Guiding LLMs The Right Way: Fast, Non-Invasive Constrained Generation

Luca Beurer-Kellner [1] **Marc Fischer** [1] **Martin Vechev** [1]

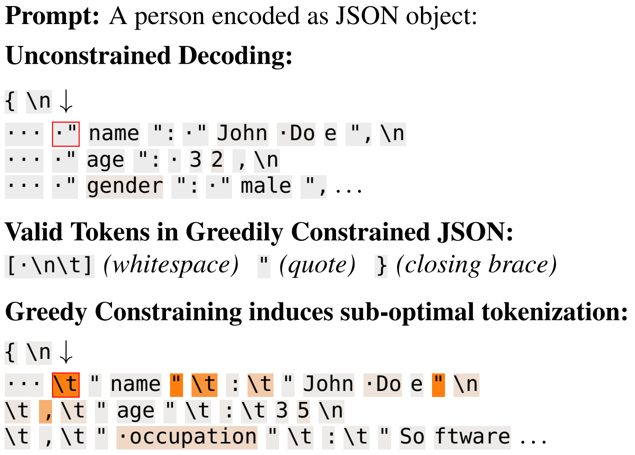
Abstract

To ensure that text generated by large language models (LLMs) is in an expected format, constrained decoding proposes to enforce strict formal language constraints during generation. However, as we show in this work, not only do such methods incur performance overhead during generation, but many of them also significantly impair task accuracy, if they do not correctly align the underlying LLM sub-word vocabularies with external constraints. To address this, we present a novel decoding algorithm, DOMINO, that can enforce constraints in a fully subword-aligned fashion, while leveraging pre-computation and speculative decoding to achieve virtually no overhead and in some cases even almost 2× speedup over unconstrained decoding – thereby outperforming existing approaches by a wide margin.

1. Introduction

The recent success of Large Language Models (LLMs) (Brown et al., 2020; Chen et al., 2021; OpenAI, 2023; Touvron et al., 2023a;b; Anil et al., 2023; Jiang et al., 2024) has lead to the development of various methods that facilitate constrained generation, a method that lets users tailor the output of an LLM to a specific task or format.

Constrained Decoding To ensure that text generated by an LLM adheres to syntactic constraints, these methods restrict the decoding procedure of an LLM in a way that only permits syntactically valid tokens at each sampling step. Doing so, the generated text can be ensured to adhere to constraints like high-level templates (Beurer-Kellner et al., 2023; Lundberg et al.), regular expressions (Beurer-Kellner et al., 2023; Lundberg et al.; Willard & Louf, 2023) or context-free grammars (Willard & Louf, 2023; Lundberg et al.). Constrained decoding provides numerous upsides. For instance, it guarantees that generated output will always adhere to the specified syntactic constraints, and reduces the need for ad-hoc parsing, retrying and prompting on top of LLMs. This facilitates the use of LLMs as part of larger pipelines or very specific tasks, without the need for fine-tuning or additional post-processing.



Prompt: A person encoded as JSON object:

Unconstrained Decoding:

```
{ \n ↓
··· ·" name ": ·" John ·Do e ", \n
··· ·" age ": · 3 2 , \n
··· ·" gender ": ·" male ", ...
```

Valid Tokens in Greedily Constrained JSON:

`[·\n\t]` *(whitespace)* `"` *(quote)* `}` *(closing brace)*

Greedy Constraining induces sub-optimal tokenization:

```
{ \n ↓
··· \t " name " \t : \t " John ·Do e " \n
\t , \t " age " \t : \t 3 5 \n
\t , \t " ·occupation " \t : \t " So ftware ...
```

Figure 1. Greedy (overly-invasive) constraining of LLMs can distort tokenization, leading to different output than with unconstrained decoding, even in the case where unconstrained generation would produce valid output for the same prompt. Gray boxes represent vocabulary tokens, orange hue is proportional to perplexity.

Key Challenge: Token Misalignment Since LLM subword token do not align directly with most given syntactic constraints, the key challenge in constrained decoding is to interface the LLM vocabulary with a syntactic constraint like *all output should be valid JSON*. We showcase this in Fig. 1: While in an unconstrained setting, the LLM picks `·"` as the fourth token during generation, naively restricting the LLM to only immediatly valid JSON grammar terminals like just `"` or `·`, leads to the less optimal choice of `\t` instead. By introducing such sub-optimal tokens, the distribution of a badly-constrained LLM can easily diverge from the unconstrained case, leading to a significant decrease in reasoning performance and therefore downstream accuracy. Here, the naively constrained LLM produces various high perplexity tokens, indicating that the model likely would not have chosen them otherwise (highlighted in Fig. 1). This is because naive constraining does not account for *bridge*

[*] Equal contribution [1] Department of Computer Science, ETH Zurich, Switzerland. Correspondence to: Luca Beurer-Kellner < luca.beurer-kellner@inf.ethz.ch>, Marc Fischer <marc.fischer@inf.ethz.ch>.

tokens that may span multiple parser terminals in the underlying grammar (e.g., whitespace and double quotes, in this example). While existing work on code generation has made this observation before (Poesia et al., 2022), solving this problem efficiently remains challenging, as the online computation of all bridge tokens at each decoding step, can be too costly in high-throughput environments.

This work: Efficient, Minimally-Invasive Constraining In this work, we study the token misalignment problem outlined above, and examine its consequences for constrained decoding, showing empirically that misalignment can lead to a significant decrease in downstream accuracy. Based on this observation, we propose the notion of *minimally invasive constrained decoding*: A form of constraining that enforces a grammar, but also intervenes as little as possible during generation, avoiding token misalignment and optimizing for faithful, low-perplexity model output.

Based on this, we propose a novel constrained decoding algorithm, DOMINO, which can enforce context-free grammars in a minimally-invasive way. In contrast to existing methods, DOMINO is highly efficient and incurs little to no overhead, and in many cases even increases the throughput of LLM inference over unconstrained generation, by leveraging pre-computation (Willard & Louf, 2023) and a novel speculative decoding procedure for constrained decoding. We compare DOMINO with other approaches in Table 1.

Main contributions In summary, our key contributions are:

- We identify the challenges of constrained decoding, most notably the correct and efficient alignment of sub-word tokens and grammar terminals (§2).
- We propose DOMINO, a novel constrained decoding algorithm, that addresses token misalignment and leverages pre-computation and speculative decoding for very low overhead generation (§3).
- An extensive evaluation thats shows that DOMINO is minimally-invasive, low-overhead, significantly outperforms other methods, and even exceeds unconstrained generation throughput in many cases (§4).

2. Challenges of Constrained Decoding

We first introduce the required background and highlight the challenges of efficient and token-aligned constraining.

Table 1. Overview of different constrained decoding methods

	Regex	CFG	Pre-Computed	Minimally Invasive
LMQL	✓	✗	✗	✗
GUIDANCE	✓	(✓)	✗	(✓) *
OUTLINES	✓	✓	(✓) †	✗
PICARD	✓	✓	✗	✗
SYNCHROMESH	✓	✓	✗	✓
LLAMA.CPP	✓	✓	✗	✓ +
GCD	✓	✓	✗	✓
DOMINO (ours)	✓	✓	✓	✓

* Boundary token healing, + Up to implementation. We observe violations in some cases, † For regex.

Algorithm 1 Constrained Decoding

```
Input: Checker $C$, LLM $f$, Tokenized Prompt $x$
Output: Completion $o$ adhering to $C$
$o \leftarrow []$
$C$.init()
loop
    $C$.update($o$)                         // advance state of $C$
    $\boldsymbol{m} \leftarrow C$.mask()                  // compute mask
    $\boldsymbol{v} \leftarrow f(x + o)$                     // compute logits
    $\boldsymbol{v}' \leftarrow \boldsymbol{m} \odot \boldsymbol{v}'$
    $t \leftarrow$ decode($\alpha'$)            // e.g., argmax or sample
    if $t = EOS$ then break
    $o$.append($t$)
end loop
return $o$                          // optionally detokenize
```

Large Language Models (LLMs) are machine learning models trained to complete a given text prompt. The current generation of these models operate on sub-word tokens, such as the commonly used Byte-Pair Encoding (BPE) (Sennrich et al., 2016; Kudo & Richardson, 2018). Conditioned on a sequence of input tokens $l_1, \ldots, l_n$ the model computes a probability distribution over the next token, from which the next token is decoded, i.e. chosen or sampled. This process is repeated for each token. In this context, we denote the set of all tokens, the vocabulary, as $\mathcal{V}$.

Constrained Decoding Algorithm 1 shows the general outline used by most constrained decoding approaches. A checker C, e.g., a parser or regex checker, is used to ensure that given an input x the generated output o adheres to the constraint. For each new token, C is first updated with the latest generated sequence and then used to generate a mask $\boldsymbol{m}$. This mask enforces that the next token must be a valid continuation. Each time the mask in Algorithm 1 rejects a token that would otherwise have been chosen, we say that the constrained decoding algorithm *intervenes* in the decoding process. We can therefore say that an algorithm that intervenes as little as possible is *minimally invasive*:

Definition 2.1 (Minimally invasive) *We consider a constrained decoding method* minimally invasive*, if every valid output that can be generated by an unconstrained model (for any given prompt), is also generated by the constrained model, given the same prompt.*

Replacing the naive constraining in Fig. 1 with a minimally invasive method, would lead to the same output as with unconstrained generation, up to the final closing `}` bracket of the JSON object. After that, an unconstrained model continues to generate text, which will not part of the JSON object. At this point, a minimally invasive decoder, however, stops decoding by forcing an the `EOS` token, to ensure that the output remains valid JSON.

As this example shows, non-minimally naive constrained decoding (as in Fig. 1) can lead to different overall output, compared to unconstrained generation. This in turn, can affect the model's task accuracy as we will show in §4, for a JSON-encoded version of the GSM8K dataset (Cobbe et al., 2021). There, naive constraining can reduce accuracy from 41.5% in the unconstrained case to 30.8%, while minimally invasive constraining actually achieves a slight increase in accuracy to 41.8% (five-shot, *Mistral 7B* Jiang et al. (2023)).

Next, we discuss the broad categories of existing constrained decoding approaches. Current approaches implement one or multiple of the following paradigms: i) *Regex-Based*, ii) *Online Parser-Guided* and iii) *Template-Based* approaches.

Regex-Based Decoding limited to regular expressions is supported by many frameworks (e.g., LMQL (Beurer-Kellner et al., 2023), OUTLINES (Willard & Louf, 2023), GUIDANCE (Lundberg et al.), LLAMA.CPP (Gerganov & et. al.)) and typically does not suffer from the token misalignment problem as it is simpler to check if a token is a legal continuation or not. For this, Willard & Louf (2023) also proposed an algorithm to pre-compute a regex checker for the model vocabulary offline, to be more efficient during inference.

Online Parser-Guided refers to running a parser and scanner in lock-step with an LLM, and then computing online, which tokens are valid continuations in each step. Such algorithms (PICARD (Scholak et al., 2021), GCD (Geng et al., 2023a), LLAMA.CPP) can support full CFGs and as Poesia et al. (2022) (SYNCHROMESH) demonstrated, can be built to support bridge tokens and thus to be minimally invasive. However, all of these approaches produce comparatively high inference overhead, since, in the worst case, they have to check the entire model vocabulary at each step.

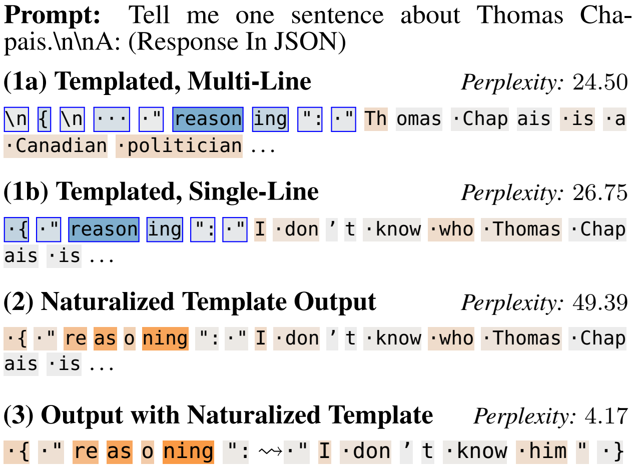

Prompt: Tell me one sentence about Thomas Chapais.\n\nA: (Response In JSON)
(1a) Templated, Multi-Line
Perplexity: 24.50
\n { \n ··· ·" reason ing ": ·" Th omas ·Chap ais ·is ·a ·Canadian ·politician ...
(1b) Templated, Single-Line
Perplexity: 26.75
·{ ·" reason ing ": ·" I ·don ’ t ·know ·who ·Thomas ·Chap ais ·is ...
(2) Naturalized Template Output
Perplexity: 49.39
·{ ·" re as o ning ": ·" I ·don ’ t ·know ·who ·Thomas ·Chap ais ·is ...
(3) Output with Naturalized Template
Perplexity: 4.17
·{ ·" re as o ning ": ⇝·" I ·don ’ t ·know ·him " ·}

Figure 2. Template-based tokens, marked as ▯, force unnatural tokenization and formatting, which can lead to different outputs and increased perplexity. Gray boxes represent vocabulary tokens, hue is proportional to perplexity.

Template-Based Approaches Since constrained generation can add additional overhead during inference, GUIDANCE and LMQL, propose a template-based approach, where some structure is fixed, and only parts of the output are sampled under a regex constraints. For instance, templated generation can be used to implement schema-driven JSON, where fields are fixed. Template-based decoding is efficient as templated tokens can be added deterministically during generation, without invoking the LLM, thereby requiring less model forward passes. However, this form of acceleration also has its downside, as discussed next.

Template-Induced Misalignment To insert templated tokens without invoking the model, an external tokenizer has to be used to translate template text into tokens. We showcase this in Fig. 2, where we use the template-based GUIDANCE with *Mistral-7B*, to generate text in JSON format, and compare it to unconstrained generation. (1a) and (1b) already show that depending on the concrete phrasing of a template (e.g., whitespace, formatting), output can vary significantly. To compare to unconstrained generation, we use the model to also generate the template text, but without imposing a fixed tokenization (algorithm in App. B). We then generate output with this *naturalized* template (3) to contrast it with template-induced tokenization.

Overall, template-based outputs exhibit clearly different outputs and much higher perplexities (perplexities 24.50 − 26.75), when compared to unconstrained generation (perplexity 4.17). Further, we observe that naturalizing the full template-based output under the model-preferred tokenization (2) in Fig. 2, results in a form of perplexity explosion (49.39), indicating that without invasive constraining, the model is highly unlikely to generate such output. While perplexity and output differences are not directly indicative of output quality, our evaluation in §4.1 extends on this experiment, and indeed shows that invasive template-based generation can lead to a significant drop in task accuracy, compared to unconstrained and less invasive approaches.

Token Healing To reduce the impact of template-induced misalignment, GUIDANCE implements *token healing* (Lundberg & Ribeiro), a method that attempts to improve tokenization at the transition points between templated and non-templated tokens. Token healing truncates the prompt to the second-to-last token boundary, and enforces the rest of the prompt as a prefix on generation. This can be effective at avoiding some tokenization issues and integrating bridge tokens, yet, most templated tokens remain fixed as demonstrated in Fig. 2, where token healing is already in effect, and still leads to significant differences in output.

Key Challenge This section, as summarized in Table 1, demonstrates that the key challenge in constrained decoding is to find a method that allows i) expressivity for regex, context-free grammars and templated decoding, ii) is minimally invasive in all settings and iii) has low inference overhead. Next, we discuss Domino, a novel constrained decoding algorithm, that fits these requirements.

3. Efficient Aligned Constrained Decoding

In this section, we address the above challenges by introducing Domino, a fast minimally-invasive constrained decoding algorithm, showcased in Fig. 3. First, §3.1 discusses the necessary preliminaries. Then, in §3.2-§3.5 we present the main algorithm and then discuss further details.

3.1. Preliminaries

Formal Languages A formal language is a set L of finite strings over a given alphabet Σ. A language is called regular if it can be described by a regular expression. A language is context-free if it can be described by a context-free grammar (CFG). Such a grammar is described by a set of production rules $A = \alpha B \gamma$, where upper case names (A, B) are non-terminals that are recursively extended and lower case greek characters (e.g., α, β) are terminals that are part of the language. For an example, see the grammar in Fig. 3 (a), with the non-terminal `E` and terminals `int`, `(`, `)`, `+`, which are defined either by a regex or a literal string.

All regular languages can be recognized by a non-deterministic finite automata (NFA). A NFA is a set of states with a start state and a set of accepting states. Connected with transitions that are labeled with a character or the empty string ε. We traverse or execute an NFA by starting in the start state and whenever we read a character follow the appropriate transitions or any ε-transitions. For a regular expression, we can construct a NFA (McNaughton & Yamada, 1960; Thompson, 1968), that when fed a string, character by character, ends in a set of state including at least one accepting state, if and only if the string matches the regular expression. To illustrate this approach consider the grammar given in Fig. 3 (a). The terminal `int` is given by a regex that allows any positive integer (not starting with leading zeros) or one or more zeros. Fig. 4 shows an NFA for this regex.

3.2. Character Scanner

Like classical parsers, Domino separates the CFG recognition into a parser and a scanner (or lexer). The parser enforces the high-level structure of the language, e.g., the rules in a context-free grammar, and the scanner enforces the low-level structure, i.e., the regular expressions of the terminals. The key idea is that any legal program in a context-free grammar is a sequence of terminals, or formally:

Lemma 3.1 *Let L_G be the language described by a CFG G. Further, let $r_1, \dots, r_n$ be the regular expressions of the terminals of G and the $r_{\text{EOS}} = \$$. Then, it holds that:*

- *The union of these regular expressions $r = r_1 | \dots | r_n | r_{\text{EOS}}$ matches any terminal in G.*
- *The regular expression $R = r+$ matches all non-empty sequences of terminals in the language.*
- *The language L_R described by R contains all legal programs in G, i.e., $L_G \subseteq L_R$, but also*
- *L_R contains some illegal programs, i.e., $L_R \nsubseteq L_G$.*

To construct an NFA for L_R, we construct the NFA for the regex r_i for each terminal in the grammar. From the accepting states of the invidual terminal NFAs, we add an ε-transition to a single NFA accepting state q_a. Similarly, we add a transition from the start state q_0 to the start state of each DFA. Finally, we add a transition from q_0 to q_a, to allow for the chaining of multiple tokens. This is the standard disjuction construction for regex NFAs, however, we do this explictly to track the sub-automata correspoding to each terminal. This construction is showcased in Fig. 3 (b) for the grammar in Fig. 3 (a). There, the boxes in the middle correspond to individual NFAs like `int` as shown in Fig. 4.

3.3. Vocabulary-Aligned Subterminal Tree

Using a scanner S at generation time can enforce that the generated string will be in L_R, but not neccesarily in L_G. To ensure this, we need to also run a parser P on the output as it is generated and dynamically allow and disallow some of the transitions in S, according to the current parser state.

For this, we first lift S from the character level to the (sub)terminal level used by the parser. To do so, for each node in S we follow the transitions for each token in the vocabulary and enumerate all reachable states. Particularly, we track which terminal NFAs are partially or fully traversed. As terminals are not neccesarily aligned with the token vocabulary, we introduce the notion of a *subterminal* as a part of a terminal NFA. In particular, for terminal α, e.g., `int`, we say that S reads a:

- **Full terminal** if it passes q_0 and reaches an accepting state q_α in the NFA for the terminal. We denote this as □. If we end in an accepting state, that also allows

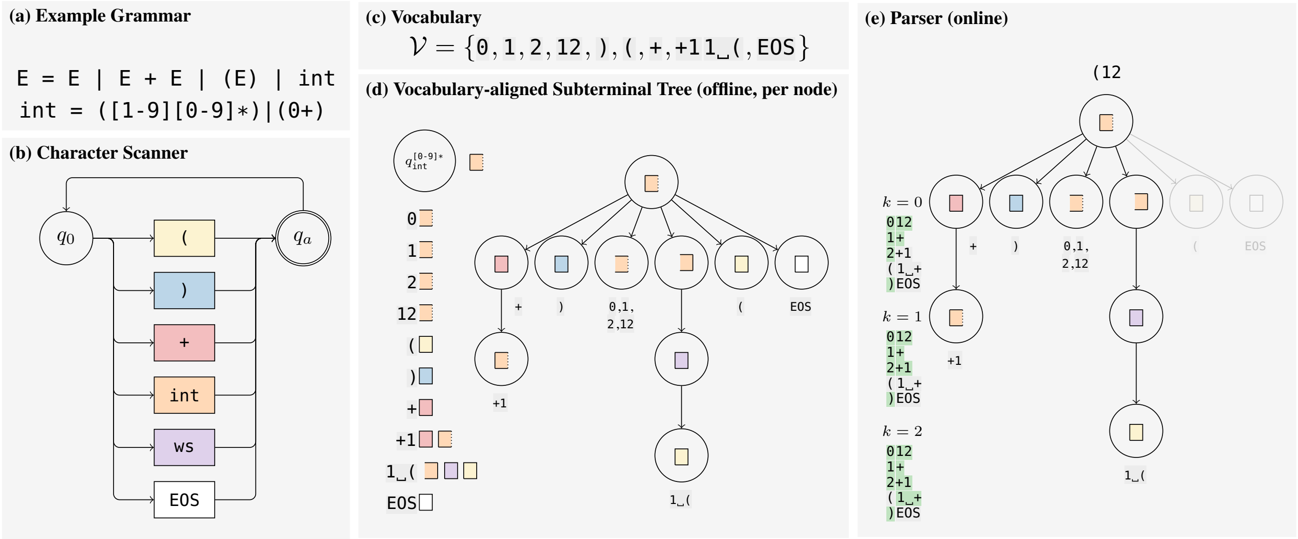


Figure 3. Running example and overview of DOMINO. (a) shows an example grammar, (b) the character level NFA for this language, (d) one of the per-state subterminal trees for the grammar in (c). (e) shows how a parser can be used to prune this tree at inference time and obtain token masks efficiently by traversing the tree.

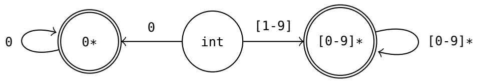


Figure 4. NFA for the `int` terminal from Fig. 3 (a). Traversed from node `int` this NFA accepts all legal inputs for the terminal.

further transitions, such as both accepting states in Fig. 4, we also consider this a full terminal, but allow further subterminals within the terminal NFA (□).

- **Start subterminal** if starting from q_0 we reach a non-accepting but valid state q_α for the NFA of the terminal. We denote this as □.
- **End subterminal** if starting from a state q_α within the NFA for α we reach an accepting state for α. We denote this as □.
- **Continuation subterminal** if starting from a state q_α within the NFA for α we reach another non-accepting state for α. We denote this as □.

We visit every state q in S, obtain the current (sub)terminal α and, for each vocabulary token $l \in \mathcal{V}$, enumerate all possible subterminal sequences $\{\alpha_1^j, \ldots, \alpha_{m_j}^j\}_j$. Typically there is only one such sequence unless there is ambiguity in the grammar, e.g., in C-style languages we can not be sure if we are reading a variable name of a keyword. Ignoring these edge cases, we show an example of this on the left hand side of Fig. 3 (d) for the vocabulary given in Fig. 3 (c), where we visualize the subterminals in the previously introduced box notation with colors correspoding to the terminals in Fig. 3 (b). Note that there may be tokens in the vocabulary for which no continuation is possible, e.g., if we add an `a` token to the language. After enumerating all these subterminal sequences, we organize them into a prefix tree T_q, where we attach the corresponding vocabulary tokens l as values to the nodes. We formalize this procedure in Algorithm 2.

Algorithm 2 Construct Terminal Tree

```
Input: CFG $G$, Alphabet $\Sigma$, Vocabulary $\mathcal{V}$
Output: Scanner $S$
$\mathcal{T} = \{\}$
for all $q \in S$.states() do
    $\alpha \leftarrow q$.subterminal()    // get current (sub)terminal
    for all $l \in \mathcal{V}$ do
        $\{\alpha_1^j, \ldots, \alpha_{m_j}^j\}_j \leftarrow q$.traverse($l$)
        $\mathcal{T} \leftarrow \mathcal{T} \cup \{(\alpha_1^j, \ldots, \alpha_{m_j}^j), l\}_j$
    end for
    $T_q \leftarrow$ PrefixTree($\mathcal{T}$)
end for
```

3.4. Parser

While subterminal trees can be pre-computed, we still need a parser at inference time, to disallow illegal continuations. For example, consider the prefix tree Fig. 3 (d). If so far, we have observed the sequence `(12` and correctly advanced the parser and scanner to this state, we are in an `int` terminal □, that can still be extended further. Following the prefix tree in Fig. 3 (d) would permit continuations such as `(`, which are clearly illegal in the grammar. Thus, we need to disallow these continuations dynamically by consulting the parser.

At inference time traversing the so-far generated sequence o through the scanner and parser will result in a scanner state S and parser state P. The active state of S will be a set of states $q_1, \ldots, s_m$. The active state of P will be a parser that tracks rules that can match the output o so far. For each of these nodes q_i we retrive the corresponding subterminal tree T_{q_i} and use the parser to check which of the possible continuations are legal. The depths to which we follow these terminal sequences in the prefix-tree is determined by the so-called *lookahead parameter* k.

We showcase this in Fig. 3 (e), where so far we have read the input `(12`. The parser thus knows that it has seen the partial rule `(E)`, in which recusively `E` was initialized with the `int` terminal. The scanner S has been advanced similarly and has a single active state s that corresponds to that for an `int` terminal 🟧, as shown in Fig. 3 (d). In the corresponding prefix tree T_s we can now check each outbound edge with the parser and find that the `EOS` and `(` can not produce legal (sub)terminal sequences, but all other tokens do. After traversing one level of this tree ($k = 0$) this includes all number terminals, `+` and `)`. By increasing k and checking further paths in the tree we can also include the tokens `+1` and `1␣(` at $k = 1$ and $k = 2$ respectively.

3.5. DOMINO

Based on Algorithm 1, DOMINO implements constraining that leverages subterminal trees to efficiently check for legal continuations in $\mathcal{V}$, that, by construction, line up with the current state, just like the pieces in a game of domino.

For this, we compute the character scanner S and corresponding prefix trees T_q for all $q \in S$ before inference starts (offline). When `update` is called in Algorithm 1, we can then advance the scanner and parser state. When `mask` is invoked , we traverse the corresponding T_q to the desired depth, and take the union over the associated tokens to compute the current mask. Here, $k = 0$ is already sufficient to ensure that the generated output is in L_G. If we always traverse the full prefix-trees ($k = \infty$ or sufficently large), this approach is minimally invasive, as all valid tokens will eventually be reached and included in the mask.

Overall, this enables us to handle expressive constraining with far less overhead than fully online approaches, as the size of subterminal trees is much smaller than the size of the model vocabulary, which we would need to traverse otherwise. Similarly DOMINO can also be extended to other forms of constraining, e.g. to execute GUIDANCE programs.

Further Optimizations On top of this, DOMINO also supports an optimization already present in LLAMA.CPP (Gerganov & et. al.)'s fully online approach, which we term *opportunistic masking*: Rather than computing the mask for the full logit vectors as in Algorithm 1, we can first run the decode step of the LLM and then use the parser to check the model-proposed token first. Only if incorrect, we need to compute the rest of the mask, and thus let the LLM guide decoding.

To realize this in DOMINO, rather than traversing the trees T_q from the root, we first determine the nodes linked to the proposed token, and only then check if there exists a parser-allowed path from the root to this node. This is can be very effective when an LLM already naturally adheres to a grammar, and parser transitions are expensive. In DOMINO, oppurtunistic masking is enabled using a runtime flag.

We finally want to note, that *token healing* can be implmented in a similar way to GUIDANCE (Lundberg et al.), i.e., by stripping the input back to the last token boundary and changing the beginning of the grammar to force a prefix. However, this means that the grammar needs to be recompiled for the current problem and can not neccesarily be shared between multiple instance. Note however, that this is of lesser concern in DOMINO, as it is only relevent for the first boundary with the prompt, where all other boundaries are embedded seamlessly into the grammar.

3.6. Speculative Decoding

Speculative Sampling (Chen et al., 2023) is a technique to speed up the LLM sampling process, by using a smaller LLM to propose multiple tokens and then only evaluate the full large LLM to confirm this token choice. This is efficient, as the parallel nature of Transformer-based models Vaswani et al. (2017), allows to validate multiple tokens with a single forward pass, where rejected tokens can simply be discarded without the need to backtrack. In DOMINO, we adopt a similar approach to further speed up inference, based on parser and scanner state. At any time, the active scanner state is (largely) given by the most recently read subterminal α (or $\{\alpha_j\}_j$ if the NFA could be in multiple possible subterminals). Similarly, we let β denote some substate of the currently used parser, e.g., the currently applied rule. Conditioned on α, β we can then learn a simple, count-based model for speculative next token prediction:

$$\mathbb{P}(l \mid \alpha, \beta) = \frac{\#\{\text{LLM chose } l \text{ in state } (\alpha, \beta)\}}{\#\{\text{reached state } (\alpha, \beta)\}}.$$

As structured languages often are very predictable and α, β can be strongly indicative of the next token, this mechanism can lead to massive speed-up during inference. Further, as we learn these counts over the parser state β, we only learn to predict tokens that are legal in the language.

In practice, we parameterize s tokens to be predicted this way at a time, if the $\mathbb{P}(l \mid \alpha, \beta)$ is sufficiently large. This form of speculative decoding is independent of standard speculative decoding applied to the underlying LLM (Chen et al., 2023), and could even be applied jointly with it.

Table 2. Task Accuracy of different constrained decoding methods on *GSM8K* Cobbe et al. (2021) and CoNLL2003 Tjong Kim Sang (2002) datasets (400 test samples). All experiments rely on 5-shot prompting with demonstrations taken from the training split.

Dataset	Model	Method	Accuracy	Well-Formed	Perplexity	Performance Impact
GSM8K	*Mistral 7B*	Unconstrained	0.415	0.952	1.636	1.0×
		GUIDANCE Lundberg et al.	0.345	0.960	1.624	0.98×
		GUIDANCE ^{WS} Lundberg et al.	0.403	0.976	1.737	0.54×
		`llama.cpp` Gerganov & et. al.	0.375	0.973	1.751	0.80×
		DOMINO ($k = \infty$)	**0.418**	0.968	1.739	**1.77**×
	Llama-2 13B	Unconstrained	**0.262**	0.904	1.650	1.0×
		GUIDANCE Lundberg et al.	0.152	0.947	1.659	1.12×
		GUIDANCE ^{WS} Lundberg et al.	0.259	0.977	1.760	0.73×
		`llama.cpp` Gerganov & et. al.	0.237	0.978	1.780	0.86×
		DOMINO ($k = \infty$)	**0.262**	0.920	1.750	**1.66**×
CoNLL2003	*Mistral 7B*	Unconstrained	**0.21**	0.988	1.573	1.0×
		GUIDANCE Lundberg et al.	0.098	0.998	1.780	2.02×
		GUIDANCE ^{WS} Lundberg et al.	0.19	0.998	1.896	0.82×
		`llama.cpp` Gerganov & et. al.	0.117	0.995	1.560	0.80×
		DOMINO ($k = \infty$)	**0.21**	0.988	1.902	**2.66**×
	Llama-2 13B	Unconstrained	**0.09**	0.897	1.579	1.0×
		GUIDANCE Lundberg et al.	0.062	1.000	1.820	2.18×
		GUIDANCE ^{WS} Lundberg et al.	0.087	0.980	1.767	0.90×
		`llama.cpp` Gerganov & et. al.	0.080	0.922	1.786	0.86×
		DOMINO ($k = \infty$)	**0.09**	0.897	1.812	**2.71**×

^{WS} GUIDANCE CFG program with flexible whitespace and formatting.

4. Experimental Evaluation

We evaluate DOMINO in terms of downstream task accuracy, compare its performance to multiple baselines and ablate key parameters such as k.

Setup We evaluate on the *Mistral 7B* (Jiang et al., 2023) and the *Llama-2 13B* (Touvron et al., 2023c) language models. As inference backends, we rely on both, `transformers` (Wolf et al., 2019) and `llama.cpp` (Gerganov & et. al.) on NVIDIA A100 40GB or H100 80GB GPUs. Because of its nature, we explicitly evaluate DOMINO in an offline setting, where all grammars are known ahead of time and do not vary across inference requests.

Datasets We assess downstream accuracy of different constraining methods with the *GSM8K* (Cobbe et al., 2021) benchmark for math reasoning and *CoNLL-2003* (Tjong Kim Sang, 2002) for named-entity recognition (subset of 400 test samples). To examine the performance properties of different decoding methods, we compare their overhead over unconstrained decoding for different tasks, including the constrained generation of *JSON*, *JSON with Schema*, the *C Programming Language*, *XML with Schema* and more static, regex-based generation templates similar to GUIDANCE or LMQL programs with simple structure.

Baselines We consider the following baselines:

- **Unconstrained Generation** We generate output without any form of constraints, using the same prompts and inference backend.
- **GUIDANCE Programs** (Lundberg et al.) We construct GUIDANCE programs to generate output in the desired output formats. We compare template-based programs (standard approach) and CFG-based variants, which are whitespace agnostic (comparison in App. A).
- **LLAMA.CPP Grammars** (Gerganov & et. al.) We rely on LLAMA.CPP's support for `ebnf` grammars as an online parsing baseline (also representative of (Poesia et al., 2022) and Geng et al. (2023b)).

4.1. Task Accuracy

We first evaluate the impact of different constrained decoding methods on downstream accuracy. For this, we use the *GSM8K* benchmark for math reasoning and *CoNLL-2003* for named-entity recognition.

Setup For both datasets, we prompt and constrain the models to generate a response in a given JSON format, instead of free-text reasoning (see App. D for examples). Our prompts consist of 5 few-shot demonstrations from the training split, for which we manually construct the corresponding JSON

Table 3. Impact on throughput of constrained decoding methods with different grammars, compared to unconstrained generation. We report the change in throughput compared to generating unconstrained output with the same model and inference backend. As CFG$^{\text{accel}}$ we report DOMINO opportunistic masking or speculative decoding, depending on which is more effective.

		llama.cpp[op] Gerganov & et. al.	guidance[HF] Lundberg et al.		DOMINO[HF] (Ours)	
Grammar	Model	CFG	Template$^{\downarrow}$	CFG	CFG	CFG$^{\text{accel}}$
JSON (no schema)	*Mistral 7B*	0.79×	1.03×	0.81×	0.88×	**0.96**× (opportunistic)
	Llama 13B	0.83×	1.03×	0.86×	0.95×	**1.12**× (spec. $s = 10$)
JSON (GSM8K schema, see App. D)	*Mistral 7B*	0.80×	0.77×	0.59×	0.99×	**1.77**× (spec. $s = 10$)
	Llama 13B	0.86×	0.94×	0.74×	0.99×	**1.66**× (spec. $s = 10$)
C Programming Language	*Mistral 7B*	0.74×	-	-	0.41×	**0.78**× (opportunistic)
	Llama 13B	0.81×	-	-	0.54×	**0.85**× (opportunistic)
XML (with schema)	*Mistral 7B*	0.80×	-	-	0.94×	**1.52**× (spec. $s = 10$)
	Llama 13B	0.87×	-	-	0.98×	**1.84**× (spec. $s = 10$)
Fixed Template	*Mistral 7B*	0.55×	1.95×	0.92×	0.97×	**1.30**× (spec. $s = 10$)
	Llama 13B	0.69×	2.05×	1.06×	0.99×	**1.91**× (spec. $s = 10$)

[op] `llama.cpp` always runs with opportunistic masking, [HF] Using the `transformers` library as inference backend.
$^{\downarrow}$ GUIDANCE templates lead to significantly worse accuracy compared to CFGs (cf. Table 2), but we show them here for completeness.

response. We then compare the accuracy of the generated JSON responses, considering both the validity of the JSON format and the accuracy of the final response. We also measure perplexity of the generated output and the overhead over unconstrained generation in terms of throughput.

Results As documented in Table 2, DOMINO achieves the best accuracy for all tasks, while also improving throughput well beyond unconstrained generation. In all cases, DOMINO's accuracy is the same or improved compared to unconstrained generation, indicating very low or no invasiveness. In contrast, with standard GUIDANCE, we observe clear artifacts of invasiveness, as accuracy drops by up to 11% points compared to unconstrained generation. And, while GUIDANCE's inference optimizations do increase throughput (up to $2.02\times$), DOMINO accellerates inference even further (up to $2.71\times$), while also maintaining high accuracy. Implementing minimally invasive GUIDANCE $^{\text{WS}}$ programs with flexible whitespace and formatting restores some accuracy, but not fully, and also lowers throughput significantly to $\sim 0.8\times$. LLAMA.CPP's online parsing approach also does not appear to be fully non-invasive, although accuracy seems less impaired, while throughput is also consistently reduced to $\sim 0.8\times$.

4.2. Parameter Study

Next, we investigate the key parameters of DOMINO.

Lookahead To experiment with lookahead parameter k, we evaluate on GSM8K as before, but varying k. We report the results in Table 4. We observe that lower k values impair performance significantly. Manual inspection shows that depending on k, the model is forced into different whitespacing behavior, as bridge tokens like `},` are unavailable, leading to irregularities. This can affect reasoning, e.g. *Llama-2* is unable to produce object lists of length greater than 1, if relevant bridge tokens are missing. DOMINO with $k = \infty$, however, recovers and even slightly exceeds unconstrained accuracy, demonstrating minimal invasiveness.

Table 4. GSM8K task accuracy with different lookahead k.

Configuration	*Mistral 7B*	*Llama-2 13B*
Unconstrained	0.415	0.155
DOMINO ($k = 0$)	0.308	0.0
DOMINO ($k = 1$)	0.1	0.036
DOMINO ($k = \infty$)	**0.418**	**0.157**

Speculation and Opportunistic Masking We also experiment with the number of speculative tokens s we propose at each decoding step, to speed up generation. For this, we compare the generation of schema-driven JSON and free-form JSON text with *Mistral 7B*. We form priors on 10 randoms samples, and then measure mean performance across 100 generated outputs per configuration, without updating counts. We report the results in Fig. 5.

We find speculative decoding with $s \in \{6, 8, 10\}$ to be particularly effective for schema-driven JSON generation, as it achieves a throughput of 1.7x compared to unconstrained

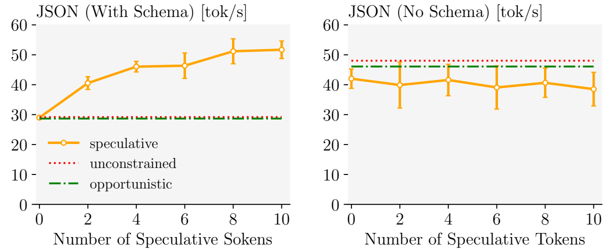


Figure 5. Impact of the number of speculative tokens k on throughput (tokens per second) with *Mistral 7B* and JSON generation with and without schema, using DOMINO with `transformers` LLMs.

generation. On free-form JSON output, speculation is not effective, and DOMINO opportunistic masking is preferable, incurring only a low 4.21% overhead.

4.3. Efficiency

Next, we examine throughput and efficiency of DOMINO. For this, we compare unconstrained generation thrroughput of each backend, and report the relative differences when running with constrained generation. We do not include DOMINO's precomputation time as part of the reported throughputs. We note that for the tested grammars, it ranges from 1-5s, with C being an outlier at around 20s.

Grammars We compare different constraining tasks, as shown in Table 3. For each task, we prepare a small set of relatively general inputs, prompting the model to generate a response in the general distribution of the given output format (details in App. C). Where practical, we also implement GUIDANCE programs, using their custom CFG-like syntax.

Setup We run 100 repetitions per configuration. In each, we sample one of 5 different prompts per workload, and sample output of up to 128 tokens from the model, using a temperature value of 1.0. This way, we ensure that the model produces in-distribution output, but that it still exhibits diversity. Before measuring, we run 10 repetitions of warmup, allowing our speculative mechanisms to form a prior. After that, the learned priors remain fixed.

Results As shown in Table 3, DOMINO is highly effective and clearly reduces the compuational load of constraining at inference time. DOMINO outcompetes both GUIDANCE and `llama.cpp`'s online parsing approach significantly. For grammars with predictable structure (e.g. schema-driven formats), speculative decoding is particularly effective, leading to up to 77% higher throughput over unconstrained generation, while remaining minimally invasive.

C code generation induces the most overhead, which can be explained by the fact that the C grammar is the most complex of the tested workloads. Here, speculative decoding does not bring any benefits, as the C code is to hard to predict using our simple count-based model. However, by relying on DOMINO's opportunistic masking mode, we still outcompete `llama.cpp`, running at $0.78\times$ vs. $0.74\times$.

5. Conclusion

We have shown the need for minimally invasive, highly-efficient constrained decoding methods. As first instantiation of this for grammars, we presented DOMINO, which leverages precomputation, speculative decoding and opportunistic masking, to implement minimally invasive constraining (no accuracy loss), often overhead-free or even faster generation, and thus, high-throughput inference.

Impact Statement

This paper presents work whose goal is to advance the field of Machine Learning. There are many potential societal consequences of our work, none which we feel must be specifically highlighted here.

Acknowledgements

This work has received funding from the Swiss State Secretariat for Education, Research and Innovation (SERI). This work has been done as part of the SERI grant SAFEAI (Certified Safe, Fair and Robust Artificial Intelligence, contract no. MB22.00088, SERI-funded ERC Consolidator Grant).

References

Anil, R., Borgeaud, S., Wu, Y., Alayrac, J., Yu, J., Soricut, R., Schalkwyk, J., Dai, A. M., Hauth, A., Millican, K., Silver, D., Petrov, S., Johnson, M., Antonoglou, I., Schrittwieser, J., Glaese, A., Chen, J., Pitler, E., Lillicrap, T. P., Lazaridou, A., Firat, O., Molloy, J., Isard, M., Barham, P. R., Hennigan, T., Lee, B., Viola, F., Reynolds, M., Xu, Y., Doherty, R., Collins, E., Meyer, C., Rutherford, E., Moreira, E., Ayoub, K., Goel, M., Tucker, G., Piqueras, E., Krikun, M., Barr, I., Savinov, N., Danihelka, I., Roelofs, B., White, A., Andreassen, A., von Glehn, T., Yagati, L., Kazemi, M., Gonzalez, L., Khalman, M., Sygnowski, J., and et al. Gemini: A family of highly capable multimodal models. *ArXiv preprint*, 2023. URL `https://arxiv.org/abs/2312.11805`.

Beurer-Kellner, L., Fischer, M., and Vechev, M. T. Prompting is programming: A query language for large language models. *Proc. ACM Program. Lang.*, (PLDI), 2023.

Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D. M., Wu, J., Winter, C., Hesse, C., Chen, M., Sigler, E., Litwin, M., Gray, S., Chess, B., Clark, J., Berner, C., McCandlish, S., Radford, A., Sutskever, I., and Amodei, D. Language models are few-shot learners. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020.

Chen, C., Borgeaud, S., Irving, G., Lespiau, J., Sifre, L., and Jumper, J. Accelerating large language model decoding with speculative sampling. *ArXiv preprint*, 2023.

Chen, M., Tworek, J., Jun, H., Yuan, Q., de Oliveira Pinto, H. P., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., Ray, A., Puri, R., Krueger, G., Petrov, M., Khlaaf, H., Sastry, G., Mishkin, P., Chan, B., Gray, S., Ryder, N., Pavlov, M., Power, A., Kaiser, L., Bavarian, M., Winter, C., Tillet, P., Such, F. P., Cummings, D., Plappert, M., Chantzis, F., Barnes, E., Herbert-Voss, A., Guss, W. H., Nichol, A., Paino, A., Tezak, N., Tang, J., Babuschkin, I., Balaji, S., Jain, S., Saunders, W., Hesse, C., Carr, A. N., Leike, J., Achiam, J., Misra, V., Morikawa, E., Radford, A., Knight, M., Brundage, M., Murati, M., Mayer, K., Welinder, P., McGrew, B., Amodei, D., McCandlish, S., Sutskever, I., and Zaremba, W. Evaluating large language models trained on code. *ArXiv preprint*, 2021.

Cobbe, K., Kosaraju, V., Bavarian, M., Chen, M., Jun, H., Kaiser, L., Plappert, M., Tworek, J., Hilton, J., Nakano, R., et al. Training verifiers to solve math word problems. *ArXiv preprint*, 2021.

Geng, S., Josifoski, M., Peyrard, M., and West, R. Grammar-constrained decoding for structured NLP tasks without finetuning. In *EMNLP*, 2023a.

Geng, S., Josifoski, M., Peyrard, M., and West, R. Grammar-constrained decoding for structured nlp tasks without finetuning. In *Proc. of EMNLP*, 2023b.

Gerganov, G. and et. al. llama.cpp: Port of facebook's llama model in c/c++. URL `https://github.com/guidance-ai/guidance`.

Jiang, A. Q., Sablayrolles, A., Mensch, A., Bamford, C., Chaplot, D. S., Casas, D. d. l., Bressand, F., Lengyel, G., Lample, G., Saulnier, L., et al. Mistral 7b. *ArXiv preprint*, 2023.

Jiang, A. Q., Sablayrolles, A., Roux, A., Mensch, A., Savary, B., Bamford, C., Chaplot, D. S., de Las Casas, D., Hanna, E. B., Bressand, F., Lengyel, G., Bour, G., Lample, G., Lavaud, L. R., Saulnier, L., Lachaux, M., Stock, P., Subramanian, S., Yang, S., Antoniak, S., Scao, T. L., Gervet, T., Lavril, T., Wang, T., Lacroix, T., and Sayed, W. E. Mixtral of experts. *ArXiv preprint*, 2024.

Kudo, T. and Richardson, J. SentencePiece: A simple and language independent subword tokenizer and detokenizer for neural text processing. In *Proc. of EMNLP*, 2018.

Lundberg, S. and Ribeiro, M. T. A. p. The Art of Prompt Design: Prompt Boundaries and Token Healing.

Lundberg, S., Ribeiro, M. T. A. p., and et. al. Guidance-ai/guidance: A guidance language for controlling large language models. URL `https://github.com/guidance-ai/guidance`.

McNaughton, R. and Yamada, H. Regular expressions and state graphs for automata. *IRE Trans. Electron. Comput.*, (1), 1960.

OpenAI. GPT-4 technical report. *ArXiv preprint*, 2023.

Poesia, G., Polozov, A., Le, V., Tiwari, A., Soares, G., Meek, C., and Gulwani, S. Synchromesh: Reliable code generation from pre-trained language models. In *Proc. of ICLR*, 2022.

Scholak, T., Schucher, N., and Bahdanau, D. PICARD: Parsing incrementally for constrained auto-regressive decoding from language models. In *Proc. of EMNLP*, 2021.

Sennrich, R., Haddow, B., and Birch, A. Neural machine translation of rare words with subword units. In *Proc. of ACL*, 2016.

Thompson, K. Regular expression search algorithm. *Commun. ACM*, (6), 1968.

Tjong Kim Sang, E. F. Introduction to the CoNLL-2002 shared task: Language-independent named entity recognition. In *COLING-02: The 6th Conference on Natural Language Learning 2002 (CoNLL-2002)*, 2002.

Touvron, H., Lavril, T., Izacard, G., Martinet, X., Lachaux, M., Lacroix, T., Rozière, B., Goyal, N., Hambro, E., Azhar, F., Rodriguez, A., Joulin, A., Grave, E., and Lample, G. Llama: Open and efficient foundation language models. *ArXiv preprint*, 2023a.

Touvron, H., Martin, L., Stone, K., Albert, P., Almahairi, A., Babaei, Y., Bashlykov, N., Batra, S., Bhargava, P., Bhosale, S., Bikel, D., Blecher, L., Canton-Ferrer, C., Chen, M., Cucurull, G., Esiobu, D., Fernandes, J., Fu, J., Fu, W., Fuller, B., Gao, C., Goswami, V., Goyal, N., Hartshorn, A., Hosseini, S., Hou, R., Inan, H., Kardas, M., Kerkez, V., Khabsa, M., Kloumann, I., Korenev, A., Koura, P. S., Lachaux, M., Lavril, T., Lee, J., Liskovich, D., Lu, Y., Mao, Y., Martinet, X., Mihaylov, T., Mishra, P., Molybog, I., Nie, Y., Poulton, A., Reizenstein, J., Rungta, R., Saladi, K., Schelten, A., Silva, R., Smith, E. M., Subramanian, R., Tan, X. E., Tang, B., Taylor, R., Williams, A., Kuan, J. X., Xu, P., Yan, Z., Zarov, I., Zhang, Y., Fan, A., Kambadur, M., Narang, S., Rodriguez, A., Stojnic, R., Edunov, S., and Scialom, T. Llama 2: Open foundation and fine-tuned chat models. *ArXiv preprint*, 2023b.

Touvron, H., Martin, L., Stone, K., Albert, P., Almahairi, A., Babaei, Y., Bashlykov, N., Batra, S., Bhargava, P., Bhosale, S., et al. Llama 2: Open foundation and fine-tuned chat models. *ArXiv preprint*, 2023c.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. Attention is all you need. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, 2017.

Willard, B. T. and Louf, R. Efficient guided generation for large language models. *ArXiv preprint*, 2023.

Wolf, T., Debut, L., Sanh, V., Chaumond, J., Delangue, C., Moi, A., Cistac, P., Rault, T., Louf, R., Funtowicz, M., et al. Huggingface's transformers: State-of-the-art natural language processing. *ArXiv preprint*, 2019.

A. Whitespace-Flexible GUIDANCE Programs

In our experiments, we differentiate standard GUIDANCE programs based on templates and whitespace-flexible GUIDANCE $^{\text{WS}}$ programs.

To demonstrate, consider the following example of a simple template-based GUIDANCE program:

```
f"""{{
        "id": {gen('id', regex='[1-9][0-9]*')},
        "description": "A nimble fighter",
        "name": "{gen('name', stop='"')}",
        "age": {gen('age', regex='[1-9][0-9]*')},
        "armor": "{select(['leather', 'chainmail', 'plate'])}",
        "weapon": "{select(['sword', 'axe', 'bow'])}",
        "class": "{gen('class', stop='"')}",
        "mantra": "{gen('mantra', stop='"')}",
        "strength": {gen('strength', regex='[1-9][0-9]*')},
        "items": [ "{gen('item', stop='"')}", "{gen('item', stop='"'
                )}", "{gen('item', stop='"')}" ],
}}"""
```

Listing 1. A standard JSON GUIDANCE program.

Here, we provide a fixed high-level template, with respect to whitespace and formatting. Only the values of the fields are generated by the LLM.

In contrast, a whitespace-flexible GUIDANCE $^{\text{WS}}$ program for the same task would look as follows:

```
nl = "\n"
WS = token_limit(zero_or_more(select([' ', nl])), 16)

f"""{{{WS}"id"{WS}:{WS}{gen('id', regex='[1-9][0-9]*')}{WS},{WS}"
        description":{WS}"A nimble fighter"{WS},{WS}"name"{WS}:{WS}"
        {gen('name', stop='"')}"{WS},{WS}"age"{WS}:{WS}{gen('age',
        regex='[1-9][0-9]*')}{WS},{WS}"armor"{WS}:{WS}"{select(['
        leather', 'chainmail', 'plate'])}"{WS},{WS}"weapon"{WS}:{WS}
        "{select(['sword', 'axe', 'bow'])}"{WS},{WS}"class":{WS}"{
        gen('class', stop='"')}"{WS},{WS}"mantra"{WS}:{WS}"{gen('
        mantra', stop='"')}"{WS},{WS}"strength"{WS}:{WS}{gen('
        strength', regex='[1-9][0-9]*')}{WS},{WS}"items":{WS}[{WS}"{
        gen('item', stop='"')}"{WS},{WS}"{gen('item', stop='"')}"{WS
        },{WS}"{gen('item', stop='"')}"{WS}]{WS},{WS}}}"""
```

Listing 2. A whitespace-flexible JSON GUIDANCE program.

As shown in the snippet, all explicit templated whitespace is replaced by a `{WS}` token, using the `zero_or_more` operator. Using this approach, all whitespace is now also generated by the LLM, allowing for more flexible formatting of the output, and less explicit constraints on the LLM.

Our experiments in §4 demonstrate that the whitespace-flexible formulation leads to higher task accuracy but also significantly higher inference time. This is because the program leaves more freedom to the LLM on how to concretely generate the output. At the same time however, inference becomes less efficient, as the LLM now also has to generate all whitespace tokens explicitly and the GUIDANCE runtime cannot skip over as many tokens as before.

Algorithm 3 Model-Based Retokenization

```
Input: LLM $f$, Prompt $x$, Target Text $s$
Output: $f$-preferred tokenization of $s$
1: $o \leftarrow []$
2: while $s \neq \emptyset$ do
3:     $\boldsymbol{v} \leftarrow f(x + o)$                       // compute logits
4:     $t \leftarrow \arg\max \{\boldsymbol{v}[t] \mid t \in \mathcal{V} \wedge t$ prefix of $s\}$
5:     $o$.append$(t)$
6:     $s \leftarrow s$[lol:]                 // remove prefix $t$ from $s$
7: end while
8: return $o$
```

B. Model-Based Retokenization

To demonstrate differences between template-based and unconstrained generation, we consider the task of *naturalizing* a given text under a model-preferred tokenization. This is the process of converting text to the tokenization a model would have chosen to represent the same text during generation, when previously conditioned on some prompt. More specifically, given a target text s, a tokenized prompt x, and a model f with vocabulary $\mathcal{V}$, we re-encode t using tokens from $\mathcal{V}$, such that we greedily maximize the sequence likelihood assigned by f.

We refer to this process as *retokenization*. We provide the procedure for this in Algorithm 3. By greedily choosing the highest likelihood token that aligns with the target text, we obtain a tokenization of s, that is consistent with the model's preference, when forced to generate s from x. This corresponds to applying $\arg\max$ decoding to f, where the token distribution of f is always masked such that it produces the target text s. Put differently, if a model was to generate s when conditioned on x, it would have produced the token sequence o under $\arg\max$ decoding.

While retokenization allows us to recover the model-preferred tokenization of a given text, template-based constrained generation methods cannot benefit from this, as its computational overhead is equivalent to the cost of generating all templated tokens from scratch, thereby negating the benefits of using a template-based approach in the first place.

C. Grammars And Prompts

Below we include the grammars and prompts used for each constraining task from our experiments in Section 4:

```
root   ::= object
value  ::= object | array | string | number |
           ("true" | "false" | "null") ws

object ::=
  "{" ws (
            string ":" ws value
    ("," ws string ":" ws value)*
  )? "}" ws
```

```

array  ::=
  "[" ws (
            value
    ("," ws value)*
  )? "]" ws

string ::=
    "\"" (
    [^"\\] |
    "\ \" (["\ \/bfnrt] |
    "u" [0-9a-fA-F]
    [0-9a-fA-F]
    [0-9a-fA-F]
    [0-9a-fA-F]) # escapes
  )* "\"" ws

number ::= ("-"? ([0-9] |
             [1-9] [0-9]*))
           ("." [0-9]+)? ([eE] [-+]? [0-9]+)? ws

ws ::= ([ \ \ t\ \ n] ws)?

# Prompts used for generation
"A JSON file describing a person:"
"A JSON file of a person John Smith:"
"A JSON file of a person John Smith with friends"
"JSON of a person Jane Doe with friends"
"A JSON person:"
```

Listing 3. Basic JSON Grammar

```
root   ::= object
value  ::= object | array | string | number | ("true" | "false" |
      "null") ws

object ::=
    ws "{" ws (
        "\"thoughts\"" ":" ws "[" ws thought (ws "," ws thought)*
              "]" ws "," ws
        "\"answer\"" ":" ws number ws
    ) "}" ws

thought ::=
    "{" ws (
        "\"step\"" ":" ws string "," ws
        "\"calculation\"" ":" ws string "," ws
        "\"result\"" ":" ws number
    ) "}" ws

array  ::=
    "[" ws (
            value
    ("," ws value)*
    )? "]" ws

string ::=
    "\"" (
    [^"\\] |
    "\\" (["\\/bfnrt] | "u" [0-9a-fA-F] [0-9a-fA-F] [0-9a-fA-F]
          [0-9a-fA-F]) # escapes
    )* "\"" ws

number ::= ("-"? ([0-9] | [1-9] [0-9]*)) ("." [0-9]+)? ([eE] [-+]?
       [0-9]+)? ws

# Optional space: by convention, applied in this grammar after
      literal chars when allowed
ws ::= ([ \t\n] ws)?

# Prompts used for generation
We use 5-shot prompts for questions from GSM8K's test split as
      prompt (cf. Task Accuracy Experiments).
```

Listing 4. Guided Math Reasoning Grammar (for GSM8K)

```
root ::= (declaration)*

declaration ::= dataType identifier ws "(" ws parameter? ws ")" ws
       "{" ws statement* "}"

dataType  ::= "int" ws | "float" ws | "char" ws
identifier ::= [a-zA-Z_] [a-zA-Z_0-9]*

parameter ::= dataType identifier

statement ::=
    ( dataType identifier ws "=" ws expression ";" ws ) |
    ( ( dataType identifier ws "[" ws expression ws "]" ws ( "="
          ws expression )? ";" ws ) ) |
    ( identifier ws "=" ws expression ";" ws ) |
    ( identifier ws "(" argList? ")" ";" ws) |
    ( "return" ws expression ";" ws ) |
    ( "while" "(" condition ")" ws "{" statement* "}" ) |
    ( "for" "(" forInit ";" ws condition ";" ws forUpdate ")" "{"
          statement* "}" ws ) |
    ( "if" "(" condition ")" "{" statement* "}" ("else" "{"
          statement* "}")? ws ) |
    ( singleLineComment ws ) |
    ( multiLineComment ws )

forInit ::= dataType identifier ws "=" ws expression | identifier
      ws "=" ws expression
forUpdate ::= identifier ws "=" ws expression

condition ::= expression relationOperator expression
relationOperator ::= ("<=" | "<" | "==" | "!=" | ">=" | ">")

expression ::= term (("+" | "-") term)*
term ::= factor(("*" | "/") factor)*

string ::=
    "\"" (
    [^"\\] |
    "\\" (["\\/bfnrt] | "u" [0-9a-fA-F] [0-9a-fA-F] [0-9a-fA-F]
          [0-9a-fA-F]) # escapes
    )* "\"" ws

factor ::= identifier | number | unaryTerm | funcCall |
      parenExpression | subscript | string
unaryTerm ::= "-" factor
funcCall ::= identifier "(" argList? ")"
parenExpression ::= "(" ws expression ws ")"
subscript ::= identifier "[" ws expression ws "]"

argList ::= expression ("," ws expression)*

number ::= [0-9]+

singleLineComment ::= "//" [^\n]* "\n"
multiLineComment ::= "/*" ( [^*] | ("*" [^/]) )* "*/"

ws ::= ([ \t\n]*)

# prompts used for generation
"A C program that prints \"Hello, world!\":\n```c\n"
"A C main function that iterates over an array of integers and
      prints each one:\n```c\n"
"A C program that prints the sum of two integers:\n```c\n"
"The following is a program that finds the sum of two integers in
      C:\n```c\n"
"A C program that fills an array with the numbers 0 to 9 and
      prints them:\n```c\n"
"A C implementation of a simple bubble sort:\n```c\n"
```

Listing 5. Simple C Program Grammar and Prompts

```
root ::= person

person ::= ( "<person>" ( ws personattributes ) "</person>" )
personattributes ::= nameattribute ageattribute jobattribute
      friends?

nameattribute ::= "<name>" NAME "</name>" ws
ageattribute ::= "<age>" NUMBER "</age>" ws
jobattribute ::= "<job>" ws jobinfo "</job>" ws
friends ::= "<friends>" ws person+ ws "</friends>" ws

```

```
jobinfo ::= jobtitle jobsalary
jobtitle ::= "<title>" NAME "</title>" ws
jobsalary ::= "<salary>" NUMBER "</salary>" ws

NAME ::= ( [^<] )+
NUMBER ::= ( [^<] )+

# Optional space: by convention, applied in this grammar after
      literal chars when allowed
ws ::= ([ \t\n] ws)?

# prompts used for generation
"An XML file describing a person:"
"An XML file of a person John Smith:"
"An XML file of a person John Smith with friends"
"XML of a person Jane Doe with friends"
"An XML person:"
```

Listing 6. XML (with schema) Grammar and Prompts

```
start: dict

dict: "{" content "}"

content: id_pair "," description_pair "," name_pair "," age_pair "
      ," armor_pair "," weapon_pair "," class_pair "," mantra_pair
       "," strength_pair "," items_pair

id_pair: "\"id\"" ":" NUMBER
description_pair: "\"description\"" ":" "\"A nimble fighter\""
name_pair: "\"name\"" ":" STRING
age_pair: "\"age\"" ":" NUMBER
armor_pair: "\"armor\"" ":" (("\"leather\"") | ("\"chainmail\"") |
       ("\"plate\""))
weapon_pair: "\"weapon\"" ":" (("\"sword\"") | ("\"axe\"") | ("\"
      bow\""))
class_pair: "\"class\"" ":" STRING
mantra_pair: "\"mantra\"" ":" STRING
strength_pair: "\"strength\"" ":" NUMBER
items_pair: "\"items\"" ":" "[" item "," item "," item "]"

item: STRING

STRING: /"[^\n\r"]+"/
NUMBER: /[0-9]+/

WS: /[ \t\n]+/
%ignore WS

# prompts used for generation
"The following is a character profile for an RPG game in JSON
      format.\n```json\n",
"A character profile for an RPG game:\n```json\n",
"A character profile for an RPG game in JSON format:\n```json\n",
"A character that is a level 5 human fighter with 10 strength, 10
      dexterity, 10 constitution, 10 intelligence, 10 wisdom, and
      10 charisma:\n```json\n",
"JSON specifying a character that is a level 5 dwarf fighter from
      a game:\n```json\n"
```

Listing 7. Fixed Template Grammar and Prompts

D. Structured Reasoning Outputs

In our experiments, we evaluate task accuracy on GSM8K and CoNLL2003. For these tasks, prompted and constrained model output looks as follows:

```
{
    "thoughts": [
        {
            "step": "Find the distance between the first and
                  second stops",
            "calculation": "60 - 20 - 15",
            "result": 25
        },
        {
            "step": "Find the distance between the first and
                  second stops",
            "calculation": "25 + 15",
            "result": 40
        }
    ],
    "answer": 40
}
```

Listing 8. Structured Reasoning Output for GSM8K

```
{
    "tokens": [
        {
            "token": "Nadim",
            "tag": "B-PER"
        },
        {
            "token": "Ladki",
            "tag": "I-PER"
        }
    ]
}
```

Listing 9. Structured Reasoning Output for CoNLL2003

In practice, such outputs greatly facilitate downstream processing of LLM outputs, as they are already in a structured format and can be easily parsed.

Few-Shot Demonstrations For few-shot demonstrations, we alternate between questions and answers using a simple `Q: ... \n A: ... \n ...` format.