Ministerul Educaţiei, Culturii şi Cercetării al Republicii Moldova
Universitatea Tehnică a Moldovei
Departamentul Ingineria Software și Automatică

REPORT

Laboratory work 6
Course: Formal Languages & Finite Automata
Theme: Parser & Building an AST

Elaborated: Latcovschi Cătălin,
FAF-221

Verified: asist.univ.
Dumitru Crețu

Chișinău 2024

Overview

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.

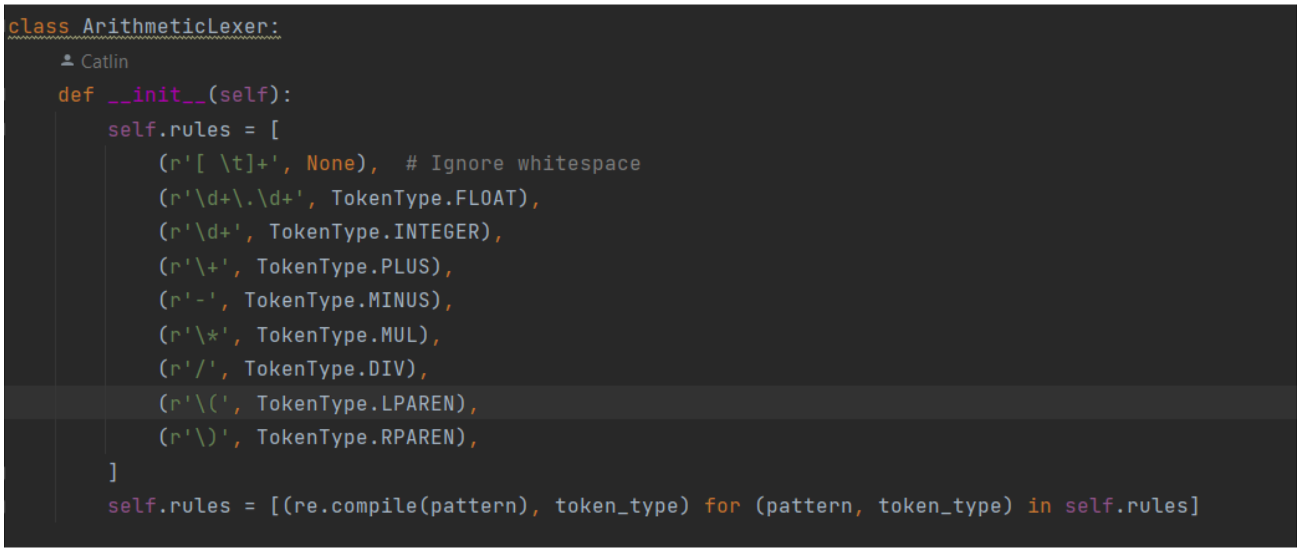
Similarly to a parse tree, in order to represent the structure of an input text one could create an Abstract Syntax Tree (AST). This is a data structure that is organized hierarchically in abstraction layers that represent the constructs or entities that form up the initial text. These can come in handy also in the analysis of programs or some processes involved in compilation.

Objectives

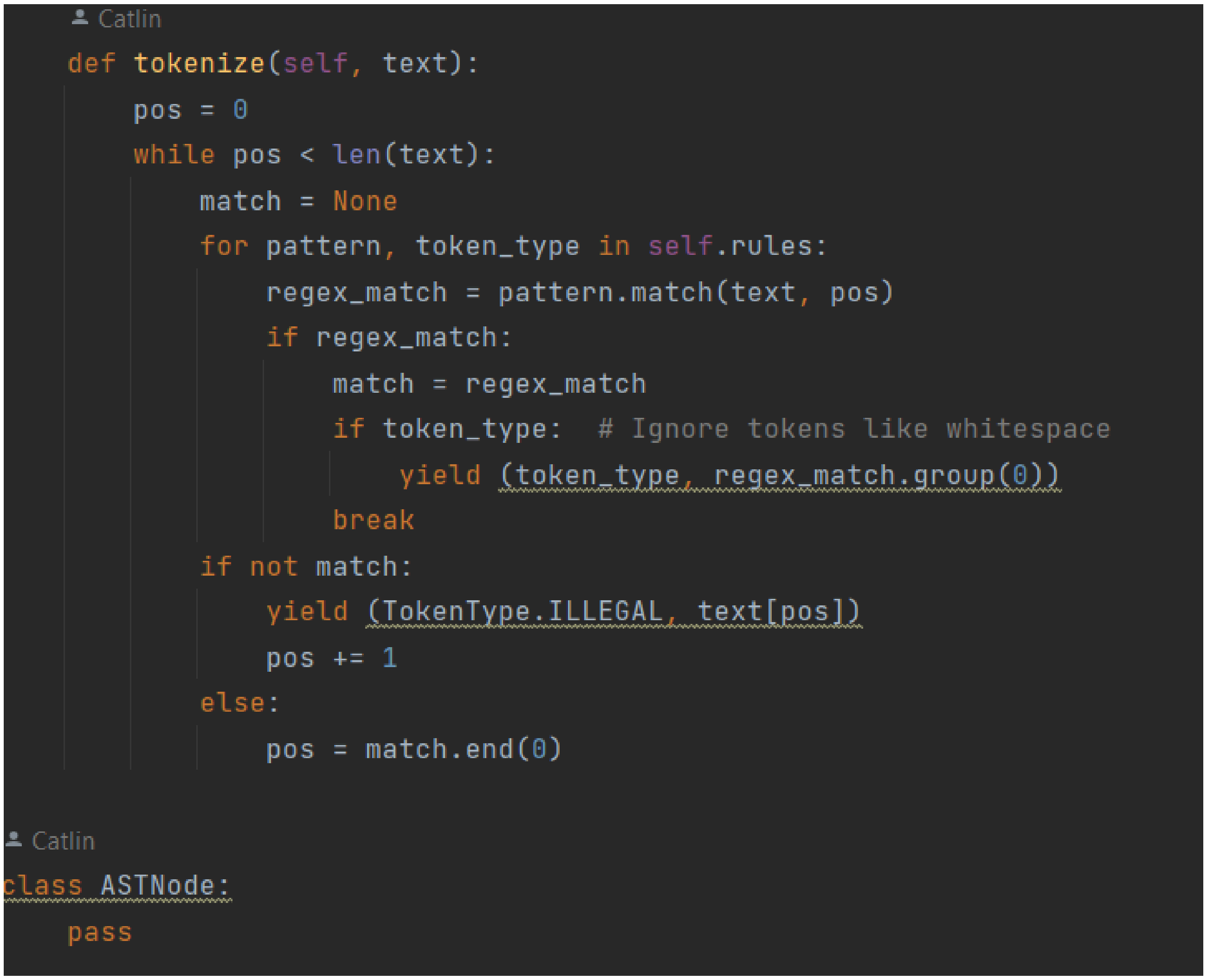
1. Get familiar with parsing, what it is and how it can be programmed [1].
2. Get familiar with the concept of AST [2].
3. In addition to what has been done in the 3rd lab work do the following:
4. In case you didn't have a type that denotes the possible types of tokens you need to:
5. Have a type TokenType (like an enum) that can be used in the lexical analysis to categorize the tokens.
6. Please use regular expressions to identify the type of the token.
7. Implement the necessary data structures for an AST that could be used for the text you have processed in the 3rd lab work.
8. Implement a simple parser program that could extract the syntactic information from the input text.

Implementation

1. **ArithmeticLexer class:**



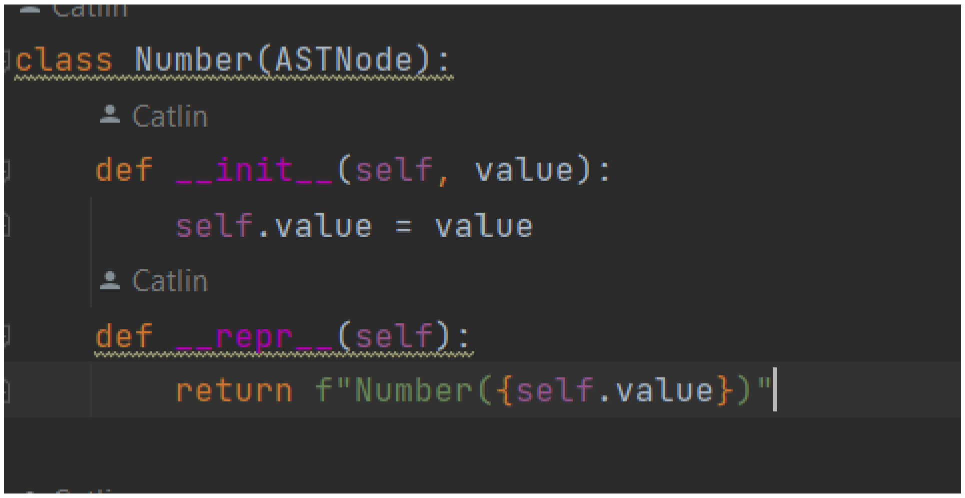
```
class ArithmeticLexer:
    Catlin
    def __init__(self):
        self.rules = [
            (r'[ \t]+', None),  # Ignore whitespace
            (r'\d+\.\d+', TokenType.FLOAT),
            (r'\d+', TokenType.INTEGER),
            (r'\+', TokenType.PLUS),
            (r'-', TokenType.MINUS),
            (r'\*', TokenType.MUL),
            (r'/', TokenType.DIV),
            (r'\(', TokenType.LPAREN),
            (r'\)', TokenType.RPAREN),
        ]
        self.rules = [(re.compile(pattern), token_type) for (pattern, token_type) in self.rules]
```



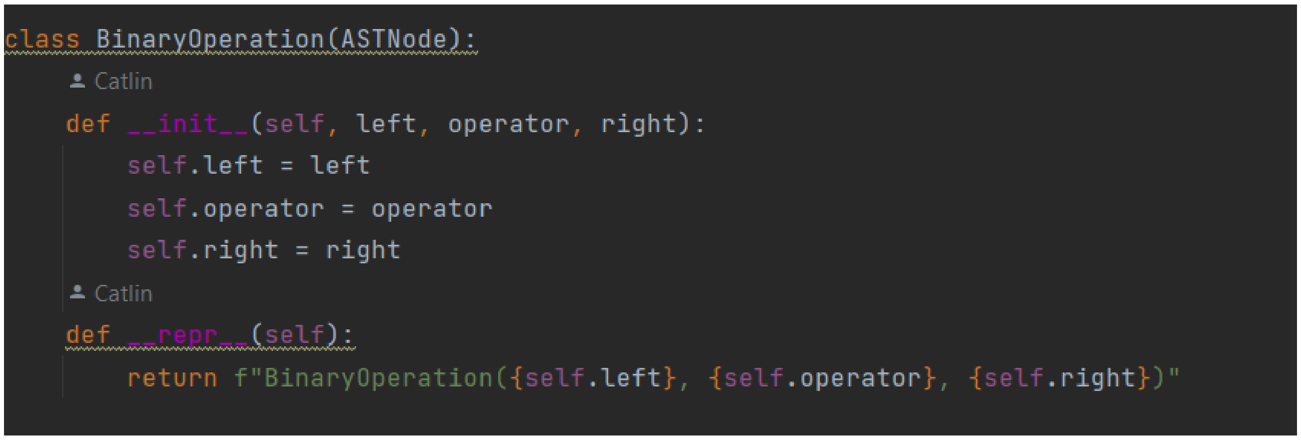
```
    Catlin
    def tokenize(self, text):
        pos = 0
        while pos < len(text):
            match = None
            for pattern, token_type in self.rules:
                regex_match = pattern.match(text, pos)
                if regex_match:
                    match = regex_match
                    if token_type:  # Ignore tokens like whitespace
                        yield (token_type, regex_match.group(0))
                    break
            if not match:
                yield (TokenType.ILLEGAL, text[pos])
                pos += 1
            else:
                pos = match.end(0)

Catlin
class ASTNode:
    pass
```

The `TokenType` class defines an enumeration for token types in an arithmetic lexer, including FLOAT, INTEGER, PLUS, MINUS, MUL, DIV, LPAREN, RPAREN, and ILLEGAL. The `ArithmeticLexer` class initializes with rules for tokenizing arithmetic expressions, utilizing regular expressions to match patterns like numbers, operators, and parentheses. The `tokenize` method iterates over the input text, matching it against the defined rules to generate tokens. If no match is found, an ILLEGAL token is yielded. This lexer serves as a foundational component for parsing arithmetic expressions. Additionally, the `ASTNode` class is a placeholder for representing nodes in an abstract syntax tree, which would be constructed during parsing.

2. **Number and BinaryOperation classes:**



```
class Number(ASTNode):
    Catlin
    def __init__(self, value):
        self.value = value
    Catlin
    def __repr__(self):
        return f"Number({self.value})"
```

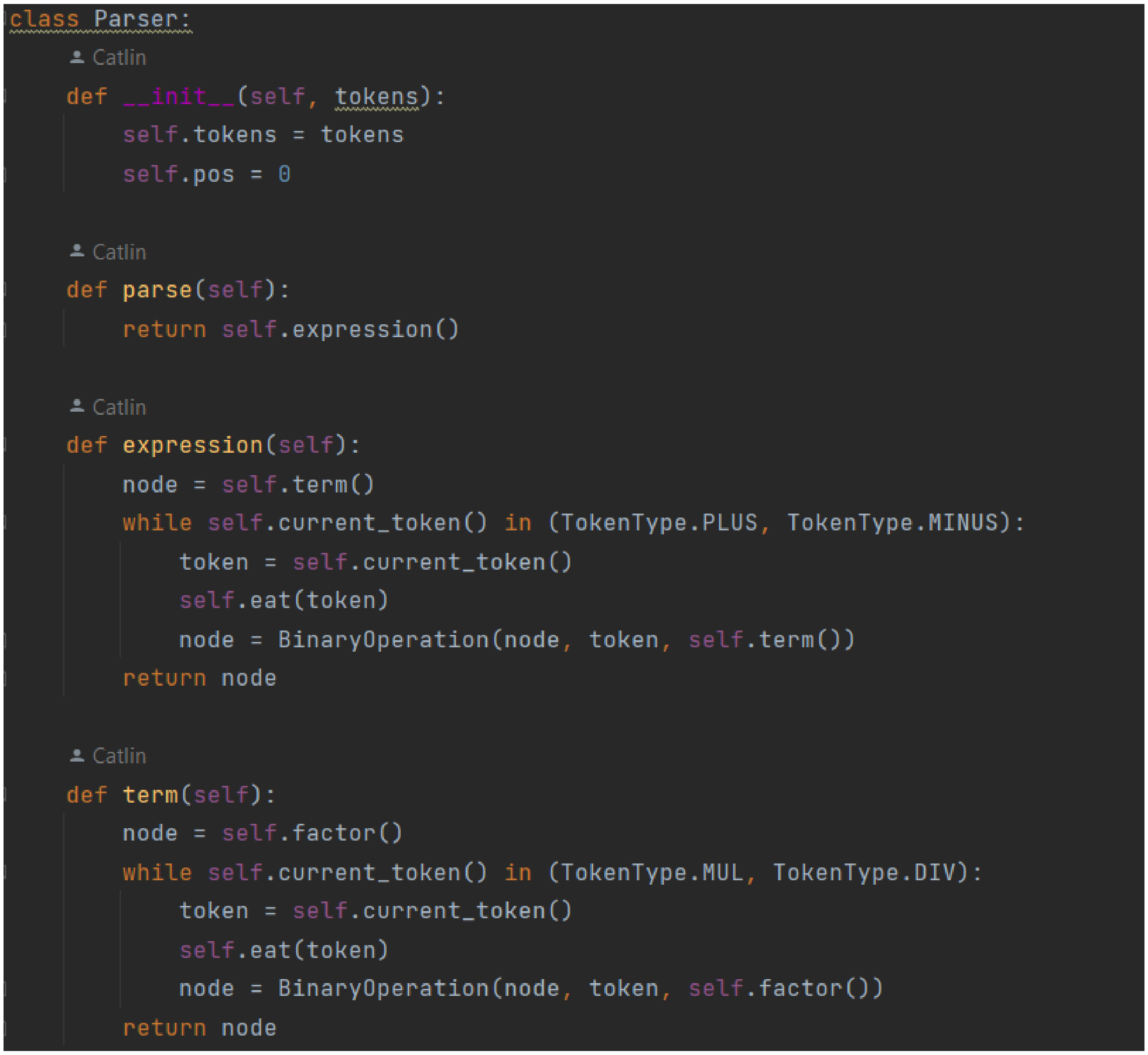


```
class BinaryOperation(ASTNode):
    Catlin
    def __init__(self, left, operator, right):
        self.left = left
        self.operator = operator
        self.right = right
    Catlin
    def __repr__(self):
        return f"BinaryOperation({self.left}, {self.operator}, {self.right})"
```

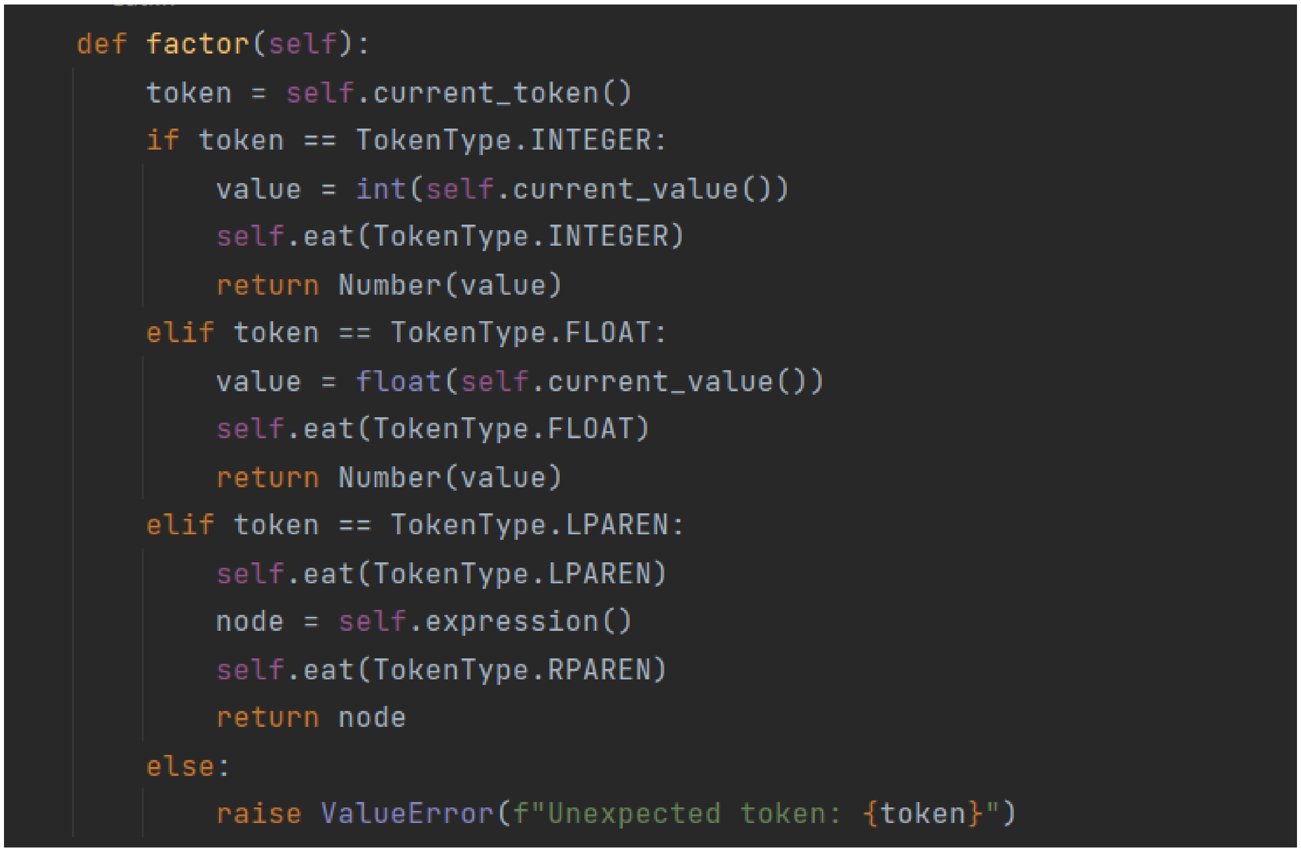
The `Number` class, inheriting from `ASTNode`, represents a numeric value in an abstract syntax tree (AST). Initialized with a numeric `value`, it provides a `__repr__` method to generate a string representation of the node with the format "Number(value)".

The `BinaryOperation` class, also inheriting from `ASTNode`, models a binary operation in the AST. It includes attributes `left` (representing the left operand), `operator` (the operation being performed), and `right` (the right operand). The `__repr__` method generates a string representation of the node in the format "BinaryOperation(left, operator, right)". These classes facilitate the construction and representation of AST nodes for arithmetic expressions.

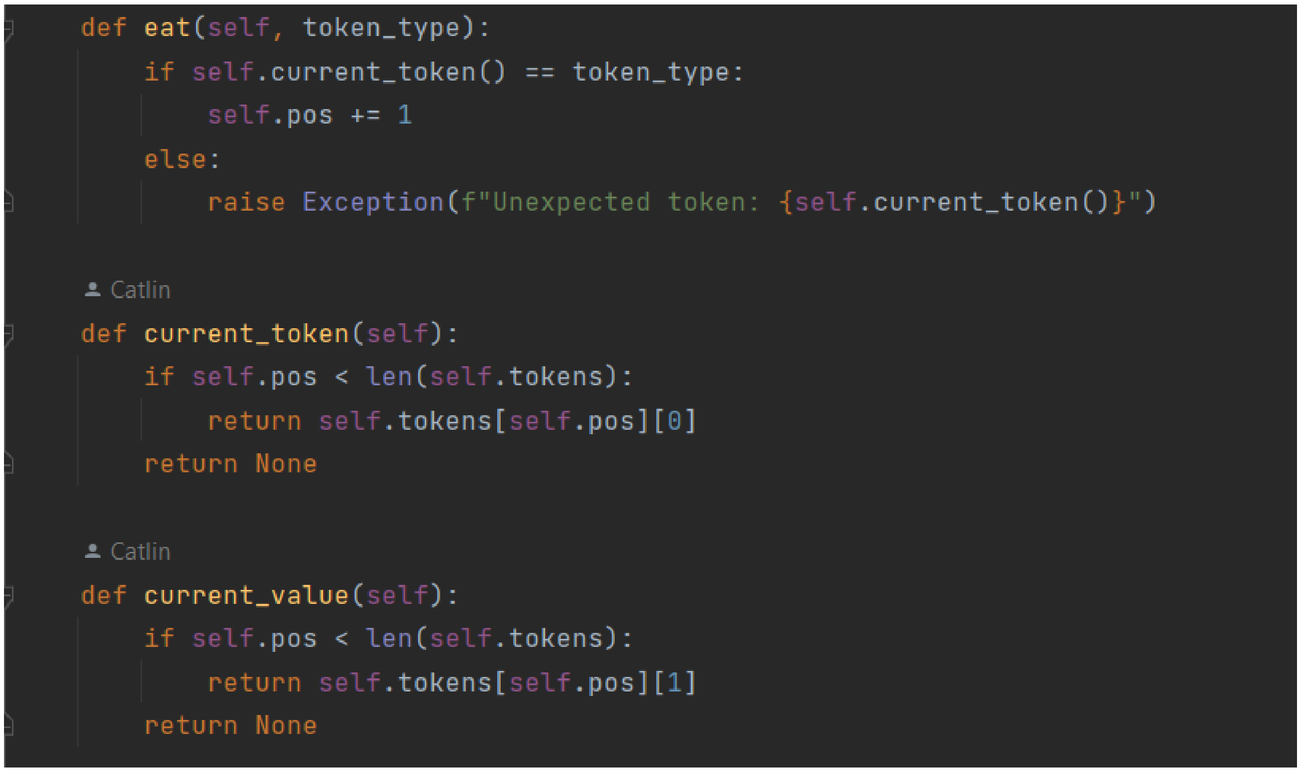
3. **Parser class:**



```
class Parser:
    Catlin
    def __init__(self, tokens):
        self.tokens = tokens
        self.pos = 0

    Catlin
    def parse(self):
        return self.expression()

    Catlin
    def expression(self):
        node = self.term()
        while self.current_token() in (TokenType.PLUS, TokenType.MINUS):
            token = self.current_token()
            self.eat(token)
            node = BinaryOperation(node, token, self.term())
        return node

    Catlin
    def term(self):
        node = self.factor()
        while self.current_token() in (TokenType.MUL, TokenType.DIV):
            token = self.current_token()
            self.eat(token)
            node = BinaryOperation(node, token, self.factor())
        return node
```



```python
    def factor(self):
        token = self.current_token()
        if token == TokenType.INTEGER:
            value = int(self.current_value())
            self.eat(TokenType.INTEGER)
            return Number(value)
        elif token == TokenType.FLOAT:
            value = float(self.current_value())
            self.eat(TokenType.FLOAT)
            return Number(value)
        elif token == TokenType.LPAREN:
            self.eat(TokenType.LPAREN)
            node = self.expression()
            self.eat(TokenType.RPAREN)
            return node
        else:
            raise ValueError(f"Unexpected token: {token}")
```



```python
    def eat(self, token_type):
        if self.current_token() == token_type:
            self.pos += 1
        else:
            raise Exception(f"Unexpected token: {self.current_token()}")

    Catlin
    def current_token(self):
        if self.pos < len(self.tokens):
            return self.tokens[self.pos][0]
        return None

    Catlin
    def current_value(self):
        if self.pos < len(self.tokens):
            return self.tokens[self.pos][1]
        return None
```

The `Parser` class is designed to parse arithmetic expressions represented as tokens. Initialized with a list of tokens, it tracks the current position within the token stream.

The `parse` method initiates the parsing process by calling the `expression` method, which represents the highest precedence operation.

The `expression` method constructs the AST nodes for addition and subtraction operations by recursively calling `term`. It iterates through tokens, identifying addition or subtraction operators and creating corresponding `BinaryOperation` nodes.

The `term` method handles multiplication and division operations similarly to `expression`, iterating through tokens and constructing `BinaryOperation` nodes accordingly.

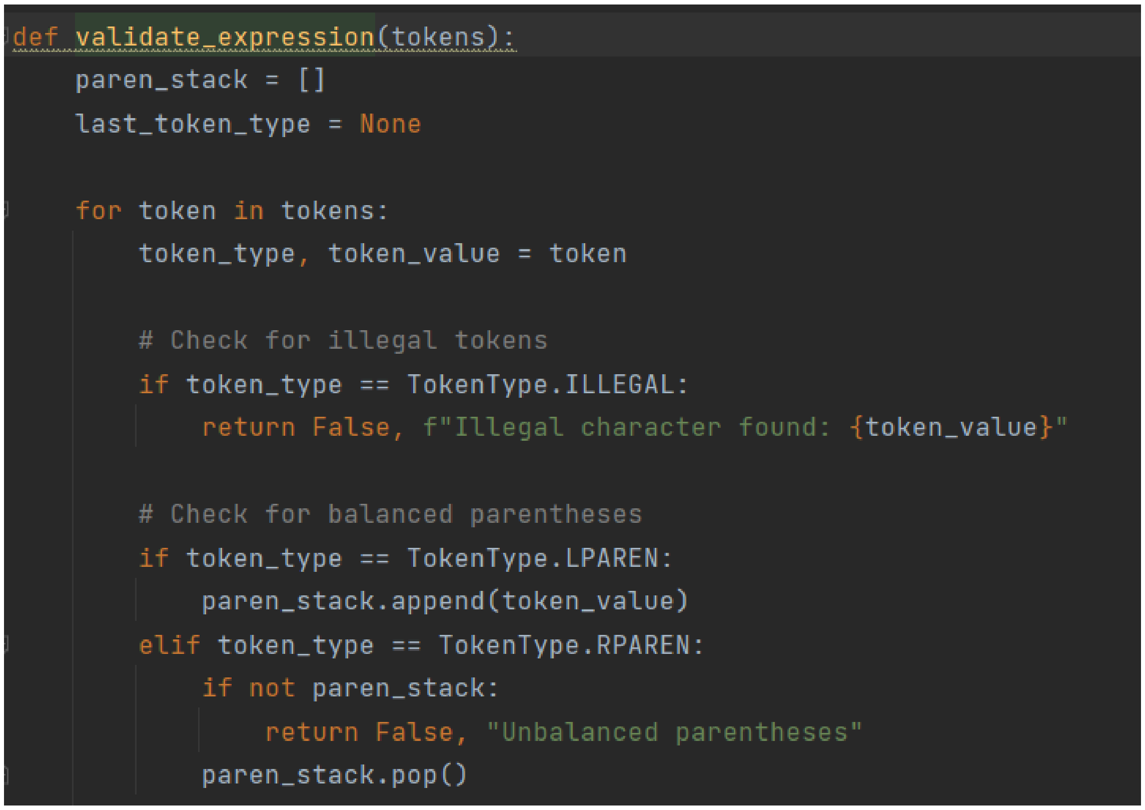
The `factor` method deals with individual factors in an expression. It recognizes integer and float literals, parentheses for grouping, and raises exceptions for unexpected tokens.

The `eat` method consumes tokens, advancing the position if the current token matches the expected type, or raising an exception otherwise.

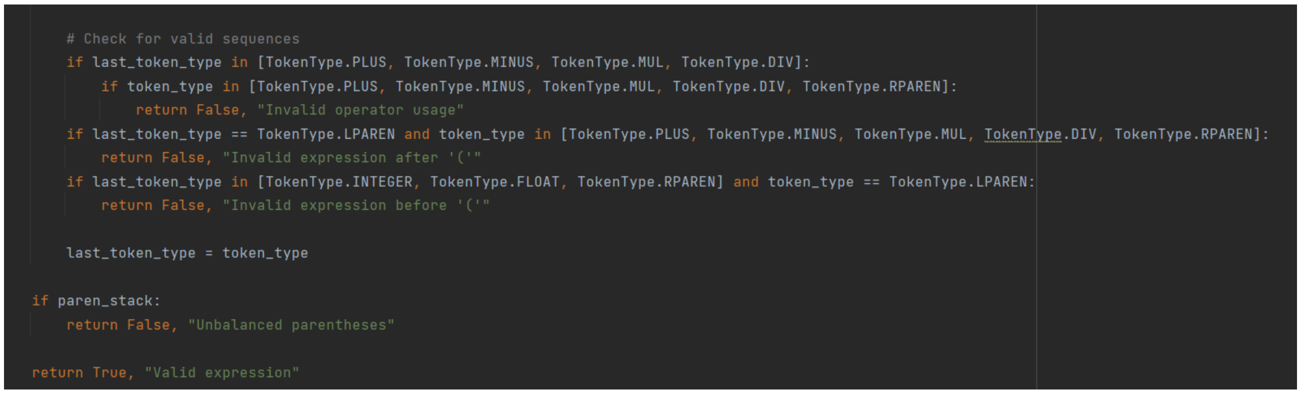
The `current_token` and `current_value` methods provide access to the current token type and value, respectively, ensuring safe traversal of the token stream.

Overall, the `Parser` class facilitates the conversion of tokenized arithmetic expressions into an abstract syntax tree (AST), enabling further analysis and evaluation.

4. **validate_expression method:**



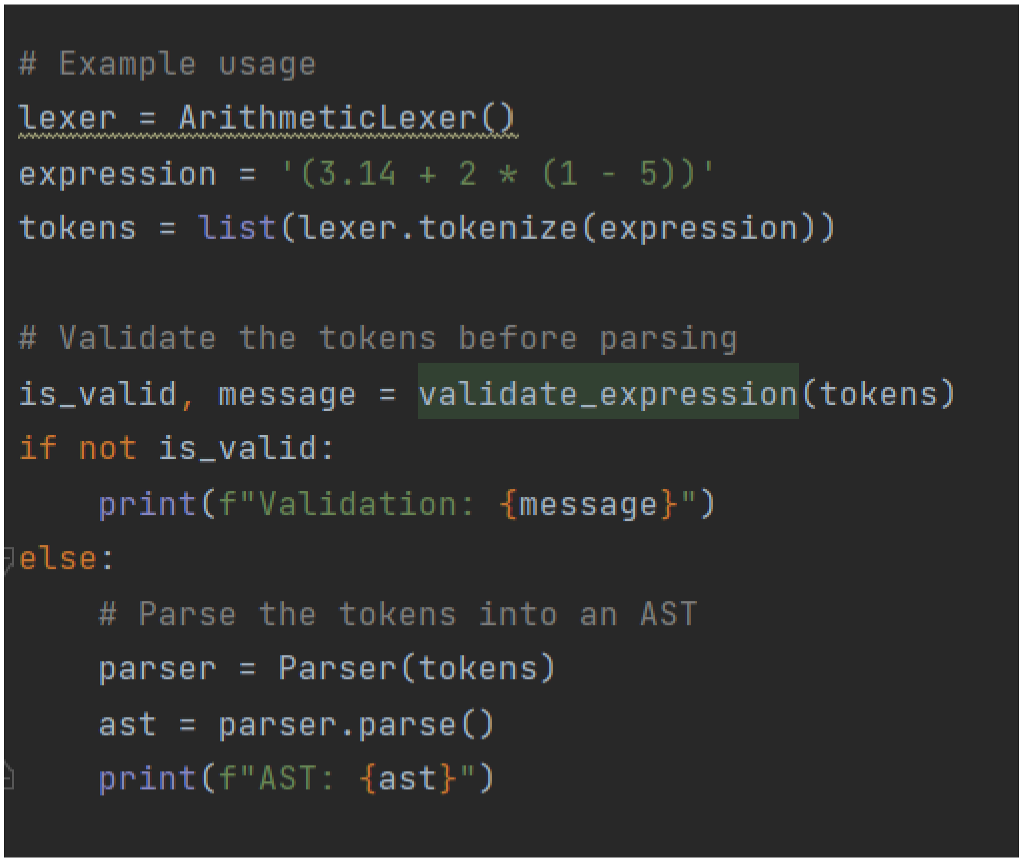
```
def validate_expression(tokens):
    paren_stack = []
    last_token_type = None

    for token in tokens:
        token_type, token_value = token

        # Check for illegal tokens
        if token_type == TokenType.ILLEGAL:
            return False, f"Illegal character found: {token_value}"

        # Check for balanced parentheses
        if token_type == TokenType.LPAREN:
            paren_stack.append(token_value)
        elif token_type == TokenType.RPAREN:
            if not paren_stack:
                return False, "Unbalanced parentheses"
            paren_stack.pop()
```



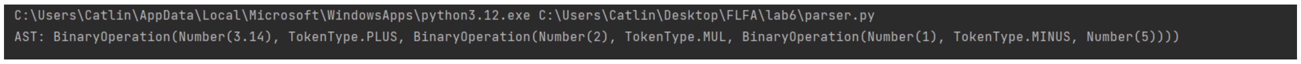
```python
        # Check for valid sequences
        if last_token_type in [TokenType.PLUS, TokenType.MINUS, TokenType.MUL, TokenType.DIV]:
            if token_type in [TokenType.PLUS, TokenType.MINUS, TokenType.MUL, TokenType.DIV, TokenType.RPAREN]:
                return False, "Invalid operator usage"
        if last_token_type == TokenType.LPAREN and token_type in [TokenType.PLUS, TokenType.MINUS, TokenType.MUL, TokenType.DIV, TokenType.RPAREN]:
            return False, "Invalid expression after '('"
        if last_token_type in [TokenType.INTEGER, TokenType.FLOAT, TokenType.RPAREN] and token_type == TokenType.LPAREN:
            return False, "Invalid expression before '('"

        last_token_type = token_type

    if paren_stack:
        return False, "Unbalanced parentheses"

    return True, "Valid expression"
```

The `validate_expression` method analyzes a list of tokens representing an arithmetic expression to ensure its syntactic validity. It maintains a stack to track balanced parentheses and checks for illegal tokens. It iterates through tokens, examining sequences and detecting potential issues such as unbalanced parentheses or invalid operator usage. Additionally, it verifies the correctness of expressions before and after parentheses. If any inconsistency is detected, it returns a failure status along with a corresponding error message. Otherwise, it confirms the expression's validity. This method serves as a crucial step in preprocessing expressions before parsing, ensuring that only well-formed expressions proceed to further analysis.

Results



```python
# Example usage
lexer = ArithmeticLexer()
expression = '(3.14 + 2 * (1 - 5))'
tokens = list(lexer.tokenize(expression))

# Validate the tokens before parsing
is_valid, message = validate_expression(tokens)
if not is_valid:
    print(f"Validation: {message}")
else:
    # Parse the tokens into an AST
    parser = Parser(tokens)
    ast = parser.parse()
    print(f"AST: {ast}")
```



```
C:\Users\Catlin\AppData\Local\Microsoft\WindowsApps\python3.12.exe C:\Users\Catlin\Desktop\FLFA\lab6\parser.py
AST: BinaryOperation(Number(3.14), TokenType.PLUS, BinaryOperation(Number(2), TokenType.MUL, BinaryOperation(Number(1), TokenType.MINUS, Number(5))))
```

Conclusions

In conclusion, the laboratory work on parsing and building an Abstract Syntax Tree (AST) provided a comprehensive understanding of syntactical analysis and representation in the context of formal languages and finite automata. The implementation encompassed various components, including the ArithmeticLexer for tokenizing arithmetic expressions, the ASTNode hierarchy for representing nodes in an AST, the Parser for parsing tokenized expressions into AST structures, and the validate_expression method for syntactic validation.

Through the implementation of these components, fundamental concepts such as tokenization, parsing, and AST construction were thoroughly explored and applied. Regular expressions were utilized for efficient tokenization, ensuring accurate categorization of tokens. The Parser class efficiently constructed AST nodes, facilitating the hierarchical representation of arithmetic expressions for further analysis or evaluation.

The validation process provided by the validate_expression method ensured the syntactic integrity of input expressions, enhancing the robustness of subsequent parsing operations. By identifying and addressing potential syntax errors, the method contributed to the overall reliability and accuracy of the parsing process.

Overall, the laboratory work enhanced problem-solving skills and deepened understanding of formal language theory, laying a solid foundation for further exploration in the field of syntactical analysis and language processing.