# Topic: Lexer & Scanner

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

**Author: Doschinescu Dan**

**Group: FAF-222**

**Variant: 8**

**Chisinau 2024**

**Theory**

Once upon a time, in the realm of programming languages and structured texts, there existed a crucial entity known as the lexical analyzer, or lexer for short. Its role was pivotal in the grand scheme of interpreting and compiling code, transforming the raw tapestry of characters into a structured narrative comprehensible to machines.

At the heart of this process lay tokenization, wherein the lexer meticulously dissected the input text into bite-sized morsels known as tokens. These tokens, akin to the building blocks of a language, encompassed everything from keywords and identifiers to literals and operators.

Guided by the principles of regular languages, the lexer danced through the labyrinth of characters, employing intricate patterns and rules to discern one token from another. Like a seasoned traveler, it traversed through states, each representing a phase in the identification of various token types.

Armed with its arsenal of recognition rules, the lexer deftly handled whitespace and comments, gracefully sidestepping them as inconsequential nuances in the grand symphony of code.

Yet, amidst its graceful ballet, the lexer remained vigilant. When faced with an enigma, a sequence of characters that defied its rules, it did not falter. Instead, it raised the flag of error, signaling the presence of an anomaly, ensuring that no ambiguity remained unchecked.

And thus, with precision akin to a master craftsman, the lexer bestowed upon the world a stream of tokens, each bearing the mark of its type and the essence of its lexeme—the very soul of the matched text.

Across realms, from the domain of programming languages to the corridors of text processing, the lexer's influence knew no bounds. It was the cornerstone of compilers and interpreters, the unsung hero behind the scenes, bridging the chasm between human-readable code and machine-understandable logic.

In the realms of syntax highlighting, where colors danced upon the canvas of code editors and IDEs, the lexer's handiwork shone brightly, guiding the eyes of programmers through the labyrinth of syntax with ease.

In the end, the tale of the lexer was one of elegance and utility, a testament to the power of formal language theory and the ingenuity of computer science. It stood as a guardian of structure, a weaver of coherence, forever etched in the annals of code.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Objectives**

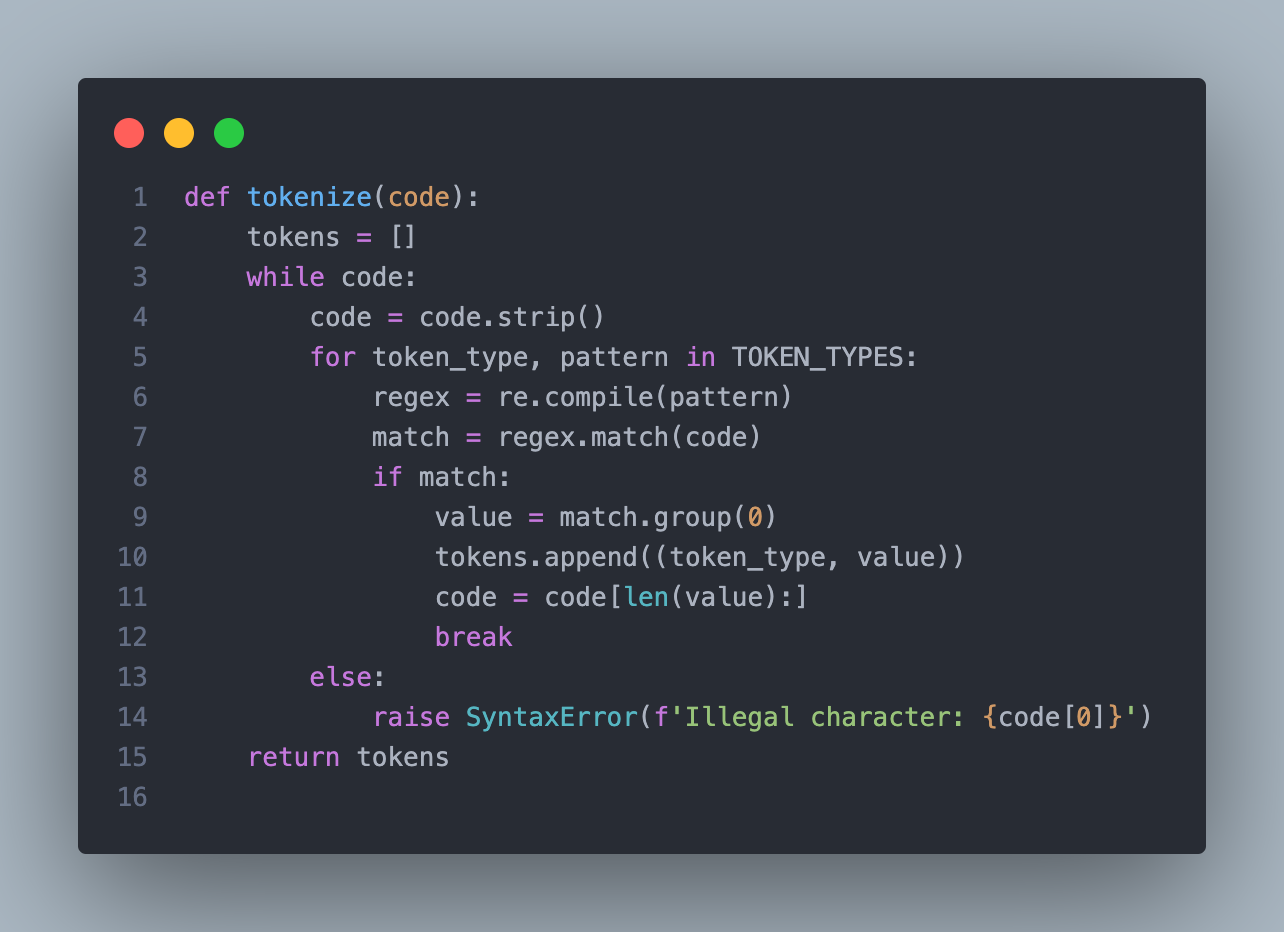
1. Understand what lexical analysis [1] is.
2. Get familiar with the inner workings of a lexer/scanner/tokenizer.
3. Implement a sample lexer and show how it works.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_



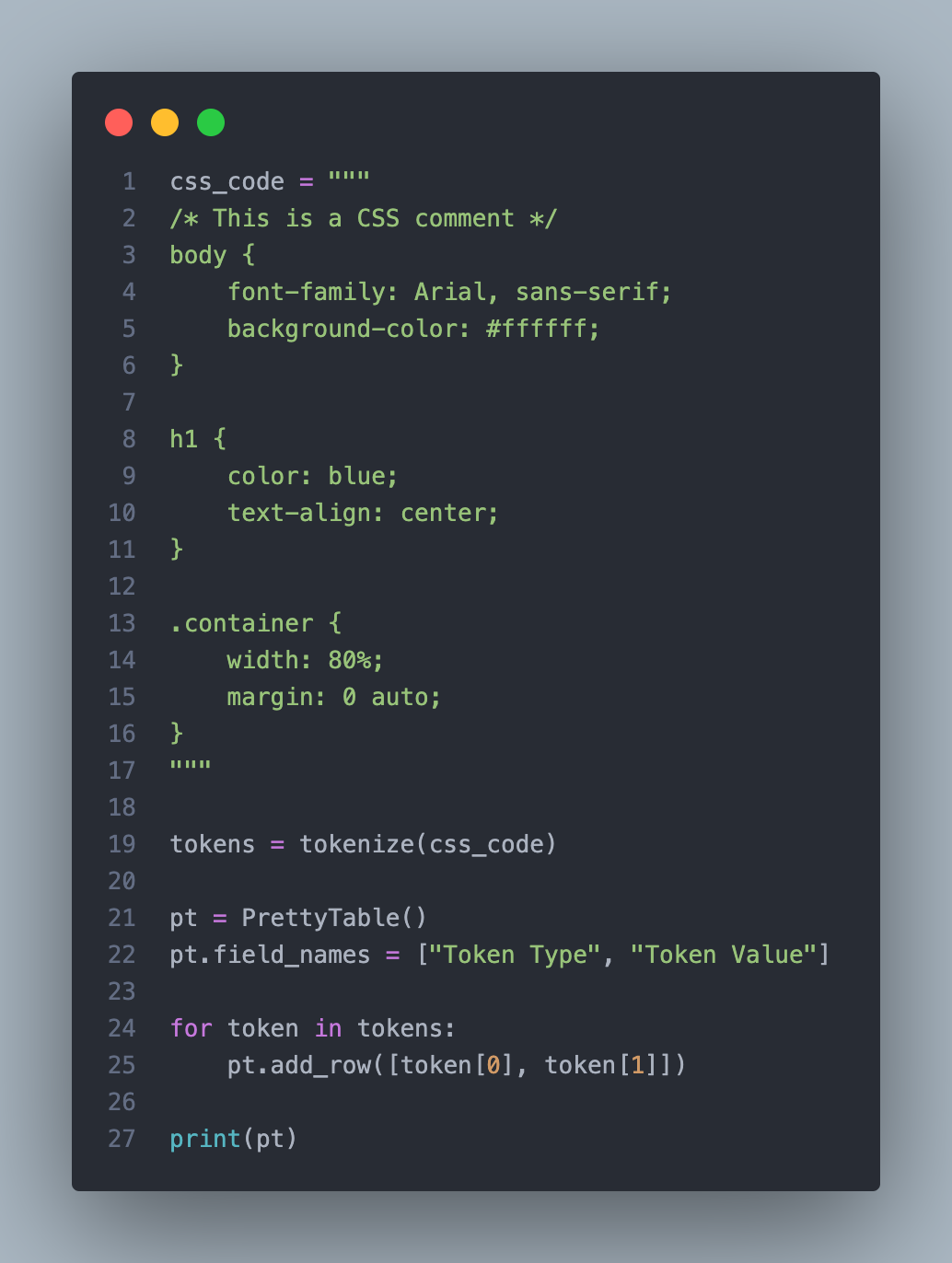
Explanation

This block of code defines a list named TOKEN\_TYPES, which contains tuples representing token types and their corresponding regular expression patterns. Each tuple consists of a token type name (e.g., 'SELECTOR', 'LEFT\_CURLY') and a regular expression pattern that describes the token's syntax. These token types are designed for parsing CSS-like syntax, where tokens represent elements such as selectors, curly braces, properties, colons, values, semicolons, comments, whitespace, and unexpected characters. The regular expressions match specific patterns within the input text, allowing for the identification and extraction of tokens during lexical analysis or parsing processes. This setup facilitates the development of a tokenizer or lexer for processing CSS-like code into structured data or tokens for further processing.



*Explanation*

*This tokenize function takes a string of code as input and returns a list of tokens. It iterates over the input code string in a loop, stripping any leading or trailing whitespace from it. Within each iteration, it iterates over each token type and its corresponding regular expression pattern defined in TOKEN\_TYPES. It attempts to match the current code string against each pattern using regular expressions. If a match is found, it extracts the matched value, appends it to the list of tokens along with its token type, and removes the matched portion from the code string. If no match is found for the current code string, it raises a SyntaxError indicating an illegal character. This function effectively breaks down the input code string into a sequence of tokens based on predefined token types and their patterns, facilitating further parsing or analysis of the code.*



**Explanation:**

This block of code processes a multiline string css\_code, which contains CSS-like code. It utilizes the tokenize function defined earlier to break down the CSS code into tokens. The resulting tokens are stored in a list. Then, it uses the PrettyTable library to create a table for displaying the token types and their corresponding values. It iterates over the list of tokens, adding each token's type and value as a row to the table. Finally, it prints the table, presenting a structured representation of the CSS code with its individual tokens categorized by type. This approach provides a clear and organized view of the lexical elements present in the CSS-like code, aiding in understanding and further processing the code.

**Output:**



**4.Conclusions:**

Throughout this lab session, I deepened my comprehension of lexers, which are fundamental in the compilation process. I explored the intricacies of regular expressions and their pivotal role in establishing syntactic rules for tokenizing programming languages and structured texts. This practical experience not only strengthened my grasp of regular expressions but also showcased their versatility in lexical analysis.

Creating a lexer from the ground up allowed me to translate theoretical concepts into real-world applications, bridging the abstract principles of computer science with software development. I encountered the challenges of designing a lexer capable of accurately parsing complex code into meaningful tokens, highlighting the significance of meticulous attention to detail and comprehensive testing in compiler design.

In summary, this lab session significantly reinforced my understanding of lexical analysis and its indispensable role in language processing. It furnished me with invaluable knowledge and skills that will undoubtedly prove beneficial in my future pursuits in computer science and programming.

**References:**

1.[Formal Languages (princeton.edu)](https://introcs.cs.princeton.edu/java/51language/)

2. [parsing - lexers vs parsers - Stack Overflow](https://stackoverflow.com/questions/2842809/lexers-vs-parsers)

3. [cs.cornell.edu/courses/cs4120/2022sp/notes/lexing/](https://www.cs.cornell.edu/courses/cs4120/2022sp/notes/lexing/)

4. [Lexical Analysis with ANTLR v4 · Jay Lim (imjching.com)](https://imjching.com/writings/2017/02/16/lexical-analysis-with-antlr-v4/)