Ministerul Educaţiei și Cercetării al Republicii Moldova
Universitatea Tehnică a Moldovei

Facultatea Calculatoare, Informatică și Microelectronică

Course: Formal Languages & Finite Automata

Laboratory work 1: Intro to formal languages. Regular grammars. Finite Automata

Elaborated:

st. gr. FAF-221 Clepa Rodion

Verified:

asist. univ. Dumitru Crețu

Chişinău – 2024

Objectives

Discover what a language is and what it needs to have in order to be considered a formal one;

Provide the initial setup for the evolving project that you will work on during this semester. You can deal with each laboratory work as a separate task or project to demonstrate your understanding of the given themes, but you also can deal with labs as stages of making your own big solution, your own project. Do the following:

a. Create GitHub repository to deal with storing and updating your project;
b. Choose a programming language. Pick one that will be easiest for dealing with your tasks, you need to learn how to solve the problem itself, not everything around the problem (like setting up the project, launching it correctly and etc.);
c. Store reports separately in a way to make verification of your work simpler (duh)

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

__init__(self, VN, VT, P): This is the constructor method for the Grammar class. It initializes the object with three attributes:

- ***VN***: A set of non-terminal symbols.
- ***VT***: A set of terminal symbols.
- ***P***: A dictionary representing production rules.

generate_string(self, start_symbol): This method generates a string from the given grammar starting from the specified ***start_symbol***. It iterates through the production rules, randomly choosing replacements for non-terminal symbols until it reaches a string consisting only of terminal symbols. The generated string is returned

```python
class Grammar:
    def __init__(self, VN, VT, P):
        self.VN = VN
        self.VT = VT
        self.P = P
    def generate_string(self, start_symbol):
        gen_string = ""
        temp = random.choice(self.P[start_symbol])
        while temp != "":
            for i in temp:
                if i in self.VT:
                    gen_string += i
                    temp = temp.replace(i, "")
                else:
                    temp = temp.replace(i, random.choice(self.P[i]))
        return gen_string

    def generate_valid_strings(self, start_symbol, num_strings):
        valid_strings = []
        for _ in range(num_strings):
            valid_strings.append(self.generate_string(start_symbol))
        return valid_strings
```

__init__(self, VN, VT, P, nameFinalState): This is the constructor method for the DFA class. It initializes the object with the following attributes:

- ***startState***: The start state of the DFA, which is set to "S" by default.
- ***nonTerm***: Set of non-terminal symbols.
- ***transitions***: A dictionary representing the transition function of the DFA
- It iterates over the productions **P** to construct the transition function based on the productions. For each production, it identifies the source state (***i***), input symbol (***perehod***), and target state (***term***). If the input symbol is empty, it considers it as a transition to the final state (***nameFinalState***).

```
class DFA():
    def __init__(self, VN, VT, P, nameFinalState):
        self.startState = "S"
        self.nonTerm = VN

        self.transictions = {}
        for i in P:
            for j in P[i]:
                term = ""
                perehod = ""
                for letter in j:
                    if(letter in VN):
                        term = letter
                    else:
                        perehod = letter
                #check if State is not in dict
                if i not in self.transictions:
                    self.transictions[i] = {}

                if term == "":
                    term = nameFinalState
                self.transictions[i][perehod] = term

        print(self.transictions)
        self.P = P
        self.finalState = nameFinalState
```

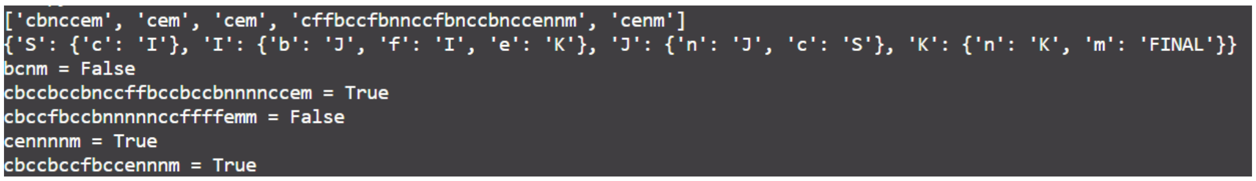
checkString – method which check whether a given input string is valid.

- It initializes ***currentTransitions*** with the start state of the DFA.
- It iterates through each symbol in the input string.
- For each symbol, it checks if there exists a transition from the current state (stored in ***currentTransitions***) with that symbol. If not, it returns False, indicating that the string is not accepted by the DFA.
- If there exists a transition, it updates ***currentTransitions*** to the next state based on the transition.
- If the final state is reached before consuming all symbols in the input string, it returns False. This part ensures that the DFA doesn't accept strings that are a prefix of an accepted string.
- If the loop completes successfully and the final state is reached, it returns True, indicating that the string is accepted by the DFA.

```
def checkString(self, string):
        currentTransitions = self.startState
        for i in range(0, len(string)):
            if string[i] not in self.transictions[currentTransitions]:
                return False
            currentTransitions = self.transictions[currentTransitions][string[i]]
            if currentTransitions == self.finalState and i!=len(string)-1:
                return False

        if currentTransitions == self.finalState:
            return True
        else:
            return False
```

Results



```
['cbnccem', 'cem', 'cem', 'cffbccfbnnccfbnccbnccennm', 'cenm']
{'S': {'c': 'I'}, 'I': {'b': 'J', 'f': 'I', 'e': 'K'}, 'J': {'n': 'J', 'c': 'S'}, 'K': {'n': 'K', 'm': 'FINAL'}}
bcnm = False
cbccbccbnccffbccbccbnnnnccem = True
cbccfbccbnnnnnccffffemm = False
cennnnm = True
cbccbccfbccennnm = True
```

Conclusions

In conclusion, this laboratory work aimed to introduce students to the concepts of formal languages, regular grammars, and finite automata. The objectives included understanding the fundamentals of formal languages. Specifically, the implementation focused on two main components, Grammar class representing a formal grammar and a DFA class representing a deterministic finite automaton. This laboratory work provided valuable hands-on experience in implementing formal language concepts in code, understanding the relationships between grammars and automata.