# Topic: Lexer & Scanner

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

**Author: Durbailo Daniel**

**Group: FAF-222**

**Variant: 9**

**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.

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

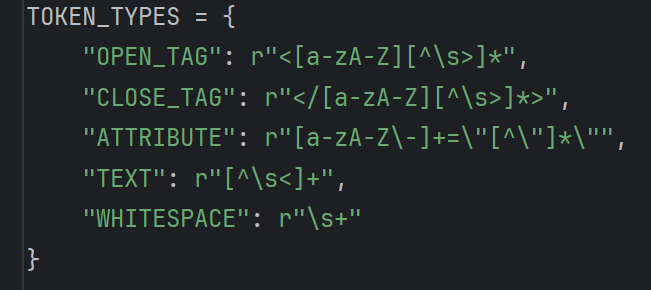
**Implementation:**

**1.Description:** For the following laboratory work I made a lexer for recognizing the types of the tokens (like numbers, keywords, dots, hashes, minuses, pluses, end tags, starting tags, identifiers) for the java language and for HTML. This means that I will show you the code for both cases in a table using the library prettytable.

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

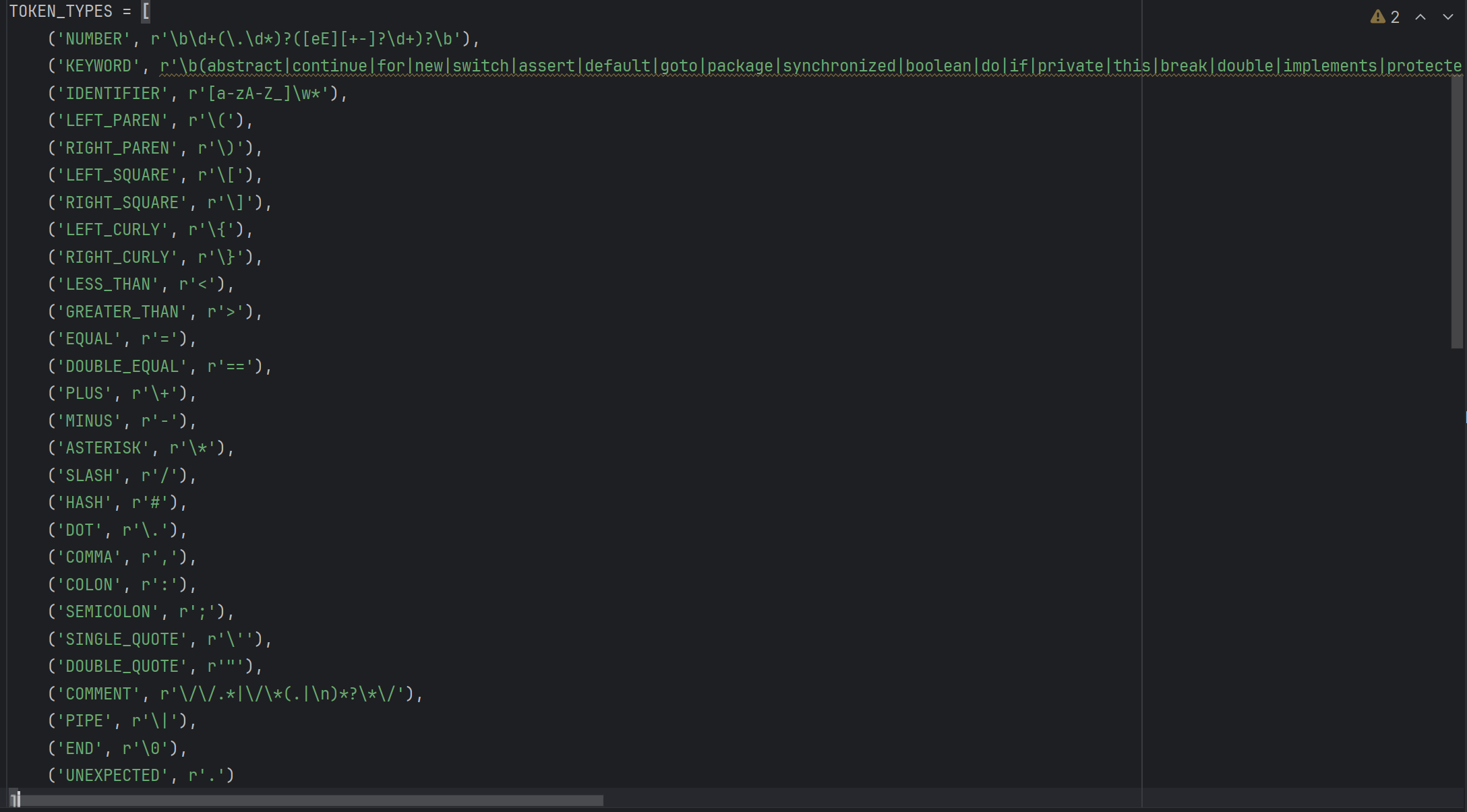
**2.Token Types:**

**HTML:**

****

*Figure 1. Tokens in HTML*

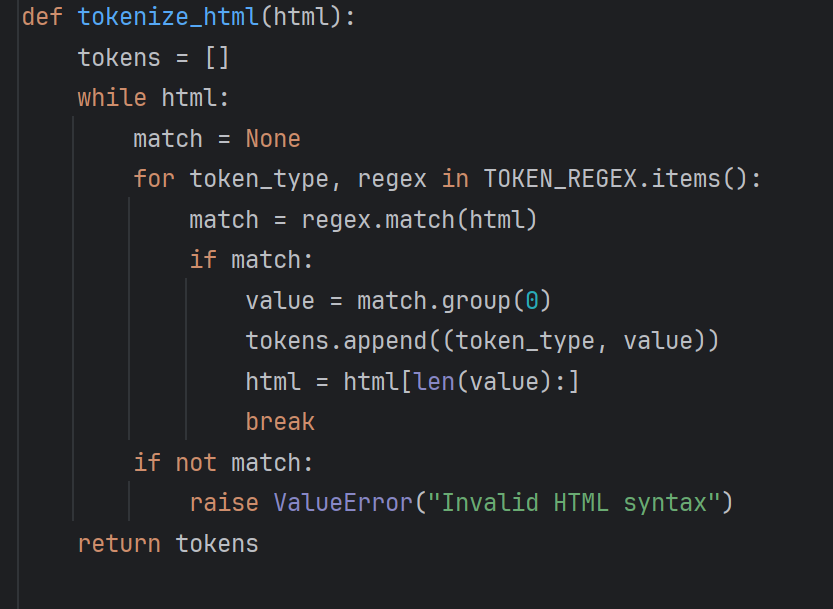
**Java:**



*Figure 2. Tokens in Java*

**3.Tokenization:**

The tokenize function serves as the cornerstone for breaking down a code string into distinct tokens according to predefined rules. It plays a vital role within a lexer, marking the initial stage in the compilation or interpretation journey of programming languages. Here's an elaborate explanation:



*Figure 3. Tokenization for HTML*

**1. Input:** The `tokenize\_html` function takes a string `html` containing HTML code as input.

**2. Initialization:** It initializes an empty list called `tokens` to store the tokens it identifies.

**3. Looping through the HTML:** The function enters a loop that continues until there's no more HTML left to tokenize.

**4. Pattern Matching:** Inside the loop, it iterates through each token type and its corresponding regular expression pattern stored in `TOKEN\_REGEX`

**5. Regular Expression Matching:** For each token type, it attempts to match the regular expression pattern against the HTML using `regex.match(html)`.

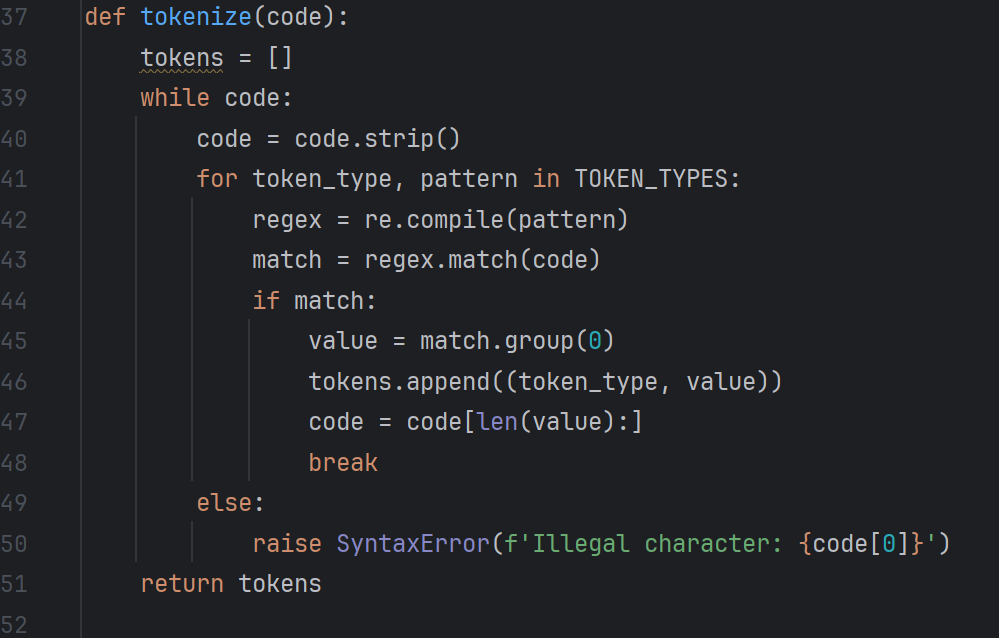
**6. Identifying Tokens:** If there's a match, it extracts the matched text (the token) using `match.group(0)` and appends it to the `tokens` list along with its token type.

**7. Updating the HTML:** It updates the HTML by removing the matched part (the token) using `html = html[len(value):]`.

**8. Handling Unrecognized HTML:** If none of the patterns match the beginning of the HTML, it raises a `ValueError` indicating that the HTML syntax is invalid.

**9. Return Tokens**: Once all the HTML is tokenized, it returns the list of tokens.

In summary, this function parses through the HTML code, matches it against predefined patterns for various token types, and breaks it down into smaller units called tokens. If it encounters any part of the HTML that doesn't conform to the expected syntax, it raises an error.



*Figure 4. Tokenization for Java*

**Input:** The tokenize function takes a piece of code (a string) as input.

**Initialization:** It initializes an empty list called tokens to store the tokens it identifies.

**Looping through the code:** The function enters a loop where it continues until there's no more code left to tokenize.

**Stripping whitespace:** Before processing, it removes any leading or trailing whitespace from the code.

**Pattern matching:** Inside the loop, the function iterates through a set of predefined token types and their corresponding patterns.

**Regular Expression Matching:** For each token type, it compiles a regular expression pattern using the re.compile() function. Then, it tries to match the pattern against the code using regex.match(code).

**Identifying Tokens:** If there's a match, it extracts the matched text (the token) using match.group(0) and appends it to the tokens list along with its token type.

**Updating the Code:** It updates the code by removing the part that was matched (the token) using code = code[len(value):].

**Handling Unrecognized Characters:** If none of the patterns match the beginning of the code, it raises a SyntaxError indicating that the character at the beginning of the code is illegal.

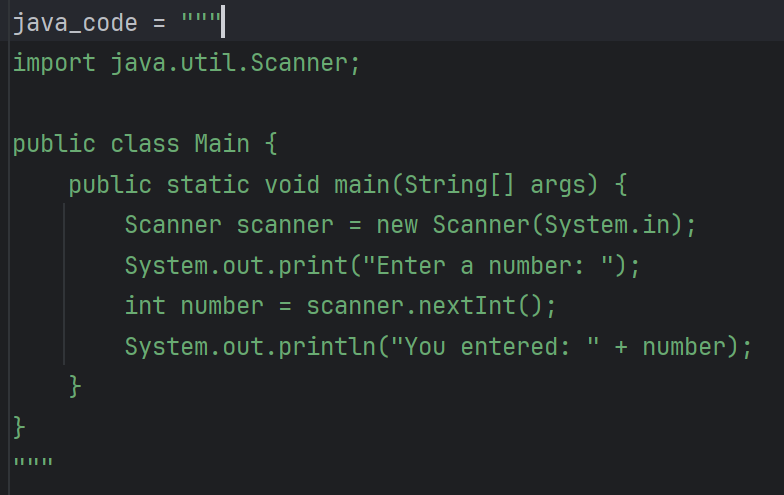
**Return Tokens:** Once all the code is tokenized, it returns the list of tokens.

In essence, this function scans through the code, looks for recognizable patterns that match predefined token types, and breaks down the code into smaller meaningful units called tokens. If it encounters any characters that don't fit any predefined pattern, it raises an error.Top of Form

**Tokenization of a Sample Code in HTML:**



*Figure 5. Sample HTML code*



*Figure 6. Sample JAVA code*

**If a match** is found: o value = match.group(0): Extracts the text that was matched by the regular expression. This is the actual string from the code that corresponds to the token. o tokens.append((token\_type, value)): Adds a tuple consisting of the token type and the matched value to the tokens list. This effectively records the token that was found. o code = code[len(value):]: Trims the matched value from the beginning of the code string. This ensures that the next iteration will start analyzing the code right after the last matched token. o break: Exits the inner for loop and goes back to the while loop to process the remaining code. The break is necessary because once a match is found, we don't want to continue trying other token types on the same segment of code.

**If no match** is found for any token type (meaning the else clause is reached): o raise SyntaxError(f'Illegal character: {code[0]}'): Raises a SyntaxError exception with a message indicating the illegal character that couldn't be matched to any token type. This is important for error handling, as it informs the user or calling function that the code contains an unexpected character.

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

**Part of the code responsible for the output:**

****

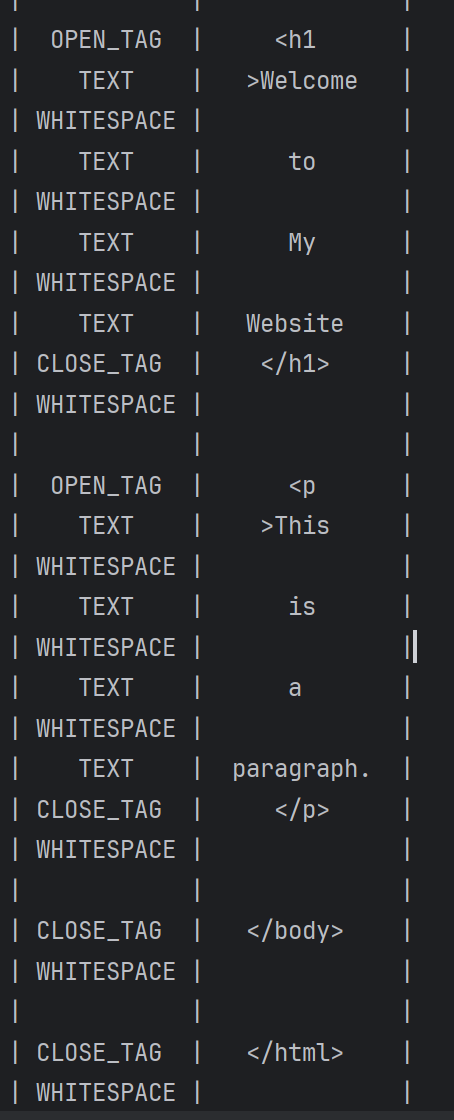
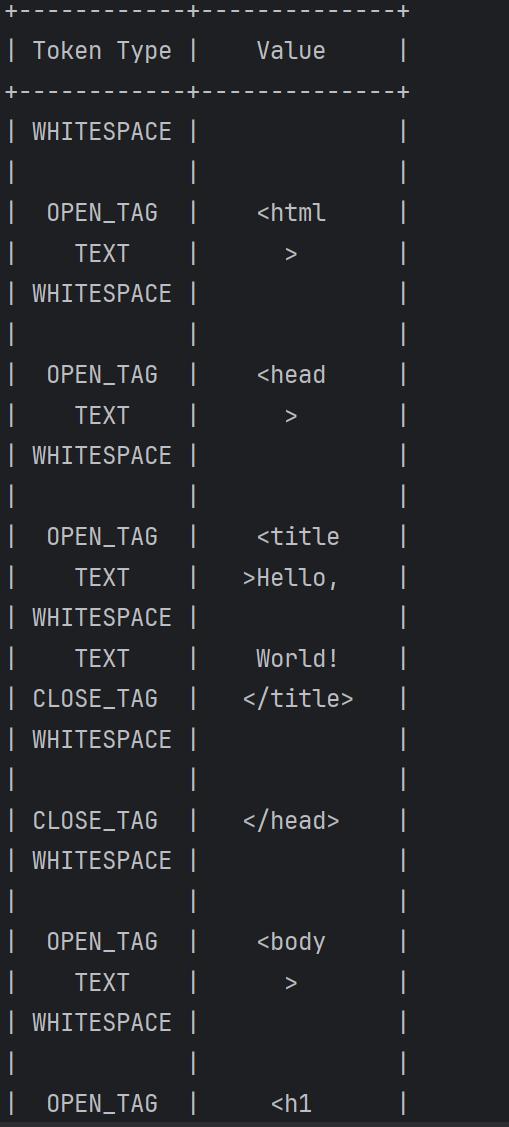
*Figure 7. Output code*

The tokens are presented in an organized manner within a table generated by PrettyTable. This table showcases two main columns: one indicating the type of token and the other displaying its corresponding value. This arrangement offers a concise and well-structured summary of the outcomes derived from the lexical analysis process.

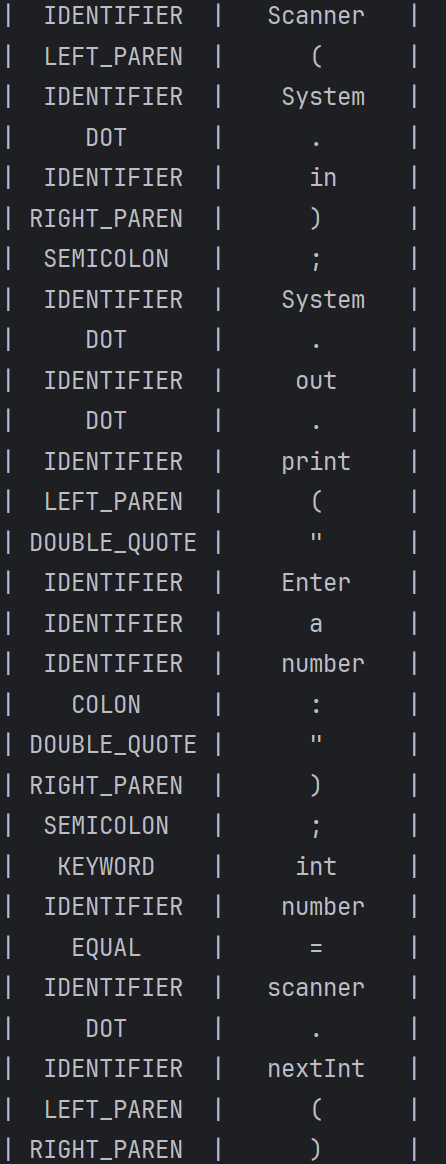
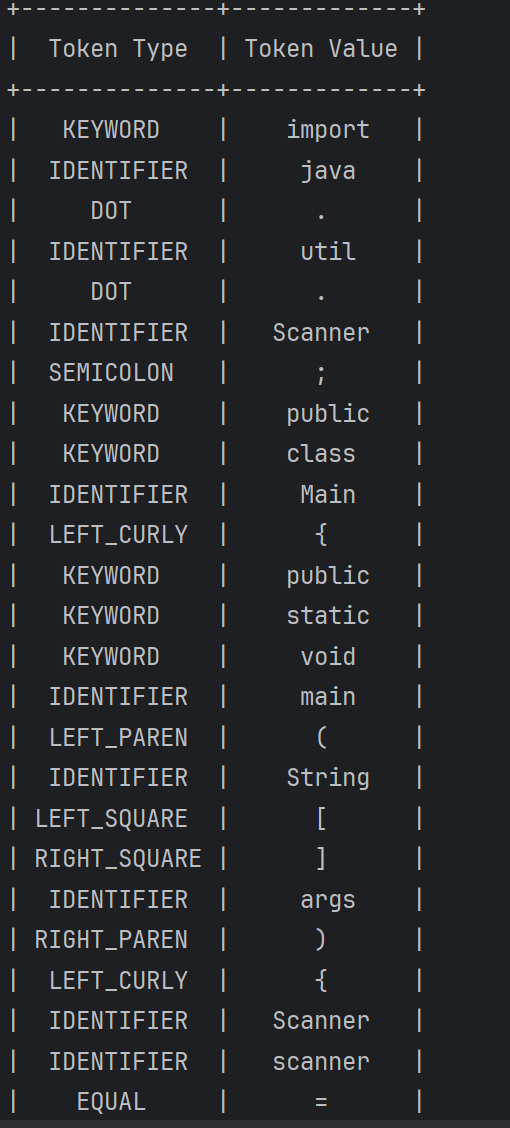
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Conclusions/Screenshots/Results:**

Here I will show you the output from the console for both Java and HTML tokenizer **(HTML FIRST):**



JAVA:



**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/)