Ministerul Educaţiei și Cercetării al Republicii Moldova
Universitatea Tehnică a Moldovei
Facultatea Calculatoare, Informatică și Microelectronică

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

Elaborated:
st. gr. FAF-221 Revenco Victor

Verified:
asist. univ. Cretu Dumitru

Chişinău - 2024

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

```
class TokenType(Enum):
    OPEN_PARENTHESIS = auto()
    CLOSE_PARENTHESIS = auto()
    MATH_OPERATION = auto()
    NUMBERS = auto()
    START = auto()

transations = {
    TokenType.OPEN_PARENTHESIS: [TokenType.NUMBERS, TokenType.OPEN_PARENTHESIS],
    TokenType.MATH_OPERATION: [TokenType.NUMBERS, TokenType.OPEN_PARENTHESIS],
    TokenType.CLOSE_PARENTHESIS: [TokenType.MATH_OPERATION, TokenType.CLOSE_PARENTHESIS],
    TokenType.NUMBERS: [TokenType.NUMBERS, TokenType.CLOSE_PARENTHESIS, TokenType.MATH_OPERATION],
    TokenType.START: [TokenType.OPEN_PARENTHESIS, TokenType.NUMBERS]
}

data = {
    TokenType.OPEN_PARENTHESIS: [r"\(", r"\["],
    TokenType.CLOSE_PARENTHESIS: [r"\)", r"\]"],
    TokenType.MATH_OPERATION: [r"[+\-*/%^]"],
    TokenType.NUMBERS: [r"\d+"]
}
```

Figure 1. Definitions

Enum for Token Types: Lists parentheses, math operations, and numbers. Transitions Dictionary: Defines valid token type transitions for parsing equations. Data Dictionary: Stores regex patterns for symbol matching in equations. Functionality: check_equation parses equations, categorizes symbols, and validates transitions. create_graph visualizes equations as trees.

```
def lexer(self):
    self.equation = self.equation.replace(" ", "")
    seq_parenthesis = []
    category_mapping = [TokenType.START]
    failed on = ""
    valid_tokens = []

    for symbol in self.equation:
        if symbol in data[TokenType.OPEN_PARENTHESIS]:
            seq_parenthesis.append(symbol)
        elif symbol in data[TokenType.CLOSE_PARENTHESIS]:
            if not seq_parenthesis:
                print(f"ERROR: Extra closing parenthesis found.")
                print(f"Failed on symbol {failed_on}")
                return False
            else:
                last_open = seq_parenthesis.pop()
                if (symbol == ')' and last_open != '(') or (symbol == ']' and
last_open != '['):
                    print(f"ERROR: Mismatched closing parenthesis found.")
                    print(f"Failed on symbol {failed_on}")
                    return False

        found_category = False
        for category, patterns in data.items():
            for pattern in patterns:
                if re.match(pattern, symbol):
                    current_category = category
                    found_category = True
                    break
            if found_category:
                break
        if not found_category:
            print(f"ERROR: Symbol '{symbol}' does not belong to any known
category.")
            print(f"Failed on symbol {failed_on}")
            return False

        if current_category not in transations[category_mapping[-1]]:
            print(f"ERROR: Transition not allowed from '{category_mapping[-
1]}' to '{current_category}'.")
            print(f"Failed on symbol {failed_on}")
            return False

        category_mapping.append(current_category)
        valid_tokens.append(symbol)
        failed_on += symbol

    if seq_parenthesis:
```

```
        print(f"ERROR: Not all parentheses were closed.")
        print(f"Failed on symbol {failed_on}")
        return False
    return category_mapping, valid_tokens
```

Figure 2. Method for token

The code segment includes functions for processing mathematical expressions, ensuring accurate tokenization and error handling. Here's a breakdown of its functionality:The method removes any whitespace from the input equation to prepare it for processing. This step initializes variables to track parentheses, category mappings, failed symbols, and valid tokens, setting the stage for the tokenization process. The method iterates through each symbol in the equation, handling parentheses, categorizing symbols using regular expressions, checking transitions between token categories, and updating mappings and tokens. After processing all symbols, the method ensures that all parentheses are properly closed, reporting an error if any remain open. Finally, the method returns the category mapping and valid tokens if no errors were encountered during tokenization.

```
class Parser:
    def __init__(self, category_mapping, valid_tokens):
        self.category_mapping = category_mapping
        self.valid_tokens = valid_tokens

    def parse(self):
        root = Node(self.category_mapping[0].name)
        parent = root
        for token, category in zip(self.valid_tokens,
self.category_mapping[1:]):
            node = Node(token, parent=parent)
            parent = Node(category.name, parent=parent)

        for pre, _, node in RenderTree(root):
            print("%s%s" % (pre, node.name))
```

Figure 3. Method to remove unproductive symbols

The provided code defines a Parser class to construct an Abstract Syntax Tree (AST) from category mappings and valid tokens obtained during lexical analysis. Here's how it works: The class constructor (**init**) takes category_mapping and valid_tokens as arguments and initializes instance variables. The parse method iterates over valid_tokens and category_mapping, creating nodes for each token and its category in the AST using the anytree library's Node class. Nodes are organized hierarchically based on category mappings, with tokens as children of their category nodes. The method then prints the AST in a hierarchical format using the RenderTree function from the anytree library. The AST is printed in a hierarchical format, showing each node representing a token or category and their relationships.

Screenshots

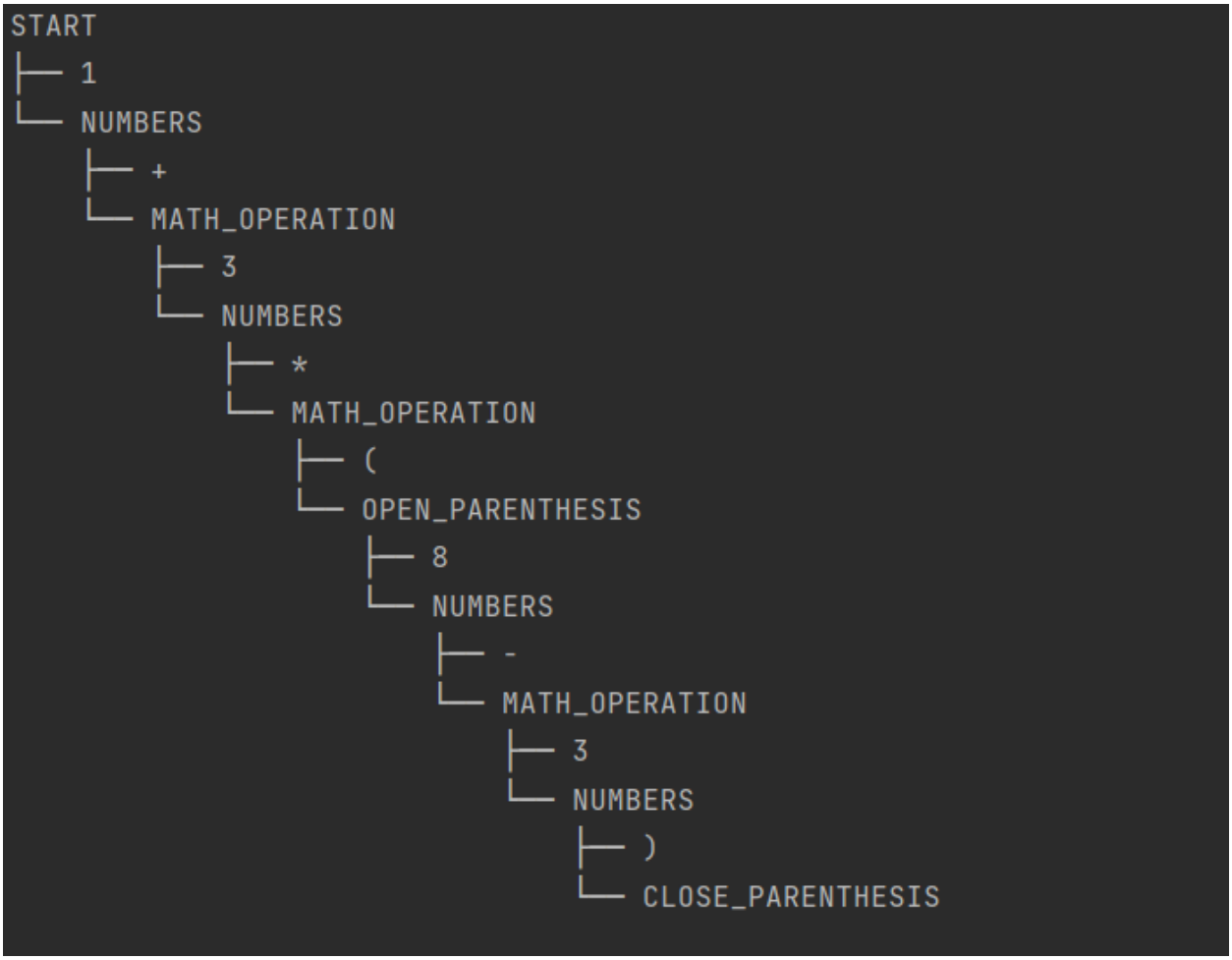

Figure 6. Output for "1+3(8-3"*

Conclusions

This report discusses a lexical analysis and parsing system for mathematical equations in Python, focusing on tokenization, categorization, and Abstract Syntax Tree (AST) construction. The system uses object-oriented programming, regular expressions, and tree data structures.

The **TokenType** Enum enumerates token types like parentheses, math operations, and numbers. Dictionaries (**transitions** and **data**) map valid transitions between token types and store regex for token categorization.

The **lexer** method cleans input, categorizes symbols with regex, validates transitions, and reports errors. The **Parser** class constructs ASTs from mappings and tokens. The **parse** method creates

nodes for tokens and categories in the AST.

The AST is printed hierarchically using the **anytree** library, providing a clear visualization of the equation's structure. This system is a foundation for lexical analysis and parsing of math equations in Python, showcasing its features and libraries effectively.