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

Course: Formal Languages & Finite Automata

Student: Gusev Roman

Group: FAF-222

Professor: Cojuhari Irina

University Assistant: Crețu Dumitru

Chișinău, 2024

Content

Theory

- **Definitions:**
  - **Alphabet** - is a finite, nonempty set of symbols
  - **String** - also called as "word", is a finite sequence of symbols chosen from the alphabet
  - **Length of the String** - indicates how many symbols are in mentioned string
  - **Language** - is a set of strings from an alphabet
  - **Concatenation of Strings** - is the process of putting Strings right next to each other [1]

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:
  1. Create a GitHub repository to deal with storing and updating your project;
  2. 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.);
  3. Store reports separately in a way to make verification of your work simpler.
- According to my variant number (11), get the grammar definition and do the following:
  1. Implement a type/class for your grammar;
  2. Add one function that would generate 5 valid strings from the language expressed by your given grammar;
  3. Implement some functionality that would convert an object of type Grammar to one of type Finite Automaton;
  4. 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

- For the start, I had to implement and introduce the alphabet and rules that were provided in my Variant (11). I began with defining 2 Lists for Non-Terminal Terms, Terminal Terms, 1 Dictionary for the rules/constraints, and the Start Term. After that, I instantiated a Grammar object with those Lists and Dictionary and instantiated a constant for the maximum length of words that will come in hand later.

```
    ...
if __name__ == '__main__':
    print("Laboratory Work 1 - Intro to formal languages. Regular
       grammars. Finite Automata.")
    print("Variant: 11")
    print("Student: Gusev Roman")
    print("Group: FAF-222\n")

    # Non-Terminal Terms
    V_n = ["S", "B", "D"]
    print("Non-Terminal Terms:", V_n, "\n")

    # Terminal Terms
    V_t = ["a", "b", "c"]
    print("Terminal Terms:", V_t, "\n")

    # Rules
    P = {
        "S": ["aB", "bB"],
        "B": ["bD", "cB", "aS"],
        "D": ["b", "aD"]
    }
    print("Rules:")
    for curr_term in P:
        print(curr_term + " -> " + str(P[curr_term]))

    # Start Term
```

```
    S = "S"
    print("\nStart Term:", S)

    # Maximum Length for generated Words
    max_length = 10
...
```

- After that, I designed a for loop that will iterate exactly 5 times and will generate valid words that will be stored in a List and will be used to verify if a word was already generated or not, and if the word is of the proper length to avoid recursion and duplicate. This loop will call a method from Grammar object and will generate words until all 5 are unique and have a valid length.

```
if __name__ == '__main__':
    ...
    # Instance of Grammar Class
    grammar = Grammar.Grammar(V_n, V_t, P, S)

    # List that will store all unique Words
    generated_words = []

    # 5 iterations = 5 words
    for i in range(1, 6):
        # Generate a word
        list_of_chars = grammar.generate_string(max_length)
        if list_of_chars is None:
            exit()
        generated_word = "".join(list_of_chars)
        # Verify if the word is duplicate (is already in the list) or if word length increases maxLength
        while generated_word in generated_words or len(generated_word) > max_length or (generated_word[-1] not in P and generated_word[-1].isupper()):
            if generated_word in generated_words:
                print("\nDuplicate: " + "".join(generated_word) +
```

```
                          " (Same as Word:", str(generated_words.index(generated_word) + 1) + ")")
            elif generated_word[-1] not in P and generated_word[-1].isupper():
                print("\nNo available further derivation for: " + "".join(generated_word))
            else:
                print("\nWord is too long: " + "".join(generated_word) + " | Length: ", len(generated_word))
            print("Generating new Word...")
            generated_word = "".join(grammar.generate_string(max_length))
        # Add the generated word to the list
        generated_words.append(generated_word)
        # Output the Word
        print("\nWord:", i, ": " + "".join(generated_word) + " : Length:", len(generated_word), "\n")

    print("Generated words are: ")
    for word in generated_words:
        print("Word", generated_words.index(word) + 1, ":", word)
```

- In order to define a Grammar, I used a new class with the same name, that has 4 variables:
  - **V_n** - List of Non-Terminal Terms;
  - **V_t** - List of Terminal Terms;
  - **P** - Constraints or Rules;
  - **S** - Starting Term;

  and defined its own Constructor, that will assign the given as arguments Lists and Dictionary and Start term from the Main class.

```python
...
class Grammar:
    # Constructor with some state variables as needed.
    # {V_n, V_t, P, S}
    def __init__(self, V_n, V_t, P, S):
        self.V_n = V_n
        self.V_t = V_t
        self.P = P
        self.S = S
...
```

- After this, I followed the implementation tips specified in the task Markdown file [2], but with small changes in the structure of the methods. I decided to use the mentioned method in the tips in the following manner:
  - First of all, I instantiated an empty String, that I used where I will build the generated string recursively.
  - After that, in order to ensure a random choice of the specific rule from the rules Dictionary, I had to import library Random, that provides methods for random integer value.
  - Now, there is the main recursive call to another method inside this class, that I will describe later in the report I pass the previous instantiated variables - String, Start Term, maxLength of the word and Random, that will be used inside the private method.
- To ensure stability in the code, I decided to cover all the possible edge-cases: here I check if the Start Term is a valid Non-Terminal Term, if yes, then proceed with generation, otherwise - return empty string.

```python
import Random
...
class Grammar:
    ...
    def generate_string(self, max_length):
        # Edge-case: If Start term is not present in the Non-
           Terminal List, then return empty string = cannot be
        # generated string from this variables
        if self.S not in self.V_n:
```

```
            print(f"Start Symbol: {self.S} is not present in Non-Terminal Terms")
            return None

        # String used in order to append more easily and manipulate String variables
        generated_string = []
        print(f"{self.S} -> ", end="")
        # First call of the generation of the string that will go recursively
        self.__generate_next_string(generated_string, self.S, max_length)
        # Return the generated String that is formed in the end of the recursion call.
        return generated_string
    ...
```

- In the end, I will describe the main method that is responsible for the generation of the next Strings/-words. It is a method that is called recursively, taking as input values:
  - `String List currentWord`, in which the word is built and concatenated.
  - `String term`, that is the next term that we get randomly from the Production of Rules that we have - the Map object P.
  - `int maxLength`, that will ensure that the word that is being generated will have a maximum length and will not exceed it, thereby ensuring that there will not be infinite recursion.
- First, I have to check if the current word, that is being generated, is not exceeding the maximum length that was passed to the method.
- If the word is longer, then I add, for the sake of a pretty output, the last Non-Terminal terms and exit recursion. Otherwise, I check if the term I got is a Non-Terminal Term, exactly if the term is contained in the Production of Rules that I got when Grammar object was created.
- If the Dictionary contains such a key that is equal to the term that is being analyzed, therefore the term is Non-Terminal and may be derived further. I get the possible derivation List for this Non-Terminal Term, then I choose one random derivation from this list, and for each term of the derivation (every char of the derivation string), I make a recursive call to the same method I am in currently and go again.

- Otherwise, if the Dictionary does not contain such a Term, then this Term is a Terminal Term and is being just added to the String List currentWord, and the recursion is stopped. Also, here I check if the Term is Non-Terminal and has no derivation, then the grammar is not suitable for our purposes and may lead to a word that has a Non-Terminal Term in it and can’t be derived furthermore.

```python
class Grammar:
...
def __generate_next_string(self, current_word, term, max_length):
    # Edge-case: word is too long (avoid infinite recursion)
    if len(current_word) >= max_length:
        current_word.append(term[-1])  # add last non-terminal term in order to display correctly the generated word
        return  # exit recursion

    # Case: if there is a rule/produce for this specific Non-Terminal Term -> goes into recursion for the next term
    if term in self.P:
        # Get all possible derivations for current non-terminal term
        curr_derivation_list = self.P[term]

        # Get one random derivation from the Produce List
        curr_derivation = random.choice(curr_derivation_list)

        # Check if last term is Non-Terminal, then add -> at the end, else - do not add
        if curr_derivation[-1].isupper():
            print(f'{"".join(current_word)}{curr_derivation} -> ', end="")
        else:
            print(f'{"".join(current_word)}{curr_derivation}', end="")

        # For every term, iterate again recursively: ensures adding
```

```
            the terminal term and going for
        # the non-terminal one in another recursion
        for separate_term in curr_derivation:
            self.__generate_next_string(current_word, separate_term, max_length)
    # Case: if there is no rule/produce for this specific Non-Terminal Term -> ends recursion
    else:
        # Edge-case: if the Term is Non-Terminal and has no further derivation
        if term not in self.P and term.isupper():
            current_word.append(term)
            print(f"\nNon-Terminal Term: {term} is not present in Rules Dictionary!", end="")
        # Case: if the Term is a Terminal Term
        else:
            current_word.append(term)  # add the terminal term and exits recursion
```

- In such a manner, I designed the code for the methods responsible for the generation of the words based on a set of Terminal and Non-Terminal Terms, a Start Term, and a Dictionary that maps the Terminal Term with their possible derivation into different expressions.
- For the second part of this Laboratory Work - Finite Automaton, I designed a new class with the same name and first thing developed was the constructor, that will hold the parameters for the Finite Automaton:
  - **Q** - List of Terminal States;
  - **Sigma** - Alphabet;
  - **Delta** - Transitions Set;
  - **q0** - Start State;
  - **F** - Final States;

```
class FiniteAutomaton:
    # Some state variables as needed.
    #    {Q, Sigma, delta, q0, F}
    def __init__(self, Q=None, delta=None, sigma=None, q0=None, F=
        None):
      if Q is None or delta is None or sigma is None or q0 is None
          or F is None:
          self.create_finite_automaton()
      else:
          self.Q = Q
          self.sigma = sigma
          self.delta = delta
          self.q0 = q0
          self.F = F
    ...
```

- In order to print all the variables in the console in a pretty format, I designed a method that will print them line by line:

```
    class FiniteAutomaton:
        ...
        # Print function to easy print the variables in the console
           .
        def print_variables(self):
            print("\nQ:", self.Q)
            print("Delta:", self.delta)
            print("Sigma:")
            for (k, v) in self.sigma.items():
                print("\u03C3" + str(k), "-", v)
            print("q0:", self.q0)
            print("F:", self.F)
```

- At this moment, I had to develop a method in the Grammar Class that will convert Grammar object into Finite Automaton. Very helpful for this step were: Chapter 2 of the Book "Formal Languages and Finite Automata" [3], Presentation during the course at TUM [4], and some Internet Resources:

JFLAP Application [5] and JFLAP textbook about "Converting Regular Grammar to DFA" [6].

- I followed the algorithm mentioned in those resources:
  - Assigned Non-Terminal Terms to the set of States.
  - Added a new state, that is Final State.
  - Assigned the Terminal Terms to the Alphabet of the FA.
  - Assigned the Start Term from the Grammar to the Start State of the FA.
  - Assigned the Final State to a new List that holds the new final state that I created earlier.
  - Declared the Transition Set as a Dictionary:
    * Keys - Tuple of form "(State, Term)",
    * Values - List of possible Next States based on the Current State and Terminal Term that are being analyzed

```python
...
import FiniteAutomaton
...
class Grammar:
    ...
    def to_finite_automaton(self):
        # Terminal States - will hold the possible states = Non-
           Terminal Terms + another terminal state
        Q = self.V_n
        new_element_terminal_state = "q_f"
        Q.append(new_element_terminal_state)
        # Alphabet = Terminal Terms
        delta = self.V_t
        # Start state = Start term
        q0 = self.S
        # Final states
        F = [new_element_terminal_state]
        # Transitions Set
        sigma = {}
    ...
```

- For transferring the derivations to transitions, I developed the next algorithm:
  - First of all, we have to iterate over the items in the Derivation Map.

– Then, for each pair of Non-Terminal Term and possible Derivations, which are represented as Lists, I iterate over every derivation that is in the list.
– I create a list of all the terms that are in the derivation, an empty string that will hold the input string with terminal terms, and an empty string for the next State.
– For each term in the string of the derivation, I check what type it is - Non-Terminal or Terminal:
  * If it is a valid Non-Terminal => append the string of the input string.
  * If it is a valid Terminal => assign Next State to this term.
– Create a Tuple of the form (Current State, Current input term), that is equivalent to the Left Hand Side of a Transition.
– After that, check if Transition Map contains already this specific tuple:
  * If Yes => append the list of the possible States in which the analyzed word may go.
  * If No => add to the Map this specific tuple and the next State.
– After that, return an object of type FiniteAutomaton constructed using these parameters.

```python
...
import FiniteAutomaton
...
class Grammar:
    ...
    def to_finite_automaton(self):
        ...
        # Iterate over all the Rules in the Product Dictionary (current state = non-terminal term from the dictionary)
        for current_state, derivations_list in self.P.items():
            # Iterate over all the derivations for a Non-Terminal Term
            for derivation in derivations_list:
                # Get the list of characters/terms in the string/derivation
                terms = list(derivation)
                # Current input term is the part of the derivation that is of Terminal terms
                current_input_term = ""
                # Next State is the Non-Terminal term in the
```

```
                  derivation string
                next_state = ""
                for term in terms:
                    if term.islower() and term in delta:
                        current_input_term += term
                    if term.isupper() and term in Q:
                        next_state = term
                # Initialize a list for the Left Hand side of the
                  transition function (current state, current
                  input
                # term)
                LHS = tuple([current_state, current_input_term])
                # Place it in the dictionary as a tuple as key and
                  its value is assigned to the next state.
                if LHS in sigma.keys():
                    sigma[LHS].append(next_state)
                else:
                    sigma[LHS] = [next_state]

        # Return object of type FiniteAutomaton, with the
          parameters that I found above
        return FiniteAutomaton.FiniteAutomaton(Q, delta, sigma, q0,
          F)
    ...
```

Next, I followed the same Markdown file I mentioned above and the implementation tip for the Finite Automaton, and all the logic that decides the belonging of the string to the Language generated by the Grammar.

First of all, I check if all the terms in the Input String are valid Terminal Terms, i.e. are contained in the Alphabet.

```
class FiniteAutomaton:
    ...
    def string_belong_to_language(self, input_string):
```

```
        # Edge-case: if Input String contains Terms that are not accepted by the Finite Automaton.
        for term in input_string:
            if term not in self.delta:
                return False
    ...
```

- If all the terms in the input string are valid, then I initialize a variable that will hold the State variable, with the value of the Start State. After that, I iterate over all the characters in the Input String. First, I check if the current state is NULL, which will ensure a correct input in case of no available transition and will return False which is equivalent to the rejection of the word.

```
class FiniteAutomaton:
    ...
    def string_belong_to_language(self, input_string):
        ...
        # Current state is q0 - Start State
        current_state = [self.q0]
        # Iterate over the Input String taking char by char
        for char in input_string:
            # Check if current state is Null, which became during the process next state. If yes, return false -
            # no possible next state for a specific terminal term and current state
            if current_state is None:
                return False
            ...
```

- After that, I sort the current state list, so that if it has "" which is a final state in the code, it will be placed at the index 0, and then I iterate over all the possible states the word may go at the current Term and try to get the next possible next States, based on the current Terminal Term and Current State:
  - First, I check if the state that is analyzed is not a final state, if yes, then I try to get the value of the key with the current State and current Terminal Term, that may lead to success, and will get the next possible states, or KeyError, that is caught, that means that there are no possible transitions, and then return False or, in other words, reject the word.

- Otherwise, I check if the current state list contains only one element and its element is "" final state, which again means that no possible transition and reject the word.
- In case that the iterations are finalized, it means that during the process of validation of the string / word, program did not encounter any of the edge-cases I found during the design of the algorithm, it means that program got to the last character and found all the possible next states that the word may go from the current state and terminal term. If it contains final state, then word is accepted, otherwise - rejected.

```python
class FiniteAutomaton:
    ...
    def string_belong_to_language(self, input_string):
        ...
        # Current state is q0 - Start State
        current_state = [self.q0]
        # Print Start transition for input string
        print(f"-> {current_state[0] if len(current_state) == 1 else current_state}", end="")
        # Iterate over Input String term by term
        for term in input_string:
            # Print current term
            print(" --" + term, end="--> ")
            # Initialize a set that will contain next possible States to translate into
            next_state = set()
            # Iterate over the current states and try to find next possible State to translate into
            for state in current_state:
                try:
                    # Retrieve all next possible States to translate from current state with current term and iterate
                    # over that list of possible next States and add them to the set that will replace the current state
                    # list
                    for next_state_single in self.sigma[(state, term)]:
                        next_state.add(next_state_single)

                    # Print next states
                    if list(current_state)[-1] == state:
```

```
                    print(next_state, end="")

                except KeyError:
                    # KeyError means that no possible transition
                       from current state with terminal term
                    # Check if there are no more possible next
                       states so that don't lose another possible
                       branch
                    if len(current_state) == 1:
                        # Goes into a dead State => Rejected Word
                        print("{q_d}", end="")
                        return False
                    # Else, go to the next possible State and check
                        that one.
                    if list(current_state)[-1] == state:
                        print(next_state, end="")
                    continue
            current_state = next_state
        # Transform list to set so that apply method intersection
        current_state = set(current_state)

        # Check if last possible state list contains final state
        return current_state.intersection(self.F)

        ...
```

- This is the whole logic for the method to validate the String by rejection or acceptance based on the Grammar we have.
- The main block for the second part of the laboratory work, I decided to go through different methods of the input.
- But first, I had to transform the Grammar from the first part of the laboratory work into a Finite Automaton, which I did. Also, I decided to print all the parameters it got after the conversion.

```
if __name__ == '__main__':
```

```
...
finite_automaton = grammar.to_finite_automaton()
finite_automaton.print_variables()
...
```

- Here are the following methods of output for the FA Part of the Lab. Work:
  - Checking the words generated by the Grammar, in a for loop, just to be sure that algorithm is working correctly:

```
if __name__ == '__main__':
...
# Check of method: should be ACCEPTED for all words,
   because they were generated using this grammar
print("\nCHECKING GENERATED WORDS FOR ACCEPTANCE:")
for word in generated_words:
    print(
        f"Word {generated_words.index(word) + 1} {word}:
            {"Accepted" if finite_automaton.
           string_belong_to_language(word)
        else "Rejected"}"
    )
...
```

  - In the following snippet is described the method of Manual checking of an input String. You will have to write in console the word you want to check and then get the response. Adjust the number in the range section in order to check more strings.

```
if __name__ == '__main__':
...
# FOR MANUAL INPUT, UNCOMMENT FOLLOWING LINES OF CODE:
iterations = 5
for i in range(iterations):
    input_word = input("\nEnter word: ")
    result = finite_automaton.string_belong_to_language(
       input_word)
```

```
        print(f"Word {input_word} is {"Accepted" if result
            else "Rejected"}")
    ...
```

- In the snippet below is described the method of checking randomly created combinations of words. In order to adjust the length of the words that are generated here, adjust the length_random number and to adjust the total number of combination - change number_words variable.

```
if __name__ == '__main__':
...
# FOR RANDOM WORD COMBINATION, UNCOMMENT FOLLOWING LINES
    OF CODE:
length_random = 3
number_words = 15
random_words = [
    # Randomly choose characters from letters for the
        given length of the string
    ''.join(random.choice(V_t) for _ in range(3)) for _
        in range(15)
]
for word in random_words:
    result = finite_automaton.string_belong_to_language(
        word)
    print(f"Word {word} is {"Accepted" if result else "
        Rejected"}")
...
```

- In the snippet below is described the main method of checking all the possible combination of Terminal Terms from length 0 (empty string) till a certain length nr_length that can adjusted.

```
if __name__ == '__main__':
...
# ALL POSSIBLE COMBINATIONS OF WORDS MADE OUT OF
   TERMINAL TERMS:
print("\nCHECKING ALL POSSIBLE COMBINATIONS OF TERMINAL
```

```
   TERMS:")
possible_words = []
nr_length = 5
for i in range(nr_length + 1):
    lst = [''.join(comb) for comb in itertools.product(
       V_t, repeat=i)]
    for word in lst:
        possible_words.append(word)

for word in possible_words:
    result = finite_automaton.string_belong_to_language(
       word)
    print(f"Word {word} is {"Accepted" if result else "
       Rejected"}")
```

– Additionally, I added a function to FiniteAutomaton class that, in case that for the creation of FA are not provided some parameters, they might be taken as input from the User:

```
class FiniteAutomaton:
...
def create_finite_automaton(self):
    print("CREATE YOUR OWN FINITE AUTOMATON:")

    Q = input("INPUT STATES SEPARATED BY COMMA: ")
    Q = Q.split(",")
    Q.append("q_f")
    print(Q)
    self.Q = Q

    delta = input("INPUT TERMINAL TERMS SEPARATED BY
       COMMA: ")
    delta = delta.split(",")
    print(delta)
    self.delta = delta
```

```
        q0 = input("INPUT START STATE: ")
        print(q0)
        self.q0 = q0

        self.F = ["q_f"]

        print(
            "INPUT TRANSITIONS (SEPARATED BY COMMA \"{STATE},{TERMINAL_TERM},{NEXT_STATE}\") AND USE FOR FINAL STATE \"q_f\": ")
        sigma = {}
        while True:
            transition_string = input("")
            transition = transition_string.split(",")
            print(transition)
            if tuple(transition[0] + transition[1]) in sigma:
                sigma[tuple(transition[0] + transition[1])].append(transition[2])
            else:
                sigma[tuple(transition[0] + transition[1])] = [transition[2]]
            print(f"\u03C3({transition[0]}, {transition[1]}) -> {[transition[2]]}")
            if input("CONTINUE? (Y/N) ").lower() == "n":
                break
        for (k, v) in sigma.items():
            print("\u03C3" + str(k), "-", v)
        self.sigma = sigma
    ...
```

Conclusions / Results

First part of the console output is the general information about the laboratory work, variant, student and group:

```
Laboratory Work 1 - Intro to formal languages. Regular grammars. Finite
    Automata.
Variant: 11
Student: Gusev Roman
Group: FAF-222
```

After that goes the condition I got in my variant:

```
Non-Terminal Terms: ['S', 'B', 'D']

Terminal Terms: ['a', 'b', 'c']

Rules:
S -> ['aB', 'bB']
B -> ['bD', 'cB', 'aS']
D -> ['b', 'aD']
```

After that goes the generation of the Words, that is described in the form we studied at the course lessons:

```
Start Term: S
S -> aB -> acB -> accB -> acccB -> acccaS -> acccaaB -> acccaacB ->
   acccaaccB -> acccaacccB -> acccaaccccB ->
Word is too long: acccaaccccB | Length:  11
Generating new Word...
S -> aB -> acB -> acaS -> acaaB -> acaabD -> acaabaD -> acaabaaD ->
   acaabaab
Word: 1 : acaabaab : Length: 8

S -> bB -> bcB -> bcaS -> bcaaB -> bcaabD -> bcaabaD -> bcaabaaD ->
   bcaabaaaD -> bcaabaaab
```

```
Word: 2 : bcaabaaab : Length: 9

S -> aB -> abD -> abaD -> abaaD -> abaab
Word: 3 : abaab : Length: 5

S -> bB -> baS -> baaB -> baabD -> baabaD -> baabab
Word: 4 : baabab : Length: 6

S -> bB -> bcB -> bcbD -> bcbb
Word: 5 : bcbb : Length: 4

Generated words are:
Word 1 : acaabaab
Word 2 : bcaabaaab
Word 3 : abaab
Word 4 : baabab
Word 5 : bcbb
```

After that goes second part of the laboratory work, with Finite Automaton, and again - the parameters I got after the conversion:

```
Q: ['S', 'B', 'D', 'q_f']
Delta: ['a', 'b', 'c']
Sigma:
σ('S', 'a') - ['B']
σ('S', 'b') - ['B']
σ('B', 'b') - ['D']
σ('B', 'c') - ['B']
σ('B', 'a') - ['S']
σ('D', 'b') - ['']
σ('D', 'a') - ['D']
q0: S
F: ['q_f']
```

After that, I check the previously generated words (they should be always accepted, because were generated by the Grammar):

```
CHECKING GENERATED WORDS FOR ACCEPTANCE:
Input String: abb
-> S --a--> {'B'} --b--> {'D'} --b--> {'q_f'}
Word 1 abb: Accepted

Input String: bcbb
-> S --b--> {'B'} --c--> {'B'} --b--> {'D'} --b--> {'q_f'}
Word 2 bcbb: Accepted

Input String: aabcaacbab
-> S --a--> {'B'} --a--> {'S'} --b--> {'B'} --c--> {'B'} --a--> {'S'}
   --a--> {'B'} --c--> {'B'} --b--> {'D'} --a--> {'D'} --b--> {'q_f'}
Word 3 aabcaacbab: Accepted

Input String: baabab
-> S --b--> {'B'} --a--> {'S'} --a--> {'B'} --b--> {'D'} --a--> {'D'}
   --b--> {'q_f'}
Word 4 baabab: Accepted

Input String: acbab
-> S --a--> {'B'} --c--> {'B'} --b--> {'D'} --a--> {'D'} --b--> {'q_f'}
Word 5 acbab: Accepted
```

As you may see, all the words were accepted.

After that, by the choice you have done (if you uncommented the code with manual input or with the random combinations), you may get the following output: * Manual Input:

```
Enter word:
```

Then input the word you want:

```
Enter word: aaacabbab

Input String: aaacabbab
```

```
-> S --a--> {'B'} --a--> {'S'} --a--> {'B'} --c--> {'B'} --a--> {'S'}
   --b--> {'B'} --b--> {'D'} --a--> {'D'} --b--> {'q_f'}
Word aaacabbab is Accepted
```

* Random combinations of Terminal Terms:

```
Input String: bbb
-> S --b--> {'B'} --b--> {'D'} --b--> {'q_f'}
Word bbb is Accepted

Input String: baa
-> S --b--> {'B'} --a--> {'S'} --a--> {'B'}
Word baa is Rejected

Input String: cab
-> S --c--> {q_d}
Word cab is Rejected

Input String: cac
-> S --c--> {q_d}
Word cac is Rejected

Input String: aba
-> S --a--> {'B'} --b--> {'D'} --a--> {'D'}
Word aba is Rejected

Input String: aab
-> S --a--> {'B'} --a--> {'S'} --b--> {'B'}
Word aab is Rejected

Input String: acc
-> S --a--> {'B'} --c--> {'B'} --c--> {'B'}
Word acc is Rejected
```

```
Input String: acc
-> S --a--> {'B'} --c--> {'B'} --c--> {'B'}
Word acc is Rejected

Input String: aca
-> S --a--> {'B'} --c--> {'B'} --a--> {'S'}
Word aca is Rejected

Input String: ccc
-> S --c--> {q_d}
Word ccc is Rejected

Input String: aac
-> S --a--> {'B'} --a--> {'S'} --c--> {q_d}
Word aac is Rejected
```

* All Possible Combinations of Terminal Terms:

```
CHECKING ALL POSSIBLE COMBINATIONS OF TERMINAL TERMS:
Input String: abb
-> S --a--> {'B'} --b--> {'D'} --b--> {'q_f'}
Word abb is Accepted

Input String: abc
-> S --a--> {'B'} --b--> {'D'} --c--> {q_d}
Word abc is Rejected

Input String: aca
-> S --a--> {'B'} --c--> {'B'} --a--> {'S'}
Word aca is Rejected

Input String: acb
-> S --a--> {'B'} --c--> {'B'} --b--> {'D'}
Word acb is Rejected
```

```

Input String: acc
-> S --a--> {'B'} --c--> {'B'} --c--> {'B'}
Word acc is Rejected

Input String: baa
-> S --b--> {'B'} --a--> {'S'} --a--> {'B'}
Word baa is Rejected

Input String: bab
-> S --b--> {'B'} --a--> {'S'} --b--> {'B'}
Word bab is Rejected

Input String: bac
-> S --b--> {'B'} --a--> {'S'} --c--> {q_d}
Word bac is Rejected

Input String: bba
-> S --b--> {'B'} --b--> {'D'} --a--> {'D'}
Word bba is Rejected

Input String: bbb
-> S --b--> {'B'} --b--> {'D'} --b--> {'q_f'}
Word bbb is Accepted
...
```

* Also, here is how creating a new FiniteAutomaton without passing some parameters looks like:

```
CREATE YOUR OWN FINITE AUTOMATON:
INPUT STATES SEPARATED BY COMMA: S,B,D
['S', 'B', 'D', 'q_f']
INPUT TERMINAL TERMS SEPARATED BY COMMA: a,b,c
['a', 'b', 'c']
INPUT START STATE: S
```

```
S
INPUT TRANSITIONS (SEPARATED BY COMMA "{STATE},{TERMINAL_TERM},{NEXT_STATE}") AND USE FOR FINAL STATE "q_f":
S,a,B
[’S’, ’a’, ’B’]
  (S, a) -> [’B’]
CONTINUE? (Y/N) y
S,b,B
[’S’, ’b’, ’B’]
  (S, b) -> [’B’]
CONTINUE? (Y/N) y
B,b,D
[’B’, ’b’, ’D’]
  (B, b) -> [’D’]
CONTINUE? (Y/N) y
D,b,q_f
[’D’, ’b’, ’q_f’]
  (D, b) -> [’q_f’]
CONTINUE? (Y/N) y
D,a,D
[’D’, ’a’, ’D’]
  (D, a) -> [’D’]
CONTINUE? (Y/N) y
B,c,B
[’B’, ’c’, ’B’]
  (B, c) -> [’B’]
CONTINUE? (Y/N) y
B,a,S
[’B’, ’a’, ’S’]
  (B, a) -> [’S’]
CONTINUE? (Y/N) n
```

* I input the grammar I got in my Variant and the output for the variables is the same as for the original FiniteAutomaton generated from the given Grammar.

```
Q: ['S', 'B', 'D', 'q_f']
Delta: ['a', 'b', 'c']
Sigma:
  ('S', 'a') - ['B']
  ('S', 'b') - ['B']
  ('B', 'b') - ['D']
  ('D', 'b') - ['q_f']
  ('D', 'a') - ['D']
  ('B', 'c') - ['B']
  ('B', 'a') - ['S']
q0: S
F: ['q_f']
```

* And to check if they are correct, I used the same function with all the possible combinations and got the same Output as for the original generated Finite Automaton from the Grammar directly through a method.

```
CHECKING ALL POSSIBLE COMBINATIONS OF TERMINAL TERMS:
Input String: abb
-> S --a--> {'B'} --b--> {'D'} --b--> {'q_f'}
Word abb is Accepted

Input String: abc
-> S --a--> {'B'} --b--> {'D'} --c--> {q_d}
Word abc is Rejected

Input String: aca
-> S --a--> {'B'} --c--> {'B'} --a--> {'S'}
Word aca is Rejected

Input String: acb
-> S --a--> {'B'} --c--> {'B'} --b--> {'D'}
Word acb is Rejected

```

```
Input String: acc
-> S --a--> {'B'} --c--> {'B'} --c--> {'B'}
Word acc is Rejected

Input String: baa
-> S --b--> {'B'} --a--> {'S'} --a--> {'B'}
Word baa is Rejected

Input String: bab
-> S --b--> {'B'} --a--> {'S'} --b--> {'B'}
Word bab is Rejected

Input String: bac
-> S --b--> {'B'} --a--> {'S'} --c--> {q_d}
Word bac is Rejected

Input String: bba
-> S --b--> {'B'} --b--> {'D'} --a--> {'D'}
Word bba is Rejected

Input String: bbb
-> S --b--> {'B'} --b--> {'D'} --b--> {'q_f'}
Word bbb is Accepted
...
```

As a conclusion to this Laboratory Work, I can say that I accomplished all the given tasks, specifically creation of 2 classes:

- Grammar - used to hold the parameters of a Grammar and methods to generate different random words and a method to convert an instance of this class into a Finite Automaton object.
- Finite Automaton - used to hold the parameters transformed from the Grammar type to a Finite Automaton ones and method that checks the validation status of the input string - if it is accepted or rejected by the FA.

Also, I managed to understand better the concept of Regular Grammars, how are words formed and generated by this specific type of Grammar. Besides that, I understood how to convert from Regular

Grammar to Finite Automaton by the use of a not very complex Algorithm and managed to make my own implementation of it. Very useful in checking the correctness of the responses I got was one website [7], that takes the Grammar and have a text field where I input the words generated by my algorithm and got the same response as on the Website, therefore I am more than sure that on some not very complex examples of Grammars, my algorithm is working fine. Although, on some inputs of the Grammar, that have some uncertainty in it, the algorithm is failing.

Bibliography

[1] Formal Languages and Finite Automata Guide for practical lessons - TUM
https://else.fcim.utm.md/pluginfile.php/110458/mod_resource/content/0/LFPC_Guide.pdf

[2] Laboratory Work 1: Intro to formal languages. Regular grammars. Finite Automata. Task - Dumitru Crudu, Vasile Drumea, Irina Cojuhari
https://github.com/filpatterson/DSL_laboratory_works/blob/master/1_RegularGrammars/task.md

[3] Formal Languages and Finite Automata Guide for practical lessons Chapter 2 - TUM
https://else.fcim.utm.md/pluginfile.php/64791/mod_resource/content/0/Chapter_2.pdf

[4] Presentation "Regular Language. Finite Automata" - TUM
https://drive.google.com/file/d/1rBGyzDN5eWMXTNeUxLxmKsf7tyhHt9Jk/view

[5] JFLAP Application Web Site - Susan H. Rodger
https://www.jflap.org/

[6] Converting Regular Grammar to DFA - JFLAP
https://www.jflap.org/modules/ConvertedFiles/Regular%20Grammar%20to%20DFA%20Conversion%20Module.pdf

[7] CFG Developer - Christopher Wong, Kevin Gibbons - https://web.stanford.edu/class/archive/cs/cs103/cs103.1156/tools/cfg/