Ministerul Educaţiei și Cercetării al Republicii Moldova Universitatea Tehnică a Moldovei

Facultatea Calculatoare, Informatică și Microelectronică

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

Elaborated:

st. gr. FAF-223 Bujilov Dmitrii

Verified:

asist. univ. Dumitru Cretu

Chişinău - 2024

Objectives:

Get familiar with parsing, what it is and how it can be programmed.

Get familiar with the concept of AST.

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

Parsing, a cornerstone of computer science, entails the systematic analysis of strings based on the rules of a formal grammar. This process is pivotal across diverse domains, including compiler construction, natural language processing, and data validation. At its core, parsing involves breaking down input sequences into smaller, more manageable units to comprehend their structural composition and semantic significance.

The Lexer, often referred to as the tokenizer, inaugurates the parsing journey by transforming raw sequences of characters into meaningful tokens. These tokens serve as the elemental building blocks of language or data format interpretation. Within the context of arithmetic expressions, tokens encompass numeric values, arithmetic operators (+, -, *, /), and parentheses, each imbued with distinctive roles in the expression's syntactic structure.

Regular Expressions, versatile patterns for string matching, assume a central role in the lexer's functionality. Leveraging regular expressions, the lexer adeptly discerns and classifies tokens by identifying recurring patterns within the input string. For instance, numeric tokens are recognized through digit patterns, while operators are identified based on specific character sequences.

Abstract Syntax Trees (ASTs) stand as architectural marvels in the realm of parsing. Acting as hierarchical representations of program syntax, ASTs abstract away textual intricacies, providing a structured blueprint of a program's logical composition. In essence, each node within an AST embodies a distinct syntactic construct, with edges delineating relationships between these constructs. For arithmetic expressions, ASTs serve as powerful tools for organizing and analyzing expression components in a hierarchical fashion, facilitating subsequent syntactic analysis.

The Lexer, embodied by the ArithmeticLexer.java class, serves as the bedrock of tokenization. Through iterative processing of input characters, the lexer diligently applies predefined rules to segment the input string into discrete tokens. To fortify robustness, error handling mechanisms are strategically integrated, enabling the detection and resolution of invalid expressions with finesse.

Token types, enshrined within the TokenType.java enumeration, epitomize the diverse categories of tokens that the lexer can discern. This systematic categorization enhances clarity and facilitates subsequent token processing stages by providing a standardized classification framework.

The Token.java class stands as a testament to token encapsulation, embodying essential metadata such as token type, value, and position within the input string. By encapsulating token properties within a dedicated class, the implementation fosters modularity and extensibility, laying a solid foundation for subsequent parsing stages.

Within the ASTNode.java class, the intricate structure of Abstract Syntax Tree nodes is meticulously delineated, encompassing properties such as token type, value, and references to child nodes. Meanwhile, the ASTBuilder.java class undertakes the arduous task of constructing ASTs from token sequences, employing sophisticated recursive algorithms that leverage operator precedence to organize nodes effectively.

In the realm of practical application, users wield the implemented lexer and parser to navigate the intricate landscape of arithmetic expressions. Through a seamless interplay of tokenization, AST construction, and syntactic analysis, users can unravel the nuances of arithmetic expressions with unparalleled precision and clarity.

By delving into the intricate world of parsing, users gain invaluable insights into the underlying principles and techniques that underpin language processing systems. Through hands-on exploration, users embark on a transformative journey that transcends theoretical abstractions, empowering them to wield parsing prowess with confidence and finesse.

Main parts of the code are:

```
    private ASTNode buildASTRecursive(int start, int end) {
        if (start > end) {
            return null;
        }
        // Find the operator with the lowest precedence
        int lowestPrecedence = Integer.MAX_VALUE;
        int index = -1;
        for (int i = start; i <= end; i++) {
            Token token = tokens.get(i);
            if (token.getType() == TokenType.OPERATOR) {
                int precedence = getPrecedence(token);
                if (precedence <= lowestPrecedence) {
                    lowestPrecedence = precedence;
                    index = i;
                }
            }
        }
        if (index == -1) {
            // If no operator found, it must be a single token (operand)
            return new ASTNode(tokens.get(start).getType(), tokens.get(start).getValue());
        }
        // Recursively build left and right subtrees
        ASTNode node = new ASTNode(tokens.get(index).getType(), tokens.get(index).getValue());
        node.setLeftChild(buildASTRecursive(start, index - 1));
        node.setRightChild(buildASTRecursive(index + 1, end));
        return node;
    }
    private void printASTRecursive(ASTNode node, int indentLevel) {
        if (node == null) {
            return;
        }
        for (int i = 0; i < indentLevel; i++) {
            System.out.print("  ");
        }
        System.out.println(node);
        // Print left subtree
        printASTRecursive(node.getLeftChild(), indentLevel + 1);
        // Print right subtree
        printASTRecursive(node.getRightChild(), indentLevel + 1);
    }
public enum TokenType {
    NUMBER,         // Represents numeric values.
    OPERATOR,       // Represents arithmetic operators (+, -, *, /).
    LEFT_PAREN,     // Represents left parenthesis '('.
    RIGHT_PAREN,    // Represents right parenthesis ')'.
    WHITESPACE,     // Represents whitespace characters.
    ERROR           // Represents an error token.
}
public class ASTNode {
    private final TokenType type;
    private final String value;
    private ASTNode leftChild;
    private ASTNode rightChild;
    public ASTNode(TokenType type, String value) {
        this.type = type;
        this.value = value;
    }
    public TokenType getType() {
        return type;
    }
    public String getValue() {
        return value;
    }
    public ASTNode getLeftChild() {
        return leftChild;
    }
    public void setLeftChild(ASTNode leftChild) {
        this.leftChild = leftChild;
```

```
    }
    public ASTNode getRightChild() {
        return rightChild;
    }
    public void setRightChild(ASTNode rightChild) {
        this.rightChild = rightChild;
    }
    @Override
    public String toString() {
        return "[" + type + ": " + value + "]";
    }
}
```

Conclusions and Results

Wrapping up our journey through parsing, we've stumbled upon a treasure trove of insights into the inner workings of language processing. From the nitty-gritty details of tokenization to the elegant structures of Abstract Syntax Trees (ASTs), each step of this adventure has been a revelation, illuminating the fascinating world of parsing.

Exploring the intricacies of lexer implementation has been both enlightening and challenging. We've delved into the magic of regular expressions, marveling at their ability to transform raw strings into meaningful tokens. Through trial and error, we've honed our lexer, ensuring it can gracefully handle unexpected input with resilience and grace.

The discovery of ASTs has been a highlight of our journey, offering a new perspective on syntactic analysis. With each node representing a unique syntactic element and edges connecting them in intricate patterns, ASTs provide a captivating glimpse into the underlying structure of program syntax, transcending the confines of mere text.

As we bid farewell to this chapter of our parsing adventure, we do so with a sense of accomplishment and anticipation. Equipped with a deeper understanding of parsing principles, we're ready to tackle new challenges and unravel the mysteries of language and syntax with confidence and clarity.