Laboratory Work Nr.6

Parser & Building an Abstract Syntax Tree

Course: Formal Languages & Finite Automata

Author: Tofan Liviu

Group: FAF-223

Chişinău – 2024

Theory

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.

Parsing, syntax analysis, or syntactic analysis is the process of analyzing a string of symbols, either in natural language, computer languages or data structures, conforming to the rules of a formal grammar. The term parsing comes from Latin pars (orationis), meaning part (of speech). The term has slightly different meanings in different branches of linguistics and computer science. Traditional sentence parsing is often performed as a method of understanding the exact meaning of a sentence or word, sometimes with the aid of devices such as sentence diagrams. It usually emphasizes the importance of grammatical divisions such as subject and predicate. Within computational linguistics the term is used to refer to the formal analysis by a computer of a sentence or other string of words into its constituents, resulting in a parse tree showing their syntactic relation to each other, which may also contain semantic information. Some parsing algorithms generate a parse forest or list of parse trees from a string that is syntactically ambiguous. The term is also used in psycholinguistics when describing language comprehension. In this context, parsing refers to the way that human beings analyze a sentence or phrase (in spoken language or text) "in terms of grammatical constituents, identifying the parts of speech, syntactic relations, etc." This term is especially common when discussing which linguistic cues help speakers interpret garden-path sentences. Within computer science, the term is used in the analysis of computer languages, referring to the syntactic analysis of the input code into its component parts in order to facilitate the writing of compilers and interpreters. The term may also be used to describe a split or separation.

An abstract syntax tree (AST) is a data structure used in computer science to represent the structure of a program or code snippet. It is a tree representation of the abstract syntactic structure of text (often source code) written in a formal language. Each node of the tree denotes a construct occurring in the text. It is sometimes called just a syntax tree. The syntax is "abstract" in the sense that it does not represent every detail appearing in the real syntax, but rather just the structural or content-related details. For instance, grouping parentheses are implicit in the tree structure, so these do not have to

be represented as separate nodes. Likewise, a syntactic construct like an if-condition-then statement may be denoted by means of a single node with three branches. This distinguishes abstract syntax trees from concrete syntax trees, traditionally designated parse trees. Parse trees are typically built by a parser during the source code translation and compiling process. Once built, additional information is added to the AST by means of subsequent processing, e.g., contextual analysis. Abstract syntax trees are also used in program analysis and program transformation systems.

Figure 1. Abstract Syntex Tree (AST) example

Objectives

---

1. Get familiar with parsing, what it is and how it can be programmed.
2. Get familiar with the concept of AST.
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:

- Have a type TokenType (like an enum) that can be used in the lexical analysis to categorize the tokens.

- 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 began by defining a class called `TokenType` using the `enum.Enum` module. It includes various token types such as whitespace, comments, identifiers, keywords, strings, numbers, operators, and punctuation symbols like braces, parentheses, commas, semicolons, and colons. Each token type is assigned a unique integer value.

```
class TokenType(enum.Enum):
    WHITESPACE = 1
    COMMENT = 2
    IDENTIFIER = 3
    KEYWORD = 4
    STRING = 5
    NUMBER = 6
    OPERATOR = 7
    BRACE_OPEN = 8
    BRACE_CLOSE = 9
    PAREN_OPEN = 10
    PAREN_CLOSE = 11
    SEMICOLON = 12
    COMMA = 13
    COLON = 14
```

Figure 2. Class TokenType

After I defined the function *lexer()*, it operates on the provided `code`. The function iterates over the code, stripping leading and trailing whitespace. Then, it checks for matches between the code and predefined token regular expressions stored in `TOKENS`. If a match is found, it extracts the matched value, excluding whitespace and comments, and appends it along with its token type to the `tokens` list. The function continues until no more code remains to process. If no match is found, it raises a `SyntaxError` indicating unknown code.

```
def lexer(code):
    tokens = []
    while code:
        code = code.strip()
        match_found = False
        for token_type, token_regex in TOKENS:
            regex = re.compile(token_regex)
            match = regex.match(code)
            if match:
                value = match.group(0).strip()
                if token_type != TokenType.WHITESPACE and token_type != TokenType.COMMENT:
                    tokens.append((token_type, value))
                code = code[match.end():]
                match_found = True
                break
        if not match_found:
            raise SyntaxError(f'Unknown code: {code}')
    return tokens
```

Figure 3. lexer() function

After I've created the `ASTNode` class to serve as the building block for constructing Abstract Syntax Trees (ASTs). This class defines the structure of individual nodes within the tree, encapsulating attributes such as `type`, `value`, and `children`. These attributes facilitate the organization and representation of hierarchical data, essential for parsing and analyzing code structures.

Furthermore, the `__repr__` method provides a succinct string representation of `ASTNode` instances, aiding in the visualization and debugging of ASTs.

Through the `ASTNode` class and associated functions, I've established a framework for efficiently parsing code and generating ASTs, laying the groundwork for subsequent code analysis and transformation tasks.

```
class ASTNode:
    def __init__(self, type, children=None, value=None):
        self.type = type
        self.value = value
        self.children = children if children is not None else []

    def __repr__(self):
        type_name = self.type.name if isinstance(self.type, enum.Enum) else self.type
        return f"{type_name}({self.value}, {self.children})"
```

Figure 4. ASTNode class and intialization

The `parse` function is designed to transform a sequence of tokens into a hierarchical representation known as an Abstract Syntax Tree (AST). Here's an overview of its operation:

1. *Initialization:* It initializes the root of the AST with a node of type `TokenType.KEYWORD`, denoted as "ROOT".

2. Token Processing Loop: It iterates over each token in the provided `tokens` list.

3. *Token Type Check*: For each token, it examines its type.

- If the token is a `TokenType.KEYWORD` or `TokenType.IDENTIFIER`, it creates a new node with that type and value, appending it as a child of the current node.

- If the token is a `TokenType.STRING` or `TokenType.NUMBER`, it creates a new node with that type and value, also appending it as a child of the current node.

4. *Traversal*: After processing all tokens, the function returns the root node of the constructed AST.

In essence, the `parse` function encapsulates the process of syntactic analysis, organizing tokens into a structured representation that reflects the hierarchical nature of

code elements. This transformation facilitates subsequent stages of code analysis, interpretation, and transformation.

After defining tokens, the code can tokenize input according to specific patterns. These patterns cover whitespace, comments, keywords, identifiers, strings, numbers, operators, and punctuation symbols such as colons, braces, parentheses, semicolons, and commas.

```
TOKENS = [
    (TokenType.WHITESPACE, r'\s+'),
    (TokenType.COMMENT, r'#[^\n]*'),
    (TokenType.KEYWORD, r'\b(?:if|else|elif|for|while|def|class|return|import|from|as|True|False|None)\b'),
    (TokenType.IDENTIFIER, r'[a-zA-Z_][a-zA-Z0-9_]*'),
    (TokenType.STRING, r'\'(?:\\.|[^\'])*\'|"(?:\\.|[^"])*"'),
    (TokenType.NUMBER, r'\b\d+(?:\.\d+)?\b'),
    (TokenType.OPERATOR, r'\+|-|\*|/|==|!=|<=|>=|<|>'),
    (TokenType.COLON, r':'),
    (TokenType.BRACE_OPEN, r'\{'),
    (TokenType.BRACE_CLOSE, r'\}'),
    (TokenType.PAREN_OPEN, r'\('),
    (TokenType.PAREN_CLOSE, r'\)'),
    (TokenType.SEMICOLON, r';'),
    (TokenType.COMMA, r','),
]
```

Figure 5. Tokens

I've implemented a function to visually represent the hierarchical structure of an abstract syntax tree (AST) using the Graphviz library. This function, named `add_nodes_edges`, recursively traverses the AST, adding nodes and edges to the graph representation. By labeling each node with its type and value, the resulting visualization offers insights into the organization of the code structure. Finally, the generated PDF file provides a graphical overview of the AST, aiding in code analysis and comprehension.

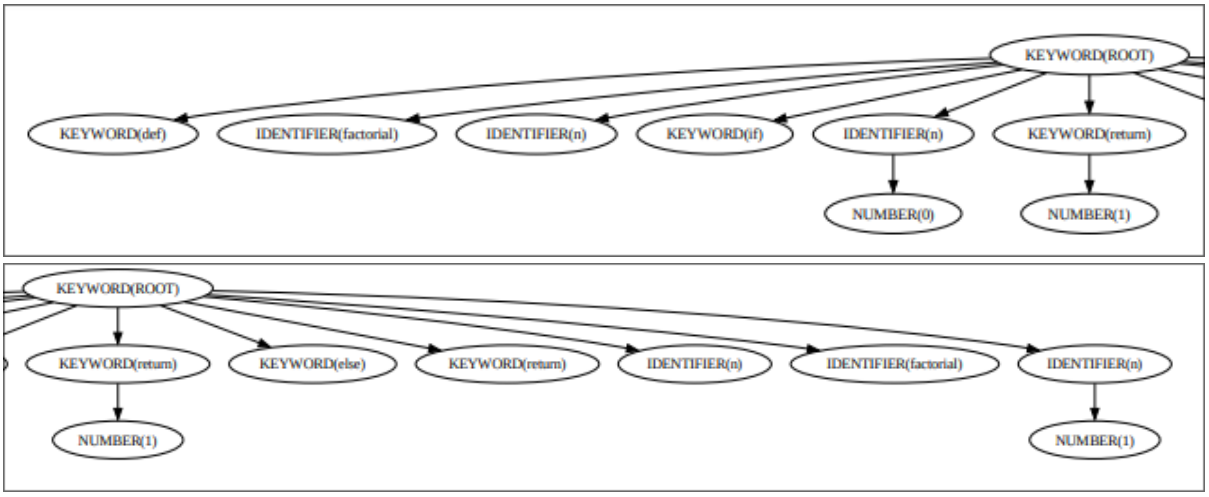

Figure 6. Graphical Overview of AST

Conclusion

This laboratory work provided a comprehensive exploration of Abstract Syntax Trees (AST), lexer implementation, and visualization using graph structures. Through this exercise, I gained a deeper understanding of how programming languages parse and represent code internally.

The lexer efficiently tokenizes the source code into meaningful units such as keywords, identifiers, strings, numbers, operators, and punctuation. This process is essential for subsequent analysis and interpretation of the code.

The parser, in turn, constructs an AST from the tokens generated by the lexer. The AST organizes the code into a hierarchical structure that captures its syntactic elements and relationships. By traversing this tree, we can analyze the structure of the code and extract valuable information.

Graph visualization played a crucial role in understanding the structure of the AST. Using the Graphviz library, we created visual representations of the AST, making it easier to comprehend complex code structures and relationships. This visual aspect greatly enhances the understanding and debugging of code.

Moreover, the code demonstrated here is not limited to a specific Python snippet but can handle any Python source code. This versatility stems from the modular design and generality of the implemented functions and classes. By encapsulating functionality into reusable components, we enable the analysis of diverse Python codebases.

In conclusion, this laboratory work provided a practical foundation for working with ASTs, lexers, and graph visualization in the context of Python programming. It equipped me with valuable skills that can be applied to various tasks such as code analysis, optimization, and transformation.