Ministry of Education and Research of the Republic of Moldova
Technical University of Moldova
Department of Software and Automation Engineering

REPORT

Laboratory work No. 1
Discipline: Formal Languages & Finite Automata
Topic: Regular grammars.

Done by: Barbarov Nadejda st.gr., FAF-221

Checked by:
asist.univ. Crețu Dumitru

Chișinău 2024

Theory

Transforming a grammar into a finite automaton is a key concept in formal language theory. A grammar defines the syntax or structure of a formal language through a set of production rules. These rules specify how symbols, both terminal and non-terminal, can be combined to form strings in the language. On the other hand, a finite automaton is a mathematical model of computation that recognizes or generates strings in a formal language. It consists of a finite set of states, an alphabet of input symbols, and transition functions that map states and input symbols to new states.

The process of transforming a grammar into a finite automaton involves mapping the grammar's production rules to transitions in the automaton. Each non-terminal symbol in the grammar corresponds to a state in the automaton, and transitions are determined by the production rules. For instance, if a rule generates a terminal symbol following a non-terminal, it leads to a transition between corresponding states. Similarly, if a rule generates a terminal symbol directly, it leads to a transition to a special final state. The resulting finite automaton reflects the structure of the language described by the grammar, aiding in language processing tasks such as parsing and pattern matching, and providing insight into the language's properties and structure.

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 description

In the provided implementation, the Grammar class defines a grammar consisting of non-terminal symbols (VN), terminal symbols (VT), and production rules (P).

```
class Grammar:
    def __init__(self):
        self.VN = {'S', 'R', 'L'}
        self.VT = {'a', 'b', 'c', 'd', 'e', 'f'}
        self.P = {
            'S': ['aS', 'bS', 'cR', 'dL'],
            'R': ['dL', 'e'],
            'L': ['fL', 'eL', 'd']
        }
```

The generate_strings method generates valid strings based on the grammar by recursively applying production rules starting from the start symbol S.

```
    def generate_string(self, symbol):
        import random
        if symbol in self.VT:
            return symbol
        else:
            production = random.choice(self.P[symbol])
            string = ''
            for char in production:
                if char in self.VT:
                    string += char
                else:
                    string += self.generate_string(char)
                return string
```

The to_finite_automaton method converts the grammar into a finite automaton, where states correspond to non-terminal symbols and transitions are determined by the production rules. If a production rule generates a terminal symbol directly, it transitions to a special final state x.

```
  def to_finite_automaton(self):
        states = self.VN.union({'x'})  # States consist only of non-terminal
symbols
        alphabet = self.VT
        transition_function = {}
        for state in self.VN:
            productions = self.P.get(state, [])
            for production in productions:
                if len(production) > 1:
                    transition_function[(state, production[0])] =
production[1]
                elif len(production) == 1:
                    transition_function[(state, production[0])] = 'x'


        start_state = 'S'
        final_states = {'x'}
        return FiniteAutomaton(states, alphabet, transition_function,
start_state, final_states)
```

The FiniteAutomaton class represents a finite automaton, comprising states, an alphabet, a transition function, a start state, and accept states.

```
class FiniteAutomaton:
    def __init__(self, states, alphabet, transition_function, start_state,
accept_states):
        self.states = states
        self.alphabet = alphabet
        self.transition_function = transition_function
        self.start_state = start_state
        self.accept_states = accept_states
        self.current_state = start_state
```

The transition_to_state_with_input method updates the current state based on the input symbol and the transition function.

```
    def transition_to_state_with_input(self, input_value):
```

```
        if (self.current_state, input_value) in self.transition_function:
            self.current_state =
self.transition_function[(self.current_state, input_value)]
        else:
            self.current_state = None
```

The in_accept_state method checks if the current state is an accept state.

```
  def in_accept_state(self):
        return self.current_state in self.accept_states
```

The go_to_initial_state method resets the current state to the start state.

```
  def go_to_initial_state(self):
        self.current_state = self.start_state
```

Finally, the run_with_input_list method processes a list of input symbols, updating the current state based on transitions and determining if the input is accepted based on the final state reached.

```
  def run_with_input_list(self, input_list):
        self.go_to_initial_state()
        for inp in input_list:
            self.transition_to_state_with_input(inp)
            if self.current_state is None:
                return False
            return self.current_state == 'x'
```

In the main_menu function, the grammar generates and prints five valid strings. Then, the grammar is converted into a finite automaton, and an input word is provided to test the automaton's functionality. The transitions of the automaton are displayed, and the acceptance of the input word is determined based on the automaton's behavior.

```
def main_menu():
    grammar = Grammar()
    valid_strings = grammar.generate_strings()
    print(" 5 generated valid strings:")
    for string in valid_strings:
        print(string)
    automaton = grammar.to_finite_automaton()
```

```
    word = 'cdefd'

    inp_program = list(word)

    print("Automaton constructed from grammar:")

    for key, value in automaton.transition_function.items():

        print(f"({key[0]}, '{key[1]}') -> {value}")

    print("Accepts '", word, "' ?",
automaton.run_with_input_list(inp_program))
```

Conclusions / Screenshots / Results

In this laboratory, we explored the process of transforming a grammar into a finite automaton, which is a fundamental concept in formal language theory. We implemented a grammar class to define the syntax of a formal language and generate valid strings based on production rules. Additionally, we created a finite automaton class to represent the grammar as a mathematical model capable of recognizing or generating strings in the language. Through this implementation, we gained insights into the structure of the language described by the grammar and its corresponding finite automaton. Overall, this laboratory provided valuable hands-on experience in understanding the relationship between grammars and finite automata, essential concepts in the study of formal languages and computational theory.

```
 5 generated valid strings:
ce
add
cdfd
bdfeed
aace
Automaton constructed from grammar:
(L, 'f') -> L
(L, 'e') -> L
(L, 'd') -> x
(S, 'a') -> S
(S, 'b') -> S
(S, 'c') -> R
(S, 'd') -> L
(R, 'd') -> L
(R, 'e') -> x
Accepts ' dd ' ? True
```

Figure 1. End results

References

1. **Github.** https://github.com/filpatterson/DSL_laboratory_works/tree/master/1_RegularGrammars
2. **Github.** https://github.com/nadea-b/LFA/blob/main/Lab_1.py