Interactive Story Compiler: A Visual Approach to Context-Free Grammar Interpretation

Laura Beltrán
Santiago Sánchez
Universidad Distrital Francisco José de Caldas

May 2025

Abstract

This technical report presents the initial version of a compiler designed to interpret structured natural language stories and generate interactive visual narratives. The project applies core concepts from the Theory of Computation, including finite-state machines, regular expressions, and context-free grammars. While the implementation is in progress, this document focuses on the design, theoretical foundation, and expected functionality of the compiler. It aims to bridge abstract computation theory with accessible user-facing applications.

Introduction

The Theory of Computation emphasizes abstract models such as regular expressions, finite-state machines (FSMs), and context-free grammars (CFGs). To better illustrate their practical value, this project proposes a compiler that transforms structured natural language into an interactive HTML story. This system will visually represent user decisions as state transitions, offering a pedagogical bridge between theoretical models and real-world applications.

Objectives

The primary objective is to design and develop a basic compiler that:

- Parses structured natural language stories using concepts from formal grammars.
- Converts these stories into a state-based interactive narrative.
- Generates a visual HTML representation with buttons for user decisions.

Secondary objectives include reinforcing theoretical concepts through implementation and delivering a self-contained educational tool.

Theoretical Background

This project leverages the concepts explored in Workshop 1:

Finite-State Machines (FSM): Each scene in the story represents a state, and each user decision acts as a transition between states.

Regular Expressions (RE): Used during lexical analysis to identify key story elements such as scene labels, choices, and narrative text.

Context-Free Grammars (CFG): The syntax of the structured natural language format can be modeled using a CFG. Scenes act as non-terminal symbols, and choices map to production rules.

Proposed Solution

The compiler accepts a story defined in a minimal syntax:

```
scene: START
text: You wake up in a dark cave.
choice: Go left -> DRAGON
choice: Go right -> EXIT

scene: DRAGON
text: A dragon appears!

scene: EXIT
text: You found the way out.
```

Each scene is parsed and stored as a state in a finite-state machine. Transitions are built from the choices listed. Internally, the compiler uses regular expressions to extract tokens and construct a graph.

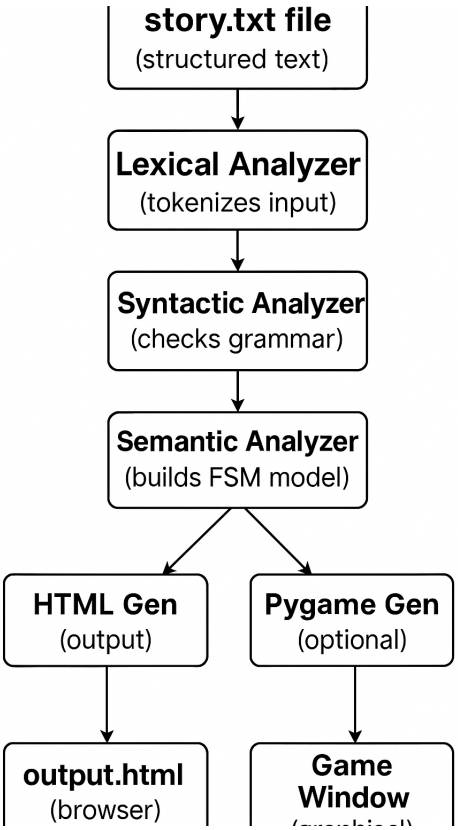
The output is a standalone HTML file with embedded JavaScript. Each scene is a hidden div that becomes visible upon selection. Choices are rendered as buttons that trigger scene transitions.

Development Plan and Tools

The compiler is being developed in Python due to its strong support for text parsing and rapid prototyping. Output generation uses HTML/CSS/JS for cross-platform compatibility. The initial version is intentionally constrained to a small number of scenes to focus on correctness and clarity.

Implementation

The compiler was implemented as a modular Python project. It consists of four main components: the lexical analyzer, syntactic parser, semantic analyzer, and code generator. Each module corresponds to a specific phase in the compilation pipeline.



- Lexical analysis (lexer.py) scans the input story and produces a sequence of tokens using regular expressions. It identifies keywords like scene:, text:, choice:, and structures like arrows and quoted strings
- Syntactic analysis (syntactic.py) verifies that the token sequence conforms to a context-free grammar. It ensures that each scene is correctly formatted, with optional choices, and raises errors when syntax violations are found.
- Semantic analysis (semantic.py) constructs a data model representing the story's finite-state machine. It ensures all referenced scenes exist and captures transitions between scenes as user choices.
- Code generation (compiler.py) supports two output modes: an HTML file or a GUI graphical interface. The HTML version generates a self-contained webpage where scenes are hidden or shown dynamically using JavaScript. The Pygame mode opens a window where the story is rendered with buttons and dynamic transitions.

The modular structure allows each component to be tested and refined independently, improving maintainability and extensibility.

Testing and Preliminary Results

To evaluate the system, a sample .txt story with four scenes and multiple branching choices was created. The compiler was able to tokenize, parse, and semantically validate the story, generating a fully functional HTML output.

Key outcomes:

- Scene transitions work correctly through clickable buttons.
- Invalid story formats (e.g., undefined scenes, missing quotes) trigger descriptive error messages
- The compiler performs well with minimal processing time for short stories.

In addition to the HTML output, an additional visual interface was tested, showing promising results. The graphical output presents the same branching logic in a stylized game-like window, enhancing the user experience and extending the tool's educational reach.

These results confirm that the initial design is feasible and aligns with the goals set at the beginning of the project.

Future Work

Future improvements include:

- Adding image and sound support to enhance the storytelling experience.
- Introducing a GUI-based code editor with live preview.
- Allowing backward navigation or undoing choices.
- Supporting longer, more complex narratives with nested structures.

Conclusions

The development of the Interactive Story Compiler successfully demonstrates how abstract computation concepts—such as finite-state machines, regular expressions, and context-free grammars—can be applied in a practical, engaging context. By translating structured natural language into interactive narratives, the project provides a clear example of how theoretical models can support creative and user-friendly applications.

The modular architecture, written entirely in Python, ensures clarity and scalability.

While the current implementation is intentionally limited in scope to maintain simplicity, the results validate the original design goals and provide a solid foundation for future expansion. Overall, the project serves as an effective bridge between theory and practice, making formal language concepts more accessible to students, educators, and creators alike.

References

- Twine. An open-source tool for telling interactive, nonlinear stories. `https://twinery.org/`
- Ink. Inkle's narrative scripting language. `https://www.inklestudios.com/ink/`
- Sierra, C. A. (2025) *Slides from Computer Science III: Programming Languages Foundations*. Universidad Distrital Francisco José de Caldas.