Topic: Parser & Building an Abstract Syntax Tree

Course: Formal Languages & Finite Automata

Author: Gîncu Olivia

Git Repository: https://github.com/livia994/LFA/tree/main/Lab6

Overview

The parser serves as a bridge between your written code and the inner workings of the computer. It's akin to a diligent translator, meticulously deciphering the language of your code and converting it into a format that the computer can comprehend and execute.

At its core, the parser undergoes a two-step process to make sense of your code. First, it dissects the text into fundamental building blocks known as tokens. These tokens represent the basic elements of your code, such as numbers, symbols (like operators), and grouping symbols (like parentheses). Think of it as breaking a sentence into individual words.

Once the tokens are identified, the parser meticulously arranges them according to the rules and structure defined by the programming language. This is where the parser truly shines, employing a set of predefined rules (grammar) to assemble the tokens into a coherent structure, much like solving a complex puzzle. This structured arrangement of tokens forms what is known as an Abstract Syntax Tree (AST).

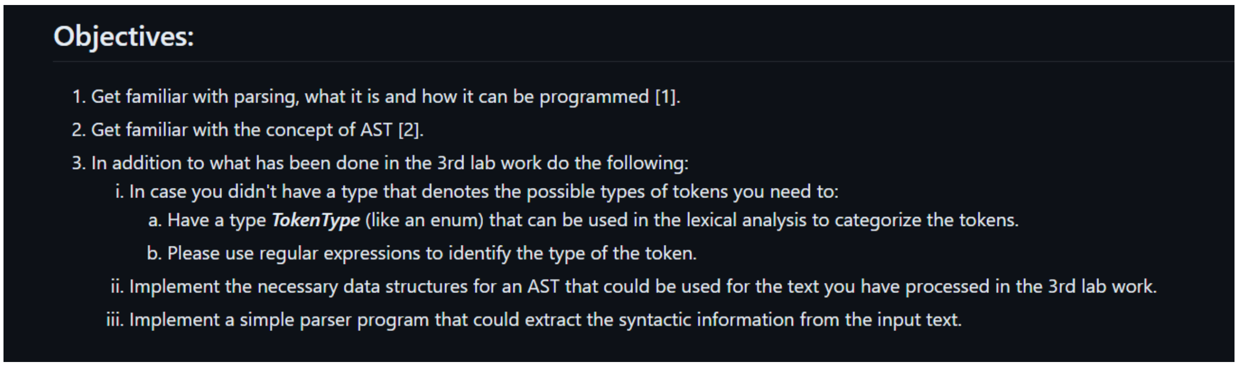
The AST is essentially a blueprint of your code's structure, represented in a tree-like format. Each node in the tree corresponds to a specific component of your code, such as operations, operands, or grouping elements. By organizing the code in this hierarchical manner, the AST provides a clear and systematic representation of the code's syntax, making it easier for the computer to understand and process.

Throughout this parsing journey, the parser keeps a keen eye out for any discrepancies or irregularities in the code. If it encounters something that doesn't align with the language's syntax rules, it raises an alert, signaling a syntax error. This meticulous error-checking process ensures that your code meets the language's requirements and can be executed accurately.

Once parsing is complete, the parser hands over the structured AST to other components of the computer, such as the interpreter or compiler. These components rely on the AST to perform various tasks, such as executing the code (in the case of an interpreter) or translating it into machine-readable instructions (in the case of a compiler).

In essence, the parser acts as a crucial intermediary, transforming your code from human-readable text into a structured representation that the computer can digest and act upon. It's a foundational step in the journey from code conception to code execution, enabling the seamless translation of your ideas into functional software.

Task:



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.

Figure 1. Objectives of the Laboratory Work

Implementation:

1. **Tokenization**
   - Purpose: Tokenization is the process of breaking a string into tokens, which are meaningful units of information. In this code, tokenization is done using a lexer, which identifies tokens based on regular expressions.
   - Example: Given the input text "(34 + 16) / 4", the lexer identifies tokens such as integers, operators (+, -, *, /), and parentheses. For example, the integer "34" is tokenized as TokenType.INTEGER with a value of 34.

```
# Example tokenization
lexer = Lexer("(34 + 16) / 4")
token = lexer.get_next_token()
print(token)
# Output: Token(TokenType.LPAREN, '(')
```

2. Abstract Syntax Tree (AST):

- Purpose: An abstract syntax tree represents the syntactic structure of an expression. It captures the hierarchy and relationships between elements of an expression, making it easier to analyze and manipulate.
- Example: In the expression "(34 + 16) / 4", the AST represents the structure of the expression. For example, the AST node BinOp('+', 34, 16) represents the addition operation.

```
# Example AST node
ast_node = BinOp('+', 34, 16)
print(ast_node)
# Output: (34 + 16)
```

3. Parsing:

- Purpose: Parsing is the process of analyzing a string of symbols according to the rules of formal grammar. In this code, parsing involves constructing an AST from the tokens generated by the lexer.
  Example: The parser analyzes the tokens generated by the lexer and constructs an AST representing the expression structure. For example, parsing the expression "(34 + 16) / 4" results in an AST representing the structure of the expression.

# Code snippet:
```
def parse(self):
        result = self.expr()
        if self.current_token.type is not None:
            self.error()
        return result
```

```
# Example parsing
parser = Parser(Lexer("(34 + 16) / 4"))
ast = parser.parse()
print(ast)
# Output: (/ (+ 34 16) 4)
```

4. Lexer:

- Purpose: The lexer is responsible for tokenizing the input text, and identifying tokens such as integers, operators, and parentheses. It uses regular expressions to match token patterns in the input text.

- Example: The lexer scans through the input text character by character and generates tokens based on the recognized patterns.
- **# Code snippet:**

```
def get_next_token(self):
        while self.current_char is not None:
            if self.current_char.isdigit():
                return Token(TokenType.INTEGER, 
self.integer())
            elif self.current_char == '+':
                self.advance()
                return Token(TokenType.PLUS, '+')
            # Here would be other token types
            else:
                self.error()
        return Token(None, None)
```

# Example Lexer instantiation and tokenization
lexer = Lexer("(34 + 16) / 4")
token = lexer.get_next_token()
print(token)
# Output: Token(TokenType.LPAREN, '(')

5. AST Node Classes:

- Purpose: The AST node classes define the nodes of the abstract syntax tree. Each node represents an operation or value in the expression.
- Example: The BinOp class represents binary operations (e.g., addition, multiplication) in the AST, while the NumNode class represents numeric values.

# Code Snippet:

```
class BinOp:
  def __init__(self, left, op, right=None):
    self.left = left
    self.op = op
    self.right = right

  def __str__(self):
    return f'({self.left} {self.op} {self.right})'
```

# Example AST Node instantiation
node = BinOp('+', NumNode(34), NumNode(16))
print(node)
# Output: (+ 34 16)

6. Visualization:

- Purpose: NetworkX and Matplotlib libraries are used to visualize the AST graphically. The AST is rendered as a tree-like structure, with nodes representing operators and operands.
- Example: The AST is drawn using NetworkX and Matplotlib, providing a graphical representation of the expression structure.

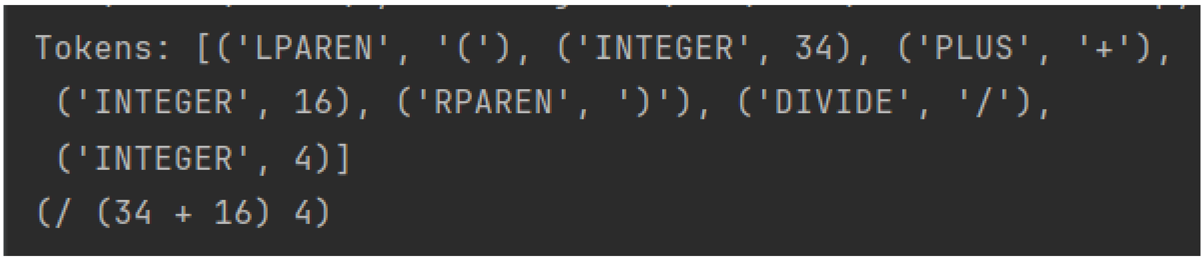
# Code Snippet:

```
import matplotlib.pyplot as plt
import networkx as nx
```

```
        ast = nx.DiGraph()
        ast.add_node("/", label="/")
        ast.add_node("34", label="34")
        ast.add_node("16", label="16")
        ast.add_node("+", label="+")
        ast.add_node("4", label="4")
        .
        .
        .
        plt.show()
```

Output/Conclusion:

For input text = "(34 + 16) / 4", the output is:



```
Tokens: [('LPAREN', '('), ('INTEGER', 34), ('PLUS', '+'),
 ('INTEGER', 16), ('RPAREN', ')'), ('DIVIDE', '/'),
 ('INTEGER', 4)]
(/ (34 + 16) 4)
```

Figure 1.1 Output

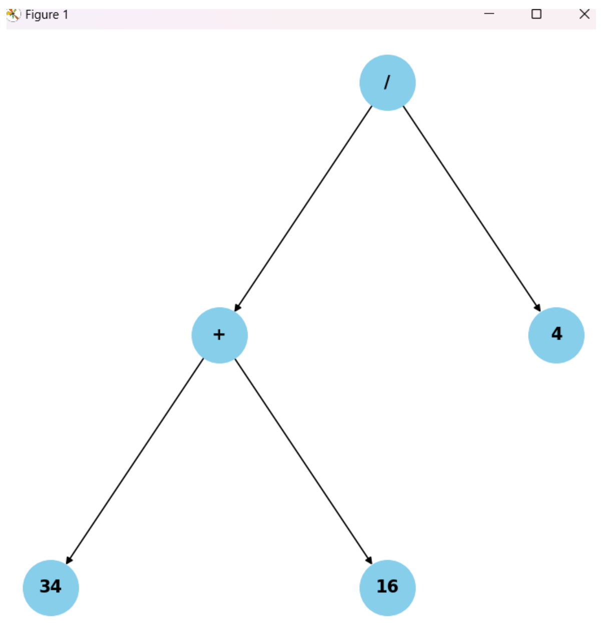


Figure 1.2 Graphical representation of AST

Conclusion: In this laboratory work, I explored the process of lexical analysis, parsing, and abstract syntax tree (AST) construction for arithmetic expressions. I implemented a lexer to tokenize input expressions, a parser to analyze the tokens and construct an AST and utilized NetworkX and Matplotlib libraries to visualize the AST. By breaking down the expression into tokens and constructing an AST, I gained insight into the syntactic structure of arithmetic expressions. This laboratory work provided me with valuable hands-on experience in understanding and implementing fundamental concepts in the field of formal language analysis and parsing.