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-211 Andrei Ceban
Verified:
asist.univ. Vasile Drumea

Chișinău, 2023

Content

Introduction

Regular grammars and finite automata are both formalisms in theoretical computer science that are used to define and recognize regular languages.

A regular grammar is a set of production rules that generate a regular language. The production rules consist of a non-terminal symbol on the left-hand side and a string of terminals and non-terminals on the right-hand side. The grammar generates strings by starting with a non-terminal symbol and applying the production rules repeatedly until only terminal symbols remain.

A finite automaton, on the other hand, is a mathematical model that recognizes regular languages. It is a finite state machine that reads an input string and transitions between a set of states based on the input symbols. The automaton accepts the input string if it reaches an accepting state after reading the entire input.

Regular grammars and finite automata are closely related, and it is known that a language is regular if and only if it can be generated by a regular grammar or recognized by a finite automaton. This connection is known as the Chomsky hierarchy, which classifies formal grammars into four types based on their generative power. Regular grammars and finite automata fall into the lowest level of this hierarchy, known as type 3.

In computer science and theoretical linguistics, a language is a set of strings or sequences of symbols that share a common structure or pattern. These symbols could be any discrete elements, such as letters, digits, words, or any other abstract entities.

A formal language is a language that has been defined in a precise and unambiguous manner using a set of formal rules. It is typically used to describe and analyze complex systems, such as programming languages, natural languages, or mathematical expressions.

To convert a grammar to a finite automaton, we can use a well-known algorithm called the subset construction algorithm. The algorithm takes as input a regular grammar G, and constructs a deterministic finite automaton (DFA) M that recognizes the same language as G.

Regular grammars and finite automata are both formalisms used in theoretical computer science to define and recognize regular languages. Here are some differences between these two formalisms:

1. Structure: A regular grammar is a set of production rules that generate a regular language, whereas a finite automaton is a state machine that recognizes a regular language by reading an input string .

2. Generative power: A regular grammar is more expressive than a finite automaton in the sense that it can generate more than one string for a given regular language. On the other hand, a finite automaton is more efficient than a regular grammar in the sense that it requires less memory to recognize regular languages.

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;

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:

```
import Grammar
#Varianta 8

# Set of terminal symbols
VT = {'a', 'b', 'c', 'd', 'e'}

# Set of non-terminal symbols
VN = {'S', 'D', 'E', 'J'}

# Set of productions, rules or transitions
P = {"S": ["aD"],"D": ["dE", "bJ", "aE"],"J": ["cS"],"E":
["eX", "aE"],"X": " " }

# Initiate the grammar calling the constructor with all 3 params
grammar = Grammar.Grammar(VN, VT, P)

# Initiate the finite automaton
finite_automaton = grammar.from_grammar_to_finite_automaton()

# Generate 5 string
word_1 = grammar.generate_string()
word_2 = grammar.generate_string()
word_3 = grammar.generate_string()
word_4 = grammar.generate_string()
```

```
word_5 = grammar.generate_string()

# Print the strings including the  validation
print("word_1 = {x} ".format(x = word_1) + " | "
+ str(finite_automaton.check_string(word_1)))
print("word_2 = {x} ".format(x = word_2) + " | "
+ str(finite_automaton.check_string(word_2)))
print("word_3 = {x} ".format(x = word_3) + " | "
+ str(finite_automaton.check_string(word_3)))
print("word_4 = {x} ".format(x = word_4) + " | "
+ str(finite_automaton.check_string(word_4)))
print("word_5 = {x} ".format(x = word_5) + " | "
+ str(finite_automaton.check_string(word_5)))



import FiniteAutomaton
import random

class Grammar:
    # Constructor of Grammar class
    def __init__(self, VN, VT, P):
        self.VT = VT
        self.VN = VN
        self.P = P

    # Generate the strings
    def generate_string(self):
        # Start of the string
        string = "S"
        # Set of final states
        final_state = " "

        # Create a random string
        while string[-1] not in final_state:
```

```
            arr = []
            #go trough items and get the keys and values
            for key, value in self.P.items():
                if key == string[-1]:
                    #move the value in array
                    arr += value
            if not arr:
                return None
            production = random.choice(arr)
            #Reverse string, count backwards
            string = string[:-1] + production
        return string

    # Convert from grammar to finite automaton, sending all the params
    def from_grammar_to_finite_automaton(self):
        # Initiate finite set of states
        Q = set(self.VN)
        # Initiate the alphabet with all the characters
        Sigma = set(self.VT)
        # Initiate the initial state from the list
        q0 = "S"
        # Initiate the set of final states from the list
        F = "X"
        # Initiate the transition function
        arr = {}
        for key, value in self.P.items():
            for symbol in value:
                if (key, symbol[0]) in arr :
                    arr[(key, symbol[0])].append(symbol[1:])
                else:
                    arr[(key, symbol[0])] = [symbol[1:]]
        print("Automation :", arr)

        # Call the constructor of Finite Automaton with all the params
```

```
        return FiniteAutomaton.FiniteAutomaton(Q, Sigma, arr, q0, F)


class FiniteAutomaton:
# Constructor of Finite_Automaton class
def __init__(self, Q, Sigma, arr, q0, F):
    self.Q = Q
    self.Sigma = Sigma
    self.arr = arr
    self.q0 = q0
    self.F = F

# Check the word if is valid
def check_string(self, word):
    index = 0
    # Set the current state to the initial state
    Q0 = self.q0

    # For each character in the word, find the next state using
    #the arr function
    for character in word:
        if ((Q0[0], character) in self.arr) and (index != len(word)-1):
            Q0 = self.arr[(Q0[0], character)]
        elif index == len(word) - 1:
            if len(Q0) > 1:
                # If the current state is in the set of final states (X),
                #return True, else return False
                if Q0[len(Q0)-1] in self.F:
                    return True
                else:
                    return False
            else:
                if Q0[0] in self.F:
                    return True
```

```
                else:
                    return False
            index = index + 1
```

1.2 Screenshot:

```
Automation : {('S', 'a'): ['D'], ('D', 'd'): ['E'], ('D', 'b'): ['J'], ('D', 'a'): ['E'],
 ('J', 'c'): ['S'], ('E', 'e'): ['X'], ('E', 'a'): ['E'], ('X', ' '): ['']}
word_1 = abcaae   | True
word_2 = abcabcaaaaae   | True
word_3 = aae   | True
word_4 = adae   | True
word_5 = abcabcaaaae   | True
```

Conclusions

Regular grammars and finite automata are two closely related formalisms used to define and recognize regular languages. While regular grammars are used to generate strings that belong to a regular language, finite automata are used to recognize whether a given string belongs to a regular language or not.

Both formalisms are important in theoretical computer science and have many practical applications, such as in compiler design, natural language processing, and regular expression matching.

A formal language can be considered as a set of rules and symbols used to communicate information in a structured way. It is a means of expressing thoughts, ideas, and information between individuals or systems. The usual components of a language are:

An alphabet is a finite set of symbols that are used to construct the words of a language. The symbols can be any discrete entities, such as letters, digits, or other abstract entities.

The vocabulary of a language is the set of words or strings that can be constructed from the symbols of the alphabet. A vocabulary may be finite or infinite, depending on the complexity of the language.

The syntax of a language is the set of rules that govern the formation of sentences from words in the vocabulary. It defines the proper sequence and combination of words to form a grammatically correct sentence.

The semantics of a language deals with the meaning of the words and sentences in the language. It is concerned with how the words and sentences are used to convey information and ideas.

The pragmatics of a language deals with the way in which the language is used in a specific context or situation. It considers factors such as the speaker, the listener, and the context in which the language is being used.

Together, these components form the foundation of a formal language, allowing individuals or systems to communicate with each other in a structured and meaningful way.

In conclusion, regular grammars and finite automata are formalisms used in theoretical computer science to define and recognize regular languages. Regular grammars generate strings that belong to a regular language, while finite automata recognize whether a given string belongs to a regular language or not. Although they have some differences in terms of their structure, generative power, and representation, they are both fundamental tools for studying regular languages and serve as a foundation for more complex formalisms in the Chomsky hierarchy. Regular grammars and finite automata have many practical applications in various fields, such as in compiler design, natural language processing, and regular expression matching. Overall, these formalisms are essential for theoretical computer science and play a crucial role in understanding the limits of computation.