Ministerul Educaţiei și Cercetării al Republicii Moldova
Universitatea Tehnică a Moldovei

Facultatea Calculatoare, Informatică și Microelectronică

Course: Formal Languages & Finite Automata

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

Elaborated:

st. gr. FAF-221 Lupan Lucian

Verified:

asist. univ. Dumitru Crețu

Chişinău – 2024

Theory

A finite automaton is a simple machine that can recognize patterns. It can be used to define a regular language (i.e., a language with regular grammar). It takes a string of symbols as input characters and changes its state accordingly, and when it finds the required symbol, it transitions to a new state. It has 2 states – accept and reject – when a string is processed successfully in its final phase, it's accepted, otherwise the finite automaton rejects it.

Grammar is used to create patterns that a finite automaton may recognize. Grammar consists of:

- Non-terminals – symbols that signify intermediate transitions.
- Terminals – symbols that are used to form the words of the language.
- Production – set of rules that define possible transitions.

A finite automaton is a collection of 5-tuple (Q, ∑, δ, q0, F), where:

- Q: finite set of states
- ∑: finite set of the input symbol
- q0: initial state
- F: final state
- δ: Transition function

Finite automatons diverge into 2 types: DFAs (Deterministic Finite Automata) and NFAS (Non-deterministic Finite Automata). The difference between the 2 is in the way transitions are defined – NFAs can transition to 2 different states with the same input; whereas DFAs only have 1 transition defined for each unique input in a set state.

Objectives

1. Discover what a language is and what it needs to have in order to be considered a formal one;
2. 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)
3. 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

For this laboratory work, I used Python. In order to define the Grammar and the Finite Automaton, I created 2 classes which store info related to my variant number (18), in the form of variables which I passed around when parsing data.

```
class FiniteAutomaton:
    Q = []
    Sigma = []
    Delta = []
    q0 = ''
    F = []
class Grammar:
    VN = []
    VT = []
    P = []
```

To achieve the goals of this laboratory work, methods were defined in order to create the desired functionality. Generating the strings is done inside the Grammar class, through the use of the Random library, creating a random sequence of a valid string for each function call and returning the transitions required to create the specific string.

```
def generate_string(self) -> tuple:
    generated_string = "S"
    transition_prerequisite = [ch[0] for ch in self.P]
    pattern = re.compile(r'->(.+)$')
    creation_transitions = []

    # Checking if we still have non-terminal characters
    while any(ch in self.VN for ch in generated_string):
        # Choosing a non-terminal character at random
        random_non_terminal = random.choice([ch for ch in generated_string if ch in self.VN])
        # Choosing a random transition for the non-terminal character
        random_transition_pos = random.choice([i for i, tr in enumerate(transition_prerequisite) if tr in random_non_terminal])
        # Getting the characters which we'll be replacing the non-terminal with using the pattern
        replacing_string = pattern.search(self.P[random_transition_pos]).group(1)
        # Replacing the random non-terminal with the transition
        generated_string = generated_string.replace(random_non_terminal, replacing_string)
        creation_transitions.append(str(random_non_terminal + "->" + replacing_string))

    return generated_string, creation_transitions
```

Next up, I had to think up a way to convert an object of Grammar type to one of Finite Automaton. This is simply solved through the use of another method, which takes the Grammar class attributes and converts them into Finite Automaton attributes. Thus, by creating an instance inside of the method, we can initialize the instance inside of the function and return the instance to the user.

```
def finite_automaton_conversion(self) -> FiniteAutomaton:
    finite_automaton = FiniteAutomaton()
    # Start state
    finite_automaton.q0 = 'S'
    # Set of states
    finite_automaton.Q = self.VN
    # Alphabet
    finite_automaton.Sigma = self.VT
    # Accepting states
    finite_automaton.F = self.VT
    # Set of rules
    finite_automaton.Delta = self.P

    return finite_automaton
```

Lastly, I needed to check if a string belongs to the language. Due to the fact that my language is an NFA, rather than a DFA (because S can transition into both ‘A’ and ‘B’ through ‘a’), this task was more challenging. However, I noticed a pattern when it came to creating the words – all of them would have 3 combinations of strings: ‘aaa’, ‘aab’ and ‘ab’. Thus, by analyzing a word to see if it didn’t have those combinations, I was able to determine whether it belonged or not to my specific language, and if it did, I would find out its transitions required to create that word.

```
def string_belong_to_language(self, input_string) -> tuple:
    backup_string = input_string

    while input_string != "":
        if "aaa" in input_string:
            input_string = input_string.replace("aaa", "")
        elif "aab" in input_string:
            input_string = input_string.replace("aab", "")
        elif "ab" in input_string:
            input_string = input_string.replace("ab", "")
        else:
            return False, []
    creation_transitions = self.generate_transitions_to_create_word(backup_string)
    return True, creation_transitions
```

Results

```
Generated string[1] = abaababababaaa
Transitions to generate the string: ['S->aA', 'A->bS', 'S->aB', 'B->aC', 'C->bS', 'S->aA', 'A->bS', 'S->aA', 'A->bS', 'S->aB', 'B->aC', 'C->a']
Generated string[2] = abaabaaa
Transitions to generate the string: ['S->aA', 'A->bS', 'S->aB', 'B->aC', 'C->bS', 'S->aB', 'B->aC', 'C->a']
Generated string[3] = abababaaa
Transitions to generate the string: ['S->aA', 'A->bS', 'S->aA', 'A->bS', 'S->aA', 'A->bS', 'S->aB', 'B->aC', 'C->a']
Generated string[4] = aaa
Transitions to generate the string: ['S->aB', 'B->aC', 'C->a']
Generated string[5] = ababaabaaa
Transitions to generate the string: ['S->aA', 'A->bS', 'S->aA', 'A->bS', 'S->aB', 'B->aC', 'C->bS', 'S->aB', 'B->aC', 'C->a']
```

```
Is the string = [abababaababababaaa] valid for the language?
True
Transitions for obtaining the word: ['S->aA', 'A->bS', 'S->aA', 'A->bS', 'S->aA', 'A->bS', 'S->aB', 'B->aC', 'C->bS', 'S->aA', 'A->bS', 'S->aA', 'A->bS', 'S->aB', 'B->aC',

Is the string = [aaa] valid for the language?
True
Transitions for obtaining the word: ['S->aB', 'B->aC', 'C->a']

Is the string = [c] valid for the language?
False

Is the string = [abbaaa] valid for the language?
False
```

Conclusions

After finishing this laboratory work, I learnt how to implement my own Finite Automaton and define the Grammar it would parse through the use of a programming language, rather than just defining it on paper.

I also gained useful knowledge about the applications of Finite Automaton and gained some insights about their use cases, being used in order to detect patterns over an input data set and accept/reject a string based on those patterns.

Last but not least, this laboratory work will (at least hopefully) be useful for future lab work implementations, as well as our ELSD classes, helping us define our own language's grammar and rules.

References:

1. Finite automata. [online]. [accessed 17.02.2024]. Available: https://www.javatpoint.com/finite-automata
2. Introduction to Finite Automata. [online]. [accessed 17.02.2024]. Available: https://www.geeksforgeeks.org/introduction-of-finite-automata/