Topic: Chomsky Normal Form

Course: Parser & Building an Abstract Syntax Tree

Author: Durbailo Daniel

Group: FAF-222

Chisinau 2024

1.Theory

What is Parsing?:

Parsing, a fundamental process in computer science, is primarily utilized to extract syntactic meaning from text. It plays a vital role in the operation of compilers and interpreters by analyzing a sequence of symbols according to formal grammar rules. The main result of parsing is the creation of a parse tree, which illustrates the syntactic arrangement of the input and may also incorporate semantic details useful for subsequent compilation phases. This functionality is essential for software to accurately interpret and manipulate input code or data, making parsing a cornerstone of programming language implementation and data processing.

What is an Abstract Syntax Tree? (AST):

An Abstract Syntax Tree (AST) delineates the syntactic structure of input text through multiple levels of abstraction. Unlike simplistic parse trees, ASTs emphasize the higher-level constructs of the text, omitting unnecessary syntactic intricacies. Each node in the AST corresponds to a specific construct in the input, organized to reflect their syntactic and sometimes semantic connections.

ASTs are invaluable in software development, particularly in compilation and code optimization. By transforming input text into a structured tree format, ASTs facilitate more efficient analysis and manipulation of code. They enable various compiler optimizations by offering a clear framework for implementing transformation rules and conducting static code analysis effectively.

Some Additional Considerations:

In addition to their role in compilation and code optimization, parsing and AST construction also find application in various other domains. For instance, they are extensively used in natural language processing (NLP) to analyze and understand human language. Furthermore, they play a crucial role in data parsing and processing tasks, enabling the extraction of structured information from unstructured data sources such as log files, web pages, and documents.

Moreover, modern programming languages often incorporate advanced parsing techniques to support features like pattern matching, syntax highlighting, and code refactoring tools. These techniques leverage the power of parsing and ASTs to provide developers with powerful tools for writing, analyzing, and maintaining code.

In summary, parsing and AST construction are foundational concepts in computer science, with applications spanning a wide range of fields including software development, natural language processing, and data analytics. Their ability to extract and represent the structure of textual information is essential for enabling intelligent processing and manipulation of data in various domains.

2.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.

3.Evaluation

1. The project should be located in a ***public*** repository on a git hosting service (preferably Github).

- You only need to have one repository.
- Please don't name your folders based on the lab work (e.g. Lab1, Lab2 etc.). You'll be penalized for this.
- Please organize your files in a logical way for easier access.

2. It should contain an explanation in the form of a Markdown with the standard structure from the ../REPORT_TEMPLATE.md file. The report needs to be put with the other reports in the corresponding folder.

3. In order to make the evaluation as optimal as possible you should obey the indications and document the project accordingly so that I could evaluate them by myself. The works can be presented by you as well if you want either at the lectures or online.

4. You need to submit on ELSE the URL to the repository. Not a huge letter/message to me telling what you did, or some .zip file or other meaningless stuff. Just the URL. And preferably a valid URL so that I won't waste time trying to find your repo on Github.

5. If the work doesn't correspond to the requirements I'll revert the submission and leave some comments. If needed you can reach out and ask questions.

6. The deadline for this lab work is until 28 April, 23:59

7. After the deadline, the students will have to present the laboratory works, and for each week, the max grade would be decreased by 1.

4.Implementation:

In this laboratory work, I utilized the Graphviz library to harness its powerful features for creating and visualizing graphs. Graphviz offers a comprehensive set of tools for constructing, rendering, and analyzing graphs and networks. Inspired by its counterpart in the Java ecosystem, Graphviz provides a versatile platform for representing and manipulating graph-based data structures. With its intuitive interface and extensive documentation, Graphviz has become a popular choice for a wide range of applications, including but not limited to, software engineering, data visualization, network analysis, and algorithm development. Its seamless integration with Python makes it an invaluable resource for researchers, developers, and data scientists seeking to leverage the power of graphical representations in their projects.

1.Description: Creating the TokenType Class:

Explanation:

This code defines a Python `enum` class named `TokenType`. Enums, short for enumerations, are a convenient way to define symbolic names for a set of unique values. In this `TokenType` enum, each symbolic name corresponds to a specific token type used in lexical analysis or parsing.

The breakdown of token types used by me:

1. `NUMBER`: Represents numeric literals.

2. `KEYWORD`: Represents keywords in a programming language.

3. `IDENTIFIER`: Represents identifiers, such as variable names.

4. `LEFT_PAREN`: Represents the left parenthesis symbol "(".

5. `RIGHT_PAREN`: Represents the right parenthesis symbol ")".

6. `LEFT_SQUARE`: Represents the left square bracket symbol "[".

7. `RIGHT_SQUARE`: Represents the right square bracket symbol "]".

8. `LEFT_CURLY`: Represents the left curly brace symbol "{".

9. `RIGHT_CURLY`: Represents the right curly brace symbol "}".

10. `LESS_THAN`: Represents the less than symbol "<".

11. `GREATER_THAN`: Represents the greater than symbol ">".

12. `EQUAL`: Represents the equal symbol "=".

13. `DOUBLE_EQUAL`: Represents the double equal symbol "==" for comparison.

14. `PLUS`: Represents the plus symbol "+" for addition.

15. `MINUS`: Represents the minus symbol "-" for subtraction.

16. `ASTERISK`: Represents the asterisk symbol "*" for multiplication.

17. `SLASH`: Represents the slash symbol "/" for division.

18. `HASH`: Represents the hash symbol "#" for comments.

19. `DOT`: Represents the dot symbol ".".

20. `COMMA`: Represents the comma symbol ",".

21. `COLON`: Represents the colon symbol ":".

22. `SEMICOLON`: Represents the semicolon symbol ";".

23. `SINGLE_QUOTE`: Represents the single quote symbol "'".

24. `DOUBLE_QUOTE`: Represents the double quote symbol """.

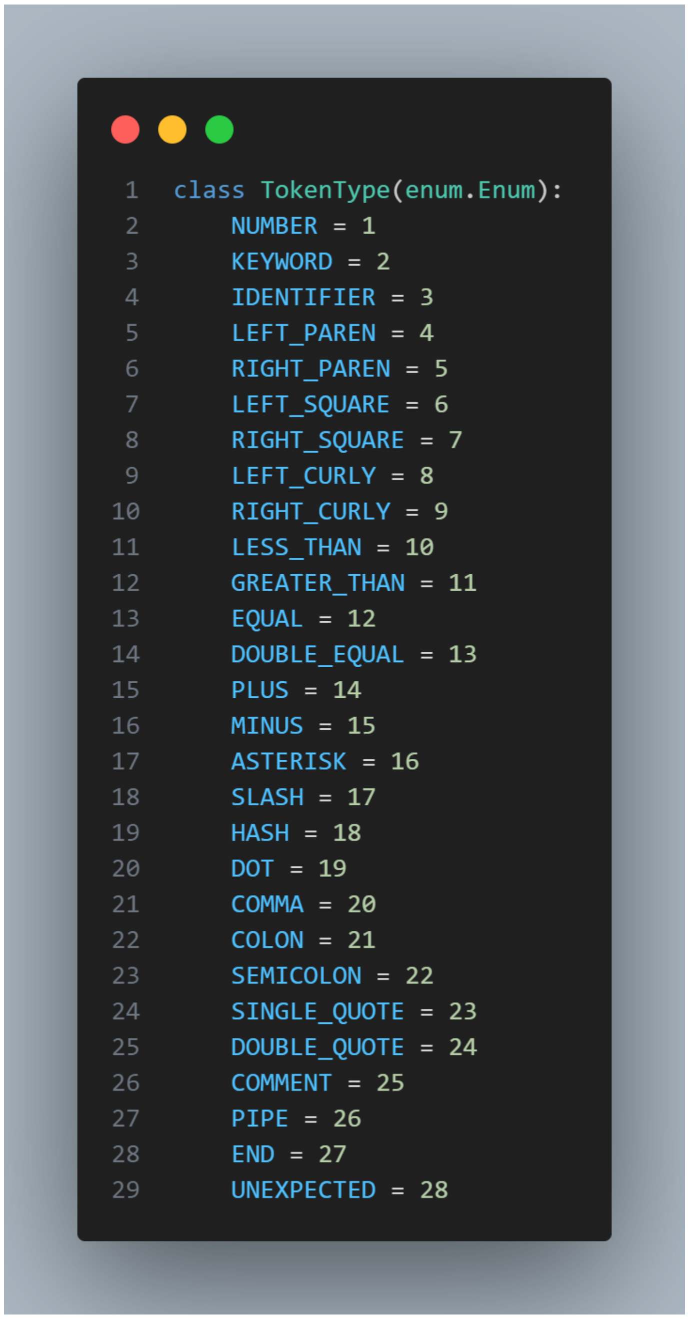
25. `COMMENT`: Represents comments in the code.

26. `PIPE`: Represents the pipe symbol "|" for bitwise operations.

27. `END`: Represents the end of the input or token stream.

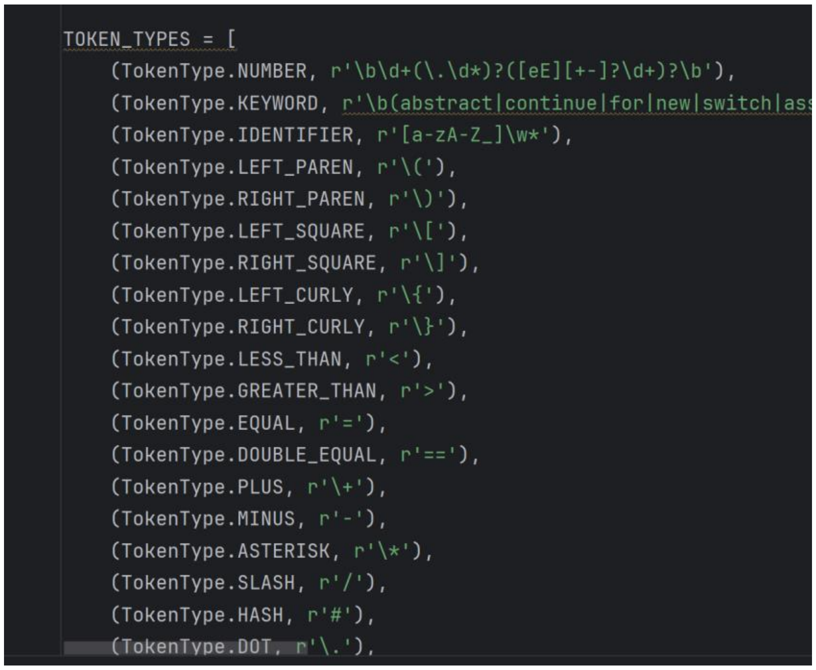
28. `UNEXPECTED`: Represents unexpected or unrecognized tokens.

By using this enumeration, I can improve code readability and maintainability by referring to token types using descriptive names rather than hard-coded numeric values throughout their codebase.



```python
class TokenType(enum.Enum):
    NUMBER = 1
    KEYWORD = 2
    IDENTIFIER = 3
    LEFT_PAREN = 4
    RIGHT_PAREN = 5
    LEFT_SQUARE = 6
    RIGHT_SQUARE = 7
    LEFT_CURLY = 8
    RIGHT_CURLY = 9
    LESS_THAN = 10
    GREATER_THAN = 11
    EQUAL = 12
    DOUBLE_EQUAL = 13
    PLUS = 14
    MINUS = 15
    ASTERISK = 16
    SLASH = 17
    HASH = 18
    DOT = 19
    COMMA = 20
    COLON = 21
    SEMICOLON = 22
    SINGLE_QUOTE = 23
    DOUBLE_QUOTE = 24
    COMMENT = 25
    PIPE = 26
    END = 27
    UNEXPECTED = 28
```

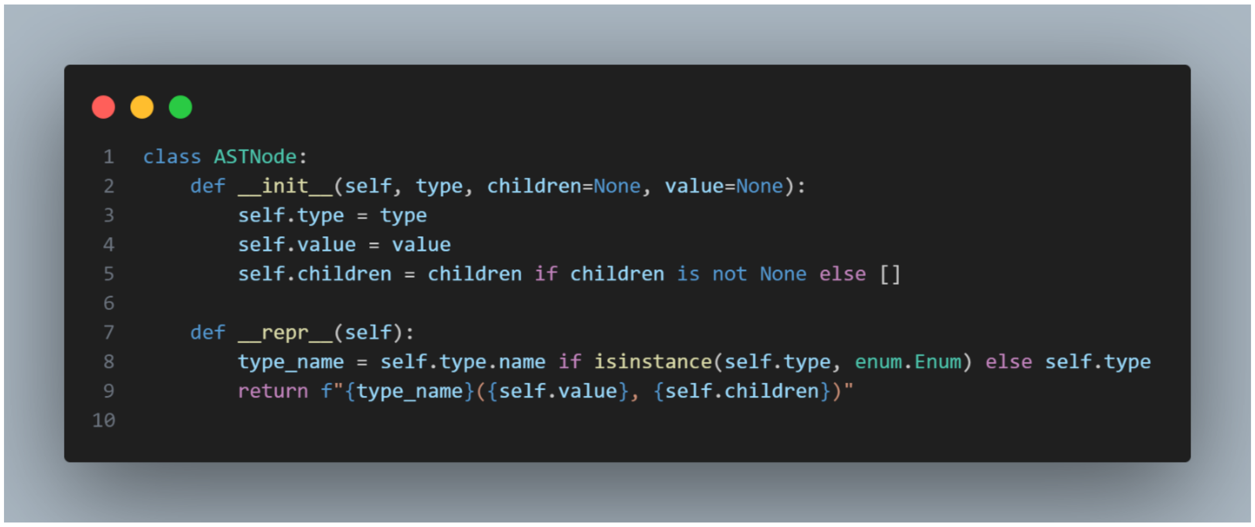
Figure 1. *TokenType Class*



```
TOKEN_TYPES = [
    (TokenType.NUMBER, r'\b\d+(\.\d*)?([eE][+-]?\d+)?\b'),
    (TokenType.KEYWORD, r'\b(abstract|continue|for|new|switch|ass
    (TokenType.IDENTIFIER, r'[a-zA-Z_]\w*'),
    (TokenType.LEFT_PAREN, r'\('),
    (TokenType.RIGHT_PAREN, r'\)'),
    (TokenType.LEFT_SQUARE, r'\['),
    (TokenType.RIGHT_SQUARE, r'\]'),
    (TokenType.LEFT_CURLY, r'\{'),
    (TokenType.RIGHT_CURLY, r'\}'),
    (TokenType.LESS_THAN, r'<'),
    (TokenType.GREATER_THAN, r'>'),
    (TokenType.EQUAL, r'='),
    (TokenType.DOUBLE_EQUAL, r'=='),
    (TokenType.PLUS, r'\+'),
    (TokenType.MINUS, r'-'),
    (TokenType.ASTERISK, r'\*'),
    (TokenType.SLASH, r'/'),
    (TokenType.HASH, r'#'),
    (TokenType.DOT, r'\.'),
```

Figure 2. *Some token types used in the TokenType Class*

2.ASTNode Class:



```
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 3. *ASTNode Class*

Explanation:

The class `ASTNode` represents a node in an Abstract Syntax Tree (AST).

- Constructor (`__init__`):

  - The constructor initializes an `ASTNode` object with three optional parameters: `type`, `children`, and `value`.

  - `type`: Represents the type of the node, which could be an enum value or any other object.

  - `value`: Represents the value associated with the node, such as a token or a semantic value.

  - `children`: Represents a list of child nodes. If no children are provided, an empty list is created by default.

- `__repr__` Method:

  - This method returns a string representation of the `ASTNode` object.

  - It retrieves the name of the `type` attribute if it's an instance of an enum using `.name`, otherwise, it uses the `type` directly.

  - The returned string contains the type name, value, and children of the node, formatted for easy interpretation.

Overall, this class provides a basic structure for representing nodes in an AST, allowing for the construction of hierarchical tree-like structures commonly used in parsing and abstracting the syntactic structure of code or data.

3. Lexer Function Which Uses the Logic from the LaboratoryWorkNumber3:

Explanation:

My lexer function `lexer(code)` is designed to tokenize input code by scanning it for patterns defined by regular expressions corresponding to different token types. Below I will tell you have the lexer works:

- Input:

  - `code`: A string representing the code to be tokenized.

- Output:

  - `tokens`: A list of tuples, where each tuple contains a token type and its corresponding value extracted from the input code.

- Functionality:

  - The function initializes an empty list `tokens` to store the tokenized output.

  - It enters a `while` loop that continues until there is no more code left to tokenize.

  - Within each iteration, the leading and trailing whitespace characters of the `code` are stripped.

  - It iterates over each `(token_type, token_regex)` tuple in the `TOKEN_TYPES` list, where `TOKEN_TYPES` likely contains predefined regular expressions for different token types.

  - For each token type, a regular expression (`regex`) is compiled based on the associated token regex pattern.

  - The `regex` is then matched against the stripped `code`.

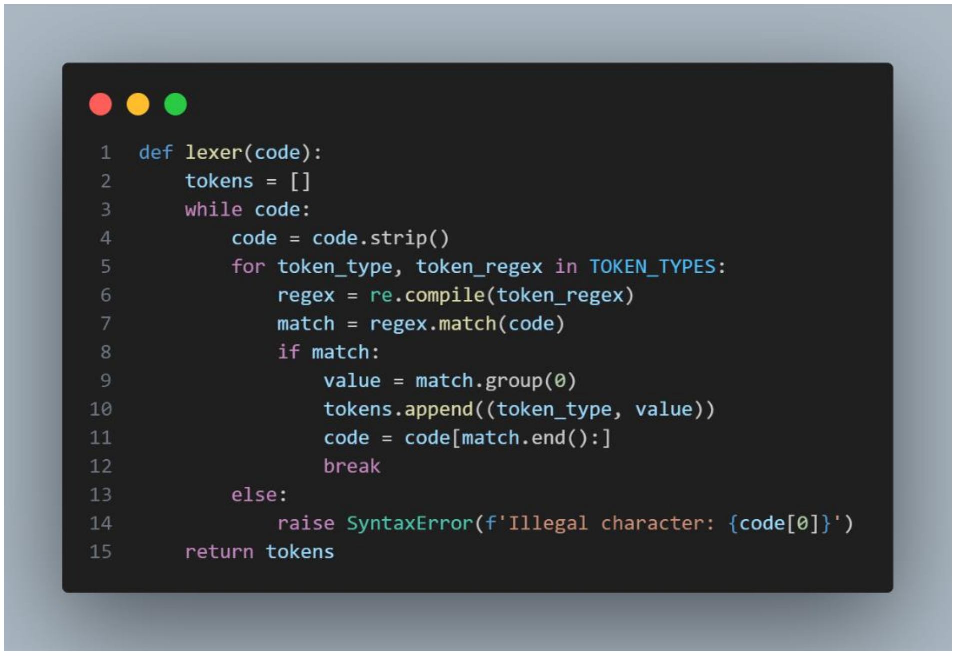
  - If a match is found (`match`), the matched value (`value`) is extracted using `match.group(0)` and appended to the `tokens` list along with its token type.

  - The portion of the `code` corresponding to the matched token is removed using slicing (`code[match.end():]`).

  - If no match is found for any token type, it raises a `SyntaxError` indicating the presence of an illegal character at the beginning of the `code`.

  - The process continues until all code is tokenized, and the function returns the list of tokens.

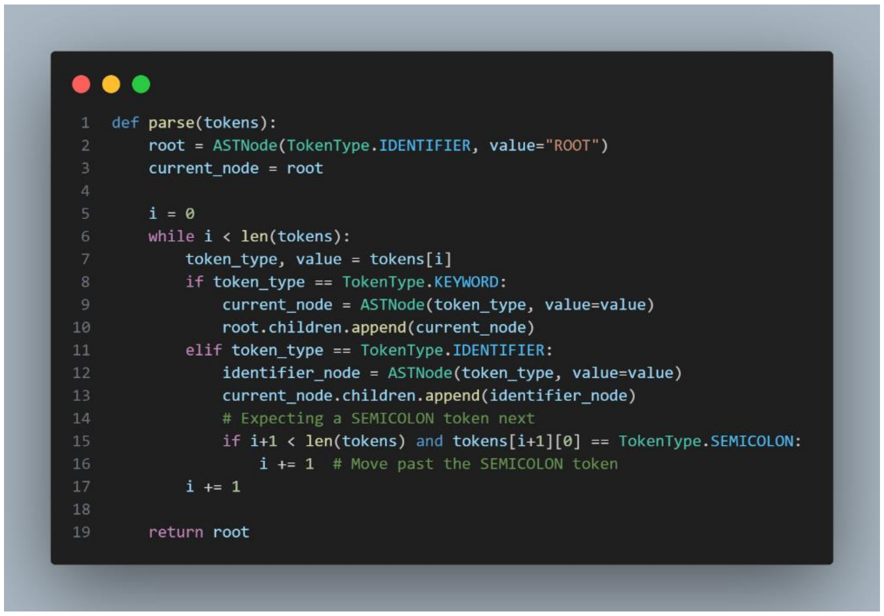
To put it all in a nutshell, this function essentially performs lexical analysis by breaking down input code into meaningful tokens based on predefined patterns, allowing for further processing in subsequent stages of parsing or interpretation.



```python
def lexer(code):
    tokens = []
    while code:
        code = code.strip()
        for token_type, token_regex in TOKEN_TYPES:
            regex = re.compile(token_regex)
            match = regex.match(code)
            if match:
                value = match.group(0)
                tokens.append((token_type, value))
                code = code[match.end():]
                break
        else:
            raise SyntaxError(f'Illegal character: {code[0]}')
    return tokens
```

Figure 4. *Lexer Code*

4. The Parse Function:



```python
def parse(tokens):
    root = ASTNode(TokenType.IDENTIFIER, value="ROOT")
    current_node = root

    i = 0
    while i < len(tokens):
        token_type, value = tokens[i]
        if token_type == TokenType.KEYWORD:
            current_node = ASTNode(token_type, value=value)
            root.children.append(current_node)
        elif token_type == TokenType.IDENTIFIER:
            identifier_node = ASTNode(token_type, value=value)
            current_node.children.append(identifier_node)
            # Expecting a SEMICOLON token next
            if i+1 < len(tokens) and tokens[i+1][0] == TokenType.SEMICOLON:
                i += 1  # Move past the SEMICOLON token
        i += 1

    return root
```

Figure 5. *Parse Function Code*

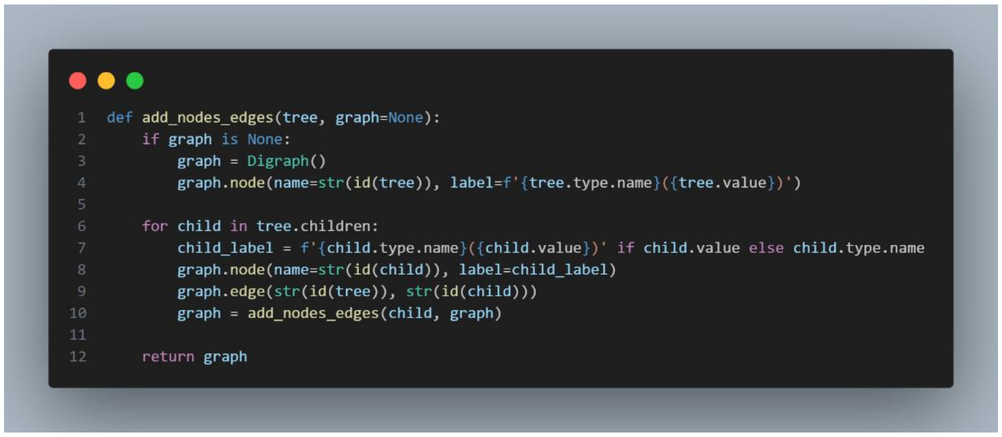
Explanation:

My `parse(tokens)` function takes a list of tokens as input and constructs an Abstract Syntax Tree (AST) based on the provided tokens. Here's a breakdown of its functionality:

- Input:
  - `tokens`: A list of token tuples, where each tuple contains a token type and its corresponding value.
- Output:
  - `root`: The root node of the constructed AST.
- Functionality:
  - It initializes the root node of the AST with the type `TokenType.IDENTIFIER` and the value "ROOT".
  - Sets `current_node` to the root node initially.
  - Enters a `while` loop that iterates over each token in the `tokens` list.
  - For each token:
    - If the token type is `TokenType.KEYWORD`, it creates a new node with the token type and value and sets it as a child of the `current_node`. This represents a new branch in the AST.
    - If the token type is `TokenType.IDENTIFIER`, it creates a new node with the token type and value and appends it as a child of the `current_node`. If the next token is a `TokenType.SEMICOLON`, it advances the token index `i` to skip the semicolon.
  - Increments the token index `i` to move to the next token.
  - Once all tokens are processed, it returns the root node of the constructed AST.

Essentially it parses the list of tokens, building an AST structure that represents the syntactic structure of the input code. It follows a simple grammar where keywords introduce new branches, and identifiers are added as children nodes.

5.The function that traverses the AST:



```python
def add_nodes_edges(tree, graph=None):
    if graph is None:
        graph = Digraph()
        graph.node(name=str(id(tree)), label=f'{tree.type.name}({tree.value})')

    for child in tree.children:
        child_label = f'{child.type.name}({child.value})' if child.value else child.type.name
        graph.node(name=str(id(child)), label=child_label)
        graph.edge(str(id(tree)), str(id(child)))
        graph = add_nodes_edges(child, graph)

    return graph
```

Figure 6. *Traversing the AST Function*

Explanation:

My `add_nodes_edges` function is responsible for traversing the Abstract Syntax Tree (AST) recursively and adding nodes and edges to a Graphviz Digraph object. Here's a breakdown of its functionality:

- Inputs:

  - `tree`: The root node of the AST to traverse.

  - `graph`: The Graphviz Digraph object to which nodes and edges are added. If not provided, a new Digraph object is created.

- Output:

  - The updated Graphviz Digraph object with nodes and edges representing the AST.

- Functionality:

  - If `graph` is not provided, a new Digraph object is created, and the root node of the tree is added as a node to the graph.

  - It then iterates over each child node of the current node in the AST.

  - For each child node:

    - If the child node has a value, its label is set to include both its type and value.

    - If the child node does not have a value, only its type is used as the label.

- A node is added to the graph for the child node, and an edge is created between the current node and the child node.

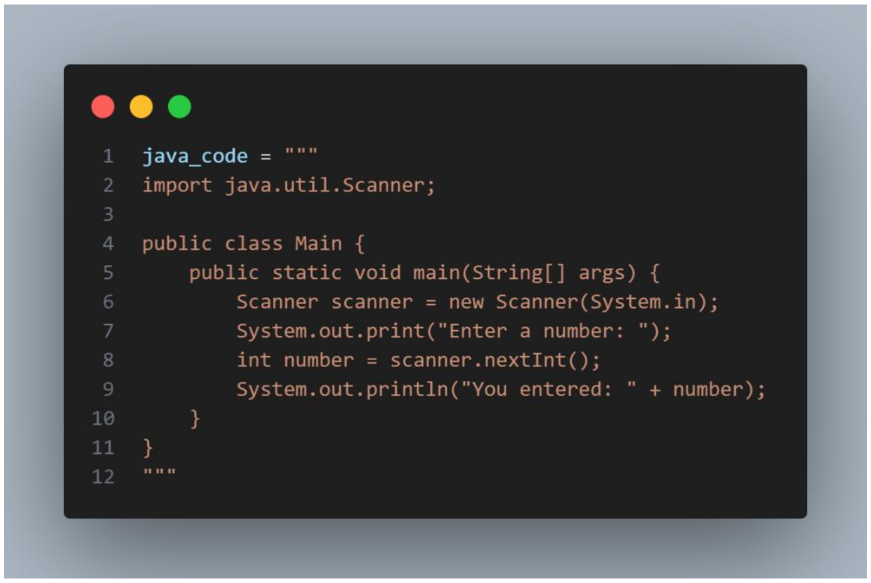
- The `add_nodes_edges` function is recursively called for the child node to traverse its children and add nodes and edges.

- The function returns the updated graph object with all nodes and edges added.

Overall, my function allows the visualization of the AST by converting it into a Graphviz Digraph, where nodes represent AST nodes and edges represent the relationships between them.

6.The JAVA Code Which I Used for Testing:

Explanation:



```
java_code = """
import java.util.Scanner;

public class Main {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter a number: ");
        int number = scanner.nextInt();
        System.out.println("You entered: " + number);
    }
}
"""
```

Figure 7. *The JAVA code which I tested*

7.Remove Inaccessible Symbols Function:

Explanation:

1. Tokenization (Lexing):

- `tokens = lexer(java_code)`: This line invokes the `lexer` function, passing the Java code (`java_code`) as input. The lexer function breaks down the Java code into tokens, which are the smallest units of meaningful characters in the code. Each token consists of a token type (e.g., KEYWORD, IDENTIFIER, etc.) and a corresponding value.

2.Parsing:

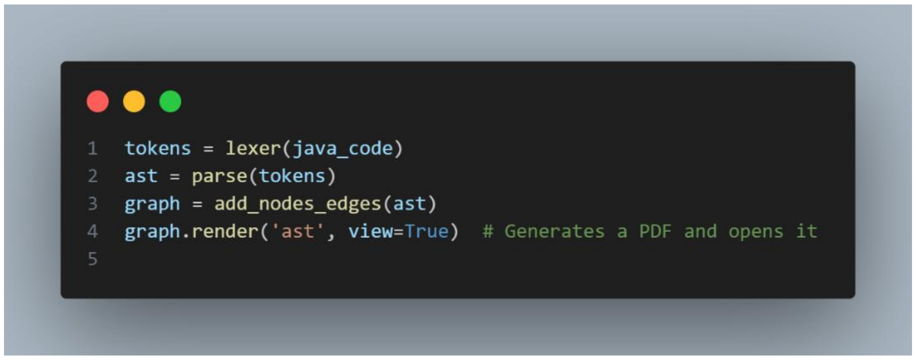
- `ast = parse(tokens)`: After obtaining the tokens from the lexer, this line calls the `parse` function, passing the tokens as input. The `parse` function constructs an Abstract Syntax Tree (AST) from the tokens. The AST represents the syntactic structure of the Java code in a hierarchical manner, where each node in the tree corresponds to a syntactic construct (e.g., keyword, identifier, etc.).

3. Graph Generation:

- `graph = add_nodes_edges(ast)`: Once the AST is constructed, this line invokes the `add_nodes_edges` function to generate a Graphviz representation of the AST. The `add_nodes_edges` function traverses the AST recursively, adding nodes and edges to a Graphviz `Digraph` object. Each node in the graph represents a node in the AST, and edges represent the hierarchical relationships between nodes.

4. Rendering and Viewing:

- `graph.render('ast', view=True)`: Finally, this line renders the Graphviz graph to a PDF file named "ast.pdf" and opens it for viewing. The `render` method generates the graphical representation of the AST using Graphviz, and setting `view=True` opens the rendered file automatically once it's generated.



```
tokens = lexer(java_code)
ast = parse(tokens)
graph = add_nodes_edges(ast)
graph.render('ast', view=True)  # Generates a PDF and opens it

```

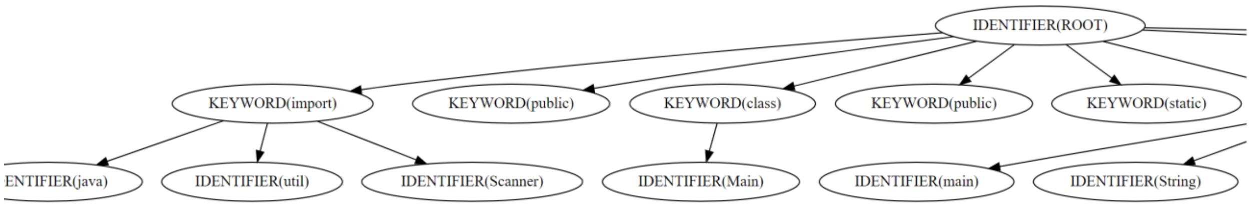
Figure 8. *Implementation of the everything above and plotting the graph*

Overall, these steps allow you to visualize the structure of the Java code as an AST, providing insights into its syntactic hierarchy and organization.

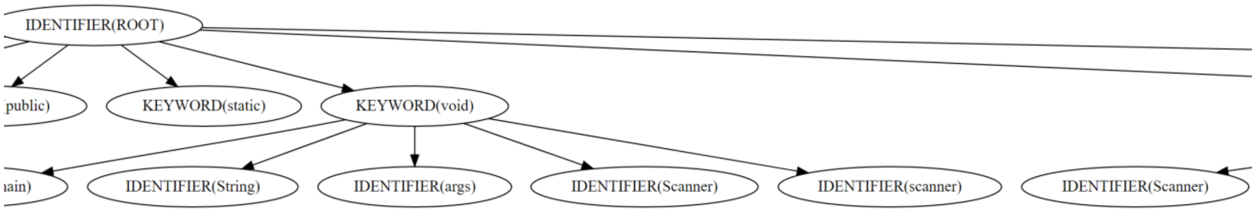
5.Conclusions/Screenshots/Results:

The Result You can See In the SVG File but you will need to zoom in because the SVG is kind of big.

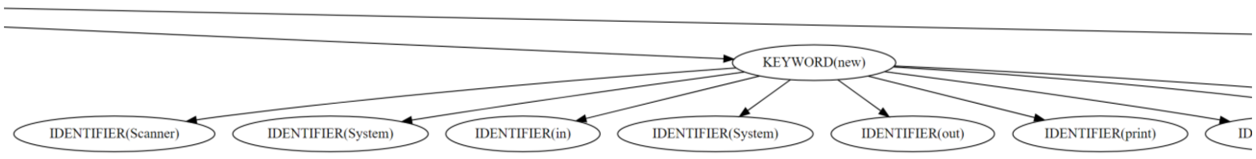
1.



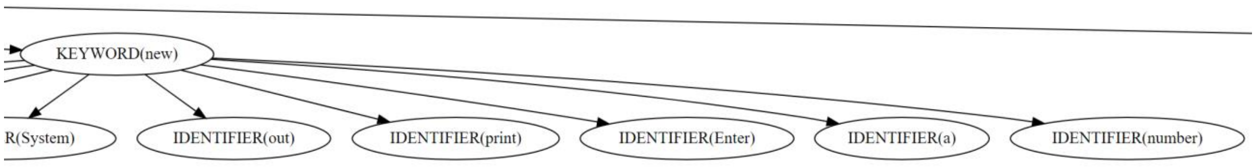
2.



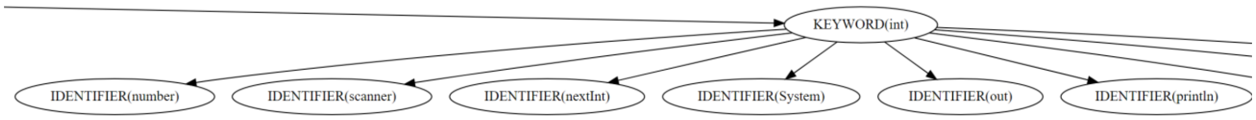
3.



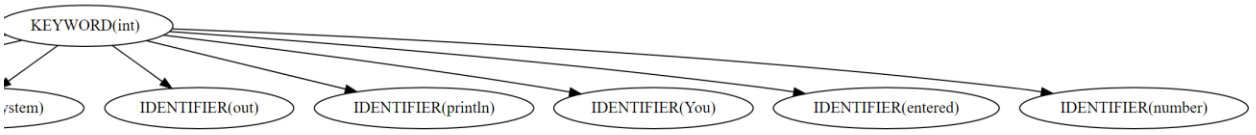
4.



5.



6.



5.Conclusions:

The recent lab work has been an eye-opening experience, significantly enhancing my understanding of parsing and Abstract Syntax Trees (ASTs), essential tools for analyzing structured text. Through the implementation of a parser and leveraging Graphviz for visualization, I was able to transform CSS text into a detailed AST, providing a clear representation of its syntactic structure. This hands-on experience not only solidified my grasp of parsing concepts but also honed my practical skills in constructing and visualizing complex data structures. Moreover, by systematically handling and categorizing textual data, I've developed crucial skills that will undoubtedly benefit my future projects in software development and computational linguistics. Overall, this lab work served as a valuable opportunity to apply theoretical knowledge to real-world scenarios, preparing me for more advanced challenges in the field as a second-year student.problems in the domain of formal languages and computational theory.

6.References:

1. https://www.javatpoint.com/automata-chomskys-normal-form

2. https://docs.python.org/3/library/re.html
3. https://cran.r-project.org/web/packages/stringr/vignettes/regular-expressions.html
4. https://en.wikipedia.org/wiki/Parsing
5. https://en.wikipedia.org/wiki/Abstract_syntax_tree