Constrained Decoding of Diffusion LLMs with Context-Free Grammars

Niels Mündler, Jasper Dekoninck, Martin Vechev
Department of Computer Science
ETH Zurich, Switzerland
{niels.muendler,jasper.dekoninck,martin.vechev}@inf.ethz.ch

https://constrained-diffusion.ai
https://github.com/eth-sri/constrained-diffusion

Abstract

Large language models (LLMs) have shown promising performance across diverse domains. Many practical applications of LLMs, such as code completion and structured data extraction, require adherence to syntactic constraints specified by a formal language. Yet, due to their probabilistic nature, LLM output is not guaranteed to adhere to such formal languages. To address this, prior work has proposed constrained decoding to restrict LLM generation to particular formal languages. However, existing works are not applicable to the emerging paradigm of diffusion LLMs, as this requires supporting token generation in arbitrary order instead of the traditional left-to-right order. In this paper, we address this challenge and present the first constrained decoding method for diffusion models, one that can handle formal languages captured by context-free grammars. Our method relies on solving a newly defined additive infilling problem, which asks whether a partial output with holes can be completed to a valid word in the target language. We reduce this problem to deciding whether the intersection of the target language and a particular regular language is empty, and present an efficient decision algorithm for context-free languages. Empirical results on various applications, such as C++ code infilling and structured data extraction in JSON, demonstrate that our method achieves near-perfect syntactic correctness while consistently preserving or improving functional correctness. Importantly, our efficiency optimizations ensure that the computational overhead remains practical.

1 Introduction

Large language models (LLMs) have recently achieved promising performance across a wide range of tasks [21, 40]. Due to their capabilities in code synthesis, they achieve impressive scores on diverse code benchmarks [10, 24, 27, 51] and are integrated into developer workflows as programming copilots [18, 49]. Further, they are used for processing information into machine-readable formats in various domains [19, 45, 46]. Despite these successes, LLMs are inherently probabilistic and offer no guarantees about syntactic validity of generated output, providing an inherent limitation for LLM users.

Constrained decoding A promising approach that mitigates this limitation is constrained decoding [7, 34, 43, 50]. This technique leverages the formal grammar of a target language to guide the generation process, ensuring that the output remains within the language's bounds. Constrained decoding leverages parsing and validation of the generated output in lockstep with the incremental generation process, allowing the model to avoid invalid continuations without restarting inference. It has been widely adopted in practice, with commercial providers offering the option to restrict output to JSON or context-free grammars [2, 41].

39th Conference on Neural Information Processing Systems (NeurIPS 2025).

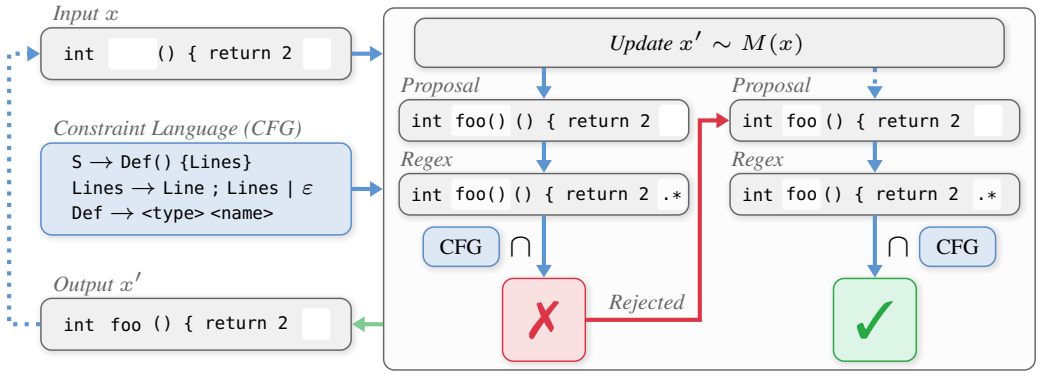


Figure 1: An overview of our approach. In each step, the input consists of a partial text x with arbitrarily many infilling regions and a context-free grammar (CFG) specifying formal constraints. During decoding, we sample an updated input x' from M, obtained, e.g., by inserting a token in one of the regions in x. Our method then intersects the CFG with the regular language of all possible completions of x'. If the intersection is empty, the update is rejected and a new x' is sampled. Otherwise, it is accepted and the decoding continues from x'. In the example, the invalid update inserting `"foo()"` is rejected and `"foo"` is accepted instead.

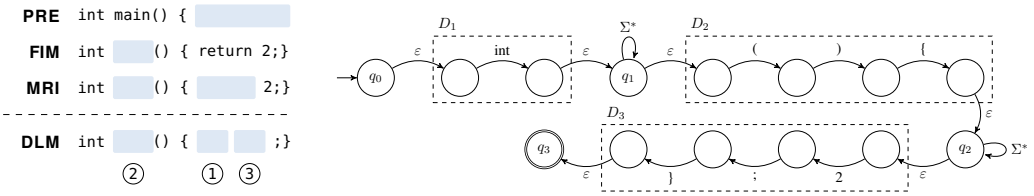
Current limitations of constrained decoding Constrained decoding is usually applied to context-free grammars (CFGs), which capture the syntax of common programming languages and popular data formats, like C++ and JSON [11, 28]. In this context, they can only be applied to left-to-right prefix completion, a common LLM generation setting. However, this setting does not capture more advanced use cases with LLMs, such as diffusion LLMs. While Melcer et al. [34] extend constrained decoding to completions between a fixed prefix and suffix, and Suresh et al. [48] constrain diffusion LLMs to regular languages, no prior work supports diffusion LLM constraining with CFGs.

This work: Constrained decoding for MRI and DLMs In this work, we present a generalized method for constrained decoding of diffusion LLMs (DLM), which also naturally subsumes the previously unaddressed setting of multi-region infilling (MRI). We first generalize the formal framework of constrained decoding to support unordered updates of a partial output with arbitrarily many infilling regions, capturing both MRI and DLM. The decoding process is illustrated in Fig. 1. A model iteratively updates, e.g., inserting a token in a specific location. We verify that the updated output is valid by intersecting the target language's CFG with the language of all possible partial output completions. This intersection is non-empty if and only if a valid completion exists.

A key challenge in this approach is efficiently determining the intersection's emptiness. To this end, we first show that the set of possible completions is described by a regular language, allowing us to describe the intersection using standard formal language operations. We then drastically reduce the worst case cubic cost of checking the emptiness of the resulting intersection language using specialized methods for grammar size reduction and search optimizations, including a custom normal form and an implicit search that avoids generating the entire language.

Experimentally confirmed consistent improvements Our experiments demonstrate a substantial improvement in the reliability of formal language adherence across all evaluated settings. Specifically, the algorithm guarantees valid completions in all settings, up to sampling timeouts. Additionally, it improves functional correctness by up to 7%. Importantly, our approach incurs no initial latency and only modest runtime overhead on tested models, with inference time less than doubling on average, enabling practical usage even for complex constraining grammars.

Key contributions Our three key contributions are: (i) a generalized formal constrained decoding framework for the MRI and DLM settings, (ii) a novel constrained decoding algorithm for these settings, and (iii) an extensive evaluation of our method using state-of-the-art open-weight infilling and diffusion LLMs, demonstrating consistent improvements in syntactic and functional correctness on C++ code generation, JSON schema extraction, and chemical molecule description.



(a) Generation paradigms

(b) NFA accepting all possible completions of the MRI example.

Figure 2: We consider three left-to-right (PRE, FIM, MRI) and one out-of-order (DLM) generation paradigms (a). The NFA in (b) describes the language of all additive completions for the MRI task.

2 Background

We outline the necessary background relevant to this work, including generation paradigms with LLMs, constrained decoding, and the relevant properties of regular and context-free languages.

2.1 LLM Generation Paradigms

We focus on four generation settings with LLMs illustrated in Fig. 2a. The first three approaches are commonly used with autoregressive models and generate outputs left-to-right.

PRE, FIM and MRI The first approach, Prefix generation (PRE) completes a fixed prefix, and is commonly used for synthesizing text or code from scratch. Second, Fill-In-the-Middle (FIM) completes text between a given prefix and suffix, and is widely used in code completion assistants [18, 25]. Third, Multi-Region Infilling (MRI) generalizes FIM by allowing prefix and suffix constraints as well as fixed segments in between, with the model infilling the gaps. This enables more flexible editing, useful for repository-level code modifications [52].

Generation with DLMs Diffusion Language Models (DLMs) [37, 57] iteratively insert tokens into an initially empty or partially filled sequence $(x_1, x_2, \ldots, x_n)$ where each x_i is either a token from the vocabulary V or a mask $\perp$. At each step, the model predicts one or more indices k of a masked token, i.e., $x_k = \perp$, and a token $t \in V$ to produce the updated sequence $(x_1, \ldots, x_{k-1}, t, x_{k+1}, \ldots, x_n)$. This process continues until no masks remain. The number of predicted indices and tokens per forward pass is a hyperparameter that controls a trade-off between increased generation speed and quality [37]. In the example in Fig. 2a, the model would generate one index and token at a time, first producing the return keyword ①, then the function name ②, and finally the return value ③.

Constrained generation Constrained generation restricts the model to produce outputs that conform to predefined syntactic or structural rules, ensuring syntactically valid code or adherence to structural patterns [43]. Formally, the model must generate an output $w \in L$, where L is a formal language defining admissible outputs for the given task. Constrained decoding is typically implemented by restricting the model's output space at each step, either by masking invalid tokens [50] or by sampling and rejecting invalid outputs [34].

2.2 Regular and Context-Free Languages

We briefly outline the properties and notation of regular and context-free languages that are relevant to our method. We provide a more detailed introduction in App. A.

Regular Languages A regular language is a set of strings that can be described by a deterministic finite automaton (DFA). A DFA is defined as a tuple $(Q, \Sigma, \delta, q_0, F)$, where: (1) Q is a finite set of states, (2) Σ is a finite alphabet of symbols, (3) $\delta : Q \times \Sigma \to Q$ is a transition function that maps a state and an input symbol to the next state, (4) $q_0 \in Q$ is the initial state, and (5) $F \subseteq Q$ is the set of accepting states. The language of a DFA consists of those strings that transition the automaton from the initial to an accepting state through the transition function. Non-deterministic finite automata (NFA) additionally allow multiple next states for the same state and symbol and traversing ε-transitions without consuming a symbol. An example is depicted in Fig. 2b. Every NFA is equivalent to some DFA.

Context-Free Languages Context-free languages (CFLs) are a superset of regular languages, including languages that enforce recursive structures, such as balanced parentheses or nested control statements. They can be described by context-free grammars (CFGs). A CFG is a tuple (V, Σ, P, S), where: (1) V is a finite set of nonterminals, (2) Σ is a finite set of terminals (with $V \cap \Sigma = \varnothing$), (3) P is a set of productions $A \to \alpha$, with $A \in V$ and $\alpha \in (V \cup \Sigma)^*$, and (4) $S \in V$ is the start symbol. The language is defined as all strings generated by the following procedure: Starting with S, apply a rule $A \to \alpha$ from P to replace nonterminal A with α, until the result contains only terminals.

3 Constrained Decoding for Infilling and Diffusion

In this section, we first define the decision problem that enables MRI and DLM generation settings, and then introduce our algorithm for efficiently deciding the problem. We then provide adapted constrained decoding algorithms for MRI and DLM. Finally, we show how to apply the algorithm to LLMs, where additional challenges arise from the need to handle tokens instead of terminals.

3.1 The Constrained Infilling Problem

Constrained decoding with infilling First, let us define a partial output $\mathbf{x}$ as a sequence of strings $x_i \in \Sigma^*$ interleaved with infilling regions $\square \notin \Sigma$, i.e., $\mathbf{x} = x_1 \square x_2 \ldots \square x_n$. In constrained decoding, illustrated in Algorithm 1, we complete $\mathbf{x}$ using model M and target language L. We iteratively sample an updated partial output $\mathbf{x}'$ from M (Lines 1 and 2). All updated outputs are derived via *additive* modifications to $\mathbf{x}$, meaning they either insert a string into infilling regions, e.g., insert b into $a \square c$ resulting in $a \square b \square c$, or remove a region by merging adjacent strings, e.g., converting $a \square b \square c$ to $a \square bc$. We then check whether the updated output can be completed into a valid word in L (Line 3). If not, we reject the update and remove it from the model distribution, preventing the loop from resampling the update (Line 8). However, if the update is completable, we replace $\mathbf{x}$ with $\mathbf{x}'$ (Line 4) and return the output if the update removes the last infilling region (Lines 5 and 6). This is valid since $\mathbf{x}$ is both completable and has no infilling regions, implying $\mathbf{x} \in L$. Because no updates remove parts of the output, and completability is preserved in updates, Algorithm 1 terminates on completable inputs.

Algorithm 1 Constrained decoding

Input: Input $\mathbf{x}$, model M, target language L
Output: Completed output $\mathbf{x} \in L$

```
1 while true do
2     x' ~ M(x)
3     if COMPLETABLE(x', L) then
4         x ← x'
5         if □ ∉ x then
6             return x
7     else
8         reject x'
```

Deciding update validity To enable constraining additive generation, we need an incremental verifier COMPLETABLE to determine whether the regions in a partial output can be filled to produce a valid output in L. We formalize the decision problem solved by COMPLETABLE as follows:

Definition 1 (Constrained infilling problem)**.** *For a language L, partial output $\mathbf{x} = x_1 \square x_2 \ldots \square x_n$ with $x_i \in \Sigma^*$ and $\square$ denoting infilling regions, the constrained infilling problem asks whether there exists a list of $n-1$ words $\mathbf{y} = (y_1, \ldots, y_{n-1})$ such that $w = x_1 \cdot y_1 \cdot x_2 \cdot \ldots \cdot y_{n-1} \cdot x_n$ is in L.*

Thus, with the incremental verifier deciding the constrained decision problem, we have effectively reduced constrained decoding with infilling to the constrained infilling problem.

Applications of the constrained infilling problem We now reduce constrained decoding for MRI and DLM generation to the constrained infilling problem. For MRI, the list of words corresponds to the list of fixed strings x_i, with infilling regions in between. For the DLM setting, we add implicit ε tokens at the beginning and end of the partially filled sequence and then merge all consecutive non-mask tokens to build $\mathbf{x}$. For example, the sequence $(a, \perp, \perp, b, c, \perp)$ becomes $\mathbf{x} = a \square bc \square \varepsilon$. Note that, similar to prior work [7, 50], we slightly overapproximate the space of possible completions in these representations by allowing infillings of arbitrary size. In practice, there might be practical limitations to the number of tokens an LLM could insert. We discuss this in more detail in App. E.

3.2 Deciding the Constrained Infilling Problem Efficiently

Overview We now give a brief overview of how to solve the constrained infilling problem efficiently. The problem is determined by two separate constraints: (1) the structural constraints on the output, described by the context-free language L, and (2) all possible completions of the partial

output $\mathbf{x}$, which form a language $C_{\mathbf{x}}$. For example, L could be the language of syntactically valid C++ programs, and $C_{\mathbf{x}}$ the language of completions of partial program $\mathbf{x}$ =`int□(){□2;}`. The infilling problem is answered positively if and only if the intersection $L_\cap = L \cap C_{\mathbf{x}}$ is not empty, i.e., some infilling of the partial output exists to generate a valid word in L. We will show that $C_{\mathbf{x}}$ is a regular language that we can describe with a simple DFA, and that $L_\cap$ can be described by a context-free grammar, which we can construct from L's grammar and $C_{\mathbf{x}}$'s DFA. The constrained infilling problem is then reduced to checking whether $L_\cap$ is empty, for which we design an efficient algorithm. In the example, a word in the intersection language is `int main() {return 2;}`.

Constructing the regular language The language $C_{\mathbf{x}}$ of all possible completions of $\mathbf{x} = x_1 \square \ldots \square x_n$ contains all words that start with x_1, end with x_n, and contain the strings x_i $(1 \leq i \leq n)$ in the correct order, with arbitrary symbols in between. We prove that $C_{\mathbf{x}}$ is regular by constructing an NFA that accepts $C_{\mathbf{x}}$. We first construct automata D_i, which accept exactly x_i. Then, we concatenate D_i with an additional state q_i that accepts any string in Σ^*, i.e., $\delta(q_i, \sigma) = q_i$ for all $\sigma \in \Sigma$. For the concatenation, we add an ε-edge from the accepting states of D_i to q_i and from q_i to the start state of D_{i+1}. A visualization for the prior example is shown in Fig. 2b. In our algorithm, we construct this NFA for each update. We then transform it into an equivalent DFA and minimize the DFA using standard methods [23].

Constructing the intersection language We leverage the well-established facts that (a) the intersection $L_\cap$ of CFL L and regular language $C_{\mathbf{x}}$ is a CFL, whose grammar can be constructed from L's grammar G and $C_{\mathbf{x}}$'s DFA, and (b) that the emptiness of a CFL can be checked in time polynomial to the size of the grammar [16, 23]. However, following the standard construction, the resulting grammar $G_\cap = (V_\cap, \Sigma, P_\cap, S_\cap)$ will have a cubic size in nonterminals and productions, with $|V_\cap| \in O(|V||Q|^2)$ and $|P_\cap| \in O(|P||Q|^3 + |P||Q|^2|\Sigma|)$ [4, 16]. To ensure practicality, we thus need specialized optimizations.

Efficient normalization The standard intersection algorithms require G to be transformed to Chomsky normal form, which only allows rules of the form $A \to BC$ or $A \to a$, where $A, B, C \in V$ and $a \in \Sigma$ [23]. The resulting grammar may have a quadratic increase in the number of production rules [31]. To avoid this increase, we extend the standard construction to support CFGs in C2F$^{+\varepsilon}$, a normal form that additionally allows productions of the form $A \to \varepsilon$ and $A \to B$. This normal form can be obtained with only a linear increase in production rules [31]. Our adaptations to the standard intersection algorithm and a proof of its correctness are provided in App. B.1. In App. B.2, we describe several further heuristics to reduce the size of the normalized CFG of G.

Avoiding nongenerating nonterminals A naive algorithm to check whether the intersection language $L_\cap$ is empty would explore all nonterminals combinations. However, many of the nonterminals in the intersection grammar are not *generating*, i.e., for such a nonterminal A there does not exist a sequence of production applications $A \to \cdots \to w$ such that $w \in \Sigma^*$ [36]. We therefore extend an efficient bottom-up search by D.W. [12] to C2F$^{+\varepsilon}$ that by construction only expands on generating nonterminals, and decides emptiness in time linear to the number of nonterminals and productions in the language. The algorithm starts with the symbols that generate terminals or empty strings directly, i.e., all A with productions $A \to \sigma$ and $A \to \varepsilon$. It marks these symbols as generating and inserts them into a queue. Next, for each symbol X in the queue, it checks whether some production has X on the right-hand side (i.e., either $A \to XC$, $A \to BX$, or $A \to X$), and optionally checks whether the other symbol (B or C) in the production was previously marked as generating. If so, we mark A as generating as well and add it to the queue. As soon as we mark the start symbol S, we conclude that the language is non-empty, since there must exist a sequence of production applications $S \to \cdots \to w$, which implies $w \in L$.

Searching through the implicit intersection language Finally, we avoid constructing the entire intersection language. Instead, we only construct the parts of the language visited during the search. All symbols in the intersection language have the form ${}^{p}\vec{A}^{q}$ for $p, q \in \Sigma$ and $A \in V$. All production rules in the intersection grammar are directly derived from corresponding rules in the original CFG. Specifically, all rules of the form ${}^{p}\vec{A}^{q} \to \varepsilon$ and ${}^{p}\vec{A}^{q} \to \sigma$ are based on corresponding rules $A \to \varepsilon$ and $A \to \sigma$ in the original grammar without further dependencies, allowing us to iterate over directly generating symbols without constructing the remaining grammar. Further, all other rules are of the

form ${}^{p}\vec{A}^{\,q} \to {}^{p}\vec{B}^{\,r}\,{}^{r}\vec{C}^{\,q}$ and ${}^{p}\vec{A}^{\,q} \to {}^{p}\vec{B}^{\,q}$ based on original productions $A \to BC$ and $A \to B$, for all $p, q, r \in Q$. This enables enumerating all such rules for a given ${}^{p}\vec{B}^{\,r}$, ${}^{r}\vec{C}^{\,q}$ or ${}^{p}\vec{B}^{\,q}$ during the search. We present the corresponding pseudo-code and additional explanations in App. B.3.

Sampling a valid completion from the intersection language The algorithm presented above decides intersection emptiness. We now extend it to return a valid completion from the intersection language. To achieve this, we modify the algorithm to track production rules that were applied when marking symbols as generating. These rules describe a parse tree for some word w in the intersection language. We traverse the terminals at the leaf nodes of this tree from left to right to reconstruct a valid completion in the intersection language. This completion is used after a fixed number of rejected updates from the LLM. Since the algorithm leverages the results from the prior emptiness search, it can be run at no additional cost. We observe that only 0.7% of valid completions do not appear in the first 50 selected LLM updates. Thus, we can significantly speed up the sampling process without losing performance by inserting a randomly sampled valid completion after k rejected updates. In our experiments, we set k to 100 to ensure we do not miss any valid completions.

3.3 Application of Constrained Infilling to LLMs

We now briefly outline how to apply the algorithm from §3.2 to LLMs, which generate arbitrary Unicode text rather than language terminals. Full details are provided in App. C.

Lexing For typical applications of CFGs, a string of Unicode characters u is converted to terminals $x = t_1 \dots t_k$ in a process called *lexing*. First, note that every terminal t corresponds to a regular language R_t over Unicode characters. During lexing, t_1 is obtained by finding the terminal t such that R_t matches a prefix p of u, i.e., $u = p \cdot s$. The lexing process then recurses on $u' = s$ to obtain the remainder of x, continuing until the string is empty. In principle, to apply this procedure to a sequence with infilling regions $s_1 \bot s_2 \bot \dots s_k$, we would lex each consecutive string s_i to obtain $\mathbf{x} = t_1 \square \dots \square t_n$, but several caveats to this procedure need to be addressed.

Handling infilling regions First, it does not accurately handle partial terminals that border infilling regions, since LLM tokens are Unicode strings that may not align with terminals. For example, the partial LLM output `if⊥2` could correspond to both `<if>□<int>` and `<identifier>`. The ambiguity stems from the possibility of filling the gap with, e.g., `_valid`, forming identifier `if_valid2`.

To address this, we treat the text around an infilling region as possibly belonging to an incomplete terminal. Specifically, in the lexing process, we additionally look for terminals t such that the current output u is a prefix of a word in R_t right before an infilling region or a suffix right after a region. Further, we include terminals t that could span across one or more infilling regions by determining if prefixes and suffixes can be infilled to form a single word in R_t, as in the example above. We can thus generate all terminal sequences consistent with a partial output. If any such sequence can be completed to a valid program, then the partial output itself admits a valid completion.

Efficiency optimizations The number of possible terminal sequences grows quickly with the number of regions and ambiguities. To improve efficiency, we introduce two optimizations. First, for each x, we directly construct a single NFA for all possible terminal sequences. This allows us to apply the intersection algorithm once rather than for each sequence. Second, we reduce ambiguity by preprocessing terminals: whenever the accepted language of terminal $t_{\leq}$ is contained within terminal $t_{\geq}$, we remove the overlap from $t_{\geq}$ and adapt the CFG to allow $t_{\leq}$ wherever $t_{\geq}$ is allowed.

Sampling a valid completion The sampling method from §3.2 returns a sequence of terminals rather than Unicode characters. To sample a Unicode completion, we first concatenate the regular languages of the terminals in the sampled completion. We then construct a regular language for the current partial LLM output and intersect the two languages. Sampling a random string from this intersection yields a valid completion at the Unicode level.

4 Experimental Evaluation

We evaluate our method across a range of tasks and models, first in the MRI setting, and then in DLM, demonstrating improvements in both syntactic and functional correctness. We provide further experimental details, ablate DLM diffusion steps, and provide a case study in App. D.

4.1 Experimental Setup

Metrics We compute two main metrics to evaluate the effectiveness of our method. First, we determine the percentage of syntactically correct completions (Syntax), which indicates how many of the obtained completions adhere to the specified grammar. We also measure functional correctness (Functional) by either comparing the sample to a golden solution, or by reporting the percentage of solutions that pass all test cases, pass@1, depending on the dataset. All results are averaged over four independent runs with different seeds. We compute confidence intervals at 95%, **boldface** the best method, and underline all methods over which the increase is not significant. The usual size of the confidence interval is 1% to 2%.

Compared methods We run unconstrained LLM sampling, reported as Vanilla (*Van.*) and constrained decoding with our method (*Con.*). This includes sampling random completions when generation aborts, i.e., when the maximum number of 256 tokens is exceeded, or 100 updates are rejected over the decoding process. As an ablation, we report *Con.*$^-$, where these aborted instances are marked as syntactically and functionally invalid.

4.2 Fill-In-the-Middle and Multi-Region-Infilling

Models We compare the performance of five recent open-weight infilling models, including STARCODER2 7B [33], CODEGEMMA 7B [58], and the DEEPSEEK CODER Family [22], covering 7B parameter models from three distinct model families and model sizes from 1.3B to 33B.

Tasks and benchmarks Infilling is commonly used to complete partial code [5, 15]. We therefore evaluate our method on the C++ translation of the HumanEval dataset [10, 59], containing 164 diverse basic coding tasks. Similar to Bavarian et al. [5], we transform the dataset into an infilling task by removing random spans from the human-written reference implementation. We evaluate up to three removed spans, resulting in 1-MRI, being equivalent to FIM, 2-MRI, with two infilling regions, and 3-MRI, with three infilling regions. We design a CFG for the subset of C++ syntax needed to solve the tasks in HumanEval. We report adherence to this CFG as syntactic correctness. Functional correctness is measured by computing the pass@1 score on provided test cases [9].

Syntactic correctness As can be seen in Table 1, our method increases syntactic correctness significantly across all models and numbers of infilling regions. Deriving a valid completion from the intersection language (Con.) recovers a syntactically valid completion in on average 95.8% of instances. Remaining errors are due to timeouts. Constrained decoding without completions (Con.$^-$) increases syntactic correctness more for code with multiple regions. This coincides with models struggling more, achieving an absolute increase of 5.2%, 22.5%, and 31.5% for 1-MRI, 2-MRI, and 3-MRI, respectively. These improvements are consistent across model families and sizes, ranging between 17% and 21% per model.

Functional correctness In the lower half of Table 1, we observe that constraining (Con.) consistently increases functional correctness, on average by 2.8%, and even without randomly sampling valid completions (Con.$^-$), the average increase is 2.4%. This is expected, as syntactically incorrect completions can not be functionally correct and are effectively prevented by our method.

Runtime overhead We compare the time per token between constrained and vanilla decoding. The median runtime overhead of constrained decoding is 125%, where the overhead on the small DEEPSEEK CODER 1.3B is higher (320%) than on the 7B models (100%) and DEEPSEEK CODER 33B (20%). Moreover, median overhead increases with more complex infilling, growing from 67% on 1-MRI to 205% on 3-MRI. Further details for this experiment are provided in App. D.4.

Table 1: Our method consistently improves the percentage of syntactically and functionally correct infillings for varying numbers of regions in MRI under standard decoding (Van.), constrained decoding (Con.$^-$), and completing partially completed outputs (Con.).

		1-MRI			2-MRI			3-MRI		
	Model	Van.	Con.$^-$	Con.	Van.	Con.$^-$	Con.	Van.	Con.$^-$	Con.
Syntax	StarCoder2 7B	88.2	95.0	**98.9**	55.4	77.7	**96.3**	24.5	57.2	**88.3**
	CodeGemma 7B	92.5	97.2	**100.0**	61.5	85.6	**99.0**	29.9	66.4	**96.0**
	DeepSeek C. 1.3B	86.5	91.7	**98.7**	51.5	72.9	**93.1**	22.7	47.7	**83.0**
	DeepSeek C. 6.7B	93.9	98.3	**100.0**	62.0	84.0	**97.3**	32.9	64.9	**94.6**
	DeepSeek C. 33B	93.1	97.6	**100.0**	66.3	86.5	**97.8**	36.4	67.8	**93.5**
Functional	StarCoder2 7B	53.8	56.1	**56.3**	20.5	23.7	**24.2**	7.5	10.3	**11.0**
	CodeGemma 7B	57.1	**59.6**	**59.6**	24.8	29.0	**29.2**	8.7	12.6	**12.8**
	DeepSeek C. 1.3B	46.5	46.4	**47.2**	16.1	18.4	**19.2**	4.9	5.4	**6.5**
	DeepSeek C. 6.7B	64.8	67.1	**67.3**	29.8	32.7	**33.2**	11.9	**13.5**	**13.5**
	DeepSeek C. 33B	69.8	71.2	**71.4**	29.8	34.0	**34.3**	12.6	14.3	**15.4**

4.3 Diffusion Language Models

Models We evaluate our method on the instruction-tuned versions of four state-of-the-art diffusion language models, LLaDA 8B [37], Dream 7B [57], DreamCoder 7B [56] and DiffuCoder 7B [20]. We run all models with 32 steps on 256 tokens and with a temperature of 0.2.

Tasks and benchmarks As DLMs are generic text generation models with many different applications, we design three distinct and diverse tasks:

C++ Based on the dataset used in §4.2, the model should generate the entire function specified in natural language [10, 59].

JSON The model should extract relevant information from natural language input, adhering to a JSON-Schema specification [38].

SMILES The model should write down a chemical molecule described in natural language in the SMILES specification language [53].

For SMILES and JSON we generate synthetic benchmarks using Gemini-2.5-Pro [21] with verification to ensure that the generated samples are correct and solvable, resulting in 167 and 272 instances respectively. More details about the dataset generation procedure can be found in App. D.3.

We implement the syntax of each language as a CFG and use it to enforce and evaluate the syntactic correctness of the generated output. For C++, we measure functional correctness using pass@1 as in §4.2. For JSON and SMILES, correctness is evaluated by comparing to a golden solution.

Syntax errors We observe that our method consistently increases syntactic correctness for all tasks and models, as shown in Table 2. Without sampling valid completions (Con.$^-$), our method increases the percentage of syntactically correct instances by 16.1%, 14.7%, and 26.0% for C++, JSON, and SMILES, respectively. We observe that many models fail to generate syntactically correct output even under constraints, with, for example, only 19.0% correct C++ generations for DreamCoder 7B. However, sampling valid completions (Con.) recovers the failed instances, increasing to 99.2%. In JSON, constrained decoding with completion achieves 100% syntactic correctness. Remaining errors are due to timeouts.

Functional correctness As shown in the lower half of Table 2, the positive effect of constraining on functional correctness is also present for DLM, with an average increase in functional correctness without completions (Con.$^-$) of 1.9%, and a slight additional boost with completions (Con.) to 2.2%. Notably, Dream 7B performance on JSON increases by 6.9%. In the SMILES setting, where models perform very poorly at only 1.5% average correctness, syntactic constraints are not able to improve functional correctness significantly, achieving only a modest average increase of 0.2%.

Table 2: Constrained decoding (Con.$^-$) consistently increases the percentage of syntactically correct completions for DLMs over standard decoding (Van.).

		C++			JSON			SMILES		
	Model	Van.	Con.$^-$	Con.	Van.	Con.$^-$	Con.	Van.	Con.$^-$	Con.
Syntax	DREAM 7B	40.5	58.7	**99.4**	22.4	44.9	**100.0**	67.5	93.7	**99.4**
	DREAMC. 7B	11.0	19.0	**99.2**	73.7	86.6	**100.0**	73.1	94.9	**100.0**
	LLADA 8B	13.3	36.1	**99.7**	77.5	89.0	**100.0**	58.2	91.3	**100.0**
	DIFFUC. 7B	39.2	54.7	**99.7**	64.5	76.3	**100.0**	69.3	92.2	**99.2**
Funct.	DREAM 7B	6.6	8.8	**9.5**	7.4	11.4	**14.3**	0.6	**1.1**	**1.1**
	DREAMC. 7B	3.7	4.9	**5.2**	44.6	**46.7**	**46.7**	**3.4**	**3.4**	**3.4**
	LLADA 8B	3.8	5.0	**5.3**	43.1	**49.5**	**49.5**	0.7	**1.0**	**1.0**
	DIFFUC. 7B	12.5	13.7	**14.8**	34.3	38.0	**38.2**	**1.1**	**1.1**	**1.1**

Runtime overhead We compare the runtime to complete samples in constrained decoding with the vanilla setting. The median completion overhead is only 30%. We observe both speed-ups of up to 19% and slowdowns of up to 190%. Speed-ups occur when the decoding is preemptively aborted. Further details for this experiment are provided in App. D.4.

5 Related Work

Large language models LLMs have gained traction for diverse tasks such as code generation [26] and structured output generation [2, 30, 42]. While the most common approach trains LLMs for PRE generation, many modern code models also support FIM settings [22, 32, 58]. More recently, diffusion language models have been scaled to billion-parameter sizes and demonstrate promising performance on a variety of tasks [20, 37, 56]. Meanwhile, LLMs are prone to errors during generation. For example, they often make mistakes in niche programming languages [17] and fundamentally struggle to model specific types of formal languages [13, 47].

Constrained decoding Constraining code generation to context-free languages has been explored extensively in prior work [6, 7, 43, 54]. Most prior works apply these techniques to the PRE setting [7, 43, 50], with some extensions to FIM and context-sensitive features [34, 35]. Suresh et al. [48] constrain DLMs specifically, but only to regular languages. To our knowledge, constrained decoding with CFGs has not yet been applied to the MRI or DLM paradigms. Additionally, unlike prior work that employs masking to reject invalid tokens [6, 43, 50], our rejection sampling approach incurs no additional latency before starting language inference, significantly reducing friction of switching to a different CFG.

Leveraging language intersections Two similar works leverage the intersection of CFLs and regular languages. First, Fazekas et al. [14] discuss subsequence matching, which asks whether w is a subsequence of any word in language L. This is a special case of our decision problem, with $\mathbf{x} = \varepsilon\square w_1\square \ldots \square w_{|w|}\square\varepsilon$, and can also be solved by using the emptiness check for intersection languages. Their work is not applicable to our setting, as it only handles this special case, does not consider practical performance, and does not consider how to handle lexing.

Second, Nederhof and Satta [36] use intersections of weighted CFGs and DFAs for parsing natural language words, using the intersection language as a succinct representation of admissible parses of lexeme sequences. To reduce the size of these intersections, they also filter non-generating symbols during the intersection construction.

6 Conclusion

We presented the first constrained decoding method for diffusion models that is able to handle context-free languages such as C++ and JSON. We showed how to reduce the problem of valid completion to an infilling decision problem solvable using formal language techniques. Our optimized algorithm demonstrates a consistent and significant increase in syntactic and functional correctness on a variety of benchmarks and models, while still ensuring efficiency at inference time.

Reproducibility Statement

We describe our implementation in detail in §4 and App. D, including details such as hyperparameters and the used compute hardware. Further, all of our experiments were run with fixed seeds and disabled optimizations that would introduce nondeterminism. To ensure complete reproducibility of our results, we publicly release the code implementation of our method, as well as datasets, models, and code used for the evaluation at Redacted Url. We also include the content of this released code as an anonymized artifact for the double-blind review.

Acknowledgements

We thank Niccolò Rigi-Luperti, Thibaud Gloaguen and Jingxuan He for many excited and fruitful discussions about this project.

References

[1] V Aho Alfred, S Lam Monica, and D Ullman Jeffrey. *Compilers principles, techniques & tools*. 2007.

[2] Anthropic. JSON Mode, 2025. URL https://docs.anthropic.com/en/docs/build-with-claude/tool-use#json-mode.

[3] Richard L. Apodaca. SMILES Formal Grammar. Depth-First blog post, 2020. URL https://depth-first.com/articles/2020/04/20/smiles-formal-grammar/.

[4] Y. Bar-Hillel, M. Perles, and E. Shamier. On formal properties of simple phrase structure grammars. *STUF*, 1961. URL https://doi.org/10.1524/stuf.1961.14.14.143.

[5] Mohammad Bavarian, Heewoo Jun, Nikolas Tezak, John Schulman, Christine McLeavey, Jerry Tworek, and Mark Chen. Efficient Training of Language Models to Fill in the Middle. *arXiv preprint*, 2022. URL https://arxiv.org/abs/2207.14255.

[6] Luca Beurer-Kellner, Marc Fischer, and Martin Vechev. Prompting Is Programming: A Query Language for Large Language Models. *PLDI*, 2023. URL https://doi.org/10.1145/3591300.

[7] Luca Beurer-Kellner, Marc Fischer, and Martin Vechev. Guiding LLMs The Right Way: Fast, Non-invasive Constrained Generation. In *ICML*, 2024. URL https://openreview.net/forum?id=pXaEYzrFae.

[8] Blue Obelisk Project and OpenSMILES Community. OpenSMILES specification (HTML version), 2025. URL http://opensmiles.org/opensmiles.html.

[9] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language Models are Few-shot Learners. In *NeurIPS*, 2020. URL https://proceedings.neurips.cc/paper/2020/hash/1457c0d6bfcb4967418bfb8ac142f64a-Abstract.html.

[10] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating Large Language Models Trained on Code. *arXiv Preprint*, 2021. URL https://arxiv.org/abs/2107.03374.

[11] Tiago Cogumbreiro. CS420: Introduction to the theory of computation, lecture 15: Context-free grammars, 2020. URL https://cogumbreiro.github.io/teaching/cs420/s20/lecture15.pdf.

[12] D.W. Solving the emptiness problem for a CFG in Chomsky normal form (linear). Computer Science Stack Exchange, 2018. URL https://cs.stackexchange.com/q/92314.

[13] Javid Ebrahimi, Dhruv Gelda, and Wei Zhang. How Can Self-attention Networks Recognize Dyck-n Languages? In *EMNLP*, 2020. URL https://aclanthology.org/2020.findings-emnlp.384/.

[14] Szilárd Zsolt Fazekas, Tore Koß, Florin Manea, Robert Mercaş, and Timo Specht. Subsequence Matching and Analysis Problems for Formal Languages. In *ISAAC*, 2024. URL https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ISAAC.2024.28.

[15] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen tau Yih, Luke Zettlemoyer, and Mike Lewis. InCoder: A Generative Model for Code Infilling and Synthesis, 2023. URL https://openreview.net/forum?id=hQwb-lbM6EL.

[16] William Gasarch. The Intersection of a CFG and a REG is CFG, 2014. URL https://www.cs.umd.edu/~gasarch/COURSES/452/F14/cfgreg.pdf.

[17] Alessandro Giagnorio, Alberto Martin-Lopez, and Gabriele Bavota. Enhancing Code Generation for Low-resource Languages: No Silver Bullet. *arXiv Preprint*, 2025. URL https://doi.org/10.48550/arXiv.2501.19085.

[18] GitHub. Introducing GitHub Copilot: your AI pair programmer. GitHub Blog, 2025. URL https://github.blog/news-insights/product-news/introducing-github-copilot-ai-pair-programmer/.

[19] Akshay Goel, Almog Gueta, Omry Gilon, Chang Liu, Sofia Erell, Lan Huong Nguyen, Xiaohong Hao, Bolous Jaber, Shashir Reddy, Rupesh Kartha, Jean Steiner, Itay Laish, and Amir Feder. Llms accelerate annotation for medical information extraction. *arXiv Preprint*, 2023. URL https://arxiv.org/abs/2312.02296.

[20] Shansan Gong, Ruixiang Zhang, Huangjie Zheng, Jiatao Gu, Navdeep Jaitly, Lingpeng Kong, and Yizhe Zhang. DiffuCoder: Understanding and Improving Masked Diffusion Models for Code Generation. *arXiv Preprint*, 2025. URL https://arxiv.org/abs/2506.20639.

[21] Google DeepMind. Gemini Pro, 2025. URL https://deepmind.google/technologies/gemini/pro/.

[22] Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y. Wu, Y. K. Li, et al. DeepSeek-Coder: When the Large Language Model Meets Programming - The Rise of Code Intelligence. *arXiv Preprint*, 2024. URL https://doi.org/10.48550/arXiv.2401.14196.

[23] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation*. 1979.

[24] Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. LiveCodeBench: Holistic and Contamination Free Evaluation of Large Language Models for Code. In *ICLR*, 2025. URL https://openreview.net/forum?id=chfJJYC3iL.

[25] JetBrains. Code completion, 2025. URL https://www.jetbrains.com/help/pycharm/auto-completing-code.html.

[26] Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. A Survey on Large Language Models for Code Generation. *arXiv Preprint*, 2024. URL https://doi.org/10.48550/arXiv.2406.00515.

[27] Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R. Narasimhan. SWE-bench: Can Language Models Resolve Real-world Github Issues? In *ICLR*, 2024. URL https://openreview.net/forum?id=VTF8yNQM66.

[28] Donald E. Knuth. On the Translation of Languages from Left to Right. *Inf. Control.*, 1965. URL https://doi.org/10.1016/S0019-9958(65)90426-2.

[29] Greg Landrum, Paolo Tosco, Brian Kelley, Ricardo Rodriguez, David Cosgrove, Riccardo Vianello, sriniker, Peter Gedeck, Gareth Jones, Eisuke Kawashima, Nadine Schneider, Dan Nealschneider, Andrew Dalke, and tadhurst-cdd et al. rdkit/rdkit: Q1 2025 Release. Zenodo, 2025. URL https://doi.org/10.5281/zenodo.16439048.

[30] LangChain Developer Documentation. Structured outputs, 2025. URL https://python.langchain.com/docs/concepts/structured_outputs/.

[31] Martin Lange and Hans Leiß. To CNF or not to CNF? An efficient yet presentable version of the CYK algorithm. *Informatica Didactica*, 2009.

[32] Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, et al. StarCoder 2 and The Stack v2: The Next Generation. *arXiv Preprint*, 2024. URL https://doi.org/10.48550/arXiv.2402.19173.

[33] Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, et al. StarCoder 2 and The Stack v2: The Next Generation. *arXiv Preprint*, 2024. URL https://doi.org/10.48550/arXiv.2402.19173.

[34] Daniel Melcer, Nathan Fulton, Sanjay Krishna Gouda, and Haifeng Qian. Constrained Decoding for Fill-in-the-middle Code Language Models via Efficient Left and Right Quotienting of Context-sensitive Grammars. *arXiv Preprint*, 2024. URL https://arxiv.org/abs/2402.17988.

[35] Niels Mündler, Jingxuan He, Hao Wang, Koushik Sen, Dawn Song, and Martin Vechev. Type-Constrained Code Generation with Language Models. In *PLDI*, 2025. URL https://doi.org/10.1145/3729274.

[36] Mark-Jan Nederhof and Giorgio Satta. Probabilistic Parsing. In *SCI*. 2008. URL https://link.springer.com/chapter/10.1007/978-3-540-78291-9_7.

[37] Shen Nie, Fengqi Zhu, Zebin You, Xiaolu Zhang, Jingyang Ou, Jun Hu, Jun Zhou, Yankai Lin, Ji-Rong Wen, and Chongxuan Li. Large Language Diffusion Models. *arXiv preprint*, 2025. URL https://arxiv.org/abs/2502.09992.

[38] NousResearch. json-mode-eval. Hugging Face Datasets, 2024. URL https://huggingface.co/datasets/NousResearch/json-mode-eval.

[39] Cyrus Omar, Ian Voysey, Ravi Chugh, and Matthew A. Hammer. Live functional programming with typed holes. *POPL*, 2019. URL https://doi.org/10.1145/3290327.

[40] OpenAI. GPT-4 Technical Report. *arXiv Preprint*, 2023. URL https://doi.org/10.48550/arXiv.2303.08774.

[41] OpenAI. Function calling - openai api: Context-free grammars, 2025. URL https://platform.openai.com/docs/guides/function-calling#context-free-grammars. Accessed: 2025-08-12.

[42] OpenAI. Structured Outputs, 2025. URL https://platform.openai.com/docs/guides/structured-outputs.

[43] Gabriel Poesia, Alex Polozov, Vu Le, Ashish Tiwari, Gustavo Soares, Christopher Meek, and Sumit Gulwani. Synchromesh: Reliable Code Generation from Pre-trained Language Models. In *ICLR*, 2022. URL https://openreview.net/forum?id=KmtVD97J43e.

[44] Julien Romero. Pyformlang: An Educational Library for Formal Language Manipulation. In *SIGCSE*, 2021. URL https://doi.org/10.1145/3408877.3432464.

[45] Mara Schilling-Wilhelmi, Martiño Ríos-García, Sherjeel Shabih, María Victoria Gil, Santiago Miret, Christoph T. Koch, José A. Márquez, and Kevin Maik Jablonka. From text to insight: Large language models for materials science data extraction. *arXiv Preprint*, 2024. URL https://arxiv.org/abs/2407.16867.

[46] Lena Schmidt, Kaitlyn Hair, Sergio Graziosi, Fiona Campbell, Claudia Kapp, Alireza Khanteymoori, Dawn Craig, Mark Engelbert, and James Thomas. Exploring the use of a large language model for data extraction in systematic reviews: a rapid feasibility study. *arXiv Preprint*, 2025. URL https://arxiv.org/abs/2405.14445.

[47] Lena Strobl, William Merrill, Gail Weiss, David Chiang, and Dana Angluin. What Formal Languages Can Transformers Express? A Survey. *TACL*, 2024. URL https://doi.org/10.1162/tacl_a_00663.

[48] Tarun Suresh, Debangshu Banerjee, Shubham Ugare, Sasa Misailovic, and Gagandeep Singh. Dingo: Constrained inference for diffusion llms. *arXiv Preprint*, 2025. URL https://arxiv.org/abs/2505.23061.

[49] Tabnine. Tabnine: AI Code Assistant, 2025. URL https://www.tabnine.com/.

[50] Shubham Ugare, Tarun Suresh, Hangoo Kang, Sasa Misailovic, and Gagandeep Singh. SynCode: LLM Generation with Grammar Augmentation. *ArXiv Preprint*, 2024. URL https://arxiv.org/abs/2403.01632.

[51] Mark Vero, Niels Mündler, Victor Chibotaru, Veselin Raychev, Maximilian Baader, Nikola Jovanović, Jingxuan He, and Martin Vechev. BaxBench: Can LLMs generate correct and secure backends? In *ICML*, 2025. URL https://openreview.net/forum?id=il3KRr4H9u.

[52] Jiayi Wei, Greg Durrett, and Isil Dillig. Coeditor: Leveraging repo-level diffs for code auto-editing. In *ICLR*, 2024. URL https://proceedings.iclr.cc/paper_files/paper/2024/file/77c7faab15002432ba1151e8d5cc389a-Paper-Conference.pdf.

[53] David Weininger. SMILES, a chemical language and information system. 1. Introduction to methodology and encoding rules. *JCIM*, 1988. URL https://doi.org/10.1021/ci00057a005.

[54] Brandon T. Willard and Rémi Louf. Efficient Guided Generation for Large Language Models. *arXiv Preprint*, 2023. URL https://doi.org/10.48550/arXiv.2307.09702.

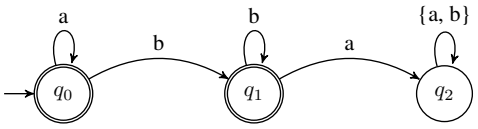
[55] Zirui Wu, Lin Zheng, Zhihui Xie, Jiacheng Ye, Jiahui Gao, Yansong Feng, Zhenguo Li, Victoria W., Guorui Zhou, and Lingpeng Kong. Dreamon: Diffusion language models for code infilling beyond fixed-size canvas, 2025. URL https://hkunlp.github.io/blog/2025/dreamon.

[56] Zhihui Xie, Jiacheng Ye, Lin Zheng, Jiahui Gao, Jingwei Dong, Zirui Wu, Xueliang Zhao, Shansan Gong, Xin Jiang, Zhenguo Li, and Lingpeng Kong. Dream-Coder 7B. HKU NLP Blog, 2025. URL https://hkunlp.github.io/blog/2025/dream-coder.

[57] Jiacheng Ye, Zhihui Xie, Lin Zheng, Jiahui Gao, Zirui Wu, Xin Jiang, Zhenguo Li, and Lingpeng Kong. Dream 7B. HKU NLP Blog, 2025. URL https://hkunlp.github.io/blog/2025/dream.

[58] Heri Zhao, Jeffrey Hui, Joshua Howland, Nam Nguyen, Siqi Zuo, Andrea Hu, Christopher A. Choquette-Choo, Jingyue Shen, Joe Kelley, Kshitij Bansal, et al. CodeGemma: Open Code Models Based on Gemma. *arXiv Preprint*, 2024. URL https://doi.org/10.48550/arXiv.2406.11409.

[59] Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Zihan Wang, Lei Shen, Andi Wang, Yang Li, et al. CodeGeeX: A Pre-Trained Model for Code Generation with Multilingual Benchmarking on HumanEval-X. In *SIGKDD*, 2023. URL https://dl.acm.org/doi/10.1145/3580305.3599790.



(a) A DFA where q_0 is the start state, $\{q_0, q_1, q_2\}$ are the states, and q_0 and q_1 are the accepting states. The arrows represent the transition function δ.

$V = \{S, B\}$ (Nonterminals)
$\Sigma = \{a, b\}$ (Terminals)
$S \to aS \mid bB \mid \varepsilon$
$B \to bB|\varepsilon$

(b) A CFG with start symbol S, terminal alphabet $\Sigma = \{a, b\}$, and nonterminals $V = \{S, B\}$. The production rules are the last two lines.

Figure 3: Two representations of a formal language: a DFA (Fig. 3a) and a CFG (Fig. 3b). Both accept strings that start with a's and end with b's.

A Extended Background on Formal Languages

Formal languages allow to unambiguously specify valid or invalid strings, usually for ensuring machine-readability, i.e., in the case of JSON schemas, or when specifying the syntactic rules of programming languages. Formal languages are, in their most general form, defined as a set of strings over an alphabet Σ. For instance, over the alphabet $\Sigma = \{a, b\}$, one can define the formal language $\{\varepsilon, b, aa, bb, aabb, aaaabbb, \ldots\}$ of strings consisting of any number of a's followed by b's. In this section, we provide a short explanation of two key classes of formal languages: regular and context-free languages.

A.1 Regular Languages

Regular languages are commonly encountered when describing string patterns with regular expressions. For example, the language of a's followed by b's is described by the regular expression `a*b*`, where the star denotes zero or more repetitions. A regular language can alternatively be described through a Deterministic Finite Automaton (DFA) that accepts the language [23]. A DFA is a state machine that processes an input string symbol by symbol, transitioning between states based on a deterministic transition function. Thus, a string gets processed by the DFA by starting in the initial state and following the transitions associated with the current input symbol until the end of the string is reached. A string is accepted if the DFA ends in an accepting state after processing the entire string. Formally, a DFA is defined as a tuple $(Q, \Sigma, \delta, q_0, F)$, where: (1) Q is the finite set of states, (2) Σ is the finite alphabet of symbols, (3) $\delta : Q \times \Sigma \to Q$ is the transition function that maps a state and an input symbol to the next state, (4) $q_0 \in Q$ is the initial state, and (5) $F \subseteq Q$ is the set of accepting states. Fig. 3a depicts the DFA recognizing the previously introduced language of strings with arbitrarily many a's followed by b's. Note that in this example, the transition function δ is defined for every state and symbol combination. Per convention, omitted transitions implicitly transfer to a state like q_2, from which no accepting state can be reached.

In DFAs, the next transition is thus uniquely determined for each combination of state and input symbol. In contrast, nondeterministic finite automata (NFAs) allow multiple transitions for a state and input symbol combination, making it nondeterministic. One often additionally adds the option to transition between states without consuming any input symbols, through so-called ε-transitions. This added flexibility allows for a more concise depiction and simplifies construction, which is why we use them throughout this work. NFAs accept a word if *any* possible transition according to the input symbols leads to an accepting state. Every NFA (including ε-transitions) can be converted to an equivalent DFA using a standard algorithm [23]. The NFAs constructed for partial LLM outputs in our method are usually converted into a DFA of around the same number of states, even though the worst-case equivalent DFA can have exponentially many states.

A.2 Context Free Languages

Context-Free Languages (CFLs) extend regular languages by enabling the expression of recursively nested structures, such as balanced parentheses or properly nested control statements in code. They are described using Context-Free Grammars (CFGs), which consist of production rules that specify how strings in the language can be generated [23]. For most programming languages, the syntactic rules of the language can be adequately captured by a CFG.

CFGs operate with two types of symbols: terminals, which are the actual characters of the language, and nonterminals, which are used to define the language patterns. A CFG is a formal grammar that consists of a finite set of production rules that describe how strings in the language can be generated. Formally, a CFG is a tuple (V, Σ, P, S), where: (1) V is a finite set of nonterminal symbols, (2) Σ is a finite set of terminal symbols (with $V \cap \Sigma = \varnothing$), (3) P is a set of production rules of the form $A \to \alpha$, with $A \in V$ and $\alpha \in (V \cup \Sigma)^*$, and (4) $S \in V$ is the start symbol. To generate a string, one starts with S and applies rules from P until the resulting string contains only terminal symbols. This process defines all valid strings in the language. Fig. 3b shows a CFG that generates strings over $\{a, b\}$ starting with arbitrarily many a's followed by b's, demonstrating that the same language recognized by a DFA can also be described by a CFG. To generate the string `aabb`, one could apply the following sequence of production rules: $S \to aS \to aaS \to aabB \to aabbB \to aabb$.

CFGs are often specified in normal forms, which restrict the grammar to certain types of production rules. The benefit of the resulting language is that it reduces edge cases to handle in productions and simplifies proofs about properties of the language. The most common normal form is the Chomsky normal form, where each production rule is of the form $A \to BC$ or $A \to a$, with $A, B, C \in V$ and $a \in \Sigma$. Many other normal forms exist, such as C2F, which is based on the Chomsky normal form but additionally allows so-called unit production rules of the form $A \to B$ [31]. Languages in Chomsky normal form and C2F can not produce the empty word, as they lack productions that generate the empty string ε [23]. We therefore introduce C2F$^{+\varepsilon}$, which additionally allows production rules of the form $A \to \varepsilon$.

B Details on Efficient Intersection Language Searches

In this section, we first detail generic optimizations to reduce the size of context-free grammars, then provide a detailed proof of the correctness of our intersection language construction, and finally provide some more details on the search algorithm we employ to decide emptiness.

B.1 Construction of the Intersection Language for CFGs in C2F$^{+\varepsilon}$

We now provide the full constructive proof that the intersection of a CFL and a regular language is a CFL, since it is rarely written out in the literature. We have further adapted it for grammars in C2F$^{+\varepsilon}$. It forms the basis of Algorithm 2, the core algorithm of our method.

Lemma 1. *The intersection language $L \cap R$ between a context-free language L and the regular language R is context-free.*

Proof. We give a constructive proof by explicitly building a CFG that generates $L \cap R$. We provide the details omitted in the proof given by Gasarch [16] and extend it to allow grammars in C2F$^{+\varepsilon}$.

Let L_{CFL} be generated by a CFG $G = (V, \Sigma, P, S)$, and let L_R be accepted by a DFA $(Q, \Sigma, \delta, q_0, F)$. We first convert G to C2F$^{+\varepsilon}$. Then, we construct a new CFG $G_\cap$ whose language is exactly $L_\cap = L_{\text{CFL}} \cap L_R$.

The idea is to simulate the CFG G and DFA $(Q, \Sigma, \delta, q_0, F)$ in parallel. Specifically, we define the nonterminals of $G_\cap$ to be of the form ${}^{p}\vec{A}^{q}$, where $A \in V$ is a nonterminal of G, and $p, q \in Q$ are states of the DFA. We then create production rules in such a way that if there exists a sequence of productions such that ${}^{p}\vec{A}^{q} \to \cdots \to w$, then there exists a sequence of productions in G such that $A \to \cdots \to w$ and w takes the DFA from state p to state q. We then add a start symbol $S_\cap$ and productions $S_\cap \to {}^{q_0}\vec{S}^{f}$ for all $f \in F$ to ensure that $L_\cap$ contains exactly the words that can be derived from the start symbol S of G and that also take the DFA from the start state q_0 to an accepting state $f \in F$, i.e., all words that are generated by the grammar and all words that are accepted by the DFA.

The productions of $G_\cap$ are defined as follows (adapting [16], additional rules in green):

1. For each production $A \to \sigma$, for all $p, q \in Q$ where $\delta(p, \sigma) = q$, we add ${}^{p}\vec{A}^{q} \to \sigma$
2. For each production $A \to \varepsilon$, for all $p \in Q$, add ${}^{p}\vec{A}^{p} \to \varepsilon$

3. For each production $A \to BC$, and for all $p, q, r \in Q$, we add ${}^{p}\vec{A}^{\,r} \to {}^{p}\vec{B}^{\,q}{}^{q}\vec{C}^{\,r}$
4. For each production $A \to B$, for all $p, q \in Q$, add ${}^{p}\vec{A}^{\,q} \to {}^{p}\vec{B}^{\,q}$

The intuition behind the additional rules is that if the automaton is in some state q, we can "switch the current symbol" ($A \to B$) or "produce an empty string" ($A \to \varepsilon$) without affecting the state. These productions cover the two additional allowed productions in C2F$^{+\varepsilon}$grammars, which are not present in CNF grammars.

Finally, we add a new start symbol S' with productions $S' \to {}^{q_0}\vec{S}^{\,f}$ for all $f \in F$.

We show that the language generated by the constructed CFG $L_\cap$ is equivalent to the intersection language of the CFL L_{CFL} and regular language L_R, i.e., $L_\cap = L_{\text{CFL}} \cap L_R$. To do so, we first need some additional notations:

- For any $p, q \in Q$ and $A \in V$, $L({}^{p}\vec{A}^{\,q})$ denotes the language generated by the nonterminal ${}^{p}\vec{A}^{\,q}$ in the constructed CFG, i.e., the set of all words that can be derived with ${}^{p}\vec{A}^{\,q}$ as the start symbol. Note that $L_\cap = \bigcup_{f \in F} L({}^{q_0}\vec{S}^{\,f})$.
- For any $A \in V$, $L(A)$ denotes the language generated by the nonterminal A in the original CFG, i.e., the set of all words that can be derived with A as the start symbol. Note that $L_{\text{CFL}} = L(S)$.
- For any $p, q \in Q$, $L(p \to q)$ denotes the language accepted by the DFA with start state p and final state q, i.e., the set of all words that can be accepted by the DFA starting in state p and ending in state q. Note that $L_R = \bigcup_{f \in F} L(q_0 \to f)$.

We will show that for any $p, q \in Q$ and $A \in V$

$$L({}^{p}\vec{A}^{\,q}) = L(A) \cap L(p \to q).$$

This immediately implies that $L_\cap = L_{\text{CFL}} \cap L_R$, as

$$L_\cap = \bigcup_{f \in F} L({}^{q_0}\vec{S}^{\,f}) = \bigcup_{f \in F} (L(S) \cap L(q_0 \to f)) = L(S) \cap \bigcup_{f \in F} L(q_0 \to f) = L_{\text{CFL}} \cap L_R.$$

We prove both inclusions separately.

($\subseteq$) We show that for any $p, q \in Q$ and $A \in V$, $L({}^{p}\vec{A}^{\,q}) \subseteq L(A) \cap L(p \to q)$. Let the generation path of $w \in L({}^{p}\vec{A}^{\,q})$ be defined as the sequence of productions that were used to derive w from ${}^{p}\vec{A}^{\,q}$. Denote

$$L_n({}^{p}\vec{A}^{\,q}) = \{w \in L({}^{p}\vec{A}^{\,q}) \mid \text{the generation path of } w \text{ has length at most } n\}.$$

We show the inclusion by induction over the length of the generation path.

$\mathbf{n = 1}$. We show that $L_1({}^{p}\vec{A}^{\,q}) \subseteq L(A) \cap L(p \to q)$. Since w is a word, the only possible productions that can be used to derive w from ${}^{p}\vec{A}^{\,q}$ are either rule 1 or rule 2.
In the first case, we know $w = \sigma$, $A \to \sigma$ is a production of the original CFG G, and $\delta(p, \sigma) = q$. Hence, $w \in L(A)$ and $w \in L(p \to q)$. In the second case, we have $w = \varepsilon$, $A \to \varepsilon$ is a production of G, and $p = q$. Hence, $w \in L(A)$ and $w \in L(p \to q)$.

$\mathbf{n > 1}$. Suppose that for all $p, q \in Q$ and $A \in V$, $L_{n-1}({}^{p}\vec{A}^{\,q}) \subseteq L(A) \cap L(p \to q)$. Let $w \in L_n({}^{p}\vec{A}^{\,q})$ be a word with a generation path of length $n > 1$. Then the first production rule applied to w cannot be rules 1 and 2, as these would yield a generation path of length one. Hence, the first rule applied must be either of the rules 3 and 4.
In the former case, we know there exist two words $w_1 \in L_{n-1}({}^{p}\vec{B}^{\,r}), w_2 \in L_{n-1}({}^{r}\vec{C}^{\,q})$ such that $w = w_1 \circ w_2$. By induction, we have $w_1 \in L(B) \cap L(p \to r)$ and $w_2 \in L(C) \cap L(r \to q)$. Since $A \to BC$ is a production of G, we have $w \in L(A)$

as well. Furthermore, since w_1 transitions the DFA from p to r and w_2 transitions from r to q, we have $w \in L(p \to q)$. Hence, $w \in L(A) \cap L(p \to q)$.

In the latter case, we have $w \in L_{n-1}({}^{p}\vec{B}^{\,q})$ for some nonterminal $B \in V$ such that $A \to B$ is a production of G. By the induction hypothesis, we have $w \in L(B) \cap L(p \to q)$. Since $A \to B$ is a production of G, we have $w \in L(A)$ as well. Hence, $w \in L(A) \cap L(p \to q)$.

($\supseteq$) We show that $L({}^{p}\vec{A}^{\,q}) \supseteq L(A) \cap L(p \to q)$. Let the generation path of w now be measured with respect to the original CFG, i.e., the sequence of productions that were used to derive w from A. Denote

$$L_n(A) = \{w \in L(A) \mid \text{the generation path of } w \text{ has length at most } n\}.$$

We once again show the inclusion by induction over the length of the generation path.

$\mathbf{n = 1}$. We show that for any $p, q \in Q$ and $A \in V$, $L_1(A) \cap L(p \to q) \subseteq L({}^{p}\vec{A}^{\,q})$. Since w is a word, the only possible productions that can be used to derive w from A directly are $A \to \sigma$ or $A \to \varepsilon$.

In the former case, we have $w = \sigma$, and since a DFA only consumes symbols one-by-one, there must be a corresponding state transition, i.e., $\delta(p, \sigma) = q$. Hence, $w \in L({}^{p}\vec{A}^{\,q})$ by rule 1.

In the latter case, $w = \varepsilon$, which immediately implies that $p = q$ since a DFA does not contain epsilon transitions. Hence, $w \in L({}^{p}\vec{A}^{\,q})$ by rule 2.

$\mathbf{n > 1}$. Suppose that for all $p, q \in Q$ and $A \in V$, $L_{n-1}(A) \cap L(p \to q) \subseteq L({}^{p}\vec{A}^{\,q})$. Let $w \in L_n(A) \cap L(p \to q)$ be a word with a generation path of length $n > 1$. Then the first rule applied cannot be $A \to \sigma$ or $A \to \varepsilon$, as these would yield a generation path of length one. Hence, the first rule applied must be either $A \to BC$ or $A \to B$ for some nonterminals $B, C \in V$.

In the former case, we know there exist two words $w_1 \in L_{n-1}(B), w_2 \in L_{n-1}(C)$ such that $w = w_1 \circ w_2$ and $A \to BC$ is a production in the original CFG. Since $w \in L(p \to q)$, we also know that consuming w transitions the DFA from state p to q. We also know that, starting in q, after consuming w_1, the DFA will arrive at some intermediate state r. Clearly therefore $w_1 \in L(p \to r)$. Moreover, since $w = w_1 \circ w_2$ and $w \in L(p \to q)$, also $w_2 \in L(r \to q)$. By induction, we then have $w_1 \in L_{n-1}(B) \cap L(p \to r) \subseteq L({}^{p}\vec{B}^{\,r})$ and similarly $w_2 \in L({}^{r}\vec{C}^{\,q})$. We know that production ${}^{p}\vec{A}^{\,q} \to {}^{p}\vec{B}^{\,r}\,{}^{r}\vec{C}^{\,q}$ is in the intersection language, due to rule 3 quantifying over all states in Q. Hence, $w \in L({}^{p}\vec{A}^{\,q})$.

In the latter case, we have $w \in L_{n-1}(B)$ and $w \in L(p \to q)$. By the induction hypothesis, we have $w \in L({}^{p}\vec{B}^{\,q})$. Since $A \to B$ is a production of the original CFG, we have $w \in L({}^{p}\vec{A}^{\,q})$ as well by rule 4.

This completes the proof of the lemma.

□

B.2 Grammar Size Optimizations

The size of the grammar used for the intersection generation is of high importance to the overall runtime, as the number of productions in the intersection grammar scales cubically with the number of productions in the original grammar. While the size of the intersection grammar also depends on the size of the intersected DFA, generic and efficient methods to minimize DFAs exist. Meanwhile minimization of CFGs is undecidable [23].

We therefore apply several heuristics to reduce the grammar size:

- Inlinable terminal elimination: Inline the productions of nonterminals that are only used in a single production. In particular, when B is only used in a single production $A \to \alpha B \beta$, with $B \to \gamma$, remove B and its production and inline it into the production of A to create $A \to \alpha\gamma\beta$.

- Shared 2-gram elimination: For the most frequent BC such that there are several rules of the form $A \to \alpha BC\beta$, (with α, β non-empty) introduce $A' \to BC$ and rewrite $A \to \alpha A'\beta$. Repeat until no more such BC with more than one occurrence can be found.
- Left factoring: We eliminate shared prefixes using left factoring [1]. Specifically, if two productions of the same nonterminal $A \to \alpha\beta$ and $A \to \alpha\beta'$ share the prefix α, we can introduce a new symbol A' and replace the productions to eliminate the duplication, concretely introducing $A \to \alpha A'$ and $A' \to \beta$, $A' \to \beta'$.

After applying these heuristics, we convert the resulting CFG to C2F$^{+\varepsilon}$using a standard algorithm, consisting of several transformation steps, such as terminal elimination and binarization [31]. In between each step, we detect and eliminate potentially constructed non-generating symbols.

B.3 Details on the Search Algorithm

We explain in detail how the search algorithm for generating nonterminals in the intersection language works. The corresponding pseudo-code is presented in Algorithm 2 and based on the algorithm presented by D.W. [12]. We leverage the construction rules of the intersection language to conduct the search on the *implicit* intersection grammar, i.e., we only build the parts of the grammar that we need to explore. Nonterminals in the intersection language have the form ${}^{p}\vec{A}^{q}$ for $p, q \in \Sigma$ and $A \in V$. All production rules of the form ${}^{p}\vec{A}^{q} \to \varepsilon$ and ${}^{p}\vec{A}^{q} \to \sigma$ are based on the corresponding productions $A \to \sigma$ (Construction 1) and $A \to \varepsilon$ (Construction 2) in the original grammar. We leverage this insight to perform the initialization of the search, which iterates over all production rules of this format, at the beginning of the algorithm in Lines 2–5. Further, all other productions are of the form ${}^{p}\vec{A}^{q} \to {}^{p}\vec{B}^{q}{}^{q}\vec{C}^{r}$ and ${}^{p}\vec{A}^{q} \to {}^{p}\vec{B}^{q}$, as constructed by Constructions 3 and 4. Importantly, all rules for all combinations of states p, q, r exist. This allows us to enumerate all such rules for a given symbol B or C on the fly, as done in Lines 9–17, without expending unnecessary execution time. For example, in Line 9, we iterate over all production rules in which the nonterminal ${}^{y}\vec{X}^{z}$ occurs. The two states of the DFAs already fixate two of the three states quantified over in Construction 3. Hence, given a production $A \to XC$ in the original grammar, which uses the nonterminal X and additional nonterminal C, we need to iterate over a single additional state variable q to evaluate all corresponding constructed productions ${}^{y}\vec{A}^{q} \to {}^{y}\vec{X}^{z}{}^{z}\vec{C}^{q}$.

C Lexing with LLM Tokens

The approach described in §3.2 operates directly on the formal language alphabet Σ. LLMs produce Unicode text that can be misaligned with Σ. In this section, we describe in more detail how to handle the resulting discrepancies.

Discrepancies between alphabet and LLM tokens For practical purposes, the alphabet Σ of the formal language usually consists of *lexemes*. These represent language components abstractly, i.e., for programming languages, they could be identifiers, literals, operators, and other syntactic elements of the language, such as `if` and `else`. Before parsing a Unicode string, it thus first needs to be converted into a string of lexemes. This process is called *lexing*.

The code generation paradigms MRI and DLM generate code on a Unicode level and thus require lexing before our method can be applied. In addition to the normal lexing process, our approach needs to handle the partial nature of the LLM outputs, taking into account potential partial lexemes and consequently several possible lexing sequences for the same character-level output. In the remainder of this section, we first explain how to convert the partial LLM output to a set of possible lexeme sequences, and then how to apply the constrained infilling algorithm to these lexeme sequences.

Lexemes and lexing Each lexeme is associated with a regular language R where Σ_R is the set of Unicode characters. For example, the `<number>` lexeme is associated with regular expression `\d+`, and the `<identifier>` lexeme with `[a-zA-Z_]\w*`. Lexing is the process of converting a Unicode-level string into a sequence of lexemes, i.e., a sequence of strings that match the regular expressions of the lexemes. We call such a sequence of lexemes a *lexeme sequence*. The lexing algorithm extracts these sequences by iteratively matching the maximum match for all lexemes that match a nonempty string at the beginning of the currently remaining output. Whitespace between lexemes

is commonly stripped. For example, the character-level string `"1234 hello12"` would be lexed into the lexeme sequence `(<number>,<identifier>)`.

Lexing partial outputs For a partial output $\mathbf{x}$ with infilling regions, we extract the represented lexeme sequences for each chunk of continuous text. For instance, the output `"x = 1234□hello12"` would be split into the chunks `"x = 1234"` and `"hello12"`, which would be lexed into the two lexeme sequences `(<identifier>, <=>, <number>)` and `(<identifier>)`. Note that the resulting list of lexeme sequences is a list of words in Σ that can be directly used to construct the regular language for the infilling problem as described in §3.2, for example here forming the infilling problem `<identifier><=><number>□<identifier>`.

Handling lexemes spanning infilling regions However, infilling regions complicate the lexing process. Concretely, we need to handle strings that match lexemes partially on the border of infilling regions.

Concretely, strings before an infilling region may end with a string that matches a prefix of some lexeme. For example the output `"□123"` could be lexed as `(<number>)`. However, the region could be filled with token `"a"`, resulting in the overall lexing `(<identifier>)`. Similarly, strings may match suffixes of lexemes after infilling regions.

Additionally, lexemes may span over an entire infilling region. For example, for the output `"123□789"`, the lexing would yield the lexeme sequences `(<number>) and (<number>)`. However, it is also possible to insert a token `"456"` into the region, such that the lexing of the final character-level text is just a single lexeme sequence `(<number>)`. In particular, for any chunk $\alpha\beta$ ending with a prefix β of a lexeme a and a consecutive chunk $\gamma\eta$ starting with a suffix γ of the same lexeme `<a>`, then a valid corresponding lexeme sequence for the entire chunk sequence would be $(\text{lex}(\alpha), \texttt{<a>}, \text{lex}(\eta))$.

Note that we need to additionally ensure the prefix and suffix of the lexeme are compatible. For instance, for fixed-width lexemes such as `<while>`, we can not insert a token into the sequence `"whi□ile"` to obtain a sequence with only a single lexeme, even though both `"whi"` and `"ile"` are true prefixes and suffixes of the lexeme `while`. We resolve this by storing the exact partially matched parts of the lexeme and deciding the infilling problem with respect to the lexeme's regular language. This effectively generalizes a similar solution to the one proposed by Melcer et al. [34], in which they explicitly store the reached states within each prefix and suffix and ensure reachability between them.

Lexing algorithm We use some helper operations on character-level DFAs for the lexing algorithm. For DFA D, we define the function MATCH, which returns l, the number of characters in the string that the suffix language automaton matches maximally. For example, $\texttt{\textbackslash d+}.\text{MATCH}(\texttt{123}) = 3$ and $\texttt{\textbackslash d+}.\text{MATCH}(\texttt{1hello}) = 1$. The function $\text{PREFIX}(D)$ returns the true prefix language of D, where a *true* prefix is a prefix that can be completed to a full match by appending at least one more character. For example, `123` is a true prefix for `\d+` but not for `\d\d\d`. `12` is a true prefix for both regular expressions. The function $\text{SUFFIX}(D)$ analogously returns the true suffix language of D. Further, we denote as $w_{\leq i}$ the string formed by the first i characters of w and $w_{>i}$ the string formed by all characters after the first i characters in w.

The lexing algorithm applied to each chunk of continuous text in $\mathbf{x}$ is described in Algorithm 3. The main mode of operation is to keep track in S of possible lexings and remainders to be processed, starting with the empty lexing and the entire string to be processed in Line 1. The method then iterates over all these lexings in Line 8, returns them if the remainder is empty (Line 10) or extends them if a non-empty remainder remains (Line 11). Crucially, Lines 2–7 check whether the text starts with the suffix of any lexeme. Additionally, Line 12 checks whether the remainder of the current text is the prefix to some lexeme.

Applying the constrained infilling algorithm Applying the lexing algorithm to every continuous chunk of text in the partial output results in a list of sets of lexeme sequences, or sets of words in our infilling problem. From this list, we obtain a set of lists of words by taking the cross product of all the sets in the list, e.g., for list $(\{(\alpha, \beta)\}, \{(\gamma), (\eta)\})$, we would obtain the two lexeme-level partial outputs $\alpha\beta\square\gamma$ and $\alpha\beta\square\eta$.

Algorithm 2 Deciding intersection emptiness of a CFG and DFA. The CFG is in CNF. $G.\text{ADD}(x)$ inserts x into G and returns true if x was not in G previously.

Input: CFG C, DFA $R = (Q, \Sigma, \delta, q_0, F)$
Output: $L(C) \cap L(R) = \varnothing$

$G \leftarrow \varnothing$
for all productions $A \to \sigma$ **do** ▷ Mark terminal and epsilon productions.
  $G \leftarrow G \cup \{{}^{p}\vec{A}^{q} \mid \delta(p, \sigma) = q\}$
for all productions $A \to \varepsilon$ **do**
  $G \leftarrow G \cup \{{}^{p}\vec{A}^{p} \mid p \in Q\}$
$s \leftarrow G.\text{COPY}()$
while $s \neq \varnothing$ **do** ▷ Explore all remaining productions
  ${}^{y}\vec{X}^{z} \leftarrow s.\text{POP}()$
  for all productions $A \to XC$, all $q \in Q$ **do**
    if ${}^{z}\vec{C}^{q} \in G$ **and** $G.\text{ADD}({}^{y}\vec{A}^{q})$ **then**
      $s.\text{ADD}({}^{y}\vec{A}^{q})$
  for all productions $A \to BX$, all $q \in Q$ **do**
    if ${}^{q}\vec{B}^{y} \in G$ **and** $G.\text{ADD}({}^{q}\vec{A}^{z})$ **then**
      $s.\text{ADD}({}^{q}\vec{A}^{z})$
  for all productions $A \to X$ **do**
    if $G.\text{ADD}({}^{y}\vec{A}^{z})$ **then**
      $s.\text{ADD}({}^{y}\vec{A}^{z})$
return $G \cap \{{}^{q_0}\vec{S}^{f} \mid f \in F\} = \varnothing$ ▷ Whether any start symbol of $L(C) \cap L(R)$ is generating

Algorithm 3 Extracting lexings of partial output

Input: Input string w, Terminals T
Output: Set $\{(x_i, s_i, p_i)\}_{0 \le i \le n}$ of n possible lexeme sequences x_i and optional partial matches to the first (s_i) or last (p_i) lexeme

$S \leftarrow \{(\varepsilon, w, \text{None}, \text{None})\}$
for $t \in T$ **do** ▷ Determine if the string starts with a suffix of any terminal
  if $\text{SUFFIX}(\text{PREFIX}(t)).\text{MATCH}(w) = |w|$ **then** ▷ If the suffix prefix spans the entire word.
    $S.\text{ADD}(t, \varepsilon, w, w)$
  $l \leftarrow \text{SUFFIX}(t).\text{MATCH}(w)$
  if $l > 0$ **then** ▷ If the suffix matches a non-zero prefix of w
    $S.\text{ADD}(t, w_{>l}, w_{\le l}, \text{None})$
while $S \neq \varnothing$ **do**
  $(x, w, s, p) \leftarrow S.\text{POP}()$
  if $w = \varepsilon$ **then yield** (x, s, p)
  for $t \in T$ **do**
    if $\text{PREFIX}(t).\text{MATCH}(w) = |w|$ **then** ▷ If the prefix spans the entire remaining word.
      $S.\text{ADD}(x \circ t, \varepsilon, s, w)$
    $l \leftarrow t.\text{MATCH}(w)$
    if $l > 0$ **then** ▷ If the suffix matches a non-zero prefix of w
      $S.\text{ADD}(x \circ t, w_{>l}, s, \text{None})$

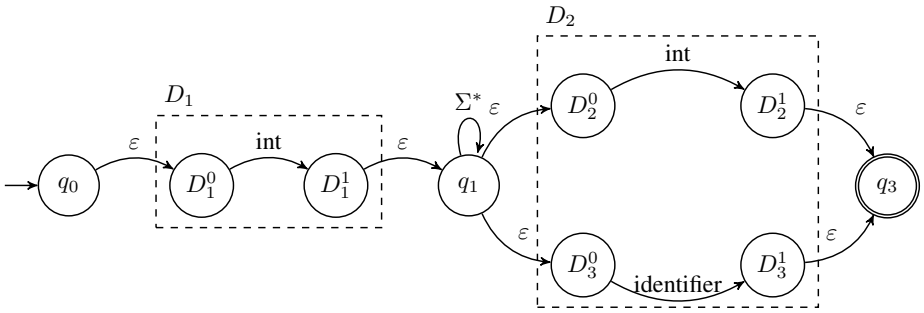


Figure 4: A union automaton in the second half of the DFA for output `"123 □ 789"`, accounts for the possibility to lex the second half as either `<int>` or `<identifier>`. The resulting automaton accepts both valid lexeme sequences `<int><int>` and `<int><identifier>`.

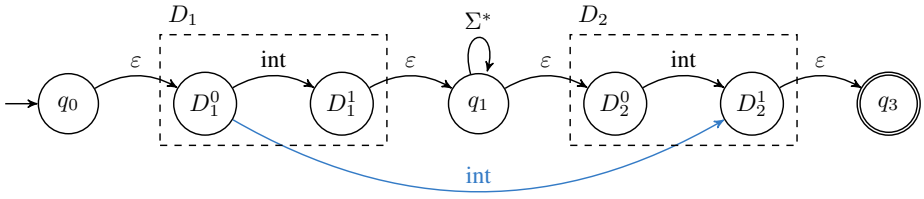


Figure 5: A skip connection, highlighted in blue, in the DFA for output `"123 □ 789"`, accounts for the possibility to lex the input as a single `<int>`. The resulting automaton accepts both valid lexeme sequences for a single int and two ints with intermediate tokens. This construction can be combined with Fig. 4.

If we decide for any of the resulting lexeme-level partial outputs that the intersection language $L_\cap$ is non-empty, then the current character-level partial output is valid, and we can continue generation. If no lexeme sequence results in a non-empty intersection, then we need to reject the current output. Thus, we have to apply the infilling algorithm to each of the word lists derived from the lexing process. In practice, we may derive a large number of lexeme sequences, as different possibilities from text chunks get combined and result in a combinatorial explosion. To further optimize the lexing process, we add two additional optimizations, which we describe in the following paragraphs.

Optimizing subset lexemes We avoid a combinatorial explosion of possible lexeme sequences by automatically removing lexemes where the accepted language is a subset of the accepted language of another lexeme. For example, in SMILES, the string `"5"` could be interpreted as `<digit>` or as `<fifteen>`, which is a special lexeme only allowing numbers from 1 to 15. We resolve this by automatically detecting lexemes α that accept a subset of valid strings of another lexeme β, and a) remove the subset α from the accepted language of lexeme β, and b) allow the lexeme α at any position in the grammar where either the subset token or the full token is allowed, in particular we substitute terminal β with $\alpha \mid \beta$. This optimization reduces the number of extracted sets of possible lexeme sequences for each continuous chunk of text.

We further manually reduce the number of lexemes that overlap and lexemes that are prefixes or suffixes of other lexemes, such as `<++>` and `<+>`, to further optimize performance.

Combining lexeme sequences to a single NFA To avoid explicitly enumerating all possible combinations of lexeme sequences of a string, we directly derive a single, larger NFA that accepts all possible combinations of lexeme sequences at the same time. This NFA is structurally similar to the NFAs of each lexeme sequence, but adds alternative paths for mergeable lexemes.

If a text chunk has two or more admissible lexings, we replace the constructed D_i with an NFA that accepts the union of admissible lexings. For example, the output `"123□789"` must also admit recognizing the second chunk as the suffix of an identifier. Thus, we obtain the two sequence sets {(`<int>`)} and {(`<int>`), (`<identifier>`)}. By generating a single NFA that accepts both sequences (`<int>`) and (`<identifier>`), we can construct a single NFA by applying the concatenation construction to the standard NFA for the first lexeme sequence and the unionized NFA for the second sequence. The resulting NFA is presented in Fig. 4.

Another example is depicted in Fig. 5. Here, for the previously shown output `"123□789"`, the first chunk ends with a prefix of the lexeme `<int>`, and the second chunk starts with a suffix of the same lexeme. In addition to the standard construction for the possible extracted list (`<int>`, `<int>`), we add an `<int>`-edge from the second-to-last state of D_0 to the second state of D_1, resulting in an alternative path that accepts the list (`<int>`). These paths are constructed by maintaining a list of suffixes of the previous D_i when constructing D_{i+1}, and adding the edge if a suffix matches a prefix of the lexing of D_{i+1}.

In contrast to the combinatorial explosion observed when considering all possible combinations of consecutive parsed lexeme sequences, this NFA grows only linearly in the number of sequences. We also observe that the generated corresponding DFA has a similar number of states, confirming that this avoids expensive combination enumeration.

D Experimental Details, Ablations and Case Study

In this section, we provide additional details about the implementation, hyperparameters, datasets, runtime overhead, an ablation on the number of diffusion steps, and a case study.

D.1 Implementation

Overview Our implementation is written in around 7000 lines of Python and 5500 lines of Rust. The main logic, concerning LLM sampling and CFG and DFA construction, is written in Python, with the more computationally expensive formal language operations, such as Algorithms 2 and 3, implemented in Rust, compiled as Python bindings. Several low-level formal language operation implementations are inspired by the educational Python implementations by Romero [44].

Grammars Our C++ grammar covers a comprehensive but not complete subset of C++, with all features used in the canonical solutions of the test set implemented, but advanced features like template functions and user-defined classes are not supported. Moreover, we disallow the insertion of multi-line comments inside function bodies, as this allows the model to generate arbitrary and broken code that is syntactically valid as long as it is finally wrapped in the multi-line comment delimiters. We further restrict models in the MRI setting to not generate additional function signatures and bodies to prevent the generation of additional main functions or test cases.

We preprocess all model outputs by marking word boundaries with special ⟨ and ⟩ tokens that do not appear in the original text and are never generated by the model[1]. For example, the string `int main()` is converted to ⟨`int`⟩ ⟨`main`⟩(). This enables us to check for such word boundaries inside the grammar, i.e., being able to distinguish whether white-space was present between symbols even after it is stripped in the lexing process.

The JSON schema grammars are obtained dynamically based on the JSON Schema for each task. We recursively build up the grammar based on the provided specification. For SMILES, we implement the specification described by Apodaca [3], which is a more precise and efficient variant of prior specifications [8, 53].

D.2 Models and Hyperparameters

All methods were run four times, with seeds 0 to 4, and we report the averaged results in all tables. We report the maximum among Van., Con.$^-$, and Con. decoding with **boldface**. We underline all results where the confidence interval of the improvement over the given method is not positive at 95%. We limit the amount of generated tokens to 256 and time out if the generation does not complete after 300 seconds. We run model inference on NVIDIA RTX A6000 GPUs.

Sampling algorithms and temperature All MRI models were sampled with temperature 1 and greedy decoding. The diffusion models are sampled with a temperature of 0.2. To pick a token from the diffusion models distribution, we use the `entropy` algorithm for the DREAM 7B based models, DREAM 7B, DREAMCODER 7B, and DIFFUCODER 7B, and `low confidence` for the LLADA 8B model, as recommended by the model developers.

[1] In particular, we use the bytes `\x02` and `\x03`

Diffusion steps Diffusion language models can be run with a varying number of diffusion steps, determining how many tokens are sampled from a single model inference [37, 57]. Lower numbers of steps imply more tokens being sampled from each inferred distribution, which in turn is updated less frequently. One of the key benefits of diffusion language models is to exploit this ability, resulting in overall faster decoding. At the same time, higher numbers of steps are usually associated with increased accuracy on the requested task, as the model can adapt its distribution more frequently to newly inserted tokens. When not explicitly stated otherwise, the diffusion models are run with 32 diffusion steps. Our choice of step size 32 represents a trade-off between speed and accuracy.

In each diffusion step, model inference is run once on the current state of the partially filled context window. Afterwards, $\frac{n}{k}$ tokens are sampled from the distribution according to the respective algorithms (`low confidence` or `entropy`) and replace mask tokens in the context window. While unconstrained decoding allows sampling all $\frac{n}{k}$ tokens in parallel, during constrained decoding, we iteratively sample single token-index pairs from this distribution, with rejections leading to masking out the rejected token-index pair and resampling. When a token is accepted, we remove the token's index from the distribution. After $\frac{n}{k}$ tokens have been accepted, we run model inference again.

D.3 Datasets

C++ We leverage the C++ translation of HumanEval in the HumanEval-X dataset [59]. It contains 164 instances of simple programming problems and canonical solutions written by humans. For the MRI tasks, we remove between 1 and 3 randomly sized spans of 5 to 100 characters from these canonical solutions, generating one MRI task per instance in the original dataset. If we end up with insufficient remaining characters after removing the required number of spans, we resample sizes and positions up to 3 times, aborting if we do not find a valid removal. Additionally, we remove 5 human-written solutions that are not valid according to our implementation of the syntax, i.e., because they contain multi-line comments or additional helper functions. This results in three MRI datasets of 159, 156, and 143 samples in 1-MRI, 2-MRI and 3-MRI respectively. An example prompt for an instance from the dataset is presented in Fig. 7. For DLM, we use all 164 tasks of the original dataset and extract the comment before the function as an instruction for the model. An example prompt is shown in Fig. 8.

We check the functional correctness in both settings by checking whether all test cases in the dataset pass with the model-generated solution.

JSON Schema We extend the JSON-Schema dataset by NousResearch [38]. Concretely, the dataset originally contains a unique schema per task. We clean the schemas by disallowing properties other than specified on the top level and repairing instances that accidentally do not require any fields. We then extend the dataset by sampling GEMINI-2.5-PRO for 10 inputs and completions for each schema. We filter these samples in three ways to ensure high quality.

First, we filter the resulting extracted outputs for syntactic validity according to the schema and discard invalid generations. Second, we require GEMINI-2.5-PRO to be able to solve the task, i.e., the model must generate a valid JSON object that passes the schema validation if it is only given the input and the schema. Third, we perform fuzzy matching to deduplicate the resulting samples. This process results in 272 instances. The prompts used for generation and verification are shown in Fig. 11 and Fig. 12. An example prompt for this task is shown in Fig. 9.

We evaluate functional correctness on this dataset by checking for exact equality between a normalized JSON dump of the golden solution and the model-generated solution.

SMILES To create a benchmark for SMILES, we query GEMINI-2.5-PRO to generate pairs of descriptions of molecules and their SMILES notation. Again, we perform three filtering steps to ensure high quality. First, we verify that the generated molecule is valid using the Rdkit library [29]. Second, we ensure the model can generate the correct SMILES string for the molecule if it is only given the description. Third, we filter out duplicates using fuzzy matching. This results in 167 pairs of descriptions and SMILES strings. Prompts for this generation procedure are shown in Fig. 13 and Fig. 14. An example prompt for this task is shown in Fig. 10.

To check the functional correctness of the model-generated molecule, we parse it using Rdkit and check the equivalence to the molecule generated by GEMINI-2.5-PRO in canonical representation.

Table 3: Median overhead per token for different infilling settings in milliseconds and percent increase over unconstrained generation. Larger models with higher inference time experience a lower slowdown due to constraining. More infilling regions also increase constraining overhead.

#Regions	1-MRI	2-MRI	3-MRI
CODEGEMMA 7B	$3.1_{\uparrow 47\%}$	$4.1_{\uparrow 63\%}$	$6.4_{\uparrow 99\%}$
STARCODER2 7B	$3.3_{\uparrow 59\%}$	$5.5_{\uparrow 98\%}$	$9.7_{\uparrow 190\%}$
DEEPSEEK C. 1.3B	$3.6_{\uparrow 158\%}$	$5.8_{\uparrow 245\%}$	$11.8_{\uparrow 557\%}$
DEEPSEEK C. 6.7B	$3.0_{\uparrow 58\%}$	$4.6_{\uparrow 90\%}$	$7.7_{\uparrow 153\%}$
DEEPSEEK C. 33B	$3.1_{\uparrow 13\%}$	$4.3_{\uparrow 19\%}$	$6.5_{\uparrow 28\%}$

Table 4: Median time difference per completion for different diffusion models in seconds, and the overhead over the original completion in percent. When the completion aborts pre-emptively, as no valid completion is sampled from the model, speed-ups are possible.

Model	C++	JSON	SMILES
DREAM 7B	$1.1_{\uparrow 36\%}$	$0.4_{\uparrow 20\%}$	$0.0_{\uparrow 0\%}$
DREAMC. 7B	$7.8_{\uparrow 190\%}$	$0.1_{\uparrow 5\%}$	$0.0_{\uparrow 1\%}$
LLADA 8B	$-1.0_{\downarrow 19\%}$	$0.5_{\uparrow 9\%}$	$0.0_{\uparrow 1\%}$
DIFFUC. 7B	$2.2_{\uparrow 74\%}$	$0.1_{\uparrow 6\%}$	$0.0_{\uparrow 2\%}$

D.4 Additional Experiments

Runtime overhead For all experiments in §4, we measure the runtime of our constraining method and unconstrained decoding. We present a detailed comparison in Tables 3 and 4. We further measure the average number of rejections per sample.

In MRI we compare time per token, as constrained decoding often rejects finalizing the current output, thus making completions longer and finalization times incomparable. The median runtime overhead of constrained decoding is 125%, where the overhead on the small DEEPSEEK CODER 1.3B is higher (320%) than on the 7B model (100%) and DEEPSEEK CODER 33B (20%). This is both due to the lower inference time of smaller models, and due to smaller models making more mistakes, with the average number of rejections increasing from 8.8 per instance on 33B, over 9.7 for 7B to 10.5 in 1.3B. Moreover, more infilling regions are more difficult, leading to more rejections, growing from 4.7 on 1-MRI to 14.1 in 3-MRI. This increases the overhead from 67% to 205% respectively.

For DLM, we compare the total runtime to finish the diffusion decoding process. The average completion overhead is only 30%, but varies strongly between domains. We observe both speed-ups of up to 19%, for LLADA 8B on C++, where many decodings are preemptively aborted, and slowdowns of up to 190%, for DREAMCODER 7B on the same dataset.

Ablation on diffusion steps We evaluate our method on common diffusion step numbers, from 16 to 256, where the lowest setting 16 implies that a single inference step inserts $\frac{256}{16} = 8$ tokens at once, while the highest setting 256 implies that every inference step inserts only a single token.

We present the results of this ablation on DREAM 7B in Table 5 and demonstrate that our method consistently improves syntactic correctness in all settings by on average 14%. Functional correctness on JSON also significantly increases by 1.2%, while the increase in C++ is 0.7% and 0.5% in SMILES. Moreover, the runtime overhead, shown in Table 6, decreases with the number of diffusion steps, from $14\% - 108\%$ down to 9% or even a speed up of 3%.

D.5 Case Study

For a qualitative evaluation, we manually inspect instances where unconstrained decoding fails and our constraining approach successfully corrects errors. We showcase three such examples in Table 7.

Table 5: Percent syntactically and functionally correct generations for DREAM 7B based on varying number of diffusion steps. Our method consistently increases syntactic correctness in all settings, even when model accuracy increases with step sizes.

		C++			JSON			SMILES		
	#Steps	Van.	Con.$^-$	Con.	Van.	Con.$^-$	Con.	Van.	Con.$^-$	Con.
Syntax	16	8.1	20.3	**99.2**	7.3	24.4	**100.0**	41.1	80.5	**99.7**
	32	40.5	58.7	**99.4**	22.4	44.9	**100.0**	67.5	93.7	**99.4**
	64	60.1	74.7	**99.8**	67.4	73.2	**100.0**	79.2	94.9	**100.0**
	128	81.1	90.7	**100.0**	90.2	94.0	**100.0**	80.1	95.8	**100.0**
	256	98.2	98.2	**100.0**	95.2	98.2	**100.0**	80.7	93.4	**100.0**
Functional	16	1.4	2.7	**4.9**	1.5	2.3	**3.4**	0.6	**1.1**	**1.1**
	32	6.6	8.8	**9.5**	7.4	11.4	**14.3**	0.6	**1.1**	**1.1**
	64	21.0	21.8	**22.4**	41.8	42.1	**42.8**	2.4	**3.0**	**3.0**
	128	**24.5**	23.9	24.1	50.7	**51.5**	**51.5**	3.1	**4.0**	**4.0**
	256	34.1	34.1	**34.8**	**54.8**	**54.8**	**54.8**	4.9	**5.2**	**5.2**

Table 6: Time difference per completion for different step sizes on DREAM 7B diffusion, in seconds, and the percentual overhead over the original completion. For larger numbers of diffusion steps, overhead reduces from $14\% - 108\%$ down to 9% or even a speedup of 1%.

#Steps	C++	JSON	SMILES
16	$1.7_{\uparrow 107\%}$	$2.1_{\uparrow 108\%}$	$0.0_{\uparrow 14\%}$
32	$1.1_{\uparrow 36\%}$	$0.4_{\uparrow 20\%}$	0.0_{-}
64	$0.6_{\uparrow 18\%}$	$0.1_{\uparrow 4\%}$	$-0.2_{\downarrow 3\%}$
128	$0.4_{\uparrow 10\%}$	$0.2_{\uparrow 4\%}$	$-0.4_{\downarrow 3\%}$
256	$0.8_{\uparrow 9\%}$	$-0.1_{\downarrow 1\%}$	$0.2_{\uparrow 1\%}$

Preventing use of invalid types In Table 7a, DREAM 7B generates a summary of a financial review for task #30 in our JSON dataset. The schema requires three values of type float. However, the model attempts to generate these values as strings. By applying our constraining method, it can be determined that the strings are misplaced, not constituting one of the required values, and can not match the intended value type. All attempts at placing such strings are thus rejected during generation. Instead, our method forces the model to generate the values without inserting quotes, resulting in a valid and correct result.

Preventing incorrect nesting In Table 7b DREAMCODER 7B generates an invalid SMILES molecule for task #166 by closing more parentheses than it opened. Since the CFG for SMILES correctly handles counting of nesting levels, attempts to generate the closing parentheses are rejected by our method. Instead, the model decides to end the generation.

Preventing inadequate syntax In Table 7c, DEEPSEEK C. 6.7B uses conditions without parentheses in an if-statement when writing a string processing function in task #150 of our C++ dataset. This confusion may stem from the dominance of Python code in training data, which does not require parentheses in if-statements. However, this is invalid according to C++ syntax. Our method can correct this mistake successfully, resulting in a correct infilling.

E Discussion

Remaining syntax errors While our method achieves substantial improvements in syntactic correctness, using only Con.$^-$ still leaves a considerable gap until guaranteeing correctness. We attribute most of this gap to the overapproximation of allowing an arbitrary number of tokens to fill regions in the partial output, as done in prior work [7, 50]. In practice, the LLM is typically limited, i.e., in FIM and MRI it can only generate up to the user-defined maximum number of tokens, and in DLM it can only generate one token per mask $\perp$. Examples of this issue occurring are presented in

```
vector<string> numerical_letter_grade(vector<float> grades){
    vector<string> out={};
    for (int i=0;i<grades.size();i++)
    {
        if (grades[i]>=3.9999) out.push_back("A+");
        if (grades[i]>3.7001 and grades[i]<3.9999) out.push_back("A");
        if (grades[i]>3.3001 and grades[i]<=3.7001) out.push_back("A-");
        if (grades[i]>3.0001 and grades[i]<=3.3001) out.push_back("B+");
        if (grades[i]>2.7001 and grades[i]<=3.0001) out.push_back("B");
        if (grades[i]>2.3001 and grades[i]<=2.7001) out.push_back("B-");
        if (grades[i]>2.0001 and grades[i]<=2.3001) out.push_back("C+");
        if (grades[i]>1.7001 and grades[i]<=2.0001) out.push_back("C");
        if (grades[i]>1.3001 and grades[i]<=1.7001) out.push_back("C-");
        if (grades[i]>1.0001 and grades[i]<=1.3001) out.push_back("D+");
        if (grades[i]>0.7001 and grades[i]<=1.0001) out.push_back("D");
        if □ i]<=3.0001) out.push_back("B");
        if (grades[i]>2.3001 and grades[i]<=2.7001) out.push_back("B-");
        if (grades[i]>2.0001 and grades[i]<=2.3001) out.push_back("C+");
        ...
```

(a) StarCoder2 7B exceeds the token limit in task #81 in 1-MRI.

Vanilla	Constrained^{-}	Constrained
`C6CCCC1)`	`C6CCCC⊥))`	`C6CCCC(c(c)(c))`

(b) LLaDA 8B leaves a single ⊥ for completion in task #153 in SMILES.

Figure 6: Syntax errors may remain when the model has fewer tokens left to complete than would be required to fulfil the syntactic constraints. This can happen both in MRI (a), when the model exceeds the maximum number of generated tokens and in DLM (b), when the model has few mask tokens ⊥ remaining.

Fig. 6a for MRI, where the model exceeds the token limit of 256 tokens as it generates large amounts of unnecessary code, and in Fig. 6b, where the DLM model needs to open several molecule branches in a single remaining token.

One approach to resolve this issue would be to accurately model the remaining number of tokens in our regular language construction. However, we observe in experiments that this significantly increases the size of the regular language, as it consequently needs to keep track of the number of inserted tokens. This drastically increases the size of the intersection language, rendering our current implementation impractical.

Another approach would be to train the model to signal requiring additional tokens. In MRI, this naturally occurs when the model does not generate an end-of-string token. For DLM, a special token could be added that is replaced with two mask tokens after sampling, increasing the size of the affected infilling region. Concurrent work by Wu et al. [55] reports that such capabilities appear to generally improve model performance for code infilling.

Our chosen approach to mitigate the issue is to automatically fill in the output based on the formal language constraints (Con.). However, this solution cannot rely on the model's probability distribution to steer generation. Determining the most effective way to handle this limitation is an important topic for future work.

Leveraging incremental parsing While we take several steps to improve the efficiency of our method, it can still require a significant amount of time to determine the emptiness of the intersection language after each generated token. Future work may leverage the fact that the CFG for the intersection is fixed and the DFA is only updated using small modifications. This may lead to an approach for incrementally computing emptiness checks by reusing the results of the previous intersection computation. Other approaches to leverage the incremental nature of the parsing, similar to

the approaches of Melcer et al. [34], Ugare et al. [50], and Mündler et al. [35] would likely also be able to decrease the worst case and practical overhead of the constraining method.

Context-sensitive language features While our method is designed for context-free languages, an interesting future direction would be extensions to handle more powerful language classes, such as context-sensitive languages. Similar to Melcer et al. [34] and Ugare et al. [50], simple context-sensitive syntactic features can likely be handled by preprocessing through adequate lexers. Beyond syntactic features, prior work suggested leveraging more semantic insights, such as type systems [35], for constructing more powerful constraint systems. Type checkers with typed holes [39] could be leveraged to achieve such systems.

F Prompts

In this section, we detail all prompts used for the respective models and tasks.

MRI Since models used for FIM and MRI tasks are not instruction-fine-tuned, we provide the model with the raw code, including only a comment above the function to guide the model for completions. We use the standard templating suggested for each model to format the prompt for FIM and MRI completion. If MRI is not supported explicitly, we emulate it by inserting `<TODO>` into the remaining infilling regions and repeatedly prompting the model for FIM completion on the first infilling region. In order to prevent the models from generating main methods and tests in the MRI setting, we add a main method at the end of the context that is marked with a TODO comment. An example prompt for the 2-MRI setting is provided in Fig. 7.

DLM The models used for DLM tasks are instruction-fine-tuned, allowing us to specify the completion intent in natural language. We provide a general description of the task in the system prompt and the specific task content as the first user prompt. The assistant response is prefilled with the start of a code fence and, in the case of C++, with the necessary header declarations and function signature to ensure results can be extracted and tests can be executed correctly. The prompts for C++, JSON and SMILES tasks are presented in Fig. 8, 9 and 10 respectively.

Benchmark generation prompts As outlined in App. D, we generate the JSON and SMILES dataset synthetically by prompting GEMINI-2.5-PRO. We provide the used prompts for generation and validation of the generated samples in Figs. 11–14.

Table 7: Three examples demonstrating the impact of constrained decoding on DLM and MRI completion. Left are unconstrained completions (Van.) with problematic tokens highlighted in red, and right constrained completions (Con.$^{-}$) with corrections highlighted in green, adapted for clarity. In (a), our method forces DREAM 7B to generate values of the correct type in a summary of a financial review for task #30 in our JSON dataset. In (b), generated by DREAMCODER 7B, our method prevents closing more parentheses than are opened when generating a SMILES molecule for task #166. In (c), our method forces DEEPSEEK C. 6.7B to add parentheses around a condition in an if-statement when writing a string processing function in task #150 of our C++ dataset.

(a)

Vanilla

```
// summarize my financial review

{
  "capitalGains": "5210.5000",
  "interestIncome": "1340.25",
  "totalReturn": "4.5"
}
```

Constrained

```
// summarize my financial review

{
  "capitalGains": 5210.5000,
  "interestIncome": 1340.25,
  "totalReturn": 4.5
}
```

(b)

Vanilla

```
// generate an allene with axial chirality

C1=CC1=CC(C(OO)O)C(OO)O )C(OO)O
```

Constrained

```
// generate an allene with axial chirality

C1=CC1=CC(C(OO)O)C(OO)O
```

(c)

Vanilla

```
// separate the groups of nested parentheses

if (chr=='(')
{
    level+=1;
    current_paren+=chr;
}
else if chr==')'
{
    ...
```

Constrained

```
// separate the groups of nested parentheses

if (chr=='(')
{
    level+=1;
    current_paren+=chr;
}
else if (chr==')')
{
    ...
```

```
region 0
    /*
    From a given vector of integers, generate a vector of rolling maximum element found
    until given moment in the sequence.
    >>> rolling_max({1, 2, 3, 2, 3, 4, 2})
    {1, 2, 3, 3, 3, 4, 4}
    */
    #include<stdio.h>
    #include<vector>
    using namespace std;
    vector<int> rolling_max(vector<int> numbers){
        vector<int> out;

region 1
        for (int i=0;i<numbers.size

region 2
        return out;
}

int main(){
    // TODO
}
```

Figure 7: Example prompt for the 2-MRI task #1. The intial comment and function signature in blue are derived from the dataset prompt, and the remaining code snippets in green are the remainders of the canonical solution with two randomly removed spans. We append a stub main function to prevent the model from attempting to generate a main function of its own.

````
system
    You are an expert in C++ programming. Solve the given problem by writing solution
     code in C++.
    When answering, insert the solution code in a ```cpp...``` block. Do neither include
     test cases not a main function.

user
    Check if in given vector of numbers, are any two numbers closer to each other than
    given threshold.
    >>> has_close_elements({1.0, 2.0, 3.0}, 0.5)
    false
    >>> has_close_elements({1.0, 2.8, 3.0, 4.0, 5.0, 2.0}, 0.3)
    true

assistant
    ```cpp
    #include<stdio.h>
    #include<vector>
    #include<math.h>
    using namespace std;
    #include<algorithm>
    #include<stdlib.h>
    bool has_close_elements(vector<float> numbers, float threshold){
````

Figure 8: Example prompt for the C++ task #1. The system prompt in black is fixed, whereas the user prompt in blue is extracted from the comment preceding the function and the assistant response is prefilled with a codefence, and in green, headers, and the function signature of each task.
````

```
system
    You are a helpful assistant that answers in JSON. Here is the JSON schema you must
    adhere to:
    <schema>
    {
        "type": "object",
        "properties": {
            "name": {
                "type": "string"
            },
            "email": {
                "type": "string"
            },
            "shippingAddress": {
                "type": "string"
            }
        },
        "required": [
            "name",
            "email",
            "shippingAddress"
        ],
        "additionalProperties": false
    }
    </schema>

user
    We are registering 'Global Exports Ltd.' for your services. The main contact person
    is Samantha Davis, and her corporate email is s.davis@globalexports.co.uk. All ship-
    ments and correspondence should be directed to our headquarters: Global Exports Ltd.,
    12 Business Park Road, Manchester, M1 1AB, United Kingdom. We are looking forward to
    a fruitful partner ship and are particularly interested in your international ship-
    ping rates.

assistant
    ```json
```

Figure 9: Example prompt for the JSON task. The JSON schema in green is task-specific as well as the the user prompt in blue from which information should be extracted into the given schema. The system prompt and prefilled assistant response are fixed.
```

````
system
    You are a specialized AI assistant that generates SMILES (Simplified Molecular Input
    Line Entry System) strings from chemical descriptions. You will be given a textual
    description of a chemical compound or a related task. Your goal is to produce the
    most accurate and valid SMILES string representing that description.

    Your Task:

    Based on the provided "input" description, generate the corresponding SMILES string.

    Output Requirements:

    - Provide only the SMILES string as your output.
    - Ensure the SMILES string is syntactically valid.
    - Represent all specified chemical features accurately (atoms, bonds, rings,
        aromaticity, charge, isotopes, stereochemistry).

    Output:

    - Provide only the smiles molecule as a raw string between triple backticks (```).
    For instance:
    ```smiles
    C1=CC=CC=C1
    ```

user
    Propan-1-amine, a primary amine with a three-carbon straight chain and the amino
    group on the first carbon.

assistant
    ```smiles
````

Figure 10: Example prompt for the SMILES task. The user prompt in blue varies per task.
````

```
user
    Your goal is to create challenging and diverse `JSON Schema` problems. You are
    given a JSON schema that describes a specific schema for a JSON problem.

    You should generate **{num_samples}** JSON benchmark samples based on the
    provided schema. A benchmark sample consists of a natural language description
    describing how the JSON schema should be filled out, along with a JSON object
    that adheres to the schema.

    For each sample, provide a JSON object with the following structure:

    ```json
    {{
        "input": "A natural language description of how the JSON schema should be
            filled out. The input should be a natural query that a user might ask an
            LLM. The input will be given to the LLM as a prompt, along with the JSON
            schema. Based on this input, the LLM should generate a JSON object that
            adheres to the schema.",
        "output": "A JSON object that adheres to the provided schema. The output
            should be a valid JSON object that matches the schema and reflects the
            input description."
    }}
    ```

    Guidelines for generating samples:

    - **Variety**: Describe a wide range of scenarios that can be expressed using
        the JSON schema. Ensure that the samples cover a wide range of possible
        scenarios, and make them sound natural and plausible.
    - **Difficulty**: User queries can and should contain distracting information
        and longer backgrounds.
    - **Realism**: Test cases should reflect plausible scenarios where the JSON
        schema would be used.
    - **Reference**: Do not reference the JSON schema in the input description. The
        input should be a natural query that a user might ask an LLM. It should not
        reference JSON at all.

    JSON Schema:
    {schema}

    Example Input (Do not use this in your samples):
    {input_query}

    Example Output (Do not use this in your samples):
    {output_query}
```

Figure 11: Prompt used to generate additional JSON Schema samples for the JSON task using GEMINI-2.5-PRO. Several samples were generated at the same time to increase diversity.

````
user
    You are a JSON Schema assistant. You will be given a textual description of how
    a JSON schema should be filled out. Your task is to generate a JSON object that
    adheres to the provided schema.

    Your Task:
    - Analyze the textual task.
    - Construct a JSON object that correctly implements the task based on the
        provided schema.

    The JSON object should be a valid JSON object that matches the schema and
    reflects the input description.

    Output:
    - Provide only the JSON object as a raw string between triple backticks
        (```json). Ensure the JSON object satisfies the JSON schema. For instance:
    ```json
    {{
    "key": "value",
    "number": 42,
    "array": [1, 2, 3]
    }}
    ```

    Json Schema:
    {schema}

    Description:
    {input_query}
````

Figure 12: Prompt used to verify additional JSON Schema samples for the JSON task using GEMINI-2.5-PRO.

```
user
    You are a specialized AI assistant tasked with generating benchmark samples for
    SMILES (Simplified Molecular Input Line Entry System) string generation. Your
    goal is to create diverse and accurate chemical structure descriptions and their
    corresponding SMILES strings.

    Please generate **{num_samples}** benchmark samples.

    The difficulty of these samples should be: **{difficulty_description}**.
    Examples of difficulty levels:
    * **Beginner**: Simple acyclic molecules, common functional groups (e.g.,
        ethanol, acetic acid, propanamine), small alkanes/alkenes/alkynes.
    * **Intermediate**: Molecules with single or multiple rings (e.g., cyclohexane,
        pyridine, naphthalene), basic stereochemistry (R/S, E/Z using `@@`, `/`, `\`),
        common drugs or biomolecules (e.g., aspirin, glucose in its open-chain form).
    * **Advanced**: Complex polycyclic systems (e.g., steroids, bridged compounds),
        detailed stereochemistry, isotopic labeling, salts, mixtures, or reaction
        SMILES (if the task is to represent a reaction).

    For each sample, provide a JSON object with the following structure:

    ```json
    {{
    "input": "A natural language description of a chemical compound or a task that
        uniquely defines a chemical structure representable by a SMILES string.
        This could be an IUPAC name, a common name, a structural description, or
        a request to modify a base structure.",
    "output": "The correct and valid SMILES string for the chemical structure
        described in the 'input'. Correctness and validity are paramount."
    }}
    ```

    Guidelines for generating samples:

    - **Accuracy**: The generated SMILES string in the "output" field MUST
        accurately represent the chemical structure described in the "input". Ensure
        correct atom types, bond orders, connectivity, aromaticity, charges,
        isotopes, and stereochemistry as implied by the input.
    - **Validity**: All generated SMILES strings must be syntactically valid.
    - **Clarity of Input**: The "input" description should be unambiguous and
        provide enough information to define a specific chemical structure. Avoid
        overly vague descriptions.
    - **Variety**: Generate a diverse set of samples covering different chemical
        families, structural features (rings, unsaturation, heteroatoms, functional
        groups), and complexities according to the specified difficulty.

    Output Format:

    Return a JSON list containing the {num_samples} generated JSON objects.
```

Figure 13: Prompt used to generate additional samples for the SMILES task using GEMINI-2.5-PRO. Several samples were generated at the same time to increase diversity.

````
user
    You are a specialized AI assistant that generates SMILES (Simplified Molecular
    Input Line Entry System) strings from chemical descriptions. You will be given
    a textual description of a chemical compound or a related task. Your goal is
    to produce the most accurate and valid SMILES string representing that
    description.

    Your Task:

    Based on the provided "input" description, generate the corresponding SMILES
    string.

    Output Requirements:

    - Provide only the SMILES string as your output.
    - Ensure the SMILES string is syntactically valid.
    - Represent all specified chemical features accurately (atoms, bonds, rings,
        aromaticity, charge, isotopes, stereochemistry).

    Output:

    - Provide only the smiles molecule as a raw string between triple backticks (```).
    For instance:
    ```smiles
    C1=CC=CC=C1
    ```

    {sample}
````

Figure 14: Prompt used to verify samples for the SMILES task using GEMINI-2.5-PRO.