Topic: Parser & Building an Abstract Syntax Tree

Course: Formal Languages & Finite Automata

Author: Moraru Gabriel

Theory:

A **parser** is a crucial component in the front-end of a compiler or interpreter. Its primary role is to analyze a sequence of tokens (usually generated by a lexer) and determine if the sequence follows the grammar rules of a programming language or mathematical expression. If it does, the parser constructs a data structure called the **Abstract Syntax Tree (AST)**.

An **Abstract Syntax Tree** is a tree-like representation of the syntactic structure of the input. Unlike concrete syntax trees, an AST omits unnecessary syntax details (like parentheses) and focuses on the hierarchical relationship between language elements. For example, in the expression 3 + 4 * 5, the AST reflects the precedence of multiplication over addition by placing * lower in the tree than +.

Parsers are generally implemented using **recursive descent** (a top-down approach) or **shift-reduce** methods (bottom-up). The parser consists of functions (or rules) that match grammar constructs like expressions, terms, and factors, and build corresponding nodes in the AST such as:

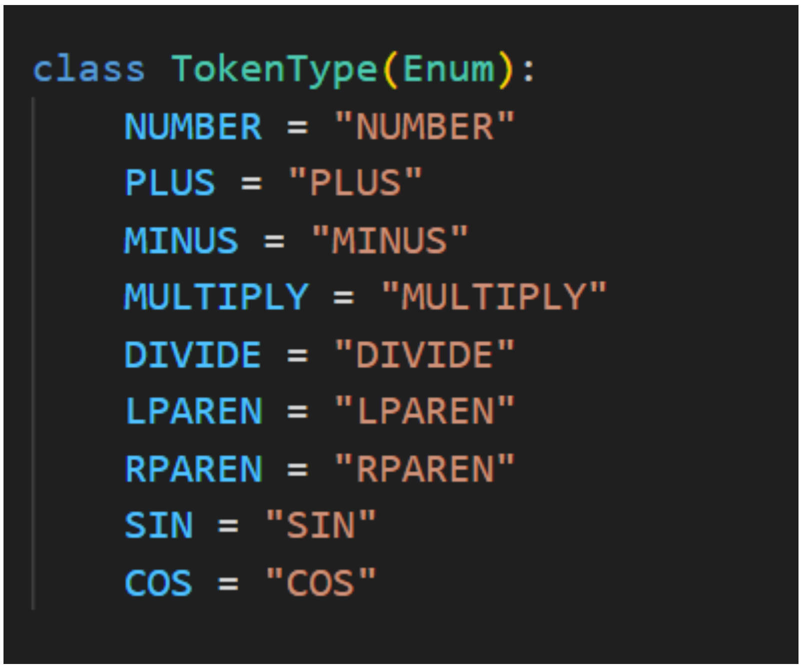
- **NumberNode** for numeric values
- **BinaryOpNode** for binary operations (e.g., +, *)
- **UnaryOpNode** for functions or signs (e.g., -, sin)

The AST becomes the foundation for later phases such as interpretation, compilation, or optimization, making parsing and tree-building essential for language processing.

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 description:



```python
class TokenType(Enum):
    NUMBER = "NUMBER"
    PLUS = "PLUS"
    MINUS = "MINUS"
    MULTIPLY = "MULTIPLY"
    DIVIDE = "DIVIDE"
    LPAREN = "LPAREN"
    RPAREN = "RPAREN"
    SIN = "SIN"
    COS = "COS"
```

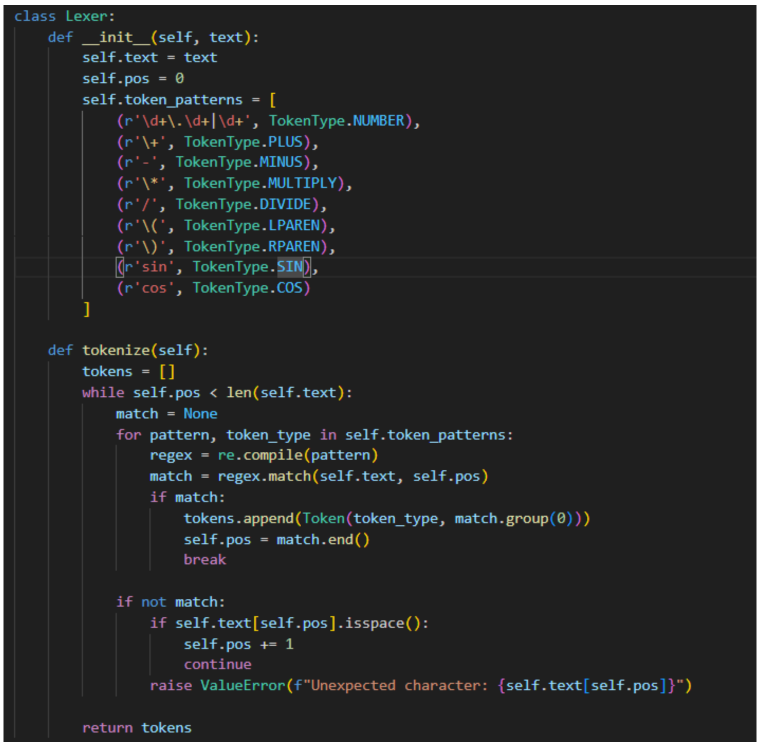
Defines the **types of tokens** (basic language elements) your lexer can recognize:

- Numbers, operators (+, -, *, /), parentheses, and functions (sin, cos).

Token

A class that represents a token with:

- type: the category (from TokenType)
- value: the actual matched string (like "sin" or "30")



```python
class Lexer:
    def __init__(self, text):
        self.text = text
        self.pos = 0
        self.token_patterns = [
            (r'\d+\.\d+|\d+', TokenType.NUMBER),
            (r'\+', TokenType.PLUS),
            (r'-', TokenType.MINUS),
            (r'\*', TokenType.MULTIPLY),
            (r'/', TokenType.DIVIDE),
            (r'\(', TokenType.LPAREN),
            (r'\)', TokenType.RPAREN),
            (r'sin', TokenType.SIN),
            (r'cos', TokenType.COS)
        ]

    def tokenize(self):
        tokens = []
        while self.pos < len(self.text):
            match = None
            for pattern, token_type in self.token_patterns:
                regex = re.compile(pattern)
                match = regex.match(self.text, self.pos)
                if match:
                    tokens.append(Token(token_type, match.group(0)))
                    self.pos = match.end()
                    break

            if not match:
                if self.text[self.pos].isspace():
                    self.pos += 1
                    continue
                raise ValueError(f"Unexpected character: {self.text[self.pos]}")

        return tokens
```

Lexer.__init__(self, text)

Initializes the lexer with:

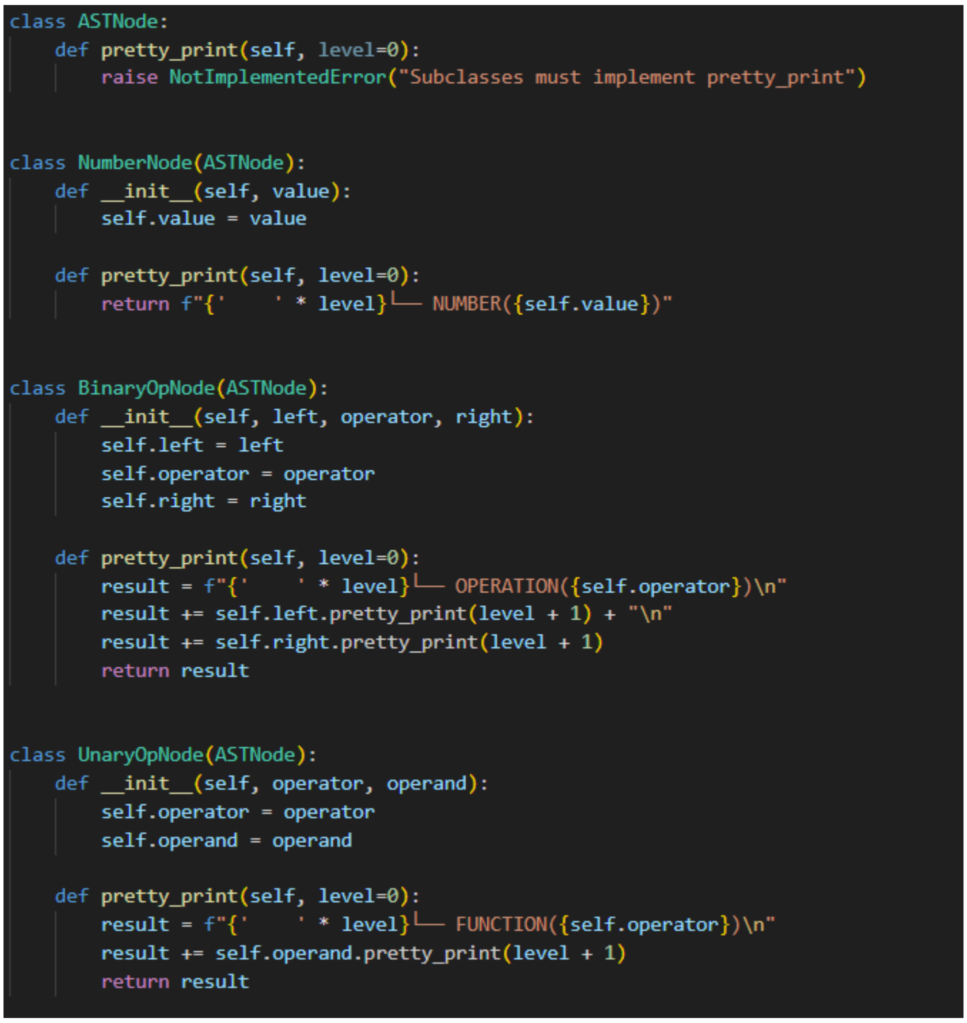
- text: the input string (e.g., "3 * sin(30) + 4 / (2 + cos(60))")
- self.pos: current position in the string
- self.token_patterns: regex patterns paired with TokenType values

Lexer.tokenize(self)

Scans the input string and breaks it into tokens:

- Iterates over the input character by character.
- For each position, it tries all patterns in token_patterns.
- If a match is found, it adds a Token to the list and moves self.pos forward.
- Skips whitespaces.
- Raises an error for unrecognized characters.

Output: A list of Token objects representing the input expression.



```python
class ASTNode:
    def pretty_print(self, level=0):
        raise NotImplementedError("Subclasses must implement pretty_print")


class NumberNode(ASTNode):
    def __init__(self, value):
        self.value = value

    def pretty_print(self, level=0):
        return f"{'    ' * level}└── NUMBER({self.value})"


class BinaryOpNode(ASTNode):
    def __init__(self, left, operator, right):
        self.left = left
        self.operator = operator
        self.right = right

    def pretty_print(self, level=0):
        result = f"{'    ' * level}└── OPERATION({self.operator})\n"
        result += self.left.pretty_print(level + 1) + "\n"
        result += self.right.pretty_print(level + 1)
        return result


class UnaryOpNode(ASTNode):
    def __init__(self, operator, operand):
        self.operator = operator
        self.operand = operand

    def pretty_print(self, level=0):
        result = f"{'    ' * level}└── FUNCTION({self.operator})\n"
        result += self.operand.pretty_print(level + 1)
        return result
```

NumberNode

Represents a numeric literal like 30 or 3.

- Stores the number as a float.
- pretty_print(): outputs the node in a tree-like format.

BinaryOpNode

Represents binary operations like +, *, /.

- Stores the left and right subtrees and the operator.
- pretty_print(): recursively prints the left and right sides.

UnaryOpNode

Represents function calls like sin(...), cos(...).

- Stores the function name and operand (which itself can be an expression).
- pretty_print(): recursively prints the operand.

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

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

    def eat(self, token_type):
        if self.current_token() and self.current_token().type ==
token_type.value:
            self.pos += 1
        else:
            raise ValueError(f"Expected token {token_type}, got
{self.current_token()}")

    def parse(self):
        return self.expr()

    def factor(self):
        token = self.current_token()

        if token.type == TokenType.NUMBER.value:
            self.eat(TokenType.NUMBER)
            return NumberNode(float(token.value))

        elif token.type == TokenType.LPAREN.value:
            self.eat(TokenType.LPAREN)
            node = self.expr()
            self.eat(TokenType.RPAREN)
            return node

        elif token.type in {TokenType.SIN.value, TokenType.COS.value}:
            self.eat(TokenType(token.type))
            operand = self.factor()
            return UnaryOpNode(token.type, operand)

        raise ValueError(f"Unexpected token: {token}")

    def term(self):
        node = self.factor()

        while self.current_token() and self.current_token().type in
{TokenType.MULTIPLY.value, TokenType.DIVIDE.value}:
            token = self.current_token()
            self.eat(TokenType(token.type))
            node = BinaryOpNode(node, token.type, self.factor())

        return node

    def expr(self):
        node = self.term()

        while self.current_token() and self.current_token().type in
{TokenType.PLUS.value, TokenType.MINUS.value}:
            token = self.current_token()
            self.eat(TokenType(token.type))
            node = BinaryOpNode(node, token.type, self.term())
        return node
```

Parser.__init__(self, tokens)

Receives the list of tokens and sets self.pos = 0 to start parsing.

Parser.current_token(self)

Returns the token at the current parsing position.

Parser.eat(self, token_type)

Checks if the current token matches the expected token_type. If so:

- It consumes the token by moving self.pos forward.
- Else, raises an error.

Parser.parse(self)

Entry point. Starts parsing with the expr() rule.

Parser.expr(self)

Handles **addition and subtraction**.

- Calls term() to handle the left side.
- If the next token is + or -, it consumes the operator and recursively builds a BinaryOpNode.

Parser.term(self)

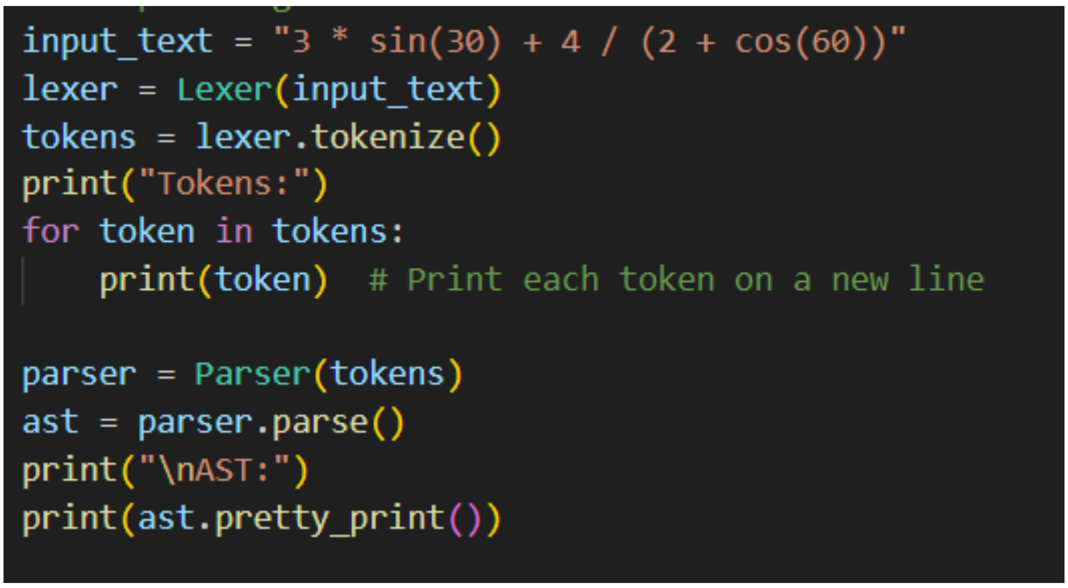
Handles **multiplication and division** (higher precedence).

- Similar to expr(), but for * and /.

Parser.factor(self)

Handles:

- Numbers → returns a NumberNode
- Parentheses → recursively calls expr() inside the parentheses
- Functions (sin, cos) → returns a UnaryOpNode



```
input_text = "3 * sin(30) + 4 / (2 + cos(60))"
lexer = Lexer(input_text)
tokens = lexer.tokenize()
print("Tokens:")
for token in tokens:
    print(token)  # Print each token on a new line

parser = Parser(tokens)
ast = parser.parse()
print("\nAST:")
print(ast.pretty_print())
```

Lexer: breaks this into tokens like 3, *, sin, (, 30,), +, etc.

Parser: builds a tree structure representing the expression.

AST.pretty_print(): prints a readable version of the tree.

Conclusion:

In this lab, we built a simple but working lexical analyzer and parser for arithmetic expressions. These expressions included basic operations like addition, subtraction, multiplication, and division, along with trigonometric functions such as sin and cos, and the use of parentheses for grouping.

The lexer read the input string and turned it into tokens, which are the smallest meaningful parts like numbers, operators, or function names. This step cleaned up the input and made it easier to process.

Next, the parser used a recursive approach to create an Abstract Syntax Tree or AST. It followed the rules of operator precedence so that expressions like 3 plus 4 times 5 were understood correctly. Unary functions like sin and cos were handled with a specific type of node, while binary operations like addition and multiplication used another.

A key part of the project was adding a way to print the AST clearly, which helped us see the structure of the expression and how parts were grouped.

Overall, this lab gave us hands-on experience with how compilers break down and understand code. It laid the groundwork for more advanced topics like semantic analysis and code generation, and showed how these skills are used in tools like interpreters, custom languages, and code editors.

References:

[1] [Parsing Wiki]()

[2] Abstract Syntax Tree Wiki