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-221 Reabciuc Daria-Brianna

Verified:

asist. univ. Crețu Dumitru

Chisinau 2024

Objectives

Parsing is the process of analyzing a sequence of symbols or tokens according to the rules of a formal grammar. It's often used in programming to understand and interpret code or text.

An Abstract Syntax Tree (AST) is a hierarchical representation of the syntactic structure of code or text. It's commonly used in compilers and interpreters to understand and manipulate the structure of programming languages.

To implement parsing, you can start by defining a TokenType type, similar to an enum, to categorize different types of tokens during lexical analysis. Regular expressions can then be used to identify and classify tokens based on their patterns.

After tokenization, you'll need to build data structures for an AST. This structure should represent the syntactic elements of the text you're processing, such as expressions, statements, and declarations.

Finally, you can create a parser program that takes the tokenized input and constructs the AST based on the grammar rules of the language or text format you're dealing with. This parser extracts syntactic information from the input text, allowing you to analyze and manipulate it programmatically.

Implementation

Tokenization involves defining a TokenType or similar type to categorize tokens like identifiers, keywords, operators, and literals. Regular expressions help match token patterns for lexical analysis (e.g., "[a-zA-Z]+" for identifiers).

For the AST, design data structures representing nodes for syntactic constructs (e.g., AddExpression, SubtractionExpression) linked hierarchically per language rules.

Develop a parsing algorithm using tokenized input to build the AST, typically via recursive descent parsing. Grammar rules guide node structure creation (e.g., recognizing "expr + expr" as an AddExpression).

Create a parser program integrating tokenization, parsing, and AST construction, with error handling for syntax issues. Optionally, add functionality to traverse and analyze the AST further.

```
class TokenType(Enum):
    OPEN_PARENTHESIS = auto()
    CLOSE_PARENTHESIS = auto()
    MATH_OPERATION = auto()
    NUMBERS = auto()
    START = auto()

transitions = {
    TokenType.OPEN_PARENTHESIS: [TokenType.NUMBERS, TokenType.OPEN_PARENTHESIS],
    TokenType.MATH_OPERATION: [TokenType.NUMBERS, TokenType.OPEN_PARENTHESIS],
    TokenType.CLOSE_PARENTHESIS: [TokenType.MATH_OPERATION, TokenType.CLOSE_PARENTHESIS],
    TokenType.NUMBERS: [TokenType.NUMBERS, TokenType.CLOSE_PARENTHESIS, TokenType.MATH_OPERATION],
    TokenType.START: [TokenType.OPEN_PARENTHESIS, TokenType.NUMBERS]
}

data = {
    TokenType.OPEN_PARENTHESIS: [r"\(", r"\["],
    TokenType.CLOSE_PARENTHESIS: [r"\)", r"\]"],
    TokenType.MATH_OPERATION: [r"[+\-*/%^]"],
    TokenType.NUMBERS: [r"\d+"]
}
```

Fig. 1. Class TokenType

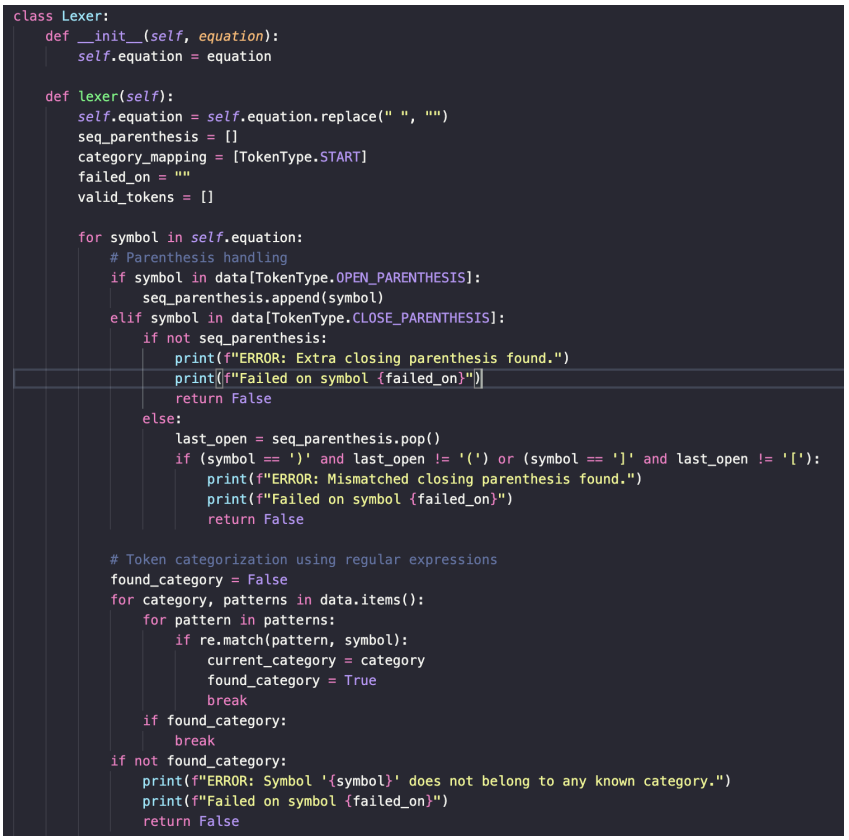
The transitions dictionary maps each token type to a list of possible following token types. For instance:

- An OPEN_PARENTHESIS can be followed by either NUMBERS or another OPEN_PARENTHESIS.
- A MATH_OPERATION can be followed by NUMBERS or an OPEN_PARENTHESIS.
- A CLOSE_PARENTHESIS can be followed by a MATH_OPERATION or another CLOSE_PARENTHESIS.
- NUMBERS can be followed by another NUMBERS, a CLOSE_PARENTHESIS, or a MATH_OPERATION.
- The START token can be followed by an OPEN_PARENTHESIS or NUMBERS.

The data dictionary specifies the regular expressions for each token type. For example:

- OPEN_PARENTHESIS is defined by the regex patterns r"\(" (for a parenthesis) and r"\[" (for a square bracket).
- CLOSE_PARENTHESIS matches r"\)" and r"\]".
- MATH_OPERATION uses the pattern [+\-*/%^] to match arithmetic operators.
- NUMBERS is defined by the regex pattern r"\d+" to match sequences of digits.

Together, these structures and mappings help in tokenizing and parsing input based on defined token types and their transitions.



```
class Lexer:
    def __init__(self, equation):
        self.equation = equation

    def lexer(self):
        self.equation = self.equation.replace(" ", "")
        seq_parenthesis = []
        category_mapping = [TokenType.START]
        failed_on = ""
        valid_tokens = []

        for symbol in self.equation:
            # Parenthesis handling
            if symbol in data[TokenType.OPEN_PARENTHESIS]:
                seq_parenthesis.append(symbol)
            elif symbol in data[TokenType.CLOSE_PARENTHESIS]:
                if not seq_parenthesis:
                    print(f"ERROR: Extra closing parenthesis found.")
                    print(f"Failed on symbol {failed_on}")
                    return False
                else:
                    last_open = seq_parenthesis.pop()
                    if (symbol == ')' and last_open != '(') or (symbol == ']' and last_open != '['):
                        print(f"ERROR: Mismatched closing parenthesis found.")
                        print(f"Failed on symbol {failed_on}")
                        return False

            # Token categorization using regular expressions
            found_category = False
            for category, patterns in data.items():
                for pattern in patterns:
                    if re.match(pattern, symbol):
                        current_category = category
                        found_category = True
                        break
                if found_category:
                    break
            if not found_category:
                print(f"ERROR: Symbol '{symbol}' does not belong to any known category.")
                print(f"Failed on symbol {failed_on}")
                return False
```

Fig. 2. Lexer class

The **Lexer** class takes an equation as input and performs lexical analysis. It initializes by storing the equation and defines structures for handling parentheses, token categories, and validation.

The **lexer** method processes the equation:

- Removes spaces from the equation.
- Initializes an empty list **seq_parenthesis** to track the sequence of open parentheses encountered.
- Sets the initial category mapping with **[TokenType.START]**, representing the start of the token sequence.
- Initializes variables **failed_on** and **valid_tokens** for error tracking and storing valid tokens.

It iterates through each symbol in the equation:

- Handles parentheses by checking if a symbol is an open or close parenthesis and updates **seq_parenthesis** accordingly. It also checks for matching pairs and reports errors for mismatched or extra closing parentheses.
- Categorizes symbols using regular expressions defined in the **data** dictionary. If a symbol doesn't match any known category, it reports an error.
- Checks transitions between token categories based on the **transitions** dictionary. If a transition is not allowed, it reports an error.

- Updates the category mapping and adds valid tokens to the **valid_tokens** list.

After processing all symbols:

- Checks for any remaining open parentheses and reports an error if found.

The method returns the final category mapping and list of valid tokens if no errors are encountered during lexical analysis. Otherwise, it prints the error message and returns False.

```
class Parser:
    def __init__(self, category_mapping, valid_tokens):
        self.category_mapping = category_mapping
        self.valid_tokens = valid_tokens

    # Construct the AST and print it
    def parse(self):
        root = Node(self.category_mapping[0].name)
        parent = root
        for token, category in zip(self.valid_tokens, self.category_mapping[1:]):
            node = Node(token, parent=parent)
            parent = Node(category.name, parent=parent)

        for pre, _, node in RenderTree(root):
            print("%s%s" % (pre, node.name))
```

Fig. 3.Parser Class

The **Parser** class takes in the category mapping and valid tokens generated by the lexer to construct an Abstract Syntax Tree (AST) and then prints it.

In the **parse** method:

- It initializes a root node using the name of the first category in the category mapping.
- Then, it iterates through the valid tokens and corresponding categories.
- For each token, it creates a node with that token's value and sets its parent to the previous node.
- After creating the token nodes, it creates category nodes to represent the structure of the AST.
- Finally, it uses the **RenderTree** function (assumed to be part of a tree rendering library) to traverse the AST and prints each node's name along with its indentation level.

This process effectively constructs the AST based on the token categories and their relationships, providing a structured representation of the parsed input.

Conclusion

The integration of the Lexer and Parser classes establishes a sophisticated system for syntactic analysis and parsing of textual inputs, particularly in the context of programming languages or mathematical expressions. The Lexer intricately categorizes symbols into distinct token types, employing regular expressions and transition rules to ensure accurate lexical analysis. This meticulous categorization serves as the foundation for subsequent parsing operations.

Upon receiving validated tokens and their respective categories from the Lexer, the Parser orchestrates the construction of an Abstract Syntax Tree (AST) with finesse. Leveraging the hierarchical relationships encoded within the category mapping and valid tokens, the Parser methodically assembles nodes representing the syntactic elements of the input. This process inherently captures the structural essence of the input, enabling a comprehensive representation conducive to further analysis and manipulation.

The synergy between the Lexer and Parser transcends mere tokenization and parsing; it embodies a comprehensive approach to understanding and processing textual data. This robust framework not only ensures syntactic correctness but also lays the groundwork for semantic analysis and execution. By encapsulating complex parsing logic within modular and reusable components, this system fosters scalability, adaptability, and maintainability—a testament to its sophistication in handling diverse parsing requirements across varied domains and languages.