**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 Gavriliuc Tudor

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.

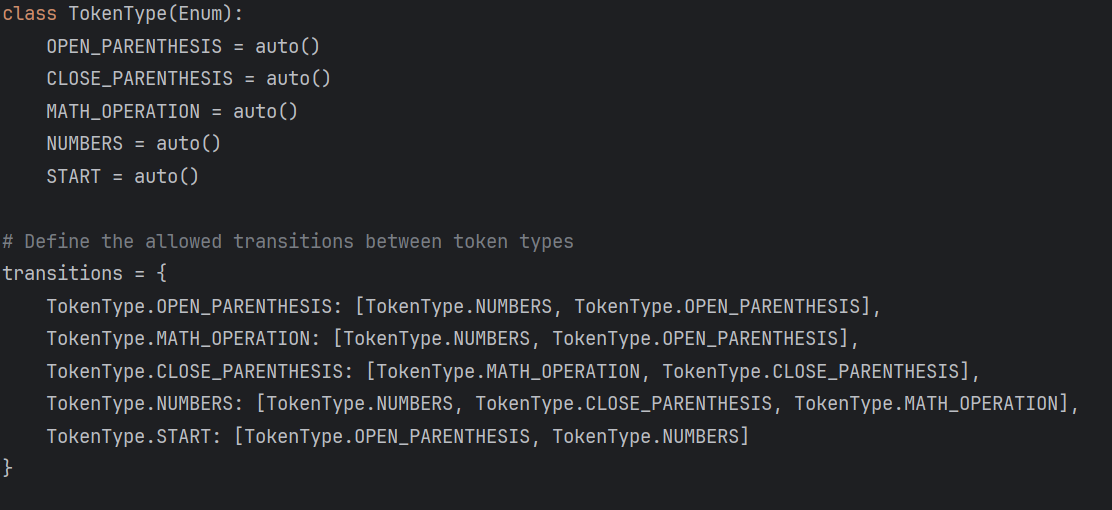
**Objectives:**

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**

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.

****

**Figure 1. TokenType and transitions definition**

The provided code segment includes the following components:

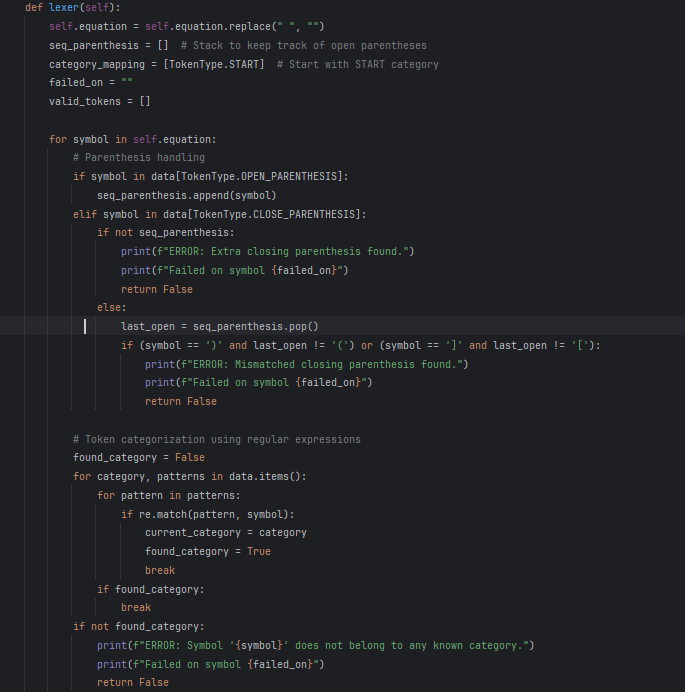
1. TokenType Enum: Enumerates different token types found in equations, including parentheses, mathematical operations, and numbers.

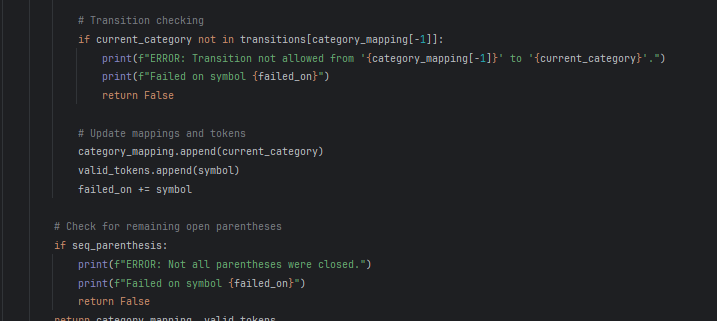
2. Transitions Dictionary: Maps valid transitions between token types during equation parsing.

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

4. 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 offers a versatile framework for parsing mathematical expressions. It leverages enums and regular expressions for efficient tokenization and categorization, providing a solid foundation for further development in equation processing..





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

The given code segment outlines a method for tokenizing mathematical equations. Here's a summary of its key functionalities:

Input Cleaning: The method starts by removing any whitespace from the input equation.

Initialization: It sets up variables to keep track of parentheses, category mappings, failed symbols, and valid tokens. Symbol Processing:

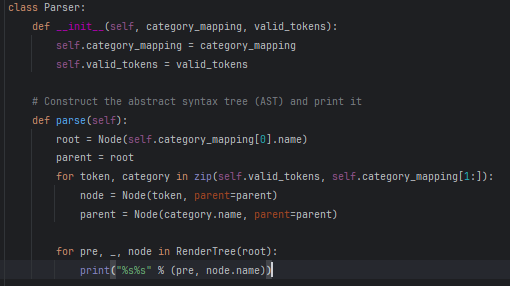
1. The method loops through each symbol in the equation.

2. It handles parentheses, ensuring that each closing parenthesis has a matching opening parenthesis. If not, it reports an error.

3. Symbols are categorized using regular expressions defined in the data dictionary. If a symbol doesn't match any category, an error is reported.

4. The method checks transitions between token categories to ensure they are valid. 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, it reports an error.

Output: Finally, the method returns the category mapping and valid tokens if no errors occur during tokenization. This approach provides a systematic method for tokenizing mathematical equations, ensuring correct symbol handling, categorization, and error detection. It uses regular expressions for efficient symbol categorization and implements checks to maintain tokenization integrity.

**Figure 3. Method to remove unproductive symbols**

The given code defines a Parser class that is responsible for creating an Abstract Syntax Tree (AST) from the category mappings and valid tokens obtained during lexical analysis. Here is a summary of its main functionality: Initialization: The class's constructor (init) accepts category\_mapping and valid\_tokens as arguments and sets instance variables to store these values.

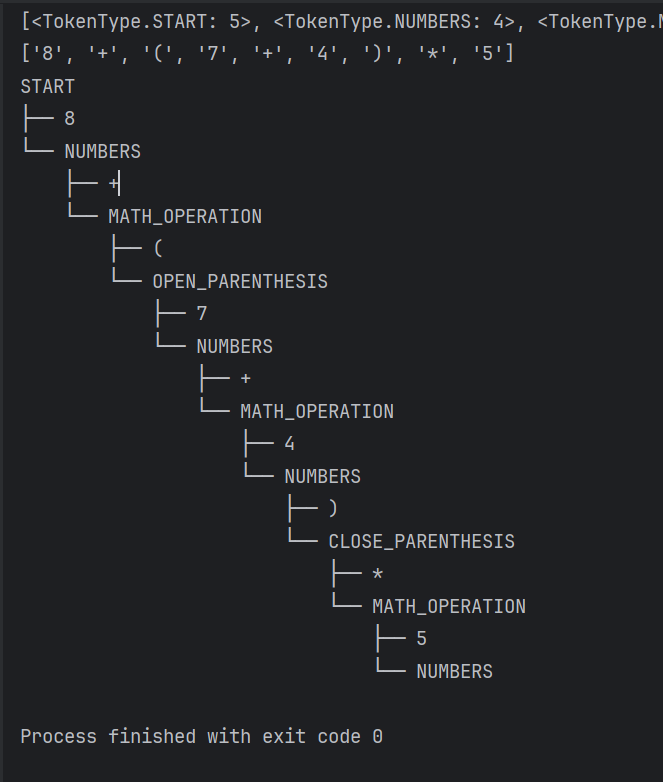
Parsing Method (parse):

The parse method builds the AST by iterating through the valid\_tokens and category\_mapping. It creates nodes in the AST for each token and its corresponding category, utilizing the Node class from the anytree library. Nodes are structured hierarchically based on their category mappings, with each token being a child node of its category node. The method then displays the AST in a hierarchical format using the RenderTree function from the anytree library.

Output:

The AST is displayed in a hierarchical format, where each node represents a token or a category, along with its parent-child relationships. This code segment offers a straightforward and efficient approach to construct and visualize an AST from tokenized mathematical equations. It leverages the anytree library to manage the hierarchical structure of the AST, providing a clear illustration of the equation's syntax.

**Screenshots**



**Figure 6. Output for “8 + (7 + 4)\*5”**

**Conclusions**

In this study, we have examined the design of a lexical analysis and parsing system for mathematical expressions in Python. This system utilizes concepts from object-oriented programming, regular expressions, and tree data structures to tokenize, categorize, and build Abstract Syntax Trees (ASTs) for mathematical expressions. We began by introducing the `TokenType` enum class, which defines the various types of tokens found in mathematical expressions, including parentheses, mathematical operations, and numerical values. We also created dictionaries such as `transitions` and `data` to manage valid transitions between token types during parsing and to store regular expressions for token categorization, respectively. The lexical analysis process, implemented within the `lexer` method, first sanitizes the input equation by removing spaces. It then categorizes symbols using regular expressions, validates transitions between token categories, and handles errors if any are encountered. This process ensures that equations are properly tokenized and ready for parsing. The parsing phase, managed by the `Parser` class, constructs ASTs from the 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 displayed in a hierarchical format using the `anytree` library, providing a visual representation of the equation's syntactic structure. Overall, the developed system provides a solid foundation for lexical analysis and parsing of mathematical expressions, showcasing the effective use of Python's capabilities and libraries. It serves as a stepping stone for the development of mathematical expression evaluators or interpreters and underscores the significance of accurate tokenization and parsing in computational systems.