Formal Languages & Finite Automata

Laboratory work 2:

Intro to Formal Languages. Regular Grammars. Finite Automata

Student:	Grigoraș Dumitru
Verified:	prof., Cojuhari Irina
	univ. assist., Crețu Dumitru

Chișinău, 2024

Theory

The Chomsky classification of Grammar organizes languages into four levels based on their complexity, from simplest to most complex. The 4 types of Grammar are:

- Regular Grammar Type 3 - the most restricted form of the language, and the only one which is accepted by the FA. The lhs (left-hand side) of the production must have a single non-terminal and the rhs (right-hand side) consisting of a single terminal or a comb between a non-terminal and terminal.
- Context-Free Grammar Type 2 - the lhs of production can have only one variable and there is no restriction on the rhs.
- Context-Sensitive Grammar Type 1 - the count of symbols on the lhs must be less or equal to the count of symbols on the rhs.
- Unrestricted Grammar Type 0 - there are no restrictions, and typically when a Grammar is not of type 3, 2 or 1, it falls into the type 0 category.

N on-Deterministic FA - there is an indeterminancy in the transitions, as with one single terminal, a state can go in more than one states. Also, there might be ε transitions, and in this case automaton is called ε-NFA.

Deterministic FA - inverse of NFA.
NFA to DFA Conversion:

- Step 1: Initially Q' = ϕ
- Step 2: Add q0 of NFA to Q'. Then find the transitions from this start state.
- Step 3: In Q', find the possible set of states for each input symbol. If this set of states is not in Q', then add it to Q'.
- Step 4: In DFA, the final state will be all the states which contain F(final states of NFA)
  ε-NFA to DFA conversion - requires one more step before converting NFA to DFA, which is determining ε-CLOSUREs.

Objectives:

Understand what an automaton is and what it can be used for.
Provide a function in your grammar type/class that could classify the grammar based on Chomsky hierarchy.
Implement conversion of a finite automaton to a regular grammar.
Determine whether your FA is deterministic or non-deterministic.
Implement some functionality that would convert an NDFA to a DFA.
Represent the finite automaton graphically.
Maintain mental health while creating FA graphs in Java (optional)

Implementation

Grammar:

- **classifyGrammar**(): A method to classify the type of grammar (e.g., Type 3, Type 2, etc.).

The classifyGrammar() method is responsible for classifying the grammar based on its productions. It checks if the grammar fits the criteria for Regular, Context-Free, or Context-Sensitive grammars based on the productions' structures.

Here's how the method works:

- It initializes two boolean variables, isType3 and isType2, to true, assuming the grammar might be Type 3 (Regular Grammar) and Type 2 (Context-Free Grammar) initially.
- It iterates through each production in the grammar's list of productions.
- For each production, it splits the production into its left-hand side (LHS) and right-hand side (RHS) parts.
- For Type 2 (Context-Free Grammar) check:
    1. It verifies if the left-hand side (LHS) of the production consists of a single non-terminal symbol (denoted by [A-Z]). If it doesn't match this pattern, it sets isType2 to false.
- For Type 3 (Regular Grammar) check:
    1. It examines the right-hand side (RHS) of the production.
    2. It checks if the RHS consists of a single terminal symbol (denoted by [a-z]), or a terminal followed by a non-terminal, or a non-terminal followed by a terminal.
- If the RHS doesn't follow any of these specific patterns, it sets isType3 to false.

After iterating through all productions, it determines the type of grammar based on the boolean flags:

If isType3 is true, it returns "Type 3 (Regular Grammar)".

If isType2 is true, it returns "Type 2 (Context-Free Grammar)".

If neither condition is met, it returns "Type 1 (Context-Sensitive Grammar) or Type 0 (Recursively Enumerable)".

AutomatonTypeChecker class contains a method to check whether a given finite automaton, represented by its transition map, is deterministic or not.
IsDeterministic() Method:
This method takes a map representing the transitions of the finite automaton as input.
It iterates through each state's transitions in the transition map.
For each state, it initializes a set called seenSymbols to keep track of symbols encountered during transitions.
Within each state's transitions, it checks if there are duplicate symbols.
If a symbol is seen more than once, it indicates non-determinism, and the method returns false.
If all symbols in a state's transitions are unique (except for the symbol "d", which is substituted with "c"), the method returns true, indicating that the automaton is deterministic.
Here's a summary of the method's logic:
For each state, it checks if there are duplicate symbols in its transitions.
It ensures that the symbol "d" is substituted with "c" to account for the non-determinism implied by "d".
If it finds any duplicate symbols, it concludes that the automaton is non-deterministic and returns false.
Otherwise, it determines that the automaton is deterministic and returns true.
This method provides a simple check for determinism based on the transitions of a finite automaton. However, it's important to note that this check is based on a specific condition (unique symbols), and there might be other conditions or constraints for determinism depending on the context of the automaton.

RegularGrammarConverter class is responsible for converting a finite automaton (FA) into a regular grammar. It constructs productions based on the transitions of the finite automaton and adds epsilon productions for accepting states.

convertToRegularGrammar Method:

- This method takes a FiniteAutomaton object as input and returns a Grammar object representing the equivalent regular grammar.
- It initializes an empty list of strings to hold the productions.
- It iterates through each state of the finite automaton.
- For each state, it retrieves the transitions associated with that state.
- It constructs productions for each transition, where the left-hand side (LHS) is the current state, and the right-hand side (RHS) consists of the symbol and the next state. If the symbol is "d", it substitutes it with "c" as part of regular grammar conversion.
- It also adds epsilon productions for accepting states.
- Finally, it constructs a Grammar object and sets its productions based on the generated list of productions.

FiniteAutomatonRenderer class is responsible for rendering a finite automaton using PlantUML, a tool for creating UML diagrams from plain text descriptions.
Let's break down the key aspects of the code:

- **Main Method:**

The main method is the entry point of the program.
It calls the generatePlantUMLCode method to generate the PlantUML code for the finite automaton.
It then calls the generateFromStringSource method to generate the UML diagram from the PlantUML code and save it as an image file.

- **generatePlantUMLCode** Method:

This method generates the PlantUML code that describes the finite automaton.
It constructs a string representation of the finite automaton using PlantUML syntax.
The string includes the title, states, transitions, and the initial and final states of the finite automaton.

- **generateFromStringSource** Method:

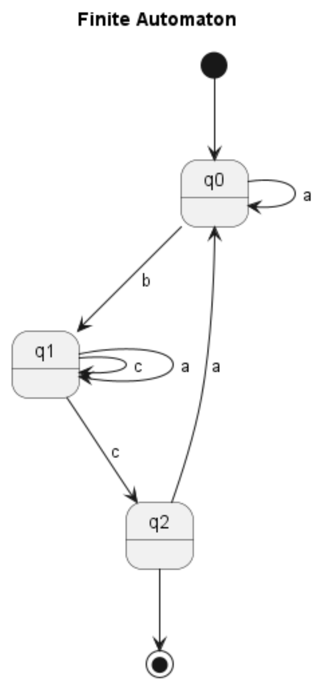
This method takes a File object as input where the generated UML diagram will be saved.
It uses the SourceStringReader class from PlantUML to read the PlantUML code generated by generatePlantUMLCode. It generates the image of the UML diagram from the PlantUML code and saves it to the specified file.

Results

Variant 14
Q = {q0,q1,q2},
∑ = {a,b,c},
F = {q2},
δ(q0,a) = q0,
δ(q0,b) = q1,
δ(q1,c) = q1,
δ(q1,c) = q2,
δ(q2,a) = q0,
δ(q1,a) = q1.

1. **Grammar type:**
   Type 3 (Regular Grammar)
2. **Convert Finite automaton to regular grammar:**
   Regular Grammar Productions:
   q1 → cq2
   q1 → cq1
   q1 → aq1
   q2 → aq0
   q0 → bq1
   q0 → aq0
   q2 → ε

3. **Check if is deterministic:**
   Is determenistic: false
4. **Generate graph image of automaton:**



Conclusion

To conclude this project, I have successfully applied the theoretical knowledge gained from lectures into a practical coding framework. By constructing a language from the given Grammar and translating it into a Finite Automaton (FA), I have navigated the intricacies of class design and deepened my understanding of Formal Languages and Automata Theory.

This task not only involved developing classes and methods integral to grammars and automata but also emphasized the significance of regular grammars (type 3) during the conversion process to FA. This practical application has solidified my comprehension of these concepts and their relevance in computer science.

The project was a reflective and insightful synthesis of theory and practice, enhancing my problemsolving abilities and reinforcing the concept that complex theoretical constructs can be effectively transformed into working models. It was a testament to the power of applying theoretical foundations to solve practical problems and has contributed to both my academic and personal development.