MINISTERUL EDUCAŢIEI ȘI CERCETĂRII AL REPUBLICII MOLDOVA
UNIVERSITATEA TEHNICĂ A MOLDOVEI
FACULTATEA CALCULATOARE, INFORMATICĂ ȘI MICROELECTRONICĂ

Formal Languages & Finite Automata

Laboratory work 1:
Intro to Formal Languages. Regular Grammars. Finite Automata

Student: Novac Felicia, FAF-222
Verified: prof., Cojuhari Irina
univ. assist., Crețu Dumitru

Chișinău, 2024

TABLE OF CONTENTS

Theory **2**

Objective **3**

Implementation **3**

Grammar 3

Finite Automata 4

Results **6**

String Generation 6

Grammar Conversion to FA 7

String Appurtenance to Language 8

Conclusion **8**

Bibliography **9**

Theory

- *Grammar* - set of rules defining how to form a language. It is consisted of the (V_n, V_t, P, S) tuple.
  - *Non-terminals* (V_n) - uppercase letters, used to derive into other pieces of text.
  - *Terminals* (V_t) - lowercase letters which form the strings of the language.
  - *Productions* (P) - rules on how the non-terminals should be converted into terminals and other non-terminals.
  - *Starting Symbol* (S) - the name speaks for itself; it is the initial symbol from which the language is generated.
- *Finite Automata* - is like a robot that follows a path based on signals, evolving through states until it reaches an end or stops. It consists of a tuple, similar yet different from that of a grammar $(Q, \Sigma, \Delta, q_0, F)$.
  - Q - set of states, equivalent to V_n plus an additional state.
  - Σ - equivalent to V_t.
  - Δ - transition function.
  - q_0 - equivalent to S, the starting state.
  - F - set of final states, a subset of Q.

Objective

- Populate the class `Grammar` with the variant.
- Implement the `generate_string()` method in `Grammar` class.
- Construct the `blueprint` of Finite Automaton.
- Implement the `toFiniteAutomaton()` method in `Grammar` class.
- Implement the `stringBelongToLanguage(String string)` method in the `FiniteAutomata` class.
- Understand the basic concepts of Grammar & Finite Automata.

Implementation

Grammar

I have stored the V_n & V_t in `Lists` of `Characters`, and the productions in a `Map`, as I saw it as the most convenient way to link a `Character` (non-terminal) to a `List of Strings` (combo between V_n & V_t).

The next addressed task was to generate strings from the stuff that we already have, but not implementing a specific length or any additional stop conditions. So, basically, I started from the prompt S, and looked for the rules in the `Map` for corresponding non-terminal symbol (extract the corresponding `List`), and then pick one of them at `Random`. Then, via a `for loop` I go through the `List` of productions for corresponding non-terminal, and then check each element of the `String` for V_n appurtenance. If V_n does not contain the given symbol, I append it via a `StringBuilder`, else I call recursively the same function for the new non-terminal. For instance, $S \to dA \to daB \to dabC \to dabcB \to dabcd$.

```
private String generateString(Character symbol) {
  if (Vt.contains(symbol)) {
    return symbol.toString();
  }
  List<String> productions = P.get(symbol);
  String production = productions.get(random.nextInt(productions.size()));
  StringBuilder result = new StringBuilder();
  for (char c : production.toCharArray()) {
    if(Vn.contains(c)){
      result.append(generateString(c));
    } else {
      result.append(c);
    }
  }
  return result.toString();
}
```

Finite Automata

Next, I defined the `FiniteAutomaton` class, I've decided to use `Lists` for representing states, alphabet, and finalStates (as it might be more than one). The transitions are represented in a `Map` with `key Character` and `Value another Map of Characters`, for convenience, in order to treat non-terminals and terminals separately (each non-terminal is assigned to a map of a key-terminal with value non-terminal/or not), which means treating the states and transitions separately. Comparing to grammar, where the behavior of the string creation is obvious, I've seen FA as a more complex structure, and avoided using `List` in this case.

In order to convert the Grammar to FA, I simply converted the defined grammar elements into the FA elements. So, for the Sigma and q0, nothing really changes, while for the Q, we convert V_n into another `List` and additionally add F (final state). And here goes the most interesting part, in order to define delta, we can first iterate through all V_n's and put them in the `Map` as keys, with an `empty Map-value` for each. Then, in the same loop, we extract the productions (a `List`) for each V_n, and now `iterate through` `these productions` in order to split the terminal from non-terminal and put them in the `inner map`. So the inner map will contain the key-terminal symbol and value-corresponding non-terminal or F. An important moment here is to check the grammar for regularity, which is done by checking the terminals to be on the left side, as this is a `Right Linear Grammar`.

```
Map<Character, Map<Character, Character>> faTransitions = new HashMap<>();
for (Character state : Vn) {
  faTransitions.put(state, new HashMap<>());
  List<String> productions = P.get(state);

  for (String production : productions) {
    char input = production.charAt(0); // Terminal symbol
    Character nextState = production.length() > 1 ? production.charAt(1) : 'F';
    if (!Vt.contains(input)) {
      continue;
    }
    faTransitions.get(state).put(input, nextState);
  }
}
```

In order to implement the next and last task which is checking the word belongness to the language via FA, I have created the following `graph representation` to better visualize how everything should work. Basically we receive a word, and we go through each letter (symbol) and try get to the final state, without any errors. So if we receive an input as *kitty*, we analyze the word letter by letter, and try to apply each symbol as a parameter to the current state, but this example obviously does not belong to our language.

The last part of the lab requires the construction of the method for checking string appurtenance to the language, as shown in Figure 1. So, basically, I've tried to transpose the algorithm from the courses into code. First, we check whether each letter in the 'String' belongs to the alphabet via a `for-loop`, if not then there is no sense of checking further. Next, in the same loop, starting with the first state, retrieve the transition rules for the state and check if there is a transition rule for the current letter in the current state, if not - the string does not belong to language, but if yes - continue by updating the current state based on the transition rule (going into the next state). This will take place until we go through the entire length of the `String`. When we reach the end, we check whether the state that we belong to is one of the final ones, and

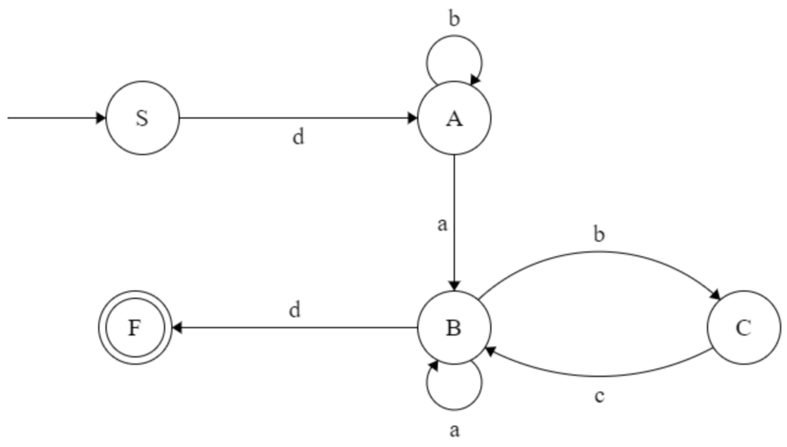

Figure 1. Finite Automaton Visualization V. 17

it is all done.

```
for (int i = 0; i < inputString.length(); i++) {
  Character currentLetter = inputString.charAt(i);

  Map<Character, Character> currentTransitions = transitions.get(currentState);
  if (currentTransitions != null && currentTransitions.containsKey(currentLetter)) {
    currentState = currentTransitions.get(currentLetter);
  } else {
     System.out.println("No transition found for state " + currentState + " with input "
    return false; // Transition not found
    }
}
```

Results

String Generation

In order to better visualize the transitions made while generating the string, I have optionally added a line of code in the `generateString()` method.

```
System.out.println(symbol + " -> " + production);
```

So, here is an example of how the program goes through the productions, and its final result:

```
S -> dA
A -> bA
A -> bA
```

```
A -> bA
A -> bA
A -> aB
B -> d
Result: dbbbbad
```

But this is not what does really matter, let's look at the generated strings instead:

```
Generated string 1 : dbabcbcd
Generated string 2 : dbad
Generated string 3 : dad
Generated string 4 : dbabcaaad
Generated string 5 : dbbabcaad
```

Grammar Conversion to FA

For checking the correctness of the conversion, I've created an additional method inside the FA class, for displaying it.

```
public void displayAutomaton() {
  System.out.println("Finite Automaton Structure:");
  System.out.println("States: " + states);
  System.out.println("Alphabet: " + alphabet);
  System.out.println("Start State: " + startState);
  System.out.println("Final States: " + finalStates);
  System.out.println("Transitions: ");
  for (Map.Entry<Character, Map<Character, Character>> entry : transitions.entrySet()) {
    Character state = entry.getKey();
    for (Map.Entry<Character, Character> trans : entry.getValue().entrySet()) {
      Character input = trans.getKey();
      Character nextState = trans.getValue();
      System.out.println("    " + state + " --(" + input + ")--> " + nextState);
    }
  }
}
```

So, here is what I've got for converting the 17th variant to FA:

```
Finite Automaton Structure:
States: [S, A, B, C, F]
Alphabet: [a, b, c, d]
Start State: S
Final States: [F]
Transitions:
    A --(a)--> B
    A --(b)--> A
    B --(a)--> B
    B --(b)--> C
    B --(d)--> F
    S --(d)--> A
    C --(c)--> B
```

String Appurtenance to Language

For checking the correctness of this part of the laboratory, I have used `Scanner` continuously, in a `while(true)` loop, to get the input from the console, and checked for the words generated by the code, and some more obviously invalid others. So here are the results from my console:

```
Enter a string to check if it belongs to the language (or type 'exit' to quit):
dbabcbcd
The string 'dbabcbcd' belongs to the language generated by the grammar.
dbad
The string 'dbad' belongs to the language generated by the grammar.
dad
The string 'dad' belongs to the language generated by the grammar.
dbabcaaad
The string 'dbabcaaad' belongs to the language generated by the grammar.
dbbabcaad
The string 'dbbabcaad' belongs to the language generated by the grammar.
aboba
The string 'aboba' does not belong to the language generated by the grammar.
dobby
The string 'dobby' does not belong to the language generated by the grammar.
```

Conclusion

To conclude this project, I have successfully applied the theoretical knowledge gained from lectures into a practical coding framework. By constructing a language from the given `Grammar` and translating it into a `Finite Automaton (FA)`, I have navigated the intricacies of class design and deepened my understanding of Formal Languages and Automata Theory.

This task not only involved developing classes and methods integral to grammars and automata but also emphasized the significance of regular grammars (type 3) during the conversion process to FA. This practical application has solidified my comprehension of these concepts and their relevance in computer science.

The project was a reflective and insightful synthesis of theory and practice, enhancing my problem-solving abilities and reinforcing the concept that complex theoretical constructs can be effectively transformed into working models. It was a testament to the power of applying theoretical foundations to solve practical problems and has contributed to both my academic and personal development.

P.S. I had fun and even managed to sleep a bit in the breaks between the lab and lab

Bibliography

[1] Formal Languages and Compiler Design Accessed February 14, 2024. https://else.fcim.utm.md/pluginfile.php/110457/mod_resource/content/0/Theme_1.pdf.

[2] Regular Language. Finite Automata. Accessed February 15, 2024. https://drive.google.com/file/d/1rBGyzDN5eWMXTNeUxLxmKsf7tyhHt9Jk/view