Topic: Intro to formal languages. Regular grammars. Finite Automata.

Course: Formal Languages & Finite Automata

Author: Grosu Renat

Theory:

This project delves into the core principles of formal languages and automata theory, specifically focusing on the transformation of context-free grammars (CFGs) into finite automata (FAs). This exploration is about demonstrating the computational equivalence between two fundamental concepts in computer science: CFGs, which define the syntax of formal languages through sets of recursive rules, and FAs, which are simple machines capable of recognizing patterns and sequences within strings of symbols from those languages.

A context-free grammar is composed of a set of non-terminal symbols, terminal symbols, a set of production rules, and a start symbol. Non-terminal symbols are placeholders for patterns of symbols that can include both terminals and non-terminals. Terminal symbols are the basic symbols of the language that appear in the strings generated by the grammar. Production rules define how non-terminal symbols can be replaced with combinations of terminals and other non-terminals. The start symbol kicks off the production process, eventually generating strings of the language.

Finite automata, on the other hand, are abstract models of computation that recognize whether a given input string belongs to a particular language. An FA consists of a finite number of states, including start and accept states, and a set of transitions between these states based on input symbols. The automaton processes a string of symbols from the language one symbol at a time, transitioning between states according to its transition function. If the automaton ends in an accept state after processing the entire string, the string is considered to be accepted by the automaton; otherwise, it is rejected.

The conversion of a CFG into an FA, specifically into a non-deterministic finite automaton (NFA), involves creating states that correspond to the non-terminals of the grammar and defining transition rules based on the production rules of the grammar. This process demonstrates that for every context-free language, there exists a finite automaton that recognizes it, highlighting the equivalence between these two different representations of formal languages. This theoretical foundation allows for practical applications such as parsing in compilers, where CFGs define the syntax of programming languages and FAs perform lexical analysis.

Understanding the interplay between CFGs and FAs highlights the power and limitations of automata in recognizing complex patterns and structures in data, a fundamental aspect of theoretical computer science that has implications for programming language design, compiler construction, and the broader field of computational linguistics.

Objectives:

According to your variant number, get the grammar definition and do the following:

a. Implement a type/class for your grammar;

b. Add one function that would generate 5 valid strings from the language expressed by your given grammar;

c. Implement some functionality that would convert and object of type Grammar to one of type Finite Automaton;

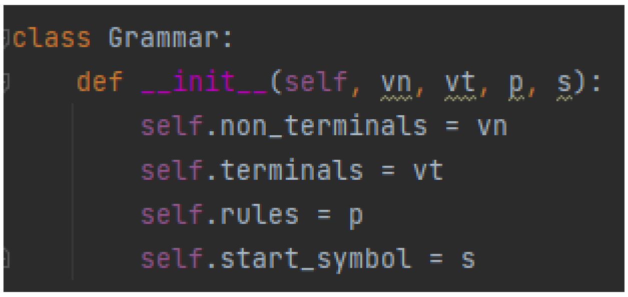
d. For the Finite Automaton, please add a method that checks if an input string can be obtained via the state transition from it;

Implementation description:

Grammar Class

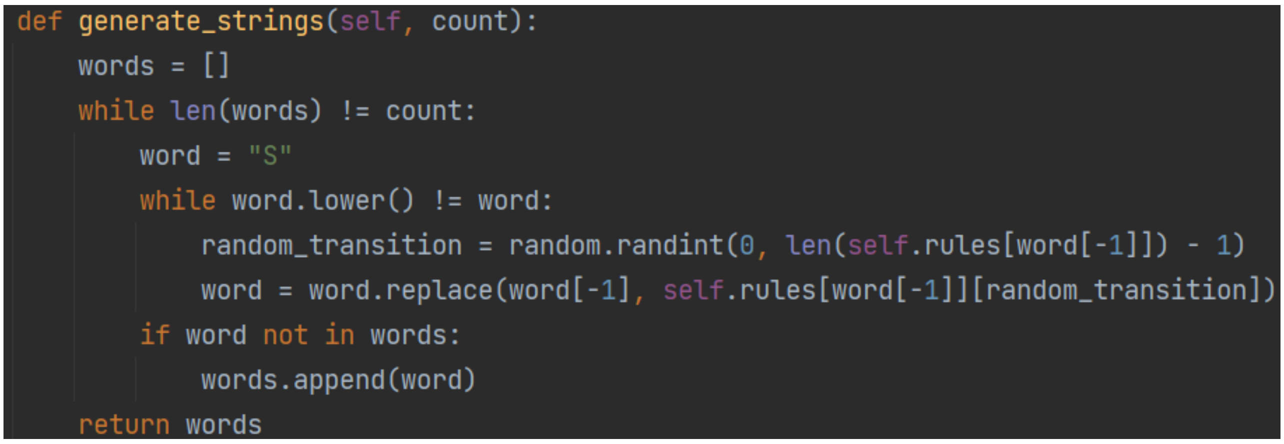
The Grammar class is central to the representation of context-free grammars within the program. It is designed to encapsulate the components of a CFG, including non-terminal symbols, terminal symbols, production rules, and the start symbol.

Initialization: The constructor of the class takes four parameters: vn (non-terminals), vt (terminals), p (production rules), and s (start symbol). These parameters are stored as properties of the class, setting up the grammar's structure.



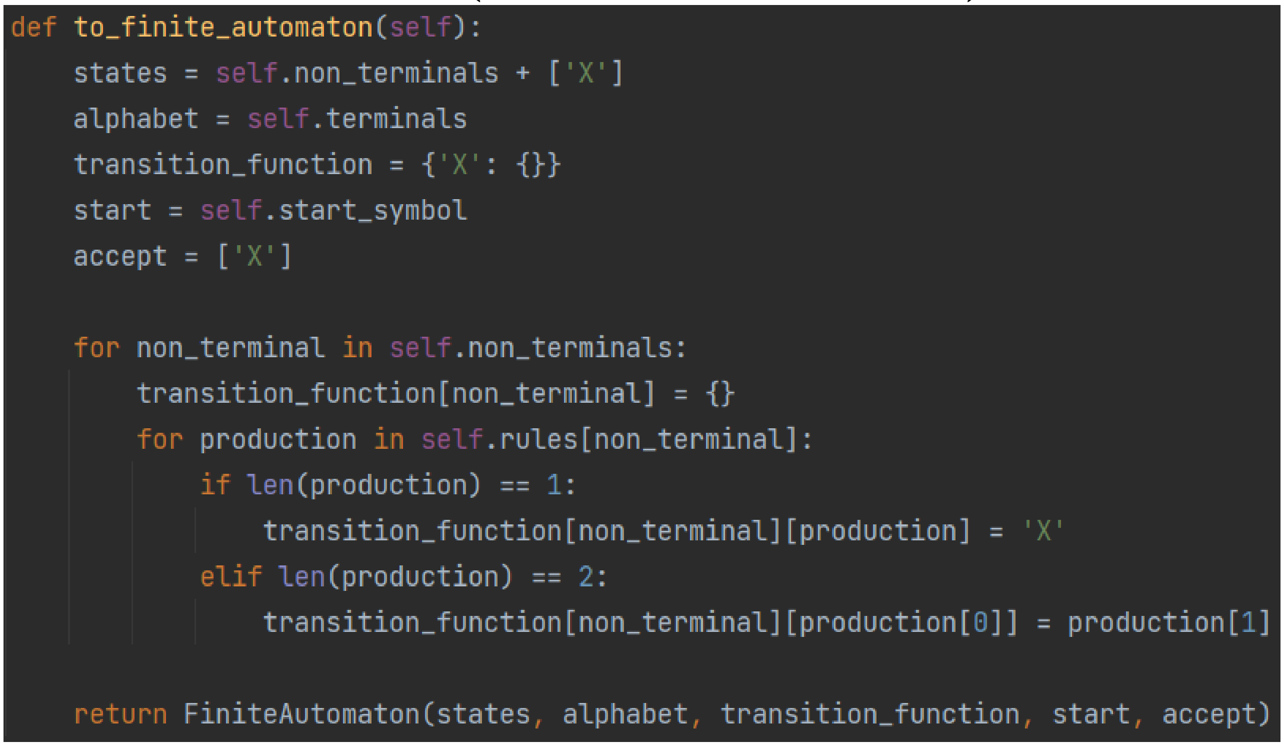
```python
class Grammar:
    def __init__(self, vn, vt, p, s):
        self.non_terminals = vn
        self.terminals = vt
        self.rules = p
        self.start_symbol = s
```

String Generation (generate_strings method): This method is responsible for generating strings that can be derived from the grammar's production rules. It starts with the start symbol and recursively replaces non-terminal symbols with their productions until only terminal symbols remain. The method aims to generate a predefined number of unique strings (in this case, five) that can be made with the given grammar. To ensure variety, it randomly selects among the possible productions for each non-terminal during the replacement process.



```python
def generate_strings(self, count):
    words = []
    while len(words) != count:
        word = "S"
        while word.lower() != word:
            random_transition = random.randint(0, len(self.rules[word[-1]]) - 1)
            word = word.replace(word[-1], self.rules[word[-1]][random_transition])
        if word not in words:
            words.append(word)
    return words
```

Conversion to Automaton (to_finite_automaton method):



```python
def to_finite_automaton(self):
    states = self.non_terminals + ['X']
    alphabet = self.terminals
    transition_function = {'X': {}}
    start = self.start_symbol
    accept = ['X']

    for non_terminal in self.non_terminals:
        transition_function[non_terminal] = {}
        for production in self.rules[non_terminal]:
            if len(production) == 1:
                transition_function[non_terminal][production] = 'X'
            elif len(production) == 2:
                transition_function[non_terminal][production[0]] = production[1]

    return FiniteAutomaton(states, alphabet, transition_function, start, accept)
```

This method translates the CFG into an equivalent finite automaton. The conversion process involves:

Creating a state for each non-terminal symbol, plus an additional "accept" state to signify successful string recognition.

Defining the transition function based on the grammar's production rules. Transitions are created from non-terminals to either other non-terminals or the accept state, depending on the structure of the production rule.

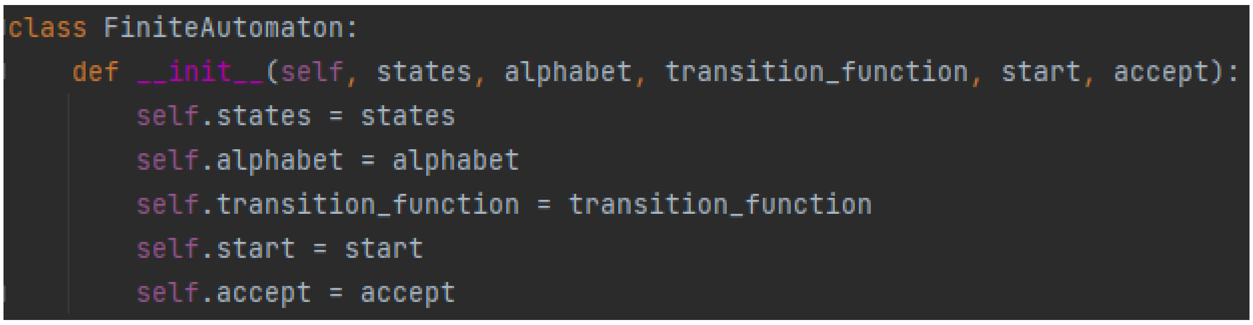
Handling productions that involve terminal symbols directly, linking states according to the input symbol's presence in the production.

This conversion shows the theoretical principle that context-free languages can be recognized by non-deterministic finite automata.

FiniteAutomaton Class

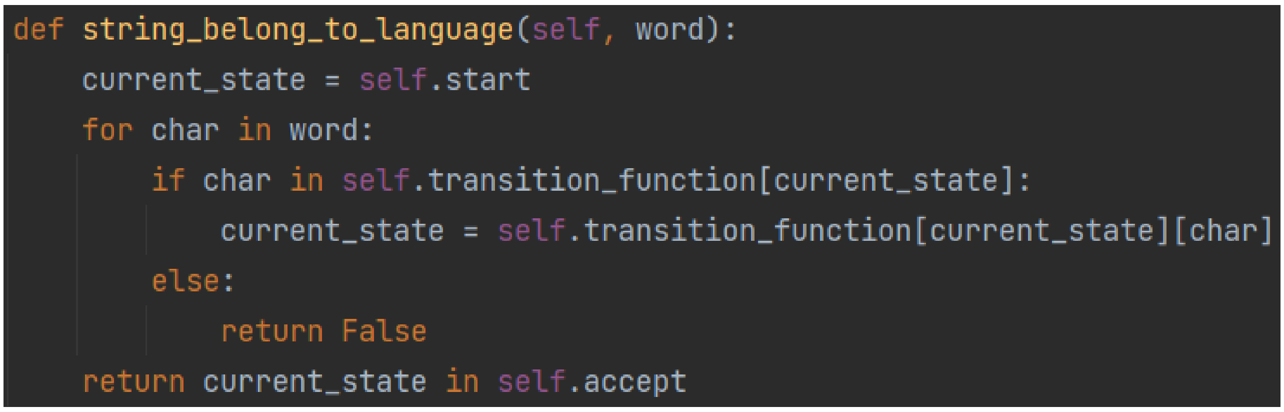
The FiniteAutomaton class represents finite automata, encapsulating states, the alphabet (terminal symbols), a transition function, a start state, and accept states.

Initialization: Similar to the Grammar class, the constructor initializes the FA with its components mentioned above.



```python
class FiniteAutomaton:
    def __init__(self, states, alphabet, transition_function, start, accept):
        self.states = states
        self.alphabet = alphabet
        self.transition_function = transition_function
        self.start = start
        self.accept = accept
```

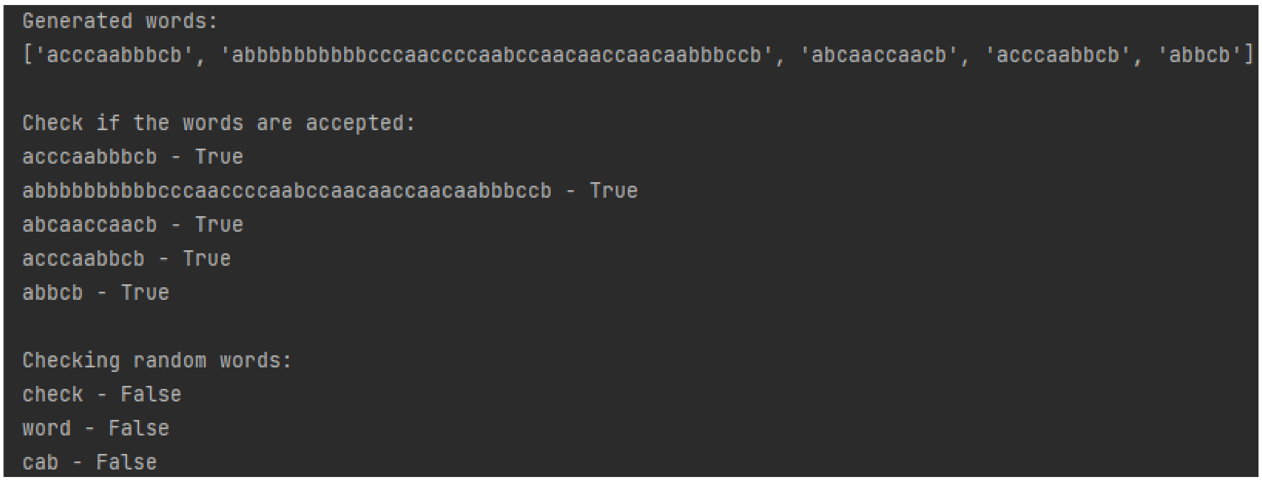
String Acceptance (string_belong_to_language method): This method determines whether a given string is accepted by the automaton. It iteratively processes each symbol of the input string, transitioning between states according to the transition function. If the automaton reaches an accept state after processing the entire string, the string is considered accepted; otherwise,it is rejected.



```python
def string_belong_to_language(self, word):
    current_state = self.start
    for char in word:
        if char in self.transition_function[current_state]:
            current_state = self.transition_function[current_state][char]
        else:
            return False
    return current_state in self.accept
```

Conclusions/Screenshots/Results:

The implemented system successfully converts a given context-free grammar into a finite automaton and tests the acceptance of strings generated from the grammar. This demonstrates the computational equivalence between CFGs and FAs and showcases the practical application of theoretical concepts in formal languages and automata theory.



```
Generated words:
['acccaabbbcb', 'abbbbbbbbbbcccaaccccaabccaacaaccaacaabbbccb', 'abcaaccaacb', 'acccaabbcb', 'abbcb']

Check if the words are accepted:
acccaabbbcb - True
abbbbbbbbbbcccaaccccaabccaacaaccaacaabbbccb - True
abcaaccaacb - True
acccaabbcb - True
abbcb - True

Checking random words:
check - False
word - False
cab - False
```

References:

https://introcs.cs.princeton.edu/java/51language/

https://www.geeksforgeeks.org/introduction-of-finite-automata/