println("Binary Operator: " + node.value);
        } else {
            System.out.println("Operand: " + node.value);
        }
        printAST(node.left);
        printAST(node.right);
    }
}
```

This program implements a lexer and parser for basic mathematical expressions. The lexer tokenizes input expressions into identifiers, numbers, operators, and parentheses, while the parser constructs an Abstract Syntax Tree (AST) to represent the expression's structure, allowing for subsequent analysis or evaluation.

3. Screen printing of program output:

After running the program, the following output is obtained, based on the input text:

```
String input = "a + b * c";
```

The output:

```
IDENTIFIER: a
OPERATOR: +
IDENTIFIER: b
ERATOR: *
IDENTIFIER: c
AST:
Binary Operator: +
Operand: a
Binary Operator: *
Operand: b
Operand: c
```

- The lexer correctly identifies the tokens in the input expression:
  - IDENTIFIER "a"
  - OPERATOR "+"
  - IDENTIFIER "b"
  - OPERATOR "*"
  - IDENTIFIER "c"

The output provides a structured representation of the AST, clearly showing the hierarchical relationships between operators and operands in the input expression. Each node in the AST is labeled to indicate whether it represents a binary operator or an operand (identifier in this case). In summary, the output demonstrates the successful tokenization and parsing of the input expression, resulting in a meaningful Abstract Syntax Tree that captures the expression's structure and facilitates further analysis or evaluation of mathematical expressions.

4. Conclusions:

The laboratory work on parser implementation and Abstract Syntax Tree (AST) construction has provided a foundational understanding of language processing concepts. By delving into tokenization, syntactic analysis, and AST representation, I gained insights into how programming languages are structured and parsed. This hands-on experience is invaluable for grasping the intricacies of compilers, interpreters, and natural language processing systems. Moreover, understanding recursive parsing techniques and error handling mechanisms enhances the ability to tackle complex language-related tasks with confidence. This work sets the stage for exploring more advanced topics in language processing, such as semantic analysis, code generation, and optimizing compilers, empowering me to engage in sophisticated language-related projects in the future.