# Topic: Parser & Building an Abstract Syntax Tree

# **Course: Formal Languages & Finite Automata**

**Author: Doschinescu Dan**

**Group: FAF-222**

**Variant: 8**

**Chisinau 2024**

**1.Theory**

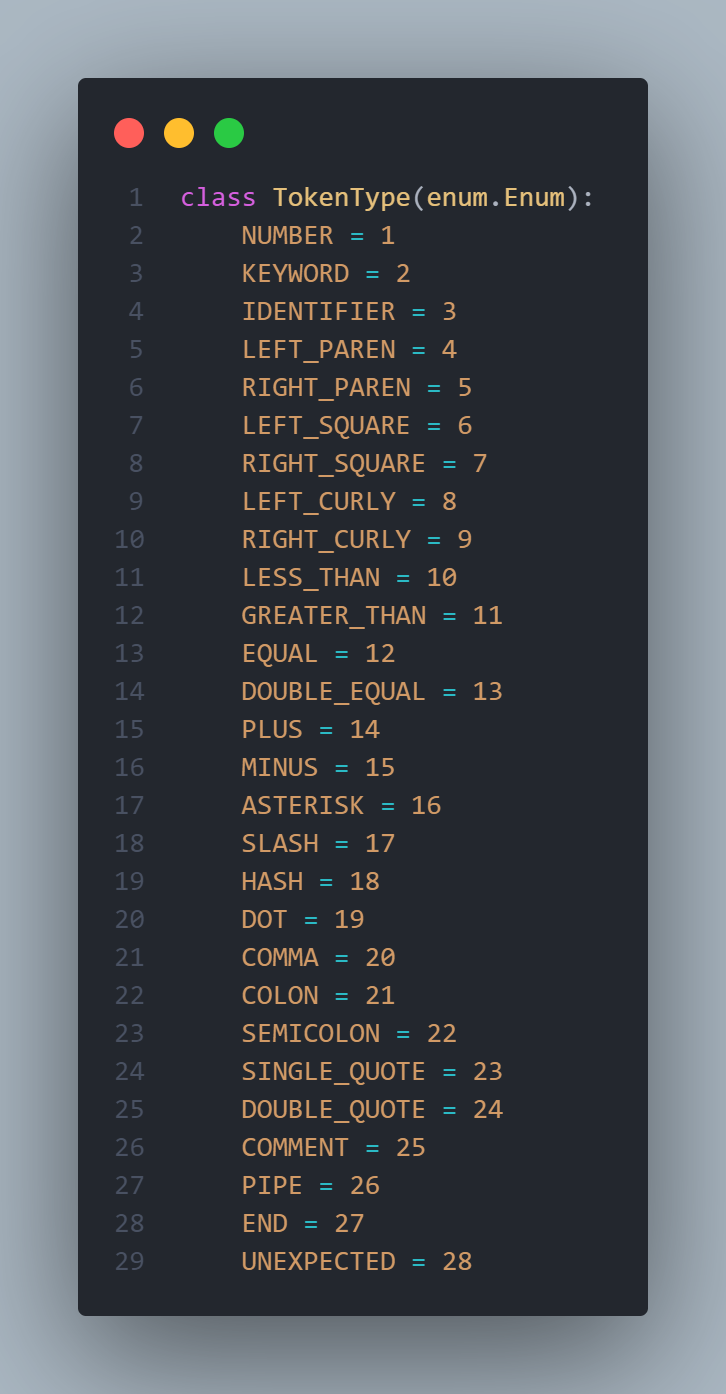
  The process of gathering syntactical meaning or doing a syntactical analysis over some text can also be called parsing. It usually results in a parse tree which can also contain semantic information that could be used in subsequent stages of compilation, for example.

    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.

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**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:
   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.

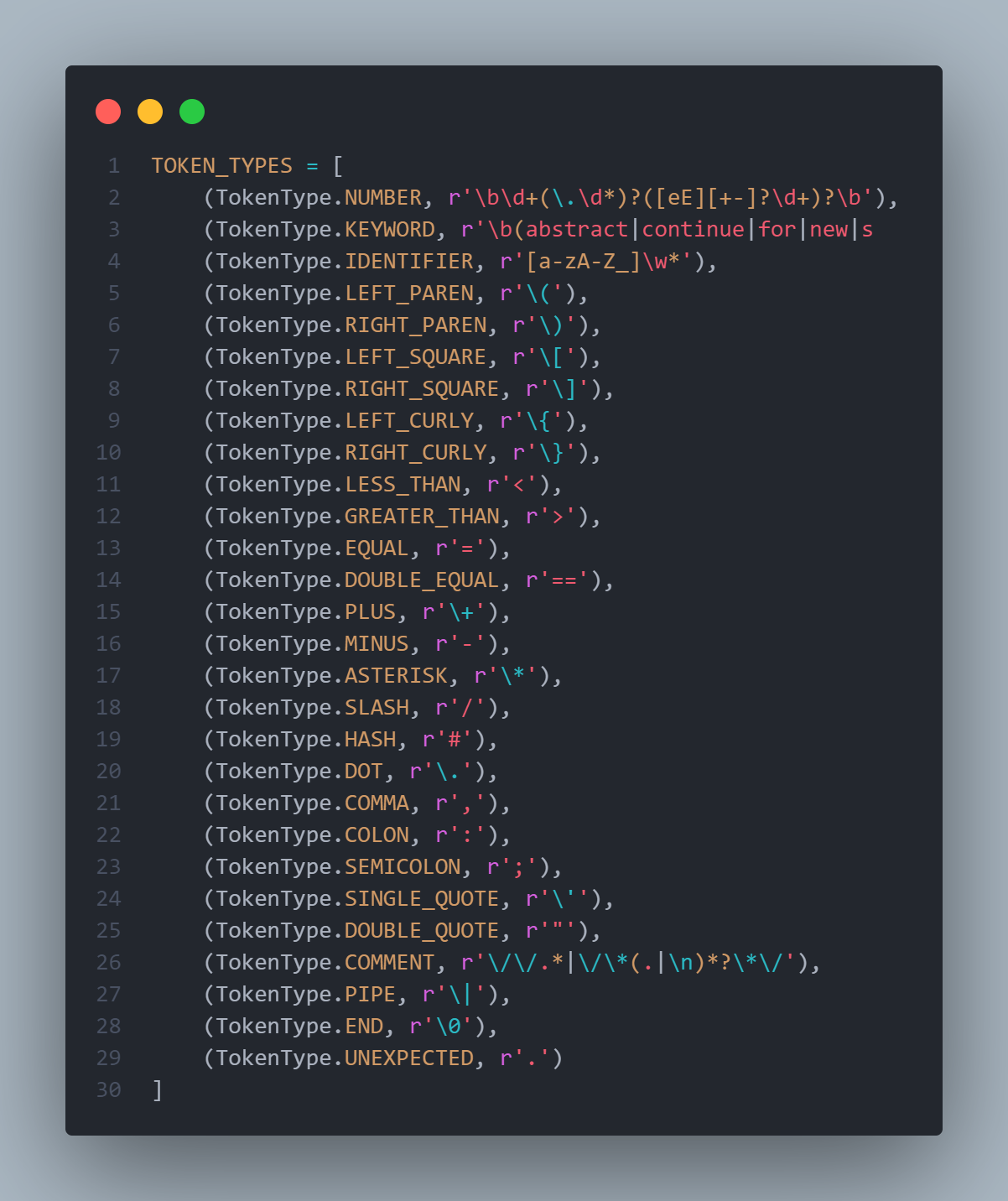


This block of code defines a Python class **TokenType** as an enumeration using the **enum** module. It enumerates various types of tokens that can occur in a programming language, such as numbers, keywords, identifiers, and different types of punctuation marks like parentheses, brackets, operators, and delimiters. Each token type is associated with an integer value for identification. The code also imports the **re** module for regular expressions and the **graphviz** module for creating graphs.

Within the **TokenType** enum, each token type is represented by a unique value. For instance, **NUMBER** is associated with the integer value 1, **KEYWORD** with 2, and so on. These token types are used to categorize different elements encountered during lexical analysis of a source code.

This enum is a part of a larger program, likely a lexical analyzer, which reads source code and tokenizes it into these different categories. It enables the program to identify and process different parts of the code based on their token types, facilitating further parsing and analysis.

The **TokenType** enum is crucial for building parsers, interpreters, or compilers for programming languages, as it provides a structured way to recognize and handle various language constructs. It simplifies the process of writing code to recognize and classify different tokens during the lexical analysis phase of language processing.

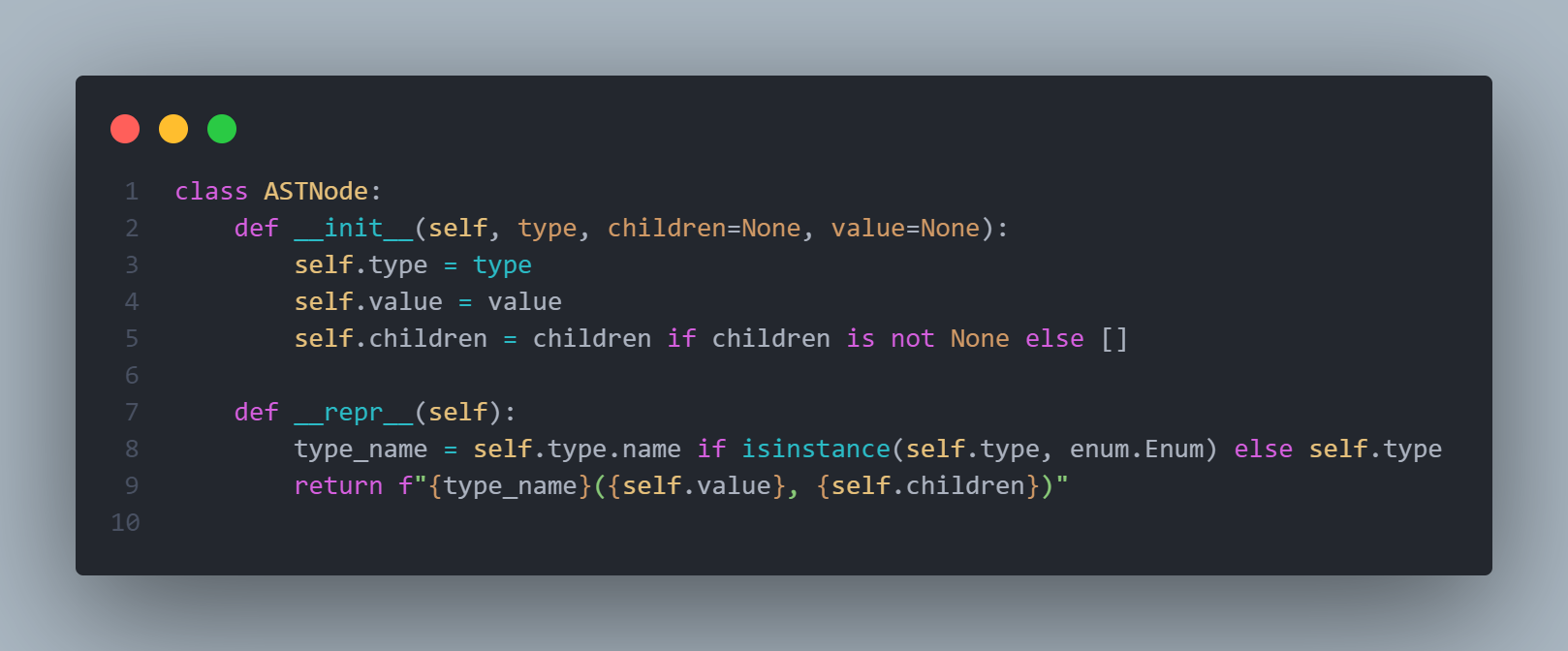


This block of code defines a list **TOKEN\_TYPES**, where each element is a tuple containing a token type and a regular expression pattern. These patterns are used to match different elements in a source code and categorize them into token types during lexical analysis. For example, the first tuple associates the token type **NUMBER** with a pattern to match numeric literals, including integers, floats, and scientific notation. Similarly, the second tuple associates **KEYWORD** token type with a pattern to match reserved keywords in a programming language.

The regular expressions in the tuples are designed to match specific language constructs. For instance, the **IDENTIFIER** token type matches valid variable or function names, starting with a letter or underscore followed by letters, digits, or underscores. Other tokens are straightforward, such as parentheses, brackets, operators, and delimiters, each matched by their respective patterns.

The **COMMENT** token type matches both single-line and multi-line comments, allowing the lexer to identify and ignore comments during tokenization. The **END** token type matches the end of input, and **UNEXPECTED** token type is used to catch any unexpected characters not matching any defined pattern.

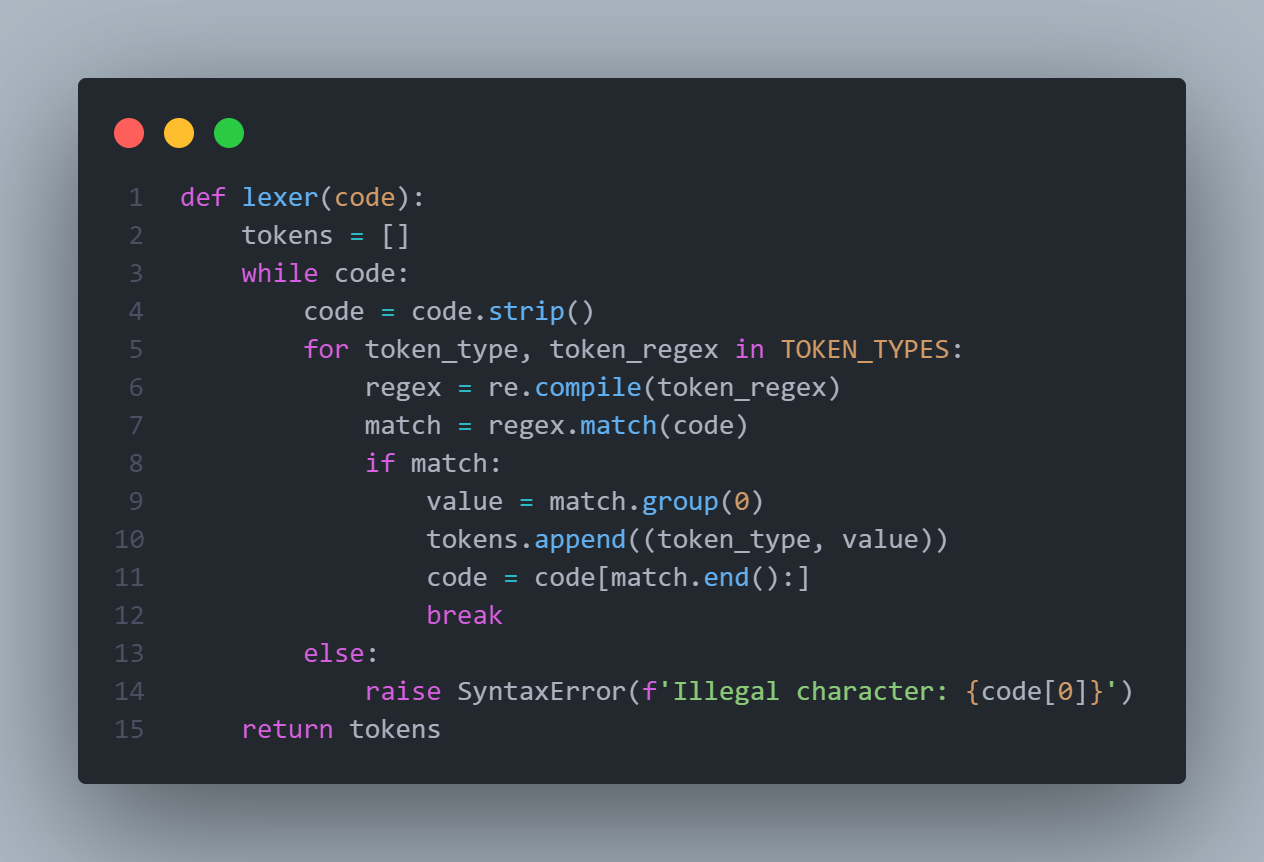
These token types and their associated patterns provide a structured way to recognize and classify different parts of a programming language, which is crucial for building lexical analyzers, parsers, or compilers. They serve as the foundation for tokenizing source code into a form that can be processed further during compilation or interpretation.



This block of code defines a Python class named **ASTNode**, which stands for Abstract Syntax Tree Node. It is used to represent nodes in an abstract syntax tree (AST), a hierarchical structure that represents the syntactic structure of code.

The class has three attributes: **type**, which represents the type of the node, **value**, which holds the value associated with the node (e.g., the value of an identifier or a literal), and **children**, which is a list holding the child nodes of the current node.

The **\_\_repr\_\_** method provides a string representation of the node, showing its type, value, and children, making it easier to visualize the structure of the AST during debugging or analysis.



This function **lexer** tokenizes the input **code** by iteratively matching its beginning with regular expressions defined in **TOKEN\_TYPES**, a list containing token types and their corresponding regular expressions.

The function iterates through each token type and its associated regex pattern, attempting to match them against the code. When a match is found, it extracts the matched value and appends it to the list of tokens along with its token type.

If no match is found for the beginning of the code, it raises a **SyntaxError** with the illegal character. The function continues this process until the entire code is tokenized.

Overall, this function is a simple lexical analyzer, splitting the code into tokens based on predefined token types and their patterns.



This `parse` function takes a list of tokens as input and constructs an Abstract Syntax Tree (AST) from those tokens. It initializes the root of the AST with the token type `IDENTIFIER` and value "ROOT". Then, it iterates through the tokens, one by one.

For each token, if it's a `KEYWORD`, it creates a new node with the same token type and appends it to the children of the current node (starting with the root).

If the token is an `IDENTIFIER`, it creates an `IDENTIFIER` node with the token's value and adds it to the children of the current node.

Additionally, it expects a `SEMICOLON` token after an `IDENTIFIER`, so if the next token is a `SEMICOLON`, it advances the token index to skip over it.

This function continues until all tokens are processed. It returns the root of the constructed AST, representing the hierarchical structure of the code where keywords are at the top level and identifiers are nested within them, separated by semicolons.

Overall, this function builds a simple AST representing a basic program structure from a sequence of tokens, with nodes corresponding to keywords and identifiers.



This function, `add\_nodes\_edges`, is designed to visualize a tree structure using the Graphviz library. It takes a tree structure (presumably an AST represented by `ASTNode` instances) and constructs a graph with nodes and edges representing the tree's structure.

The function first checks if a graph is provided. If not, it creates a new Digraph object from Graphviz and adds a node representing the current tree node, labeled with the node's type and value.

Then, it iterates over the children of the current tree node. For each child, it creates a node in the graph with a label showing its type and value. It also adds an edge between the current node and its child.

Recursively, the function calls itself on each child node to add their children and edges to the graph.

The function returns the graph object, which now contains nodes and edges representing the entire tree structure.

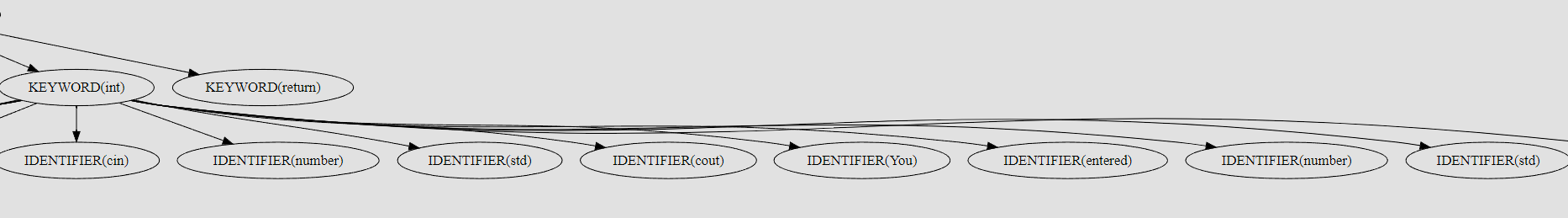
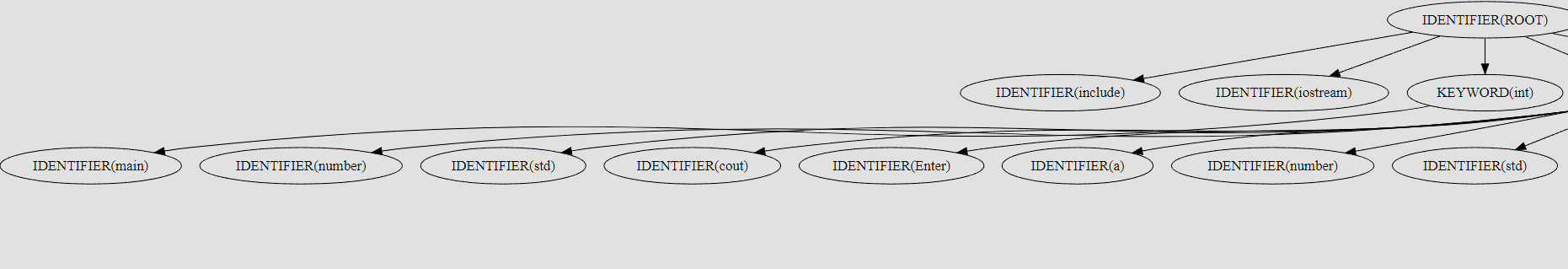
Overall, this function serves to visualize tree structures, enabling developers to understand the hierarchical relationships between nodes in the tree. It's particularly useful for visualizing ASTs generated during parsing or analysis of code.



This block of code tokenizes a C++ code snippet into a list of tokens using the `lexer` function, then parses these tokens into an Abstract Syntax Tree (AST) using the `parse` function. Afterwards, it generates a graphical representation of the AST using the `add\_nodes\_edges` function, and renders this AST graph as a PDF file named 'ast', which is then opened for viewing.

In summary, this code processes a C++ code snippet, constructs an AST from it, visualizes the AST, and opens the visualization in a PDF viewer for inspection.

**Output:**



**5.Conclusions:**

Together, these blocks of code represent a process commonly found in the field of programming language processing.

First, the `lexer` function tokenizes the input code, breaking it down into individual tokens based on predefined patterns. These tokens are categorized into different types such as numbers, keywords, and identifiers.

Next, the `parse` function takes these tokens and constructs an Abstract Syntax Tree (AST) representing the structure of the code. It organizes the tokens into a hierarchical tree where keywords are at the top level and identifiers are nested within them, separated by semicolons.

The `add\_nodes\_edges` function visualizes this AST using the Graphviz library, creating a graphical representation of the code's structure with nodes representing AST nodes and edges representing the relationships between them.

Finally, by using these functions on a sample C++ code snippet, the entire process from tokenization to AST visualization is demonstrated. The C++ code is tokenized, parsed into an AST, and then visualized as a graphical representation.

This process is crucial in building compilers, interpreters, and other language processing tools. Tokenization breaks down code into its fundamental components, parsing organizes these components into a structured format, and visualization helps developers understand the code's structure for debugging and analysis.

Overall, these blocks of code illustrate a fundamental aspect of programming language processing, showcasing how code can be transformed from raw text into a structured representation and visually analyzed to gain insights into its structure and behavior.

1. <https://www.geeksforgeeks.org/write-regular-expressions/>
2. <https://coderpad.io/blog/development/the-complete-guide-to-regular-expressions-regex/>
3. <https://docs.python.org/3/library/re.html>
4. <https://cran.r-project.org/web/packages/stringr/vignettes/regular-expressions.html>