**Topic: Parser & Building an Abstract Syntax Tree**

**Course: Formal Languages & Finite Automata**

**Author: Obrijan Filip**

**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:

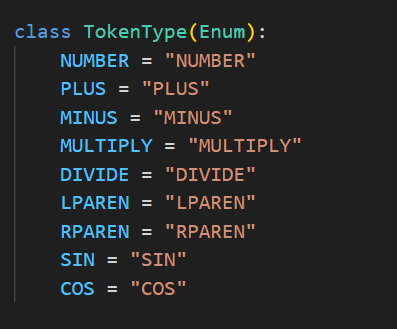
* **NumberNode** for numeric values
* **BinaryOpNode** for binary operations (e.g., +, \*)
* **UnaryOpNode** for functions or signs (e.g., -, sin)

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:
   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:**

****

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")



**Lexer.\_\_init\_\_(self, text)**

Initializes the lexer with:

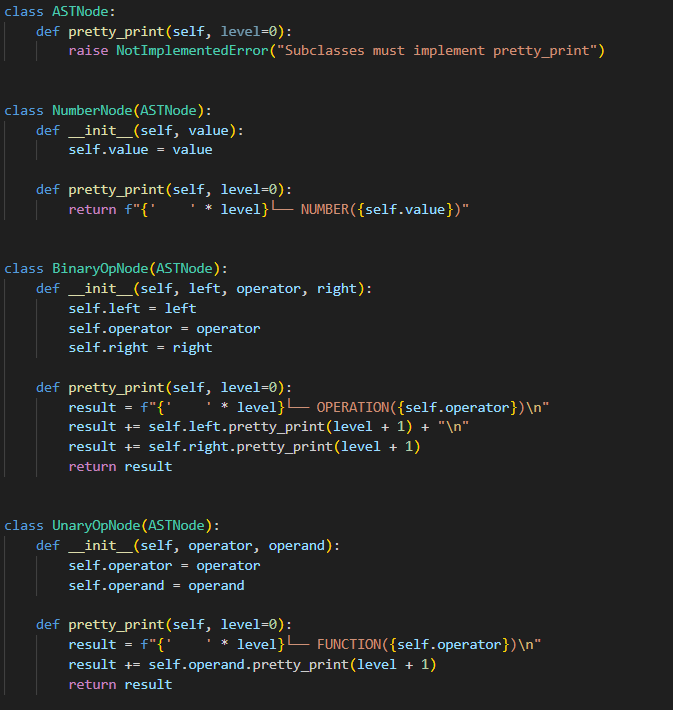
* text: the input string (e.g., "3 \* sin(30) + 4 / (2 + cos(60))")
* self.pos: current position in the string
* self.token\_patterns: regex patterns paired with TokenType values

**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.



**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)**

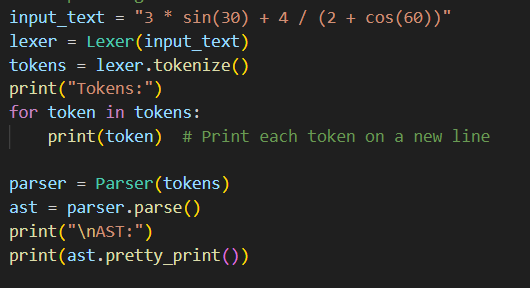
Handles **multiplication and division** (higher precedence).

* 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



**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 laboratory work, we developed a simplified yet functional **lexical analyzer (lexer)** and **syntax analyzer (parser)** for arithmetic expressions, which include not only basic operations such as addition, subtraction, multiplication, and division but also support for trigonometric functions like sin and cos, as well as parentheses for grouping. The primary goal was to understand how high-level expressions in programming languages are broken down and structured internally, leading to the construction of an **Abstract Syntax Tree (AST)**.

The lexer was responsible for reading the input string and converting it into a list of **tokens**, which are atomic units such as numbers, operators, or function names. This step is crucial in the compilation process, as it isolates meaningful components from raw input, handling complexities like ignoring whitespace and recognizing multi-character tokens (e.g., sin, cos, or multi-digit numbers).

Once tokenized, the input is passed to the parser, which applies a recursive descent strategy to build the AST. The parser respects operator precedence and associativity, ensuring that expressions like 3 + 4 \* 5 are parsed correctly (i.e., multiplication before addition). Additionally, it demonstrates how **unary operations** such as sin(x) or cos(x) are handled using a dedicated node structure (UnaryOpNode), while **binary operations** (like +, \*, /) are managed through BinaryOpNode.

One of the key achievements of this lab was the implementation of the pretty\_print method for AST nodes, which gives a clear visual representation of the tree structure. This feature is particularly useful for debugging and understanding the hierarchical nature of expressions and how sub-expressions relate to one another.

Through this exercise, we deepened our understanding of fundamental principles in **compiler design**, particularly the front-end stages such as lexical and syntactic analysis. We also saw how these concepts are applied in real interpreters and compilers to convert source code into an internal form suitable for evaluation or code generation.

In conclusion, this lab not only provided practical experience in building parsers and working with abstract syntax trees but also laid the foundation for more advanced topics in programming language processing, such as semantic analysis, code optimization, and virtual machine execution. The skills developed here are broadly applicable in fields like interpreters, domain-specific languages (DSLs), and even modern development tools like code editors and linters.

**References:**

**[1]**[**Parsing Wiki**](https://en.wikipedia.org/wiki/Parsing)

**[2]**[**Abstract Syntax Tree Wiki**](https://en.wikipedia.org/wiki/Abstract_syntax_tree)