

Laboratory work nr. 6
Course: Formal languages and finite
automata
Topic: Parser & Building an Abstract
Syntax Tree

Elaborated:
st. gr. FAF-221

Berco Andrei

Verified:
asist. univ.

Cretu Dumitru

Theory

The process of gathering syntactical meaning or doing a syntactical analysis over some text can also be called parsing. It usually results in a parse tree which can also contain semantic information that could be used in subsequent stages of compilation, for example.

Similarly to a parse tree, in order to represent the structure of an input text one could create an Abstract Syntax Tree (AST). This is a data structure that is organized hierarchically in abstraction layers that represent the constructs or entities that form up the initial text. These can come in handy also in the analysis of programs or some processes involved in compilation.

Objectives:

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

Implementation Description

For implementation I chose to use Python, because it is a familiar language.

First of all, I define a class with constructor for grammar. Also, it will have classes for token types, lexer and parser.

```

1  class TokenType(Enum):
2      OPEN_PARENTHESIS = auto()
3      CLOSE_PARENTHESIS = auto()
4      MATH_OPERATION = auto()
5      NUMBERS = auto()
6      START = auto()
7
8  transitions = {
9      TokenType.OPEN_PARENTHESIS: [TokenType.NUMBERS, TokenType.OPEN_PARENTHESIS],
10     TokenType.MATH_OPERATION: [TokenType.NUMBERS, TokenType.OPEN_PARENTHESIS],
11     TokenType.CLOSE_PARENTHESIS: [TokenType.MATH_OPERATION, TokenType.CLOSE_PARENTHESIS],
12     TokenType.NUMBERS: [TokenType.NUMBERS, TokenType.CLOSE_PARENTHESIS, TokenType.MATH_OPERATION],
13     TokenType.START: [TokenType.OPEN_PARENTHESIS, TokenType.NUMBERS]
14 }
15
16 data = {
17     TokenType.OPEN_PARENTHESIS: [r"\\(", r"\\[",
18     TokenType.CLOSE_PARENTHESIS: [r"\\)", r"\\]"),
19     TokenType.MATH_OPERATION: [r"\\+\\-\\*\\/%^"],
20     TokenType.NUMBERS: [r"\\d+"]
21 }

```

Figure 1. TokenType and transitions definition

TokenType Enum: Enumerates token types such as opening and closing parentheses, mathematical operations, and numbers.

Transitions Dictionary: Maps valid transitions between token types during equation parsing.

Data Dictionary: Contains regular expressions for each token type to match symbols in equations.

Functionality: The check_equation function parses equations, categorizes symbols using regular expressions, and validates transitions. The create_graph function visualizes the parsed equation as a tree structure. This code provides a flexible framework for parsing mathematical expressions, making use of enums and regular expressions for efficient tokenization and categorization.

```

1 def lexer(self):
2     self.equation = self.equation.replace(" ", "")
3     seq_parenthesis = []
4     category_mapping = [TokenType.START]
5     failed_on = ""
6     valid_tokens = []
7
8     for symbol in self.equation:
9         # Parenthesis handling
10        if symbol in data[TokenType.OPEN_PARENTHESIS]:
11            seq_parenthesis.append(symbol)
12        elif symbol in data[TokenType.CLOSE_PARENTHESIS]:
13            if not seq_parenthesis:
14                print(f"ERROR: Extra closing parenthesis found.")
15                print(f"Failed on symbol {failed_on}")
16                return False
17            else:
18                last_open = seq_parenthesis.pop()
19                if (symbol == ')' and last_open != '(') or (symbol == ']' and last_open != '['):
20                    print(f"ERROR: Mismatched closing parenthesis found.")
21                    print(f"Failed on symbol {failed_on}")
22                    return False
23
24        # Token categorization using regular expressions
25        found_category = False
26        for category, patterns in data.items():
27            for pattern in patterns:
28                if re.match(pattern, symbol):
29                    current_category = category
30                    found_category = True
31                    break
32            if found_category:
33                break
34        if not found_category:
35            print(f"ERROR: Symbol '{symbol}' does not belong to any known category.")
36            print(f"Failed on symbol {failed_on}")
37            return False
38
39        # Transition checking
40        if current_category not in transitions[category_mapping[-1]]:
41            print(f"ERROR: Transition not allowed from '{category_mapping[-1]}' to '{current_category}'.")
42            print(f"Failed on symbol {failed_on}")
43            return False
44
45        # Update mappings and tokens
46        category_mapping.append(current_category)
47        valid_tokens.append(symbol)
48        failed_on += symbol
49
50    # Check for remaining open parentheses
51    if seq_parenthesis:
52        print(f"ERROR: Not all parentheses were closed.")
53        print(f"Failed on symbol {failed_on}")
54        return False
55    return category_mapping, valid_tokens

```

Figure 2. Method to analyze the input and get the tokens used

Input Cleaning: The method first removes any whitespace from the input equation.

Initialization: It initializes variables to track parentheses, category mappings, failed symbols, and valid tokens.

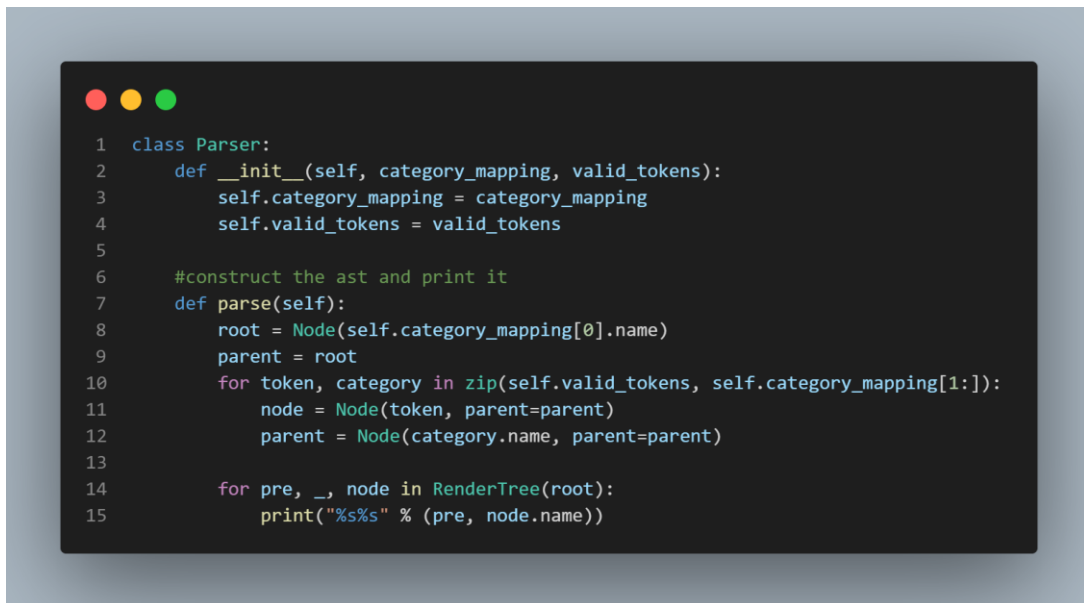
Symbol Processing:

1. The method iterates through each symbol in the equation.
2. It handles parentheses by checking if they are opening or closing. If a closing parenthesis is encountered without a matching opening parenthesis, an error is reported.
3. Symbol categorization is performed using regular expressions defined in the data dictionary. If a symbol does not match any category, an error is reported.
4. The method checks transitions between token categories to ensure validity. If a transition is not allowed, an error is reported.
5. It updates category mappings and collects valid tokens.

Parenthesis Checking: After processing all symbols, the method checks if there are any remaining open parentheses. If so, an error is reported.

Output: Finally, the method returns the category mapping and valid tokens if no errors are encountered during tokenization.

This code segment provides a systematic approach to tokenize mathematical equations, ensuring proper handling of symbols, categorization, and error reporting. It integrates regular expressions for efficient symbol categorization and employs error checks to ensure the integrity of the tokenization process.



```
1 class Parser:
2     def __init__(self, category_mapping, valid_tokens):
3         self.category_mapping = category_mapping
4         self.valid_tokens = valid_tokens
5
6     #construct the ast and print it
7     def parse(self):
8         root = Node(self.category_mapping[0].name)
9         parent = root
10        for token, category in zip(self.valid_tokens, self.category_mapping[1:]):
11            node = Node(token, parent=parent)
12            parent = Node(category.name, parent=parent)
13
14        for pre, _, node in RenderTree(root):
15            print("%s%s" % (pre, node.name))
```

Figure 3. Method to remove unproductive symbols

The provided code defines a Parser class responsible for constructing an Abstract Syntax Tree (AST) from category mappings and valid tokens obtained from the lexical analysis. Here's an overview of its functionality:

Initialization: The class constructor (`__init__`) takes `category_mapping` and `valid_tokens` as arguments and initializes instance variables to store these values.

Parsing Method (parse):

The parse method constructs the AST by iterating over the `valid_tokens` and `category_mapping`. It creates nodes for each token and its corresponding category in the AST using the `Node` class from the `anytree` library.

Nodes are organized hierarchically according to their category mappings, with each token being a child node of its category node.

The method then prints the AST in a hierarchical format using the RenderTree function from the anytree library.

Output: The AST is printed in a hierarchical format, with each node representing a token or a category, along with its parent-child relationships.

This code segment provides a simple yet effective way to construct and visualize an AST from tokenized mathematical equations. It utilizes the anytree library to manage the hierarchical structure of the AST and offers a clear representation of the equation's syntactic structure.

Screenshots

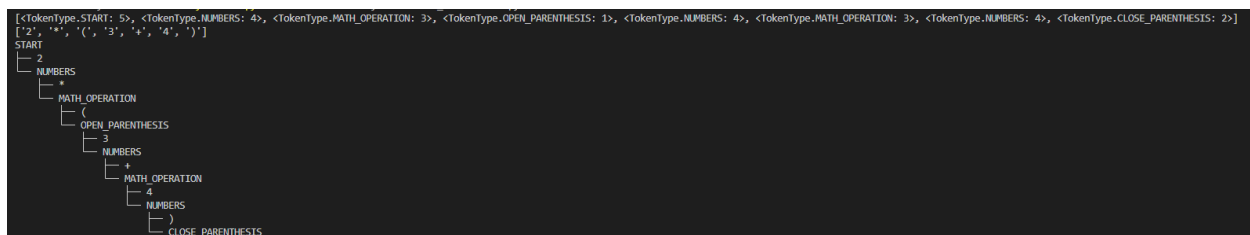


Figure 6. Output for “2 * (3 + 4)”

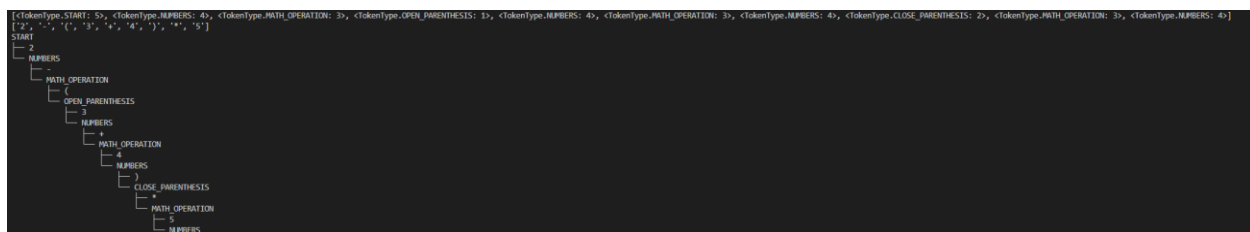


Figure 7. Output for “2 - (3 + 4) * 5”

Conclusions

In this report, we have explored the implementation of a lexical analysis and parsing system for mathematical equations in Python. The system utilizes object-oriented programming concepts, regular expressions, and tree data structures to tokenize, categorize, and construct Abstract Syntax Trees (ASTs) for mathematical expressions.

First, we introduced the `TokenType` enum class, which enumerates the different types of tokens that can appear in mathematical equations, such as opening and closing parentheses, mathematical operations, and numerical values. We then defined dictionaries like `transations` and `data` to map valid transitions between token types during equation parsing and to store regular expressions for token categorization, respectively.

The lexical analysis process, implemented within the `lexer` method, cleans the input equation,

categorizes symbols using regular expressions, validates transitions between token categories, and reports errors if encountered. This process ensures proper tokenization of equations and prepares the input for parsing.

The parsing phase, encapsulated within the ``Parser`` class, constructs ASTs from category mappings and valid tokens obtained during lexical analysis. The ``parse`` method of the ``Parser`` class iterates through the tokens and mappings, creating nodes for each token and its corresponding category in the AST. The resulting AST is then printed in a hierarchical format using the ``anytree`` library, providing a clear visualization of the equation's syntactic structure.

Overall, the implemented system offers a robust framework for lexical analysis and parsing of mathematical equations, demonstrating effective utilization of Python's features and libraries. It provides a foundation for further development of mathematical expression evaluators or interpreters and highlights the importance of proper tokenization and parsing in computational systems.