Parser & Building an Abstract Syntax Tree

Course: Formal Languages & Finite Automata

Author: Popov Nichita

Theory

Parsing and constructing an Abstract Syntax Tree (AST) form the backbone of many programming language compilers and interpreters. The process involves analyzing source code to understand its structure and meaning, facilitating subsequent stages like optimization and code generation. At its core, parsing involves transforming raw code into a structured representation that a computer can manipulate. One common approach is using a parser generator tool like YACC or ANTLR, which takes a formal grammar specification and generates code to recognize and parse input according to that grammar. Once parsed, the code is typically transformed into an AST. An AST is a hierarchical representation of the code's syntax, where each node corresponds to a syntactic construct, like a statement or an expression, and edges represent relationships like containment or order.Building an AST involves traversing the parsed code and constructing nodes for each syntactic element encountered. For instance, in a programming language like Python, an AST might have nodes for functions, loops, and conditional statements, each containing further nodes representing their components. The AST serves as an intermediate representation that captures the essential structure and semantics of the code. This abstraction simplifies subsequent analysis and transformation stages, such as type checking, optimization, and code generation. Overall, parsing and constructing an AST are critical steps in the compilation or interpretation process, enabling efficient and accurate processing of programming language code.

Objectives:

1. Get familiar with parsing, what it is and how it can be programmed [1].
2. Get familiar with the concept of AST.
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

The software is designed to visually represent Abstract Syntax Trees (AST) for arithmetic expressions parsed from text input. The system is structured into multiple components, each responsible for distinct aspects of processing and visualization, as detailed below:

1. **Lexer** (`Lexer.py`):

- This component is responsible for lexical analysis, converting the input text into tokens. Each token represents either an arithmetic operator (`+` , `-` , `*` , `/`), parentheses, or integer values. The lexer handles whitespace and recognizes the end of the input. Here's how tokens are generated:

```
def get_next_token(self):
    while self.current_char is not None:
        ...
        if self.current_char.isdigit():
            return Token(TokenType.INTEGER, self.integer())
        ...
```

2. **Parser** (`Parser.py`):

- Utilizing the tokens produced by the lexer, this component parses the input according to the grammar rules of arithmetic expressions. It constructs an AST where each node represents an arithmetic operation (binary operators) or operand (number). The parsing methods (`expr` , `term` , `factor`) ensure that the operations adhere to conventional arithmetic priorities and associativity:

```
def expr(self):
    node = self.term()
    while self.current_token.type in (TokenType.PLUS, TokenType.MINUS):
        ...
        node = BinaryOperator(left=node, operator=operator, right=self.term())
    return node
```

3. **AST Nodes** (`AST.py`):

- The AST is composed of nodes, specifically `BinaryOperator` and `Number` . Each node type has the capability to add itself and its children to a directed graph, which represents the AST structure. The `BinaryOperator` nodes represent arithmetic operations and include references to their left and right child nodes, whereas `Number` nodes represent operand values:

```
class BinaryOperator(ASTNode):
    def add_to_graph(self, graph, parent=None):
        operator_label = f"{self.operator} (BinaryOperator)"
        operator_node = f"{self.operator}_{id(self)}"
        graph.add_node(operator_node, label=operator_label)
        ...
        self.left.add_to_graph(graph, operator_node)
        self.right.add_to_graph(graph, operator_node)
```

4. **Visualization** (`main.py`):
    - After parsing the expression into an AST, the `visualize_ast` function is called to create a visual representation of the AST using the `networkx` and `matplotlib` libraries. This function constructs a directed graph, sets node labels, and applies a spring layout for clear visualization. The resulting graph is then displayed, showing the structure of the parsed arithmetic expression:

    ```
    def visualize_ast(root):
        G = nx.DiGraph()  # Directed graph to represent the AST
        root.add_to_graph(G)
        pos = nx.spring_layout(G, seed=42)  # Use a fixed seed for consistent layouts
        ...
        plt.show()
    ```

Example Execution:

- The `main` function in `main.py` serves as the entry point of the application. It initializes the lexer and parser with a sample arithmetic expression, "3 + 5 - (7 - 2)". The parser processes this expression to construct the AST, which is then visualized. This example clearly demonstrates the parsing and visualization process, making the program useful for educational purposes or debugging complex expressions:

  ```
  def main():
      text = "3 + 5 - ( 7 - 2 )"
      lexer = Lexer(text)
      parser = Parser(lexer)
      ast = parser.expr()
      visualize_ast(ast)  # Call to visualize the AST
  ```

Code Execution Flow:

- Execution begins in the `main.py` script, which imports and utilizes the components. The process starts with lexical analysis, followed by parsing to build the AST, and concludes with the visualization of the AST. This flow highlights the modular design of the application, allowing for easy maintenance and scalability. This comprehensive and interactive approach provides a clear insight into how arithmetic expressions are parsed and visualized, facilitating understanding and further development.

Results

The results of the implementation illustrate the successful visualization of Abstract Syntax Trees (ASTs) from parsed arithmetic expressions. The system effectively parses input expressions into ASTs, which are then dynamically represented through directed graphs using the `networkx` and `matplotlib` libraries. The visualizations confirm the accuracy of the parsing process, displaying each node and its connections clearly, which corresponds precisely to the structure of the input expression. This visualization not only aids in understanding the hierarchical structure of expressions but also serves as a powerful tool for debugging and educational purposes, offering insights into the parsing logic and node relationships within the AST.

These visualizations enhance the comprehensibility of expression parsing, making the abstract concept of ASTs tangible and visually accessible. By displaying the nodes and their connections, users can easily trace the computation steps and understand the precedence and association of operations in arithmetic expressions.

Conclusion

This project has demonstrated the essential role of ASTs in the interpretation and visualization of arithmetic expressions, a fundamental aspect of compiler design and expression evaluation tools. Through the development and integration of the Lexer, Parser, and AST visualization components, we have successfully translated textual arithmetic expressions into clear and interactive visual structures. This process not only reinforces the theoretical understanding of expression parsing and AST construction but also highlights the practical application of these concepts in software development and educational tools.

The modular design of the implementation ensures that each component—lexical analysis, parsing, and visualization—can be independently developed and tested, enhancing maintainability and scalability. The implementation fulfills educational objectives by providing a clear demonstration of parsing techniques and the visualization of complex structures, preparing users for more advanced topics in compiler construction and syntax analysis.

Overall, the project not only meets its initial goals of parsing and visualizing arithmetic expressions but also extends its utility to debugging and teaching, demonstrating the practical and educational benefits of mastering ASTs in computing contexts. This reinforces the value of visual learning tools in both academic and practical applications, making complex theoretical concepts more accessible and easier to understand.