Structured Output and Formal Languages: Regex, Grammars, and Backus-Naur Form

AI Research Team

May 14, 2025

Abstract

This article explores the relationship between structured output generation in Large Language Models (LLMs) and concepts from formal language theory, including regular expressions, context-free grammars, and the Backus-Naur Form (BNF). We examine how these classical concepts from computational linguistics have found new applications in controlling and constraining the output of modern language models, enabling more reliable and structured text generation for downstream applications.

1 Introduction

Structured output generation with Large Language Models (LLMs) represents a convergence of modern deep learning techniques with classical concepts from formal language theory. While LLMs are trained to generate natural language in a probabilistic manner, many applications require outputs that conform to specific syntactic structures or formats. This need has led to the adaptation of formal language concepts—such as regular expressions, context-free grammars, and the Backus-Naur Form—to constrain and guide LLM outputs.

These formal language concepts, which have been foundational in computer science and linguistics for decades, provide powerful tools for defining and constraining the structure of generated text. By leveraging these tools, developers can ensure that LLM outputs adhere to specific formats, making them more reliable for downstream processing and integration with other systems.

2 Regular Expressions and LLMs

2.1 Overview of Regular Expressions

Regular expressions (regex) are a sequence of characters that define a search pattern. They are commonly used for string matching and manipulation in various programming languages. Regular expressions are based on the theory of regular languages, which are the simplest class of formal languages in the Chomsky hierarchy.

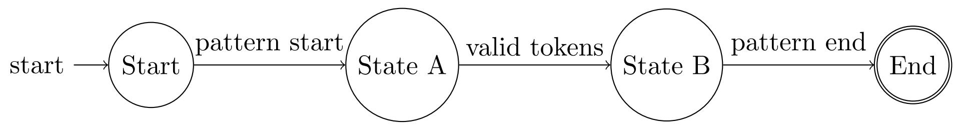

Figure 1: Simplified finite state machine for regex-based generation

A regular expression consists of:

- Literal characters that match themselves
- Meta-characters with special meanings (e.g., *, +, ?, —)
- Character classes and quantifiers
- Grouping constructs

For example, the regex pattern ‘
s*([Yy]es—[Nn]o—[Nn]ever—[Aa]lways)‘ matches strings that start with optional whitespace followed by one of the words ”yes”, ”no”, ”never”, or ”always” (case-insensitive for the first letter).

2.2 Regex-Based Constrained Generation

In the context of LLMs, regular expressions can be used to constrain the model's output to follow specific patterns. This is particularly useful for ensuring that the generated text adheres to a predefined format or structure.

Libraries like Outlines implement regex-based constrained generation by:

1. Converting the regex pattern into a finite state machine (FSM)
2. Using the FSM to guide the token generation process
3. Masking out tokens that would lead to invalid states in the FSM
4. Renormalizing the probability distribution over the remaining valid tokens

This approach effectively transforms the unconstrained next-token prediction problem into a constrained generation problem where only tokens that maintain the validity of the regex pattern are considered.

3 Context-Free Grammars and LLMs

3.1 Overview of Context-Free Grammars

Context-free grammars (CFGs) are a formal grammar used to describe the syntax of programming languages, natural languages, and other structured formats. They are more expressive than regular expressions and can handle nested structures that regex cannot.

A context-free grammar consists of:

- A set of terminal symbols (the actual characters in the language)
- A set of non-terminal symbols (abstractions that generate terminal symbols)
- A set of production rules that map non-terminals to sequences of terminals and non-terminals
- A start symbol (a designated non-terminal)

For example, a simple CFG for JSON-like objects might include rules like:

```
<object> ::= "{" <members> "}"
<members> ::= <pair> | <pair> "," <members>
<pair> ::= <string> ":" <value>
<value> ::= <string> | <number> | <object> | <array> | "true" | "false" | "null"
```

3.2 Grammar-Based Constrained Generation

Grammar-based constrained generation extends the concept of regex-based generation to handle more complex structures. Instead of using a finite state machine, grammar-based generation typically uses parsing algorithms like CYK or Earley parsing to guide the generation process.

The key steps in grammar-based constrained generation are:

1. Define a context-free grammar for the desired output format
2. Track the parsing state during generation
3. At each step, determine which tokens are valid continuations according to the grammar
4. Mask out invalid tokens and renormalize the probability distribution

This approach is particularly useful for generating structured outputs like JSON, XML, or programming code, where nested structures are common.

4 Backus-Naur Form and LLMs

4.1 Overview of Backus-Naur Form

Backus-Naur Form (BNF) is a notation technique for context-free grammars, often used to describe the syntax of programming languages, document formats, and communication protocols. It was developed by John Backus and Peter Naur in the late 1950s.

BNF uses a set of derivation rules written as:

```
<symbol> ::= expression
```

where '¡symbol¿' is a non-terminal and 'expression' consists of sequences of symbols and/or sequences separated by the vertical bar '—', indicating a choice.

Extended BNF (EBNF) adds additional notation for optional elements, repetition, and grouping, making it even more expressive.

4.2 BNF-Based Constrained Generation

BNF provides a formal way to specify the syntax of structured outputs, making it a natural choice for constraining LLM generation. By defining a BNF grammar for the desired output format, developers can ensure that the generated text adheres to the specified syntax.

The process of BNF-based constrained generation is similar to grammar-based generation:

1. Define a BNF grammar for the desired output format
2. Parse the grammar to create a representation that can guide the generation process
3. During generation, track the current position in the grammar
4. Constrain token selection to only those that are valid according to the grammar

This approach is particularly useful for generating outputs in formats with well-defined syntaxes, such as programming languages or data exchange formats.

5 Implementation in Modern Tools

5.1 Outlines: Finite State Machines for Structured Generation

Outlines is a Python library that implements constrained text generation using finite state machines. It converts constraints (like regex patterns or grammars) into FSMs and uses them to guide the generation process.

The key innovation in Outlines is its efficient indexing approach that reformulates neural text generation using finite-state machines. This approach involves two main stages:

1. **Preprocessing Step:** Converting a character-level deterministic finite automaton (DFA) into a token-level DFA that tests whether a token sequence decodes into a string matching the constraint.
2. **Decoding Step:** Using the DFA to determine which tokens are allowed at each step, applying the corresponding mask to the next token probabilities, and updating the state of the DFA after each token is generated.

```python
import outlines
import re

model = outlines.models.transformers("gpt2")

# Define a regex pattern for a simple email format
email_pattern = r"[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}"

# Create a generator with the regex constraint
generator = outlines.generate.regex(model, email_pattern)

# Generate an email address
prompt = "Generate an email address: "
email = generator(prompt)
print(email)
```

Listing 1: Using Outlines with regex constraints

5.2 JSON Schema and Pydantic Models

JSON Schema and Pydantic models provide another approach to structured output generation. These tools define the structure and validation rules for JSON data, which can be used to guide LLM output.

Libraries like LangChain and Outlines support JSON Schema and Pydantic models for structured output generation:

```python
from pydantic import BaseModel, Field
from typing import List
import outlines

# Define a Pydantic model for the desired output structure
class Person(BaseModel):
    name: str
    age: int
    email: str
    skills: List[str] = Field(default_factory=list)

# Create a generator with the Pydantic model constraint
model = outlines.models.transformers("gpt2")
generator = outlines.generate.json(model, Person)

# Generate a person object
prompt = "Generate information for a software developer: "
person = generator(prompt)
print(person)
```

Listing 2: Using Pydantic for structured output

6 Theoretical Connections and Practical Implications

6.1 Chomsky Hierarchy and LLM Constraints

The Chomsky hierarchy classifies formal grammars into four types based on their expressive power:

1. Type-0: Unrestricted grammars
2. Type-1: Context-sensitive grammars
3. Type-2: Context-free grammars
4. Type-3: Regular grammars

LLM constraint mechanisms typically operate at the level of Type-3 (regular expressions) or Type-2 (context-free grammars) in the Chomsky hierarchy. This is because these types of grammars are computationally tractable and can be efficiently implemented as constraints during the generation process.

The choice of constraint mechanism depends on the complexity of the desired output structure:

- Regular expressions are sufficient for simple patterns without nested structures
- Context-free grammars are necessary for formats with nested structures (like JSON or programming languages)
- More complex constraints might require post-processing or additional validation

6.2 Practical Implications for LLM Applications

The integration of formal language concepts with LLMs has several practical implications:

- **Improved Reliability:** By constraining outputs to follow specific formats, LLMs become more reliable for critical applications.
- **Reduced Post-Processing:** Structured outputs reduce the need for complex post-processing logic to extract information from free-form text.
- **Better Integration:** Structured outputs facilitate seamless integration with downstream systems that expect data in specific formats.
- **Enhanced User Experience:** Consistent, well-formatted outputs improve the user experience in applications like chatbots or code generation tools.

7 Future Directions

The integration of formal language concepts with LLMs is an evolving field with several promising directions for future research and development:

- **More Expressive Constraints:** Developing constraint mechanisms that can handle more complex grammars while remaining computationally efficient.
- **Semantic Constraints:** Extending beyond syntactic constraints to include semantic constraints that ensure the generated content is not only well-formed but also meaningful and accurate.
- **Hybrid Approaches:** Combining different constraint mechanisms (e.g., regex for simple patterns and grammars for complex structures) to achieve the best balance of flexibility and control.
- **User-Friendly Interfaces:** Creating more intuitive interfaces for defining constraints, making these techniques accessible to a wider range of users.

8 Conclusion

The integration of formal language concepts with LLMs represents a powerful approach to structured output generation. By leveraging regular expressions, context-free grammars, and the Backus-Naur Form, developers can ensure that LLM outputs adhere to specific formats, making them more reliable for downstream processing and integration with other systems.

As LLMs continue to evolve and find applications in more domains, the importance of structured output generation will only increase. Understanding the theoretical foundations and practical implementations of these techniques will be essential for developers looking to build robust and reliable LLM-powered applications.

9 References

1. Chomsky, N. (1956). Three models for the description of language. IRE Transactions on Information Theory, 2(3), 113-124.
2. Hopcroft, J. E., Motwani, R., Ullman, J. D. (2006). Introduction to Automata Theory, Languages, and Computation. Pearson Education.
3. Liang, P., et al. (2024). Holistic Evaluation of Language Models. arXiv preprint arXiv:2211.09110.
4. Outlines. (2024). Outlines: Structured Generation with LLMs. https://github.com/outlines-dev/outlines

5. Tran-Thien, L. (2024). Outlines: Guided Generation with LLMs. https://outlines-dev.github.io/outlines/

6. Willard, B., Louf, J. (2023). Guided Generation: Constrained Decoding for LLMs. arXiv preprint arXiv:2307.09702.