Grammar-Based Code Representation: Is It a Worthy Pursuit for LLMs?

Anonymous ACL submission

Abstract

Grammar serves as a cornerstone in programming languages and software engineering, providing frameworks to define the syntactic space and program structure. Existing research demonstrates the effectiveness of grammar-based code representations in small-scale models, showing their ability to reduce syntax errors and enhance performance. However, as language models scale to the billion level or beyond, syntax-level errors become rare, making it unclear whether grammar information still provides performance benefits. To explore this, we develop a series of billion-scale Grammar-Coder models, incorporating grammar rules in the code generation process. Experiments on HumanEval (+) and MBPP (+) demonstrate a notable improvement in code generation accuracy. Further analysis shows that grammar-based representations enhance LLMs' ability to discern subtle code differences, reducing semantic errors caused by minor variations. These findings suggest that grammar-based code representations remain valuable even in billion-scale models, not only by maintaining syntax correctness but also by improving semantic differentiation.

1 Introduction

Context-free grammars are the fundamental way to specify the syntactic space of a programming language, and with the grammar specified, a program can be parsed into a syntax tree, revealing its structure (Aho et al., 1986). Building on this foundation, leveraging grammatical knowledge (e.g., grammar rules) to pre-train large language models (LLMs) has emerged as a promising strategy for code-related tasks, such as code generation (Zhu et al., 2024; Sun et al., 2020; Guo et al., 2020).

Existing research has explored grammar-based code representation (Jiang et al., 2021; Guo et al., 2022; Wang et al., 2021a; Zhu et al., 2024; Sun et al., 2020; Xiong and Wang, 2022; Rabinovich et al., 2017), where each grammar rule serves as an identity token, and a sequence of grammar rules and terminal tokens represents the program. Figure 1 illustrates a program that determines whether the sum of two integers is odd (top left), along with its corresponding abstract syntax tree (AST) representation (right) and grammar-based representation (bottom left). The grammar-based representation is derived by performing a preorder traversal on the AST. Each grammar rule is extracted independently (e.g., module → function_definition'), while terminals are tokenized using a standard tokenizer (e.g., get'). Grammar-based representation has been shown to be effective in preventing syntax errors in encoder-decoder architecture (Zhu et al., 2024). Moreover, it facilitates program analysis and enables the pruning of incorrect branches (e.g., filtering out type-error programs (Xiong and Wang, 2022; Zhu et al., 2023)) during code generation, thereby enhancing accuracy. Due to these benefits, many code generation models adopt grammar-based representation (Sun et al., 2019; Zhu et al., 2024).

However, as language models scale to the billion-parameter level and beyond, extensive pre-training on large code datasets enables them to implicitly learn syntax rules, making syntax errors increasingly rare (OpenAI, 2024; Yang et al., 2024; Team, 2024; DeepSeek-AI et al., 2024). For example, even 1B scale models, such as DeepSeek-Coder (Guo et al., 2024) and Qwen2.5 (Team, 2024), achieve high accuracy in code generation, consistently producing syntactically valid code. This phenomenon suggests that large models are able to understand the structure of the program and raises a critical question: *Is grammar-based code representation still beneficial for billion-scale LLMs?*

To answer this question, we conduct an experiment comparing grammar-based representation

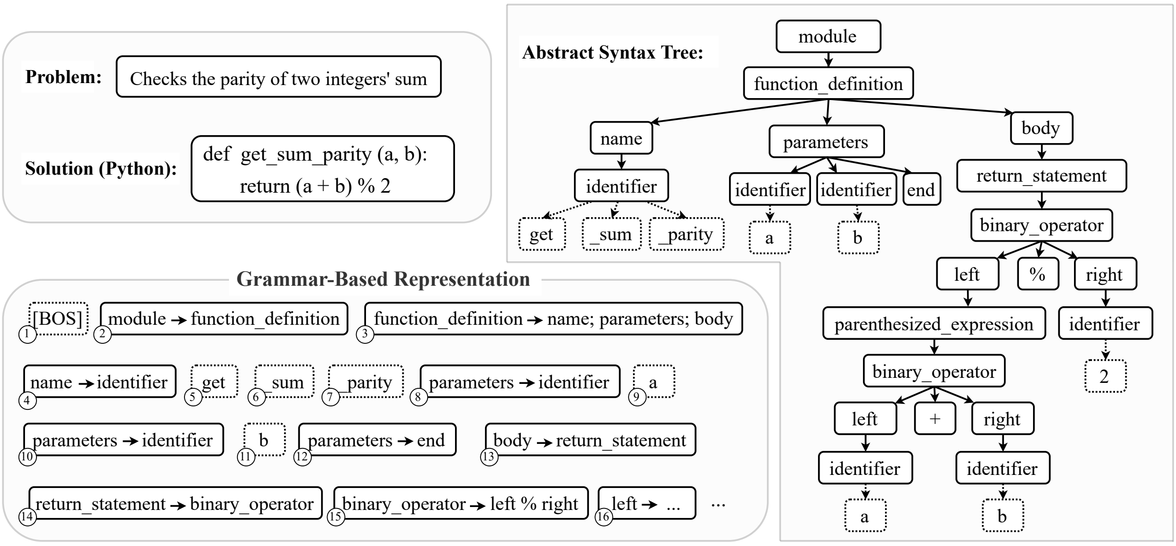

Figure 1: An example of a grammar representation. The top-left part presents a programming problem along with its corresponding Python solution. The right part illustrates the abstract syntax tree (AST) representation of the Python code. The bottom-left section presents the grammar-based representation.

and token-based representation approaches on 1.3B and 1.5B parameter models, respectively. The results demonstrate that grammar-based models (i.e., GrammarCoder-1.3B-Base and GrammarCoder-1.5B-Base) significantly outperform token-based models, even though token-based models rarely make syntax errors. For example, on the MBPP dataset (Austin et al., 2021), GrammarCoder-1.3B-Base achieves an almost seven percentage point improvement in Pass@1 compared to DeepSeek-Coder-1.3B-Base trained on the same data. This suggests that grammar rules enhance code generation beyond syntax correction, even in billion-scale models.

The result leads to a second question: *Why do grammar-based models improve performance if token-based models already produce syntactically correct code?*

To investigate this question, we examine the differences between grammar-based and token-based code representations. Our analysis reveals that minor token-level modifications can lead to substantial semantic shifts, rendering correct programs incorrect. In contrast, while these subtle variations may appear insignificant at the token level, they often map to clear structural differences in grammar-based representations, enabling the model to distinguish more effectively between correct and incorrect code. Experimental results further confirm a correlation between higher performance and the ability of grammar-based code representation to amplify representational differences for semantic shifts, indicating that grammar-based representation helps mitigate such semantic issues.

Our main contributions are as follows:

- We are the first to conduct an experiment on grammar-based code representation in billion-scale LLMs, finding that it remains effective compared to token-based approaches.
- We are the first to explain the effectiveness of grammar-based representation beyond syntax correctness and validate our hypothesis through empirical experiments, demonstrating its role in enhancing code semantic differentiation.
- We release a series of code LLMs trained with grammar rules, providing a valuable resource for further research (GrammarCode, 2025).

2 GrammarCoder

2.1 Model Overview

We propose GrammarCoder, a grammar-based model built on a decoder-only architecture (Vaswani et al., 2017; Radford et al., 2018), which excels in auto-regressive tasks like code generation, completion, and translation (OpenAI, 2024; Guo et al., 2024; Team, 2024; Hui et al., 2024; DeepSeek-AI et al., 2024). To enhance its ability of code generation, we apply continued pre-training and instruction tuning on existing code model weights (i.e., DeepSeek-Coder-1.3B-Base and Qwen2.5-1.5B-Base), expanding its knowledge base. A.1 provides the configuration of the base model we used. In this section, we first introduce our grammar-based code representation. Then, we describe our training strategy and corpus.

2.2 Grammar-Based Code Representation

The main idea of grammar-based code representation is to guide the model in generating grammar rules rather than merely producing a sequence of normal tokens. Traditional code LLMs primarily rely on token-level composition to construct complete code text. In contrast, grammar-based models first generate a complete AST by composing grammar rules and then reconstruct executable code from it, thereby enhancing the model's understanding of code structure and logic. Specifically, normal tokens are obtained using the byte pair encoding (BPE) algorithm, which learns tokenized representations from text corpora, forming a vocabulary $V_{\text{normal}} = \{t_1, t_2, \ldots, t_m\}$. This follows the standard approach used in natural language model training. To integrate grammar information in code representation, we introduce grammar rule sequences, which represent the step-by-step derivation process of an AST. We define a grammar rule vocabulary $V_{\text{rule}} = \{r_1, r_2, \ldots, r_k\}$, where each rule encodes a structural transformation in code generation. Unlike token-based representations, grammar rule sequences explicitly capture logical dependencies and hierarchical structures, providing a more structured view of code. By integrating normal tokens V_{normal} with grammar rules V_{rule}, the model can leverage syntactic rules to strengthen its understanding of code structure. For example, in the bottom-right section of Figure 1, the solid-boxed elements represent grammar rules that guide the construction of the AST (e.g., 'parameters $\rightarrow$ identifier'), ensuring that the generated structure adheres to syntax constraints. Meanwhile, the dashed-boxed elements denote normal tokens (e.g., 'get' and 'a'), which fill in leaf nodes such as variable names and string literals. These tokens can be directly reused from existing BPE tokenization, preserving syntactic correctness while maintaining flexibility in code generation.

GrammarCoder assigns a unique ID to each normal token and grammar rule, storing them in one vocabulary. For example, in the first 10 tokens of Figure 1, IDs 2, 3, 4, 8, and 10 represent grammar rules, while IDs 1, 5, 6, 7, and 9 correspond to normal tokens. Given a base model vocabulary of size m and k grammar rules, the extended vocabulary of GrammarCoder, denoted as V_{grammar}, has a total size of $m + k$. With this grammar-augmented vocabulary, raw code text is converted into a grammar-based representation, enabling the model to learn beyond token-level generation through syntax-aware parsing. Unlike traditional models that rely solely on normal tokens, imposing weak constraints, GrammarCoder incorporates grammar rules, aligning serialized code directly with the preorder traversal of its AST.

2.3 Training Strategy

We train the grammar-based code representation using a next-token prediction strategy, a fundamental approach for auto-regressive language models. The core idea is to predict the most probable next token given a prefix sequence, continuing the process until the full content is generated. In the training process, we treat the grammar-based representation of each code file as the training sample, using the sequence encoded by V_{grammar}. The model learns to predict the next most probable token (whether a normal token or a grammar rule) based on the tokens generated so far. Formal descriptions in A.2.

This training strategy enables the model to dynamically incorporate grammar rules during code generation, allowing the final output adhere to syntax constraints and AST structures.

2.4 Training Corpus

We organize our training corpus in two stages: base model training and instruction tuning. Python is selected as the primary programming language for data collection due to its rich syntax and widespread use in diverse programming paradigms. This makes it an ideal candidate for evaluating the effectiveness of grammar-based representations.

For base model training, we sample 10B tokens of Python code from TheStackV2 (Lozhkov et al., 2024) dataset as the primary training data. Additionally, inspired by previous studies (Huang et al., 2024), we sample 0.5B tokens of self-contained code textbooks from open-source datasets (Huang et al., 2024; Nakamura et al., 2025) to enhance the model's adaptability to real-world interactive scenarios, bridging the gap between standard pre-training and practical applications.

For instruction tuning, we leverage publicly available instruction datasets (Huang et al., 2024; Nakamura et al., 2025) and employ the data synthesis (Wei et al., 2024a,b) approach to collect a total of 6B tokens of instruction data. This ensures the

Model	HumanEval(+)	MBPP(+)
Original		
DeepSeek-Coder-1.3B-Base	34.8 (28.7)	56.7 (47.9)
Qwen2.5-1.5B-Base	37.2 (32.9)	60.2 (49.6)
Normal Token-Based CPT		
DeepSeek-Coder-1.3B-Base (CPT)	43.9 (39.6)	61.4 (51.3)
Qwen2.5-1.5B-Base (CPT)	50.6 (42.7)	60.3 (51.1)
Grammar-Based CPT		
GrammarCoder-1.3B-Base	**63.4** (57.3)	**68.3** (**56.9**)
GrammarCoder-1.5B-Base	**63.4** (**59.1**)	64.8 (55.3)

Table 1: Comparison of code generation performance between token-based and grammar-based models. The CPT refers to continued pre-training, while the SFT denotes supervised fine-tuning.

model is better aligned with instruction-following tasks, improving its ability to handle real-world programming scenarios. A.3 provides detailed information about the training datasets.

3 Experiments

To evaluate the performance of grammar-based code representations, we develop two sets of models, one with grammar-based code representation and one with token-based code representation. These models are built through continued pre-training from open-source code models, DeepSeek-Coder-1.3B-Base and Qwen2.5-1.5B, on high-quality code data. We begin by evaluating these models on code generation tasks, which are among the most widely recognized and commonly used benchmarks for assessing code-related capabilities (i.e., Experiment I in 3.1).

To further explore the differences between grammar-based and token-based representations, we analyze the reason contributing to the performance gains of grammar-based representation (i.e., Experiment II in 3.2).

3.1 Experiment I: Performance on Code Generation

Evaluation Benchmarks. We use HumanEval (Chen et al., 2021) and MBPP (Austin et al., 2021), the most widely used datasets for code generation tasks, as evaluation benchmarks. HumanEval contains 164 tasks, while MBPP includes 500 testing tasks, both equipped with built-in test cases for evaluation. EvalPlus (Liu et al., 2023) extends these datasets by introducing stricter test cases to improve assessment robustness. We conduct evaluations using both the original benchmark test cases and their EvalPlus-enhanced versions, denoted by a "+" suffix.

Baselines. To evaluate the effectiveness of grammar-based code representation, we select DeepSeek-Coder-1.3B (Guo et al., 2024) and Qwen2.5-1.5B (Team, 2024) as baseline models and perform continued pre-training. DeepSeek-Coder-1.3B, trained on high-quality large-scale code datasets, serves as a strong representative of the code model. Meanwhile, Qwen2.5-1.5B-Base, despite being a general-purpose model, demonstrates competitive performance on code-related tasks, making it a valuable point of comparison.

Metrics. We adopt Pass@1 as the evaluation metric. Specifically, for each problem, the model generates a single code sample, which is deemed correct if it passes all predefined unit tests. The Pass@1 score is calculated as:

$$\text{Pass@1} = \frac{\text{Number of problems solved correctly}}{\text{Total number of problems}}$$

Implementation Details. GrammarCoder models are trained with 8 NVIDIA H800 GPUs. During the base model training phase, we adopt a two-stage learning rate strategy, following approaches from OpenCoder (Huang et al., 2024) and MiniCPM (Hu et al., 2024). Initially, we use a higher learning rate of 3e-4 to accelerate convergence to a reasonable parameter range. The learning rate is reduced to 5e-5 in the annealing stage for further performance optimization. During the instruct model training phase, we set the learning rate to 5e-5 and trained on an instruction dataset to improve generalization in instruction understanding and code tasks. Throughout the training process, we apply 100 warm-up steps and use a cosine learning rate scheduler to ensure smooth learning rate adjustments, maintaining training stability and efficiency. Additionally, during both token-based and grammar-based continued pre-training, we utilize the same settings to ensure a fair comparison. 2.4 and A.3 provide the detailed information of training dataset.

Results. Table 1 presents our main experimental results, showing that the GrammarCoder-base model significantly outperforms both the original model and the token-based model trained on the same datasets. For example, on the HumanEval dataset, GrammarCoder-1.3B-Base achieves 82% and 44% improvements over DeepSeek-Coder-1.3B-Base and DeepSeek-Coder-1.3B-Base (CPT), respectively. Notably, after performing continued pre-training on the training dataset, both

Model	HumanEval	HumanEval+	MBPP	MBPP+
Base Models				
DeepSeek-Coder-1.3B-Base (Guo et al., 2024)	34.8	28.7	56.7	47.9
Qwen2.5-1.5B-Base (Team, 2024)	37.2	32.9	60.2	49.6
OpenCoder-1.5B-Base (Huang et al., 2024)	54.3	49.4	**70.6**	**58.7**
Yi-Coder-1.5B (AI et al., 2025)	41.5	32.9	27.0	22.2
CodeGemma-2B-Base (Team et al., 2024)	26.8	20.7	55.6	46.6
StarCoder2-3B (Lozhkov et al., 2024)	31.7	27.4	60.2	49.1
CodeGemma-7B-Base (Team et al., 2024)	44.5	41.5	65.1	52.4
StarCoder2-7B (Lozhkov et al., 2024)	35.4	29.9	54.4	45.6
GrammarCoder-1.3B-Base	63.4	57.3	68.3	56.9
GrammarCoder-1.5B-Base	**63.4**	**59.1**	64.8	55.3
Instruct Models				
DeepSeek-Coder-1.3B-Instruct (Guo et al., 2024)	65.9	60.4	64.3	54.8
Qwen2.5-1.5B-Instruct (Team, 2024)	61.6	49.4	63.2	55.6
OpenCoder-1.5B-Instruct (Huang et al., 2024)	72.5	67.7	72.7	**61.9**
Yi-Coder-1.5B-Chat (AI et al., 2025)	67.7	63.4	68.0	59.0
Phi-3-Mini-4K-3.8B-Instruct (Abdin et al., 2024)	64.6	59.1	65.9	54.2
CodeGemma-7B-Instruct (Team et al., 2024)	60.4	51.8	70.4	56.9
GrammarCoder-1.3B-Instruct	70.7	64.0	71.2	58.7
GrammarCoder-1.5B-Instruct	**73.2**	**68.3**	**73.3**	61.1

Table 2: Performance of various base models and instruct models on HumanEval and MBPP.

the token-based and grammar-based models exhibit performance gains. Moreover, even without grammar-based representation, neither the original token-based model nor the continued pre-trained model produces syntax errors, with syntax correctness nearly reaching 100%. Occasional syntax errors (fewer than three) only occur due to random variations on the HumanEval and MBPP datasets. Despite this near-perfect syntax correctness, the grammar-based model still demonstrates superior performance, indicating that incorporating grammar rules provides additional benefits beyond merely preventing syntax errors.

Building on the base model, we further conduct supervised instruction tuning to enhance the model's adaptability to instruction-based tasks. Table 2 compares the performance of GrammarCoder with current state-of-the-art code models of similar or larger scales. Experimental results show that grammar-based code representation achieves performance comparable to the best token-based models. For example, on the HumanEval (+) dataset, both the base and instruct versions of GrammarCoder outperform other models (e.g., CodeGemma-7B and Yi-Coder-1.5B), while the instruct version achieves performance on par with OpenCoder. However, on the MBPP+ dataset, GrammarCoder-Base does not surpass OpenCoder-Base, which may be attributed to differences in training data volume and quality during the base model pre-training stage. OpenCoder benefits from training on over 900B tokens of high-quality data, whereas GrammarCoder is pre-trained on only around 10B tokens in grammar-based representation. This suggests that while grammar-based representation proves to be effective, the scale and quality of training data also play a crucial role in achieving state-of-the-art performance. Future work can explore expanding the amount of high-quality code data processed into grammar-based representations to further enhance model performance.

3.2 Experiment II: Understanding the Performance Difference

Experiment Design. While our experimental results demonstrate that grammar-based representation enhances code generation, it remains crucial to understand what drives this improvement, especially given that syntax errors are already rare in billion-scale LLMs. To explore the reason behind these results, we focus on why grammar-based representations help mitigate semantic errors beyond preventing syntactic errors, aiming to uncover their role in reducing overall mistakes. Analyzing the different representation results, we observe that grammar-based representation may amplify differences between correct and incorrect programs that appear minimal at the token level. This heightened sensitivity to fine-grained variations may help prevent LLMs from behaving like "careless programmers", who often make mistakes by overlooking subtle details. By capturing these distinctions more effectively, grammar-based models could re-

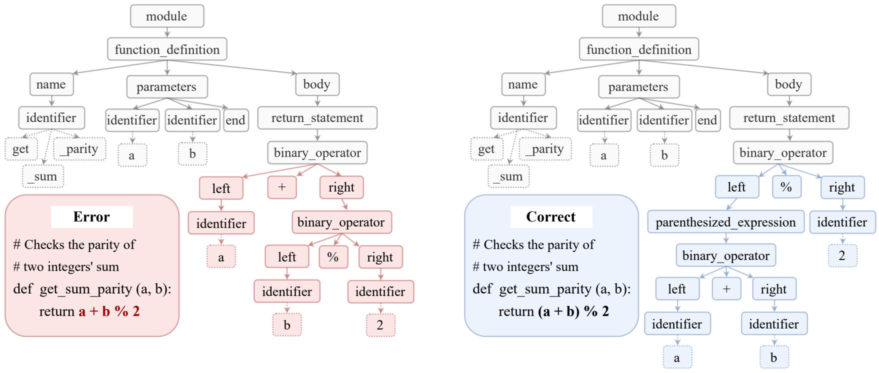

Figure 2: An example showing the differences of code representations between error and correct code.

duce such semantic errors, leading to higher performance in code generation tasks.

To validate this hypothesis, we design a new set of experiments focusing on subtle semantic changes that are likely to be overlooked by both humans and token-based models. Specifically, we investigate (1) whether grammar-based representation amplifies these differences, and (2) whether grammar-based models can better capture these changes. These experiments aim to provide deeper insight into how grammar-based representation improves the model's ability to distinguish between correct and error code, making it more effective in avoiding semantic errors and improving performance.

First, we conduct a quantitative analysis to explore the potential differences between grammar-based and token-based representations. Specifically, we encode code snippets that are similar at the token level but semantically different using both representation strategies and compare their edit distance when transforming one code into another.

Next, we train separate grammar-based and token-based code semantic classifiers to evaluate the impact of grammar-based representations on semantic classification. By training classification models on differently represented code datasets, we examine the extent to which each representation affects the model's ability to capture semantic differences.

Finally, we assess whether the differences introduced by grammar rules contribute to performance improvements, confirming their effectiveness in enhancing LLMs.

Result 1: Grammar-Based Representation Amplifies Subtle Token-Level Differences. We analyze whether grammar-based representation amplifies subtle differences by comparing the edit distances between semantically different code snippets under grammar-based and token-based representations. CodeNanoFix (Liang et al., 2024) dataset is used to measure the edit distance, providing a quantitative assessment of how grammar-based and token-based approaches represent code. This dataset consists of 1,000+ programming problems and nearly 100,000 code sample pairs with minimal token differences but significant semantic variations. A subset of 120 programming problems and 3,583 sample pairs serve as the test set. Each sample in the dataset consists of error code submitted by human programmers while solving a problem, along with its corrected version modified by the programmer, both exhibiting minimal token-level differences. Since the differences between error and corrected code typically involve subtle yet crucial semantic changes, such as control flow modifications, variable scope adjustments, and operator usage corrections, this dataset is well-suited for analyzing the differences between token-based and grammar-based representations. To mitigate the impact of outliers, we focus on code pairs with minimal token-level differences (edit distance less than 50, covering 91.18% of the CodeNanoFix's test set). Additionally, we use GrammarCoder-1.3B's vocabulary to produce grammar-based representations and DeepSeek-Coder-1.3B's vocabulary to produce token-based representations, ensuring a

	Precision	Recall	F1-Score
DeepSeek-Coder-1.3B-Base	71.99	62.77	67.06
DeepSeek-Coder-1.3B-Instruct	74.20	65.59	69.63
Qwen2.5-1.5B-Base (Team, 2024)	72.16	64.97	68.38
Qwen2.5-1.5B-Instruct	71.42	67.32	69.31
Condor-1.3B (Liang et al., 2024)	74.39	72.40	73.38
GrammarCoder-1.3B-Base	**77.39**	**81.30**	**79.30**
GrammarCoder-1.5B-Base	72.34	76.50	74.36

Table 3: The performance of semantic classification tasks.

fair comparison is made with the maximum overlap of shared tokens.

The results show that grammar-based representation typically produces larger edit distances compared to token-based representation. Specifically, the average edit distance from error to correct code at the token level is 14.33, while for grammar-based representations, it increases to 27.43, a 91.18% increase. B.1 shows the edit distance distribution for error-correct code pairs. Figure 2 presents a concrete example, where the left side shows an error code snippet caused by neglecting operator precedence, while the right side displays the correct version. At the token level, the difference between the two codes consists of only two characters (i.e., '(' and ')'), resulting in an edit distance of 2. However, at the grammar representation level, the change in operator precedence leads to significant differences in the AST structure and the grammar rules applied, increasing the edit distance to 6. B.2 also presents the further analysis results of the LLM's generated outputs on CodeNanoFix, which reveal similar conclusions. These results indicate that the introduction of grammar rules amplifies representation differences that may be overlooked at the token level. Consequently, the grammar-based representation provides a more distinct encoding of correct and incorrect code, allowing the model to better capture semantic variations.

Result 2: Grammar-Based Representation Strengthens Semantic Distinction. We evaluate whether grammar-based models more effectively capture these changes by training classifiers using different code representation approaches. Specifically, we use CodeNanoFix as a dataset for a semantic classification task, evaluating the model's ability to distinguish between semantically correct and incorrect code. By training classifiers to identify code correctness, we examine whether grammar-based representations improve the model's understanding of code semantics. To ensure a fair comparison, we select baselines that align with GrammarCoder's architecture. Specifically, we use its corresponding base models, DeepSeek-Coder-1.3B and Qwen2.5-1.5B, along with Condor-1.3B (Liang et al., 2024), a model specifically designed for the CodeNanoFix classification task. Precision, Recall, and F1 score are utilized as key metrics for classification performance. Precision evaluates the accuracy of correct code predictions, Recall measures the model's ability to identify the actual correct code, and F1 score provides a balanced assessment of overall classification performance. Similar to Condor, GrammarCoder is fine-tuned on the CodeNanoFix dataset to enhance its understanding of code semantics and alignment with problem descriptions. In the implementation, a classification layer is added to the original model to output probability scores, with 0.5 sets as the classification threshold. Code snippets with scores above 0.5 are considered correct, while those below are classified as errors. During fine-tuning, the learning rate is set to 5e-5 to ensure stable optimization for the code classification task.

Table 3 illustrates the impact of different code representation approaches on the model's ability to determine code semantic correctness. The results indicate that incorporating grammar rules significantly enhances the model's ability to distinguish correct from incorrect code. For example, GrammarCoder-1.3B-Base and GrammarCoder-1.5B-Base improve F1 scores by 18.25% and 8.75%, respectively, compared to their base models DeepSeek-Coder-1.3B-Base and Qwen-1.5B-Base. These results demonstrate that the incorporation of grammar rules enables the model to more precisely differentiate token-level similar but semantically distinct code snippets, improving its ability to recognize subtle semantic differences. Furthermore, even compared to Condor, the current best-performing model on the CodeNanoFix dataset, GrammarCoder-1.3B-Base achieves nearly nine percentage points higher Recall and improves the F1 score by almost six percentage points. Notably, both Condor and GrammarCoder-1.3B-Base are trained from the same baseline model, DeepSeek-Coder-1.3B-Base. This further highlights the effectiveness of grammar-based representation in distinguishing semantic differences caused by subtle token-level changes in code.

Result 3: Correlation Between Representation and Performance. We conduct a correlation analysis to examine whether the increase in edit distance is related to GrammarCoder's ability to distinguish between semantically correct and incorrect code. A chi-square test confirms a statistically significant correlation, with GrammarCoder-1.3B-Base and GrammarCoder-1.5B-Base achieving p-values of 0.0051 and 0.0006, respectively. As a p-value below 0.05 indicates statistical significance, the results suggest that grammar-based representation contributes to performance improvements by amplifying structural differences in code. B.3

also presents case studies where the token-based model's generated outputs can be corrected with minor modifications at the token level. This ability brought by grammar-based representation helps prevent the model from exhibiting oversight-prone tendencies akin to a "careless programmer," where minor but critical details are ignored, potentially leading to semantic errors. As a result, grammar-based representation not only improves the model's understanding of code semantics but also enhances overall performance in code generation.

4 Related Work

4.1 Large Language Models for Code

Since the release of ChatGPT-3.5 (cha, 2022) sparked a new wave of interest in LLMs, increasing focus has been on training and utilizing LLMs for code-related tasks. These models can be broadly categorized into two types. The first category consists of general-purpose models, which perform well in various natural language tasks, while also showing strong capabilities in code-related tasks. Examples of models in this category include ChatGPT (OpenAI, 2024), Gemini (Reid et al., 2024), Claude (Anthropic, 2025), Qwen (Team, 2024), and DeepSeek (Bi et al., 2024). The second category comprises models specifically trained on code data, including models such as CodeLlama (Rozière et al., 2024), OpenCoder (Huang et al., 2024), and DeepSeek-Coder (Guo et al., 2024). Compared to general-purpose models, these specialized models can achieve comparable or superior performance on code-related tasks with fewer parameters and offer broader support for less common programming languages. However, regardless of whether they have been specifically trained for code-related tasks, these models represent programming languages in the same way as natural language—using token sequences. This hinders the model's ability to recognize the inherent structural information of programming languages. Therefore, we leverage the grammar-based code representation to train GrammarCoder, which enhances the model's ability to capture structural information inherent in programming languages.

4.2 Grammar-Based Code Representation

Many models attempt to incorporate grammar-based information into code representations (Sun et al., 2019; Guo et al., 2022; Zhu et al., 2024; Sun et al., 2020; Xiong and Wang, 2022; Rabinovich et al., 2017). These models have been validated on relatively small-scale models (fewer than 220M parameters), demonstrating that grammar-based representation helps prevent syntax errors and enhances code generation performance. For example, GrammarT5 (Zhu et al., 2024) is a pre-trained model based on grammatical rules. It is trained based on CodeT5 (220M) (Wang et al., 2021b) with an encoder-decoder architecture using the same training data, demonstrating that grammar-based representations can enhance model performance. However, with the emergence of LLMs, models' size has expanded rapidly, and decoder-only architectures have gradually become mainstream. It's unclear whether grammar-based representations remain effective in larger-scale (e.g., billion-size) decoder-only models. Moreover, beyond preventing grammatical errors, it remains unclear whether grammar-based representations provide any additional benefits. Therefore, we bridge these gaps by training and evaluating grammar-based representations in billion-scale decoder-only models. Additionally, we explore why grammar-based representation remains effective when syntax errors are rare in LLMs, providing insights into its broader impact on model performance.

5 Conclusion

In this paper, we introduce GrammarCoder, a series of models trained using grammar-based code representations. To evaluate whether this approach remains effective even when billion-scale models basically no longer make syntax errors, we assess GrammarCoder on widely used code generation benchmarks, HumanEval(+) and MBPP(+). Experimental results show that after continued pre-training on the same datasets, GrammarCoder significantly outperforms models trained with normal token-based representations. To further investigate why grammar-based code representations are effective, we first quantify the differences between grammar-based and token-based approaches in representing code. Additionally, we train a classification model to assess their ability to capture subtle code variations. Our findings reveal that while modern LLMs rarely make syntax errors, grammar-based representations still enhance their ability to distinguish fine-grained token-level differences. This reduces semantic errors caused by minor variations and ultimately improves model performance in code-related tasks.

References

2022. Chatgpt: Optimizing language models for dialogue. Accessed: 2023-01-16.

Marah Abdin, Jyoti Aneja, Hany Awadalla, Ahmed Awadallah, Ammar Ahmad Awan, Nguyen Bach, Amit Bahree, Arash Bakhtiari, Jianmin Bao, Harkirat Behl, Alon Benhaim, Misha Bilenko, Johan Bjorck, Sébastien Bubeck, Martin Cai, Qin Cai, Vishrav Chaudhary, Dong Chen, Dongdong Chen, Weizhu Chen, Yen-Chun Chen, Yi-Ling Chen, Hao Cheng, Parul Chopra, Xiyang Dai, Matthew Dixon, Ronen Eldan, Victor Fragoso, Jianfeng Gao, Mei Gao, Min Gao, Amit Garg, Allie Del Giorno, Abhishek Goswami, Suriya Gunasekar, Emman Haider, Junheng Hao, Russell J. Hewett, Wenxiang Hu, Jamie Huynh, Dan Iter, Sam Ade Jacobs, Mojan Javaheripi, Xin Jin, Nikos Karampatziakis, Piero Kauffmann, Mahoud Khademi, Dongwoo Kim, Young Jin Kim, Lev Kurilenko, James R. Lee, Yin Tat Lee, Yuanzhi Li, Yunsheng Li, Chen Liang, Lars Liden, Xihui Lin, Zeqi Lin, Ce Liu, Liyuan Liu, Mengchen Liu, Weishung Liu, Xiaodong Liu, Chong Luo, Piyush Madan, Ali Mahmoudzadeh, David Majercak, Matt Mazzola, Caio César Teodoro Mendes, Arindam Mitra, Hardik Modi, Anh Nguyen, Brandon Norick, Barun Patra, Daniel Perez-Becker, Thomas Portet, Reid Pryzant, Heyang Qin, Marko Radmilac, Liliang Ren, Gustavo de Rosa, Corby Rosset, Sambudha Roy, Olatunji Ruwase, Olli Saarikivi, Amin Saied, Adil Salim, Michael Santacroce, Shital Shah, Ning Shang, Hiteshi Sharma, Yelong Shen, Swadheen Shukla, Xia Song, Masahiro Tanaka, Andrea Tupini, Praneetha Vaddamanu, Chunyu Wang, Guanhua Wang, Lijuan Wang, Shuohang Wang, Xin Wang, Yu Wang, Rachel Ward, Wen Wen, Philipp Witte, Haiping Wu, Xiaoxia Wu, Michael Wyatt, Bin Xiao, Can Xu, Jiahang Xu, Weijian Xu, Jilong Xue, Sonali Yadav, Fan Yang, Jianwei Yang, Yifan Yang, Ziyi Yang, Donghan Yu, Lu Yuan, Chenruidong Zhang, Cyril Zhang, Jianwen Zhang, Li Lyna Zhang, Yi Zhang, Yue Zhang, Yunan Zhang, and Xiren Zhou. 2024. Phi-3 technical report: A highly capable language model locally on your phone. *Preprint*, arXiv:2404.14219.

Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. 1986. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley series in computer science / World student series edition. Addison-Wesley.

01. AI, :, Alex Young, Bei Chen, Chao Li, Chengen Huang, Ge Zhang, Guanwei Zhang, Guoyin Wang, Heng Li, Jiangcheng Zhu, Jianqun Chen, Jing Chang, Kaidong Yu, Peng Liu, Qiang Liu, Shawn Yue, Senbin Yang, Shiming Yang, Wen Xie, Wenhao Huang, Xiaohui Hu, Xiaoyi Ren, Xinyao Niu, Pengcheng Nie, Yanpeng Li, Yuchi Xu, Yudong Liu, Yue Wang, Yuxuan Cai, Zhenyu Gu, Zhiyuan Liu, and Zonghong Dai. 2025. Yi: Open foundation models by 01.ai. *Preprint*, arXiv:2403.04652.

Anthropic. 2025. Introducing the next generation of claude.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. 2021. Program synthesis with large language models. *Preprint*, arXiv:2108.07732.

Xiao Bi, Deli Chen, Guanting Chen, Shanhuang Chen, Damai Dai, Chengqi Deng, Honghui Ding, Kai Dong, Qiushi Du, Zhe Fu, et al. 2024. Deepseek llm: Scaling open-source language models with longtermism. *arXiv preprint arXiv:2401.02954*.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating large language models trained on code.

Cognitive Computations. 2024. Code-290k-sharegpt-vicuna.

DeepSeek-AI, Qihao Zhu, Daya Guo, Zhihong Shao, Dejian Yang, Peiyi Wang, Runxin Xu, Y. Wu, Yukun Li, Huazuo Gao, Shirong Ma, Wangding Zeng, Xiao Bi, Zihui Gu, Hanwei Xu, Damai Dai, Kai Dong, Liyue Zhang, Yishi Piao, Zhibin Gou, Zhenda Xie, Zhewen Hao, Bingxuan Wang, Junxiao Song, Deli Chen, Xin Xie, Kang Guan, Yuxiang You, Aixin Liu, Qiushi Du, Wenjun Gao, Xuan Lu, Qinyu Chen, Yaohui Wang, Chengqi Deng, Jiashi Li, Chenggang Zhao, Chong Ruan, Fuli Luo, and Wenfeng Liang. 2024. Deepseek-coder-v2: Breaking the barrier of closed-source models in code intelligence. *Preprint*, arXiv:2406.11931.

Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. 2024. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*.

GrammarCode. 2025. Grammarcode.

Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. Unixcoder: Unified cross-modal pre-training for code representation. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 7212–7225.

Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. 2020. Graphcodebert: Pre-training code representations with data flow. *arXiv preprint arXiv:2009.08366*.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Yu Wu, YK Li, et al. 2024. Deepseek-coder: When the large language model meets programming–the rise of code intelligence. *arXiv preprint arXiv:2401.14196*.

Shengding Hu, Yuge Tu, Xu Han, Chaoqun He, Ganqu Cui, Xiang Long, Zhi Zheng, Yewei Fang, Yuxiang Huang, Weilin Zhao, et al. 2024. Minicpm: Unveiling the potential of small language models with scalable training strategies. *arXiv preprint arXiv:2404.06395*.

Siming Huang, Tianhao Cheng, J. K. Liu, Jiaran Hao, Liuyihan Song, Yang Xu, J. Yang, J. H. Liu, Chenchen Zhang, Linzheng Chai, Ruifeng Yuan, Zhaoxiang Zhang, Jie Fu, Qian Liu, Ge Zhang, Zili Wang, Yuan Qi, Yinghui Xu, and Wei Chu. 2024. Opencoder: The open cookbook for top-tier code large language models. *Preprint*, arXiv:2411.04905.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, Kai Dang, Yang Fan, Yichang Zhang, An Yang, Rui Men, Fei Huang, Bo Zheng, Yibo Miao, Shanghaoran Quan, Yunlong Feng, Xingzhang Ren, Xuancheng Ren, Jingren Zhou, and Junyang Lin. 2024. Qwen2.5-coder technical report. *Preprint*, arXiv:2409.12186.

Xue Jiang, Zhuoran Zheng, Chen Lyu, Liang Li, and Lei Lyu. 2021. Treebert: A tree-based pre-trained model for programming language. In *Proceedings of the Thirty-Seventh Conference on Uncertainty in Artificial Intelligence*, volume 161 of *Proceedings of Machine Learning Research*, pages 54–63. PMLR.

Qingyuan Liang, Zhao Zhang, Chen Liu, Zeyu Sun, Wenjie Zhang, Yizhou Chen, Zixiao Zhao, Qi Luo, Wentao Wang, Yanjie Jiang, et al. 2024. Condor: A code discriminator integrating general semantics with code details. *arXiv preprint arXiv:2412.17429*.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023. Is your code generated by chatGPT really correct? rigorous evaluation of large language models for code generation. In *Thirty-seventh Conference on Neural Information Processing Systems*.

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.

Taishi Nakamura, Mayank Mishra, Simone Tedeschi, Yekun Chai, Jason T Stillerman, Felix Friedrich, Prateek Yadav, Tanmay Laud, Vu Minh Chien, Terry Yue Zhuo, et al. 2025. Aurora-m: Open source continual pre-training for multilingual language and code. In *Proceedings of the 31st International Conference on Computational Linguistics: Industry Track*, pages 656–678.

OpenAI. 2024. Chatgpt.

Maxim Rabinovich, Mitchell Stern, and Dan Klein. 2017. Abstract syntax networks for code generation and semantic parsing. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1139–1149.

Alec Radford, Karthik Narasimhan, Tim Salimans, Ilya Sutskever, et al. 2018. Improving language understanding by generative pre-training.

Machel Reid, Nikolay Savinov, Denis Teplyashin, Dmitry Lepikhin, Timothy P. Lillicrap, Jean-Baptiste Alayrac, Radu Soricut, Angeliki Lazaridou, Orhan Firat, Julian Schrittwieser, Ioannis Antonoglou, Rohan Anil, Sebastian Borgeaud, Andrew M. Dai, Katie Millican, Ethan Dyer, Mia Glaese, Thibault Sottiaux, Benjamin Lee, Fabio Viola, Malcolm Reynolds, Yuanzhong Xu, James Molloy, Jilin Chen, Michael Isard, Paul Barham, Tom Hennigan, Ross McIlroy, Melvin Johnson, Johan Schalkwyk, Eli Collins, Eliza Rutherford, Erica Moreira, Kareem Ayoub, Megha Goel, Clemens Meyer, Gregory Thornton, Zhen Yang, Henryk Michalewski, Zaheer Abbas, Nathan Schucher, Ankesh Anand, Richard Ives, James Keeling, Karel Lenc, Salem Haykal, Siamak Shakeri, Pranav Shyam, Aakanksha Chowdhery, Roman Ring, Stephen Spencer, Eren Sezener, and et al. 2024. Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context. *CoRR*, abs/2403.05530.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. 2024. Code llama: Open foundation models for code. *Preprint*, arXiv:2308.12950.

Zeyu Sun, Qihao Zhu, Lili Mou, Yingfei Xiong, Ge Li, and Lu Zhang. 2019. A grammar-based structural cnn decoder for code generation. In *Proceedings of the AAAI conference on artificial intelligence*, volume 33, pages 7055–7062.

Zeyu Sun, Qihao Zhu, Yingfei Xiong, Yican Sun, Lili Mou, and Lu Zhang. 2020. Treegen: A tree-based transformer architecture for code generation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 8984–8991.

CodeGemma Team, Heri Zhao, Jeffrey Hui, Joshua Howland, Nam Nguyen, Siqi Zuo, Andrea Hu, Christopher A. Choquette-Choo, Jingyue Shen, Joe Kelley, Kshitij Bansal, Luke Vilnis, Mateo Wirth, Paul Michel, Peter Choy, Pratik Joshi, Ravin Kumar, Sarmad Hashmi, Shubham Agrawal, Zhitao Gong, Jane Fine, Tris Warkentin, Ale Jakse Hartman, Bin Ni, Kathy Korevec, Kelly Schaefer, and Scott Huffman. 2024. Codegemma: Open code models based on gemma. *Preprint*, arXiv:2406.11409.

Qwen Team. 2024. Qwen2.5: A party of foundation models.

TokenBender. 2024. code_instructions_122k_alpaca_style.

TreeSitter. 2024. Treesitter.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems*, 30.

Xin Wang, Yasheng Wang, Fei Mi, Pingyi Zhou, Yao Wan, Xiao Liu, Li Li, Hao Wu, Jin Liu, and Xin Jiang. 2021a. Syncobert: Syntax-guided multi-modal contrastive pre-training for code representation. *arXiv preprint arXiv:2108.04556*.

Yue Wang, Weishi Wang, Shafiq R. Joty, and Steven C. H. Hoi. 2021b. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, EMNLP 2021, Virtual Event / Punta Cana, Dominican Republic, 7-11 November, 2021*, pages 8696–8708. Association for Computational Linguistics.

Yuxiang Wei, Federico Cassano, Jiawei Liu, Yifeng Ding, Naman Jain, Zachary Mueller, Harm de Vries, Leandro von Werra, Arjun Guha, and Lingming Zhang. 2024a. Selfcodealign: Self-alignment for code generation. *arXiv preprint arXiv:2410.24198*.

Yuxiang Wei, Zhe Wang, Jiawei Liu, Yifeng Ding, and Lingming Zhang. 2024b. Magicoder: Empowering code generation with oss-instruct. In *Forty-first International Conference on Machine Learning*.

Yingfei Xiong and Bo Wang. 2022. L2s: A framework for synthesizing the most probable program under a specification. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 31(3):1–45.

An Yang, Baosong Yang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Zhou, Chengpeng Li, Chengyuan Li, Dayiheng Liu, Fei Huang, Guanting Dong, Haoran Wei, Huan Lin, Jialong Tang, Jialin Wang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Ma, Jianxin Yang, Jin Xu, Jingren Zhou, Jinze Bai, Jinzheng He, Junyang Lin, Kai Dang, Keming Lu, Keqin Chen, Kexin Yang, Mei Li, Mingfeng Xue, Na Ni, Pei Zhang, Peng Wang, Ru Peng, Rui Men, Ruize Gao, Runji Lin, Shijie Wang, Shuai Bai, Sinan Tan, Tianhang Zhu, Tianhao Li, Tianyu Liu, Wenbin Ge, Xiaodong Deng, Xiaohuan Zhou, Xingzhang Ren, Xinyu Zhang, Xipin Wei, Xuancheng Ren, Xuejing Liu, Yang Fan, Yang Yao, Yichang Zhang, Yu Wan, Yunfei Chu, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, Zhifang Guo, and Zhihao Fan. 2024. Qwen2 technical report. *Preprint*, arXiv:2407.10671.

Tianyu Zheng, Ge Zhang, Tianhao Shen, Xueling Liu, Bill Yuchen Lin, Jie Fu, Wenhu Chen, and Xiang Yue. 2024. Opencodeinterpreter: Integrating code generation with execution and refinement. *arXiv preprint arXiv:2402.14658*.

Qihao Zhu, Qingyuan Liang, Zeyu Sun, Yingfei Xiong, Lu Zhang, and Shengyu Cheng. 2024. Grammart5: Grammar-integrated pretrained encoder-decoder neural model for code. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*, pages 76:1–76:13. ACM.

Qihao Zhu, Zeyu Sun, Wenjie Zhang, Yingfei Xiong, and Lu Zhang. 2023. Tare: Type-aware neural program repair. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 1443–1455.

A Approach Details

A.1 Mode Configuration

Config	DeepSeek-Coder	Qwen2.5
# parameters	1.3 B	1.5 B
# hidden_layer	24	28
# hidden_size	2,048	1,537
# intermediate_size	5,504	8,960
# attention_head	16	12
# vocabulary	32,256	151,936

Table 4: The main configuration of two different base models.

Table 4 presents the key configurations of the base models used in our study: DeepSeek-Coder and Qwen2.5. While both models are billion-scale in terms of parameter count, they exhibit differences in architectural details, particularly in vocabulary size. DeepSeek-Coder has a vocabulary size of 32,256, whereas Qwen2.5 employs a significantly larger vocabulary of 151,936. Since grammar-based representations restructure code at a syntactic level rather than relying solely on the token level, their effectiveness is not dependent on the original vocabulary. Therefore, this difference in vocabulary size can underscore the robustness of our grammar-based code representation. After incorporating grammar rules, our vocabulary sizes expand to 33,465 for DeepSeek-Coder and 153,108 for Qwen2.5. If GrammarCoder demonstrates improved performance across both base models, it would further indicate that grammar-based approaches are adaptable to different model architectures and tokenization strategies.

A.2 Training Objective

The training objective of GrammarCoder is to maximize the conditional probability of the next token given the preceding sequence. The loss function of training objective can be formalized as:

$$\mathcal{L} = -\sum_{t=1}^{N} \log P(x_t | x_1, x_2, \ldots, x_{t-1}; \theta)$$

, where x_t represents the token (either a normal token or a grammar rule from V_{grammar}) at step t, $x_1, x_2, ..., x_{t-1}$ denotes the previously generated sequence, θ represents the model parameters, and the objective is to maximize the conditional probability of the correct token given the current context $P(x_t \mid x_1, x_2, ..., x_{t-1})$.

This ensures that the final output adheres to syntax constraints while effectively capturing correct program logic, aligning with the preorder traversal of the complete AST.

A.3 Training Datasets and Filter

Name	# Samples
code_contests_instruct	4.4 M
Opencoder-sft-stage1	4.2 M
Opencoder-sft-stage2	375K
Code-290k-ShareGPT-Vicuna-Clean	285K
CodeFeedback-Filtered-Instruction	156K
code_instructions_122k_alpaca_style	121K

Table 5: Open-source instruction datasets are used in our instruction tuning process.

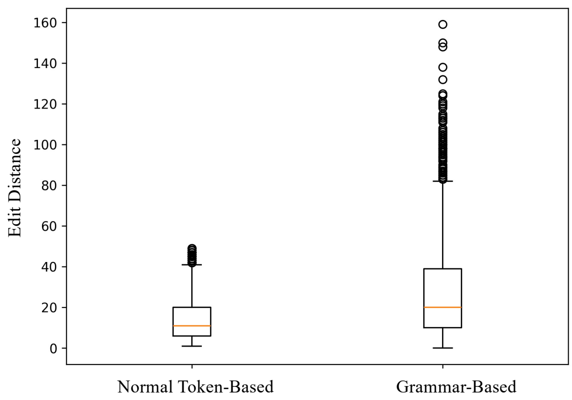

Figure 3: Edit distance distribution across different code representation approaches.

For training the base models, we primarily use high-quality Python code, aligning with our focus on grammar-based code representation. Our dataset is composed of two key sources. First, we sample 10B tokens from TheStackV2 (Lozhkov et al., 2024), a large-scale code corpus that provides diverse and high-quality programming samples across various domains, ensuring a strong foundation in general coding patterns and structures. Second, inspired by previous studies (Huang et al., 2024; Yang et al., 2024), we incorporate 0.5B tokens of self-contained code textbooks from open-source repositories (Huang et al., 2024). Unlike context-dependent snippets, these samples consist of independent tasks and corresponding code snippets, helping the model learn to generate independent and coherent programs, bridging the gap between standard pre-training and real-world interactive programming scenarios.

For training the instruct models, we use instruction data consisting of two main sources: publicly available instruction datasets and synthetically generated instruction data. Table 5 lists the open-source instruct datasets used in our training (Hu et al., 2024; Huang et al., 2024; Computations, 2024; Zheng et al., 2024; TokenBender, 2024), each contributing to the diversity and quality of instruction tuning. All of the datasets have a permissive license for the training LLM. For synthetic instruct data, we use LLaMA3.1-70B as the base model to generate high-quality data, leveraging OSS-Instruct (Wei et al., 2024b) and Self-CodeAlign (Wei et al., 2024a) as synthesis methods. This approach enables us to create a large-scale instruct dataset totaling 5B tokens, further enhancing the model's ability to follow instructions

```
def encode_cyclic(s: str):
    """
    returns encoded string by cycling groups of three characters.
    """
    # split string to groups. Each of length 3.
    groups = [s[(3 * i):min((3 * i + 3), len(s))] for i in range((len(s) + 2) // 3)]
    # cycle elements in each group. Unless group has fewer elements than 3.
    groups = [(group[1:] + group[0]) if len(group) == 3 else group for group in groups]
    return "".join(groups)


def decode_cyclic(s: str):
    """
    takes as input string encoded with encode_cyclic function. Returns decoded string.
    """
    # split string to groups. Each of length 3.
    groups = [s[(3 * i):min((3 * i + 3), len(s))] for i in range((len(s) + 2) // 3)]
    # cycle elements in each group. Unless group has fewer elements than 3.
    groups = [(group[1:] + group[0]) if len(group) == 3 else group for group in groups]
    groups = [(group[-1] + group[:-1]) if len(group) == 3 else group for group in groups]
    return "".join(groups)
```

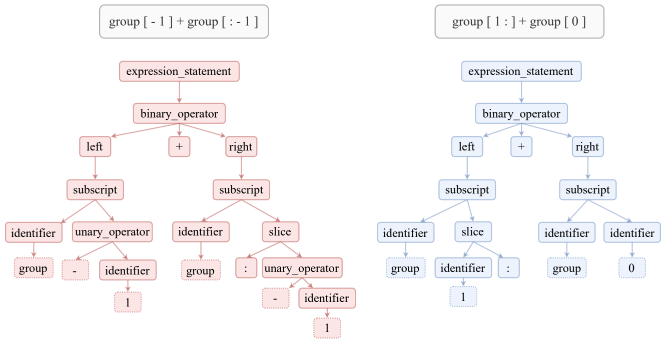

Figure 4: DeepSeek-Coder-1.3B-Base (CPT)'s generated output for Task 38 in the HumanEval dataset (left) and the required AST modifications to correct the code (right).

```
def get_max_triples(n):
    """
    You are given a positive integer n. You have to create an integer array a of length n.
    For each i (1 ≤ i ≤ n), the value of a[i] = i * i - i + 1.
    Return the number of triples (a[i], a[j], a[k]) of a where i < j < k,
    and a[i] + a[j] + a[k] is a multiple of 3.

    Example :
    Input: n = 5
    Output: 1
    Explanation:
    a = [1, 3, 7, 13, 21]
    The only valid triple is (1, 7, 13).
    """
    count = 0
    for i in range(1, n):
        for j in range(i+1, n):
            for k in range(j+1, n+1):
                if (i*i - i + 1) + (j*j - j + 1) + (k*k - k + 1) % 3 == 0:
                if ((i*i - i + 1) + (j*j - j + 1) + (k*k - k + 1)) % 3 == 0:
                    count += 1
    return count
```

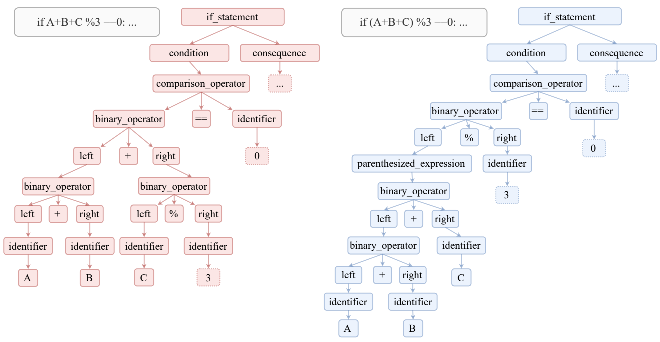

Figure 5: DeepSeek-Coder-1.3B-Base (CPT)'s generated output for Task 147 in the HumanEval dataset (left) and the required AST modifications to correct the code (right). For clarity, we represent identical computational units before and after modification using A, B, and C, respectively.

effectively.

To ensure data quality, we apply data filtering for both base models and instruct models, primarily focusing on deduplication and syntax validation. Deduplication is performed through string-based text matching to eliminate redundant samples. For syntax validation, we use Tree-sitter (TreeSitter, 2024) to check whether the code can be parsed into a valid syntax tree; if parsing fails, the sample is removed. These filtering steps help maintain a high-quality and diverse instruction dataset for training.

B Experimental Details

B.1 Distribution of Edit Distance

Figure 3 shows the edit distance distribution for error-correct code pairs with small edit distances (less than 50, accounting for 91.18% of the test set) under different code representation approaches.

B.2 Analysis of Model Outputs

While we have examined the differences between representations on existing datasets, it is also crucial to analyze whether grammar-based representation amplifies token-level subtle differences in the model's generated outputs. Therefore, we further analyzed the inference results of Meta-Llama-3.1-70B (Dubey et al., 2024) on the CodeNanoFix dataset, focusing on the edit distance between correct and incorrect code samples for the same data samples. The results show that in 25.56% of the samples, the token-level edit distance between incorrect and correct code is relatively small (less than 50). Among these samples, the average edit distance for token-based representations is 28.04, whereas for grammar-based representations, it increases to 44.56. These findings suggest that even for a 70B-scale model, generating the correct code remains challenging when token-level differences are minimal. Relying solely on token-level information may not be sufficient to distinguish critical semantic differences in code. In contrast, grammar-based representations provide additional structural information, helping the model better differentiate between similar yet semantically distinct code snippets.

B.3 Errors caused by subtle differences.

Figures 4 and 5 illustrate errors made by the token-based LLM (DeepSeek-Coder-1.3B-Base (CPT)) on the HumanEval dataset, highlighting how these mistakes can be corrected with minimal token-level modifications. For example, in Figure 4, fixing the error requires only adjusting the range of operations within the 'group' list, while in Figure 5, the bug can be fixed by adding a single pair of parentheses to enforce the correct order of operations.

However, since these examples require only minor token-level modifications, they may be overlooked by token-based LLMs. In contrast, grammar-based representations introduce larger structural changes in the corresponding AST, making the model more sensitive to differences between correct and incorrect code. These examples demonstrate that grammar-based models, by explicitly organizing code through grammar rules, can better capture subtle code variations. As a result, grammar-based models are more effective in recognizing and generating correct code, even in cases where small token-level changes drastically alter program behavior.

C Limitations

While grammar-based representations excel in code understanding and generation, they might face the following limitations. First, they may struggle with non-standard or incomplete code. Real-world datasets often contain code mixed with natural language or truncated snippets, which may fail AST parsing, reducing data utilization. Second, grammar-based models may struggle with incomplete syntax. When dealing with incomplete variable names or missing key symbols (e.g., brackets, commas), grammar-based approaches may face higher parsing or pre-processing costs. In these cases, token-based approaches offer greater flexibility.

Generally, grammar-based representation remains effective in billion-scale LLMs, enhancing the model's ability to capture subtle semantic changes. This leads to improvements in code generation and semantic classification accuracy. However, its reliance on AST parsing introduces challenges in processing incomplete or syntactically incorrect code, limiting its flexibility.