**MINISTRY OF EDUCATION AND RESEARCH OF REPUBLIC OF MOLDOVA TECHNICAL UNIVERSITY OF MOLDOVA FACULTY OF COMPUTERS, INFORMATICS AND MICROELECTRONICS DEPARTMENT OF SOFTWARE ENGINEERING AND AUTOMATICS**

**Laboratory work 3:**

**Topic: Lexer & Scanner**

**Course: Formal Languages & Finite Automata**

**Author:Ostafi Eugen, FAF-222**

**Chișinău, 2024**

**THEORY**

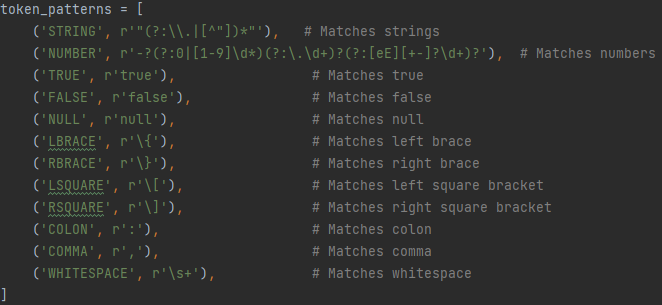
The term lexer comes from lexical analysis which, in turn, represents the process of extracting lexical tokens from a string of characters. There are several alternative names for the mechanism called lexer, for example tokenizer or scanner. The lexical analysis is one of the first stages used in a compiler/interpreter when dealing with programming, markup or other types of languages.     The tokens are identified based on some rules of the language and the products that the lexer gives are called lexemes. So basically the lexer is a stream of lexemes. Now in case it is not clear what's the difference between lexemes and tokens, there is a big one. The lexeme is just the byproduct of splitting based on delimiters, for example spaces, but the tokens give names or categories to each lexeme. So the tokens don't retain necessarily the actual value of the lexeme, but rather the type of it and maybe some metadata.

**OBJECTIVES**

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

**IMPLEMENTATION DESCRIPTION**

**1.** **Define token types**

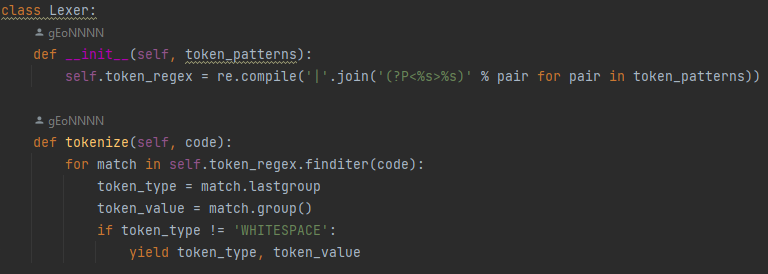


The **token\_patterns** part of the code defines a list of tuples, where each tuple contains a token type and its corresponding regular expression pattern. These token types and patterns are used to match specific elements in a JSON-like structure:

1. **STRING**: Matches text strings enclosed in double quotes. It allows for escaped characters (preceded by a backslash) and ignores unescaped double quotes inside the text.
2. **NUMBER**: Matches integers and floating-point numbers, including those with a negative sign, decimals, and scientific notation (e.g., 1e10).
3. **TRUE**: Matches the boolean value **true**.
4. **FALSE**: Matches the boolean value **false**.
5. **NULL**: Matches the null value.
6. **LBRACE**: Matches the left curly brace **{**, which typically indicates the beginning of an object in JSON.
7. **RBRACE**: Matches the right curly brace **}**, which typically indicates the end of an object in JSON.
8. **LSQUARE**: Matches the left square bracket **[**, which typically starts an array in JSON.
9. **RSQUARE**: Matches the right square bracket **]**, which typically ends an array in JSON.
10. **COLON**: Matches the colon **:**, used in JSON to separate keys from their values.
11. **COMMA**: Matches the comma **,**, used in JSON to separate elements in arrays or different fields in objects.
12. **WHITESPACE**: Matches spaces, tabs, newlines, and other whitespace characters. It's used to ignore whitespace in JSON texts during tokenization.

These patterns are used by the lexer to break down the input text into identifiable pieces, or "tokens," based on the syntax rules defined by the regular expressions. This process helps in parsing and interpreting the structure and data of JSON-like content.

**2. Lexer**



The **Lexer** class is designed for tokenizing strings based on a set of predefined token patterns, such as those for parsing JSON-like structures.

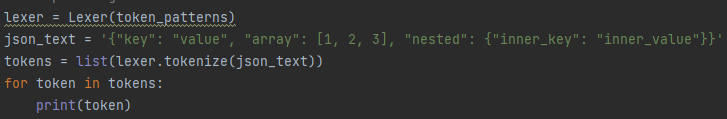
1. **init method**: This is the initializer method for the **Lexer** class. It takes a list of token patterns as its argument. Each token pattern is a tuple containing a token type and a regular expression pattern. Inside this method, a combined regular expression is created from the provided token patterns. This is achieved by joining each individual pattern into one large pattern with alternatives (separated by **|**), where each alternative is named according to its token type (**(?P<%s>%s)** format). This combined regular expression is then compiled into a **self.token\_regex** object for later use. This object can match any of the specified token types in the input string.
2. **tokenize method**: This method is used to tokenize an input string, referred to as **code**. It employs the **self.token\_regex** object created in the initializer to find all matches of the token patterns within the input string. This is done using the **finditer** method of the compiled regular expression, which iteratively finds non-overlapping matches and returns them as match objects.

For each match object found:

* + **token\_type** is set to the name of the matched pattern (thanks to the named groups used in the regular expression). This indicates the type of the current token, such as STRING, NUMBER, or COMMA.
  + **token\_value** is set to the entire string that was matched, representing the actual content of the token.

If the token type is not 'WHITESPACE' (since whitespace is typically irrelevant for the structure of the code being tokenized), the method yields a tuple containing the **token\_type** and **token\_value**. This means the **tokenize** method generates a sequence of tokens from the input string, which can be iterated over or converted into a list for further processing. This process effectively breaks down the input code into a series of identifiable pieces based on the predefined syntax rules, facilitating parsing and analysis.

**3. Example usage**



**Conclusion**

Implementation of a lexer (lexical analyzer) tailored for parsing JSON-like structures. The lexer is designed with flexibility in mind, as indicated by its ability to accept various token patterns and efficiently tokenize input strings according to these patterns.

The Lexer class uses regular expressions to define a comprehensive set of token patterns, which cover the essential components of JSON syntax such as strings, numbers, booleans, null, as well as structural characters like braces and brackets. This approach ensures that diverse JSON elements can be accurately recognized and extracted from the input text.

By employing the tokenize method, users can convert a complex JSON-like string into a sequence of more manageable tokens, facilitating further parsing or analysis. This tokenization process is crucial for understanding and manipulating the structure of JSON data, enabling developers to build more robust and efficient data processing applications.

Overall, this lexer example serves as a foundational tool for JSON parsing and highlights the importance of regular expressions in text processing. It demonstrates how a seemingly complex string can be deconstructed into identifiable elements, laying the groundwork for more advanced parsing techniques and applications in data handling and software development.