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

Elaborated: st.gr. FAF-213, Cozlov Valeria

Verified: asist.univ. Vasile Drumea

Chișinău, 2023

Introduction

A Regular Grammar is a set of rules in a formal language that generates strings. A regular grammar, in other words, is a way of describing how to produce strings using a finite set of symbols from an alphabet. These symbols could be anything from letters to numbers to punctuation marks. The formal language is defined as a collection of strings composed of symbols from the given alphabet.

Regular grammars describe a subset of the formal languages that Finite Automata can recognize. A Finite Automaton is a mathematical model that can detect patterns in strings. It is made up of a number of states, including an initial state, a number of accepting states, and a transition function that maps from the current state and input symbol to the next state.

Finite automata are classified into two types: deterministic and non-deterministic. Each input symbol and state has only one possible transition in a deterministic finite automaton. A non-deterministic finite automaton, on the other hand, may have multiple possible transitions for a given input symbol and state.

Regular grammars and finite automata have a very close relationship. Every deterministic finite automaton can be associated with a regular grammar, and every regular grammar can be associated with a deterministic finite automaton. Regular grammars and finite automata are frequently used interchangeably in this context.

In computer science, regular grammars and finite automata have numerous practical applications. They're frequently used in parsing, which is the process of analyzing a text to figure out its grammatical structure. They are also used in pattern matching, which is the process of looking for a specific pattern in a text. Furthermore, regular grammars and finite automata are required for lexical analysis, which is the process of breaking down a text into a series of tokens for processing. Finally, these ideas are useful in text processing, which involves manipulating text to accomplish various tasks.

Regular grammars and finite automata are also used to lay the groundwork for more advanced topics like context-free grammars, pushdown automata, and Turing machines. These ideas are critical for understanding computational complexity and algorithm design. Computer scientists can develop more efficient algorithms and solve more complex problems by understanding regular grammars and finite automata.

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 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

1. You can use two classes to represent the two main objects, which are the grammar and the finite automaton. Additional data models, helper classes, and so on can be added but must be used (i.e., there should be no source code files that are not used).
2. To demonstrate the execution, create a client class/type, which is simply a "Main" class/type in which you can instantiate the types/classes. If you are familiar with unit tests, another approach would be to write them.

Code:

Main.py

```
from FiniteAutomaton import FiniteAutomaton
from Grammar import Grammar


class Main:

    def __init__(self):
        # Create a context-free grammar by starting the class with a set of
productions.
        self.productions = {
            'S': ['aB'],
            'B': ['aD', 'bB', 'cS'],
            'D': ['aD', 'bS', 'c'],
        }
        self.start_symbol = 'S'
        self.grammar = Grammar(self.productions, self.start_symbol)
        self.finite_automaton = self.grammar.create_finite_automaton()
        self.automaton = FiniteAutomaton

    # Create a function that will generate strings from the grammar.
    def Create_string(self, n):
        for i in range(n):
            string = self.grammar.create_string()
            print(string)


if __name__ == '__main__':
    main = Main()
    main.Create_string(6)
    automatons = main.grammar.create_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'}
    }
    verification = FiniteAutomaton(automaton)
    verification.verify_strings(['abababa', 'babac', 'ababaa', 'aa', 'aca', 'acacacab',
'cabca'])
    print('Automaton:', automatons)
```

Grammar.py

```
import random
class Grammar:
    def __init__(self, productions, start_symbol):
        self.productions = productions
        self.start_symbol = start_symbol

    # Create a function that will generate a string from the grammar.
    def create_string(self):
        return self.generate_string(self.start_symbol)

    # DCreate a recursive function that will generate a string from the grammar.
    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 create_finite_automaton(self):
        start_state = 0
        automatons = {start_state: {}}
        count_state = 1

        for symbol in self.productions:
            for production in self.productions[symbol]:
                current_state = start_state
                for s in production:
                    if s not in automatons[current_state]:
                        # Add a new state and transition
                        automatons[current_state][s] = count_state
                        automatons[count_state] = {}
                        count_state += 1
                    current_state = automatons[current_state][s]
                # Add a transition to the last symbol's final state.
                if current_state not in automatons:
                    automatons[current_state] = {}
                automatons[current_state][''] = start_state

        return automatons
```

FiniteAutomaton.py

```
class FiniteAutomaton:
    # Initialize the given automaton's states, alphabet, transitions, start state, and
final states.
    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 verify_string(self, string):
        current_state = self.start_state
        # Follow the corresponding transition if it exists for each symbol in the
string.
        for symbol in string:
            try:
                current_state = self.transitions[current_state][symbol]
            except KeyError:
                # The string is invalid if the transition does not exist.
                return False

        return current_state in self.final_states

    def verify_strings(self, strings):
```

```
        print(f'-------------------------------------------------------------------------
-')
        for string in strings:
            if self.verify_string(string):
                print(f'String "{string}" is accurate.')
                print(f'-----------------------------------------_----------------')
            else:
                print(f'String "{string}" isn’t accurate.')
                print(f'-------------------------------------------------------------')
```

Console Output:

```
abbaababaabaaaaaaaabaaaaababbbcacabcaaabacaaababbbcaac
aabacababaaac
acacabcabbbac
abbcacacaaaabacacaabacacaaac
abbaaabacacacaaabaabacaababcabaac
aababac
-------------------------------------------------------------------------
String "abababa" is accurate.
-------------------------------------------------------------------------
String "babac" is not accurate.
-------------------------------------------------------------------------
String "ababaa" is accurate.
-------------------------------------------------------------------------
String "aa" is not accurate.
-------------------------------------------------------------------------
String "aca" is accurate.
-------------------------------------------------------------------------
String "acacacab" is not accurate.
-------------------------------------------------------------------------
String "cabca" is not accurate.
-------------------------------------------------------------------------
Automaton: {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}}
```

Conclusion

For an understanding of the potential and limitations of computers, knowledge of regular grammars and finite automata is essential. These tools are used by computer scientists to create efficient algorithms for tasks like text processing, pattern matching, and lexical analysis. These algorithms are frequently used in practical applications like data compression, spam filters, and search engines. Additionally, automata and regular languages allow for standardized communication between computer systems, which is essential for network communication.

The study of finite automata and regular grammars has important theoretical ramifications as well. Computer scientists can create new computation models that are more powerful and capable of solving challenging problems if they are aware of the limitations of these models. For instance, pushdown automata and context-free grammars enhance the capabilities of finite automata and regular grammars, enabling the recognition of more intricate patterns. These models serve as the foundation for more complex models like Turing machines, which can address any issue that a computer is capable of addressing.

Regular grammars and finite automata have connections to other branches of mathematics, including group theory, topology, and algebra, in addition to their useful uses in computer science. These connections shed light on the fundamentals of computation and how it relates to other ideas in mathematics.

In conclusion, regular grammars and finite automata are fundamental ideas in computer science with numerous real-world applications in areas like artificial intelligence, compilers, and natural language processing. They have theoretical ramifications for computer science and are also crucial tools for understanding the potential and constraints of computers. The study of regular grammars and finite automata remains an active area of research and will undoubtedly shape the future of computer science.