HyGenar: An LLM-Driven Hybrid Genetic Algorithm for Few-Shot Grammar Generation

Weizhi Tang, Yixuan Li, Chris Sypherd, Elizabeth Polgreen, Vaishak Belle
University of Edinburgh
Weizhi.Tang@ed.ac.uk , yixuan.li.cs@ed.ac.uk , c.n.sypherd@sms.ed.ac.uk ,
elizabeth.polgreen@ed.ac.uk , vbelle@ed.ac.uk

Abstract

Grammar plays a critical role in natural language processing and text/code generation by enabling the definition of syntax, the creation of parsers, and guiding structured outputs. Although large language models (LLMs) demonstrate impressive capabilities across domains, their ability to infer and generate grammars has not yet been thoroughly explored. In this paper, we aim to study and improve the ability of LLMs for few-shot grammar generation, where grammars are inferred from sets of a small number of positive and negative examples and generated in Backus-Naur Form. To explore this, we introduced a novel dataset comprising 540 structured grammar generation challenges, devised 6 metrics, and evaluated 8 various LLMs against it. Our findings reveal that existing LLMs perform sub-optimally in grammar generation. To address this, we propose an LLM-driven hybrid genetic algorithm, namely HyGenar, to optimize grammar generation. HyGenar achieves substantial improvements in both the syntactic and semantic correctness of generated grammars across LLMs [1].

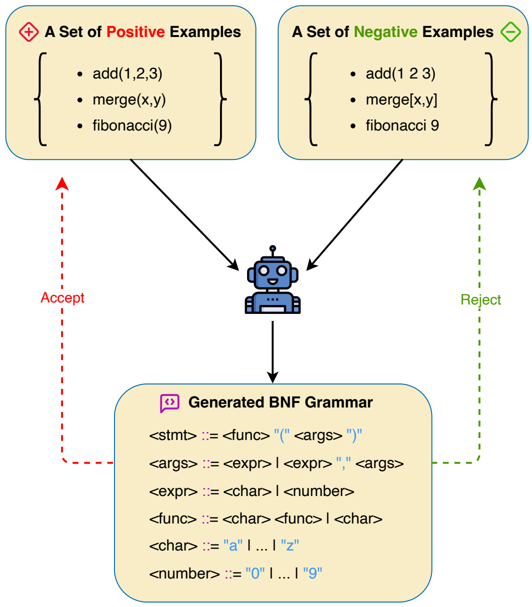

Figure 1: Given a small set of positive and negative examples, LLMs should infer and generate a grammar that accepts all positives and rejects all negatives.

1 Introduction

Grammar inference, also known as grammar induction, consists of inferring a grammar from a set of examples (Horning, 1969; De la Higuera, 2010; Stevenson and Cordy, 2014b; D'Ulizia et al., 2011). It has been studied and used in various fields, such as natural language processing, where it can reduce the effort required to generate syntactic or semantic models automatically (Kai et al., 2024; D'ulizia et al., 2011), and software engineering, where inferred grammars can guide reverse engineering and automated parser generation (Stevenson and Cordy, 2014a). By relying on characteristic examples (De la Higuera, 2010), grammar inference enables automated discovery of underlying syntactic or structural patterns.

Backus-Naur Form (BNF) is used to define the grammar of formal languages (Chomsky, 1956; Backus, 1959; Backus et al., 1960), typically Context-Free Grammars (CGFs) (Chomsky, 1956; Aho, 2007). It has also been used in various ways such as in parser generators like ANTLR4 (Parr, 2013) and Yacc (Johnson and Hill, 1978) or in constraining output structure of Large Language Models (LLMs) (Willard and Louf, 2023; Beurer-Kellner et al., 2024).

Although LLMs have exhibited remarkable capabilities across diverse domains (Gaur and Saunshi, 2023; Imani et al., 2023; Pan et al., 2023; Tang and Belle, 2024; Li et al., 2024; Jiang et al., 2024a), their capacity for grammar inference, and partic-

[1] The code is open-source and available at https://github.com/RutaTang/HyGenar.

Findings of the Association for Computational Linguistics: ACL 2025, pages 13640–13665
July 27 - August 1, 2025

ularly for generating grammars in BNF, has not yet been well explored. This paper focuses on investigating and improving the ability of LLMs to infer a CFG and generate it in BNF from a given set of positive and negative examples. A correctly generated grammar should accept all positives and reject all negatives. Typically, grammar inference requires a large set of characteristic examples that can uniquely identify a CFG (De la Higuera, 2010). However, we instead explore whether LLMs can infer a CFG from fewer examples without imposing any constraints on the examples, based on their experience and knowledge acquired through training on large-scale corpora. We refer to this process as "few-shot grammar generation", emphasizing both the grammar inference of CFGs with fewer examples and their generation in BNF, and we use "grammar" to denote a CFG represented in BNF. An example is shown in Figure 1.

To explore the capacity of LLMs for few-shot grammar generation, we first construct a dedicated dataset of 540 challenges, each with only 3 positive and 3 negative examples, and use it to evaluate the performance of 8 LLMs, encompassing both open- and closed-source models of varying parameter sizes, including those specialized for code generation. We then propose an LLM-driven hybrid genetic algorithm, namely HyGenar, which adapts the principles of genetic algorithms with the integration of LLM-driven population initialization and mutation. We devised and adopted 6 metrics to comprehensively evaluate and analyze their performance from the perspectives of syntax and semantic correctness, over-fitting, over-generalization, and utility. The results show that, while most LLMs demonstrate unsatisfactory performance, our proposed algorithm significantly enhances the grammar generation ability across most of the evaluated LLMs.

We summarize the main contributions of this paper as follows:

1. We constructed a dedicated dataset of 540 challenges for few-shot grammar generation and comprehensively evaluated 8 LLMs;

2. We designed 6 metrics for measuring the ability of LLMs in this task and performed an extensive analysis based on them;

3. We proposed a novel method, HyGenar, to improve the grammar generation performance of LLMs and showed that it achieves significant improvements across LLMs.

2 Background

2.1 Context-Free Grammar

A *context-free grammar* (CFG) consists of terminals, non-terminals, a start symbol, and production rules (Hopcroft et al., 2001; Aho, 2007). It can be formally defined as a quadruple $\mathcal{G} = (V, \Sigma, \Pi, S)$, where V is a finite set of non-terminal symbols, Σ is the set of terminals, $\Sigma \cap V = \emptyset$, Π is a finite set of production rules, $\Pi \subseteq V \times (V \cup \Sigma)^*$, and $S \in V$ is the start symbol of the grammar. Elements of $(V \cup \Sigma)^*$ are known as sentential forms.

The language generated by $\mathcal{G}$, denoted $\mathcal{L}(\mathcal{G})$, comprises all strings derivable from S using the rules in Π: $\mathcal{L}(\mathcal{G}) = \{\sigma \in \Sigma^* \mid S \xrightarrow{*} \sigma\}$. For $\alpha, \beta \in (V \cup \Sigma)^*$, we say α directly derives β in one step as $\alpha \to \beta$, and define $\alpha \xrightarrow{*} \beta$ as α deriving β in zero or more steps if there exists a finite sequence of $\gamma_0, \ldots, \gamma_n \in (V \cup \Sigma)^*$ where $n \geq 0$, such that $\alpha = \gamma_0 \to \gamma_1 \to \cdots \to \gamma_n = \beta$.

2.2 Backus-Naur Form

Backus–Naur Form (BNF) is a notation used to define the grammar of formal languages, typically CFG (Chomsky, 1956; Backus, 1959; Backus et al., 1960). In this paper, LLM generates grammars in BNF.

A context-free grammar $\mathcal{G} = (V, \Sigma, \Pi, S)$ is given in BNF notation as a list of grouped production rules, where each rule group is written:

$$<v_i> ::= \alpha_1 | \alpha_2 | \ldots$$

where v_i enclosed between the pair "< >" is a nonterminal symbol in V, and $\alpha_1, \alpha_2 \ldots \in (\Sigma \cup V)^*$ is the list of sentential forms that can be derived in one step from v_i. This represents the set of all production rules with v_i on the left-hand side, i.e., $v_i \to \alpha_1, v_i \to \alpha_2, \ldots$.

We extend the definition of a CFG to define a grammar in BNF to be $G = (V, \Sigma, \Pi, S, R)$, where V, Σ, Π, and S are defined as before. R is a set of sets of production rules, denoted $\{r_1, \ldots r_n\}$. Each $r_i \in R$ is a set of production rules where the left-hand side of all rules is the i^{th} non-terminal symbol v_i, i.e., r_i is $(v_i \times (\Sigma \cup V)^*) \cap \Pi$. Since each non-terminal symbol has a corresponding rule set, $n = |V| = |R|$. The language of the BNF grammar G is denoted by $\mathcal{L}(G)$ and is equal to the language of the original CFG, $\mathcal{L}(\mathcal{G})$.

We say that a grammar G is a *valid* grammar, and that $valid(G)$ evaluates to true, if it has a correct BNF syntax, and if R is as defined above, and if all nonterminal symbols have at least one rule in their corresponding rule set.

2.3 Grammar Inference

Grammar inference aims to learn grammar automatically from a set of examples (Horning, 1969; De la Higuera, 2010; Stevenson and Cordy, 2014b; D'Ulizia et al., 2011). In this paper, we focus on inferring a CFG, in BNF notation, from a small set of positive and negative examples.

Given a set of positive examples $\mathcal{P}$ and negative examples $\mathcal{N}$, consisting of strings that must be, respectively, accepted and rejected, the objective is to infer a BNF grammar G. The generated G should satisfy $\mathcal{P} \subseteq \mathcal{L}(G)$ and $\mathcal{N} \cap \mathcal{L}(G) = \emptyset$, which ensures G accepts all positive examples and rejects all negative examples.

3 Related Work

3.1 Grammar Generation

Grammar inference has been widely studied and applied across various fields, such as natural language processing (Kai et al., 2024; D'Ulizia et al., 2011), bio-informatics (De la Higuera, 2010), pattern recognition (Pedro et al., 2013; Richetin and Vernadat, 1984; De la Higuera, 2010), and software engineering (Schröder and Cito, 2022; Stevenson and Cordy, 2014b). Previous works have also proposed various approaches for grammar inference (Rodrigues and Lopes, 2007; Cohen et al., 2017; Li et al., 2023; D'ulizia et al., 2011; Chen, 1995). However, few works are directly related to exploring the ability of LLMs for few-shot grammar generation, which, to reiterate, is to infer grammars from a small set of positives and negatives while generating them in BNF.

3.2 Code Generation

LLMs demonstrate the ability of code generation (Jiang et al., 2024b; Huang et al., 2023; Dehaerne et al., 2022), with various approaches proposed to improve it (Shinn et al., 2023; Madaan et al., 2023; Huang et al., 2023; Jiang et al., 2023b; Chen et al., 2023). We consider grammar generation to share notable similarities with code generation, since in grammar generation it is not only required to infer grammars but also to generate grammars in BNF. Thus, following a similar approach to Reflexion (Shinn et al., 2023) and Self-Refine (Madaan et al., 2023) to enhance code generation, we propose a method as one of the baselines for evaluation.

4 Grammar Generation Ability of LLMs

In this section, we describe the dedicated dataset constructed to evaluate the ability of LLMs in grammar generation, introduce 6 metrics we use in evaluation, explain experiments, and analyze results in detail. We detail each in the following subsections.

4.1 Dataset

To evaluate the capacity of LLMs for grammar generation, we present a dedicated dataset.

During dataset construction, for each $k \in \{1, 2, \ldots, 9\}$, we prompted GPT-4o (OpenAI et al., 2024) to generate 10 distinct reference grammars, where each reference grammar G^{ref} has k nonterminal symbols and thus $|R| = k$. This gives a total of 90 reference grammars. We used G^{ref} to prompt GPT-4o to produce 6 different challenges with each challenge consisting of a set of positives $\mathcal{P}$ and negatives $\mathcal{N}$ where $|\mathcal{P}| = 3$ and $|\mathcal{N}| = 3$, in a way that $\mathcal{P} \subseteq \mathcal{L}(G^{ref})$ and $\mathcal{N} \cap \mathcal{L}(G^{ref}) = \emptyset$. However, GPT-4o often failed to produce challenges with valid reference grammars, positives, and negatives as k increased. We manually corrected erroneous reference grammars, positives, and negatives by using a BNF parser which takes a grammar and outputs whether a grammar is in valid BNF, and whether positives are accepted, and negatives are rejected[2].

Following this process, we obtained a dataset of 540 challenges, each consisting of 3 positives and 3 negatives. Figure 1 shows an example challenge and a corresponding solution.

4.2 Metrics

Let C be a set of N challenges where each is a tuple $(G^{ref}, \mathcal{P}, \mathcal{N}, G^*)$ consisting of a reference grammar, and a set of positive examples and negative examples, respectively, and the corresponding candidate grammar G^* generated by an LLM. We evaluate the quality of generated grammars using 6 key metrics:

Syntax Correctness (SX) The syntax correctness metric SX quantifies the proportion of guesses

[2]Refer to Appendix A for the details of dataset construction.

that conform to the valid BNF syntax defined in Section 2.2. We define an indicator function as:

$$\mathbb{I}_{SX}(G^*) = \begin{cases} 1 & \text{if } valid(G^*), \\ 0 & \text{otherwise.} \end{cases}$$

$SX(C)$ is defined as: $\frac{1}{N}\sum_{i=1}^{N}\mathbb{I}_{SX}(G_i^*)$.

Semantic Correctness (SE) SE captures the proportion of guesses that are semantically correct. We define an indicator function as follows, noting that if G^* is not in valid BNF, then $\mathcal{L}(G^*) = \emptyset$:

$$\mathbb{I}_{SE}(G^*, \mathcal{P}, \mathcal{N}) = \begin{cases} 1 & \text{if } \mathcal{P} \subseteq \mathcal{L}(G^*) \wedge \\ & \mathcal{N} \cap \mathcal{L}(G^*) = \emptyset \\ 0 & \text{otherwise.} \end{cases}$$

$SE(C)$ is given by: $\frac{1}{N}\sum_{i=1}^{N}\mathbb{I}_{SE}(G_i^*, \mathcal{P}_i, \mathcal{N}_i)$.

Estimating Grammar Quality Given a set of positive and negative examples, there are many possible valid and semantically correct solutions that are undesirable. For instance, the following grammar would be an undesirable solution for Figure 1. Here the $\mathcal{L}(G^*) = |P|$, and is overfitted to the examples:

```
<stmt> ::= "add(1,2,3)" | "merge(x,y)" |
           "fibonacci(9)"
```

Equally undesirable is a grammar that over-generalizes from the examples, and defines a significantly larger language than the reference grammar, because this is highly likely to contain invalid strings. There are no standard metrics to measure over-fitting or over-generalization in grammar generation, so we devise 4 metrics based around the number of production rules used in parsing the positive examples.

First, let $\Pi_{\mathcal{P}} \subseteq \Pi$ be the set of production rules that are used in the left-most derivations of all positive examples in $\mathcal{P}$. That is, the set of rules in Π which occur in a sequence of rules $S \rightarrow \alpha_1 \rightarrow \ldots \rightarrow \alpha_n \rightarrow p$ where $p \in \mathcal{P}$, and all rules expand the left-most non-terminal in $\alpha_1, \ldots, \alpha_n$.

We report metrics across only *solved* challenges, i.e., a challenge where G^* is syntactically and semantically correct. Our four metrics are defined as follows[3]:

- *Diff*(C), calculates the difference between the number of production rules in G^* used in parsing the positive examples and the number of production rules used by G^{ref} for a given challenge, i.e., $|\Pi_{\mathcal{P}}^{ref}| - |\Pi_{\mathcal{P}}^*|$. A grammar that uses substantially fewer production rules has probably overfitted to the examples, and a grammar that uses substantially more production rules may have over-generalized. We report the average of *Diff* across all solved challenges as *Diff*$^\diamond$.
- OF estimates over-fitting by counting the percentage of solved challenges on which G^* uses fewer than half the number of production rules used by G^{ref}, i.e., the number of times $|\Pi_{\mathcal{P}}^{ref}| - |\Pi_{\mathcal{P}}^*| > \frac{|\Pi_{\mathcal{P}}^{ref}|}{2}$.
- OG estimates over-generalization by counting the percentage of solved challenges on which G^* uses more than half the number of production rules used by G^{ref}, i.e., $|\Pi_{\mathcal{P}}^{ref}| - |\Pi_{\mathcal{P}}^*| < -\frac{|\Pi_{\mathcal{P}}^{ref}|}{2}$.
- TU calculates the proportion of production rules that are used in parsing the positive examples for a given challenge, i.e., $\frac{|\Pi_{\mathcal{P}}^*|}{\Pi^*}$, indicating the utility. We report the average of TU across all solved challenges as $TU^\diamond$.

4.3 Baselines

To establish baselines, we adopted two approaches, Direct Prompting (DP) and Optimization of the BNF Parser for LLM-Friendly Feedback (OPF).

In DP, we directly prompted LLMs with positive and negative examples, asking them to produce grammars that accept all positives and reject all negatives[4].

In OPF, inspired by Reflexion (Shinn et al., 2023) and Self-Refine (Madaan et al., 2023), we further optimized the BNF parser by enabling it to provide more LLM-friendly error messages as feedback, aiming to improve the performance of grammar generation for LLMs[5].

4.4 Experiment Settings

For a comprehensive evaluation, we selected a total of 8 LLMs, ensuring a diverse selection of both open- and closed-source LLMs, along with LLMs with varying parameter sizes and LLMs specifically designed for code generation.

Specifically, we selected two closed-source LLMs, *GPT-4o* (OpenAI et al., 2024) and *GPT-3.5-Turbo* (Brown et al., 2020a). For the other 6

[3]Refer to Appendix B for the details of formal definitions.

[4]Refer to Appendix C for the details of DP.

[5]Refer to Appendix D for the details of OPF.

open-source LLMs, we note them in the notation: {*model_name*}:{*parameter_size*}-{*model_type*}. We selected the following LLMs: *Llama3:70b-Instruct* (Grattafiori et al., 2024), *Qwen:72b-Chat* (Bai et al., 2023), *Gemma2:27b-Instruct* (Team et al., 2024), *Mistral:7b-Instruct* (Jiang et al., 2023a), *Codestral:22b* (MistralAI, 2024), and *Starcoder2:15b-Instruct* (Lozhkov et al., 2024). Among the selected LLMs, *Starcoder2:15b-Instruct* and *Codestral:22b* are LLMs for code generation.

For the DP baseline, we set the temperature to 0 and the maximum token to 2000.

For the OPF baseline[6], we set the temperature to 0.3, the maximum token to 2000, and the maximum number of turns, *max_turns*, to 5.

4.5 Results & Analysis

The results of SX and SE for the 8 LLMs are presented in Table 1. We observed that *GPT-4o*, *GPT-3.5-Turbo*, and *Gemma2:27b-Instruct* achieve relatively high SX, while the other LLMs generally fare worse in DP baseline. Compared to the DP baseline, applying OPF leads to a significant enhancement in SX for both *Mistral:7b-Instruct* and *Codestral:22b*, yielding an 18% improvement for each of them, suggesting that parser feedback can help increase SX. Nevertheless, for most LLMs, OPF yields only slight gains in SX. Furthermore, it is worth noting that for *Starcoder2:15b-Instruct*, SX decreases by 16% after applying OPF. This indicates that feedback from the parser can sometimes lead to performance degradation if LLMs fail to interpret the feedback correctly.

Although several LLMs attain high SX, their SE remains low in DP, with an average of 39%. For example, *GPT-3.5-Turbo* has 94% SX but only 37% SE. With OPF, LLMs like *Mistral:7b-Instruct* and *Codestral:22b* achieve notable SE improvement with enhanced SX, illustrating that improved SX can positively influence SE. However, for most LLMs, OPF yields only marginal SE gains, particularly in the case that their SX is already high and OPF fails to contribute significant SX improvement.

Noticeably, although OPF helps improve SX, a significant gap still persists between SX and SE for most LLMs. For example, even after OPF, *GPT-3.5-Turbo* maintains a 57% gap between SX and SE, and *Starcoder2:15b-Instruct* has a 40% gap. Since the ultimate objective is to improve SE, a more advanced approach is needed to address this limitation.

Furthermore, as other four metrics shown in Table 2, on average, for most of LLMs, the lower $Diff^\diamond$, OF, and OG indicate negligible over-fitting or over-generalization issues, and the higher $TU^\diamond$ means LLMs are not predisposed to generate irrelevant production rules. Nevertheless, as the number of production rules increases, $Diff^\diamond$ and OF exhibit slight increases, indicating a tendency toward mild over-fitting, while no significant over-generalization issues are observed [7].

In addition to the quantitative metrics, we also conducted a qualitative analysis by examining the generated grammars. We compared the grammars containing syntactic errors generated in the DP but corrected after the OPF. We primarily found four significant issues causing lower SX: unsupported symbols such as injecting quantifiers like “*” and “?” or character classes like “[a-z]”, erroneously introduced and misplaced brackets such as wrapping two terminals with round and square brackets, failing to wrap non-terminal with angle brackets, and omission the separators “|” between sentential forms. While the the first two issues are rampant across most LLMs, the third issue is mainly found in *Mistral:7b-Instruct* and the last one is sporadic. For most LLMs, OPF can occasionally mitigate these issues, but insignificantly. Furthermore, we also observed LLMs show an ability to recognize keywords in the examples, treating them as complete terminals rather than decomposing them into multiple terminals as individual characters. For example, they treat “if” and “SELECT” as complete terminals rather than splitting them into multiple characters. This may be benefited from the common sense acquired by LLMs from the corpus (Brown et al., 2020b).

5 LLM-Driven Hybrid Genetic Algorithm

The results of the baselines demonstrate the unsatisfactory performance of LLMs in grammar generation. To address this, we propose an LLM-driven hybrid genetic algorithm, namely HyGenar, a novel algorithm inspired by the concept of genetic algorithms. The following sections detail HyGenar and elaborate on our experiment settings, results, and analysis.

[6]Refer to Appendix D for the reason of setting the temperature greater than 0 for OPF.

[7]Refer to Appendix F.3 and F.4 for more details.

Models	Syntax Correctness (SX)			Semantics Correctness(SE)		
	DP	OPF	**HyGenar**	DP	OPF	**HyGenar**
GPT-4o	93	**97** ↑4	96 ↑3 ↓1	84	85 ↑1	**93** ↑9 ↑8
GPT-3.5-Turbo	94	95 ↑1	**99** ↑5 ↑4	37	38 ↑1	**61** ↑24 ↑23
Llama3:70b-Instruct	57	61 ↑4	**75** ↑18 ↑14	41	42 ↑1	**61** ↑20 ↑19
Qwen:72b-Chat	47	49 ↑2	**76** ↑29 ↑27	20	21 ↑1	**38** ↑18 ↑17
Mistral:7b-Instruct	1	**19** ↑18	1 – ↓18	0	**8** ↑8	1 ↑1 ↓7
Gemma2:27b-Instruct	91	92 ↑1	**98** ↑7 ↑6	56	57 ↑1	**79** ↑23 ↑22
Starcoder2:15b-Instruct	76	60 ↓16	**98** ↑22 ↑38	30	20 ↓10	**44** ↑14 ↑24
Codestral:22b	53	71 ↑18	**80** ↑27 ↑9	44	52 ↑8	**67** ↑23 ↑15

Table 1: The results of syntax and semantic correctness for LLMs grammar generation are presented as percentages (%). For each LLM, the best syntax and semantic correctness are highlighted with bold font. Blue arrows ↑↓ represent performance differences relative to the DP baseline, while red arrows ↑↓ indicate differences relative to the OPF baseline.

5.1 Methodology

HyGenar consists of four main components: Fitness, Selection, Crossover, and Mutation. It begins by prompting an LLM to generate an initial population of candidate grammars from positive and negative examples. In each generation, the Fitness function scores each candidate and the Select function chooses a subset of the population. To form a new population, the Cross function operates on two randomly selected candidates from this subset to generate a new candidate, which is then modified by the Mutate function and added to the new population, until the maximum population size is reached. The new population then advances to the next generation. We provide pseudocode in Algorithm 1 in Appendix E. We detail each component as follows:

Fitness Given a generated grammar G^*, if it is syntactically incorrect, it is assigned a score of -1. For a valid grammar, we define two indicator functions:

$$\mathbb{I}_{\mathcal{A}}(G^*, p) = \begin{cases} 1 & \text{if } p \in \mathcal{L}(G^*), \\ 0 & \text{otherwise.} \end{cases}$$

$$\mathbb{I}_{\mathcal{R}}(G^*, n) = \begin{cases} 1 & \text{if } n \notin \mathcal{L}(G^*), \\ 0 & \text{otherwise.} \end{cases}$$

The Fitness function $\text{Fitness}(G^*, \mathcal{P}, \mathcal{N})$ is then defined as:

$$\sum_{p_i \in \mathcal{P}} \mathbb{I}_{\mathcal{A}}(G^*, p_i) + \sum_{n_i \in \mathcal{N}} \mathbb{I}_{\mathcal{R}}(G^*, n_i).$$

Selection Let $\mathbb{G} = \{G_1^*, G_2^*, \ldots, G_k^*\}$ be a population of candidate grammars. Each $G_i^* \in \mathbb{G}$ is assigned a fitness score $s_i \in S$ by the Fitness function, where $S = [s_1, s_2, \ldots, s_k]$ is a sequence of their corresponding scores. We define the Select function as:

$$\text{Select}(\mathbb{G}, S) = \{G^*_{\sigma(1)}, G^*_{\sigma(2)}, \ldots, G^*_{\sigma(\frac{k}{2})}\},$$

where $\sigma : \{1, 2, \ldots, k\} \to \{1, 2, \ldots, k\}$ is a permutation of the indices such that: $s_{\sigma(1)} \geq s_{\sigma(2)} \geq \cdots \geq s_{\sigma(k)}$. Hence, half of the candidates from the population are selected in decreasing order of their fitness scores.

Crossover The crossover function splices the production rules from two grammars together, at a randomly chosen splicing point. Let G_a^* and G_b^* be two candidate grammars, with rules R_a^* and R_b^* respectively. If both R are empty, we return one grammar at random. If one grammar has a non-empty R, we return that grammar. Otherwise, we apply the crossover function with probability ρ (the crossover rate).

The crossover operation works in the following way. Let $\ell = \min(|R_a^*|, |R_b^*|)$. First sample a crossover point $w \sim \text{Uniform}(\{1, 2, \ldots, \ell\})$. Let $R_a^* = r_1^a, r_2^a \ldots$ be the rules from G_a^* and R_b^* be the rules from G_b^*. We take the first $w - 1$ rule sets from R_a^* and then prefix these to the last $n - w$ rule sets from R_b^*, where $n = |R_b^*|$. This generates a new rule set $R' = \{r_a^1, \ldots, r_a^{w-1}, r_b^w, \ldots r_b^n\}$.

Crossover returns a new grammar $G' = (V', \Sigma', \Pi', S_a, R')$, where V' and Σ' are all non-terminal and terminal symbols in R', Π' is all rules in R', and S_a is the start symbol from G_a^*.

Mutation We use two mutation methods: mutation by LLM, and local mutation. We chose whether to mutate a grammar at all, with probability μ (the mutation rate). If the grammar has no production rules, we apply the LLM mutation. Otherwise, we apply local mutation with a probability 0.5 and LLM mutation otherwise.

LLM-Driven Mutation The LLM-driven mutation uses G^*, $\mathcal{P}$, and $\mathcal{N}$ to prompt[8] an LLM to produce a new grammar. In this approach, we expect that, with the knowledge and experience obtained by training on vast corpora, LLMs can heuristically provide more novel and dramatic modifications such as introducing new terminals or non-terminals, adding or removing production rules, or reshaping the structure of grammars, which is hard to approach with local mutation.

Local Mutation The local mutation is designed to produce incremental, targeted alterations to the grammar while preserving the majority of its original form. It is less flexible than LLM-driven mutation and it is unable to introduce new non-terminals or to drastically restructure grammars. However, it can not only potentially find a grammar candidate but also provide insights for LLM-driven mutation. Given a grammar G^*, with a set of sets of production rules R^*, local mutation comprises the following steps:

- **Rule set selection** We sample an integer $i \sim \mathrm{Uniform}(\{1, \ldots, |R|\})$. This index i chooses the rule set $r_i \in R^*$ that will be mutated.

- **Shuffle** The $\mathrm{Shuffle}$ mutation shuffles the order of symbols on the right-hand side of a production rule. That is, each rule is a mapping from a non-terminal v to a sequence of non-terminal and terminal symbols $(V \cup \Sigma)^*$, and we shuffle this sequence randomly. For example, the rule set: `<e> ::= <e> "*" <e> | <e> "/" <e>` with infix operators, may be shuffled to `<e> ::= "*" <e> <e> | "/" <e> <e>`, switching to prefix operators.

  Shuffle is applied to all production rules in r_i.

- **Space Insertion** The $\mathrm{SpaceInsert}$ mutation inserts a randomly chosen number of whitespace terminal symbols `"␣"` into the right-hand side of a production rule. Given a rule $v_i \to \alpha$, Shuffle chooses the number of whitespace terminals to be inserted by sampling $I \sim \mathrm{Uniform}(0, |\alpha|)$, where $|\alpha|$ is the number of symbols in α. Each space is inserted before or after a randomly chosen symbol in α. As an example, the rule set: `<s> ::= <noun> <verb>` may be changed to `<s> ::= <noun> "␣" <verb>`.

  For each production rule in r_i, we randomly decide whether $\mathrm{SpaceInsert}$ should be applied to that rule with a low probability[9].

The $\mathrm{Shuffle}$ alteration heuristic is motivated by two key insights. First, shuffling the right-hand side of production rules may yield a grammar that accepts more positive examples and rejects more negative ones. Second, although $\mathrm{Shuffle}$ may rarely yield a better candidate for a complex target grammar, the new variant of the grammar produced from $\mathrm{Shuffle}$ is expected to provide alternative perspectives and new insights for LLMs to help generate subsequent grammars in future generations.

The $\mathrm{SpaceInsert}$ alteration was introduced because some LLMs tended to omit explicit space symbols between symbols in an alternative, resulting in degraded grammar generation, even if they were prompted to pay attention to space inclusion[10]. We expect that incorporating $\mathrm{SpaceInsert}$ will offer insights for LLMs of the explicit inclusion of spaces to thereby enhance performance.

5.2 Experiment Settings

In addition to a set of positive and negative examples and an LLM, HyGenar takes four parameters: *population size* (grammars per generation), *generations* (number of evolution iterations), *crossover rate* (probability of crossover), *mutation rate* (probability of mutation). In our experiments, we set these to 10, 5, 0.7, and 0.3, respectively.

We selected the same 8 LLMs as the DP and OPF baselines, setting maximum tokens to 2000 and temperature to 0.7. In HyGenar, a nonzero temperature is necessary for diversity. To ensure it does not significantly impact results and show its robustness, we further repeat the experiments 5 times with *GPT-3.5-Turbo* and *GPT-4o*.

5.3 Results & Analysis

As shown in Table 1, HyGenar substantially boosts *SX* for most LLMs. For example, *Qwen:72b-Instruct* gains 29% over DP and 27% over OPF.

[8]Refer to Prompt Template 2 in Appendix E for the prompt we designed for LLM-driven mutation.

[9]We fix it to 0.1.

[10]Refer to Prompt Template 1 in Appendix C for details.

Methods	$Diff^{\diamond}$	OF	OG	$TU^{\diamond}$
DP	1.12	3.83	0.63	88.74
OPF	1.10	4.72	1.31	90.76
HyGenar	1.19	4.44	0.92	91.27

Table 2: The averages of $Diff^{\diamond}$, $OF(\%)$, $OG(\%)$, and $TU^{\diamond}(\%)$ across all LLMs.

Even for LLMs already improved by OPF, such as *Codestral:22b* with an 18% improvement, HyGenar adds an additional 9%. Meanwhile, *Starcoder2:15b-Instruct*, which experiences a 16% drop under OPF, achieves a 22% improvement compared to DP and a 38% improvement over OPF, with HyGenar. On average, it improves SX 13.88% compared to DP and 9.88% over OPF.

While enhancing SX is essential, the ultimate objective is to improve SE. As shown in Table 1, our method significantly boosts SE for all LLMs except *Mistral:7b-Instruct*. For example, with HyGenar, *GPT-4o* rises from 84% with DP and 85% with OPF to 93%, and *GPT-3.5-Turbo*, noted for low semantic accuracy, increases by 24% over DP and 23% over OPF. Notably, although the contribution from the enhancement of SX is essential to SE, HyGenar does not rely solely on enhancing SX to achieve significant improvement of SE, as five LLMs demonstrated higher SE increases than their SX. Across the selected LLMs, the average SE improvement is 16.5% compared to DP and 15.13% compared to OPF.

We further analyzed the performance as the number of non-terminals and production rules increases. For non-terminals, we partition the dataset into 3 groups: C_1 (1–3 non-terminals), C_2 (4-6 non-terminals), and C_3 (7-9 non-terminals). For production rules, we split the dataset into another 3 groups: P_1 (1–6 production rules), P_2 (6-15 production rules), and P_3 (greater than 16 production rules). We observed that the performance of LLMs decreases as the number of non-terminals and production rules increases. Nevertheless, HyGenar still substantially improves both SX and SE[11].

Furthermore, as shown in Table 2, OF does not significantly increase in HyGenar, indicating that the substantial improvements observed in both the SX and SE for HyGenar are not attributed to overfitting[12].

[11]Refer to Tables 4 and 5 in Appendix F.1 and F.2 for details.

[12]Refer to Tables 6, 7, and 8 in Appendix F.3 for the more details.

Moreover, we also conducted a qualitative analysis of how HyGenar improves SX and SE. For SX, HyGenar significantly reduces the issues of unsupported symbols injection and misplaced brackets. However, it fails to address the issue of unwrapped non-terminals, which is also the issue mainly happened in *Mistral:7b-Instruct*. Unlike OPF, which benefits from more explicit syntax error feedback, our approach lacks such direct syntax corrective guidance, meaning that if an LLM inherently struggles to generate syntactically correct grammars, our method may fail to produce valid candidates and process evolution, thereby lowering both SX and SE. Nevertheless, as long as at least a few candidates are generated in correct syntax, HyGenar can optimize their generations during the evolutionary process to mitigate the aforementioned issues and improve SX. For SE, attributing the significant improvement is complex. However, we still observed two phenomena. First, after applying HyGenar, terminals that were not previously considered in the grammars generated by DP or OPF have been introduced. Second, the semantic errors that were caused by the absence of space terminals in DP or OPF have been alleviated.

Due to the relatively high temperature of 0.7 used for HyGenar, we repeated the experiments 5 times with *GPT-4o* and *GPT-3.5-Turbo*[13] to ensure robustness. The averages of SX for *GPT-4o* and *GPT-3.5-Turbo* are 95.8% and 98.6%, with standard deviations 0.4% and 0.49% respectively, while the averages of SE are 93.2% and 61.6% with standard deviations 0.4% and 0.49% respectively. These results indicate that setting the temperature to 0.7 has a negligible impact on performance and show the robustness of HyGenar.

6 Conclusion

To explore the few-shot grammar generation ability of LLMs, we constructed a dedicated dataset consisting of 540 challenges, devised and adopted 6 metrics, and evaluated 8 various LLMs. Due to their unsatisfactory performance, we introduced HyGenar, an LLM-driven hybrid genetic algorithm for grammar generation. Our results indicate that HyGenar significantly enhances both syntax and semantic correctness compared to the two baselines. We believe this work provides valuable insights into LLM-based grammar generation and highlights the potential of LLM-driven hybrid genetic algorithms

[13]Refer to Table 3 in Appendix F.5 for details.

in this domain.

7 Limitations

We discuss several limitations and concerns in this work, revealing potential challenges, constraints, and confusion.

First, although the results indicate that *GPT-4o* exhibits remarkable SX and SE, it is important to note that these results may be attributable to the use of GPT-4o during dataset construction. Nonetheless, even though *GPT-4o* already demonstrates excellent performance, HyGenar can still enhance it significantly.

Second, as demonstrated, our method does not outperform OPF for *Mistral:7b-Instruct* in SX and SE due to its inherent failure to generate syntactically correct grammars. Nevertheless, our approach yields significant SX and SE improvements for all other LLMs. We also propose to combine syntactical feedback and HyGenar to mitigate this limitation and further improve the performance.

Third, one may argue that given any finite set of positive and negative examples, it could always be possible to construct a regular grammar rather than a CFG that can accept all positives and reject all negatives. However, such an approach may function more like a classifier rather than a grammar and may lack applicability in subsequent tasks, such as constructing an abstract syntax tree.

Finally, in this work, we primarily focus on LLM-based few-shot grammar generation without comparing algorithms that are not LLM-based. The reasons behind this are that most algorithms require a large set of characteristic examples to uniquely determine the target grammar (De la Higuera, 2010). Instead, we do not impose such constraints on our example set and hypothesize that the experience and knowledge acquired from corpus can enable LLMs to handle few-shot grammar generation tasks. Consequently, those algorithms may not be directly applicable. In addition, since we focus on the exploration and improvement of the ability of LLMs in few-shot grammar generation, we construct two LLM-based baselines for fair comparison.

References

2007. *Compilers: principles, techniques, tools*, 2nd ed edition. Pearson/Addison Wesley, Boston.

John W Backus. 1959. The syntax and the semantics of the proposed international algebraic language of the zurich acm-gamm conference. In *ICIP Proceedings*, pages 125–132.

John W Backus, Friedrich L Bauer, Julien Green, Charles Katz, John McCarthy, Alan J Perlis, Heinz Rutishauser, Klaus Samelson, Bernard Vauquois, Joseph Henry Wegstein, et al. 1960. Report on the algorithmic language algol 60. *Communications of the ACM*, 3(5):299–311.

Jinze Bai, Shuai Bai, Yunfei Chu, Zeyu Cui, Kai Dang, Xiaodong Deng, Yang Fan, Wenbin Ge, Yu Han, Fei Huang, Binyuan Hui, Luo Ji, Mei Li, Junyang Lin, Runji Lin, Dayiheng Liu, Gao Liu, Chengqiang Lu, Keming Lu, Jianxin Ma, Rui Men, Xingzhang Ren, Xuancheng Ren, Chuanqi Tan, Sinan Tan, Jianhong Tu, Peng Wang, Shijie Wang, Wei Wang, Shengguang Wu, Benfeng Xu, Jin Xu, An Yang, Hao Yang, Jian Yang, Shusheng Yang, Yang Yao, Bowen Yu, Hongyi Yuan, Zheng Yuan, Jianwei Zhang, Xingxuan Zhang, Yichang Zhang, Zhenru Zhang, Chang Zhou, Jingren Zhou, Xiaohuan Zhou, and Tianhang Zhu. 2023. Qwen technical report. *Preprint*, arXiv:2309.16609.

Luca Beurer-Kellner, Marc Fischer, and Martin T. Vechev. 2024. Guiding llms the right way: Fast, non-invasive constrained generation. *ArXiv*, abs/2403.06988.

Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020a. Language models are few-shot learners. *Preprint*, arXiv:2005.14165.

Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020b. Language models are few-shot learners. In *Proceedings of the 34th International Conference on Neural Information Processing Systems*, NIPS '20, Red Hook, NY, USA. Curran Associates Inc.

Stanley F Chen. 1995. Bayesian grammar induction for language modeling. *arXiv preprint cmp-lg/9504034*.

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. 2023. Teaching large language models to self-debug. *ArXiv*, abs/2304.05128.

N. Chomsky. 1956. Three models for the description of language. *IRE Transactions on Information Theory*, 2(3):113–124.

Mor Cohen, Avi Caciularu, Idan Rejwan, and Jonathan Berant. 2017. Inducing regular grammars using recurrent neural networks. *ArXiv*, abs/1710.10453.

Colin De la Higuera. 2010. *Grammatical inference: learning automata and grammars*. Cambridge University Press.

Enrique Dehaerne, Bappaditya Dey, Sandip Halder, Stefan De Gendt, and Wannes Meert. 2022. Code generation using machine learning: A systematic review. *IEEE Access*, 10:82434–82455.

Arianna D'ulizia, F. Ferri, and P. Grifoni. 2011. A survey of grammatical inference methods for natural language learning. *Artificial Intelligence Review*, 36:1–27.

Arianna D'Ulizia, Fernando Ferri, and Patrizia Grifoni. 2011. A survey of grammatical inference methods for natural language learning. *Artificial Intelligence Review*, 36(1):1–27.

Vedant Gaur and Nikunj Saunshi. 2023. Reasoning in large language models through symbolic math word problems. In *Findings of the Association for Computational Linguistics: ACL 2023*, pages 5889–5903, Toronto, Canada. Association for Computational Linguistics.

Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, Amy Yang, Angela Fan, Anirudh Goyal, Anthony Hartshorn, Aobo Yang, Archi Mitra, Archie Sravankumar, Artem Korenev, Arthur Hinsvark, Arun Rao, Aston Zhang, Aurelien Rodriguez, Austen Gregerson, Ava Spataru, Baptiste Roziere, Bethany Biron, Binh Tang, Bobbie Chern, Charlotte Caucheteux, Chaya Nayak, Chloe Bi, Chris Marra, Chris McConnell, Christian Keller, Christophe Touret, Chunyang Wu, Corinne Wong, Cristian Canton Ferrer, Cyrus Nikolaidis, Damien Allonsius, Daniel Song, Danielle Pintz, Danny Livshits, Danny Wyatt, David Esiobu, Dhruv Choudhary, Dhruv Mahajan, Diego Garcia-Olano, Diego Perino, Dieuwke Hupkes, Egor Lakomkin, Ehab AlBadawy, Elina Lobanova, Emily Dinan, Eric Michael Smith, Filip Radenovic, Francisco Guzmán, Frank Zhang, Gabriel Synnaeve, Gabrielle Lee, Georgia Lewis Anderson, Govind Thattai, Graeme Nail, Gregoire Mialon, Guan Pang, Guillem Cucurell, Hailey Nguyen, Hannah Korevaar, Hu Xu, Hugo Touvron, Iliyan Zarov, Imanol Arrieta Ibarra, Isabel Kloumann, Ishan Misra, Ivan Evtimov, Jack Zhang, Jade Copet, Jaewon Lee, Jan Geffert, Jana Vranes, Jason Park, Jay Mahadeokar, Jeet Shah, Jelmer van der Linde, Jennifer Billock, Jenny Hong, Jenya Lee, Jeremy Fu, Jianfeng Chi, Jianyu Huang, Jiawen Liu, Jie Wang, Jiecao Yu, Joanna Bitton, Joe Spisak, Jongsoo Park, Joseph Rocca, Joshua Johnstun, Joshua Saxe, Junteng Jia, Kalyan Vasuden Alwala, Karthik Prasad, Kartikeya Upasani, Kate Plawiak, Ke Li, Kenneth Heafield, Kevin Stone, Khalid El-Arini, Krithika Iyer, Kshitiz Malik, Kuenley Chiu, Kunal Bhalla, Kushal Lakhotia, Lauren Rantala-Yeary, Laurens van der Maaten, Lawrence Chen, Liang Tan, Liz Jenkins, Louis Martin, Lovish Madaan, Lubo Malo, Lukas Blecher, Lukas Landzaat, Luke de Oliveira, Madeline Muzzi, Mahesh Pasupuleti, Mannat Singh, Manohar Paluri, Marcin Kardas, Maria Tsimpoukelli, Mathew Oldham, Mathieu Rita, Maya Pavlova, Melanie Kambadur, Mike Lewis, Min Si, Mitesh Kumar Singh, Mona Hassan, Naman Goyal, Narjes Torabi, Nikolay Bashlykov, Nikolay Bogoychev, Niladri Chatterji, Ning Zhang, Olivier Duchenne, Onur Çelebi, Patrick Alrassy, Pengchuan Zhang, Pengwei Li, Petar Vasic, Peter Weng, Prajjwal Bhargava, Pratik Dubal, Praveen Krishnan, Punit Singh Koura, Puxin Xu, Qing He, Qingxiao Dong, Ragavan Srinivasan, Raj Ganapathy, Ramon Calderer, Ricardo Silveira Cabral, Robert Stojnic, Roberta Raileanu, Rohan Maheswari, Rohit Girdhar, Rohit Patel, Romain Sauvestre, Ronnie Polidoro, Roshan Sumbaly, Ross Taylor, Ruan Silva, Rui Hou, Rui Wang, Saghar Hosseini, Sahana Chennabasappa, Sanjay Singh, Sean Bell, Seohyun Sonia Kim, Sergey Edunov, Shaoliang Nie, Sharan Narang, Sharath Raparthy, Sheng Shen, Shengye Wan, Shruti Bhosale, Shun Zhang, Simon Vandenhende, Soumya Batra, Spencer Whitman, Sten Sootla, Stephane Collot, Suchin Gururangan, Sydney Borodinsky, Tamar Herman, Tara Fowler, Tarek Sheasha, Thomas Georgiou, Thomas Scialom, Tobias Speckbacher, Todor Mihaylov, Tong Xiao, Ujjwal Karn, Vedanuj Goswami, Vibhor Gupta, Vignesh Ramanathan, Viktor Kerkez, Vincent Gonguet, Virginie Do, Vish Vogeti, Vítor Albiero, Vladan Petrovic, Weiwei Chu, Wenhan Xiong, Wenyin Fu, Whitney Meers, Xavier Martinet, Xiaodong Wang, Xiaofang Wang, Xiaoqing Ellen Tan, Xide Xia, Xinfeng Xie, Xuchao Jia, Xuewei Wang, Yaelle Goldschlag, Yashesh Gaur, Yasmine Babaei, Yi Wen, Yiwen Song, Yuchen Zhang, Yue Li, Yuning Mao, Zacharie Delpierre Coudert, Zheng Yan, Zhengxing Chen, Zoe Papakipos, Aaditya Singh, Aayushi Srivastava, Abha Jain, Adam Kelsey, Adam Shajnfeld, Adithya Gangidi, Adolfo Victoria, Ahuva Goldstand, Ajay Menon, Ajay Sharma, Alex Boesenberg, Alexei Baevski, Allie Feinstein, Amanda Kallet, Amit Sangani, Amos Teo, Anam Yunus, Andrei Lupu, Andres Alvarado, Andrew Caples, Andrew Gu, Andrew Ho, Andrew Poulton, Andrew Ryan, Ankit Ramchandani, Annie Dong, Annie Franco, Anuj Goyal, Aparajita Saraf, Arkabandhu Chowdhury, Ashley Gabriel, Ashwin Bharambe, Assaf Eisenman, Azadeh Yazdan, Beau James, Ben Maurer, Benjamin Leonhardi, Bernie Huang, Beth Loyd, Beto De Paola, Bhargavi Paranjape, Bing Liu, Bo Wu, Boyu Ni, Braden Hancock, Bram Wasti, Brandon Spence, Brani Stojkovic, Brian Gamido, Britt Montalvo, Carl Parker, Carly Burton, Catalina Mejia, Ce Liu, Changhan Wang, Changkyu Kim, Chao Zhou, Chester Hu, Ching-Hsiang Chu, Chris Cai, Chris Tindal, Christoph Feichtenhofer, Cynthia Gao, Damon Civin, Dana Beaty, Daniel Kreymer, Daniel Li, David Adkins, David Xu, Davide Testuggine, Delia David, Devi Parikh, Diana Liskovich, Didem Foss, Dingkang Wang, Duc Le, Dustin Holland, Edward Dowling, Eissa Jamil, Elaine Montgomery, Eleonora Presani, Emily Hahn, Emily Wood, Eric-Tuan Le, Erik Brinkman, Esteban Arcaute, Evan Dunbar, Evan Smothers, Fei Sun, Felix Kreuk, Feng Tian, Filippos Kokkinos, Firat Ozgenel, Francesco Caggioni, Frank Kanayet, Frank Seide, Gabriela Medina Florez, Gabriella Schwarz, Gada Badeer, Georgia Swee, Gil Halpern, Grant Herman, Grigory Sizov, Guangyi, Zhang, Guna Lakshminarayanan, Hakan Inan, Hamid Shojanazeri, Han Zou, Hannah Wang, Hanwen Zha, Haroun Habeeb, Harrison Rudolph, Helen Suk, Henry Aspegren, Hunter Goldman, Hongyuan Zhan, Ibrahim Damlaj, Igor Molybog, Igor Tufanov, Ilias Leontiadis, Irina-Elena Veliche, Itai Gat, Jake Weissman, James Geboski, James Kohli, Janice Lam, Japhet Asher, Jean-Baptiste Gaya, Jeff Marcus, Jeff Tang, Jennifer Chan, Jenny Zhen, Jeremy Reizenstein, Jeremy Teboul, Jessica Zhong, Jian Jin, Jingyi Yang, Joe Cummings, Jon Carvill, Jon Shepard, Jonathan McPhie, Jonathan Torres, Josh Ginsburg, Junjie Wang, Kai Wu, Kam Hou U, Karan Saxena, Kartikay Khandelwal, Katayoun Zand, Kathy Matosich, Kaushik Veeraraghavan, Kelly Michelena, Keqian Li, Kiran Jagadeesh, Kun Huang, Kunal Chawla, Kyle Huang, Lailin Chen, Lakshya Garg, Lavender A, Leandro Silva, Lee Bell, Lei Zhang, Liangpeng Guo, Licheng Yu, Liron Moshkovich, Luca Wehrstedt, Madian Khabsa, Manav Avalani, Manish Bhatt, Martynas Mankus, Matan Hasson, Matthew Lennie, Matthias Reso, Maxim Groshev, Maxim Naumov, Maya Lathi, Meghan Keneally, Miao Liu, Michael L. Seltzer, Michal Valko, Michelle Restrepo, Mihir Patel, Mik Vyatskov, Mikayel Samvelyan, Mike Clark, Mike Macey, Mike Wang, Miquel Jubert Hermoso, Mo Metanat, Mohammad Rastegari, Munish Bansal, Nandhini Santhanam, Natascha Parks, Natasha White, Navyata Bawa, Nayan Singhal, Nick Egebo, Nicolas Usunier, Nikhil Mehta, Nikolay Pavlovich Laptev, Ning Dong, Norman Cheng, Oleg Chernoguz, Olivia Hart, Omkar Salpekar, Ozlem Kalinli, Parkin Kent, Parth Parekh, Paul Saab, Pavan Balaji, Pedro Rittner, Philip Bontrager, Pierre Roux, Piotr Dollar, Polina Zvyagina, Prashant Ratanchandani, Pritish Yuvraj, Qian Liang, Rachad Alao, Rachel Rodriguez, Rafi Ayub, Raghotham Murthy, Raghu Nayani, Rahul Mitra, Rangaprabhu Parthasarathy, Raymond Li, Rebekkah Hogan, Robin Battey, Rocky Wang, Russ Howes, Ruty Rinott, Sachin Mehta, Sachin Siby, Sai Jayesh Bondu, Samyak Datta, Sara Chugh, Sara Hunt, Sargun Dhillon, Sasha Sidorov, Satadru Pan, Saurabh Mahajan, Saurabh Verma, Seiji Yamamoto, Sharadh Ramaswamy, Shaun Lindsay, Shaun Lindsay, Sheng Feng, Shenghao Lin, Shengxin Cindy Zha, Shishir Patil, Shiva Shankar, Shuqiang Zhang, Shuqiang Zhang, Sinong Wang, Sneha Agarwal, Soji Sajuyigbe, Soumith Chintala, Stephanie Max, Stephen Chen, Steve Kehoe, Steve Satterfield, Sudarshan Govindaprasad, Sumit Gupta, Summer Deng, Sungmin Cho, Sunny Virk, Suraj Subramanian, Sy Choudhury, Sydney Goldman, Tal Remez, Tamar Glaser, Tamara Best, Thilo Koehler, Thomas Robinson, Tianhe Li, Tianjun Zhang, Tim Matthews, Timothy Chou, Tzook Shaked, Varun Vontimitta, Victoria Ajayi, Victoria Montanez, Vijai Mohan, Vinay Satish Kumar, Vishal Mangla, Vlad Ionescu, Vlad Poenaru, Vlad Tiberiu Mihailescu, Vladimir Ivanov, Wei Li, Wenchen Wang, Wenwen Jiang, Wes Bouaziz, Will Constable, Xiaocheng Tang, Xiaojian Wu, Xiaolan Wang, Xilun Wu, Xinbo Gao, Yaniv Kleinman, Yanjun Chen, Ye Hu, Ye Jia, Ye Qi, Yenda Li, Yilin Zhang, Ying Zhang, Yossi Adi, Youngjin Nam, Yu, Wang, Yu Zhao, Yuchen Hao, Yundi Qian, Yunlu Li, Yuzi He, Zach Rait, Zachary DeVito, Zef Rosnbrick, Zhaoduo Wen, Zhenyu Yang, Zhiwei Zhao, and Zhiyu Ma. 2024. The llama 3 herd of models. *Preprint*, arXiv:2407.21783.

John E Hopcroft, Rajeev Motwani, and Jeffrey D Ullman. 2001. Introduction to automata theory, languages, and computation. *Acm Sigact News*, 32(1):60–65.

James Jay Horning. 1969. *A study of grammatical inference*. Stanford University.

Dong Huang, Qingwen Bu, Jie M. Zhang, Michael Luck, and Heming Cui. 2023. Agentcoder: Multi-agent-based code generation with iterative testing and optimisation. *ArXiv*, abs/2312.13010.

Shima Imani, Liang Du, and Harsh Shrivastava. 2023. MathPrompter: Mathematical reasoning using large language models. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 5: Industry Track)*, pages 37–42, Toronto, Canada. Association for Computational Linguistics.

Albert Q. Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, Lélio Renard Lavaud, Marie-Anne Lachaux, Pierre Stock, Teven Le Scao, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. 2023a. Mistral 7b. *Preprint*, arXiv:2310.06825.

Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. 2024a. A survey on large language models for code generation. *Preprint*, arXiv:2406.00515.

Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. 2024b. A survey on large language models for code generation. *ArXiv*, abs/2406.00515.

Xue Jiang, Yihong Dong, Lecheng Wang, Qiwei Shang, and Ge Li. 2023b. Self-planning code generation with large language models. *ACM Transactions on Software Engineering and Methodology*.

S. C. Johnson and Murray Hill. 1978. Yacc: Yet another compiler-compiler.

Jushi Kai, Shengyuan Hou, Yusheng Huang, and Zhouhan Lin. 2024. Leveraging grammar induction for language understanding and generation. *Preprint*, arXiv:2410.04878.

Yixuan Li, Federico Mora, Elizabeth Polgreen, and Sanjit A Seshia. 2023. Genetic algorithms for searching a matrix of metagrammars for synthesis. *arXiv preprint arXiv:2306.00521*.

Yixuan Li, Julian Parsert, and Elizabeth Polgreen. 2024. Guiding enumerative program synthesis with large language models. In *Computer Aided Verification*, CAV 2024, pages 280–301, Cham. Springer Nature Switzerland.

Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, Tianyang Liu, Max Tian, Denis Kocetkov, Arthur Zucker, Younes Belkada, Zijian Wang, Qian Liu, Dmitry Abulkhanov, Indraneil Paul, Zhuang Li, Wen-Ding Li, Megan Risdal, Jia Li, Jian Zhu, Terry Yue Zhuo, Evgenii Zheltonozhskii, Nii Osae Osae Dade, Wenhao Yu, Lucas Krauß, Naman Jain, Yixuan Su, Xuanli He, Manan Dey, Edoardo Abati, Yekun Chai, Niklas Muennighoff, Xiangru Tang, Muhtasham Oblokulov, Christopher Akiki, Marc Marone, Chenghao Mou, Mayank Mishra, Alex Gu, Binyuan Hui, Tri Dao, Armel Zebaze, Olivier Dehaene, Nicolas Patry, Canwen Xu, Julian McAuley, Han Hu, Torsten Scholak, Sebastien Paquet, Jennifer Robinson, Carolyn Jane Anderson, Nicolas Chapados, Mostofa Patwary, Nima Tajbakhsh, Yacine Jernite, Carlos Muñoz Ferrandis, Lingming Zhang, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. 2024. Starcoder 2 and the stack v2: The next generation. *Preprint*, arXiv:2402.19173.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. 2023. Self-refine: Iterative refinement with self-feedback. In *Thirty-seventh Conference on Neural Information Processing Systems*.

MistralAI. 2024. Codestral. `https://mistral.ai/news/codestral/`.

OpenAI, Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, Red Avila, Igor Babuschkin, Suchir Balaji, Valerie Balcom, Paul Baltescu, Haiming Bao, Mohammad Bavarian, Jeff Belgum, Irwan Bello, Jake Berdine, Gabriel Bernadett-Shapiro, Christopher Berner, Lenny Bogdonoff, Oleg Boiko, Madelaine Boyd, Anna-Luisa Brakman, Greg Brockman, Tim Brooks, Miles Brundage, Kevin Button, Trevor Cai, Rosie Campbell, Andrew Cann, Brittany Carey, Chelsea Carlson, Rory Carmichael, Brooke Chan, Che Chang, Fotis Chantzis, Derek Chen, Sully Chen, Ruby Chen, Jason Chen, Mark Chen, Ben Chess, Chester Cho, Casey Chu, Hyung Won Chung, Dave Cummings, Jeremiah Currier, Yunxing Dai, Cory Decareaux, Thomas Degry, Noah Deutsch, Damien Deville, Arka Dhar, David Dohan, Steve Dowling, Sheila Dunning, Adrien Ecoffet, Atty Eleti, Tyna Eloundou, David Farhi, Liam Fedus, Niko Felix, Simón Posada Fishman, Juston Forte, Isabella Fulford, Leo Gao, Elie Georges, Christian Gibson, Vik Goel, Tarun Gogineni, Gabriel Goh, Rapha Gontijo-Lopes, Jonathan Gordon, Morgan Grafstein, Scott Gray, Ryan Greene, Joshua Gross, Shixiang Shane Gu, Yufei Guo, Chris Hallacy, Jesse Han, Jeff Harris, Yuchen He, Mike Heaton, Johannes Heidecke, Chris Hesse, Alan Hickey, Wade Hickey, Peter Hoeschele, Brandon Houghton, Kenny Hsu, Shengli Hu, Xin Hu, Joost Huizinga, Shantanu Jain, Shawn Jain, Joanne Jang, Angela Jiang, Roger Jiang, Haozhun Jin, Denny Jin, Shino Jomoto, Billie Jonn, Heewoo Jun, Tomer Kaftan, Łukasz Kaiser, Ali Kamali, Ingmar Kanitscheider, Nitish Shirish Keskar, Tabarak Khan, Logan Kilpatrick, Jong Wook Kim, Christina Kim, Yongjik Kim, Jan Hendrik Kirchner, Jamie Kiros, Matt Knight, Daniel Kokotajlo, Łukasz Kondraciuk, Andrew Kondrich, Aris Konstantinidis, Kyle Kosic, Gretchen Krueger, Vishal Kuo, Michael Lampe, Ikai Lan, Teddy Lee, Jan Leike, Jade Leung, Daniel Levy, Chak Ming Li, Rachel Lim, Molly Lin, Stephanie Lin, Mateusz Litwin, Theresa Lopez, Ryan Lowe, Patricia Lue, Anna Makanju, Kim Malfacini, Sam Manning, Todor Markov, Yaniv Markovski, Bianca Martin, Katie Mayer, Andrew Mayne, Bob McGrew, Scott Mayer McKinney, Christine McLeavey, Paul McMillan, Jake McNeil, David Medina, Aalok Mehta, Jacob Menick, Luke Metz, Andrey Mishchenko, Pamela Mishkin, Vinnie Monaco, Evan Morikawa, Daniel Mossing, Tong Mu, Mira Murati, Oleg Murk, David Mély, Ashvin Nair, Reiichiro Nakano, Rajeev Nayak, Arvind Neelakantan, Richard Ngo, Hyeonwoo Noh, Long Ouyang, Cullen O'Keefe, Jakub Pachocki, Alex Paino, Joe Palermo, Ashley Pantuliano, Giambattista Parascandolo, Joel Parish, Emy Parparita, Alex Passos, Mikhail Pavlov, Andrew Peng, Adam Perelman, Filipe de Avila Belbute Peres, Michael Petrov, Henrique Ponde de Oliveira Pinto, Michael, Pokorny, Michelle Pokrass, Vitchyr H. Pong, Tolly Powell, Alethea Power, Boris Power, Elizabeth Proehl, Raul Puri, Alec Radford, Jack Rae, Aditya Ramesh, Cameron Raymond, Francis Real, Kendra Rimbach, Carl Ross, Bob Rotsted, Henri Roussez, Nick Ryder, Mario Saltarelli, Ted Sanders, Shibani Santurkar, Girish Sastry, Heather Schmidt, David Schnurr, John Schulman, Daniel Selsam, Kyla Sheppard, Toki Sherbakov, Jessica Shieh, Sarah Shoker, Pranav Shyam, Szymon Sidor, Eric Sigler, Maddie Simens, Jordan Sitkin, Katarina Slama, Ian Sohl, Benjamin Sokolowsky, Yang Song, Natalie Staudacher, Felipe Petroski Such, Natalie Summers, Ilya Sutskever, Jie Tang, Nikolas Tezak, Madeleine B. Thompson, Phil Tillet, Amin Tootoonchian, Elizabeth Tseng, Preston Tuggle, Nick Turley, Jerry Tworek, Juan Felipe Cerón Uribe, Andrea Vallone, Arun Vijayvergiya, Chelsea Voss, Carroll Wainwright, Justin Jay Wang, Alvin Wang, Ben Wang, Jonathan Ward, Jason Wei, CJ Weinmann, Akila Welihinda, Peter Welinder, Jiayi Weng, Lilian Weng, Matt Wiethoff, Dave Willner, Clemens Winter, Samuel Wolrich, Hannah Wong, Lauren Workman, Sherwin Wu, Jeff Wu, Michael

Wu, Kai Xiao, Tao Xu, Sarah Yoo, Kevin Yu, Qiming Yuan, Wojciech Zaremba, Rowan Zellers, Chong Zhang, Marvin Zhang, Shengjia Zhao, Tianhao Zheng, Juntang Zhuang, William Zhuk, and Barret Zoph. 2024. Gpt-4 technical report. *Preprint*, arXiv:2303.08774.

Liangming Pan, Alon Albalak, Xinyi Wang, and William Yang Wang. 2023. Logic-lm: Empowering large language models with symbolic solvers for faithful logical reasoning. (arXiv:2305.12295). ArXiv:2305.12295 [cs].

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

Ricardo Wandré Dias Pedro, Fátima L. S. Nunes, and Ariane Machado-Lima. 2013. Using grammars for pattern recognition in images. *ACM Computing Surveys (CSUR)*, 46:1 – 34.

M. Richetin and F. Vernadat. 1984. Efficient regular grammatical inference for pattern recognition. *Pattern Recognit.*, 17:245–250.

Ernesto Rodrigues and Heitor Silvério Lopes. 2007. Genetic programming for induction of context-free grammars. *Seventh International Conference on Intelligent Systems Design and Applications (ISDA 2007)*, pages 297–302.

Michael Schröder and Jürgen Cito. 2022. Grammars for free: toward grammar inference for ad hoc parsers. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results*, page 41–45, Pittsburgh Pennsylvania. ACM.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik R Narasimhan, and Shunyu Yao. 2023. Reflexion: language agents with verbal reinforcement learning. In *Thirty-seventh Conference on Neural Information Processing Systems*.

Andrew Stevenson and J. Cordy. 2014a. A survey of grammatical inference in software engineering. *Sci. Comput. Program.*, 96:444–459.

Andrew Stevenson and James R. Cordy. 2014b. A survey of grammatical inference in software engineering. *Science of Computer Programming*, 96:444–459.

Weizhi Tang and Vaishak Belle. 2024. Tom-lm: Delegating theory of mind reasoning to external symbolic executors in large language models. *arXiv preprint arXiv:2404.15515*.

Gemma Team, Morgane Riviere, Shreya Pathak, Pier Giuseppe Sessa, Cassidy Hardin, Surya Bhupatiraju, Léonard Hussenot, Thomas Mesnard, Bobak Shahriari, Alexandre Ramé, Johan Ferret, Peter Liu, Pouya Tafti, Abe Friesen, Michelle Casbon, Sabela Ramos, Ravin Kumar, Charline Le Lan, Sammy Jerome, Anton Tsitsulin, Nino Vieillard, Piotr Stanczyk, Sertan Girgin, Nikola Momchev, Matt Hoffman, Shantanu Thakoor, Jean-Bastien Grill, Behnam Neyshabur, Olivier Bachem, Alanna Walton, Aliaksei Severyn, Alicia Parrish, Aliya Ahmad, Allen Hutchison, Alvin Abdagic, Amanda Carl, Amy Shen, Andy Brock, Andy Coenen, Anthony Laforge, Antonia Paterson, Ben Bastian, Bilal Piot, Bo Wu, Brandon Royal, Charlie Chen, Chintu Kumar, Chris Perry, Chris Welty, Christopher A. Choquette-Choo, Danila Sinopalnikov, David Weinberger, Dimple Vijaykumar, Dominika Rogozińska, Dustin Herbison, Elisa Bandy, Emma Wang, Eric Noland, Erica Moreira, Evan Senter, Evgenii Eltyshev, Francesco Visin, Gabriel Rasskin, Gary Wei, Glenn Cameron, Gus Martins, Hadi Hashemi, Hanna Klimczak-Plucińska, Harleen Batra, Harsh Dhand, Ivan Nardini, Jacinda Mein, Jack Zhou, James Svensson, Jeff Stanway, Jetha Chan, Jin Peng Zhou, Joana Carrasqueira, Joana Iljazi, Jocelyn Becker, Joe Fernandez, Joost van Amersfoort, Josh Gordon, Josh Lipschultz, Josh Newlan, Ju yeong Ji, Kareem Mohamed, Kartikeya Badola, Kat Black, Katie Millican, Keelin McDonell, Kelvin Nguyen, Kiranbir Sodhia, Kish Greene, Lars Lowe Sjoesund, Lauren Usui, Laurent Sifre, Lena Heuermann, Leticia Lago, Lilly McNealus, Livio Baldini Soares, Logan Kilpatrick, Lucas Dixon, Luciano Martins, Machel Reid, Manvinder Singh, Mark Iverson, Martin Görner, Mat Velloso, Mateo Wirth, Matt Davidow, Matt Miller, Matthew Rahtz, Matthew Watson, Meg Risdal, Mehran Kazemi, Michael Moynihan, Ming Zhang, Minsuk Kahng, Minwoo Park, Mofi Rahman, Mohit Khatwani, Natalie Dao, Nenshad Bardoliwalla, Nesh Devanathan, Neta Dumai, Nilay Chauhan, Oscar Wahltinez, Pankil Botarda, Parker Barnes, Paul Barham, Paul Michel, Pengchong Jin, Petko Georgiev, Phil Culliton, Pradeep Kuppala, Ramona Comanescu, Ramona Merhej, Reena Jana, Reza Ardeshir Rokni, Rishabh Agarwal, Ryan Mullins, Samaneh Saadat, Sara Mc Carthy, Sarah Cogan, Sarah Perrin, Sébastien M. R. Arnold, Sebastian Krause, Shengyang Dai, Shruti Garg, Shruti Sheth, Sue Ronstrom, Susan Chan, Timothy Jordan, Ting Yu, Tom Eccles, Tom Hennigan, Tomas Kocisky, Tulsee Doshi, Vihan Jain, Vikas Yadav, Vilobh Meshram, Vishal Dharmadhikari, Warren Barkley, Wei Wei, Wenming Ye, Woohyun Han, Woosuk Kwon, Xiang Xu, Zhe Shen, Zhitao Gong, Zichuan Wei, Victor Cotruta, Phoebe Kirk, Anand Rao, Minh Giang, Ludovic Peran, Tris Warkentin, Eli Collins, Joelle Barral, Zoubin Ghahramani, Raia Hadsell, D. Sculley, Jeanine Banks, Anca Dragan, Slav Petrov, Oriol Vinyals, Jeff Dean, Demis Hassabis, Koray Kavukcuoglu, Clement Farabet, Elena Buchatskaya, Sebastian Borgeaud, Noah Fiedel, Armand Joulin, Kathleen Kenealy, Robert Dadashi, and Alek Andreev. 2024. Gemma 2: Improving open language models at a practical size. *Preprint*, arXiv:2408.00118.

Brandon T. Willard and Rémi Louf. 2023. Efficient guided generation for large language models. *Preprint*, arXiv:2307.09702.

A Dataset Construction

To evaluate the capacity of LLMs in few-shot grammar generation, we present a dedicated dataset. We explain the details of the construction process in this section.

For clear explanation, let $\mathfrak{G}^{ref} = \bigcup_{k=1}^{K} \mathbb{G}_k^{ref}$ be a set of reference grammars, where $\mathbb{G}_k^{ref}$ is a set of reference grammars in which each reference grammar G_k^{ref} having exactly k non-terminals and thus its $|R| = k$. Each set of reference grammars $\mathbb{G}_k^{ref} = \{G_{k,1}^{ref}, G_{k,2}^{ref}, \ldots, G_{k,n}^{ref}\}$ contains n reference grammar with k non-terminals.

Initially, we constructed $\mathfrak{G}^{ref} = \bigcup_{k=1}^{9} \mathbb{G}_k^{ref}$. For each number k, we prompted GPT-4o to produce $n = 10$ reference grammars to yield $\mathbb{G}_k^{ref} = \{G_{k,1}^{ref}, G_{k,2}^{ref}, \ldots, G_{k,10}^{ref}\}$, with the prompt template demonstrated in Prompt Template 3. In Prompt Template 3, k means the placeholder of the number of non-terminals, and n means the number of reference grammars needed to be produced. However, GPT-4o failed to consistently generate reference grammars in the correct syntax or with the correct number of non-terminals, especially as the number of non-terminals increased. To ensure that the generated reference grammars are syntactically correct and have the correct number of non-terminals, we used a BNF parser to do verification. It takes a G^{ref} and checks whether $valid(G^{ref})$ is true and whether it has the required number of non-terminals. Any reference grammar that is not valid or has a wrong number of non-terminals were manually corrected. For duplicated reference grammars, we prompted GPT-4o to generate an alternative. This resulted in 90 reference grammars (i.e., $|\mathfrak{G}^{ref}| = |\bigcup_{k=1}^{9} \mathbb{G}_k^{ref}| = 90$), which are the reference grammars used to generate positive and negative examples subsequently.

For each reference grammar $G^{ref} \in \mathfrak{G}^{ref}$, we prompted GPT-4o to generate 6 various challenges. For each challenge, GPT-4o is prompted by Prompt Template 6 and Prompt Template 4 to produce a set of 3 positive examples ($\mathcal{P} \subseteq \mathcal{L}(G^{ref})$), and a set of three negative examples ($\mathcal{N} \cap \mathcal{L}(G^{ref}) = \emptyset$), respectively. In both Prompt Template 6 and Prompt Template 4, m means the number of examples needed to be generated, and *reference_grammar* means the given reference grammar by which the generated examples should be accepted or rejected. However, we observed that GPT-4o frequently failed to produce valid positive and negative examples, leading to either $\mathcal{P} \not\subseteq \mathcal{L}_(G^{ref})$ or $\mathcal{N} \cap \mathcal{L}_(G^{ref}) \neq \emptyset$, or both. The number of failures tends to increase as the number of non-terminals of G^{ref} increases. To ensure the correctness of the generated challenges, we used a BNF parser to verify whether all given positive examples and negative examples can be accepted and rejected, respectively, by their corresponding G^{ref}. Erroneous positive and negative examples were manually corrected. Ultimately, we obtained a dataset consisting of a total of 540 challenges.

We visually summarize the dataset construction procedure in Figure 2.

B Grammar Quality Metrics

This section covers the formal definitions of the grammar quality metrics.

First define $\Pi_{\mathcal{P}} \subseteq \Pi$ to be the set of production rules that are used in the left-most derivations of all positive examples in $\mathcal{P}$. That is, the set of rules in Π which occur in a sequence of rules $S \to \alpha_1 \to \ldots \to \alpha_n \to p$ where $p \in \mathcal{P}$, and all rules expand the left-most non-terminal in $\alpha_1, \ldots, \alpha_n$.

Let Π^* be the production rules in G^* and Π^{ref} be the production rules in G^{ref}. Let us define $\mathit{Diff}(G^{ref}, G^*) = |\Pi_{\mathcal{P}}^{ref}| - |\Pi_{\mathcal{P}}^*|$. The average difference in production rules used over the whole set of k solved challenges, C', is given by:

$$\mathit{Diff}^{\diamond} = \frac{1}{k} \sum_{i=1}^{k} \mathit{Diff}(G_i^{ref}, G_i^*).$$

We define two indicator functions, which indicate when a grammar uses substantially fewer rules than the reference grammar, and substantially more rules than the reference grammar:

$$\mathbb{I}_{OF}(G^{ref}, G^*, \mathcal{P}) = \begin{cases} 1 & \text{if } |\Pi_{\mathcal{P}}^{ref}| - |\Pi_{\mathcal{P}}^*| \\ & \quad > \frac{|\Pi_{\mathcal{P}}^{ref}|}{2} \\ 0 & \text{otherwise.} \end{cases}$$

$$\mathbb{I}_{OG}(G^{ref}, G^*, \mathcal{P}) = \begin{cases} 1 & \text{if } |\Pi_{\mathcal{P}}^{ref}| - |\Pi_{\mathcal{P}}^*| \\ & \quad < -\frac{|\Pi_{\mathcal{P}}^{ref}|}{2} \\ 0 & \text{otherwise.} \end{cases}$$

The metric to estimate whether the generated grammars overfit the examples, is then given by

$$OF(C') = \frac{1}{k} \sum_{i=1}^{k} \mathbb{I}_{OF}(G_i^{ref}, G_i^*, \mathcal{P}_i).$$

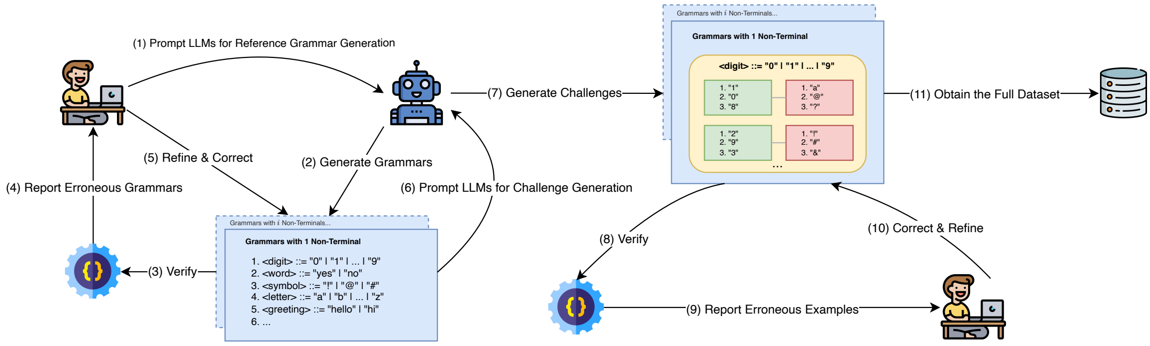

Figure 2: The Dataset Construction Process: (1) GPT-4o is prompted with Prompt Template 3 to generate a set of reference grammars; (2) A set of reference grammars $\mathfrak{G}^{ref} = \bigcup_{k=1}^{9} \mathbb{G}_k^{ref}$ are generated by LLMs; (3) A BNF parser is used to check the correctness of each generated reference grammar; (4) Erroneous reference grammars are reported to humans; (5) Reported reference grammars are modified and corrected manually; (6) GPT-4o is prompted with Prompt Template 6 and Prompt Template 4 to generate challenges for each reference grammars; (7) Challenges are generated by LLMs, in which each challenge consists of 3 positive and 3 negative examples; (8) A BNF parser is used to verify whether positive and negative examples are accepted and rejected by their corresponding reference grammar respectively; (9) Erroneous challenges are reported to humans; (10) Reported challenges are corrected manually; (11) The final dataset consisting of 540 challenges are obtained.

The metric to estimate whether the generated grammars overgeneralize the examples, OG, is given as:

$$OG(C') = \frac{1}{k} \sum_{i=1}^{k} \mathbb{I}_{OG}(G_i^{ref}, G_i^*, \mathcal{P}_i).$$

In addition, the TU metric, for a given challenge, to measure the percentage of $|\Pi^*|$ taken up by $|\Pi^*_{\mathcal{P}}|$, indicating the utility of G^*, is given as:

$$TU(G^*, \mathcal{P}) = \frac{|\Pi^*_{\mathcal{P}}|}{|\Pi^*|}$$

for which lower TU indicates a bunch of irrelevant or nonsensical production rules of G^* while higher TU indicates the opposite. The average utility over C' is given by:

$$TU^{\diamond} = \frac{1}{k} \sum_{i=1}^{k} TU(G_i^*, \mathcal{P})$$

C Direct Prompting

For the DP approach, Prompt Template 1 is used to prompt LLMs to directly generate a grammar with a given set of positive and negative examples. In Prompt Template 1, *positive_examples* and *negative_examples* are placeholders for a set of positive and negative examples. In addition, it also specifies a list of requirements LLMs should take care of and obey when generating grammars.

D Optimization of BNF Parser for Providing LLM-Friendly Feedback

For the OPF approach, we use the same prompt template used in DP, as shown in Prompt Template 1, to prompt LLMs to generate an initial grammar. Then, Prompt Template 5 is used in the iterations of the feedback loop to construct prompts from feedback offered by the BNF parser to LLMs. In Prompt Template 5, *positive_examples* and *negative_examples* are the placeholders for a set of positive and negative examples, *bnf_grammar* means the previously generated erroneous grammar, and *parser_feedback* is the placeholder for feedback provided from the BNF parser.

Furthermore, for each feedback given by the BNF parser, in addition to giving essential feedback such as notifying the line number for the place the error occurs, we optimize the BNF parser to also provide LLM-friendly feedback to LLMs, such as the possible reasons for the error or ways to fix it. We have shown some of them in Parser Feedback 1 and Parser Feedback 2.

In addition, OPF includes a parameter called *max_turns*, which specifies the maximum number of feedback iterations. If an LLM can generate a valid grammar based on earlier feedback, the algorithm stops early; otherwise, it continues until reaching the specified maximum.

Moreover, it is worth noting that this approach does not aim to follow every component of Reflexion or Self-Refine strictly. For instance, it does not maintain a long-term context or external memory. Instead, it uses only the most recent feedback in each turn to guide self-refinement. Concretely, in each feedback iteration, an LLM is provided with the previously generated erroneous grammar, corresponding positive and negative examples, and the latest feedback, to produce revised grammars. Therefore, since, in each iteration, an LLM may produce similar and even the same grammar as the previous ones, especially when they fail to fix the previous errors leading to the same feedback provided by the parser, we set the temperature to 0.3 to expect to enable LLMs to generate more diverse grammars even for encountering the same feedback, to optimize the performance. This approach highlights the optimization from the perspective of the parser to provide more LLM-friendly feedback. However, due to limited space and the trivial-yet-complex optimization process, for the details of the optimization of the BNF parser, please refer to our source code and the comments.

E LLM-Driven Hybrid Genetic Algorithm

The pseudocode of HyGenar is presented in Algorithm 1, with detailed descriptions of its primary functions including $\mathrm{Fitness}$, Select, Cross, and Mutate provided in Section 5. In Algorithm 1, we note $\mathrm{Fitness}$ function as FITNESS, Select function as SELECT, Cross function as CROSS, and Mutate function as MUTATE.

As shown in Algorithm 1, it takes seven parameters: $\mathcal{P}$ and $\mathcal{N}$ means a set of positive and negative examples respectively, k indicates *population size*, g represents *generations* which means the number of iterations of evolution, ρ means the *crossover rate*, μ means the *mutation rate*, and LLM means an LLM which takes a prompt and returns a response. In addition, PROMPTGENERATOR means to generate a prompt from the prompt template shown in Prompt Template 1. MAXFITNESSSCORE is a constant indicating the highest fitness score any candidate grammar can achieve and due to each challenge in the constructed dataset only having 3 positive and 3 negative examples, the highest fitness score is 6. We thus set MAXFITNESSSCORE to 6.

In addition, for *LLMMut*, we have shown the prompt template used to prompt LLMs to mutate a given grammar in Prompt Template 2, in which *bnf_grammar* means the placeholder for a candidate grammar while *positive_examples* and *negative_examples* means a set of positive and negative examples respectively.

F Additional Results

In addition to the results shown in Table 1, we have shown 7 more results and discussed them respectively in subsections F.1, F.2, F.3, F.4, and F.5.

F.1 Results for C_1, C_2, and C_3

Table 4 presents the results categorized into three subsets: C_1, C_2, and C_3. The subset C_1 includes challenges where the reference grammars have $1 \sim 3$ non-terminals, C_2 are those with $4 \sim 6$ non-terminals, and C_3 consists of challenges with $7 \sim 9$ non-terminals. Therefore, it aims to demonstrate and analyze the performance of LLMs as the number of non-terminals increases.

As the results are shown in Table 4, as the number of non-terminals increases, both SX and SE decrease across all LLMs. While DP and OPF exhibit suboptimal and unsatisfactory performance, HyGenar consistently demonstrates and contributes substantial improvements across most LLMs, even as the number of non-terminals increases. For example, in the case of *GPT-4o*, HyGenar increases the SE by 21% compared to DP and OPF on C_3. Similarly, with *GPT-3.5-Turbo*, compared to DP and OPF, HyGenar improves the SE by 30% and 28% on C_2 and 20% and 19% on C_3, respectively.

F.2 Results for P_1, P_2, and P_3

Table 5 demonstrates the results grouped into three subsets: P_1, P_2, and P_3. The subset P_1 consists of challenges where the reference grammars have $1 \sim 6$ production rules, P_2 includes those with $7 \sim 15$ production rules, and P_3 consists of challenges with the number of production rules greater than 16. Therefore, it aims to show and analyze the performance of LLMs as the number of production rules increases.

Similar to the results of C_1, C_2, and C_3, as the number of production rules increases, both SX and SE decrease across all LLMs. Nevertheless, HyGenar can still steadily improve both SE and SX, even as the number of production rules increases.

F.3 Results for $Diff$, OF, and OG Metrics

To investigate whether LLMs generate grammars in an overfitted manner and whether HyGenar improves performance through overfitting, as well as

Algorithm 1 HyGenar

```
procedure GenerateGrammar(P, N, k, g, ρ, μ, LLM)
    population ← []                                          ▷ Initialize population as an empty list
    G*.best ← null                                           ▷ Keep track of overall best grammar
    fitness.best ← −1                                        ▷ Track highest fitness found so far
    for i ← 1 to k do
        prompt ← PromptGenerator                             ▷ Use prompt template from Prompt Template 1
        G* ← LLM(prompt)                                     ▷ Use LLM to get an initial candidate
        score ← Fitness(G*, P, N)                            ▷ Compute fitness
        if score = MaxFitnessScore then
            return G*                                        ▷ Return early if perfect score is achieved
        population ← population ∥ [G*]                       ▷ Add candidate to population
    for i ← 1 to g do
        fitnessScores ← []
        for G* ∈ population do
            score ← Fitness(G*, P, N)
            fitnessScores ← fitnessScores ∥ [(score, G*)]    ▷ Add a tuple of score and grammar
            if score > fitness.best then
                fitness.best ← score
                G*.best ← G*
        if fitness.best = MaxFitnessScore then
            return G*.best                                   ▷ Return the best grammar found
        𝔾 = [G* | (score, G*) ∈ fitnessScores]
        S = [score | (score, G*) ∈ fitnessScores]
        selected ← Select(𝔾, S)
        population.new ← []
        while |population.new| < k do
            G*_a, G*_b ← RandomChoice(selected)
            G* ← cross(G*_a, G*_b, ρ)
            if Uniform(0, 1) < μ then
                G* ← mutate(G*, P, N, LLM)
            population.new ← population.new ∥ [G*]
        population ← population.new                          ▷ Proceed to the next generation
    return G*.best
```

to examine whether LLMs and our method produce overly generalized grammars, we employed the three evaluation metrics: *Diff*, *OF*, *OG*. The results are presented in Table 6, 7, and 8, respectively. The notation "N/A" indicates inapplicability. Since these three metrics are only applicable when a grammar possesses correct semantics, "N/A" thus signifies that no grammar in the evaluation set exhibits correct semantics.

Through the *Diff* metric, as shown in Tabel 6, we observe that the number of production rules used in derivations of the generated grammars and the reference grammars does not differ significantly on average. However, as the number of production rules in the reference grammar increases, the $Diff^{\diamond}$ exhibits a slight upward trend. The $Diff^{\diamond}$ is almost always positive across most LLMs and methods, which indicates that, in most cases, the number of production rules used by the generated grammars is lower than that of the reference grammars.

Furthermore, Table 7 presents the *OF* metric. For some models, particularly *GPT-3.5-Turbo*, an increasing number of production rules corresponds to a certain degree of overfitting. Nevertheless, on average, most LLMs do not exhibit significant overfitting. Additionally, we observe that in HyGenar, the *OF* metric does not show a significant difference compared to the baselines, indicating that HyGenar does not improve performance through overfitting.

Additionally, we present the OG metric in Table 8. We observed that, for some models, such as *Qwen:72b-Instruct*, as the number of production rules in the reference grammar increases, they tend to generate overly generalized grammars. Nevertheless, on average, most LLMs do not tend to generate overgeneralized grammars.

F.4 Results for TU Metrics

To investigate whether LLMs generate irrelevant production rules, we employ the TU metric. For example, given the challenge shown in Figure 1, a generated grammar might be:

```
<stmt> ::= <func> "(" <args> ")"
<args> ::= <expr> | <expr> "," <args>
<expr> ::= <char> | <number>
<func> ::= <char> <func> | <char>
<char> ::= "a" | ... | "z"
<number> ::= "0" | ... | "9"
<hello> ::= "hello"
<world> ::= "world"
```

in which:

```
<hello> ::= "hello"
<world> ::= "world"
```

are two irrelevant production rules.

As shown in Table 9, on average, across both baselines and in HyGenar, TU remains relatively high, indicating that LLMs do not tend to produce irrelevant production rules. However, as the number of production rules increases, TU shows a tendency of declination. Nevertheless, this does not imply that LLMs generate more irrelevant rules. Considering the results from OG, we think this decrease may more likely be attributable to the generated grammar becoming more generalized.

F.5 Results for Robustness Evaluation

Since HyGenar requires setting the temperature greater than 0 which we set to 0.7, we repeated 5 independent experiments for both *GPT-4o* and *GPT-3.5-Turbo* to ensure the temperature does not affect the results significantly.

The results are demonstrated in Table 3, in which each row means the results of one experiment. The averages of syntax correctness of *GPT-4o* and *GPT-3.5-Turbo* are 95.8% and 98.6% and the standard deviations are 0.4% and 0.49%, respectively. The averages of semantic correctness of *GPT-4o* and *GPT-3.5-Turbo* are 93.2% and 61.6% and the standard deviations are 0.4% and 0.49%. Therefore, it indicates that although we set the temperature to 0.7 in HyGenar, the fluctuation of both syntax correctness and semantic correctness are very slight and the performance across multiple experiments stays steady. Thus, the results demonstrated the robustness of our proposed method, HyGenar.

Prompt Template 1: Generate a Grammar Directly with a Given Set Positive and Negative Examples

Given a set of positive and negative examples, generate the Backus–Naur Form (BNF) grammar that accepts all positive examples and rejects all negative examples.
1. Only generate the standard BNF grammar;
2. The generated BNF grammar MUST accept all positive examples and reject all negative examples;
3. Each terminal symbol MUST be quoted with double quotes and MUST NOT escape double quotes or pipeline in terminal symbols;
4. Pay special attention to whether spaces, line breaks, or other special symbols are required between each symbol, and if so, these need to be explicitly specified, e.g. <term> ::= "1" "+" "2" can handle "1+2" but not "1 + 2" while <term> ::= "1" " " "+" " " "2" can handle "1 + 2" but not "1+2";
5. The entry point of the generated BNF grammar MUST be the non-terminal symbol in the first production rule;
6. Only the generated BNF should be wrapped in a pair of triple backtick;
7. Do NOT output any additional texts, comments, or explanations.

===Positive Examples===
{*positive_examples*}
===Negative Examples===
{*negative_examples*}

Experiment	*SX*	*SE*
	GPT-4o	
1st	95	93
2nd	96	93
3rd	96	94
4th	96	93
5th	96	93
	GPT-3.5-Turbo	
1st	98	62
2nd	98	62
3rd	99	62
4th	99	61
5th	99	61

Table 3: Results of Syntax and Semantic Correctness for HyGenar with *GPT-4o* and *GPT-3.5-Turbo* on Grammar Generation by Conducting 5 Independent Experiments (%)

Parser Feedback 1: Invalid Production Rule

This error is likely due to not satisfying one of the following requirements:
1. A rule MUST start with a non-terminal definition;
2. A non-terminal symbol MUST be in angle brackets, e.g. <non-terminal>;
3. A non-terminal definition must be followed by '::=' to indicate the start of the right-hand side;

Prompt Template 2: LLM-Driven Mutation

Modify the following BNF grammar slightly to improve its acceptance of the positive examples and rejection of the negative examples.

===BNF Grammar===
{*bnf_grammar*}

===Positive Examples===
{*positive_examples*}
===Negative Examples===
{*negative_examples*}

Only output the modified BNF grammar wrapped in triple backticks.

Prompt Template 3: Generate Grammars

Generate a list of random standard Backus-Naur Form (BNF) grammar with the following constraints:
1. Each generated BNF grammar MUST be SELF-CONTAINED and VALID, which means it should be able to recognize a valid string;
2. Each generated BNF grammar MUST have exactly $\{k\}$ lines;
3. Each generated BNF grammar MUST be unique;
4. Each generated BNF grammar MUST be separated by a newline in addition to the linebreak;
5. For each generated BNF grammar, the entry point MUST be at the first line;
6. Only generate $\{n\}$ BNF grammars;
7. Only output BNF grammars WITHOUT any additional text or code block, like "```".

Prompt Template 4: Generate Negative Examples with a Given Grammar

Generate a list of negative examples with the following constraints:
1. Each example MUST be separated by a newline in addition to the linebreak;
2. Only output examples WITHOUT any additional text or code block, like "```";
3. Only output $\{m\}$ examples;
4. Each example MUST be generated based on the given BNF grammar;
5. Each example should be greatly related to the given BNF grammar, but ensure it is NOT a valid string for the given BNF grammar.
For example, given the following BNF grammar:
<term> ::= "0" | "1" | "2"
you should output negative examples like:
6

*

9

Then, the given BNF grammar is:
{*reference_grammar*}

Challenge Set	SX_{DP}	SX_{OPF}	$SX_{\textbf{HyGenar}}$	SE_{DP}	SE_{OPF}	$SE_{\textbf{HyGenar}}$
GPT-4o						
C_1	100	100	100	99	99	100 $\uparrow 1$ $\uparrow 1$
C_2	100	100	100	93	95 $\uparrow 2$	100 $\uparrow 7$ $\uparrow 5$
C_3	79	92 $\uparrow 13$	87 $\uparrow 8$ $\downarrow 5$	59	59	80 $\uparrow 21$ $\uparrow 21$
All	93	97 $\uparrow 4$	96 $\uparrow 3$ $\downarrow 1$	84	85 $\uparrow 1$	93 $\uparrow 9$ $\uparrow 8$
GPT-3.5-Turbo						
C_1	98	97 $\downarrow 1$	100 $\uparrow 2$ $\uparrow 3$	72	71 $\downarrow 1$	93 $\uparrow 21$ $\uparrow 22$
C_2	98	99 $\uparrow 1$	100 $\uparrow 2$ $\uparrow 1$	28	30 $\uparrow 2$	58 $\uparrow 30$ $\uparrow 28$
C_3	84	90 $\uparrow 6$	96 $\uparrow 12$ $\uparrow 6$	11	12 $\uparrow 1$	31 $\uparrow 20$ $\uparrow 19$
All	94	95 $\uparrow 1$	99 $\uparrow 5$ $\uparrow 4$	37	38 $\uparrow 1$	61 $\uparrow 24$ $\uparrow 23$
Qwen:72b-Chat						
C_1	73	76 $\uparrow 3$	96 $\uparrow 23$ $\uparrow 20$	52	53 $\uparrow 1$	76 $\uparrow 24$ $\uparrow 23$
C_2	48	48	77 $\uparrow 29$ $\uparrow 29$	6	8 $\uparrow 2$	26 $\uparrow 20$ $\uparrow 18$
C_3	20	23 $\uparrow 3$	56 $\uparrow 36$ $\uparrow 33$	1	2 $\uparrow 1$	11 $\uparrow 10$ $\uparrow 9$
All	47	49 $\uparrow 2$	76 $\uparrow 29$ $\uparrow 27$	20	21 $\uparrow 1$	38 $\uparrow 18$ $\uparrow 17$
Llama3:70b-Instruct						
C_1	88	90 $\uparrow 2$	97 $\uparrow 9$ $\uparrow 7$	78	77 $\downarrow 1$	94 $\uparrow 16$ $\uparrow 17$
C_2	54	60 $\uparrow 6$	76 $\uparrow 22$ $\uparrow 16$	31	35 $\uparrow 4$	61 $\uparrow 30$ $\uparrow 26$
C_3	28	34 $\uparrow 6$	52 $\uparrow 24$ $\uparrow 18$	15	14 $\downarrow 1$	29 $\uparrow 14$ $\uparrow 15$
All	57	61 $\uparrow 4$	75 $\uparrow 18$ $\uparrow 14$	41	42 $\uparrow 1$	61 $\uparrow 20$ $\uparrow 19$
Gemma2:27b-Instruct						
C_1	99	100 $\uparrow 1$	100 $\uparrow 1$	91	92 $\uparrow 1$	98 $\uparrow 7$ $\uparrow 6$
C_2	97	97	99 $\uparrow 2$ $\uparrow 2$	49	49	84 $\uparrow 35$ $\uparrow 35$
C_3	76	79 $\uparrow 3$	93 $\uparrow 17$ $\uparrow 14$	26	29 $\uparrow 3$	54 $\uparrow 28$ $\uparrow 25$
All	91	92 $\uparrow 1$	98 $\uparrow 7$ $\uparrow 6$	56	57 $\uparrow 1$	79 $\uparrow 23$ $\uparrow 22$
Mistral:7b-Instruct						
C_1	1	25 $\uparrow 24$	3 $\uparrow 2$ $\downarrow 22$	0	17 $\uparrow 17$	2 $\uparrow 2$ $\downarrow 15$
C_2	1	20 $\uparrow 19$	1 $\downarrow 19$	1	6 $\uparrow 5$	0 $\downarrow 1$ $\downarrow 6$
C_3	1	11 $\uparrow 10$	0 $\downarrow 1$ $\downarrow 11$	0	1 $\uparrow 1$	0 $\downarrow 1$
All	1	19 $\uparrow 18$	1 $\downarrow 18$	0	8 $\uparrow 8$	1 $\uparrow 1$ $\downarrow 7$
Codestral:22b						
C_1	82	96 $\uparrow 14$	99 $\uparrow 17$ $\uparrow 3$	82	92 $\uparrow 10$	98 $\uparrow 16$ $\uparrow 6$
C_2	53	77 $\uparrow 24$	86 $\uparrow 33$ $\uparrow 9$	36	45 $\uparrow 9$	69 $\uparrow 33$ $\uparrow 24$
C_3	23	39 $\uparrow 16$	57 $\uparrow 34$ $\uparrow 18$	15	19 $\uparrow 4$	33 $\uparrow 18$ $\uparrow 14$
All	53	71 $\uparrow 18$	80 $\uparrow 27$ $\uparrow 9$	44	52 $\uparrow 8$	67 $\uparrow 23$ $\uparrow 15$
Starcoder2:15b-Instruct						
C_1	97	68 $\downarrow 29$	100 $\uparrow 3$ $\uparrow 32$	67	42 $\downarrow 25$	84 $\uparrow 17$ $\uparrow 42$
C_2	73	65 $\downarrow 8$	99 $\uparrow 26$ $\uparrow 34$	14	12 $\downarrow 2$	31 $\uparrow 17$ $\uparrow 19$
C_3	58	48 $\downarrow 10$	94 $\uparrow 36$ $\uparrow 46$	11	7 $\downarrow 4$	17 $\uparrow 6$ $\uparrow 10$
All	76	60 $\downarrow 16$	98 $\uparrow 22$ $\uparrow 38$	30	20 $\downarrow 10$	44 $\uparrow 14$ $\uparrow 24$

Table 4: Averages of Syntax and Semantic Correctness Grouped in C_1, C_2, and C_3 (%)

Challenge Set	SX_{DP}	SX_{OPF}	$SX_{\textbf{HyGenar}}$	SE_{DP}	SE_{OPF}	$SE_{\textbf{HyGenar}}$
GPT-4o						
P_1	100	100	100	99	99	100 ↑1 ↑1
P_2	100	100	100	96	95 ↓1	100 ↑4 ↑5
P_3	81	93 ↑12	89 ↑8 ↓4	62	64 ↑2	82 ↑20 ↑18
All	93	97 ↑4	96 ↑3 ↓1	84	85 ↑1	93 ↑9 ↑8
GPT-3.5-Turbo						
P_1	98	97 ↓1	100 ↑2 ↑3	69	67 ↓2	93 ↑24 ↑26
P_2	99	100 ↑1	100 ↑1	18	22 ↑4	42 ↑24 ↑20
P_3	86	90 ↑4	96 ↑10 ↑6	23	24 ↑1	46 ↑23 ↑22
All	94	95 ↑1	99 ↑5 ↑4	37	38 ↑1	61 ↑24 ↑23
Qwen:72b-Chat						
P_1	72	74 ↑2	96 ↑24 ↑22	42	43 ↑1	69 ↑27 ↑26
P_2	51	51	76 ↑25 ↑25	12	13 ↑1	24 ↑12 ↑11
P_3	21	25 ↑4	59 ↑38 ↑34	6	7 ↑1	21 ↑15 ↑14
All	47	49 ↑2	76 ↑29 ↑27	20	21 ↑1	38 ↑18 ↑17
Llama3:70b-Instruct						
P_1	86	90 ↑4	97 ↑11 ↑7	73	73	92 ↑19 ↑19
P_2	63	69 ↑6	86 ↑23 ↑17	38	40 ↑2	67 ↑29 ↑27
P_3	26	30 ↑4	47 ↑21 ↑17	15	15	29 ↑14 ↑14
All	57	61 ↑4	75 ↑18 ↑14	41	42 ↑1	61 ↑20 ↑19
Gemma2:27b-Instruct						
P_1	99	100 ↑1	100 ↑1	87	88 ↑1	98 ↑11 ↑10
P_2	99	99	100 ↑1 ↑1	48	47 ↓1	84 ↑36 ↑37
P_3	77	80 ↑3	94 ↑17 ↑14	33	37 ↑4	58 ↑25 ↑21
All	91	92 ↑1	98 ↑7 ↑6	56	57 ↑1	79 ↑23 ↑22
Mistral:7b-Instruct						
P_1	2	26 ↑24	3 ↑2 ↓23	1	15 ↑14	2 ↑1 ↓13
P_2	0	19 ↑19	1 ↑1 ↓18	0	5 ↑5	0 ↓5
P_3	0	13 ↑13	0 ↓13	0	3 ↑3	0 ↓3
All	1	19 ↑18	1 ↓18	0	8 ↑8	1 ↑1 ↓7
Codestral:22b						
P_1	82	97 ↑15	99 ↑17 ↑2	79	89 ↑10	97 ↑18 ↑8
P_2	51	73 ↑22	80 ↑29 ↑7	35	38 ↑3	66 ↑31 ↑28
P_3	29	47 ↑18	64 ↑35 ↑17	21	30 ↑9	41 ↑20 ↑11
All	53	71 ↑18	80 ↑27 ↑9	44	52 ↑8	67 ↑23 ↑15
Starcoder2:15b-Instruct						
P_1	96	68 ↓28	100 ↑4 ↑32	54	34 ↓20	74 ↑20 ↑40
P_2	65	59 ↓6	99 ↑34 ↑40	16	10 ↓6	26 ↑10 ↑16
P_3	67	55 ↓12	95 ↑28 ↑40	20	15 ↓5	31 ↑11 ↑16
All	76	60 ↓16	98 ↑22 ↑38	30	20 ↓10	44 ↑14 ↑24

Table 5: Averages of Syntax and Semantic Correctness Grouped in P_1, P_2, and P_3 (%)

Challenge Set	$Diff^{\diamond}_{DP}$	$Diff^{\diamond}_{OPF}$	$Diff^{\diamond}_{\textbf{HyGenar}}$
		GPT-4o	
P_1	0.22	0.22	0.21
P_2	1.38	1.32	1.18
P_3	4.37	3.95	3.36
All	1.76	1.63	1.65
		GPT-3.5-Turbo	
P_1	0.42	0.37	0.27
P_2	2.43	2.34	2.21
P_3	3.38	3.42	4.30
All	1.40	1.43	1.81
		Qwen:72b-Chat	
P_1	0.12	0.08	-0.02
P_2	0.32	0.38	0.38
P_3	2.33	3.00	3.02
All	0.04	0.50	0.68
		Llama3:70b-Instruct	
P_1	0.39	0.38	0.44
P_2	1.56	1.62	1.50
P_3	2.52	2.42	2.85
All	1.00	1.00	1.21
		Gemma2:27b-Instruct	
P_1	0.55	0.54	0.59
P_2	1.77	1.74	1.50
P_3	4.03	4.27	4.40
All	1.64	1.74	1.93
		Mistral:7b-Instruct	
P_1	1.00	-0.59	0.00
P_2	N/A	1.88	N/A
P_3	N/A	3.17	N/A
All	1.00	0.44	0.00
		Codestral:22b	
P_1	0.18	0.16	0.10
P_2	1.26	1.17	0.75
P_3	1.86	2.34	2.96
All	0.72	0.85	0.94
		Starcoder2:15b-Instruct	
P_1	0.39	0.41	0.20
P_2	0.40	0.62	0.71
P_3	2.90	3.13	4.11
All	1.02	1.22	1.32

Table 6: Averages of $Diff^{\diamond}$ Grouped in P_1, P_2, and P_3

Challenge Set	OF_{DP}	OF_{OPF}	$OF_{\textbf{HyGenar}}$
		GPT-4o	
P_1	0.00	0.00	0.00
P_2	4.03	0.68	1.28
P_3	13.49	11.54	11.31
All	5.08	3.50	4.17
		GPT-3.5-Turbo	
P_1	0.00	0.00	0.00
P_2	35.71	34.29	24.24
P_3	17.02	16.67	24.47
All	9.05	9.85	11.89
		Qwen:72b-Chat	
P_1	0.00	0.00	0.00
P_2	0.00	0.00	0.00
P_3	25.00	21.43	19.05
All	2.8	2.65	3.92
		Llama3:70b-Instruct	
P_1	0.00	0.00	0.00
P_2	1.67	4.76	0.95
P_3	6.45	6.45	10.17
All	1.35	2.21	2.12
		Gemma2:27b-Instruct	
P_1	0.00	0.00	0.00
P_2	9.33	9.59	6.11
P_3	16.18	16.00	14.41
All	6.00	6.19	5.88
		Mistral:7b-Instruct	
P_1	0.00	0.00	0.00
P_2	N/A	12.05	N/A
P_3	N/A	33.33	N/A
All	0	7.32	0
		Codestral:22b	
P_1	0.00	0.00	0.00
P_2	0.00	0.00	0.00
P_3	4.76	6.56	10.84
All	0.84	1.42	2.49
		Starcoder2:15b-Instruct	
P_1	0.00	0.00	0.00
P_2	16.00	6.25	4.88
P_3	12.20	12.90	15.87
All	5.49	4.63	5.04

Table 7: Averages of OF Grouped in P_1, P_2, and P_3 (%)

Challenge Set	OG_{DP}	OG_{OPF}	$OG_{\textbf{HyGenar}}$
		GPT-4o	
P_1	0.00	0.00	1.11
P_2	0.00	0.00	0.00
P_3	0.00	0.00	0.60
All	0.00	0.00	0.60
		GPT-3.5-Turbo	
P_1	0.00	0.00	1.19
P_2	0.00	0.00	0.00
P_3	2.13	0.00	1.06
All	0.50	0.00	0.91
		Qwen:72b-Chat	
P_1	1.32	2.56	3.20
P_2	0.00	0.00	2.70
P_3	8.33	0.00	0.00
All	1.87	1.77	2.45
		Llama3:70b-Instruct	
P_1	0.00	0.00	0.00
P_2	0.00	0.00	0.00
P_3	0.00	0.00	0.00
All	0.00	0.00	0.00
		Gemma2:27b-Instruct	
P_1	0.00	0.00	0.00
P_2	0.00	1.37	1.53
P_3	0.00	0.00	0.00
All	0.00	0.33	0.47
		Mistral:7b-Instruct	
P_1	0.00	11.11	0.00
P_2	N/A	0.00	N/A
P_3	N/A	0.00	N/A
All	0.00	7.32	0.00
		Codestral:22b	
P_1	0.70	1.25	1.14
P_2	1.85	1.67	0.97
P_3	0.00	0.00	0.00
All	0.84	1.07	0.83
		Starcoder2:15b-Instruct	
P_1	0.00	0.00	2.24
P_2	12.00	0.00	4.88
P_3	0.00	0.00	0.00
All	1.83	0.00	2.10

Table 8: Averages of OG Grouped in P_1, P_2, and P_3 (%)

Challenge Set	$TU^{\diamond}_{DP}$	$TU^{\diamond}_{OPF}$	$OG^{\diamond}_{\textbf{HyGenar}}$
		GPT-4o	
P_1	100	100	99.81
P_2	99.60	99.55	99.67
P_3	93.04	92.80	91.39
All	97.93	97.81	96.96
		GPT-3.5-Turbo	
P_1	99.64	99.33	97.93
P_2	90.91	89.71	89.11
P_3	78.56	79.74	77.57
All	93.43	93.04	90.32
		Qwen:72b-Chat	
P_1	92.95	94.66	90.47
P_2	76.56	74.29	81.31
P_3	79.79	77.88	77.53
All	88.56	88.79	86.14
		Llama3:70b-Instruct	
P_1	100	100	99.80
P_2	88.48	89.06	93.03
P_3	97.73	98.24	87.33
All	96.58	96.71	95.41
		Gemma2:27b-Instruct	
P_1	99.82	99.82	99.25
P_2	92.67	94.21	95.80
P_3	93.06	94.02	90.61
All	96.50	97.07	95.78
		Mistral:7b-Instruct	
P_1	50.00	68.34	83.33
P_2	N/A	64.67	N/A
P_3	N/A	64.17	N/A
All	50.00	67.01	83.33
		Codestral:22b	
P_1	98.83	98.11	98.36
P_2	88.76	90.31	89.58
P_3	87.33	85.27	83.37
All	94.54	93.66	92.41
		Starcoder2:15b-Instruct	
P_1	97.86	96.44	96.64
P_2	83.29	87.57	81.92
P_3	84.91	85.39	80.48
All	92.40	91.96	89.83

Table 9: Averages of $TU^{\diamond}$ Grouped in P_1, P_2, and P_3 (%)

Parser Feedback 2: Lack of Alternatives

This error is likely due to the reason that the right-hand side is not defined after '::='.

Prompt Template 5: Feedback Prompt in OPF

Given a set of positive and negative examples, generate the Backus–Naur Form (BNF) grammar that accepts all positive examples and rejects all negative examples.
1. Only generate the standard BNF grammar;
2. The generated BNF grammar MUST accept all positive examples and reject all negative examples;
3. Each terminal symbol MUST be quoted with double quotes and MUST NOT escape double quotes or pipeline in terminal symbols;
4. Pay special attention to whether spaces, line breaks, or other special symbols are required between each symbol, and if so, these need to be explicitly specified, e.g. <term> ::= "1" "+" "2" can handle "1+2" but not "1 + 2" while <term> ::= "1" " " "+" " " "2" can handle "1 + 2" but not "1+2";
5. The entry point of the generated BNF grammar MUST be the non-terminal symbol in the first production rule;
6. Only the generated BNF should be wrapped in a pair of triple backtick;
7. Do NOT output any additional texts, comments, or explanations.

===Positive Examples===
{*positive_examples*}
===Negative Examples===
{*negative_examples*}

===Generated BNF===
{*bnf_grammar*}

===Feedback===
The generated BNF grammar has incorrect syntax and please consider fixing it by referring to the feedback.
Here is the feedback from the BNF parser:
{*parser_feedback*}

Prompt Template 6: Generate Positive Examples with a Given Grammar

Generate a list of positive examples with the following constraints:
1. Each example MUST be separated by a newline in addition to the linebreak;
2. Only output examples WITHOUT any additional text or code block, like "```";
3. Only output $\{m\}$ examples;
4. Each example MUST be generated based on the given BNF grammar;
5. Pay attention to whether the whitespaces are allowed between symbols.
For example, given the following BNF grammar:
<term> ::= "0" | "1" | "2"
you should output positive examples like:
0

1

2

Then, the given BNF grammar is:
{*reference_grammar*}