

**MINISTERUL EDUCAȚIEI, CULTURII ȘI CERCETĂRII AL REPUBLICII MOLDOVA**

**Universitatea Tehnică a Moldovei**

**Facultatea Calculatoare, Informatică şi Microelectronică**

**Departamentul Inginerie Software și Automatică**

**Tabanschi Nichita FAF-222**

**Report**

*Laboratory work n.6*

***of Limbaje Formale și Automate***

Checked by:

**Dumitru Cretu**, *university assistant*

DISA, FCIM, UTM

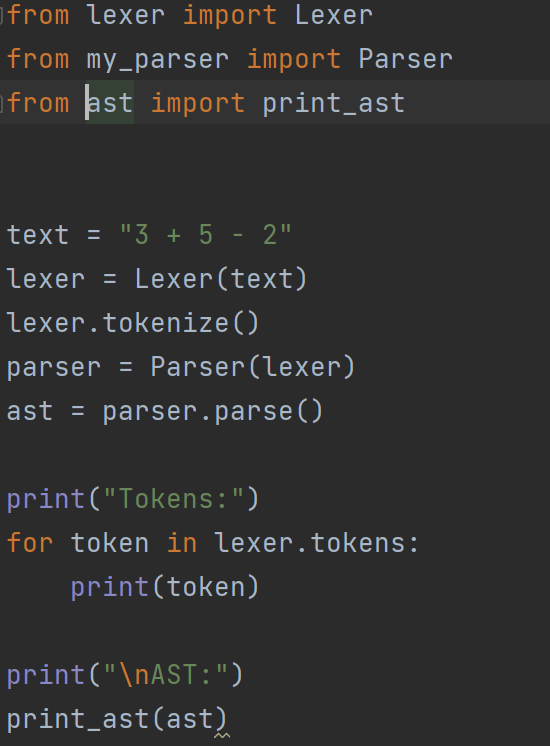
**Chișinău – 2024**

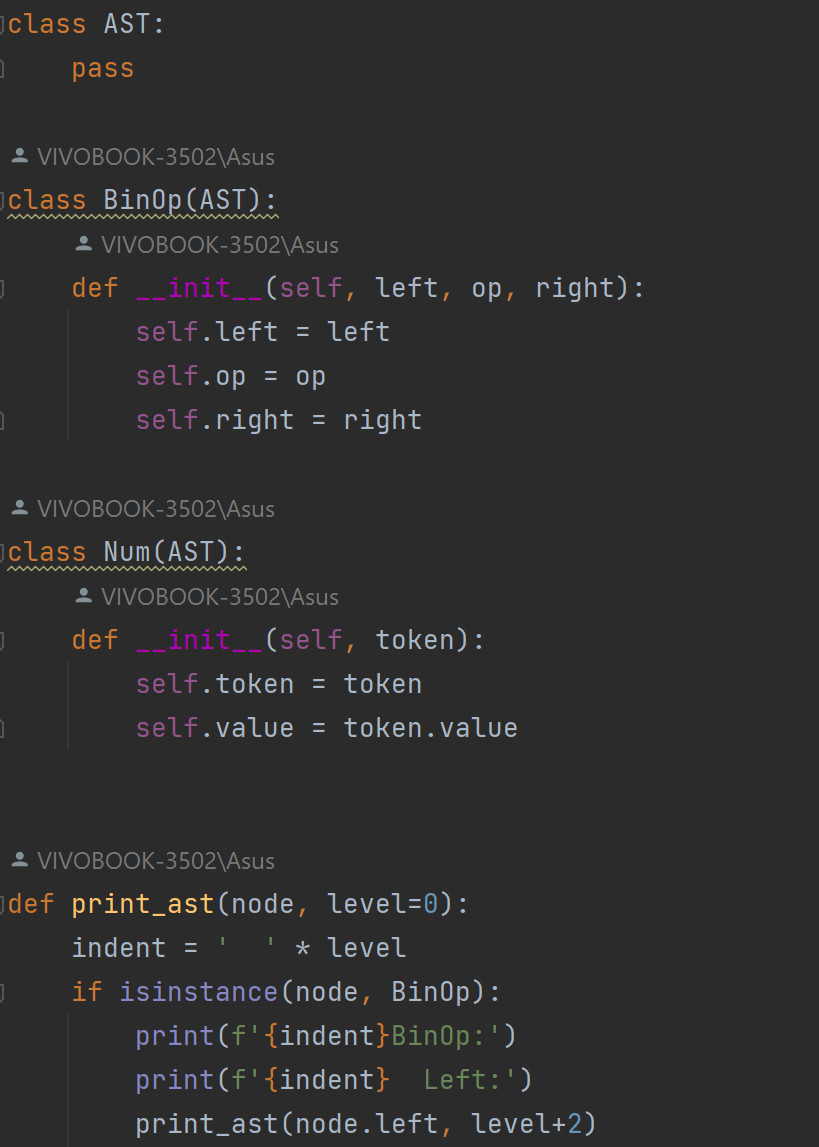
### Course: Formal Languages & Finite Automata

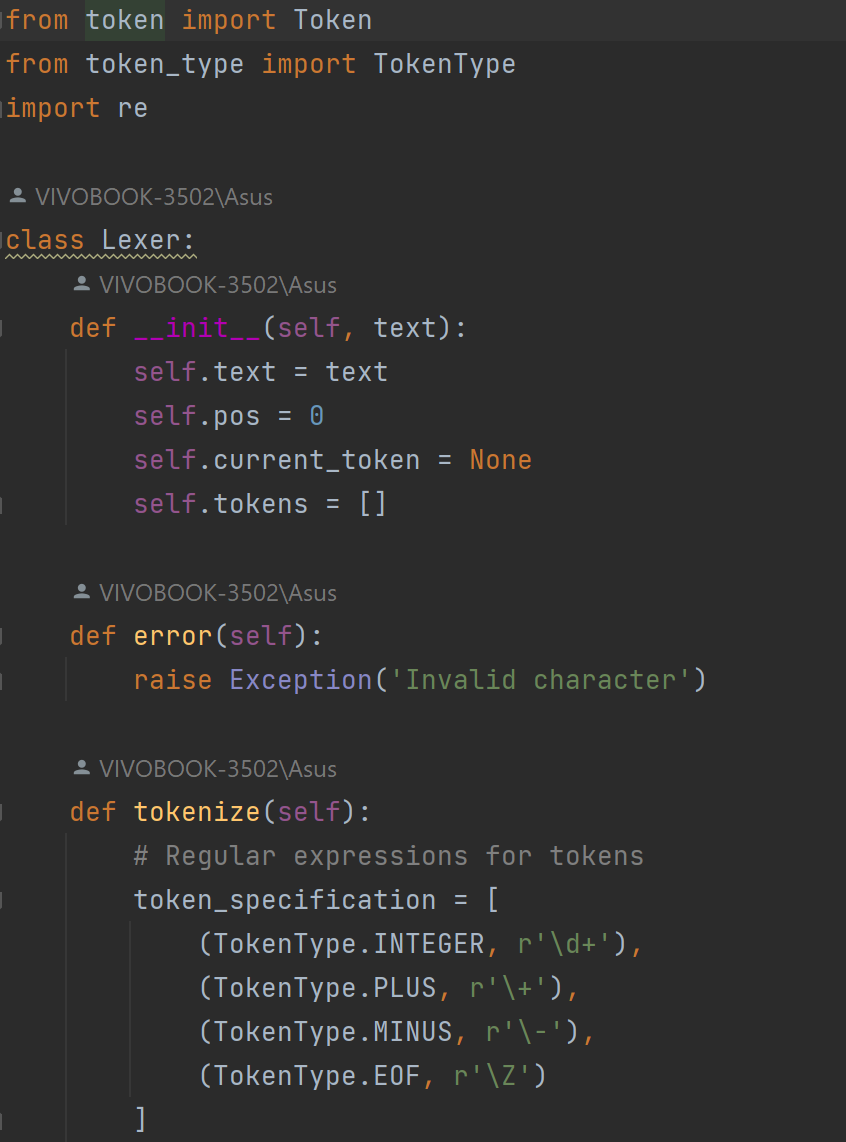
**Objectives:**

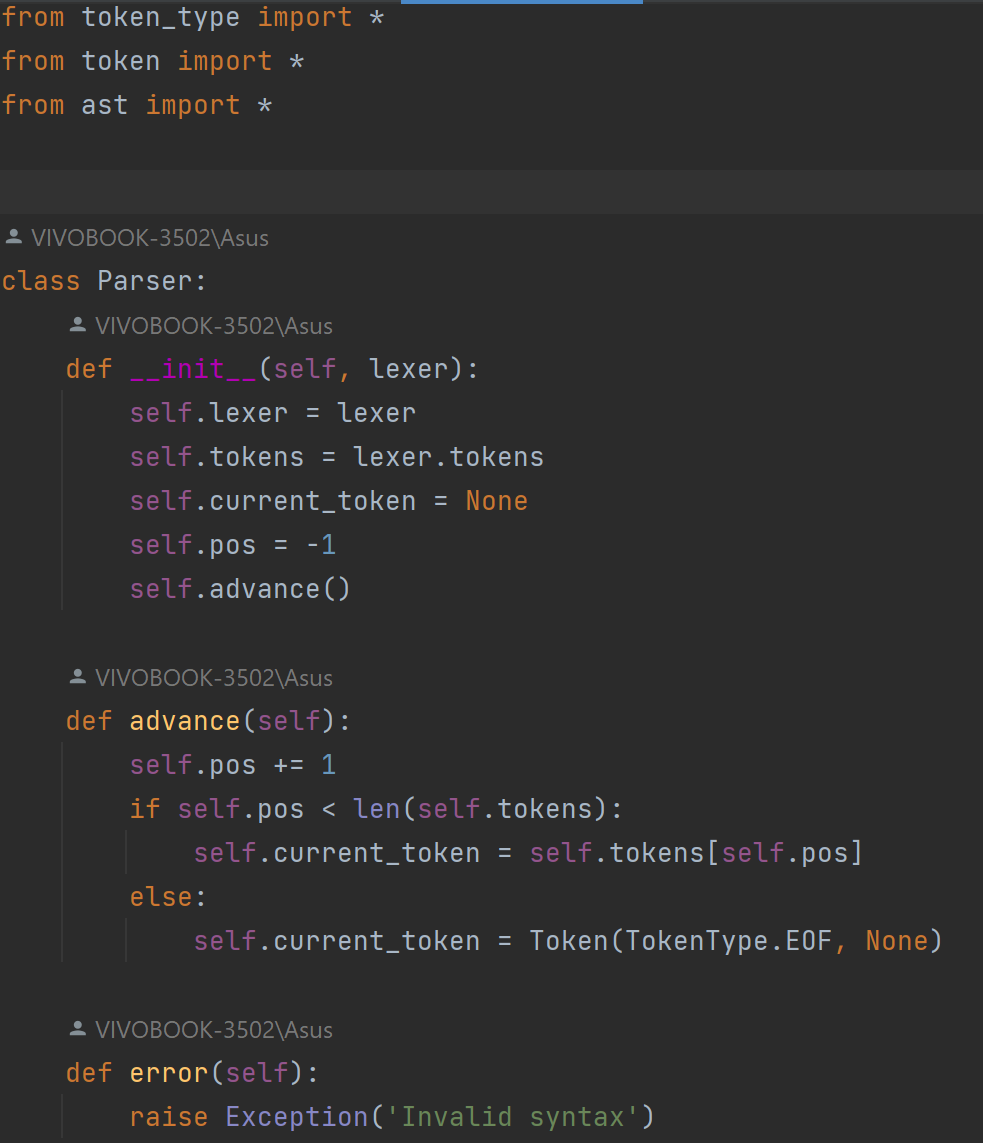
1. Get familiar with parsing, what it is and how it can be programmed [1].
2. Get familiar with the concept of AST [2].
3. In addition to what has been done in the 3rd lab work do the following:
   1. In case you didn't have a type that denotes the possible types of tokens you need to:
      1. Have 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.

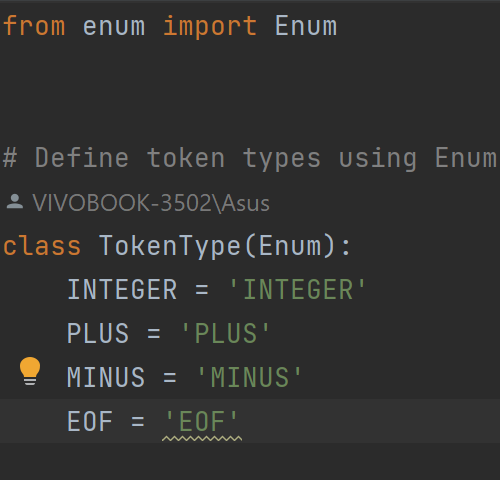
**Main.py**

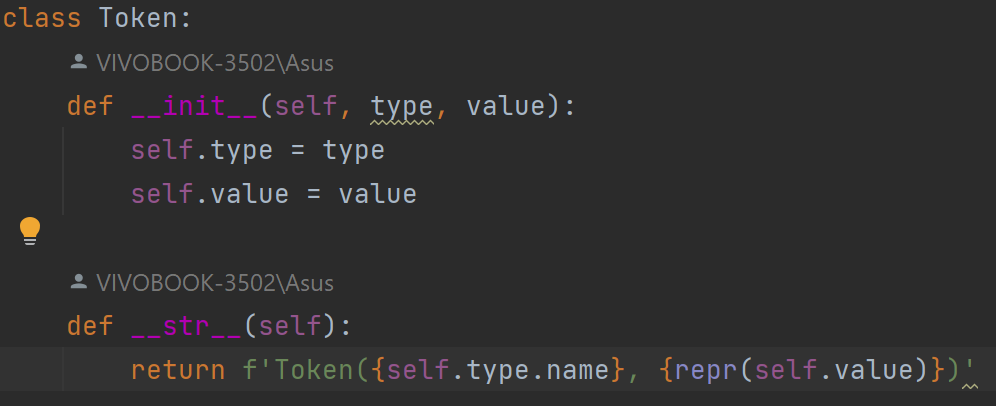
****

****

****

****

****

****

lexer.py: This module defines a Lexer class responsible for tokenizing input text. It uses regular expressions to match patterns corresponding to different token types such as integers, addition, subtraction, and end of file (EOF). The Lexer scans the input text, identifies tokens based on the specified patterns, and creates Token objects for each token found.

my\_parser.py: This module defines a Parser class responsible for parsing tokens generated by the Lexer and building an abstract syntax tree (AST). The Parser uses recursive descent parsing to handle arithmetic expressions consisting of addition and subtraction operations. It defines methods for advancing through tokens, handling errors, and parsing expressions into AST nodes.

ast.py: This module defines classes representing nodes of the abstract syntax tree (AST) for arithmetic expressions. It includes classes such as BinOp (binary operation) and Num (number) to represent different components of arithmetic expressions. The AST is constructed by the Parser as it parses the input tokens.

token.py: This module defines a Token class representing individual tokens generated by the Lexer. Each Token object contains a type (e.g., INTEGER, PLUS) and a corresponding value (e.g., the numeric value for INTEGER tokens).

token\_type.py: This module defines an enumeration (Enum) class TokenType containing different token types used by the Lexer and Parser. Each token type is represented as a unique string identifier.

Overall, this code implements a simple arithmetic expression parser that tokenizes input text, parses tokens into an abstract syntax tree (AST), and provides functionality to print the AST. It demonstrates the process of lexical analysis, parsing, and abstract syntax tree construction commonly used in compiler design and parsing algorithms.