Parser & Building an Abstract Syntax Tree

Course: Formal Languages & Finite Automata

Author: Cravcenco Dmitrii

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

Implementation description

CustomAstBuilder

The `CustomAstBuilder` class is responsible for building and printing an Abstract Syntax Tree (AST) from a list of tokens. It parses the tokens and constructs a hierarchical representation of the program's structure.

Building the AST

The `buildAst` method is responsible for constructing an abstract syntax tree (AST) from a list of tokens obtained through lexical analysis. Below is a detailed description of what the method does:

1. **Initialization**:
    - Creates an `AstNode` named `programNode` with the type `PROGRAM` and value "Program". This serves as the root node of the AST.
    - Initializes a stack `contextStack` to keep track of the current context during tree construction. The initial context is set to the `programNode`.
2. **Token Processing**:
    - Iterates over each token in the input list of tokens.
    - For each token, a switch statement is used to handle different token types.

3. **Token Type Handling**:
   - **TYPE**: If the token represents a type declaration:
     - Checks if the current context is not a line node. If not, creates a new line node and sets it as the current context.
     - Creates a type declaration node with the token value and adds it to the current context's children.
   - **VARIABLE**: If the token represents a variable:
     - Checks if the current context is not a line node. If not, creates a new line node and sets it as the current context.
     - Checks if the first child of the current context is a type declaration node. If yes, creates a variable declaration node and adds it to the current context's children. Otherwise, creates a variable usage node.
   - **EQUALS**: If the token represents an assignment operator "=":
     - Creates an assignment node and adds it to the current context's children.
   - **VALUE**: If the token represents a value:
     - Creates a value node with the token value and adds it to the current context's children.
   - **PLUS, MINUS, MULTIPLY, DIVIDE**: If the token represents an arithmetic operation:
     - Creates an operation node with the corresponding operation type and adds it to the current context's children.
   - **SEMICOLON**: If the token represents the end of a line:
     - Pops the current context from the stack to end the current line context.
     - Creates a new line node, adds it to the parent context's children, and sets it as the new current context.
4. **Return**:
   - Returns the `programNode`, which represents the root of the constructed AST.

Overall, the `buildAst` method iterates over tokens, creates appropriate AST nodes based on token types, maintains the context using a stack, and constructs a hierarchical AST representing the structure of the input code.

```
public static AstNode buildAst(List<Token> tokens) {
    // Initialize the root node of the AST as the program node
    AstNode programNode = new AstNode(AstNodeType.PROGRAM, "Program");

    // Initialize a stack to keep track of the current context (e.g., line or expression)
    Stack<AstNode> contextStack = new Stack<>();
    contextStack.push(programNode); // Start with the program node as the context

    // Iterate through each token
    for (Token token : tokens) {
        switch (token.type()) {
```

```
                // Handle different types of tokens
                case TYPE:
                    // If the current context is not a line, create a new line node
                    // Create a type declaration node and add it to the current context
                    break;
                case VARIABLE:
                    // If the current context is not a line, create a new line node
                    // Determine whether to create a variable declaration or usage node based on
                    break;
                case EQUALS:
                    // Create an assignment node and add it to the current context
                    break;
                case VALUE:
                    // Create a value node and add it to the current context
                    break;
                case PLUS:
                case MINUS:
                case MULTIPLY:
                case DIVIDE:
                    // Create an operation node and add it to the current context
                    break;
                case SEMICOLON:
                    // End the current line context and create a new line node
                    break;
                // Handle other token types as needed
                default:
                    throw new RuntimeException("Unrecognized token: " + token.value());
            }
        }

        return programNode; // Return the root node of the AST
    }
```

Printing the AST

```
    public static void printAst(AstNode node, int depth) {
        String indent = "  ".repeat(depth);

        boolean isLastLineEmpty = node.type == AstNodeType.LINE && node.children.isEmpty();

        // Print the node if it's not an empty line
        if (!isLastLineEmpty) {
            System.out.println(indent + node.type + ": " + node.value);
        }

        // Recursively print child nodes
```

```
    for (AstNode child : node.children) {
        printAst(child, depth + 1);
    }
}
```

The `printAst` method recursively traverses the AST and prints each node with an appropriate indentation level. It skips printing empty line nodes to avoid cluttering the output.

Conclusions and Results

Conclusion

In conclusion, the development of the `CustomAstBuilder` class represents a significant step forward in understanding and implementing parsing techniques and abstract syntax trees (ASTs) within the context of formal languages and finite automata.

Parsing and AST Construction The implementation of the `buildAst` method showcases the intricacies of parsing tokens and constructing an AST to represent the syntactic structure of a custom programming language. By leveraging lexical analysis techniques and regular expressions, the method effectively categorizes tokens into meaningful types, such as types, variables, operators, and values. Subsequently, it organizes these tokens into a hierarchical tree structure that captures the relationships between different elements of the code.

Hierarchical Representation The AST serves as a hierarchical representation of the code's structure, providing a systematic way to navigate and analyze program elements. Each node in the tree corresponds to a specific construct in the code, such as variable declarations, assignments, arithmetic operations, and type declarations. Through the recursive traversal of the AST, it becomes possible to inspect and manipulate the code at different levels of abstraction, facilitating tasks such as optimization, code generation, and error detection.

Learning Outcomes The development of the `CustomAstBuilder` class has provided valuable insights and learning outcomes in the following areas: - Understanding the principles of parsing and lexical analysis. - Appreciating the role of abstract syntax trees in representing program structure. - Gaining proficiency in implementing parsing algorithms and AST construction techniques. - Exploring practical applications of formal language concepts in software development and compiler design.

In summary, the development of the `CustomAstBuilder` class represents a significant milestone in the journey of understanding formal languages and finite

automata. By mastering parsing techniques and AST construction principles, we have laid a solid foundation for tackling more advanced topics in compiler design, language theory, and software engineering.