**Parser & Building an Abstract Syntax Tree**

**Course: Formal Languages & Finite Automata**

**Author: Vladlen Mîțu**

**Theory:**

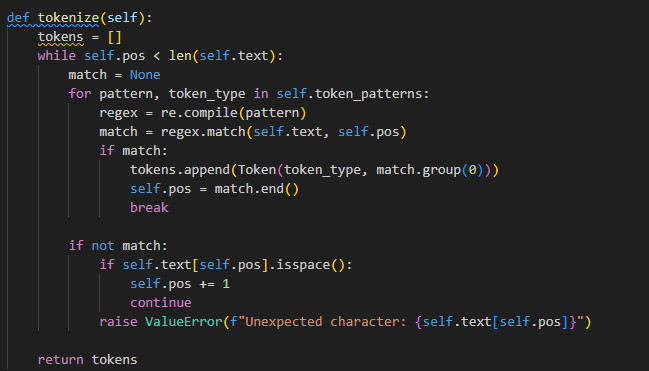
The process of gathering syntactical meaning or doing a syntactical analysis over some text can also be called parsing. It usually results in a parse tree which can also contain semantic information that could be used in subsequent stages of compilation, for example.

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.

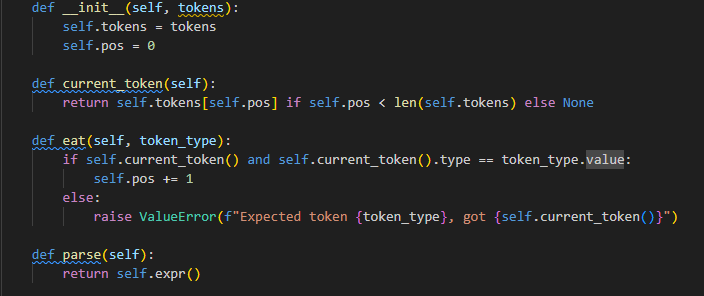
**Objectives:**

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

**Implementation description:**

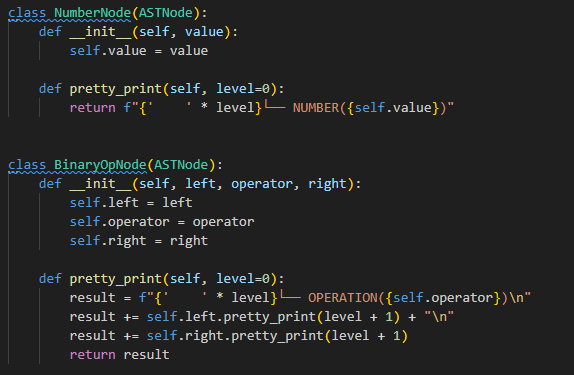
****

The tokenize method in the Lexer class scans the input text and converts it into a list of tokens using regular expressions. It matches patterns for numbers, operators, parentheses, and functions (e.g., sin, cos) and raises an error for unexpected characters.



The methods in the Parser class are responsible for constructing an Abstract Syntax Tree (AST) from a list of tokens:

* current\_token: Returns the current token being processed.
* eat: Validates and consumes the current token if it matches the expected type; otherwise, raises an error.
* parse: Initiates the parsing process and returns the root of the AST.
* factor: Parses numbers, parentheses, or unary operations (e.g., sin, cos).
* term: Parses multiplication and division operations.
* expr: Parses addition and subtraction operations, forming the top-level AST structure.



1. **ASTNode**: A base class for all Abstract Syntax Tree (AST) nodes. It defines a pretty\_print method that must be implemented by subclasses to display the node in a structured format.
2. **NumberNode**: Represents a numeric value in the AST. It stores the value and implements pretty\_print to display the number with proper indentation.
3. **BinaryOpNode**: Represents a binary operation (e.g., addition, multiplication) in the AST. It stores the left operand, operator, and right operand, and its pretty\_print method recursively displays the operation and its operands in a hierarchical format.

**Conclusion:**

In conclusion, this code implements a lexer, parser, and Abstract Syntax Tree (AST) to process mathematical expressions. The lexer tokenizes the input string into tokens using regular expressions. The parser uses these tokens to build an AST, representing the syntactic structure of the expression. The AST supports numbers, binary operations (e.g., addition, multiplication), and unary functions (e.g., sin, cos). The pretty\_print method displays the AST in a hierarchical format, making it easy to visualize the structure of the parsed expression. This code demonstrates a complete pipeline for lexical analysis, parsing, and AST generation.

**References:**

1. Else Course FAF.LFA21.1
2. Parser: https://www.techtarget.com/searchapparchitecture/definition/parser
3. Abstract Syntax Tree: https://dev.to/balapriya/abstract-syntax-tree-ast-explained-in-plain-english-1h38