

MINISTRY OF EDUCATION, CULTURE AND RESEARCH OF REPUBLIC OF MOLDOVA
TECHNICAL UNIVERSITY OF MOLDOVA
FACULTY OF COMPUTERS, INFORMATICS AND MICROELECTRONICS
DEPARTMENT OF SOFTWARE ENGINEERING AND AUTOMATICS

Formal Languages and Finite Automata

Laboratory work 1 : Regular Grammars and Finite Automata

Variant 13

Elaborated:

st.gr. FAF-211

Gazea Sandu

Verified:

asist.univ.

Vasile Drumea

Chişinău, 2023

Content

| | |
|-------------------------|----------|
| Introduction | 3 |
| Objectives | 4 |
| Implementation | 5 |
| 1 Implementation | 5 |
| 1.1 Code: | 5 |
| 1.2 Screenshot: | 8 |
| Conclusions | 9 |

Introduction

Regular Grammars and Finite Automata are fundamental concepts in theoretical computer science and play a crucial role in the study of formal languages and automata theory.

A Regular Grammar is a set of production rules used to generate strings in a formal language. A formal language is a set of strings made up of symbols from a given alphabet, and regular grammars describe a subset of those languages that can be recognized by Finite Automata. Regular grammars are defined by a set of production rules, which specify how to rewrite a symbol in the language to a string of other symbols. The production rules are applied repeatedly to generate all the valid strings in the language.

A Finite Automaton is a mathematical model used to recognize patterns in strings, and it consists of a set of states, an initial state, a set of accepting states, and a transition function that maps from a current state and input symbol to a next state. Finite automata are classified as either deterministic or non-deterministic, depending on whether or not multiple transitions are possible for a given input symbol and state.

The relationship between regular grammars and finite automata is very close, and they are often used interchangeably. In particular, every regular grammar can be associated with a deterministic finite automaton, and every deterministic finite automaton can be associated with a regular grammar.

Regular grammars and finite automata have numerous applications in computer science, including parsing, pattern matching, lexical analysis, and text processing. They provide a foundation for more advanced topics, such as context-free grammars, pushdown automata, and Turing machines, which are essential in the study of computational complexity and algorithm design.

Objectives

1. Understand 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. I said project because usually at lab works, I encourage/impose students to treat all the labs like stages of development of a whole project. Basically you need to do the following:
 - (a) Create a local remote repository of a VCS hosting service (let us all use Github to avoid unnecessary headaches);
 - (b) Choose a programming language, and my suggestion would be to choose one that supports all the main paradigms;
 - (c) Create a separate folder where you will be keeping the report. This semester I wish I won't see reports alongside source code files, fingers crossed;
3. According to your variant number (by universal convention it is register ID), get the grammar definition and do the following tasks:
 - (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 an 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;

1 Implementation

1. You can use 2 classes in order to represent the 2 main object which are the grammar and finite automaton. Additional data model, helper classes etc. can be added but should be used (i.e. you shouldn't have source code file that are not used).
2. In order to show the execution you can implement a client class/type, which is just a "Main" class/type in which you can instantiate the types/classes. Another approach would be to write unit tests if you are familiar with them.

1.1 Code:

```
from FiniteAutomaton import FiniteAutomaton
from Grammar import Grammars
class Main:

    def __init__(self):
        self productions = {
            'S': ['aB'],
            'B': ['aD', 'bB', 'cS'],
            'D': ['aD', 'bS', 'c'],
        }
        self.start_symbol = 'S'
        self.grammar = Grammars(self productions, self.start_symbol)
        self.finite_automaton = self.grammar.to_finite_automaton()
        self.automaton = FiniteAutomaton

    def generate_strings(self, num_strings):
        for i in range(num_strings):
            string = self.grammar.generate_string()

            print(string)

if __name__ == '__main__':
    main = Main()
    main.generate_strings(6)
```

```

automatons = main.grammar.to_finite_automaton()
automaton = {
    'states': {'q0', 'q1', 'q2', 'q3', 'q4', 'q5'},
    'alphabet': {'a', 'b', 'c'},
    'transitions': {
        'q0': {'a': 'q1'},
        'q1': {'a': 'q4', 'b': 'q2', 'c': 'q5'},
        'q2': {'a': 'q3', 'b': 'q2', 'c': 'q5'},
        'q3': {'a': 'q3', 'b': 'q4', 'c': 'q5'},
        'q4': {'a': 'q3', 'b': 'q2', 'c': 'q5'},
        'q5': {'a': 'q5', 'b': 'q0', 'c': 'q5'}
    },
    'start_state': 'q0',
    'final_states': {'q3', 'q5'}
}
checker = FiniteAutomaton(automaton)
checker.check_strings(['aca', 'abaca', 'ababaa', 'aa',
    'abababa', 'acacacacacaca', 'baba'])
print(automatons)

```

```
import random
```

```

class Grammars:
    def __init__(self, productions, start_symbol):
        self.productions = productions
        self.start_symbol = start_symbol

    def generate_string(self):
        return self._generate_string(self.start_symbol)

    def _generate_string(self, symbol):
        if symbol not in self.productions:
            return symbol

```

```

        production = random.choice(self productions[symbol])
        return ''.join(self._generate_string(s) for s in production)

def to_finite_automaton(self):
    start_state = 0
    automaton = {start_state: {}}
    state_count = 1

    for symbol in self productions:
        for production in self productions[symbol]:
            current_state = start_state
            for s in production:
                if s not in automaton[current_state]:
                    # Add a new state and transition
                    automaton[current_state][s] = state_count
                    automaton[state_count] = {}
                    state_count += 1

                current_state = automaton[current_state][s]
            # Add a transition to the final state for the last symbol
            if current_state not in automaton:
                automaton[current_state] = {}
            automaton[current_state][''] = start_state

    return automaton

class FiniteAutomaton:
    def __init__(self, automaton):
        self.states = automaton['states']
        self.alphabet = automaton['alphabet']
        self.transitions = automaton['transitions']
        self.start_state = automaton['start_state']
        self.final_states = automaton['final_states']

    def check_string(self, string):

```

```

current_state = self.start_state

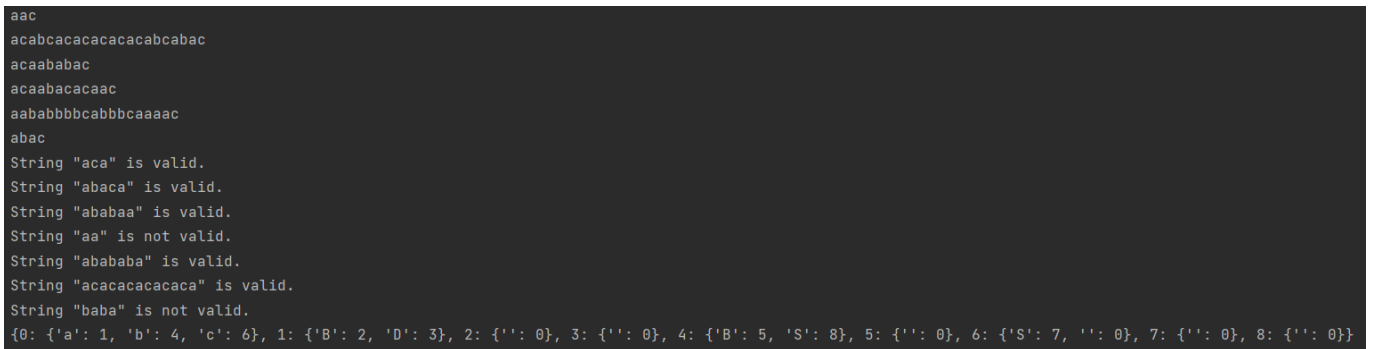
for symbol in string:
    try:
        current_state = self.transitions[current_state][symbol]
    except KeyError:
        return False

return current_state in self.final_states

def check_strings(self, strings):
    for string in strings:
        if self.check_string(string):
            print(f'String "{string}" is valid.')
        else:
            print(f'String "{string}" is not valid.')

```

1.2 Screenshot:



```

aac
acabcacacacacacababac
acaababac
acaabacacaac
aababbbbcabbbcaaac
abac
String "aca" is valid.
String "abaca" is valid.
String "ababaa" is valid.
String "aa" is not valid.
String "abababa" is valid.
String "acacacacacaca" is valid.
String "baba" is not valid.
{0: {'a': 1, 'b': 4, 'c': 6}, 1: {'B': 2, 'D': 3}, 2: {'': 0}, 3: {'': 0}, 4: {'B': 5, 'S': 8}, 5: {'': 0}, 6: {'S': 7, '': 0}, 7: {'': 0}, 8: {'': 0}}

```


Conclusions

To comprehend the abilities and constraints of computers, the study of regular grammars and finite automata is crucial. This understanding allows computer scientists to create efficient algorithms for tasks such as text processing, lexical analysis, and pattern matching, which are used in real-world applications like spam filters, data compression, and search engines. Regular grammars and finite automata are essential for computer systems to interact in a standardized manner, which is critical for network communication.

Moreover, studying regular grammars and finite automata has theoretical implications in computer science. Knowing the limitations of these models helps develop more powerful models like context-free grammars, pushdown automata, and Turing machines that can solve more complex problems. Regular grammars and finite automata have connections to other areas of mathematics, such as topology, group theory, and algebra, which provide insight into the relationship between computation and mathematical concepts.

To summarize, regular grammars and finite automata are fundamental concepts in computer science, having practical applications in natural language processing, compilers, and artificial intelligence. They are vital tools to understand computer capabilities and have links to other mathematical fields. With their continued study, regular grammars and finite automata will undoubtedly have a critical role in shaping the future of computer science.