Efficient Unicode-Compatible Grammar-Constrained Decoding via String Homomorphism

Anonymous ACL submission

Abstract

Grammar-constrained decoding (GCD) is a powerful technique that enforces formal grammar constraints on the outputs of large language models (LLMs). This method ensures that generated text adheres to predefined structural rules, making it highly suitable for tasks requiring precise output formats. Despite its broad applications, the theoretical fundamentals of GCD remain underexplored, particularly in the context of formal language theory. In this work, we introduce the concept of tokenization as an inverse homomorphism, which maps the original string language to a token language defined on the alphabet of token IDs. The fact that tokenization is an inverse homomorphism is important for the efficiency of GCD, providing both a theoretical basis and an efficient construction method for the GCD algorithm. We further extend this framework to support Unicode characters, which are essential for multilingual NLP applications.

Our implementation is available at the following URL: `Anonymous`.

1 Introduction

Grammar-constrained decoding (GCD) is a technique that enforces formal grammar constraints on the outputs of large language models (LLMs). This method allows users to define the desired structure of the output using formal grammars as an interface and ensures that the generated text adheres to these constraints with hard guarantees. Due to its generality and robustness, grammar-constrained decoding has been widely applied in various tasks, including code synthesis (Poesia et al., 2022; Scholak et al., 2021), semantic parsing for domain-specific languages (Shin et al., 2021; Wang et al., 2023), and other structured outputs (Geng et al., 2024b; Zaratiana et al., 2024; Li et al., 2024). Various optimization techniques have been proposed in subsequent works to improve the efficiency (Beurer-Kellner et al., 2024) and extend to black-box LLMs without logit access (Geng et al., 2024a).

1. Detokenization is homomorphic from token IDs to ASCII[a]

$$d(15496) = \text{“Hello”} \;\wedge\; d(2159) = \text{“␣World”}.$$
$$d([15496, 2159]) = \text{“Hello␣World ”}$$
$$\equiv$$
$$[d(15496), d(2159)] = \text{“Hello␣World”}.$$

2. Detokenization is not homomorphic from token IDs to Unicode

$$d(19526) = \text{“ä¥”} \;\wedge\; d(254) = \text{“ł”}.$$
$$d([19526, 254]) = \text{“你”}$$
$$\not\equiv$$
$$[d(19526), d(254)] = \text{“ä¥ł”}$$

3. Detokenization is homomorphic from token IDs to Unicode bytes Tokenization

$$d(19526) = \text{“E4 BD”} \;\wedge\; d(254) = \text{“A0”}.$$
$$d([19526, 254]) = \text{“E4 BD A0 (你)”}$$
$$\equiv$$
$$[d(19526), d(254)] = \text{“E4 BD A0(你)”}$$

- Detokenization $= d : \mathbb{N}^* \to \Sigma^*$
- Tokenization $= d^{-1} : \Sigma^* \to \mathbb{N}^*$

[a] A leading space is omitted before the token *Hello*

Figure 1: Tokenization and Detokenization functions illustrating the broken homomorphism property in OpenAI GPT-2's Tokenization scheme.

From a high-level perspective, grammar-constrained decoding can be viewed as a checking mechanism that continuously validates the generated text against a set of formal grammar rules until the completion of the generation process. This continuous validation process is similar to the incremental parsing process in traditional parsing algorithms, where the parser checks the validity of the input string as it is being read. The difference is that as LLMs employ a token-based representation, the validation process is done on the sequence of token IDs rather than the sequence of characters. As shown in Fig. 2, tokenizing a few strings with a very simple structure, such as balanced parentheses, results in a non-trivial sequence of token IDs The misalignment between the token IDs and the original characters is further amplified when the strings contain Unicode characters, which are represented by multiple bytes (or multiple token IDs) as shown in Fig. 1. As the central question of grammar-constrained decoding is how to efficiently validate the token IDs sequence, it is crucial to understand the relationship between the sequence of token IDs and the original string of characters.

While numerous implementations of grammar-constrained decoding have been available, there is no investigation into the structural properties of the token ID sequence and how this impacts the efficiency of GCD. In this work, we aim to bridge this gap by viewing the grammar-constrained decoding problem from a formal language perspective. We consider the grammar-constrained decoding problem as a *decision problem* that involves determining whether a given (partial) string belongs to a formal language L defined by a specific grammar G. The sequence of token IDs generated by the tokenizer can be viewed as a new language L', which we refer to as the token language because it is defined on the alphabet of token IDs. We start by showing that the tokenization process can be viewed as an inverse homomorphism from the token ID alphabet to the character alphabet. This homomorphism property allows us to establish a connection between the token language L' and the original string language L, i.e., the token language is an inverse-homomorphic image of the original string language. We then show that this homomorphic property is crucial for the efficiency of grammar-constrained decoding, as it provides both a **theoretical foundation** and an **efficient construction** for the grammar-constrained decoding algorithm.

After establishing the homomorphic property of tokenization, we extend our discussion to the support for Unicode characters in grammar-constrained decoding. While most of the existing works on GCD focus on ASCII characters, the support for Unicode characters is crucial for multilingual NLP applications. Due to the fact that Unicode characters are represented by multiple bytes, the tokenization process on Unicode characters has an extra layer of complexity. We show that the support for Unicode characters can be naturally integrated into the homomorphic framework by transforming the grammar G with Unicode characters to a new grammar G' with byte-level alphabet. Once the transformation is done, we can reuse the same algorithm for grammar-constrained decoding on the byte-level alphabet, thereby removing the difficulty of handling Unicode characters in the token space

Contributions

- We establish a theoretical foundation for grammar-constrained decoding by viewing the problem as a decision problem in formal language theory.
- We show that tokenization can be seen as an inverse homomorphism between token IDs and characters (or bytes), which paves the way for efficient grammar-constrained decoding on token space.
- We show that the homomorphic property is broken with Unicode characters but can be restored by transforming the grammar to a byte-level alphabet. This allows to extend grammar-constrained decoding to Unicode- with minimal effort

2 Preliminaries

2.1 Context-free grammar and language

Definition 2.1 (Context-free Grammar)**.** *A* context-free grammar (CFG) *is a 4-tuple* $G = (V, \Sigma, P, S)$*, where*

- V *is a finite set of non-terminal symbols (variables),*
- Σ *is a finite set of terminal symbols,*
- P *is a finite set of production rules, each of the form* $A \to \alpha$*, where* $A \in N$ *and* $\alpha \in (N \cup \Sigma)^*$*,*
- $S \in N$ *is the start symbol.*

Definition 2.2 (Formal Language)**.** *A* formal language L *is a set of strings over an alphabet* Σ*, where a string is a finite sequence of symbols from* Σ.

Depth	String	Tokenization	Tokens
0	""	[1]	BOS = 1
1	"[]"	[1, 5159]	␣[= 518
2	"[[]]"	[1, 518, 2636, 29962]	[] = 2636
3	"[[[]]]"	[1, 5519, 2636, 5262]	␣[[= 5519
4	"[[[[]]]]"	[1, 5519, 29961, 2636, 5262, 29962]	[[= 29961
5	"[[[[[]]]]]"	[1, 5519, 8999, 2636, 5262, 5262]	[[[= 8999
6	"[[[[[[]]]]]]"	[1, 5519, 8999, 29961, 2636, 5262, 5262, 29962]	] = 29962
7	"[[[[[[[]]]]]]]"	[1, 5519, 8999, 8999, 2636, 5262, 5262, 5262]	]] = 5262
8	"[[[[[[[[]]]]]]]]"	[1, 5519, 8999, 8999, 29961, 2636, 5262, 5262, 5262, 29962]	␣[] = 5159

Figure 2: Tokenization Output for Nested Brackets Using LLaMA Tokenizer

If $G(V, \Sigma, P, S)$ is a CFG, the language of G, denoted $L(G)$, is the set of all strings of terminal symbols that can be derived from the start symbol S. If a language L is the language of some CFG, then L is called a *context-free language(CFL)*.

Definition 2.3 (Pushdown Automaton)**.** *A pushdown automaton (PDA) is a 7-tuple* $M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$*, where*

- Q *is a finite set of states,*
- Σ *is a finite set of input symbols,*
- Γ *is a finite set of stack symbols,*
- $\delta : Q \times (\Sigma \cup \{\epsilon\}) \times \Gamma \to 2^{Q \times \Gamma^*}$ *is the transition function,*
- $q_0 \in Q$ *is the start state,*
- $Z_0 \in \Gamma$ *is the initial stack symbol,*
- $F \subseteq Q$ *is the set of accepting states.*

Theorem 2.1 (Pushdown Automaton and Context-free Grammar)**.** *For every context-free grammar G, there exists a pushdown automaton M that accepts the language $L(G)$.*

Thm. 2.1 implies that one can always construct a PDA to decide whether a given string belongs to a context-free language.

Definition 2.4 (String Homomorphism)**.** *Given two operations $\oplus$ and $\odot$ on two alphabets Σ^* and Γ^* respectively, a function $h : \Sigma^* \to T^*$ is a string homomorphism if $\forall u, v \in \Sigma^*, h(u \oplus v) = h(u) \odot h(v)$.*

In the following, we assume that $\oplus$ and $\odot$ are both string concatenation operations and use xy to denote the concatenation of two elements x and y. Thus, a mapping h is a string homomorphism if it preserves the concatenation of strings. One can apply a homomorphism to a language L by applying it to each string in the language, which results in a new language $h(L)$. That is, $h(L) = \{h(w) \mid w \in L\}$ is the image of L under h.

Definition 2.5 (Inverse Homomorphism)**.** *Given a string homomorphism $h : \Sigma^* \to \Gamma^*$, the inverse function $h^{-1} : \Gamma^* \to \Sigma^*$ is called an inverse homomorphism.*

The inverse homomorphism $h^{-1}(L)$ includes all strings in Σ^* that map to strings in L under h.

Theorem 2.2 (Closure under Inverse Homomorphism)**.** *If L is a context-free(regular) language and $h : \Sigma^* \to \Gamma^*$ is a homomorphism, then the inverse homomorphic image $h^{-1}(L)$ is also a context-free(regular) language (Hopcroft et al., 2006, Theorem 7.30).*

2.2 Tokenization and Unicode

Tokenization[1] is the process of splitting a string into sub-word units known as tokens and converting these tokens into numerical representations (integers) which can be fed into the model. This step improves the efficiency of the model by reducing the vocabulary size and the length of the input sequences.

Detokenization is the reverse process of tokenization; it involves converting the token IDs back into their respective token strings, and subsequently concatenating these strings to form the original text.

Unicode Support. Byte-level tokenization[2] (Wang et al., 2019; Radford et al., 2019) is the standard way to provide support for Unicode characters. Unlike traditional character-level tokenization, byte-level tokenization first converts the text into a byte sequence according to a format such as UTF-8, and then applies tokenization to the byte sequence. The resulting token IDs are chunks of bytes instead

[1] In some litterature, *tokenization* only refers to the process of spliting the text and the term *encoding* is used to describe the mapping from token to ID. In this work, we follow our definition.

[2] also known as byte-level encoding

of characters, which allows the model to support effectively any Unicode character.

3 Tokenization is inverse homomorphism

In this section, we start with showing that tokenization can be seen as an inverse homomorphism between token IDs and characters (or bytes), which implies that the structure of the source language is preserved. We then show that this homomorphism leads to an efficient construction of a PDA for grammar-constrained decoding on token space.

3.1 Tokenization preserves structure

In the context of LLM, we have two alphabets:

1. the character alphabet Σ which is typically a charset, i.e. Unicode characters or ASCII.

2. the token alphabet Γ which is the set of all possible token IDs in a language model's vocabulary, i.e. $\Gamma = \{0, 1, \ldots, |V| - 1\}$ where V is the vocabulary of the language model's tokenizer.

The tokenization function is a mapping from the character alphabet to the token alphabet:

$$\text{tok} : \Sigma^* \to \Gamma^*$$

The detokenization function is the inverse of the tokenization function:

$$\text{tok}^{-1} : \Gamma^* \to \Sigma^*$$

where $\text{tok}^{-1}(\text{tok}(x)) = x$ for all $x \in \Sigma^*$.

In the context of grammar-constrained decoding, we have a formal grammar G that we want to enforce on the output of the language model, e.g. JSON grammar. The language generated by the grammar G is denoted by $L(G)$. The image of $L(G)$ under the tokenization function tok forms another language $tok(L(G))$, which we call the *token language*.

Proposition 3.1. *Tokenization functions are generally **not** homomorphic from the character space to the token space under the concatenation operation.*

One can easily verify this by considering the following example.

Example 3.1. *With GPT-3 tokenizer, the brackets* `[` *and* `]` *are individually tokenized as* 58 *and* 60, *respectively, but the combined* `[]` *is tokenized as* 21737.

In contrast, the detokenization function is homomorphic, i.e. $d([t_1, t_2]) = [d(t_1), d(t_2)] \forall t_1, t_2 \in \Gamma^*$ as shown in Fig. 1 This is not surprising, as the detokenization function, as the name suggests, performs the following steps::

1. Mapping token IDs back to their corresponding tokens.
2. Concatenating these tokens to reconstruct the string.
3. Performing necessary post-processing to restore the original string format.

We can now state the following proposition:

Proposition 3.2. *The detokenization function is an homomorphism from the token space to the character space under the concatenation operation. The tokenization is thus an inverse homomorphism.*

All major subword encoding schemes, including Byte Pair Encoding (BPE) (Sennrich et al., 2016), WordPiece, and SentencePiece (Kudo and Richardson, 2018), exhibit this homomorphic property in their detokenization functions. We provide a more detailed analysis of how real-world tokenizers act as inverse homomorphisms in Appendix A. As a direct consequence of the tokenization function being an inverse homomorphism and the closure properties of context-free languages under inverse homomorphism (Theorem 2.2), we have the following proposition:

Proposition 3.3. *The token language $tok(L(G))$ is a context-free(regular) language for any context-free(regular) language $L(G)$.*

3.2 Token-space membership problem

Membership problem is a fundamental problem in formal language theory, which involves determining whether a given string s belongs to a formal language $L(G)$ defined by a specific grammar G. Membership problem is at the core of grammar-constrained decoding, where the goal is to ensure that the generated text adheres to specific constraints on the output. The main challenge in grammar-constrained decoding is to efficiently solve the membership problem[3] for a given grammar and a candidate sequence of tokens generated by a large language model (LLM).

The membership problem is *decidable* for context-free languages and regular languages (Hopcroft et al., 2006, Chap 7.4.4) , which

[3] More precisely, this is a partial membership problem, as we are interested in whether the string is a prefix of a valid string in the language.

are the two most common types of grammars used in grammar-constrained decoding. This is a well-established result and various algorithms, efficient or not, exist to solve the membership problem for context-free languages. Now that we have established that tokenization is an inverse homomorphism and the token language retains the structure of the original string language Thm. 3.3, we can make the following claim:

Proposition 3.4. *The membership problem for a context-free (or regular) language in the token space is decidable and can be solved with the same algorithms used in the character space.*

In practice, this means that we can:

1. solve the membership problem directly in the token space without the need to convert it back to the character space with the existing *parsing* algorithms for context-free languages.

2. test the recognition power of LLMs for a certain category, such as context-free languages, by writing a context-free grammar in character space and feeding it directly to the LLM without worrying about the structure being lost after tokenization.

3.3 Token-space automata construction

In this section, we explain how to construct a parser for the token language $tok(L)$ based on the parser for the string language L by using the homomorphism property of tokenization. The main idea is analogous to the construction of a pushdown automaton (PDA) for the inverse homomorphism of a context-free language sketched in Hopcroft et al. (2006, Theorem 7.30). Given a homomorphism h from alphabet Γ to alphabet Σ, and L being a context-free language over Σ, the construction of a PDA to accept language $L' = h^{-1}(L)$ is shown in Fig. 3. As stated in Thm. 2.1, we can always construct a PDA M which reads the input string in the alphabet Σ and accepts the language L. The construction of such a PDA is standard and well-known in the literature (Hopcroft et al., 2006, Chap 6.3.1). We then construct a PDA M' which reads the input string in the alphabet Γ (token IDs in our case) and accepts the language $L' = \text{tok}(L)$. The working of the PDA M' is as follows:

1. It applies the homomorphism h(detokenization in our case) to the input token ID a and puts the result $h(a)$ into the buffer, i.e. mapping the token IDs back to the character space.

2. The underlying PDA M in the character space reads the input characters $h(a)$ and updates its state and stack accordingly.

The resulting PDA M' reads the token IDs as input and decides whether the token IDs form a valid string in the token language $tok(L)$.

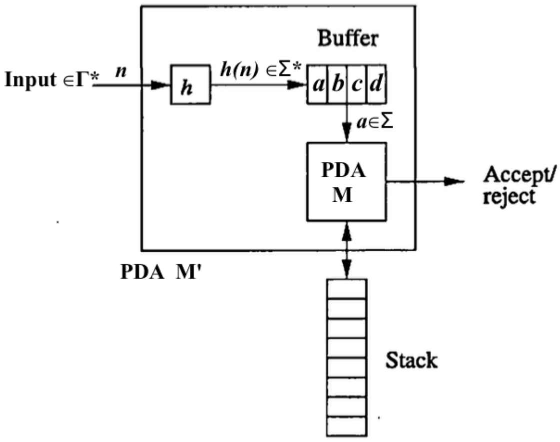

Figure 3: **Construction of a PDA M' to accept language $h^{-1}(L)$.** In the context of LLM, the input a is a token ID, the homomorphism h is *detokenization*, the *buffer* is used to store the token $h(a)$, the *PDA state* is the current state of the PDA in the character space, and the *PDA stack* is the stack of the PDA in the character space.

Once we have constructed the PDA M', we can use it to validate the generated token IDs from the LLM in an incremental manner as shown in 1 (Line 6). In case the tokenization is not an inverse homomorphism, we can still accept the token IDs from the LLM by converting them back to the character space and then feeding them to the PDA in the character space. But this approach is less efficient as it does not allow for incremental validation of the token IDs as shown in Algorithm 1 (Line 4).

Incorrect construction We also present an intuitive yet incorrect approach to construct a parser for the token language $tok(L)$. This approach may seem reasonable at first glance but is incorrect due to the non-homomorphic nature of tokenization The idea behind this approach consists of two steps:

1. Apply the tokenization function tok to the terminal symbols of the grammar G to obtain a new grammar G'.

Algorithm 1 Grammar Constrained Decoding

Require: Grammar G, parser P, language model **LLM**, , prompt $\mathbf{x}$, tokenization function tok and detokenization function tok^{-1}, token vocabulary V

Ensure: Generation $\mathbf{y}$ adhering to G

1: $\mathbf{y} : [List[int]] \leftarrow []$
2: $P.init(G)$
3: **loop**
4: $\quad P.update(tok^{-1}(\mathbf{y}_{:\mathbf{t}}))$ ▷ advance state of P with entire partial generation o (non-homomorphic)
5: $\quad$ **OR**
6: $\quad P.update(tok^{-1}(y_t))$ ▷ advance state of P with new token t (homomorphic)
7: $\quad \mathbf{m} \leftarrow [0,0,\dots,0]$ ▷ initialize mask as all zeros
8: $\quad$ **for** each token t_i in vocab V **do**
9: $\quad\quad$ **if** $P.accept(t_i)$ **then**
10: $\quad\quad\quad \mathbf{m}[i] \leftarrow 1$ ▷ set mask at position i to 1 if token is accepted
11: $\quad\quad$ **end if**
12: $\quad$ **end for**
13: $\quad \mathbf{p} \leftarrow \mathrm{LLM}(\mathbf{x} \oplus \mathbf{y})$ ▷ compute logits
14: $\quad \mathbf{p}' \leftarrow \mathbf{p} \odot \mathbf{m}$ ▷ element-wise product
15: $\quad y_{t+1} \leftarrow sample(\mathbf{p}')$ ▷ e.g., argmax or sample
16: $\quad$ **if** $t = EOS$ **then**
17: $\quad\quad$ **break**
18: $\quad$ **end if**
19: $\quad \mathbf{y}_{:\mathbf{t+1}} \leftarrow \mathbf{y}_{:\mathbf{t}}.\mathrm{append}(y_{t+1})$
20: **end loop**
21: **return** y ▷ optionally detokenize y to string

2. Build a parser for the new grammar G' to parse the token language $tok(L)$.

This approach would have been correct if the tokenization function was a homomorphism, i.e. $tok(ab) = tok(a)tok(b)$ for all $a,b \in \Sigma^*$. However, as we have shown in Thm. 3.1, the tokenization function is not a homomorphism. With the above incorrect approach, the parser would eliminate many valid token sequences that are not generated by the grammar G'.

3.4 Runtime complexity analysis

We analyze the runtime complexity of Algorithm 1. Given a prompt of length n tokens and a target generation of m tokens, the computation mainly involves step 5 to compute the mask of the next allowed tokens. Assuming the parsing complexity is $f(n)$ and the incremental parsing complexity is $\delta f(n)$ for each step, we have :

1. **Token verification**: For each token, Step 6 requires an incremental parsing of newly added token $\delta f(n)$.

2. **Vocabulary verification**: Without any optimization, verifying all tokens in the vocabulary results in a factor of $|V|$.

The total complexity of generating the entire sequence with GCD is:

$$\sum_{i=1}^{m} |V| \cdot \delta f(n) = |V| \cdot f(n+m)$$

In case the parser complexity is $O(n^3)$ (e.g., Earley parser), the total complexity would be:

$$O(|V| \cdot (n+m)^3)$$

for generating the entire sequence.

Non-homomorphic case

When the tokenization is not an inverse homomorphism, we lose the ability to validate the token IDs incrementally. We must convert the token IDs back to the character space and feed them to the PDA, which requires parsing the entire sequence from the beginning at each decoding step, as shown in Algorithm 1 (Line 4).

The **token verification** complexity will be

$$f(n+i)$$

for each token, and the complexity of generating the entire sequence will be

$$O(m \cdot |V| \cdot (n+m)^3)$$

, which is significantly higher than the homomorphic case.

4 Grammar with Unicode characters

When the grammar contains Unicode characters in the terminal alphabet, the tokenization process becomes more complex because a single character can be represented by multiple tokens which are not detokenizable independently. For example, the Chinese character 你 is tokenized as $[19526, 254]$ in the GPT-2 tokenizer but the token 19526 or 254 alone does not correspond to any character. Knowing only the token 19526 is insufficient to determine the character 你, as the context provided by the token 254 is also necessary, as illustrated in Fig. 1. This dependency of the next token breaks the homomorphic property of the detokenization function as shown in Fig. 1. However, considering that the tokenization function is actually operating on byte-level encodings of the Unicode characters, we can restore the homomorphic property by transforming the grammar to byte-level as well.

4.1 Grammar transformation

We propose a simple transformation that allows us to handle Unicode characters in the grammar-constrained decoding framework. The transformation involves transforming the grammar G from character alphabet Σ to byte alphabet B by substituting terminal symbols with their byte-level encodings. It is nothing more than just adding additional rules that map terminal symbols to their Unicode encodings in the grammar G, resulting in a new grammar G'.

Algorithm 2 Byte-Level Grammar Transformation

Require: Original Grammar $G = (N, T, P, S)$, parser P

Ensure: New Grammar G' suitable for Unicode encoding

Grammar transformation steps:

1: $N' \leftarrow N \cup \{\mathrm{T}\}$ ▷ Extend non-terminal set with Unicode terminal holder T

2: $T' \leftarrow \{\text{Unicode Encodings}\}$ ▷ Define new set of terminal symbols as Unicode bytes

3: $P' \leftarrow P \cup \{\mathrm{T} \rightarrow \text{Unicode Encodings}\}$ ▷ Extend production rules to include mappings from terminals to their byte encodings

4: $G' \leftarrow (N', T', P', S)$ ▷ Define new grammar with updated rules, terminals, non-terminals

---

The new grammar is defined as $G' = (N', T', P, S)$ where $N' = N \cup \{\mathrm{T}\}$ and $T' = \{\text{Unicode Encodings}\}$. The new rules are of the form $\mathrm{T} \rightarrow$ Unicode Encodings, where Unicode Encodings represent the byte-level encoding of the Unicode characters. The new grammar G' has a vocabulary of size 256, where each element corresponds to a byte.

With this new grammar G', we eliminate cases where a terminal symbol in the grammar corresponds to multiple tokens. This transformation ensures that:

- A single token may represent multiple terminal symbols (multiple bytes)
- A single terminal symbol(byte) corresponds to a single token.

As a result, the detokenization function is now homomorphic again from the token space to the byte space as shown in Fig. 1. Since ASCII characters are represented by a single byte in UTF-8 encoding, the byte-level construction is backward compatible with ASCII characters.

4.2 Complexity analysis

Given a grammar $G = (N, T, P, S)$, the grammar transformation in Algorithm 2 involves adding a fixed number of new rules $|N| + 256$ to the grammar. Both the *time* and *space* complexity of this transformation is $O(|N|)$, where $|N|$ is the number of non-terminal symbols in the grammar.

5 Experiment

We compare the runtime of grammar-constrained decoding in the token space under both homomorphic and non-homomorphic settings. For the non-homomorphic case, we assume the detokenization is not an inverse homomorphism, and we always parse the entire sequence from the beginning at each decoding step.

Experimental setup We evaluate the runtime of grammar-constrained decoding algorithms in both homomorphic (line 6 in Algorithm 1) and non-homomorphic (Line 4 in Algorithm 1) settings. We use the recursive descent parser as the parsing algorithm, and the LLaMA tokenizer for tokenization. We prompt the model to generate a json string containing N key-value pairs, where N ranges from 1 to 65. This prompt allows us to reliably measure the runtime of the decoding algorithm with different output lengths.

Grammar We use a simplified JSON grammar for the experiments as shown below:

$$\begin{aligned}
S &\rightarrow \text{Object} \\
\text{Object} &\rightarrow \{\ \} \mid \{\ \text{Members}\ \} \\
\text{Members} &\rightarrow \text{Pair} \mid \text{Pair}\ ,\ \text{Members} \\
\text{Pair} &\rightarrow \text{String}\ :\ \text{Value} \\
\text{Value} &\rightarrow \text{String} \mid \text{Number} \mid \text{Object} \mid \text{Array} \mid \text{true} \mid \text{false}
\end{aligned}$$

Array → [ ] | [Elements]

Elements → Value | Value , Elements

Metrics are:

- the constraint checking time for each decoding step,
- the cumulative constraint checking time for generating the entire sequence of tokens.

Results The growth of the runtime is shown in Fig. 4. We can observe that the incremental constraint checking in the homomorphic setting is significantly faster than the non-homomorphic setting.

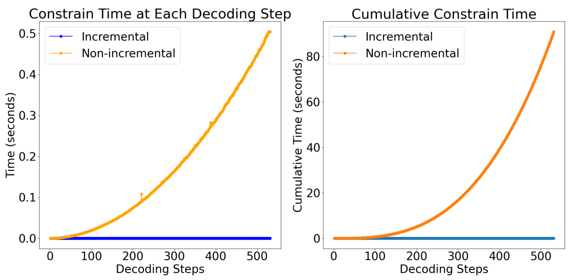

Figure 4: **Grammar-constrained decoding runtime.** The runtime of grammar-constrained decoding in both homomorphic(incremental) and non-homomorphic(non-incremental) settings. The left subfigure shows the runtime at each decoding step, while the right subfigure shows the cumulative runtime. (LLM forward pass time is not included in the runtime.)

6 Related Work

Guiding the decoding process of LLMs with grammar constraints is a well-established approach. Deutsch et al. (2019) proposed a general method to constrain the generation process of language models using a pushdown automaton, the computational model for context-free languages. Shin et al. (2021) and Poesia et al. (2022) suggested constraining the output of LLMs to a specific grammar to enhance performance in code synthesis and semantic parsing tasks. Shin et al. (2021) implemented an Earley parser to parse the grammar, while Poesia et al. (2022) and Geng et al. (2024b) used ANTLR (Parr, 2013) and Grammatical-Framework (Ranta, 2019) to generate the parser. Slatton (2023) and Jones (2023) contributed the feature of grammar-constrained decoding to the `Llama.cpp` library. Guidance (guidance-ai, 2024) and Outlines (Willard and Louf, 2023), as general-purpose constraint-generation frameworks, also added support for context-free grammars, with guidance-ai (2024) using an Earley parser for grammar parsing. Kuchnik et al. (2023) and Beurer-Kellner et al. (2024) discussed how to achieve efficient and effective constrained decoding for regular expressions and context-free grammar constraints, respectively.

Comparing to the existing work, our work focuses on the theoretical foundation of grammar-constrained decoding by leveraging the homomorphic properties of LLM tokenizers. However, there exist already implementations of grammar-constrained decoding that effectively utilize the homomorphic properties of LLM tokenizers without explicitly invoking the formal language theory. For example, guidance-ai (2024); Poesia et al. (2022); Beurer-Kellner et al. (2024) have achieved incremental parsing in the token space. Our work provides a formal foundation for these methods and extends them to handle Unicode characters in the grammar-constrained decoding framework.

7 Conclusion

In this work, we present a theoretical framework for grammar-constrained decoding from the formal language theory perspective. We show that the tokenization process is an inverse homomorphism, which maps a string to a sequence of tokens. We prove that the token language retains the structure of the original string language, which allows us to efficiently solve the membership problem in the token space. We show how this homomorphism property can be used to construct a parser for the token language based on the parser for the string language. Finally, we propose a simple transformation that allows us to handle Unicode characters in the grammar-constrained decoding framework, which extends to multilingual NLP applications.

8 Limitations

In this work, we extends the grammar-constrained decoding framework to handle Unicode characters by transforming the grammar to byte-level. However, there is one major limitation in the proposed method. EBNF(Extended Backus-Naur Form) is a widely used notation for specifying the syntax of programming languages. The proposed grammar transformation method is not directly applicable to grammar written in EBNF. The reason is that EBNF allows the use of meta-symbols like $*$, $+$, $|$ and range symbols like $[a - z]$. While most of the meta-symbols can be easily transformed to byte-level, the range symbols pose a challenge. For example, the range symbol `[你-我]` in EBNF cannot be directly transformed to byte-level because the byte-level encoding of the Unicode characters in the range is not contiguous. To address this limitation, an additional transformation would be required to handle the range symbols in the grammar. We leave the exploration of this problem for future work.

Our work doesn't improve the efficiency of the parsing algorithm per se, but rather provides a general construction that is compatible with existing parsing algorithms. With our construction, parsing in the token space can be done just **as fast as** in the string space, as long as the tokenizer is an inverse homomorphism (which is the case for all major tokenizers). The worst-case time complexity of parsing in the token space is still cubic, which is the same as parsing in the string space.

9 Responsible NLP

In this section, we respond to the call for responsible NLP research by discussing the implications of our work and suggesting guidelines for future research.

- potential risks: we don't see any potential risks in our work.
- privacy: our work does not involve any data collection or processing, so privacy is not a concern.
- energy consumption: our work involves parsing and decoding algorithms, which are run on CPU with negligible energy consumption. A few experiments running GCD with LLMs are run on A100 GPU for a few hours.
- AI assistant: we used copilot for code and paper writing, ChatGPT for paper review and revision suggestions.

References

Luca Beurer-Kellner, Marc Fischer, and Martin Vechev. 2024. Guiding llms the right way: Fast, non-invasive constrained generation.

Daniel Deutsch, Shyam Upadhyay, and Dan Roth. 2019. A general-purpose algorithm for constrained sequential inference. In *Proceedings of the 23rd Conference on Computational Natural Language Learning (CoNLL)*, pages 482–492, Hong Kong, China. Association for Computational Linguistics.

Saibo Geng, Berkay Döner, Chris Wendler, Martin Josifoski, and Robert West. 2024a. Sketch-guided constrained decoding for boosting blackbox large language models without logit access.

Saibo Geng, Martin Josifoski, Maxime Peyrard, and Robert West. 2024b. Grammar-constrained decoding for structured nlp tasks without finetuning.

guidance-ai. 2024. Guidance. `https://github.com/guidance-ai/guidance`. Accessed: 2024-03-12.

John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. 2006. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Addison-Wesley Longman Publishing Co., Inc., USA.

Evan Jones. 2023. Added grammar-based sampling to llama.cpp: support alternates in root rule. `https://github.com/ggerganov/llama.cpp/pull/1773/commits/421c6e1ca158a1b34e4f2f8d148e819c2fb7da62`. Accessed: 2024-05-21.

Michael Kuchnik, Virginia Smith, and George Amvrosiadis. 2023. Validating large language models with relm.

Taku Kudo and John Richardson. 2018. Sentencepiece: A simple and language independent subword tokenizer and detokenizer for neural text processing.

Zelong Li, Wenyue Hua, Hao Wang, He Zhu, and Yongfeng Zhang. 2024. Formal-llm: Integrating formal language and natural language for controllable llm-based agents.

Terence Parr. 2013. *The Definitive ANTLR 4 Reference*, 2nd edition. Pragmatic Bookshelf.

Gabriel Poesia, Oleksandr Polozov, Vu Le, Ashish Tiwari, Gustavo Soares, Christopher Meek, and Sumit Gulwani. 2022. Synchromesh: Reliable listings generation from pre-trained language models.

Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. Language models are unsupervised multitask learners.

Aarne Ranta. 2019. Grammatical framework: an interlingual grammar formalism. In *Proceedings of the 14th International Conference on Finite-State Methods and Natural Language Processing*, pages 1–2, Dresden, Germany. Association for Computational Linguistics.

Torsten Scholak, Nathan Schucher, and Dzmitry Bahdanau. 2021. PICARD: Parsing incrementally for constrained auto-regressive decoding from language models. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 9895–9901, Online and Punta Cana, Dominican Republic. Association for Computational Linguistics.

Rico Sennrich, Barry Haddow, and Alexandra Birch. 2016. Neural machine translation of rare words with subword units. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1715–1725, Berlin, Germany. Association for Computational Linguistics.

Richard Shin, Christopher Lin, Sam Thomson, Charles Chen, Subhro Roy, Emmanouil Antonios Platanios, Adam Pauls, Dan Klein, Jason Eisner, and Benjamin Van Durme. 2021. Constrained language models yield few-shot semantic parsers. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 7699–7715, Online and Punta Cana, Dominican Republic. Association for Computational Linguistics.

Grant Slatton. 2023. Added arbitrary context free grammar constraints to llama.cpp. `https://x.com/GrantSlatton/status/1657559506069463040`. Accessed: 2024-05-21.

Bailin Wang, Zi Wang, Xuezhi Wang, Yuan Cao, Rif A. Saurous, and Yoon Kim. 2023. Grammar prompting for domain-specific language generation with large language models. In *Advances in Neural Information Processing Systems*, volume 36, pages 65030–65055. Curran Associates, Inc.

Changhan Wang, Kyunghyun Cho, and Jiatao Gu. 2019. Neural machine translation with byte-level subwords.

Brandon T. Willard and Rémi Louf. 2023. Efficient guided generation for large language models.

Urchade Zaratiana, Nadi Tomeh, Pierre Holat, and Thierry Charnois. 2024. An autoregressive text-to-graph framework for joint entity and relation extraction. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, pages 19477–19487.

A Example of Homomorphic Tokenization API

In this section, we investigate the implementation of tokenization in real-world and show that they still preserve the context-free property of the source language.

Recall that a function $f : \Sigma^* \to \Gamma^*$ is homomorphic if $f(x \oplus y) = f(x) \oplus f(y)$ for any $x, y \in \Sigma^*$. In the context of LM, we want to know whether the decoding function `def tokenizer_decode(token_ids: List[int]) -> str:` is homomorphic. In the following, we will use the API of the `tokenizers` library[4] to illustrate the tokenization process. Generally speaking, the decoding function consists of two steps:

1. convert the token ids to tokens. `tokenizer.convert_ids_to_tokens(token_ids:List[int])-> List[str]`
2. join the tokens to form a string and apply some post-processing if needed. `tokenizer.convert_tokens_to_string(tokens:List[str]-> str)`

We will show that the step (2) can cause the homomorphism to break.

[4] `https://github.com/huggingface/tokenizers`

B Leading space in tokenization

Many tokenizers, including LLaMA, T5 employ a longstanding practice of distinguishing between prefix token and non-prefix token by baking the space character into the prefix token. This heuristic breaks the homomorphism because the leading space in the token will be lost if the token is at the beginning of a string. An example of Hello World tokenized by T5 is given below:

"Hello World" is tokenized as `[22172, 3186]` `["␣Hello", "␣World"]` by LLAMA.

We define h as the detokenization function and h^{-1} as the tokenization function: Given

$$h(22172) = \text{“}\llcorner\!\lrcorner Hello'',$$
$$h(3186) = \text{“}\llcorner\!\lrcorner World''.$$

We see that the homomorphism is broken:

$$\begin{aligned} h(22172, 3186) &= \text{“}Hello\llcorner\!\lrcorner World'' \\ &\neq \\ h(22172) + h(3186) &= \text{“}\llcorner\!\lrcorner Hello\llcorner\!\lrcorner World'' \end{aligned}$$

And if we reverse the order of the tokens, we still get the same problem:

$$\begin{aligned} h(3186, 22172) &= \text{“}World\llcorner\!\lrcorner Hello'' \\ &\neq \\ h(3186) + h(22172) &= \text{“}\llcorner\!\lrcorner World\llcorner\!\lrcorner Hello'' \end{aligned}$$

The above example shows that the tokenization process is not homomorphic and depends on the **context** of the token in the string, i.e. whether the token is at the beginning of the string or not.

However, this is break is relatively easy to fix by simply considering an intermediate CFL, i.e. the language with a leading space.

As the operation of adding a leading space to a string is a regular operation, we still get CFL.