Grammar to Finite Automaton Conversion and String Generation

Course: Formal Languages & Finite Automata

Author: Durbailo Daniel

Group: FAF-222

Variant: 9

Chisinau 2024

My Variant (9):

```
119     Variant 9:
120     VN={S, B, D, Q},
121     VT={a, b, c, d},
122     P={
123         S → aB
124         S → bB
125         B → cD
126         D → dQ
127         Q → bB
128         D → a
129         Q → dQ
130     }
```

Theory:

A language, in a broad sense, is a system of communication that uses symbols (such as words, gestures, or symbols) and rules to convey meaning. Languages can be spoken, written, or signed, and they are essential for human communication and the expression of thoughts, ideas, and emotions.

Formal languages, on the other hand, have specific characteristics that distinguish them from natural languages (like English or Spanish). In the context of computer science, mathematics, and logic, a formal language is a well-defined set of symbols and rules for manipulating those symbols. Here are some key features that a language needs to have in order to be considered formal:

1. Alphabet:
    - A formal language has an alphabet, which is a finite set of symbols or characters that are used to construct expressions or statements in the language.
2. Syntax:
    - A formal language has a syntax that defines the rules for constructing valid expressions or statements. Syntax specifies the arrangement and combination of symbols to create meaningful structures.
3. Grammar:
    - The grammar of a formal language defines the set of rules governing the structure of expressions or statements. It describes how symbols can be combined to create valid constructs within the language.
4. Semantics:
    - Semantics deals with the meaning of expressions or statements in the formal language. It defines the relationship between the symbols and the real-world entities they represent or the operations they perform.
5. Precision:
    - Formal languages are characterized by precision and lack ambiguity. Every symbol and rule must have a clear and unambiguous interpretation.
6. Completeness:
    - The formal language should be complete, meaning that it should be able to express all possible statements or truths within its scope.

7. Rigor:
    - Rigor is a high level of accuracy and strict adherence to rules. Formal languages often require rigor in both syntax and semantics to ensure unambiguous interpretation.
8. Formalism:
    - Formal languages are often associated with formal systems, which are mathematical or logical structures that provide a foundation for reasoning about the language.

Examples of formal languages include programming languages (like Java, Python, or C++), mathematical notations (like set theory or propositional logic), and markup languages (like HTML or XML). These languages play a crucial role in various fields, providing a precise and structured way to express information and carry out computations.

Objectives:

1.Discover 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. You can deal with each laboratory work as a separate task or project to demonstrate your understanding of the given themes, but you also can deal with labs as stages of making your own big solution, your own project. Do the following:

a. Create GitHub repository to deal with storing and updating your project;

b. Choose a programming language. Pick one that will be easiest for dealing with your tasks, you need to learn how to solve the problem itself, not everything around the problem (like setting up the project, launching it correctly and etc.);

c. Store reports separately in a way to make verification of your work simpler (duh)

According to your variant number, get the grammar definition and do the following:

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;

How to Run the Program?:

In order to run the program you need to press **run* in the *_Main* file.

Implementation:

1.Main Class:

```java
package LAB1;
import java.util.Scanner;
 selavii
public class Main {
    selavii
    public static void main(String[] args) {
        Grammar grammar = new Grammar();
        FiniteAutomaton finiteAutomaton = grammar.convertToFiniteAutomaton();
        System.out.println("Generated Strings:");
        for (String str : grammar.generateStrings( count: 5)) {
            System.out.println(str);
        }
        System.out.println("Finite Automaton:");
        System.out.println(finiteAutomaton);


        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter input string: ");
        String inputString = scanner.nextLine();


        boolean canReach = finiteAutomaton.canReachString(inputString);
        if (canReach) {
            System.out.println("Input string can be obtained via state transitions.");
        } else {
            System.out.println("Input string cannot be obtained via state transitions.");
        }

        scanner.close();
    }
}
```

This code demonstrates how to use the Grammar and FiniteAutomaton classes to generate strings and check if a given input string can be obtained via state transitions. Make sure to adjust any package names or class names as necessary to match your project structure.

2.Grammar Class:

Defines a context-free grammar with non-terminal symbols (VN), terminal symbols (VT), and production rules (P).

```java
package LAB1;
import java.util.*;

public class Grammar {
    private Set<Character> VN;
    private Set<Character> VT;
    private Map<Character, List<String>> P;

    public Grammar() {
        VN = new HashSet<>(Arrays.asList('S', 'B',
'D', 'Q'));
        VT = new HashSet<>(Arrays.asList('a', 'b',
'c', 'd'));
        P = new HashMap<>();
        P.put('S', Arrays.asList("aB", "bB"));
        P.put('B',
Collections.singletonList("cD"));
        P.put('D', Arrays.asList("dQ", "a"));
        P.put('Q', Arrays.asList("bB", "dQ"));
    }

    public Set<Character> getVN() {
        return VN;
    }

    public Set<Character> getVT() {
        return VT;
    }

    public Map<Character, List<String>> getP() {
        return P;
    }

    public List<String> generateStrings(int count)
{
        Set<String> uniqueStrings = new
HashSet<>();
        List<String> validStrings = new
```

```
ArrayList<>();

        while (uniqueStrings.size() < count) {
            StringBuilder sb = new StringBuilder();
            generateStringHelper('S', sb);
            String generatedString = sb.toString();
            if (uniqueStrings.add(generatedString))
{
                validStrings.add(generatedString);
            }
        }

        return validStrings;
    }

    private void generateStringHelper(char symbol,
StringBuilder sb) {
        List<String> productions = P.get(symbol);
        if (productions != null) {

            String production = productions.get(new
Random().nextInt(productions.size()));

            for (char c : production.toCharArray())
{
                if (VN.contains(c)) {

                    generateStringHelper(c, sb);
                } else {

                    sb.append(c);
                }
            }
        }
    }

    public FiniteAutomaton
convertToFiniteAutomaton() {
```

```
        FiniteAutomaton finiteAutomaton = new
FiniteAutomaton();

        finiteAutomaton.setStates(VN);
        finiteAutomaton.setAlphabet(VT);


        for (char symbol : P.keySet()) {
            for (String production : P.get(symbol))
{
                char inputSymbol =
production.charAt(0);
                char nextState =
(production.length() == 1) ? 'f' :
production.charAt(1);

finiteAutomaton.addTransition(symbol, inputSymbol,
nextState);

finiteAutomaton.getAlphabet().add(inputSymbol); //
Add all symbols to alphabet
            }
        }

        finiteAutomaton.setInitialState('S');

finiteAutomaton.setFinalStates(getFinalStates());

        return finiteAutomaton;
    }

    private Set<Character> getFinalStates() {
        Set<Character> finalStates = new
HashSet<>();
        for (char symbol : VN) {
            for (String production : P.get(symbol))
{
                if (production.length() == 1) {
```

```
finalStates.add(production.charAt(0));
                }
            }
        }
        return finalStates;
    }
}
```

This Java code defines a Grammar class representing a context-free grammar (CFG). It includes methods to generate valid strings based on the grammar and to convert the grammar into a finite automaton. The grammar is initialized with predefined non-terminal symbols, terminal symbols, and production rules. The generateStrings method produces a specified number of unique strings following the grammar rules, while convertToFiniteAutomaton converts the grammar into a finite automaton representation.

3.FiniteAutomaton Class:

```
package LAB1;

import java.util.*;

public class FiniteAutomaton {
    private Set<Character> states;
    private Set<Character> alphabet;
    private Map<Character, Map<Character,
Character>> transitions;
    private Character initialState;
    private Set<Character> finalStates;

    public FiniteAutomaton() {
        states = new HashSet<>();
```

```
        alphabet = new HashSet<>();
        transitions = new HashMap<>();
        finalStates = new HashSet<>();
    }

    public void setStates(Set<Character> states) {
        this.states = states;
    }

    public void setAlphabet(Set<Character>
alphabet) {
        this.alphabet = alphabet;
    }

    public void addTransition(Character fromState,
Character symbol, Character toState) {
        if (!transitions.containsKey(fromState)) {
            transitions.put(fromState, new
HashMap<>());
        }
        transitions.get(fromState).put(symbol,
toState);
    }

    public void setInitialState(Character
initialState) {
        this.initialState = initialState;
    }

    public void setFinalStates(Set<Character>
finalStates) {
        this.finalStates = finalStates;
    }

    public boolean canReachString(String
inputString) {

        Character currentState = initialState;
```

```
        for (char symbol :
inputString.toCharArray()) {
            System.out.println("Current State: " +
currentState + ", Symbol: " + symbol);


            if
(!transitions.containsKey(currentState)) {
                System.out.println("No transitions
defined for state " + currentState);
                return false;
            }


            if
(!transitions.get(currentState).containsKey(symbol)
) {
                System.out.println("No transition
defined for symbol " + symbol + " in state " +
currentState);
                return false;
            }


            currentState =
transitions.get(currentState).get(symbol);
            System.out.println("New State: " +
currentState);


            if (currentState == 'f') {
                System.out.println("Reached final
state 'f'");
                return true;
            }
        }
```

```
        return finalStates.contains(currentState);
    }




    public Set<Character> getStates() {
        return states;
    }

    public Set<Character> getAlphabet() {
        return alphabet;
    }

    @Override
    public String toString() {
        StringBuilder sb = new StringBuilder();
        sb.append("States:
").append(states).append("\n");
        sb.append("Alphabet:
").append(alphabet).append("\n");
        sb.append("Transitions:
").append(transitions).append("\n");
        sb.append("Initial State:
").append(initialState).append("\n");
        sb.append("Final States:
").append(finalStates).append("\n");
        return sb.toString();
    }
}
```

This Java code represents a class called FiniteAutomaton, which models a finite automaton. It includes methods to set states, alphabet, transitions, initial state, and final states of the automaton. Additionally, it provides a method canReachString to determine if a given input string can be reached by transitioning through the automaton's states.

Conclusions / Screenshots / Result:

Example 1:

```
Generated Strings:
aca
acdbcddbca
bcdbcdddbca
bca
bcdbcdbcdbcddddbca
Finite Automaton:
States: [Q, B, S, D]
Alphabet: [a, b, c, d]
Transitions: {Q={b=B, d=Q}, B={c=D}, S={a=B, b=B}, D={a=f, d=Q}}
Initial State: S
Final States: [a]
```

In this example you can see 5 generated strings based on the given Grammar.

Example 2:

```
Generated Strings:
aca
acddbca
acdbca
bcdbca
acdddbcdbcddbcdbca
Finite Automaton:
States: [Q, B, S, D]
Alphabet: [a, b, c, d]
Transitions: {Q={b=B, d=Q}, B={c=D}, S={a=B, b=B}, D={a=f, d=Q}}
Initial State: S
Final States: [a]
```

Another example where you can see the generated strings based on the grammar.

```
Enter input string: aca
Current State: S, Symbol: a
New State: B
Current State: B, Symbol: c
New State: D
Current State: D, Symbol: a
New State: f
Reached final state 'f'
Input string can be obtained via state transitions.
```

Above you can see the algorithm that a word goes through in order to be checked if it validates the Grammar Rules. In this example it validates the rules. Below I will attach an example of a string that does not validate the rules:

```
Enter input string: agdsgags
Current State: S, Symbol: a
New State: B
Current State: B, Symbol: g
No transition defined for symbol g in state B
Input string cannot be obtained via state transitions.
```