Ministerul Educaţiei și Cercetării al Republicii Moldova
Universitatea Tehnică a Moldovei
Facultatea Calculatoare, Informatică și Microelectronică

Laboratory work nr. 2
Course: Formal languages and finite automata
Topic: Determinism in Finite Automata. Conversion from NDFA to DFA. Chomsky Hierarchy.
Variant-20

Elaborated:
st. gr. FAF-223 Pereteatcu Arina

Verified:
asist. univ. Cretu Dumitru

Chişinău - 2024

Theoretical considerations

The most basic machine that can identify patterns is a finite automata (FA). This is employed to describe a Regular Language. It is also employed in the analysis and identification of natural language expressions. An abstract machine with five elements, or tuples, is known as a finite automata or finite state machine. It has a set of states and rules for changing between them, but how it does so is dependent on the input symbol that is applied. The input string can be allowed or refused based on the states and rules. In essence, it's an abstract representation of a digital computer that reads an input string and modifies its internal state in response to the input symbol that's being read at that moment. Each automaton has a language, or a set of strings that it can understand.

A finite or infinite collection of strings over a finite number of symbols is the definition of a formal language in computer science. An 'alphabet' is a finite set of symbols. The formal language is made up of the ordered strings that are produced with this alphabet according to the established grammar rules.

Alphabet: In the scope of formal languages, an alphabet, often denoted by the Greek letter Σ, is simply a finite set of distinct symbols.

String: A string is a finite sequence of symbols selected from an alphabet. It is notable that the order of symbols matters in a string. An empty string, denoted often as lambda, is a string with zero symbols.

Grammar: Grammar is a set of formal rules that governs the combination of symbols to compose strings in a formal language. The structural nature of these string-producing rules is intrinsically tied into the classification of formal languages: regular, context-free, context-sensitive, and recursively enumerable.

Finite state machines known as *deterministic finite automata, or DFAs*, parse character strings via a sequence that is specifically determined for each string before accepting or rejecting them. The word "deterministic" describes the uniqueness of every string and consequently every state sequence. Every input symbol in a DFA moves to the next state that can be identified when it is parsed using a DFA automaton. Because there are only a finite number of states that these machines can attain, they are known as finite machines. Only when a finite automaton satisfies both requirements is it referred to as deterministic.

DFAs are actually composed of five components, and they are frequently represented by a 5-tuple, which is a set of five symbols. These elements consist of:

1. limited quantity of states

2. alphabetic system of symbols that has a limited quantity
3. function that manages each symbol's transition between states
4. first-start phase in which the first input is provided or handled
5. a final state or states that are deemed acceptable.

Non-deterministic finite automata are referred to as NFAs. Building an NFA is simpler than creating a DFA for a given regular language. When there are numerous paths for a given input from one state to the next, the finite automata are referred to as NFAs. While not all NFAs are DFA, all NFAs can be converted into DFA. With the following two exceptions—it has numerous future states and an ε transition—NFA is defined similarly to DFA.

Objectives:

1. Understand what an automaton is and what it can be used for.
2. Continuing the work in the same repository and the same project, the following need to be added:

   a. Provide a function in your grammar type/class that could classify the grammar based on Chomsky hierarchy.

   b. For this you can use the variant from the previous lab.
3. According to your variant number (by universal convention it is register ID), get the finite automaton definition and do the following tasks:

   a. Implement conversion of a finite automaton to a regular grammar.

   b. Determine whether your FA is deterministic or non-deterministic.

   c. Implement some functionality that would convert an NDFA to a DFA.

   d. Represent the finite automaton graphically (Optional, and can be considered as a bonus point):

   You can use external libraries, tools or APIs to generate the figures/diagrams.

   Your program needs to gather and send the data about the automaton and the lib/tool/API return the visual representation.

Implementation Description

My laboratory work was done in Python language. Variant 20:

Q = {q0,q1,q2,q3},

∑ = {a,b,c},

F = {q3},

δ(q0,a) = q0,

δ(q0,a) = q1,

δ(q2,a) = q2,

δ(q1,b) = q2,

δ(q2,c) = q3,

δ(q3,c) = q3.

Code

```
class FiniteAutomaton:
    def __init__(self):
        self.Q = ['q0','q1','q2','q3']
        self.Sigma = ['a','b','c']
        self.Delta = {
            ('q0', 'a'): ['q0', 'q1'],
            ('q1', 'b'): ['q2'],
            ('q2', 'c'): ['q3'],
            ('q3', 'c'): ['q3'],
            ('q2', 'a'): ['q2']
        }
        self.q0 = 'q0'
        self.F = ['q3']
```

First there is a class “FiniteAutomaton” that represents a finite automaton.

```
def convert_to_grammar(self):
    S = self.Q[0]
    Vn = self.Q
    Vt = self.Sigma
    P = []
    for state in self.Q:
        for symbol in self.Sigma:
            if (state, symbol) in self.Delta:
                next_states = self.Delta[(state, symbol)]
                for next_state in next_states:
                    P.append((state, symbol, next_state))
    for final_state in self.F:
        P.append((final_state, '', 'e'))
```

Then I added a method “convert_to_grammar” in the class, it converts the finite automaton into a grammar. We have a starting symbol S, non-terminal symbol Vn and terminal symbol VT and P for production rules, which includes a nested loop that iterates over each state in Q and each input symbol in Sigma. For each state-symbol pair, it checks if there is a transition defined in the Delta function, if there is one, it retrieves the next state for the state-symbol pair and creates the production rules. After that, it checks for final states, for each one it adds an epsilon-transition to the grammar and transition to an empty string. The method maps the transitions of the finite automaton to production rules of a grammar, and captures the behaviour of the automaton in a language that grammars can describe.

```
def check_deterministic(self):
    for _, value in self.Delta.items():
        if len(value) > 1:
            return False
    return True
```

This method is responsible for determining whether the finite automaton is deterministic. It iterates over the items in Delta dictionary(the transition function of the finite automaton), and for each key-value pair it checks the length of the list of next states. If length is greater than 1, then there are multiple possible transitions for the same state and input symbol. For DFA, each state has only one next state for each input symbol.

```
def nfa_to_dfa(self):
    input_symbols = self.Sigma
    initial_state = self.q0
    states = []
    final_states = self.F

    transitions = {}
    new_states = [initial_state]
    while new_states:
        state = new_states.pop()
        if state not in transitions:
            transitions[state] = {}
            for el in input_symbols:
                next_states = set()
                for s in state.split(','):
                    if (s, el) in self.Delta:
                        next_states.update(self.Delta[(s, el)])
                next_states = ','.join(sorted(list(next_states)))
                transitions[state][el] = next_states
                if next_states not in transitions and next_states != '':
                    new_states.append(next_states)

    states = list(transitions.keys())
```

```python
    print(f"Q = {states}")
    print(f"Sigma = {input_symbols}")
    print(f"Delta = {transitions}")
    print(f"q0 = {initial_state}")
    print(f"F = {final_states}")
```

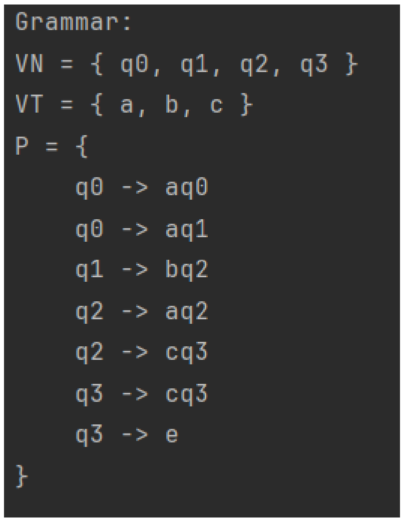
This method is responsible for converting non-deterministic finite automaton into a deterministic finite automaton. Here we have a while loop that, for each state, it checks if it has already been processed, if not, it initializes an entry in the transition dictionary for the given state. For each input set it computes the set of next states for the given state and input symbol combination by iterating over each state in the NFA transition function. Overall, it explores the states and transitions of the NFA to construct and equivalent DFA.

```python
class Grammar:
    def __init__(self, S, Vn, Vt, P):
        self.S = S
        self.Vn = Vn
        self.Vt = Vt
        self.P = P

    def show_grammar(self):
        print("VN = {", ', '.join(map(str, self.Vn)), '}')
        print("VT = {", ', '.join(map(str, self.Vt)), '}')
        print("P = { ")
        for el in self.P:
            a, b, c = el
            print(f"    {a} -> {b}{c}")
        print("}")


# main
finite_automaton = FiniteAutomaton()
grammar = finite_automaton.convert_to_grammar()
print("Grammar:")
grammar.show_grammar()
print()

if not finite_automaton.check_deterministic():
    print('NDFA\n')
else:
    print('DFA\n')

print("Deterministic Finite Automaton:")
finite_automaton.nfa_to_dfa()
```

Finally, I added a class Grammar and "show_grammar" method that prints out the grammar in a readable way.

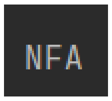
The main execution contains a calling of method "convert_to_grammar" to generate the representation of the finite automaton, the "show_grammar" is called to print out the grammar, it is checked if the automaton is deterministic or not by calling "check _deterministic" method and finally shows the results.

Outputs



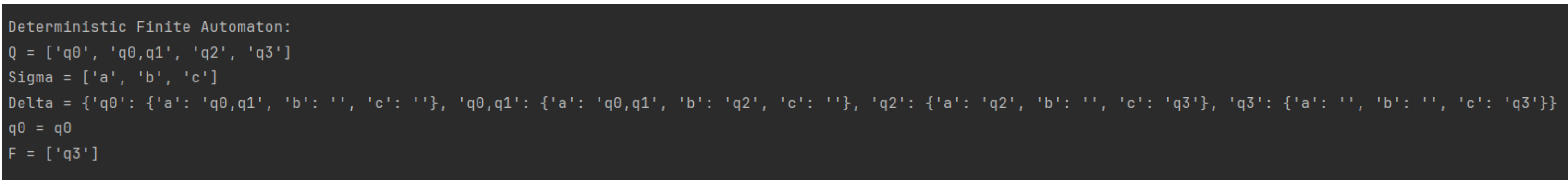
```
Grammar:
VN = { q0, q1, q2, q3 }
VT = { a, b, c }
P = {
    q0 -> aq0
    q0 -> aq1
    q1 -> bq2
    q2 -> aq2
    q2 -> cq3
    q3 -> cq3
    q3 -> e
}
```

Figure1. Grammar



```
NFA
```

Figure2. Check if DFA or NFA



```
Deterministic Finite Automaton:
Q = ['q0', 'q0,q1', 'q2', 'q3']
Sigma = ['a', 'b', 'c']
Delta = {'q0': {'a': 'q0,q1', 'b': '', 'c': ''}, 'q0,q1': {'a': 'q0,q1', 'b': 'q2', 'c': ''}, 'q2': {'a': 'q2', 'b': '', 'c': 'q3'}, 'q3': {'a': '', 'b': '', 'c': 'q3'}}
q0 = q0
F = ['q3']
```

Figure3. Converting

Conclusions

To sum up, this lab work has given me a thorough understanding of formal languages and finite automata. Deterministic or non-deterministic finite automata are basic building blocks for language analysis and pattern recognition. Formal languages are strings created from finite sets of symbols according to grammar rules. By studying these strings, we can learn more about the structure and families of languages. Deterministic finite automata (DFAs) and non-deterministic finite automata (NFAs) are two different categories that show off the flexibility and computing capacity of these abstract machines. While NFAs allow for numerous pathways from one state to another, DFAs follow a fixed sequence of states for every input string. In addition, the goals set forth for this lab emphasize the useful extensions and applications of automata theory, such as the creation of algorithms for converting one kind of automata to another and the classification of grammars according to the Chomsky hierarchy. Students get a deeper grasp of automata theory and its applicability in a variety of computational areas by completing these objectives and exercises. They learn how to create and analyze finite automata and formal languages through practice and experimentation, which opens up new avenues for theoretical computer science research.