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

Report

Laboratory work 2:
Intro to formal languages. Regular grammar. Finite Automata.

Elaborated:
st. gr. FAF-223 Irina Nedealcova

Verified:
asist. univ. Cretu Dumitru

Chişinău – 2024

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.

Please consider that all elements of the task 3 can be done manually, writing a detailed report about how you've done the conversion and what changes have you introduced. In case if you'll be able to write a complete program that will take some finite automata and then convert it to the regular grammar - this will be a good bonus point.

Point 1

In this section of a laboratory work, we're examining a Python class structure that represents a Chomsky Grammar and its classification system. The Chomsky Hierarchy classifies formal grammars into four types based on their production rules. The primary components of this implementation include:

1. ProductionRule Class: This class models a production rule of a formal grammar. It has two attributes: `left_side` for the left-hand side (LHS) of the production rule, and `right_side` for the right-hand side (RHS). These sides are represented as strings.

2. ChomskyGrammar Class: This class encapsulates a grammar in the Chomsky Hierarchy, including its non-terminal symbols (`VN`), terminal symbols (`VT`), and a list of `ProductionRule` objects (`P`). It includes methods to classify the grammar according to the Chomsky Hierarchy:

classify` Method: Determines the type of the grammar (Type 0, Type 1,

Type 2, or Type 3) by calling other methods that check for specific criteria defining each grammar type.

is_type_0` Method: Checks for the most general grammar type where any production rule is allowed, essentially classifying all grammars as Type 0 by default.

is_type_1` Method (Context-Sensitive): Verifies that all production rules adhere to the form $\alpha \rightarrow \beta$, where the length of α is less than or equal to the length of β.

is_type_2` Method (Context-Free): Ensures all production rules are in the form $A \rightarrow \gamma$, where A is a single non-terminal symbol, and γ is a string of terminals and/or non-terminals.

is_type_3` Method (Regular): Checks that all production rules are either $A \rightarrow aB$ or $A \rightarrow a$, where A and B are non-terminal symbols and a is a terminal symbol.

The purpose of this lab section is to demonstrate how to implement and utilize a class-based approach to model formal grammars and to apply the Chomsky Hierarchy for grammar classification. Through this, students can gain hands-on experience with concepts from theoretical computer science, specifically in the domain of formal languages and automata theory.

Point 2

```
class ProductionRule:
    def __init__(self, left_side, right_side):
        self.left_side = left_side
        self.right_side = right_side

class ChomskyGrammar:
    def __init__(self):
        self.VN = {'S', 'A', 'B', 'C'}
        self.VT = {'a', 'b'}
        self.P = [
            ProductionRule('S', 'aA'),
            ProductionRule('A', 'bS'),
            ProductionRule('S', 'aB'),
            ProductionRule('B', 'aC'),
            ProductionRule('C', 'a'),
            ProductionRule('C', 'bS'),
        ]

    def classify(self):
        if self.is_type_3():
            return "Type 3 (Regular)"
        elif self.is_type_2():
            return "Type 2 (Context-Free)"
        elif self.is_type_1():
            return "Type 1 (Context-Sensitive)"
        elif self.is_type_0():
            return "Type 0 (Unrestricted)"
        else:
            return "Invalid grammar"

    def is_type_0(self):
        # All grammars are Type 0
```

```
        return True

    def is_type_1(self):
        # Check if all production rules are of the form α -> β where |α| <=
|β|
        for rule in self.P:
            if len(rule.left_side) > len(rule.right_side):
                return False
        return True

    def is_type_2(self):
        # Check if all production rules are of the form A -> γ where A is a
non-terminal and γ is a string of terminals or non-terminals
        for rule in self.P:
            if len(rule.left_side) != 1 or rule.left_side not in self.VN:
                return False
            if not all(c in self.VN.union(self.VT) for c in rule.right_side):
                return False
        return True

    def is_type_3(self):
        # Check if all production rules are of the form A -> aB or A -> a
where A and B are non-terminals and a is a terminal
        for rule in self.P:
            if len(rule.right_side) > 2:
                return False
            if len(rule.right_side) == 1 and rule.right_side[0] not in
self.VT:
                return False
            if len(rule.right_side) == 2 and not (rule.right_side[0] in
self.VT and rule.right_side[1] in self.VN):
                return False
        return True

grammar = ChomskyGrammar()
print(grammar.classify())
```

Point 3

In this assignment I realized that I am given NFA, so I made a graph and a table for it. Then I implemented NFA in DFA and made a table of transitions. Then I realized a visual graph of DFA on the basis of this table

Variant 17

Q = {q0,q1,q2,q3},
∑ = {a,b,c},
F = {q3},
δ(q0,a) = q0,
δ(q0,a) = q1,
δ(q1,b) = q1,
δ(q2,b) = q3,
δ(q1,a) = q2,
δ(q2,a) = q0.

V17

$Q = \{q_0, q_1, q_2, q_3\}$

$\Sigma = \{a, b, c\}$

$F = \{q_3\}$

$\delta(q_0, a) = q_0$

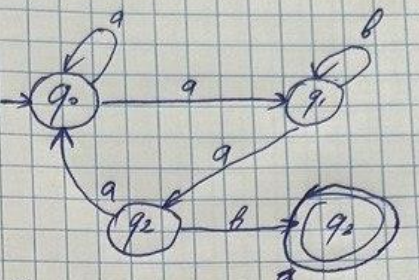
$\delta(q_0, a) = q_1$

$\delta(q_1, b) = q_1$

$\delta(q_2, b) = q_3$

$\delta(q_1, a) = q_2$

$\delta(q_2, a) = q_0$



Non-d

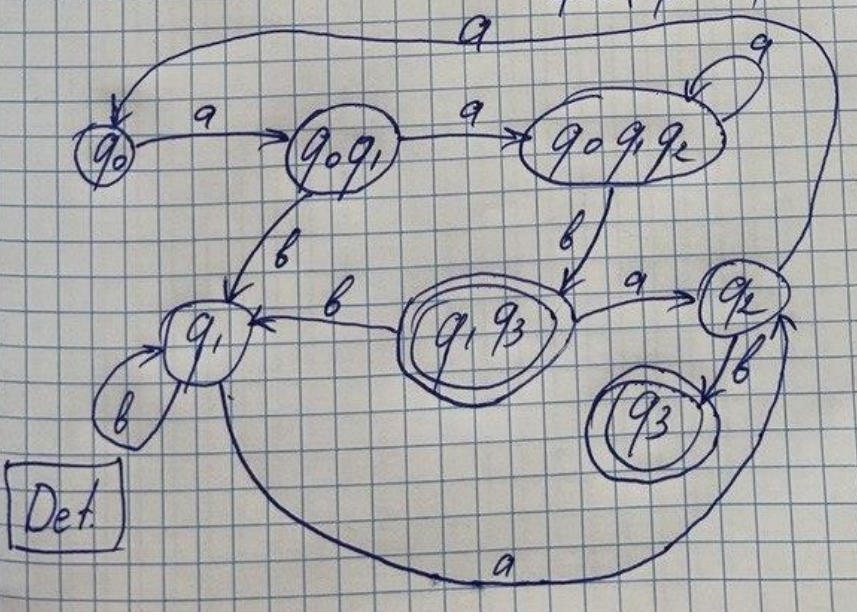
$P = \{ q_0 \to a q_0 \| a q_1 \|$,
$q_1 \to b q_1 \| a q_2$,
$q_2 \to b q_3 \| a q_0$,
$q_3 \to \varepsilon \}$

Non-deterministic

δ	a	b	c
→ q_0	$q_0 q_1$	$\emptyset$	$\emptyset$
q_1	q_2	q_1	$\emptyset$
q_2	q_0	q_3	$\emptyset$
* q_3	$\emptyset$	$\emptyset$	$\emptyset$

→ Deterministic

δ	a	b
→ $\{q_0\}$	$\{q_0 q_1\}$	$\{\emptyset\}$
$\{q_0 q_1\}$	$\{q_0 q_1 q_2\}$	$\{q_1\}$
$\{q_0 q_1 q_2\}$	$\{q_0 q_1 q_2\}$	$\{q_1 q_3\}$
$\{q_1\}$	$\{q_2\}$	$\{q_1\}$
* $\{q_1 q_3\}$	$\{q_2\}$	$\{q_1\}$
$\{q_2\}$	$\{q_0\}$	$\{q_3\}$
* $\{q_3\}$	$\emptyset$	$\emptyset$



Det.

Conclusion:

In this report, we explored foundational concepts in theoretical computer science, focusing on automatons, grammar classifications within the Chomsky Hierarchy, and the transformation of finite automatons (FAs) into regular grammars, along with determinism analysis. Our journey began with an understanding of automata and their applications, showcasing their utility in modeling computational processes and recognizing patterns within data streams.

The progression of our work within a consistent repository and project framework allowed for an in-depth exploration of grammatical structures through the implementation of a function capable of classifying grammars based on the Chomsky Hierarchy. This step was pivotal, as it not only reinforced our understanding of different grammar types but also provided practical experience in manipulating these structures programmatically.

Furthermore, aligned with our specific variant number, we delved into the realm of finite automata (FA), undertaking tasks that expanded our knowledge and practical skills significantly:

1. Conversion of a Finite Automaton to a Regular Grammar: This task bridged the conceptual gap between state machines and grammatical representations, highlighting the procedural and structural parallels between these two fundamental theoretical constructs. It illustrated the versatility of formal languages and their transformative capabilities.

2. Determining Automaton Determinism: Through analysis, we categorized our FA as either deterministic (DFA) or non-deterministic (NDFA), emphasizing the importance of understanding the nature of state transitions within automata. This distinction is crucial for grasping the operational and complexity differences between DFAs and NDFAs.

3. Converting an NDFA to a DFA: This complex task required us to apply algorithms that transform non-deterministic frameworks into deterministic ones, underscoring the universality of deterministic models and their preferential status in certain computational contexts due to their predictability and simplicity.

In conclusion, this report reflects a comprehensive exploration of automata theory, from understanding basic concepts to applying sophisticated transformations between different formal structures. The practical implementations not only enriched our theoretical knowledge but also honed our programming skills, demonstrating the interplay between computer science theory and practice. This endeavor has laid a solid foundation for further exploration and application of theoretical computer science principles in various computational and real-world scenarios.