# Topic: Regular expressions

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

**Author: Doschinescu Dan**

**Group: FAF-222**

**Variant: 8**

**Chisinau 2024**

**1.Theory**

Regular languages and regular expressions are fundamental concepts in computer science, particularly in the field of formal language theory and automata theory. Here's some theory about regular languages and regular expressions framed in the narrative style you provided:

In the vast landscape of computer science, nestled within the realms of formal language theory, there exists a captivating concept known as regular languages. These languages, with their structured simplicity, serve as the bedrock of computational theory, offering a concise yet powerful framework for expressing patterns in strings.

At the heart of regular languages lies the notion of regular expressions—a language of patterns, a lexicon of symbols that bestows upon the programmer the ability to articulate intricate textual compositions with elegance and precision. Like the brushstrokes of a painter on a canvas, regular expressions weave a tapestry of characters, delineating the contours of recognizable patterns within the vast expanse of textual data.

With their arsenal of metacharacters and quantifiers, regular expressions offer a lexicon of syntactic constructs that transcend the mundane constraints of literal interpretation. Anchors tether the expression to the beginning or end of a line, while character classes and ranges provide a palette from which to select specific characters or ranges thereof. Alternation grants the freedom to choose between divergent paths, while repetition operators bestow upon the expression the power of iteration, enabling the recognition of patterns repeated ad infinitum.

Guided by the principles of Kleene's theorem, which postulates the equivalence between finite automata and regular expressions, the regular expression engine embarks on a journey through the labyrinth of characters, navigating the intricate maze of patterns with grace and precision. With each step, it traverses through states, each representing a momentary glimpse into the structure of the text, each transition a subtle revelation of the underlying pattern.

In the realm of lexical analysis, regular expressions serve as the guiding light, the cornerstone upon which lexers are built. Through their judicious application, lexers transform the raw stream of characters into a structured narrative—a symphony of tokens, each imbued with meaning and purpose, each a testament to the power of pattern recognition and formal language theory.

And thus, in the grand tapestry of computational discourse, regular languages and regular expressions stand as pillars of elegance and utility, forever entwined in the annals of computer science, forever revered as the guardians of structure and coherence in the realm of textual data.

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

**2.Objectives**

1. Write and cover what regular expressions are, what they are used for;
2. Below you will find 3 complex regular expressions per each variant. Take a variant depending on your number in the list of students and do the following:

a. Write a code that will generate valid combinations of symbols conform given regular expressions (examples will be shown).

b. In case you have an example, where symbol may be written undefined number of times, take a limit of 5 times (to evade generation of extremely long combinations);

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

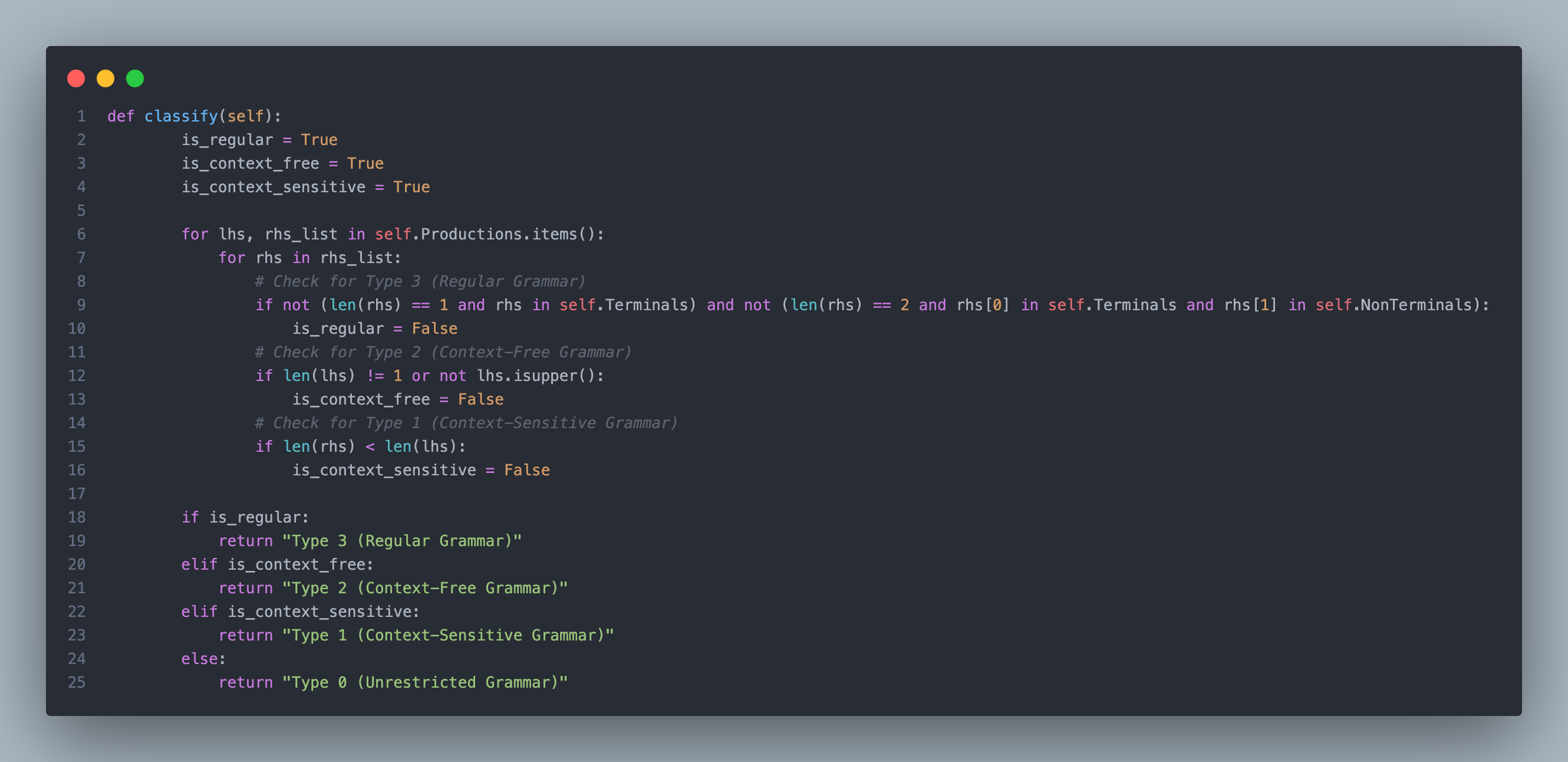
**3.Implementation:**

**1.Description:** For the following laboratory work I made a program that generates sets of valid strings based on some regular expressions.



Explanation:

The code defines a class named BaseGrammar, which represents a basic grammar structure. It has attributes for non-terminals, terminals, productions, and the start symbol of the grammar. The constructor method initializes these attributes with the values passed as parameters. Additionally, there's a method named display\_grammar() that prints out information about the grammar, including its non-terminals, terminals, productions, and start symbol. This class provides a foundational framework for working with grammars in Python, allowing users to create, store, and display grammatical structures easily. It can serve as a building block for implementing more complex grammar-related functionalities and algorithms.



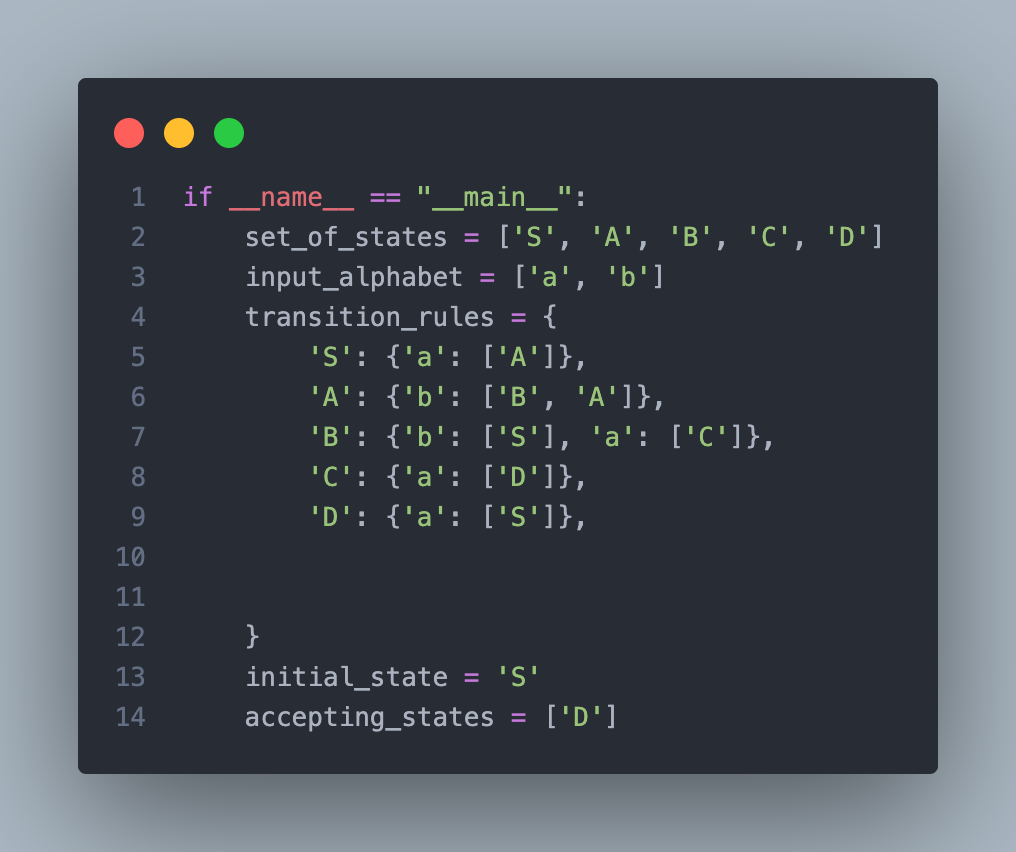
**Explanation:**

This block of code defines a method named classify within the BaseGrammar class. It evaluates the type of grammar based on its productions and structure. It iterates over each production in the grammar, checking conditions to determine its classification. For regular grammars, it verifies that productions adhere to specific patterns of terminals and non-terminals. Context-free grammars are identified by ensuring that the left-hand side of productions consists of a single uppercase letter. Context-sensitive grammars are recognized if the length of the right-hand side of productions is greater than or equal to the left-hand side. Depending on the fulfilled conditions, the method returns the type of grammar classification, ranging from Type 3 (Regular Grammar) to Type 0 (Unrestricted Grammar).



Explanation

This code defines a class named BaseAutomaton, representing a basic automaton structure. It comprises attributes such as States, Alphabet, Transitions, Initial, and Accepting, which respectively denote the set of states, input alphabet, transition rules, initial state, and accepting states of the automaton. The constructor method \_\_init\_\_ initializes these attributes with values passed as parameters. Additionally, the class includes a method named display\_automaton() that prints out information about the automaton, including its states, alphabet, transition rules, initial state, and accepting states. Overall, this class provides a foundational framework for working with automata in Python, enabling users to create, store, and display automaton structures easily. It serves as a base for implementing more complex automata-related functionalities and algorithms.



**Explanation**

This block of code serves as the main program that runs when the script is executed directly. It defines parameters for a simple automaton, including its set of states, input alphabet, transition rules, initial state, and accepting states. The set of states consists of five states labeled from 'S' to 'D', while the input alphabet comprises the characters 'a' and 'b'. Transition rules are defined as a dictionary where each state maps to another dictionary representing transitions for each input symbol. The initial state is set to 'S', and the accepting state is 'D'. This setup essentially describes a finite automaton with transitions defined for each state based on specific input symbols, aiming to recognize strings that lead to the accepting state 'D'.

**5.Conclusions:**

Throughout this lab session, I delved into the realm of lexical analysis, a cornerstone in the compilation process. My exploration centered on the intricate world of regular expressions, which serve as the bedrock for establishing syntactic rules in tokenizing programming languages and structured texts.

By creating a lexer from scratch, I translated theoretical concepts into tangible applications, bridging the abstract principles of computer science with practical software development. This hands-on experience not only deepened my understanding of regular expressions but also highlighted their versatility in lexical analysis.

Designing the lexer posed challenges, requiring meticulous attention to detail and comprehensive testing to ensure accurate parsing of complex code into meaningful tokens. This process underscored the importance of rigorous testing and iterative refinement in compiler design.

In summary, this lab session was instrumental in reinforcing my comprehension of lexical analysis and its indispensable role in language processing. It equipped me with valuable knowledge and skills that will undoubtedly be beneficial in my future endeavors in computer science and programming.

**References:**

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>