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

Laboratory work 6:
Parser & Building an Abstract Syntax Tree

Elaborated: st. gr. FAF-223 Friptu Ludmila

Verified: asist. univ. Cretu Dumitru

THEORY

1. <u>Parsing</u>

Parsing, in the context of programming, is the process of analyzing a string of symbols, either in natural or computer languages, based on the rules of a formal grammar. The primary goal of parsing is to determine if the string can be generated by the grammar and to construct a parse tree that represents the structure of the string according to the grammar.

Parsing can be implemented through various algorithms, which are broadly categorized into two types: top-down and bottom-up parsers.

1. Top-Down Parsers - These parsers start by looking at the highest level of the parse tree and try to match the input string from the start, based on the productions of the grammar. Recursive descent parsers are a common example of top-down parsers, which involve writing a set of recursive functions where each function implements one of the non-terminals of the grammar.

2. Bottom-Up Parsers - These parsers start with the input symbols and attempt to construct the parse tree by working their way up to the start symbol of the grammar. Shift-reduce parsers like the LR and SLR parsers are typical examples, where the parser shifts symbols onto a stack and applies reductions based on the grammar's rules.

2. <u>Abstract Syntax Tree</u>

An Abstract Syntax Tree (AST) is a tree representation of the abstract syntactic structure of source code written in a programming language. Each node of the tree denotes a construct occurring in the source code. The purpose of the AST is to represent the syntactic structure of the code in a way that is easier for computer programs to manipulate.

ASTs are used in various aspects of a compiler, including:

- Syntax Analysis - Building the AST from source code.
- Semantic Analysis - Enhancing the AST with additional information that is used for checking correctness of the program (like type checking).
- Code Generation - Translating the AST into executable code.

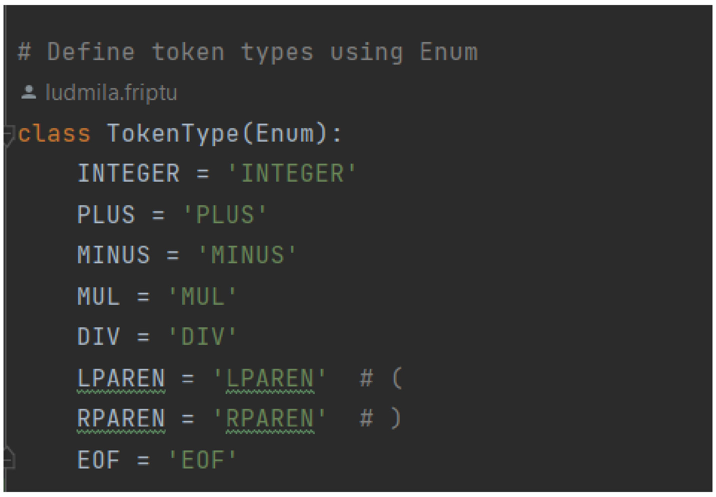
OBJECTIVES

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:
    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

I defined my TokenType class, which is used to represent a set of named constant values. Each member of the enum represents a specific type of token that the lexer can recognize in the input text. Here's what each member stands for:

- INTEGER: Represents a numeric integer value.
- PLUS: Represents the plus operator (+). It's used in arithmetic operations for addition.
- MINUS: Represents the minus operator (-), used for subtraction.
- MUL: Represents the multiplication operator (*).
- DIV: Represents the division operator (/).
- LPAREN: Represents a left parenthesis (().
- RPAREN: Represents a right parenthesis ()).
- EOF: Stands for End of File. In the context of parsing, it represents the end of the input or token stream. It's useful for signaling to the parser that there are no more tokens to be read, helping to gracefully exit parsing loops.



```
# Define token types using Enum
ludmila.friptu
class TokenType(Enum):
    INTEGER = 'INTEGER'
    PLUS = 'PLUS'
    MINUS = 'MINUS'
    MUL = 'MUL'
    DIV = 'DIV'
    LPAREN = 'LPAREN'   # (
    RPAREN = 'RPAREN'   # )
    EOF = 'EOF'
```

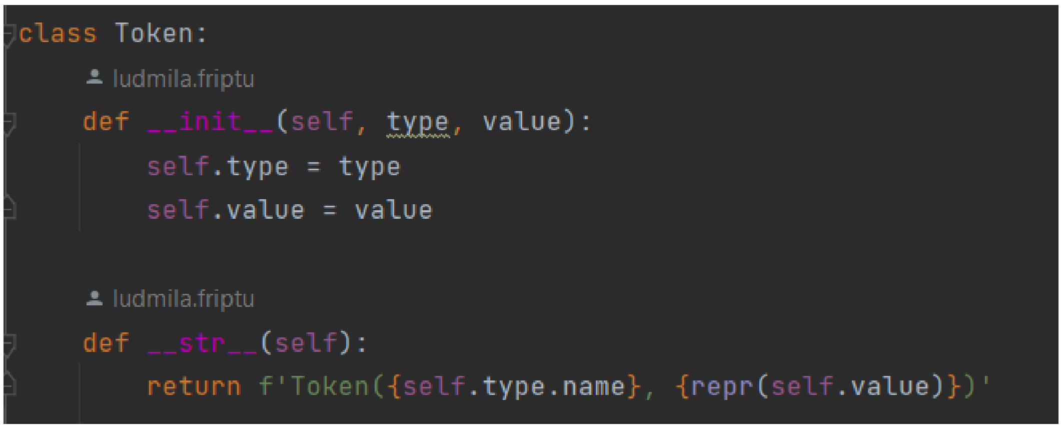
The Token class is a component of the lexer, which serves as the basic unit for representing tokens of the input string during the lexical analysis phase of parsing.

Constructor Method (__init__)

- *def __init__(self, type, value):* This is the constructor of the Token class. It initializes a new instance of the Token class whenever it is called.
    - type: A parameter that should be an instance of the TokenType enum. This parameter represents the type of the token (e.g., INTEGER, PLUS, MINUS, etc.).
    - value: The actual value of the token. For example, if the token type is INTEGER, the value might be a numeric value like 42. If the token type is PLUS, the value might simply be the + character.

String Representation Method (__str__)

- *def __str__(self):* This method provides a human-readable string representation of the Token instance.
    - *return f'Token({self.type.name}, {repr(self.value)})'*: This line formats the string output for the token. The self.type.name accesses the name property of the TokenType enum, which gives a readable name for the token type (e.g., "INTEGER"). The repr(self.value) function is used to get a string representation of the value that includes any necessary Python syntax (like quotes around strings).

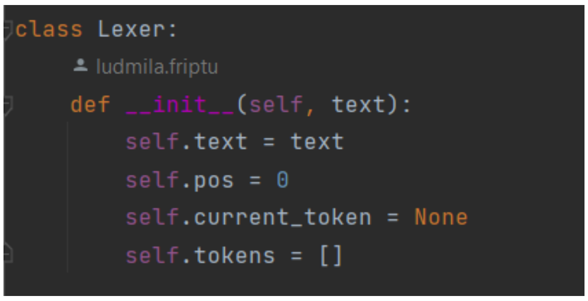


```
class Token:
    ludmila.friptu
    def __init__(self, type, value):
        self.type = type
        self.value = value

    ludmila.friptu
    def __str__(self):
        return f'Token({self.type.name}, {repr(self.value)})'
```

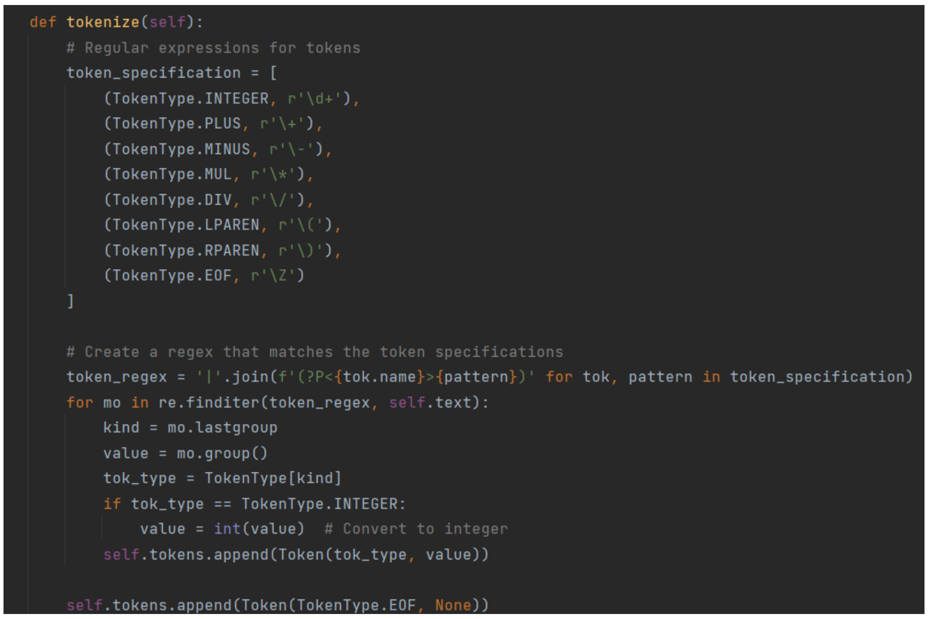
The Lexer class is crucial for the initial phase of parsing—lexical analysis—which involves scanning the input text to convert it into a series of tokens.
- def __init__(self, text): The constructor for the Lexer class. It initializes a new instance with the input text that needs to be tokenized.
  - self.text: Stores the input text to be tokenized.

- self.pos: Keeps track of the current position in the text. This is used to traverse through the string.
- self.current_token: Initially set to None, this attribute will later hold the current token being processed.
- self.tokens: A list to store all the tokens generated from the input text.

- def error(self): A method that raises an exception when an unexpected character is encountered.
- def tokenize(self): This method converts the input text into tokens based on predefined patterns.
  - token_specification: A list of tuples where each tuple contains a TokenType and a regular expression pattern that defines how tokens of this type can be recognized in the text.
  - token_regex: This string is a combination of all the patterns in token_specification formatted to be used with Python's re module for matching.
  - Loop through Matches: re.finditer is used to find all non-overlapping matches of token_regex in self.text. For each match:
    - kind: The name of the group that matched, which corresponds to a token type.
    - value: The actual text that matched the token pattern.
    - tok_type: The TokenType corresponding to kind.
    - Conversion: If the token type is INTEGER, the string is converted to an integer.
    - Appending Token: A new Token instance is created with tok_type and value and added to the self.tokens list.
  - End of File Token: After all tokens are extracted, an EOF token is appended to signify the end of the token stream. This is important for the parser to know when it has processed all input.

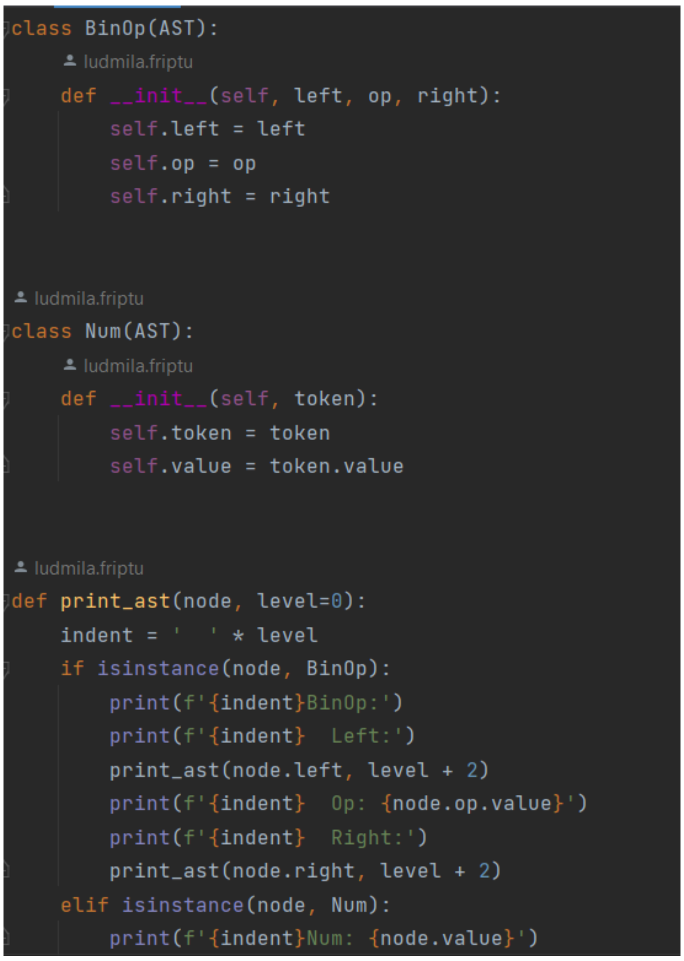


```python
class Lexer:
    ludmila.friptu
    def __init__(self, text):
        self.text = text
        self.pos = 0
        self.current_token = None
        self.tokens = []
```



```python
def tokenize(self):
    # Regular expressions for tokens
    token_specification = [
        (TokenType.INTEGER, r'\d+'),
        (TokenType.PLUS, r'\+'),
        (TokenType.MINUS, r'\-'),
        (TokenType.MUL, r'\*'),
        (TokenType.DIV, r'\/'),
        (TokenType.LPAREN, r'\('),
        (TokenType.RPAREN, r'\)'),
        (TokenType.EOF, r'\Z')
    ]

    # Create a regex that matches the token specifications
    token_regex = '|'.join(f'(?P<{tok.name}>{pattern})' for tok, pattern in token_specification)
    for mo in re.finditer(token_regex, self.text):
        kind = mo.lastgroup
        value = mo.group()
        tok_type = TokenType[kind]
        if tok_type == TokenType.INTEGER:
            value = int(value)  # Convert to integer
        self.tokens.append(Token(tok_type, value))

    self.tokens.append(Token(TokenType.EOF, None))
```

Next, I defined the AST class, a base class for all nodes in the AST.
- class BinOp(AST): This class represents a binary operation in the AST.
  - __init__(self, left, op, right): The constructor takes three parameters:
    - left: The left operand of the binary operation, which is itself an AST node.
    - op: The operator, typically a token representing the operation (e.g., `+`, `-`, `*`, `/`).
    - right: The right operand of the binary operation, which is also an AST node.
- class Num(AST): This class represents a numeric literal in the AST.
  - __init__(self, token): The constructor accepts a token that contains information about the number.
    - self.token: The token corresponding to the number.
    - self.value: Extracts the numeric value from the token and stores it. This is used in the evaluation or any processing that requires the actual number.
- def print_ast(node, level=0): This function is used to print a visual representation of the AST.
  - node: The current node to be printed.
  - level: The current level in the tree, used to determine the amount of indentation.
  - indent: A string of spaces that increases with the level of depth in the tree, used to visually represent the hierarchy of nodes.
    - BinOp: If it's a binary operation node, it prints the operator and recursively calls print_ast for the left and right operands, increasing the indentation level.
    - Num: If it's a number node, it prints the numeric value.



```python
class BinOp(AST):
    ludmila.friptu
    def __init__(self, left, op, right):
        self.left = left
        self.op = op
        self.right = right


ludmila.friptu
class Num(AST):
    ludmila.friptu
    def __init__(self, token):
        self.token = token
        self.value = token.value


ludmila.friptu
def print_ast(node, level=0):
    indent = '  ' * level
    if isinstance(node, BinOp):
        print(f'{indent}BinOp:')
        print(f'{indent}  Left:')
        print_ast(node.left, level + 2)
        print(f'{indent}  Op: {node.op.value}')
        print(f'{indent}  Right:')
        print_ast(node.right, level + 2)
    elif isinstance(node, Num):
        print(f'{indent}Num: {node.value}')
```

Finally I defined the Parser class. The class reads tokens produced by the lexer and builds an abstract syntax tree that represents the grammatical structure of the input based on the rules of the language.

- self.lexer: Stores the lexer instance.
- self.tokens: Retrieves the list of tokens from the lexer.
- self.current_token: Used to keep track of the current token being processed.
- self.pos: An index that tracks the position in the token list.
- self.advance(): An initial call to this method sets up self.current_token to the first token.

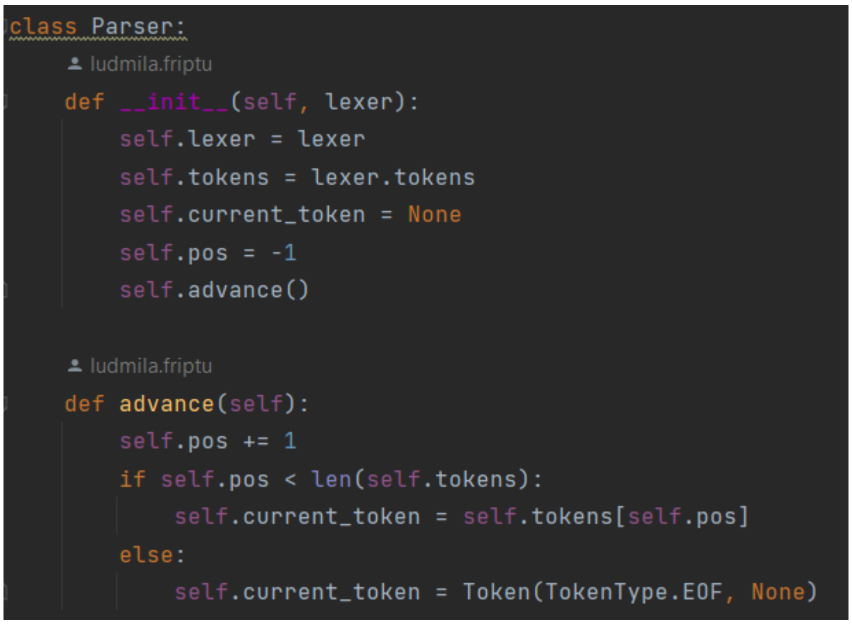
- def advance(self): This method increments self.pos and updates self.current_token with the next token in the list or sets it to EOF if the end of the token list is reached.
- def error(self): Raises an exception with a message indicating invalid syntax. This is typically called when the parser encounters an unexpected token sequence.

Parsing Methods

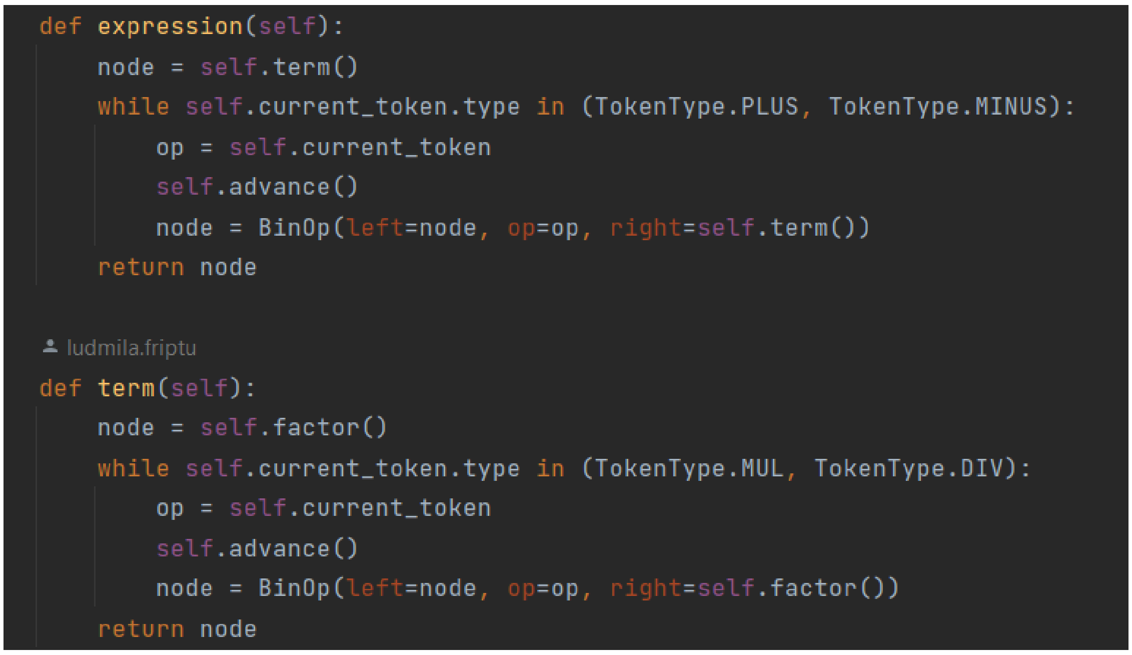
- def parse(self): This method starts the parsing process by calling the expression method and returns the root of the AST.
- def expression(self): Handles expressions involving addition and subtraction.

Recursive Building: If a `+` or `-` operator is found after a term, it constructs a BinOp node with the current term as the left child, the operator as the node's operator, and recursively parses the next term as the right child.

- def term(self): Similar to expression, but handles multiplication and division. It ensures multiplication and division have higher precedence than addition and subtraction.
- def factor(self): Parses the smallest units of an expression, which are integers and nested expressions:
  - Integers: If the current token is an integer, it creates a Num node.
  - Nested Expressions: If the current token is a left parenthesis (`(`), it recursively parses the enclosed expression until it finds a matching right parenthesis (`)`), handling the expression inside the parentheses with higher precedence.

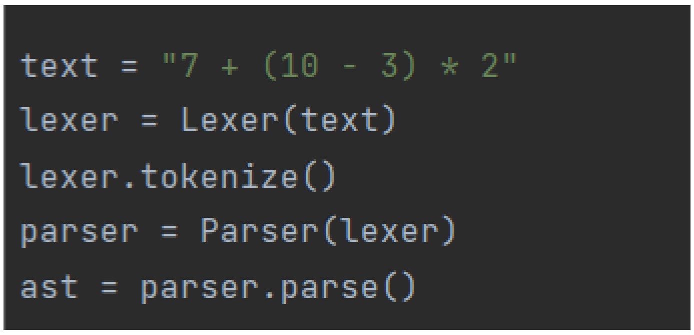


```
class Parser:
    ludmila.friptu
    def __init__(self, lexer):
        self.lexer = lexer
        self.tokens = lexer.tokens
        self.current_token = None
        self.pos = -1
        self.advance()

    ludmila.friptu
    def advance(self):
        self.pos += 1
        if self.pos < len(self.tokens):
            self.current_token = self.tokens[self.pos]
        else:
            self.current_token = Token(TokenType.EOF, None)
```



```python
def expression(self):
    node = self.term()
    while self.current_token.type in (TokenType.PLUS, TokenType.MINUS):
        op = self.current_token
        self.advance()
        node = BinOp(left=node, op=op, right=self.term())
    return node

def term(self):
    node = self.factor()
    while self.current_token.type in (TokenType.MUL, TokenType.DIV):
        op = self.current_token
        self.advance()
        node = BinOp(left=node, op=op, right=self.factor())
    return node
```

RESULTS



```python
text = "7 + (10 - 3) * 2"
lexer = Lexer(text)
lexer.tokenize()
parser = Parser(lexer)
ast = parser.parse()
```

```
Tokens:
Token(INTEGER, 7)
Token(PLUS, '+')
Token(LPAREN, '(')
Token(INTEGER, 10)
Token(MINUS, '-')
Token(INTEGER, 3)
Token(RPAREN, ')')
Token(MUL, '*')
Token(INTEGER, 2)
Token(EOF, '')
Token(EOF, None)
```

```
AST:
BinOp:
  Left:
    Num: 7
  Op: +
  Right:
    BinOp:
      Left:
        BinOp:
          Left:
            Num: 10
          Op: -
          Right:
            Num: 3
      Op: *
      Right:
        Num: 2
```

CONCLUSION

In this laboratory work, I explored the processes of lexical analysis, parsing, and the construction of Abstract Syntax Trees through the implementation of a simple arithmetic expression evaluator. The seamless integration of the Lexer, Parser, and AST node components demonstrated the mechanisms involved in interpreting arithmetic expressions.
The Lexer played a foundational role, effectively tokenizing the input string into a structured series of meaningful tokens based on predefined regular expressions. This conversion from raw text into a manageable series of tokens set the stage for the more complex phase of syntactic analysis. Following lexical analysis, the Parser took the lead in constructing an AST by adhering to the grammatical rules of arithmetic operations. It ensured the correct application of operator precedence and associativity through recursive methods that built the tree structure step-by-step.

Through this exercise, I gained a deeper understanding of how parsers and ASTs function at the core of programming environments, offering a clear view of the potential and efficiency these structures bring to the field of computer science.