**MINISTRY OF EDUCATION AND RESEARCH OF REPUBLIC OF MOLDOVA TECHNICAL UNIVERSITY OF MOLDOVA FACULTY OF COMPUTERS, INFORMATICS AND MICROELECTRONICS DEPARTMENT OF SOFTWARE ENGINEERING AND AUTOMATICS**

**Laboratory work 6:**

# Topic: Parser & Building an Abstract Syntax Tree

**Course: Formal Languages & Finite Automata**

**Author:Ostafi Eugen, FAF-222**

**Chișinău, 2024**

**THEORY**

Introduction to Parsing Parsing is a fundamental process in computer science, primarily used to extract syntactical meaning from text. It is integral to the functioning of compilers and interpreters, where it involves analyzing a string of symbols based on formal grammar rules. The primary outcome of parsing is the construction of a parse tree, which outlines the syntactic structure of the input and may also embed semantic information useful for subsequent compilation stages. This capability is crucial for the software to interpret and manipulate the input code or data accurately, making parsing a cornerstone of programming language implementation and data processing. Understanding Abstract Syntax Trees (AST) An Abstract Syntax Tree (AST) represents the syntactic structure of input text through a hierarchy of abstraction layers. Unlike simple parse trees, ASTs focus on the high-level constructs of the text, omitting unnecessary syntactic details. Each node in the AST corresponds to a specific construct in the input, organized to reflect their syntactic and sometimes semantic relationships. ASTs are invaluable in software development, particularly in the areas of compilation and code optimization. By abstracting the input text into a structured tree format, ASTs allow for more efficient analysis and manipulation of code. They facilitate various compiler optimizations by providing a clear framework for implementing transformation rules and conducting static code analysis efficiently. Together, parsing and AST construction provide the tools necessary for effective software development, enabling precise control over the processing of programming languages and data formats. By transforming text into structured data, these processes support a wide range of applications, from simple code compilation to complex software engineering tasks.

**OBJECTIVES**

1. Get familiar with parsing, what it is and how it can be programmed.

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

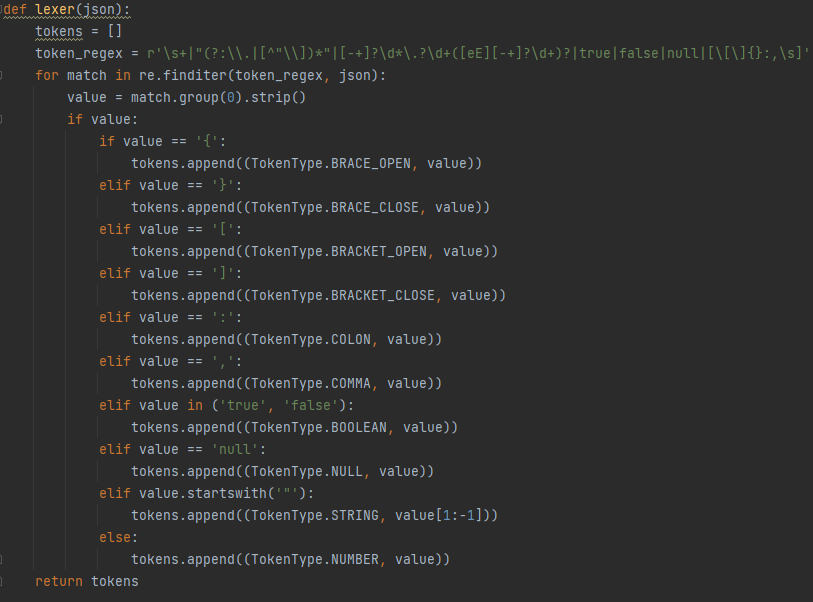
you have processed in the 3rd lab work.

iii. Implement a simple parser program that could extract the syntactic information

from the input text.

**IMPLEMENTATION DESCRIPTION**

1. **Lexer**

****

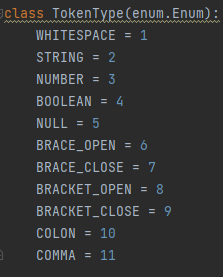
Lexer is part of a compiler or interpreter that analyzes a sequence of characters like source code to produce a sequence of tokens. Tokens are the meaningful elements used for further parsing and semantic analysis. This specific lexer is designed for tokenizing JSON data.

The function lexer(json) initializes an empty list tokens to store the tokens generated during the analysis. The token\_regex defines a complex pattern that matches different JSON tokens, including symbols like curly braces, square brackets, colons, and commas, Boolean values like true and false, null, numbers including those in scientific notation, and strings which are characters enclosed in double quotes.

Using re.finditer, the function iterates over all matches of the token\_regex in the input JSON string. Each match is stripped of whitespace and then processed to determine its type. Special characters like braces, brackets, colons, and commas are identified by their specific characters and added to the token list with their corresponding token type. Boolean values and null are recognized specifically. Numbers are processed if a token does not match any previous conditions and does not start with a quotation mark. Tokens starting and ending with quotation marks are recognized as strings.

After processing all characters, the function returns the list of tokens. This lexer is a crucial first step in processing JSON data, allowing a program to understand and manipulate the structure and contents of JSON inputs effectively.

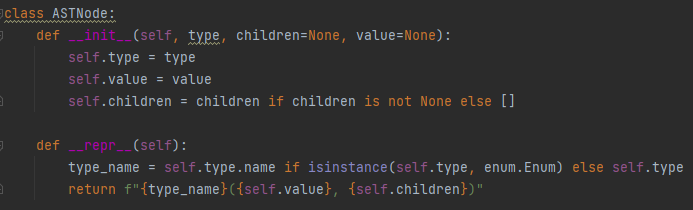
1. **TokenType**

****

TokenType used in Python to define a set of named constants, each representing a specific type of token that can be identified in text, such as JSON data. This enum class includes various token types each associated with a unique integer value. It starts with WHITESPACE which is assigned the number 1 and is likely used to represent spaces or tabs in the text. STRING is given the number 2, used to identify text surrounded by quotes. NUMBER has the value 3, representing numerical values. BOOLEAN is assigned 4, used for true and false values. NULL, with a value of 5, represents the null value. The braces and brackets used to denote structure in JSON are represented as BRACE\_OPEN, BRACE\_CLOSE, BRACKET\_OPEN, and BRACKET\_CLOSE with values 6, 7, 8, and 9 respectively. COLON and COMMA, which are used as separators in JSON, are assigned the values 10 and 11. This enumeration facilitates the identification and processing of these various elements when parsing text, ensuring each type of token can be recognized and handled appropriately during lexical analysis of code or

data formats like JSON.

1. **ASTNode**

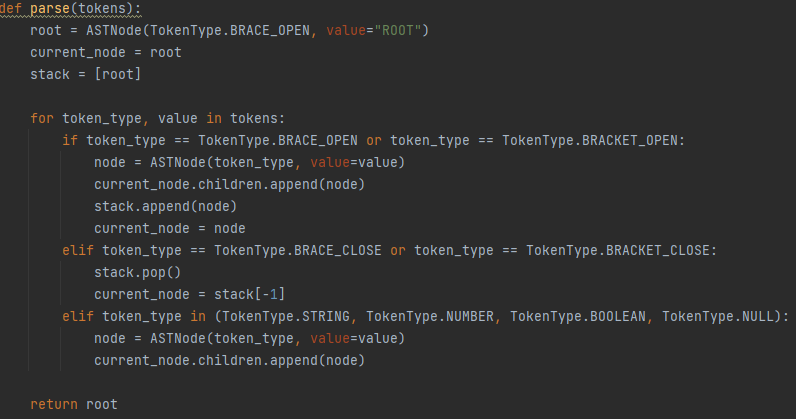
****

Abstract Syntax Tree Node. This class is used to represent a node in an abstract syntax tree, a data structure commonly used in compilers to represent the structure of program code.

The class is initialized with three parameters type, children, and value, where type is a mandatory parameter, and children and value are optional with default values of None. Within the initialization method, the type attribute is directly assigned the value of the type parameter. The value attribute is similarly assigned the value of the value parameter. The children attribute is assigned the value of the children parameter if it is provided; otherwise, it defaults to an empty list.

The class also includes a repr method that provides a string representation of the object for debugging purposes. This method checks if the type attribute is an instance of an enum, using Python's enum.Enum to do this check. If type is indeed an enum, it converts the type to its name; otherwise, it simply uses the type's value. The string representation includes the type name and optionally includes the node's value and its children if they exist. This makes it easier to visualize the structure of an abstract syntax tree when debugging or logging.

**4. Parser**

****

Parse which is designed to build an abstract syntax tree, or AST, from a sequence of tokens representing some structured data, possibly JSON. The function begins by creating an initial ASTNode called root with a token type for opening a brace and a value labeled "ROOT". This root node is also the current node in focus and it starts alone in a list called stack that helps track the nodes during the parsing process.

As the function loops through each token, it decides what to do based on the type of token:

If it finds an opening brace or bracket, it creates a new node for this token, adds it as a child to the current node, updates the current node to this new child, and pushes the child to the stack.

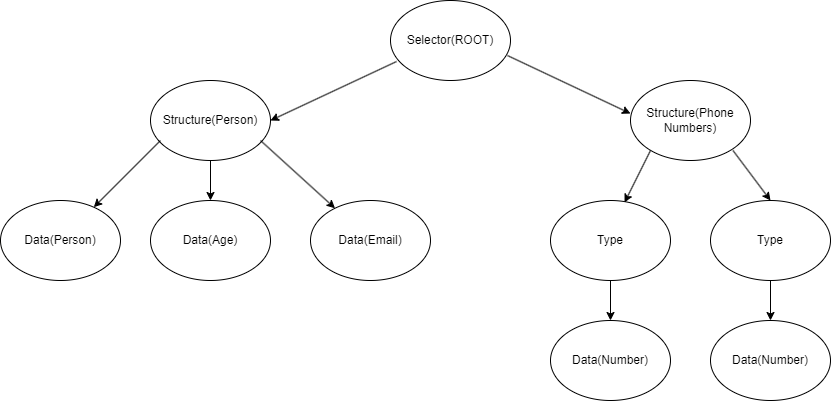
If it finds a closing brace or bracket, it removes the last node from the stack and updates the current node to the next node down in the stack, effectively stepping back up the tree.

If the token is a string, number, boolean, or null, the function simply creates a new node for this token and attaches it as a child to the current node without changing the current position in the tree hierarchy.

Once all tokens are processed, the function returns the root node, which now contains all the other nodes structured according to the input tokens. This tree can be used for further processing or analysis of the structured data.

**Input**





**Conclusion**

Code and its output offer a comprehensive view into the mechanics of regular expression processing and string generation in Python. Initially, the tokenize\_pattern function deconstructs a given regular expression into tokens, which are pairs that associate specific characters or groups of characters with their designated repetition rules. This tokenization is crucial for interpreting the regex in a way that a string generator can utilize.

The build\_strings function then takes these tokens and employs randomness within the bounds defined by the tokens to produce a series of strings that adhere to the original pattern. The output showcases the variety of strings generated from a single regex, highlighting the flexibility of the pattern.

Further clarity is provided by the detail\_string\_construction function, which not only generates a string following the regex rules but also provides a verbose, step-by-step narrative of the generation process. This narrative delineates how each token influences the string being built, making the often opaque process of regex matching more transparent and understandable.

Overall, this set of functions and their output demonstrate a powerful method for regex-driven development, testing, or educational purposes, where understanding and visualizing how patterns translate into potential strings is essential. It reveals the underlying logic of regex string generation, demystifying the complexity of regex patterns through a clear and methodical approach.