

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	3
Objectives	3
Implementation	4
Conclusions	21
Bibliography	26

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.

```
1     ...
2 if __name__ == '__main__':
3     print("Laboratory Work 1 - Intro to formal languages. Regular
4         grammars. Finite Automata.")
5     print("Variant: 11")
6     print("Student: Gusev Roman")
7     print("Group: FAF-222\n")
8
9     # Non-Terminal Terms
10    V_n = ["S", "B", "D"]
11    print("Non-Terminal Terms:", V_n, "\n")
12
13    # Terminal Terms
14    V_t = ["a", "b", "c"]
15    print("Terminal Terms:", V_t, "\n")
16
17    # Rules
18    P = {
19        "S": ["aB", "bB"],
20        "B": ["bD", "cB", "aS"],
21        "D": ["b", "aD"]
22    }
23    print("Rules:")
24    for curr_term in P:
25        print(curr_term + " -> " + str(P[curr_term]))
26
27    # Start Term
```

```

27     S = "S"
28     print("\nStart Term:", S)
29
30     # Maximum Length for generated Words
31     max_length = 10
32     ...

```

- 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.

```

1  if __name__ == '__main__':
2      ...
3      # Instance of Grammar Class
4      grammar = Grammar.Grammar(V_n, V_t, P, S)
5
6      # List that will store all unique Words
7      generated_words = []
8
9      # 5 iterations = 5 words
10     for i in range(1, 6):
11         # Generate a word
12         list_of_chars = grammar.generate_string(max_length)
13         if list_of_chars is None:
14             exit()
15         generated_word = "".join(list_of_chars)
16         # Verify if the word is duplicate (is already in the list)
17         # or if word length increases maxLength
18         while generated_word in generated_words or len(
19             generated_word) > max_length or (generated_word[-1] not
20             in P and generated_word[-1].isupper()):
21             if generated_word in generated_words:
22                 print("\nDuplicate: " + "".join(generated_word) +

```

```

20         " (Same as Word:", str(generated_words.index(
        generated_word) + 1) + ")")
21     elif generated_word[-1] not in P and generated_word
        [-1].isupper():
22         print("\nNo available further derivation for: " + "
        ".join(generated_word))
23     else:
24         print("\nWord is too long: " + "".join(
        generated_word) + " | Length: ", len(
        generated_word))
25         print("Generating new Word...")
26         generated_word = "".join(grammar.generate_string(
        max_length))
27     # Add the generated word to the list
28     generated_words.append(generated_word)
29     # Output the Word
30     print("\nWord:", i, ": " + "".join(generated_word) + " :
        Length:", len(generated_word), "\n")
31
32     print("Generated words are: ")
33     for word in generated_words:
34         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.

```

1  ...
2  class Grammar:
3      # Constructor with some state variables as needed.
4      # {V_n, V_t, P, S}
5      def __init__(self, V_n, V_t, P, S):
6          self.V_n = V_n
7          self.V_t = V_t
8          self.P = P
9          self.S = S
10 ...

```

- 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.

```

1  import Random
2  ...
3  class Grammar:
4      ...
5      def generate_string(self, max_length):
6          # Edge-case: If Start term is not present in the Non-
7          # Terminal List, then return empty string = cannot be
8          # generated string from this variables
9          if self.S not in self.V_n:

```

```

9         print(f"Start Symbol: {self.S} is not present in Non-
           Terminal Terms")
10        return None
11
12        # String used in order to append more easily and manipulate
           String variables
13        generated_string = []
14        print(f"{self.S} -> ", end="")
15        # First call of the generation of the string that will go
           recursively
16        self.__generate_next_string(generated_string, self.S,
           max_length)
17        # Return the generated String that is formed in the end of
           the recursion call.
18        return generated_string
19    ...

```

- 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.

```
1 class Grammar:
2     ...
3 def __generate_next_string(self, current_word, term, max_length):
4     # Edge-case: word is too long (avoid infinite recursion)
5     if len(current_word) >= max_length:
6         current_word.append(term[-1]) # add last non-terminal term
7         # in order to display correctly the generated word
8         return # exit recursion
9
10    # Case: if there is a rule/produce for this specific Non-
11    # Terminal Term -> goes into recursion for the next term
12    if term in self.P:
13        # Get all possible derivations for current non-terminal
14        # term
15        curr_derivation_list = self.P[term]
16
17        # Get one random derivation from the Produce List
18        curr_derivation = random.choice(curr_derivation_list)
19
20        # Check if last term is Non-Terminal, then add -> at the
21        # end, else - do not add
22        if curr_derivation[-1].isupper():
23            print(f'{" ".join(current_word)}{curr_derivation} -> ',
24                  end="")
25        else:
26            print(f'{" ".join(current_word)}{curr_derivation}', end=
27                  "")
28
29        # For every term, iterate again recursively: ensures adding
```

```

        the terminal term and going for
24         # the non-terminal one in another recursion
25         for separate_term in curr_derivation:
26             self.__generate_next_string(current_word, separate_term
                , max_length)
27         # Case: if there is no rule/produce for this specific Non-
            Terminal Term -> ends recursion
28     else:
29         # Edge-case: if the Term is Non-Terminal and has no further
            derivation
30         if term not in self.P and term.isupper():
31             current_word.append(term)
32             print(f"\nNon-Terminal Term: {term} is not present in
                Rules Dictionary!", end="")
33         # Case: if the Term is a Terminal Term
34     else:
35         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;

```

1  class FiniteAutomaton:
2      # Some state variables as needed.
3      #      {Q, Sigma, delta, q0, F}
4      def __init__(self, Q, delta, sigma, q0, F):
5          self.Q = Q
6          self.sigma = sigma
7          self.delta = delta
8          self.q0 = q0
9          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:

```

1  class FiniteAutomaton:
2      ...
3      # Print function to easy print the variables in the console
4      .
5      def print_variables(self):
6          print("\nQ:", self.Q)
7          print("Delta:", self.delta)
8          print("Sigma:")
9          for (k, v) in self.sigma.items():
10             print("\u03C3" + str(k), "-", v)
11         print("q0:", self.q0)
12         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

```

1  ...
2  import FiniteAutomaton
3  ...
4  class Grammar:
5      ...
6      def to_finite_automaton(self):
7          # Terminal States - will hold the possible states = Non-
            Terminal Terms + another terminal state
8          Q = self.V_n
9          new_element_terminal_state = "q_f"
10         Q.append(new_element_terminal_state)
11         # Alphabet = Terminal Terms
12         delta = self.V_t
13         # Start state = Start term
14         q0 = self.S
15         # Final states
16         F = [new_element_terminal_state]
17         # Transitions Set
18         sigma = {}
19         ...

```

- 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.

```

1  ...
2  import FiniteAutomaton
3  ...
4  class Grammar:
5      ...
6      def to_finite_automaton(self):
7          ...
8          # Iterate over all the Rules in the Product Dictionary (
9              current state = non-terminal term from the dictionary)
10         for current_state, derivations_list in self.P.items():
11             # Iterate over all the derivations for a Non-Terminal
12                 Term
13             for derivation in derivations_list:
14                 # Get the list of characters/terms in the string/
15                     derivation
16                 terms = list(derivation)
17                 # Current input term is the part of the derivation
18                     that is of Terminal terms
19                 current_input_term = ""
20                 # Next State is the Non-Terminal term in the
21                     derivation string
22                 next_state = ""
23                 for term in terms:
24                     if term.islower() and term in delta:
25                         current_input_term += term
26                     if term.isupper() and term in Q:

```

```

22         next_state = term
23         # Initialize a list for the Left Hand side of the
           transition function (current state, current
           input
24         # term)
25         LHS = tuple([current_state, current_input_term])
26         # Place it in the dictionary as a tuple as key and
           its value is assigned to the next state.
27         if LHS in sigma.keys():
28             sigma[LHS].append(next_state)
29         else:
30             sigma[LHS] = [next_state]
31
32         # Return object of type FiniteAutomaton, with the
           parameters that I found above
33         return FiniteAutomaton.FiniteAutomaton(Q, delta, sigma, q0,
           F)
34     ...

```

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.

```

1 class FiniteAutomaton:
2     ...
3     def string_belongs_to_language(self, input_string):
4         # Edge-case: if Input String contains Terms that are not
           accepted by the Finite Automaton.
5         for term in input_string:
6             if term not in self.delta:
7                 return False
8     ...

```

- 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.

```

1 class FiniteAutomaton:
2     ...
3     def string_belong_to_language(self, input_string):
4         ...
5         # Current state is q0 - Start State
6         current_state = [self.q0]
7         # Iterate over the Input String taking char by char
8         for char in input_string:
9             # Check if current state is Null, which became during
10              the process next state. If yes, return false -
11              # no possible next state for a specific terminal term
12              and current state
13              if current_state is None:
14                  return False
15         ...

```

- 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 (this part may need improvement in the future because when there are multiple transitions for a state with the same input term, it may lead to uncertainty and give wrong output, I have tried to fix this for 3 days in a row, but no success) and reject the word.

```

1 class FiniteAutomaton:
2     ...
3     def string_belong_to_language(self, input_string):
4         ...
5         # Current state is q0 - Start State
6         current_state = [self.q0]
7         # Iterate over the Input String taking char by char
8         for char in input_string:
9             # Check if current state is Null, which became during
10                the process next state. If yes, return false -
11                # no possible next state for a specific terminal term
12                and current state
13            if current_state is None:
14                return False
15            # This method puts final state on the index 0 of the
16                list.
17            current_state.sort()
18            # There might be multiple possible next states from a
19                current state, so we iterate over them
20            for state in current_state:
21                # Try to get from the dictionary next state by the
22                    state
23                try:
24                    # Case: if state is not "", that is final state
25                        , then try to get the next state
26                    # from the transitions list, which might give
27                        Key Error (such transition does not exit in
28                        the list
29                    # therefore no possible transition for the
30                        current state and terminal term => reject
31                        the word)
32                    if state != '':
33                        current_state = self.sigma[(state, char)]

```



```

24         # Edge-case: if state is final state, and it is
           the only possible next State, then return
           false,
25         # because this term program checks is not the
           last character in the input string therefore
           no
26         # possible further transition.
27         elif len(current_state) == 1 and current_state
           [0] == "":
28             return False
29     except KeyError:
30         # As I mentioned, if the transition is not
           present in the list, it gives error when
           trying to get
31         # that specific transition, therefore return
           False aka reject the word
32         return False
33     ...

```

- 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.

```

1 class FiniteAutomaton:
2     ...
3     def string_belong_to_language(self, input_string):
4         ...
5         # Iterate over the Input String taking char by char
6         for char in input_string:
7             ...
8         else:
9             # When entire string is parsed, check whether the final

```

```

        state is an accepted state
10         # for possible_state in next_state:
11         if "" in current_state:
12             return True
13         else:
14             return False

```

- 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.

```

1 if __name__ == '__main__':
2     ...
3     finite_automaton = grammar.to_finite_automaton()
4     finite_automaton.print_variables()
5     ...

```

- 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:

```

1     if __name__ == '__main__':
2         ...
3         # Check of method: should be ACCEPTED for all words,
4         because they were generated using this grammar
5         print("\nCHECKING GENERATED WORDS FOR ACCEPTANCE:")
6         for word in generated_words:
7             print(
8                 f"Word {generated_words.index(word) + 1} {word}:
9                 {"Accepted" if finite_automaton.
10                  string_belong_to_language(word)
11                  else "Rejected"}"

```

9
10

```
)  
...  

```

- 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.

1
2
3
4
5
6
7
8
9

```
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.

1
2
3
4
5
6
7
8
9

```
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)  
    ]  

```

```

10     for word in random_words:
11         result = finite_automaton.string_belong_to_language(
            word)
12         print(f"Word {word} is {"Accepted" if result else "
            Rejected"})
13     ...

```

- 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.

```

1     if __name__ == '__main__':
2         ...
3         # ALL POSSIBLE COMBINATIONS OF WORDS MADE OUT OF
            TERMINAL TERMS:
4         print("\nCHECKING ALL POSSIBLE COMBINATIONS OF TERMINAL
            TERMS:")
5         possible_words = []
6         nr_length = 5
7         for i in range(nr_length + 1):
8             lst = [''.join(comb) for comb in itertools.product(
                V_t, repeat=i)]
9             for word in lst:
10                 possible_words.append(word)
11
12         for word in possible_words:
13             result = finite_automaton.string_belong_to_language(
                word)
14             print(f"Word {word} is {"Accepted" if result else "
                Rejected"})

```

Conclusions / Results

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

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

After that goes the condition I got in my variant:

```
1 Non-Terminal Terms: ['S', 'B', 'D']
2
3 Terminal Terms: ['a', 'b', 'c']
4
5 Rules:
6 S -> ['aB', 'bB']
7 B -> ['bD', 'cB', 'aS']
8 D -> ['b', 'aD']
```

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

```
1 Start Term: S
2 S -> aB -> acB -> accB -> acccB -> acccaS -> acccaaB -> acccaacB ->
  acccaaccB -> acccaacccB -> acccaaccccB ->
3 Word is too long: acccaaccccB | Length: 11
4 Generating new Word...
5 S -> aB -> acB -> acaS -> acaaB -> acaabD -> acaabaD -> acaabaaD ->
  acaabaab
6 Word: 1 : acaabaab : Length: 8
7
8 S -> bB -> bcB -> bcaS -> bcaaB -> bcaabD -> bcaabaD -> bcaabaaD ->
  bcaabaaaD -> bcaabaaab
```

```

9 Word: 2 : bcaabaaab : Length: 9
10
11 S -> aB -> abD -> abaD -> abaaD -> abaab
12 Word: 3 : abaab : Length: 5
13
14 S -> bB -> baS -> baaB -> baabD -> baabaD -> baabab
15 Word: 4 : baabab : Length: 6
16
17 S -> bB -> bcB -> bcbD -> bcbb
18 Word: 5 : bcbb : Length: 4
19
20 Generated words are:
21 Word 1 : acaabaab
22 Word 2 : bcaabaaab
23 Word 3 : abaab
24 Word 4 : baabab
25 Word 5 : bcbb

```

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

```

1 Q: ['S', 'B', 'D', 'q_f']
2 Delta: ['a', 'b', 'c']
3 Sigma:
4  $\sigma('S', 'a') = ['B']$ 
5  $\sigma('S', 'b') = ['B']$ 
6  $\sigma('B', 'b') = ['D']$ 
7  $\sigma('B', 'c') = ['B']$ 
8  $\sigma('B', 'a') = ['S']$ 
9  $\sigma('D', 'b') = ['']$ 
10  $\sigma('D', 'a') = ['D']$ 
11 q0: S
12 F: ['q_f']

```

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

generated by the Grammar):

```
1 CHECKING GENERATED WORDS FOR ACCEPTANCE:
2 Word 1 acaabaab: Accepted
3 Word 2 bcaabaaab: Accepted
4 Word 3 abaab: Accepted
5 Word 4 baabab: Accepted
6 Word 5 bcbb: 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:

```
1 Enter word:
```

Then input the word you want:

```
1 Enter word: aaacabbab
2 Word aaacabbab is Accepted
```

* Random combinations of Terminal Terms:

```
1 Word bba is Rejected
2 Word bca is Rejected
3 Word bba is Rejected
4 Word abc is Rejected
5 Word bcb is Rejected
6 Word bba is Rejected
7 Word bcb is Rejected
8 Word cca is Rejected
9 Word bcc is Rejected
10 Word aca is Rejected
11 Word cac is Rejected
12 Word bac is Rejected
13 Word cba is Rejected
14 Word aca is Rejected
15 Word bca is Rejected
```

* All Possible Combinations of Terminal Terms:

```
1  CHECKING ALL POSSIBLE COMBINATIONS OF TERMINAL TERMS:
2  Word  is Rejected
3  Word a is Rejected
4  Word b is Rejected
5  Word c is Rejected
6  Word aa is Rejected
7  Word ab is Rejected
8  Word ac is Rejected
9  Word ba is Rejected
10 Word bb is Rejected
11 Word bc is Rejected
12 Word ca is Rejected
13 Word cb is Rejected
14 Word cc is Rejected
15 Word aaa is Rejected
16 Word aab is Rejected
17 Word aac is Rejected
18 Word aba is Rejected
19 Word abb is Accepted
20 Word abc is Rejected
21 Word aca is Rejected
22 Word acb is Rejected
23 Word acc is Rejected
24 Word baa is Rejected
25 ...
```

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/>