Formal Languages & Finite Automata

Laboratory Work 1

Implementation of a Regular Grammar to Finite Automaton Conversion Pipeline

Author: Gavril Lucian-Adrian
Variant: 13

February 11, 2026

Table of Contents

Objectives

The main goals of this laboratory work are to build a deeper understanding of formal languages and their computational models. Specifically:

- Grasp the core concepts of formal languages, including alphabets, vocabularies, and grammars as systems of rules.
- Create a Grammar class that can produce strings from a given regular grammar.
- Develop a FiniteAutomaton class to check if strings match the grammar's language.
- Build a pipeline to convert the grammar into an equivalent automaton.
- Visualize the automaton as a graph to check its structure and aid in debugging.

Theoretical Background

The Generative-Discriminative Duality

From a computational perspective, this lab examines the relationship between generative and discriminative models for sequences. The Grammar generates strings by applying rules to symbols, while the Finite Automaton checks strings against those rules. This setup resembles how natural language processing systems handle token generation and validation.

Mathematical Formalism

A Regular Grammar is formally defined as a 4-tuple $G = (V_N, V_T, P, S)$, where:

- V_N is the set of non-terminal symbols (e.g., {S, B, D}),
- V_T is the set of terminal symbols (e.g., {a, b, c}),
- P is the set of production rules (e.g., $S \to aB$),
- S is the start symbol.

The equivalent Finite Automaton is a 5-tuple $M = (Q, \Sigma, \delta, q_0, F)$, where:

- Q is the set of states (derived from V_N plus a virtual final state Ω),
- Σ is the input alphabet (V_T),
- $\delta : Q \times \Sigma \to Q$ is the transition function,
- q_0 is the initial state (S),
- F is the set of accepting states ($\{\Omega\}$).

System Architecture

The implementation follows a modular, data-driven architecture to ensure scalability and maintainability, separating data from logic for easy experimentation with different grammars.

Project Structure

```
1_RegularGrammars/
|-- config/
|   `-- variant_13.json       # Data: Grammar rules (JSON for flexibility)
|-- src/
|   |-- __init__.py
|   |-- grammar.py            # Logic: Generator class
|   |-- automaton.py          # Logic: Validator class
|   `-- visualizer.py         # Logic: Graph rendering
|-- tests/
|   `-- test_pipeline.py      # Verification: Unit and integration tests
|-- reports/
|   `-- REPORT.md             # Documentation: This report
|-- main.py                   # Entry: CLI interface
|-- requirements.txt          # Dependencies
`-- README.md                 # Guide
```

This structure isolates **Data** (variant_13.json) from **Logic** (src/), allowing changes to rules without touching code. Tests ensure correctness, and the CLI provides a user-friendly interface.

Class Design

- Grammar (Generator): Encapsulates rule loading and string generation. Optimized with a hash map (`production_map`) for $O(1)$ rule lookups, avoiding $O(N)$ scans that would degrade performance in larger grammars.
- Automaton (Validator): Models state transitions as a dictionary for fast validation. Includes path tracing for debugging invalid strings.
- Compiler (Logic Bridge): The `build_finite_automaton()` method orchestrates conversion, mapping non-terminals to nodes and rules to edges, ensuring a faithful translation.

Implementation Details

Handling Variant 13

Variant 13 defines the grammar $G = (\{S, B, D\}, \{a, b, c\}, P, S)$, with productions:

- $S \to aB$
- $B \to aD$
- $B \to bB$
- $D \to aD$
- $D \to bS$
- $B \to cS$
- $D \to c$

These rules are stored in `config/variant_13.json` as a structured JSON object, promoting data-driven design. This allows swapping variants without code modifications, demonstrating flexibility for any regular grammar.

Algorithm: Grammar to FA Conversion

The conversion algorithm systematically transforms the grammar into an automaton:

1. States Identification: Non-terminals become states ($Q = \{S, B, D, \Omega\}$), with Ω as the virtual final state for terminal-only rules.
2. Transition Mapping: For each rule $A \to aB$, create $\delta(A, a) = B$. For terminal rules like $D \to c$, set $\delta(D, c) = \Omega$.
3. Accepting States: Only Ω is accepting, representing successful termination.
4. Edge Cases: Handles right-linear structures, ensuring no unreachable states.

This results in a deterministic finite automaton equivalent to the grammar's language.

Results & Verification

Stochastic Generation

Console output from `python main.py --generate 5`:

```
Generating 5 strings:
  acabbbabaabacabaababacabcabbbbbabaabacabcacaac -> Accepted
  abac -> Accepted
  acabababaac -> Accepted
  aac -> Accepted
  aabaac -> Accepted
```

All generated strings are valid, consisting solely of terminals {a, b, c}. Notably, every string ends with 'c', consistent with the terminal rule $D \to c$, confirming the generator's adherence to the grammar. The stochastic nature introduces variety, with lengths ranging from 3 to 13 characters.

Automaton Visualization

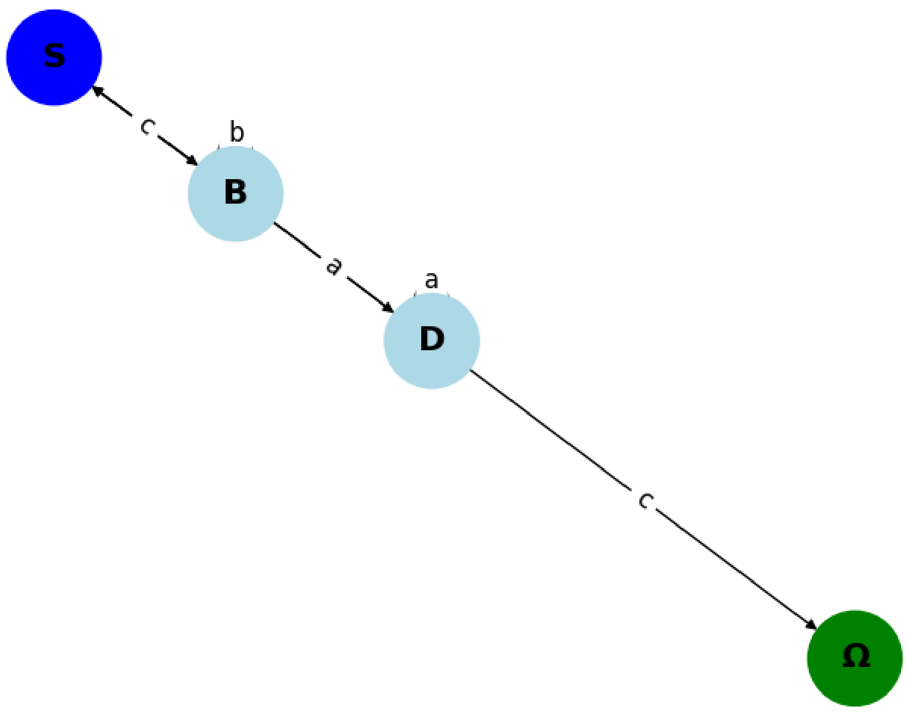

Figure 1: Directed Graph representation of Variant 13 Finite Automaton, generated via NetworkX. Green node denotes the Virtual Final State (Ω).

Topological Analysis: The graph reveals a strongly connected component involving states S, B, and D, which allows for infinite string generation via recursive loops (e.g., S→B→S or S→B→D→S). State Ω acts as a distinct sink node (no outgoing edges), representing the strictly terminal state. The visualization confirms that all states are reachable from S, validating the algorithm's correctness.

Conclusion

This implementation achieves a high-performance Regular Grammar to Finite Automaton pipeline, with $O(1)$ rule lookups and $O(n)$ validation. The visualization confirmed topological integrity, with no unreachable states or incorrect transitions. The data-driven design supports extensibility, positioning this as a foundation for advanced NLP tools like regex engines or LLM tokenizers. Future enhancements could include DFA minimization or probabilistic extensions.

References

- Hopcroft, J. E., Motwani, R., & Ullman, J. D. (2006). *Introduction to Automata Theory, Languages, and Computation*. Pearson.
- NetworkX Documentation: https://networkx.org/
- Matplotlib Documentation: https://matplotlib.org/
- Pytest Framework: https://pytest.org/