Rethinking Repetition Problems in Code Generation

Anonymous ACL submission

Abstract

With the advent of neural language models, the performance of code generation has been significantly boosted. However, the problem of repetitions during the generation process continues to linger. Previous work has primarily focused on content repetition, which is merely a fraction of the broader repetition problem in code generation. A more prevalent and challenging problem is structural repetition. In structural repetition, the repeated code appears in various patterns but possesses a fixed structure, which can be inherently reflected in grammar. In this paper, we formally define structural repetition and propose an efficient decoding approach called RPG, which stands for Repetition Penalization based on Grammar, to alleviate the repetition problems in code generation for LLMs. Specifically, RPG first leverages grammar rules to identify repetition problems during code generation, and then strategically decays the likelihood of critical tokens that contribute to repetitions, thereby mitigating them in code generation. To facilitate this study, we construct a new dataset CodeRepetEval to comprehensively evaluate approaches for mitigating the repetition problems in code generation. Extensive experimental results demonstrate that RPG substantially outperforms the best-performing baselines on CodeRepetEval dataset as well as HumanEval and MBPP benchmarks, effectively reducing repetitions and enhancing the quality of generated code.

1 Introduction

Code generation seeks to automatically produce code that aligns with user intents, which is a research hotspot in the fields of artificial intelligence, natural language processing, and software engineering (Le et al., 2022; Chen et al., 2023; Liu et al., 2023). In recent years, the emergence of neural language models has shown remarkable advancements in code generation (Chen et al., 2021; OpenAI, 2023). However, even well-trained large language models (LLMs) may suffer from repetition problems, which hurts the code generation quality of LLMs substantially (Liu et al., 2024).

Recent studies about repetition problems of LLMs are primarily focused on content repetition (Xu et al., 2022; Li et al., 2023a), which refers to the results of the generation system always containing duplicate fragments (Fu et al., 2021a). However, our preliminary investigation of LLM's repetition problem in code generation reveals that content repetition constitutes only a minor portion of them, as shown in Figure 1. In contrast, a predominant form of repetition in the generated results involves the repeated occurrence of similar codes with fixed structural patterns, which we term 'structural repetition'[1]. Distinct from content repetitions, the pattern of different structural repetitions varies markedly (e.g., structural repetitions I-IV), making structural repetitions hard to detect and handle. Given the diversity and complexity of structural repetitions, previous approaches tailored for content repetitions are insufficient to address them effectively. Therefore, it is necessary and significant to explore the structural repetitions in code generation.

In this paper, we propose an effective decoding approach RPG: **R**epetition **P**enalization based on **G**rammar, to alleviate repetition problems of LLMs in code generation. Considering different code fragments with the same structural patterns can be represented by identical grammar rules, RPG employs the pushdown automaton built on grammar rules to detect repetition problems during the generation process, and then strategically decreases the likelihood of key tokens that contribute to repetitions. RPG offers two main benefits: 1) it curtails the endless generation of meaningless, repetitive code, thereby saving tokens and time-consuming;

[1]Generally, in code generation, content repetition can be regarded as a special type of structural repetition. However, in this paper, structural repetition is defined to exclude content repetition for clarity and distinction.

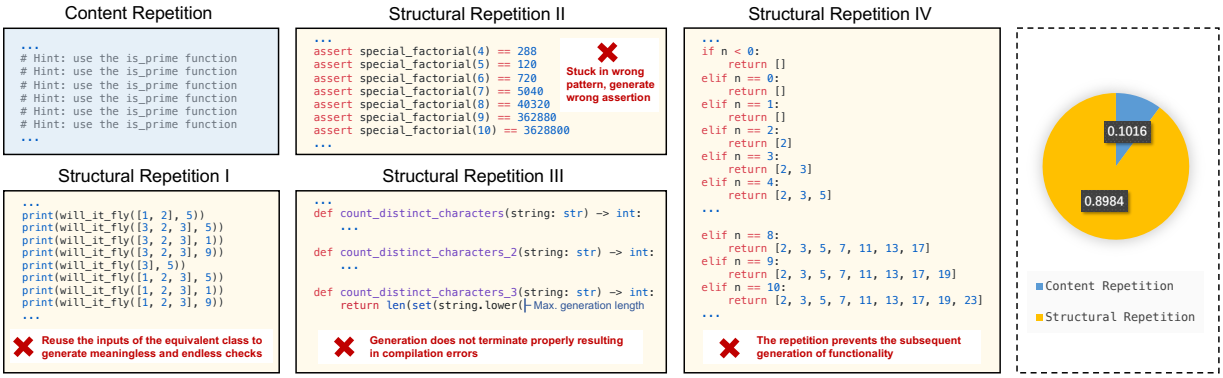

Figure 1: Examples of repetition problems in code generation, collected from the well-trained LLMs, e.g., CodeLlama (Rozière et al., 2023) and ChatGPT (OpenAI, 2022) (**Left**). The statistical percentage of two repetition forms occurs in the generated code of LLMs (**Right**).

2) it realigns the LLMs' generation back to the correct generation path, enhancing the quality of code generation. Moreover, we construct a new dataset, named CodeRepetEval, for evaluating approaches to mitigate the repetition problems in code generation. Extensive experimental results and analyses verify the effectiveness and generality of RPG.

Our main contribution can be summarized as fourfold. 1) We first formally define structural repetitions, which are more prevalent than content repetitions in code generation. 2) We present RPG, a novel decoding approach that leverages pushdown automaton to identify and mitigate repetitive problems in code generation from grammar perspective. 3) We construct CodeRepetEval dataset covering three scenarios, with data derived from artificial synthesis, code generation benchmarks, and real-world repositories, to facilitate subsequent research for repetitive problems in code generation. 4) RPG substantially outperforms the best-performing baselines in various scenarios, which alleviates both structural and content repetitions, and achieves better code generation quality.

2 Motivation Example

The repetition problem in code generation remains an underexplored challenge, usually resulting in redundancy and errors. Figure 2 showcases an example where LLMs generate the code containing structural repetitions. This generated code is plagued by repetitions with the fixed structural pattern starting with 'elif'. In each repetition, LLMs generate different conditions following the start token 'elif' and varying statements under these conditions. Despite the content of each repetition differing, both the probability of the start token and the average probabilities of all tokens in each repetition exhibit an upward trend as the number of repetitions increases, showing a self-reinforcement effect as the right side of Figure 2. This phenomenon means that the start token in each repetition will serve as an anchor point. As the model continues to generate code, it relies on this anchor, reinforcing its choice and making it increasingly difficult to diverge from the structural repetition. This ultimately leads to the subsequent generation getting stuck in endless and meaningless (or even erroneous) repetitions.

According to principles of compilation (Alfred et al., 2007), we discover that massive patterns of structural repetition are inherently reflected in grammar, i.e., the potential positions where code can be repeated are determined by explicit grammar rules. For example, the structural repetitions of the code in Figure 2 adhere to *('elif' test ':' suite)** within *if_stmt* of grammar, where $*$ denotes that its preceding expression can be repeated zero or more times. Although the grammar rules impose no limit on the number of repetitions, human-written code are not repeat endlessly, with the higher the number of repetitions, the lower the likelihood of their occurrence. Therefore, the prediction confidence of further repetition ought to decrease with a growing number of repetitions, rather than exhibiting the self-reinforcement usually observed in LLMs.

In this paper, we first formally define structural repetitions, and propose RPG to effectively detect and alleviate them in code generation, thereby enhancing the quality of generated code for LLMs.

3 Definition of Structural Repetition

Given a token sequence $X = [x_1, x_2, \cdots, x_{|X|}]$, where x_i denotes the i-th token and $|X|$ rep-

```
Grammar     if_stmt: 'if' test ':' suite ('elif' test ':' suite)* ['else' ':' suite]
```

```
def count_up_to(n):
    """Implement a function that takes an
non-negative integer and returns an array of
the first n integers that are prime numbers
and less than n.
    """
    -----

    if n < 0:
        return []
    elif n == 0:
        return []
    elif n == 1:
        return []
    elif n == 2:
        return [2]
    elif n == 3:
        return [2, 3]
    elif n == 4:
        return [2, 3, 5]
    elif n == 5:
        return [2, 3, 5, 7]
    ...
```

Structural Repetition

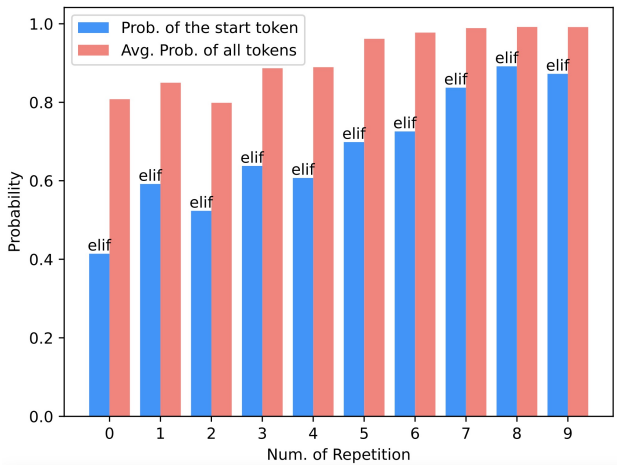

Figure 2: A case of structural repetition generated by CodeLlama with temperature = 0, where the dashed-underline text is the prompt (**Left**). The corresponding grammar rules of structural patterns (**Top**). LLM's probabilities of generated tokens in each repetition (**Right**).

resents the length of X. We denote $X_{p:q} = [x_p, x_{p+1}, \cdots, x_q]$, where $(1 \leq p < q \leq |X|)$, as a continuous subsequence of X. Given a mapping function G to represent the underlying structure of X, the mapped sequence $\hat{R} = G(X)$ is obtained by applying G to X, where $\hat{R} = [\hat{r}_1, \hat{r}_2, \cdots, \hat{r}_{|\hat{R}|}]$. In this paper, G indicates the context-free grammar[2], which can be defined as a quad tuple (N, Σ, P, S), where N is a set of non-terminal symbols, Σ is a set of terminal symbols, representing the basic symbols of the language, P is a set of production rules, with each rule in the form $A \rightarrow \beta$, where $A \in N$ and β is a sequence of elements from $N \cup \Sigma$, and $S \in N$ is the start symbol, used to begin derivations of strings.

Specifically, the generated sequence X will be reduced in accordance with G, ensuring that the reduction of X does not exceed the statement levels of grammar rules, which includes twelve simple statements and nine compound statements[3]. Thus, the patterns of structural repetitions within X can be defined as:

$$\mathrm{SR}(X) = \{\hat{R}_{p:q} | \exists 1 \leq p < q \leq |\hat{R}| - q + p, \forall i \in [p, q], \hat{r}_{i+q-p} = \hat{r}_i\}, \quad (1)$$

[2]Programming languages (such as Python, Java, C++, etc.) belong to context-free languages, which means that they are governed by context-free grammars.

[3]For instance, simple statements contain 'expr_stmt', 'return_stmt', 'raise_stmt', 'import_stmt', 'assert_stmt', etc. Compound statements contain 'if_stmt', 'while_stmt', 'for_stmt', 'try_stmt', 'funcdef', 'classdef', etc. Detailed descriptions can be found in Appendix F.

where structural repetitions exist in X, if $\mathrm{SR}(X) \neq \emptyset$, and the elements in $\mathrm{SR}(X)$ represent the patterns of structural repetitions. For example, if X denotes the generated code in Figure 2, then $\hat{R}$ would be "$\cdots$ *'elif' test ':' suite 'elif' test ':' suite* $\cdots$". Thus, $\hat{R}_{p:q}$ can be the subsequence of $\hat{R}$, i.e., "*'elif' test ':' suite*", where $\forall i \in [p, q], \hat{r}_{i+q-p} = \hat{r}_i$. Note that for the same input, G has a unique output, i.e., if content repetitions exist in X, the repetitions will be preserved in $G(X)$ as well, which implies content repetitions can also be detected by Eq. (1).

Structural repetition negatively impacts code generation in the following two ways: 1) it fails to terminate properly, rendering the code uncompilable. As repetition persists, the generation of LLMs becomes meaningless or incorrect gradually; 2) it disrupts the generation of code, leading to the absence of a portion of functionality, thereby severely hurting the quality of generated code.

4 Methodology

In this section, we will introduce RPG in detail, including three parts, i.e., Reduction to Grammar Rules (§ 4.1), Detection of Repetition (§ 4.2), and Penalization of Repetition (§ 4.3).

4.1 Reduction to Grammar Rules

In problems of structural repetition, while the forms of repeated statements vary, their underlying grammar rules demonstrate similarities. Following the previous work (Dong et al., 2023), we employ

the pushdown automaton (PDA) for the reduction of generated codes into their underlying grammar rules during code generation[4]. A PDA can be defined as a seven tuple $(Q, \Sigma, \Gamma, \delta, q_0, Z, F)$, where Q, Σ, Γ are finite sets representing the states, input symbols, and stack symbols, respectively. q_0 is the initial state, z_0 is the initial stack symbol, and A is the set of accepting states, with $q_0 \in Q$, $z_0 \in \Gamma$, and $A \subseteq Q$. $\delta : Q \times (\Sigma \cup \{\epsilon\}) \times \Gamma \rightarrow \mathcal{P}(Q \times \Gamma^*)$ is the transition function, where ϵ denotes the empty string, and Γ^* represents all sequences of stack symbols. Based on δ of PDA, we have

$$q_t, z_t = \delta(q_{t-1}, x_t, z_{t-1}), \tag{2}$$

where q_t is the state of t-th time step, z_t is the stack symbol of t-th time step, and x_t is the generated token of t-th time step. According to q_t and z_t, x_t can be reduced to its corresponding grammar rule uniquely, which is expressed as:

$$\hat{x}_t = g(x_t) = [q_t, z_t], \tag{3}$$

We merge the same adjacent parts in $\hat{X}_{1:t} = [\hat{x}_1, \hat{x}_2, \cdots, \hat{x}_t]$ to get the final reduction sequence of grammar rules for x_t.

$$\hat{R}_{1:t} = \text{merge}(\hat{X}_{1:t}), \tag{4}$$

For example, multiple adjacent generated tokens in Figure 2 belong to the same terms such as 'test' and 'suite', the adjacent ones will be merged into a single entity for each term.

4.2 Detection of Repetition

Considering that once repetition problems arise in code generation, they tend to persist until the end of the generation process, we need to detect these repetition problems as they emerge during code generation. To detect the repetitions in $\hat{R}_{1:t}$, we employ suffix arrays and longest common prefix (LCP) arrays, which are efficient with both time and space complexities of $O(n)$. The pseudo-code for suffix arrays and the LCP array is provided in Appendix D.

Suffix Array: The suffix array is an array of integers representing the starting indices of the suffixes of $\hat{R}_{1:t}$, sorted in lexicographical order. Thus, Suf$[i]$ points to the starting index of the i-th smallest suffix in $\hat{R}_{1:t}$.

[4]We adapt PDA to accommodate the Byte-Pair Encoding (BPE) (Sennrich et al., 2016) tokenization used by LLMs. Detailed descriptions can be found in Appendix E.

Longest Common Prefix Array: The LCP array is defined such that LCP$[i]$ is the length of the longest common prefix between suffixes starting at Suf$[i-1]$ and Suf$[i]$ for all $1 \leq i < n$ (with LCP$[0]$ typically set to 0 for convenience).

Using the LCP array, we identify all positions of $\hat{R}_{1:t}$ where LCP$[i] > 0$. These positions indicate the presence of repetitions of length at least LCP$[i]$. Therefore, the repetition patterns within $X_{1:t}$ can be expressed as:

$$\begin{aligned}\text{Rep}(X_{1:t}) = \{&\hat{R}_{\text{Suf}[i]:\text{Suf}[i]+\text{LCP}[i]} \mid \forall i \in [1, t-1], \\ &\text{LCP}[i] > 0 \ \wedge \ \text{Suf}[i-1] = \text{Suf}[i] + \text{LCP}[i]\},\end{aligned} \tag{5}$$

4.3 Penalization of Repetition

Given the identified repetitions, RPG applies a penalization mechanism to discourage the model from generating them in future outputs. The penalization mechanism integrates into the code generation model's scoring function, modifying how tokens are weighted during the generation process.

Dynamic Weight Adjustment: We define a dynamic weight function, Pn$(\cdot)$, which applies a decreasing factor to the score of a token based on the frequency and recency of its associated grammar rule in the sequence $\hat{R}_{1:t}$. The weight for each token is adjusted as follows:

$$\text{Pn}(x_t|x_{<t}) = \lambda^{\text{Count}(\text{Rep}(X_{1:t}))}, \tag{6}$$

where $x_{<t}$ represents $X_{1:t-1}$, λ is a decay factor between 0 and 1, and Count(Rep$(X_{1:t})$) is the count of times the repetition patterns in Rep$(X_{1:t})$ has appeared. This exponential decay effectively reduces the likelihood of selecting tokens associated with repetitive grammar rules.

Token Scoring Adjustment: During the code generation, each token's score is recalculated by incorporating the dynamic weight:

$$s'(x_t|x_{<t}) = s(x_t|x_{<t}) \cdot \text{Pn}(x_t|x_{<t}), \tag{7}$$

where $s(x_t)$ is the original score of the token x_t provided by the model, and $g(x_t)$ is the grammar rule associated with x_t. The adjusted score $s'(x_t)$ influences the token selection process, guiding the model toward less repetitive and more diverse code generation.

Finally, our RPG approach can be defined as:

$$\text{RPG}(x_t|x_{<t}) = \arg\max_{x_t} s'(x_t|x_{<t}) \tag{8}$$

5 Experiment Setup

In this section, we will provide the setups of our experiments below. The detailed description of experiment setups can be found in Appendix C.

5.1 Datasets

Considering the absence of datasets for repetition problems in code generation, we dedicate more than 400 hours to constructing and examining **CodeRepetEval** dataset. We simulate repetition problems in code generation covering three scenarios, including artificial synthesis, code generation benchmarks, and real-world repositories. Specifically, **Artificial Synthesis** scenario involves 512 test samples. Each sample consists of a correct code concatenated with its last repetition patterns 5 to 10 times. **Code Generation Benchmarks** scenario comprising 512 test samples, which are selected from the generated repetitive codes of three LLMs (i.e., CodeLlama (Rozière et al., 2023), DeepSeek Coder (Guo et al., 2024), CodeGen (Nijkamp et al., 2023), and ChatGPT (OpenAI, 2022)) on HumanEval and MBPP benchmarks. **Real-world Repositories** scenario includes 1024 test samples, picked from the partial code in real-world repositories, which is identified to induce repetition problems in the generated outputs of the aforementioned LLMs. We employ CodeRepetEval to assess the effectiveness of RPG for addressing repetition problems in code generation scenarios.

Moreover, We also involve four public benchmarks to evaluate the RPG's performance in code generation, including **HumanEval** (Chen et al., 2021), **MBPP** (Austin et al., 2021), as well as their extended version **HumanEval-ET** and **MBPP-ET** (Dong et al., 2024).

5.2 Baselines

As our approach is based on decoding that does not require modification and training of the model, we compare it to the four most commonly used decoding approaches, including **Greedy Sampling**, **Topk Sampling** (Fan et al., 2018), **Temperature Sampling** (Caccia et al., 2019), **Topp Sampling** (Holtzman et al., 2020). Furthermore, we also compare two representative baselines for addressing content repetition in text generation, including **Repetition Penalty** (Keskar et al., 2019) for the decoding phase and **Repetition Dropout** (Li et al., 2023a) applied on the training phase. These baselines follow the settings in their original paper.

5.3 Metrics

We mainly use six metrics to evaluate approaches for addressing the repetition problems in code generation. **End-of-sentence Generation Percentage (EGP)** quantifies the frequency with which a model successfully interrupts repetitive sequences to conclude generation, which is determined by calculating the proportion of end-of-sentence (EOS) tokens across all samples generated by the model. **TR-N** is calculated to measure structural repetitions within a generated sequence at phrase-level, which is defined as $1.0 - \frac{|\{G(x)' | \exists p \in [1, |G(x)|-n+1], G(x)' = G(x)_{p:p+n-1}\}|}{|G(x)|-n+1}$, where $|G(x)|$ means the number of elements in $G(x)$. It effectively quantifies the proportion of duplicate n-grams present in its underlying grammar rules, and $n = 4$ in this paper. In contrast, **TR-S** measures structural repetitions within a generated sequence at statement-level, which is defined as $1.0 - \frac{|\text{unique statements in } G(x)|}{|\text{statements in } G(x)|}$. **Compiler Correctness Percentage (CCP)** evaluates whether the generated code is compilable, which is measured by the proportion of code samples that successfully compile. We use **Time** and **GenLen** to verify the approaches' efficiency, which means the average time of model generation and the average length of model-generated outputs, respectively.

For HumanEval(-ET) and MBPP(-ET) benchmarks which contain the test cases, we employ **Pass@k** (Li et al., 2022) metric to measure the functional correctness of the generated code by executing test cases.

5.4 Implementation Details

In this paper, all experiments are conducted on an A6000 GPU (48GB). We employ CodeLlama-7B as our base model. The decay factor λ for the penalization of repetition in RPG is set at 0.9 by default. The maximum token length of each approach is set to 1024 in all datasets and scenarios, except for CodeRepetEval (real-world repository) setting to 4096. Following the previous work (Chen et al., 2021; Rozière et al., 2023), the default temperature for the baselines is set at 0.8. To mitigate the instability of the model sampling, we report the average results of five trials in the experiments.

6 Experimenal Results

We systematically evaluate our approach from two main aspects. First, regarding the effectiveness of mitigating repetition problem in code gen-

Table 1: Comparison of RPG with commonly used decoding approaches and representative content repetition baselines on CodeRepetEval dataset in three scenarios.

Approach	CodeRepetEval					
	EGP ↑	TR-N ↓	TR-S ↓	CCP ↑	Time ↓	GenLen
Code Generation Benchmarks						
Rep_Penalty	0.721	0.374	0.425	0.413	16.88	689
Rep_Dropout	0	0.569	0.536	0.218	35.65	1024
Greedy	0	0.598	0.637	0.455	33.87	1024
Temp (t=0.1)	0.007	0.599	0.622	0.39	37.72	1019
Temp (t=0.2)	0.03	0.603	0.63	0.413	36.13	950
Temp (t=0.8)	0.578	0.423	0.441	0.433	27.13	755
Topk (k=5)	0.536	0.465	0.484	0.394	20.61	763
Topk (k=10)	0.547	0.441	0.464	0.421	29.11	752
Topk (k=30)	0.628	0.415	0.443	0.442	33.99	737
Topp (p=0.8)	0.046	0.559	0.549	0.379	35.17	995
Topp (p=0.9)	0.102	0.53	0.512	0.391	42.03	966
Topp (p=0.95)	0.114	0.508	0.518	0.399	30.99	959
RPG (Ours)	**0.912**	**0.352**	**0.391**	**0.805**	**13.68**	**565**
Artificial Synthesis						
Rep_Penalty	0.679	0.624	0.521	0.467	20.91	615
Rep_Dropout	0	0.713	0.597	0.188	32.89	1024
Greedy	0	0.807	0.789	0.459	40.45	1024
Temp (t=0.1)	0.016	0.798	0.785	0.452	40.16	1010
Temp (t=0.2)	0.018	0.798	0.768	0.438	39.95	982
Temp (t=0.8)	0.522	**0.613**	**0.519**	0.433	23.30	675
Topk (k=5)	0.503	0.659	0.558	0.45	24.35	723
Topk (k=10)	0.552	0.636	0.524	0.48	22.42	661
Topk (k=30)	0.561	0.628	0.521	0.475	22.00	653
Topp (p=0.8)	0.097	0.752	0.691	0.431	37.27	946
Topp (p=0.9)	0.146	0.724	0.658	0.501	35.82	915
Topp (p=0.95)	0.201	0.681	0.637	0.454	33.19	868
RPG (Ours)	**0.871**	0.618	0.556	**0.731**	**17.37**	**489**
Real-world Repositories						
Rep_Penalty	0.828	0.461	0.358	0.395	62.01	1753
Rep_Dropout	0	0.705	0.519	0.074	208.51	4096
Greedy	0	0.738	0.631	0.297	203.96	4096
Temp (t=0.1)	0.031	0.726	0.62	0.292	189.35	3997
Temp (t=0.2)	0.052	0.728	0.627	0.309	185.38	3944
Temp (t=0.8)	0.769	0.496	0.384	0.365	64.72	1926
Topk (k=5)	0.732	0.534	0.427	0.411	64.58	1937
Topk (k=10)	0.783	0.51	0.399	0.381	63.29	1911
Topk (k=30)	0.783	0.495	0.386	0.372	64.72	1940
Topp (p=0.8)	0.128	0.686	0.585	0.344	172.11	3728
Topp (p=0.9)	0.221	0.658	0.55	0.349	155.82	3443
Topp (p=0.95)	0.291	0.642	0.521	0.373	144.96	3270
RPG (Ours)	**0.889**	**0.416**	**0.335**	**0.638**	**60.36**	**1415**

eration, we conduct multi-angle evaluations on CodeRepetEval dataset: 1) We compare the performance of our RPG approach with baselines in three scenarios: Code Generation Benchmarks, Artificial Synthesis, and Real-world Repositories; 2) We valid the effect of RPG on the base LLMs across different series and sizes; 3) We explore the generalizability of RPG across different programming languages (PLs). 4) We evaluate the impact of different values of hyperparameter λ on the performance of RPG. 5) We conduct a case study to qualitatively analyze the RPG's performance (Appendix A). Second, concerning the effectiveness of code generation, we evaluate the RPG's performance compared to baselines on HumanEval(-ET) and MBPP(-ET) benchmarks.

6.1 Repetition Mitigation

Effectiveness of RPG. As illustrated in Tables 1, we assess our RPG approach along with various baselines on CodeRepetEval dataset. Our analysis of experimental results yields several insights: 1) **Enhancing the model's confidence in its output tends to improve the likelihood of generating repetitive sequences.** Specifically, when the hyperparameters for temperature, Topk, and Top-p sampling are set to lower values, the TR-N and TR-S metrics across the three scenarios of CodeRepetEval dataset show poorer performance, and the generation of EOS tokens becomes more challenging, thereby increasing the generation time and length of models. 2) **The approaches for addressing content repetition in text generation are not applicable to the structural repetition problems in code generation.** Although Repetition Penalty can reduce TR-N and TR-S to a certain extent, they achieve this at the expense of generated code quality. Its generated code usually terminates prematurely at a position where it should not end. Repetition Dropout masks the content repetitions for self-attention during training, but it has little effect on the structural repetitions. 3) **RPG substantially outperforms other baselines on CodeRepetEval dataset, which effectively reduces repetitions and enhances the quality of the generated code.** RPG achieves the best performance in terms of EGP and CCP metrics on CodeRepetEval dataset across three scenarios. For the TR-N and TR-S metrics, RPG also achieves the optimal performance except for Artificial Synthesis scenario. This may be attributed to the fact that although artificially synthesized prompts tend to induce repetitions, since these prompts are not inherently natural to the model, the model more readily escapes these repetitions when sampling from a smoother distribution.

Performance on different LLMs. We also conduct experiments for evaluating the performance of RPG based on different LLMs across three scenarios of CodeRepetEval dataset, as presented in Table 2. The experimental results indicate that RPG exhibits consistent and significant enhancements on different base models, highlighting the robustness of RPG for base model selections. Moreover, we find that training LLMs on code is likely to increase their susceptibility to structural repetitions during greedy sampling. Given that CodeLlama, although fine-tuned on Llama2 for coding tasks, demon-

Table 2: The impact of RPG using different base models on CodeRepetEval dataset across three scenarios, where AS, CGB, and RR donate Artificial Synthesis, Code Generation Benchmarks, and Real-world Repositories respectively.

Approach	AS		CGB		RR	
	TR-S ↓	CCP ↑	TR-S ↓	CCP ↑	TR-S ↓	CCP ↑
CodeLlama						
Greedy	0.789	0.459	0.637	0.455	0.631	0.297
RPG	**0.613**	**0.731**	**0.435**	**0.805**	**0.379**	**0.638**
CodeGen						
Greedy	0.684	0.541	0.657	0.509	0.657	0.369
RPG	**0.437**	**0.841**	**0.417**	**0.867**	**0.375**	**0.720**
DeepSeek-Coder						
Greedy	0.453	0.821	0.664	0.479	0.467	0.332
RPG	**0.369**	**0.951**	**0.515**	**0.732**	**0.341**	**0.585**
Llama2						
Greedy	0.528	0.758	0.508	0.597	0.542	0.477
RPG	**0.422**	**0.917**	**0.427**	**0.912**	**0.383**	**0.764**

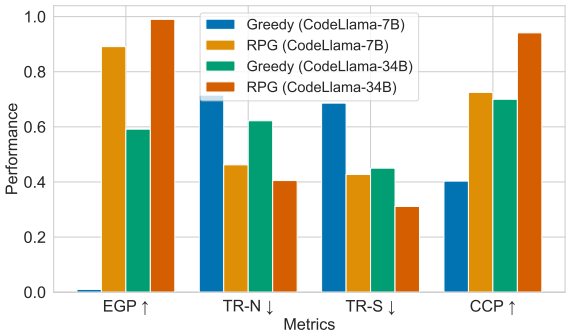

Figure 3: The performance of RPG applied to LLMs of different sizes. This result is the average value across three scenarios on CodeRepetEval dataset.

strates markedly worse performance in terms of structural repetitions in code generation.

As shown in Figure 3, we observe that the repetition also occurs on LLMs larger than 7B, i,e., CodeLlama-34B, and RPG is effective for it as well, which demonstrates the same trends as other LLMs evaluated in Table 1 and Table 2, indicating the generalizability across LLMs of varying sizes.

Table 3: The performance of RPG on CodeRepetEval-Go dataset.

Approach	EGP ↑	TR-N ↓	TR-S ↓	CCP ↑	Time ↓
Greedy	0.133	0.750	0.353	0.403	38.28
Temp (t=0.8)	0.601	0.554	0.231	0.396	26.78
RPG (Ours)	**0.875**	**0.518**	**0.215**	**0.725**	**21.07**

Performance on different PLs. RPG can be applied to other PLs, requiring only their grammatical rules, which are readily obtainable from the web. To demonstrate the convenience of RPG, we have extended it to the PL of Go, and the experimental results are shown in Table 3. We can find that RPG still achieved substantial improvements in all five metrics for the PL of Go.

Influence of hyperparamter λ. In our experiments, we fix the hyperparameter λ intuitively for RPG. As shown in Figure 5 of Appendix B, we investigate the influence of varying λ empirically on all scenarios of CodeRepetEval dataset and HumanEval and MBPP benchmarks by changing itself. The results indicate that the reduction of the hyper-parameter λ for RPG leads to a more pronounced suppression of repetition problems in code generation and decreases the sampling time. Furthermore, there is still room for further improvements with the better hyper-parameter setup of λ.

Case Study. In Figure 4 of Appendix A, we showcase two examples from Code Generation Benchmarks and Real-World Repository to conduct qualitative analysis. In these cases, the original model falls into a repetition trap, continuing until it exhausts its token budget. Case (a) features a repetition pattern: num = [i for i in num if i != NUM], where LLMs repeatedly generate statements that increment NUM. Case (b)'s repetition pattern involves a sequence of imports, where modules 'attention' and 'conv' are endlessly added. The code generated under these repetition patterns is utterly nonsensical and fails completely. However, our approach can break out of these loops, returning to a normal code generation trajectory, and ultimately succeeding in generating correct code.

Table 4: The performance of RPG on code generation benchmarks HumanEval(-ET) and MBPP(-ET). Improvement represents the relative improvement of RPG compared to Greedy sampling.

Approach	HumanEval	HumanEval-ET	MBPP	MBPP-ET
Rep_Penalty	0.092	0.067	0.143	0.115
Rep_Dropout	0.139	0.116	0.167	0.141
Greedy	0.301	0.232	0.396	0.303
Temp (t=0.8)	0.226	0.183	0.305	0.218
RPG (Ours)	**0.325**	**0.258**	**0.421**	**0.334**
Improvement	**↑ 8.0%**	**↑ 11.3%**	**↑ 6.4%**	**↑ 10.3%**

6.2 Code Generation

In addition to CodeRepetEval dataset, we further validate the effectiveness of RPG on widely used code generation benchmarks, i.e., HumanEval(-ET) and MBPP(-ET), as presented in Table 4.

The results demonstrate that RPG outperforms both standard decoding approaches and specialized approaches aimed at reducing content repetition. Although Repetition Penalty and Repetition Dropout forcibly reduce content repetition in generated code, they also significantly impair the performance of code generation. In contrast, RPG not only effectively eliminates both content and structural repetition, but also enhances the accuracy of code generation for LLMs, achieving relative improvements of up to 11.3% in Pass@1.

7 Related Work

In this section, we outline the two most relevant directions and associated papers of this work.

7.1 Code Generation

Since the advent of artificial intelligence in the 1950s, code generation has been considered the Holy Grail of computer science research (Gulwani et al., 2017). With the rapid expansion of codebases and the increasing capacity of deep learning models, using deep learning for program generation has shown great potential and practicality (Raychev et al., 2014; Ling et al., 2016; Wei et al., 2019; Sun et al., 2020; Mukherjee et al., 2021). In recent years, the rise of pre-training techniques has brought new momentum to the field of code generation. For example, studies like CodeT5 (Wang et al., 2021) and UniXcoder (Guo et al., 2022) pre-train models for code generation tasks. With the continual increase in model parameters, researchers have discovered emergent phenomena in LLMs, leading to new breakthroughs. Against this backdrop, LLMs such as AlphaCode (Li et al., 2022), Codex (Chen et al., 2021), CodeGeeX (Zheng et al., 2023), Starcoder (Li et al., 2023b), CodeLlama (Rozière et al., 2023), and DeepSeek Coder (Guo et al., 2024) have emerged.

Some work focuses on grammar-based code generation approaches (Yin et al., 2018; Sun et al., 2019; Jiang et al., 2021; Dong et al., 2023), which primarily utilize learning or decoding based on grammar rulers to enhance the grammatical correctness of generated code. However, given that all structural repetitions adhere to the grammar, i.e., they are grammatically correct, merely using grammar rules during decoding or learning grammar rules during training is not applicable to the structural repetition problem. Therefore, these approaches fail to address this problem.

7.2 Repetition in Neural Text Generation.

Repetition problems in neural language models have drawn increasing attention, with various interpretations and proposed solutions emerging from recent research, especially in the field of text generation (Holtzman et al., 2020). Repetition Penalty (Holtzman et al., 2020) is a commonly used approach to reduce content repetition, which prevents words or phrases that have already appeared during the generation process from being generated again. However, there are lots of key tokens that appear frequently in code generation, such as '=', '(', '[' (Eghbali et al., 2022). Uniformly preventing these tokens in subsequent generations would be extremely detrimental to code generation.

Previous work (Fu et al., 2021b) points out that repetition is caused by the phenomenon of self-reinforcement. Some works address this problem during the training phase (Fu et al., 2021b; Xu et al., 2022; Su et al., 2022). Repetition dropout (Li et al., 2023a) finds a link between training data degradation and repetition, mitigating it by lowering attention to repeated words. However, compared to our RPG approach, these approaches have three primary disadvantages: 1) They require extensive training and necessitate the construction of a large amount of data for fine-tuning LLMs, which incurs substantial costs. 2) They usually hurt the code generation performance of models obviously. 3) They only focus on addressing content repetitions in text generation, without involving the prevalent issue of structural repetitions in code generation.

8 Conclusion

In this paper, we have formally defined structural repetition, which is the major repetition problem in code generation. We have proposed a novel decoding approach called RPG to alleviate repetition problems in code generation from grammar perspective for LLMs. By leveraging the grammar rules, RPG can recognize repetitions and strategically decay the output probability of critical tokens that contribute to repetitions. We also construct a new dataset CodeRepetEval, designed to provide a comprehensive evaluation for addressing repetition problems in code generation. Extensive experiments demonstrate the effectiveness and generalization of RPG in repetition mitigation of code generation. Through our work, we hope to shed light on this direction and call more attention to repetition problems in code generation.

9 Limitations

Our work has the following two main limitations.

First, RPG demands slightly more computational resources than sampling to detect the repetitions. However, compared to the enormous computational overhead of LLMs, it is marginal and acceptable.

Second, the potential reasons why LLMs induce structural repetitions in code generation remain unclear. Our current analysis has not touched on this aspect, which we leave for future work.

References

V Aho Alfred, S Lam Monica, and D Ullman Jeffrey. 2007. Compilers Principles, Techniques & Tools. pearson Education.

Jacob Austin, Augustus Odena, Maxwell I. Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie J. Cai, Michael Terry, Quoc V. Le, and Charles Sutton. 2021. Program synthesis with large language models. CoRR, abs/2108.07732.

Massimo Caccia, Lucas Caccia, William Fedus, Hugo Larochelle, Joelle Pineau, and Laurent Charlin. 2019. Language gans falling short. In (ICLR).

Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. 2023. CodeT: Code generation with generated tests. In ICLR.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harrison 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, Joshua 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. CoRR.

Yihong Dong, Jiazheng Ding, Xue Jiang, Ge Li, Zhuo Li, and Zhi Jin. 2024. Codescore: Evaluating code generation by learning code execution. ACM TOSEM.

Yihong Dong, Ge Li, and Zhi Jin. 2023. CODEP: grammatical seq2seq model for general-purpose code generation. In ISSTA, pages 188–198. ACM.

Aryaz Eghbali and Michael Pradel. 2022. Crystalbleu: Precisely and efficiently measuring the similarity of code. In ICSE-Companion, pages 341–342. ACM/IEEE.

Angela Fan, Mike Lewis, and Yann N. Dauphin. 2018. Hierarchical neural story generation. In ACL (1), pages 889–898. Association for Computational Linguistics.

Zihao Fu, Wai Lam, Anthony Man-Cho So, and Bei Shi. 2021a. A theoretical analysis of the repetition problem in text generation. In AAAI, pages 12848–12856. AAAI Press.

Zihao Fu, Wai Lam, Anthony Man-Cho So, and Bei Shi. 2021b. A theoretical analysis of the repetition problem in text generation. In AAAI, pages 12848–12856. AAAI Press.

Sumit Gulwani, Oleksandr Polozov, Rishabh Singh, et al. 2017. Program synthesis. Foundations and Trends® in Programming Languages, 4(1-2):1–119.

Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. Unixcoder: Unified cross-modal pre-training for code representation. In ACL (1), pages 7212–7225. Association for Computational Linguistics.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y. Wu, Y. K. Li, Fuli Luo, Yingfei Xiong, and Wenfeng Liang. 2024. Deepseek-coder: When the large language model meets programming - the rise of code intelligence. CoRR, abs/2401.14196.

Ari Holtzman, Jan Buys, Li Du, Maxwell Forbes, and Yejin Choi. 2020. The curious case of neural text degeneration. In ICLR. OpenReview.net.

Xue Jiang, Zhuoran Zheng, Chen Lyu, Liang Li, and Lei Lyu. 2021. Treebert: A tree-based pre-trained model for programming language. In UAI, volume 161 of Proceedings of Machine Learning Research, pages 54–63. AUAI Press.

Nitish Shirish Keskar, Bryan McCann, Lav R. Varshney, Caiming Xiong, and Richard Socher. 2019. CTRL: A conditional transformer language model for controllable generation. CoRR, abs/1909.05858.

Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven Chu-Hong Hoi. 2022. Coderl: Mastering code generation through pretrained models and deep reinforcement learning. In NeurIPS.

Huayang Li, Tian Lan, Zihao Fu, Deng Cai, Lemao Liu, Nigel Collier, Taro Watanabe, and Yixuan Su. 2023a. Repetition in repetition out: Towards understanding neural text degeneration from the data perspective. In NeurIPS.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny

Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, João Monteiro, Oleh Shliazhko, Nicolas Gontier, Nicholas Meade, Armel Zebaze, Ming-Ho Yee, Logesh Kumar Umapathi, Jian Zhu, Benjamin Lipkin, Muhtasham Oblokulov, Zhiruo Wang, Rudra Murthy V, Jason Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Nour Moustafa-Fahmy, Urvashi Bhattacharyya, Wenhao Yu, Swayam Singh, Sasha Luccioni, Paulo Villegas, Maxim Kunakov, Fedor Zhdanov, Manuel Romero, Tony Lee, Nadav Timor, Jennifer Ding, Claire Schlesinger, Hailey Schoelkopf, Jan Ebert, Tri Dao, Mayank Mishra, Alex Gu, Jennifer Robinson, Carolyn Jane Anderson, Brendan Dolan-Gavitt, Danish Contractor, Siva Reddy, Daniel Fried, Dzmitry Bahdanau, Yacine Jernite, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. 2023b. Starcoder: may the source be with you! CoRR, abs/2305.06161.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. 2022. Competition-level code generation with alphacode. Science, 378(6624):1092–1097.

Wang Ling, Phil Blunsom, Edward Grefenstette, Karl Moritz Hermann, Tomás Kociský, Fumin Wang, and Andrew W. Senior. 2016. Latent predictor networks for code generation. In ACL (1). The Association for Computer Linguistics.

Fang Liu, Yang Liu, Lin Shi, Houkun Huang, Ruifeng Wang, Zhen Yang, and Li Zhang. 2024. Exploring and evaluating hallucinations in llm-powered code generation. CoRR, abs/2404.00971.

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 NeurIPS.

Rohan Mukherjee, Yeming Wen, Dipak Chaudhari, Thomas W. Reps, Swarat Chaudhuri, and Christopher M. Jermaine. 2021. Neural program generation modulo static analysis. In NeurIPS, pages 18984–18996.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2023. Codegen: An open large language model for code with multi-turn program synthesis. In ICLR. OpenReview.net.

OpenAI. 2022. ChatGPT.

OpenAI. 2023. GPT-4 technical report. CoRR, abs/2303.08774.

Veselin Raychev, Martin T. Vechev, and Eran Yahav. 2014. Code completion with statistical language models. In PLDI, pages 419–428. ACM.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, 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. 2023. Code llama: Open foundation models for code. CoRR, abs/2308.12950.

Rico Sennrich, Barry Haddow, and Alexandra Birch. 2016. Neural machine translation of rare words with subword units. In ACL (1). The Association for Computer Linguistics.

Yixuan Su, Tian Lan, Yan Wang, Dani Yogatama, Lingpeng Kong, and Nigel Collier. 2022. A contrastive framework for neural text generation. In NeurIPS.

Zeyu Sun, Qihao Zhu, Lili Mou, Yingfei Xiong, Ge Li, and Lu Zhang. 2019. A grammar-based structural CNN decoder for code generation. In AAAI, pages 7055–7062. AAAI Press.

Zeyu Sun, Qihao Zhu, Yingfei Xiong, Yican Sun, Lili Mou, and Lu Zhang. 2020. Treegen: A tree-based transformer architecture for code generation. In AAAI, pages 8984–8991. AAAI Press.

Yizhong Wang, Yeganeh Kordi, Swaroop Mishra, Alisa Liu, Noah A. Smith, Daniel Khashabi, and Hannaneh Hajishirzi. 2023. Self-instruct: Aligning language models with self-generated instructions. In ACL (1), pages 13484–13508. Association for Computational Linguistics.

Yue Wang, Weishi Wang, Shafiq R. Joty, and teven C. H. Hoi. 2021. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In EMNLP (1), pages 8696–8708.

Bolin Wei, Ge Li, Xin Xia, Zhiyi Fu, and Zhi Jin. 2019. Code generation as a dual task of code summarization. In NeurIPS, pages 6559–6569.

Jin Xu, Xiaojiang Liu, Jianhao Yan, Deng Cai, Huayang Li, and Jian Li. 2022. Learning to break the loop: Analyzing and mitigating repetitions for neural text generation. In NeurIPS.

Pengcheng Yin and Graham Neubig. 2018. TRANX: A transition-based neural abstract syntax parser for semantic parsing and code generation. In EMNLP (Demonstration), pages 7–12. Association for Computational Linguistics.

Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Zihan Wang, Lei Shen, Andi Wang, Yang Li, Teng Su, Zhilin Yang, and Jie Tang. 2023. Codegeex: A pre-trained model for code generation with multilingual evaluations on humaneval-x. CoRR, abs/2303.17568.

A Case Study

Input

```
def hex_key(num):
    """You have been tasked to write a function that receives
    a hexadecimal number as a string and counts the number of hexadecimal digits that are primes (prime
    number, or a prime, is a natural number greater than 1 that is not a product of two smaller natural
    numbers). Hexadecimal digits are 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F. Prime numbers are 2,
    3, 5, 7, 11, 13, 17,... So you have to determine a number of the following digits: 2, 3, 5, 7,
    B (=decimal 11), D (=decimal 13).
    Note: you may assume the input is always correct or empty string, and symbols A,B,C,D,E,F are always
    uppercase.
    Examples:
    For num = "AB" the output should be 1.
    For num = "1077E" the output should be 2.
    For num = "ABED1A33" the output should be 4.
    For num = "123456789ABCDEF0" the output should be 6.
    For num = "2020" the output should be 2.
    """
```

Ours

```
num = [0] + [(ord(i)) for i in num]
num = [i - 55 if i > 64 else i - 48 for i in num]
num = [i for i in num if i != 0]
count = 0
for i in num:
    if i > 1:
        for j in range(2, i):
            if i % j == 0:
                break
            else:
                count += 1
return count
```

✔

Repetition

```
num = [0] + [(ord(i)) for i in num]
num = [i - 55 if i > 64 else i - 48 for i in num]
num = [i for i in num if i != 0]
num = [i for i in num if i != 10]
num = [i for i in num if i != 13]
num = [i for i in num if i != 16]
num = [i for i in num if i != 17]
num = [i for i in num if i != 18]
num = [i for i in num if i != 19]
num = [i for i in num if i != 20]
num = [i for i in num if i != 21]
num = [i for i in num if i != 22]
num = [i for i in num if i != 23]
num = [i for i in num if i != 24]
num = [i for i in num if i != 25]
num = [i for i in num if i != 26]
num = [i for i in num if i != 27]
num = [i for i in num if i != 28]
num = [i ...
```

✘

Unable to escape from repetition !

(a) A case from HumanEval in Code Generation Benchmark.

Input

```
filename>lib/variables/latent_variables/__init__.py
from .fully_connected import FullyConnectedLatentVariable
from .convolutional import ConvolutionalLatentVariable
from .recurrent import RecurrentLatentVariable
```

Ours

```
from .base import LatentVariable

__all__ = [
'LatentVariable',
'FullyConnectedLatentVariable',
'ConvolutionalLatentVariable',
'RecurrentLatentVariable',
]
```

✔

Repetition

```
from .lstm import LSTMLatentVariable
from .gru import GRULatentVariable
from .attention import AttentionLatentVariable
from .attention_lstm import AttentionLSTMLatentVariable
from .attention_gru import AttentionGRULatentVariable
from .attention_conv import AttentionConvLatentVariable
from .attention_conv_lstm import AttentionConvLSTMLatentVariable
from .attention_conv_gru import AttentionConvGRULatentVariable
from .attention_conv_attention_lstm import AttentionConvAttentionLSTMLatentVariable
from .attention_conv_attention_gru import AttentionConvAttentionGRULatentVariable
from .attention_conv_attention_conv import AttentionConvAttentionConvLatentVariable
from .attention_conv_attention_conv_lstm import AttentionConvAttentionConvLSTMLatentVariable
from .attention_conv_attention_conv_gru import AttentionConvAttentionConvGRULatentVariable
from .attention_conv_attention_conv_attention_lstm import
AttentionConvAttentionConvAttentionLSTMLatentVariable
from .attention_conv_attention_conv_attention_gru import
AttentionConvAttentionConvAttentionGRULatentVariable
from .attention_conv_attention_conv_attention_conv import
AttentionConvAttentionConvAttentionConvLatentVariable
from .attention_conv_attention_conv_attention_conv_lstm import
AttentionConvAttentionConvAttentionConvLSTMLatentVariable
from .attention_conv_attention_conv_attention_conv_gru import
AttentionConvAttentionConvAttentionConvGRULatentVariable
from .attention_conv_attention_conv_attention_conv_attention_lstm import
AttentionConvAttentionConvAttentionConvAttentionLSTMLatentVariable
from .attention_conv_attention_conv_attention_conv_attention_gru import
AttentionConvAttentionConvAttentionConvAttentionGRULatentVariable
from .attention_conv_attention_conv_attention_conv_attention_conv import
AttentionConvAttentionConvAttentionConvAttentionConvLatentVariable
...
```

✘

Unable to escape from repetition !

(b) A case from Real-world Repository

Figure 4: Two cases of generating structural repetition and the effect of our approach on them. LLMs succumb to endless loops of repetition. Our proposed approach can effectively break out of these loops, steering back to a normal code generation trajectory, and ultimately succeeding in producing correct code.

B The Influence of Hyper-parameters λ

We shown the influence of hyper-parameters λ in Figure 5.

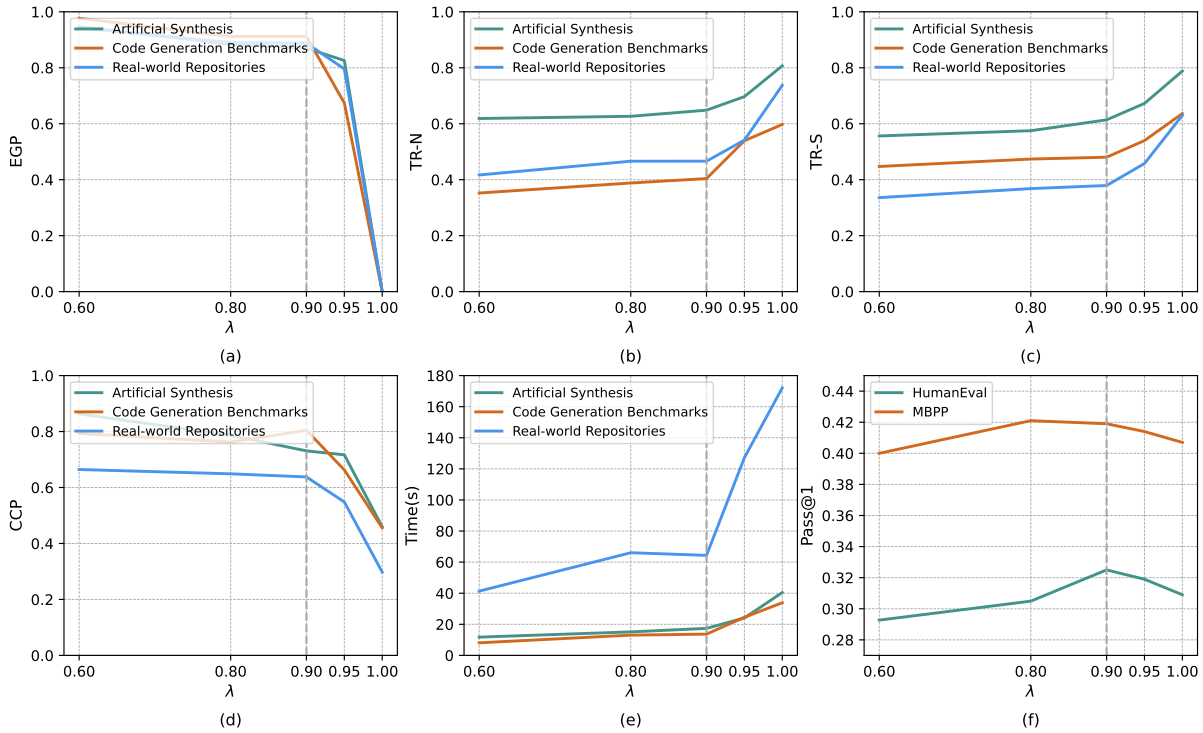

Figure 5: The influence of hyper-parameters λ on Artificial Synthesis, Code Generation Benchmarks, and Real-world Repositories scenarios of CodeRepetEval dataset, as well as HumanEval and MBPP benchmarks. We use the gray dashed line to represent the employed hyper-parameters.

C Details of Experiment Setup

C.1 Dataset Construction and Evaluation Methodology of CodeRepetEval

The purpose of CodeRepetEval datasets is to evaluate the performance of different approaches for avoiding repetitions in code generation under various scenarios. We provide a detailed description of the dataset construction process and the evaluation methodology as follows.

The construction processes for three scenarios, i.e., Artificial Synthesis, Code Generation Benchmarks, and Real-world Repositories, are similar, with mere differences in the source of code samples. For Artificial Synthesis, we employ Self-instruct (Wang et al., 2023) to construct instruction sets that induce CodeLlama-7B to generate code samples containing repetitions. For Code Generation Benchmarks, we use CodeLlama-7B to sample codes on HumanEval and MBPP benchmarks. For Real-world Repositories, we select code from 100 high-quality open-source projects on GitHub, following the selection criteria of Starcoder (Li et al., 2023b).

After obtaining the code samples, we employ the TR-S metric to sort them in descending order based on the degree of repetition. We then select the repetitive code segments from the top 2*N samples, where N equals 512, 512, and 1024 for the respective scenarios. Subsequently, we randomly truncate the code samples at positions containing 2 to 4 repeated statements, retain the first half, and combine it with the original input to generate new prompts. Finally, the researchers conduct a manual evaluation to filter out N prompts, which are then incorporated into our datasets.

For the evaluation, we employ the constructed prompts as input and generate code using different approaches. We apply three metrics—TR-S, TR-N, and EGP—to determine the extent to which the code generated by these approaches continues to manifest repetition issues. Moreover, we apply the CCP metric to assess the compilability and quality of generated codes.

C.2 Code Generation Benchmarks

We evaluate our approach using four public code generation benchmarks.

HumanEval (Chen et al., 2021) consists of 164 handwritten programming tasks, proposed by OpenAI. Each task includes a function signature, NL description, use cases, function body, and several unit tests (average 7.7 per task).

MBPP (Austin et al., 2021) contains 974 manually verified Python programming tasks, covering programming fundamentals, standard library functionality, and more. Each task consists of an NL description, a code solution, and 3 automated test cases.

HumanEval-ET and **MBPP-ET** (Dong et al., 2024) are expanded versions of MBPP and HumanEval respectively, where each includes over 100 additional test cases per task. This updated version enhances the soundness of code evaluation compared to the original benchmarks.

C.3 Baselines

We detail the baseline approaches compared in this work:

- **Greedy Sampling** chooses the highest probability token from the model's output distribution at each time step, $P'_{\text{greedy}}(w) = \mathbb{1}(w = \arg\max_w P(w|w_{<t}))$.

- **Topk Sampling** (Fan et al., 2018) limits the next-word selection to the top k most likely candidates as determined by the model, $P'_{\text{topk}}(w) = P(w|w_{<t})$ if $w \in$ Topk, otherwise 0.

- **Topp Sampling** (Holtzman et al., 2020) involves choosing from a smaller set of plausible candidates by dynamically selecting a variable-sized subset of tokens (the "nucleus") that cumulatively make up a certain probability mass (e.g., top 90%), $P'_{\text{topp}}(w) = P(w|w_{<t})$ if $\sum_{w' \in S} P(w'|w_{<t}) \leq p$, otherwise 0.

- **Temperature Sampling** (Caccia et al., 2019) controls the randomness of the token selection process—higher temperatures lead to more uniform distributions, while lower temperatures make high-probability tokens even more likely, $P'_{\text{temp}}(w) = \frac{\exp(\log(P(w|w_{<t}))/T)}{\sum_{w'} \exp(\log(P(w'|w_{<t}))/T)}$.

- **Repetition Penalty** (Keskar et al., 2019) penalizes sampling works by discounting the scores of previously generated tokens, $P'(w) = \frac{\exp(\log(P(w|w_{<t}))/(T \cdot I(i \in g))}{\sum_{w'} \exp(\log(P(w'|w_{<t}))/(T \cdot I(i \in g))}$, where $I(c) = \theta$ if c is True else 1. Unless otherwise specified, the settings of baselines follow their original paper.

- **Repetition Dropout** (Li et al., 2023a) applies masking vectors to sentences, randomly dropping out repetitive n-grams based on a pre-specified dropout rate, thereby preventing the model from over-relying on repetitive patterns during training.

C.4 Metrics

Compiler Correctness Percentage (CCP). CCP is defined as the ratio of the number of code samples that pass compilation to the total number of samples in the dataset.

Pass@k. We use the unbiased version (Chen et al., 2021) of Pass@k, where $n >= k$ samples are generated for each problem, count the number of correct samples $c <= n$ which pass test cases and calculate the following estimator, i.e.,

$$\text{Pass@k} = \mathop{\mathbb{E}}_{\text{Problems}} \left[1 - \frac{\binom{n-c}{k}}{\binom{n}{k}} \right]. \tag{9}$$

D Pseudo-code of Suffix Array and LCP Array

```
def constructSuffixArray(S):
    n = length of S
    Create an array suffixes[n] where each element is a tuple (index, suffix)
    for i from 0 to n-1:
        suffixes[i] = (i, S[i:n]) // Store index and suffix starting at index
            i
    Sort suffixes based on the suffix part of each tuple
    Initialize SA[n]
    for i from 0 to n-1:
        SA[i] = suffixes[i].index
    return SA

def constructLCPArray(S, SA):
    n = length of S
    Initialize LCP[n] with zeros
    Initialize rank[n] to store the rank of each suffix in SA
    for i from 0 to n-1:
        rank[SA[i]] = i
    h = 0 // Length of the longest common prefix
    for i from 0 to n-1:
        if rank[i] > 0:
            j = SA[rank[i] - 1] // Index of the previous suffix in the sorted
                list
            while i + h < n and j + h < n and S[i + h] == S[j + h]:
                h += 1
            LCP[rank[i]] = h
            if h > 0:
                h -= 1 // Decrease h for the next calculation
    return LCP

def findConsecutiveRepetitions(S):
    SA = constructSuffixArray(S)
    LCP = constructLCPArray(S, SA)
    repetitions = set()
    for i from 1 to length of S - 1:
        if LCP[i] > 0:
            duplicate_substring = S[SA[i]:SA[i] + LCP[i]]
            # Check for consecutive occurrence
            previous_suffix_length = SA[i-1]
            current_suffix_length = SA[i]
            if previous_start == current_start + LCP[i]:
                repetitions.add(duplicate_substring)
    return repetitions
```

E PDA for LLMs

LLMs usually employ Byte-Pair Encoding (BPE) (Sennrich et al., 2016) for tokenization, which causes the tokens in the vocabulary of LLMs to deviate from the terminal symbols in grammar. Specifically, this discrepancy manifests in three primary scenarios, i.e., one token corresponds to one terminal symbol, one

token corresponds to multiple terminal symbols, and multiple tokens correspond to one terminal symbol. Therefore, to effectively utilize PDA with LLMs, it is essential to develop an approach that adapts PDA operations to accommodate these tokenization scenarios.

E.1 One Token to One Terminal Symbol

In scenarios where one token directly corresponds to one terminal symbol, the adaptation of PDA is relatively straightforward. The PDA can process each token as a single unit that matches exactly one terminal symbol in the grammar of the language being parsed. Here, the transition functions of PDA can be directly applied without modification. For instance, if a token from the LLM's output matches a terminal symbol in a programming language's grammar, the PDA can push, pop, or transition based on this token following standard PDA rules. This case represents the simplest form of interaction between LLM outputs and grammar-based parsing.

E.2 One Token to Multiple Terminal Symbols

This scenario arises when a single token encapsulates multiple grammatical elements, due to a compact or compressed representation of the language. For example, "[]", ")," ")\n", and so on. To handle this, we should first decompose the token into its constituent terminal symbols. Then, we sequentially check whether each terminal symbol is in the PDA candidate set of its prefixes. Finally, the tokens that all constitute terminal symbols satisfying the condition are retained. This approach ensures that even complex tokens can be seamlessly integrated into the grammar-based processing framework of the PDA.

E.3 Multiple Tokens to One Terminal Symbol

In contrast, when multiple tokens collectively represent a single terminal symbol, we should aggregate these tokens before using the transition function of PDA. This scenario typically occurs with the token-types in terminal symbols, such as NAME, NUMBER, and STRING. In this case, we constructed a lexical grammar PDA to accumulate tokens until a complete terminal symbol is formed. The PDA operations then proceed based on these aggregated terminal symbols.

F Full Grammar specification

For example, the full Python grammar is shown as follows:

```
# Grammar for Python

# NOTE WELL: You should also follow all the steps listed at
# https://devguide.python.org/grammar/

# Start symbols for the grammar:
#       single_input is a single interactive statement;
#       file_input is a module or sequence of commands read from an input file
    ;
#       eval_input is the input for the eval() functions.
#       func_type_input is a PEP 484 Python 2 function type comment
# NB: compound_stmt in single_input is followed by extra NEWLINE!
# NB: due to the way TYPE_COMMENT is tokenized it will always be followed by
    a NEWLINE
single_input: NEWLINE | simple_stmt | compound_stmt NEWLINE
file_input: (NEWLINE | stmt)* ENDMARKER
eval_input: testlist NEWLINE* ENDMARKER

decorator: '@' dotted_name [ '(' [arglist] ')' ] NEWLINE
```

```
decorators: decorator+
decorated: decorators (classdef | funcdef | async_funcdef)

async_funcdef: ASYNC funcdef
funcdef: 'def' NAME parameters ['->' test] ':' [TYPE_COMMENT] func_body_suite

parameters: '(' [typedargslist] ')'

# The following definition for typedarglist is equivalent to this set of
    rules:
#
#    arguments = argument (',' [TYPE_COMMENT] argument)*
#    argument = tfpdef ['=' test]
#    kwargs = '**' tfpdef [','] [TYPE_COMMENT]
#    args = '*' [tfpdef]
#    kwonly_kwargs = (',' [TYPE_COMMENT] argument)* (TYPE_COMMENT | [',' [
    TYPE_COMMENT] [kwargs]])
#    args_kwonly_kwargs = args kwonly_kwargs | kwargs
#    poskeyword_args_kwonly_kwargs = arguments ( TYPE_COMMENT | [',' [
    TYPE_COMMENT] [args_kwonly_kwargs]])
#    typedargslist_no_posonly = poskeyword_args_kwonly_kwargs |
    args_kwonly_kwargs
#    typedarglist = (arguments ',' [TYPE_COMMENT] '/' [',' [[TYPE_COMMENT]
    typedargslist_no_posonly]])|(typedargslist_no_posonly)"
#
# It needs to be fully expanded to allow our LL(1) parser to work on it.

typedargslist: (
  (tfpdef ['=' test] (',' [TYPE_COMMENT] tfpdef ['=' test])* ',' [
      TYPE_COMMENT] '/' [',' [ [TYPE_COMMENT] tfpdef ['=' test] (
        ',' [TYPE_COMMENT] tfpdef ['=' test])* (TYPE_COMMENT | [',' [
            TYPE_COMMENT] [
        '*' [tfpdef] (',' [TYPE_COMMENT] tfpdef ['=' test])* (TYPE_COMMENT |
            [',' [TYPE_COMMENT] ['**' tfpdef [','] [TYPE_COMMENT]]])
      | '**' tfpdef [','] [TYPE_COMMENT]]])
  | '*' [tfpdef] (',' [TYPE_COMMENT] tfpdef ['=' test])* (TYPE_COMMENT | [','
       [TYPE_COMMENT] ['**' tfpdef [','] [TYPE_COMMENT]]])
  | '**' tfpdef [','] [TYPE_COMMENT]]] )
|  (tfpdef ['=' test] (',' [TYPE_COMMENT] tfpdef ['=' test])* (TYPE_COMMENT |
     [',' [TYPE_COMMENT] [
   '*' [tfpdef] (',' [TYPE_COMMENT] tfpdef ['=' test])* (TYPE_COMMENT | [','
       [TYPE_COMMENT] ['**' tfpdef [','] [TYPE_COMMENT]]])
  | '**' tfpdef [','] [TYPE_COMMENT]]])
  | '*' [tfpdef] (',' [TYPE_COMMENT] tfpdef ['=' test])* (TYPE_COMMENT | [','
       [TYPE_COMMENT] ['**' tfpdef [','] [TYPE_COMMENT]]])
  | '**' tfpdef [','] [TYPE_COMMENT])
)
tfpdef: NAME [':' test]
```

```
# The following definition for varargslist is equivalent to this set of rules:

#
#     arguments = argument (',' argument )*
#     argument = vfpdef ['=' test]
#     kwargs = '**' vfpdef [',']
#     args = '*' [vfpdef]
#     kwonly_kwargs = (',' argument )* [',' [kwargs]]
#     args_kwonly_kwargs = args kwonly_kwargs | kwargs
#     poskeyword_args_kwonly_kwargs = arguments [',' [args_kwonly_kwargs]]
#     vararglist_no_posonly = poskeyword_args_kwonly_kwargs |
    args_kwonly_kwargs
#     varargslist = arguments ',' '/' [','[(vararglist_no_posonly)]] | (
    vararglist_no_posonly)
#
# It needs to be fully expanded to allow our LL(1) parser to work on it.

varargslist: vfpdef ['=' test ](',' vfpdef ['=' test])* ',' '/' [',' [ (
    vfpdef ['=' test] (',' vfpdef ['=' test])* [',' [
        '*' [vfpdef] (',' vfpdef ['=' test])* [',' ['**' vfpdef [',']]]
      | '**' vfpdef [',']]]
  | '*' [vfpdef] (',' vfpdef ['=' test])* [',' ['**' vfpdef [',']]]
  | '**' vfpdef [',']) ]] | (vfpdef ['=' test] (',' vfpdef ['=' test])* [','
      [
        '*' [vfpdef] (',' vfpdef ['=' test])* [',' ['**' vfpdef [',']]]
      | '**' vfpdef [',']]]
  | '*' [vfpdef] (',' vfpdef ['=' test])* [',' ['**' vfpdef [',']]]
  | '**' vfpdef [',']
)
vfpdef: NAME

stmt: simple_stmt | compound_stmt
simple_stmt: small_stmt (';' small_stmt)* [';'] NEWLINE
small_stmt: (expr_stmt | del_stmt | pass_stmt | flow_stmt |
            import_stmt | global_stmt | nonlocal_stmt | assert_stmt)
expr_stmt: testlist_star_expr (annassign | augassign (yield_expr|testlist) |
                   [('=' (yield_expr|testlist_star_expr))+ [TYPE_COMMENT]] )
annassign: ':' test ['=' (yield_expr|testlist_star_expr)]
testlist_star_expr: (test|star_expr) (',' (test|star_expr))* [',']
augassign: ('+=' | '-=' | '*=' | '@=' | '/=' | '%=' | '&=' | '|=' | '^=' |
           '<<=' | '>>=' | '**=' | '//=')
# For normal and annotated assignments, additional restrictions enforced by
    the interpreter
del_stmt: 'del' exprlist
pass_stmt: 'pass'
flow_stmt: break_stmt | continue_stmt | return_stmt | raise_stmt | yield_stmt
break_stmt: 'break'
continue_stmt: 'continue'
return_stmt: 'return' [testlist_star_expr]
yield_stmt: yield_expr
```

```
raise_stmt: 'raise' [test ['from' test]]
import_stmt: import_name | import_from
import_name: 'import' dotted_as_names
# note below: the ('.' | '...') is necessary because '...' is tokenized as
    ELLIPSIS
import_from: ('from' (('.' | '...')* dotted_name | ('.' | '...')+)
             'import' ('*' | '(' import_as_names ')' | import_as_names))
import_as_name: NAME ['as' NAME]
dotted_as_name: dotted_name ['as' NAME]
import_as_names: import_as_name (',' import_as_name)* [',']
dotted_as_names: dotted_as_name (',' dotted_as_name)*
dotted_name: NAME ('.' NAME)*
global_stmt: 'global' NAME (',' NAME)*
nonlocal_stmt: 'nonlocal' NAME (',' NAME)*
assert_stmt: 'assert' test [',' test]

compound_stmt: if_stmt | while_stmt | for_stmt | try_stmt | with_stmt |
    funcdef | classdef | decorated | async_stmt
async_stmt: ASYNC (funcdef | with_stmt | for_stmt)
if_stmt: 'if' namedexpr_test ':' suite ('elif' namedexpr_test ':' suite)* ['
    else' ':' suite]
while_stmt: 'while' namedexpr_test ':' suite ['else' ':' suite]
for_stmt: 'for' exprlist 'in' testlist ':' [TYPE_COMMENT] suite ['else' ':'
    suite]
try_stmt: ('try' ':' suite
          ((except_clause ':' suite)+
           ['else' ':' suite]
           ['finally' ':' suite] |
          'finally' ':' suite))
with_stmt: 'with' with_item (',' with_item)* ':' [TYPE_COMMENT] suite
with_item: test ['as' expr]
# NB compile.c makes sure that the default except clause is last
except_clause: 'except' [test ['as' NAME]]
suite: simple_stmt | NEWLINE INDENT stmt+ DEDENT

namedexpr_test: test [':=' test]
test: or_test ['if' or_test 'else' test] | lambdef
test_nocond: or_test | lambdef_nocond
lambdef: 'lambda' [varargslist] ':' test
lambdef_nocond: 'lambda' [varargslist] ':' test_nocond
or_test: and_test ('or' and_test)*
and_test: not_test ('and' not_test)*
not_test: 'not' not_test | comparison
comparison: expr (comp_op expr)*
# <> isn't actually a valid comparison operator in Python. It's here for the
# sake of a __future__ import described in PEP 401 (which really works :-)
comp_op: '<'|'>'|'=='|'>='|'<='|'<>'|'!='|'in'|'not' 'in'|'is'|'is' 'not'
star_expr: '*' expr
expr: xor_expr ('|' xor_expr)*
xor_expr: and_expr ('^' and_expr)*
```

```
and_expr: shift_expr ('&' shift_expr)*
shift_expr: arith_expr (('<<'|'>>') arith_expr)*
arith_expr: term (('+'|'-') term)*
term: factor (('*'|'@'|'/'|'%'|'//') factor)*
factor: ('+'|'-'|'~') factor | power
power: atom_expr ['**' factor]
atom_expr: [AWAIT] atom trailer*
atom: ('(' [yield_expr|testlist_comp] ')' |
      '[' [testlist_comp] ']' |
      '{' [dictorsetmaker] '}' |
      NAME | NUMBER | STRING+ | '...' | 'None' | 'True' | 'False')
testlist_comp: (namedexpr_test|star_expr) ( comp_for | (',' (namedexpr_test|
    star_expr))* [','] )
trailer: '(' [arglist] ')' | '[' subscriptlist ']' | '.' NAME
subscriptlist: subscript (',' subscript)* [',']
subscript: test | [test] ':' [test] [sliceop]
sliceop: ':' [test]
exprlist: (expr|star_expr) (',' (expr|star_expr))* [',']
testlist: test (',' test)* [',']
dictorsetmaker: ( ((test ':' test | '**' expr)
                 (comp_for | (',' (test ':' test | '**' expr))* [','])) |
                ((test | star_expr)
                 (comp_for | (',' (test | star_expr))* [','])) )

classdef: 'class' NAME ['(' [arglist] ')'] ':' suite

arglist: argument (',' argument)* [',']

# The reason that keywords are test nodes instead of NAME is that using NAME
# results in an ambiguity. ast.c makes sure it's a NAME.
# "test '=' test" is really "keyword '=' test", but we have no such token.
# These need to be in a single rule to avoid grammar that is ambiguous
# to our LL(1) parser. Even though 'test' includes '*expr' in star_expr,
# we explicitly match '*' here, too, to give it proper precedence.
# Illegal combinations and orderings are blocked in ast.c:
# multiple (test comp_for) arguments are blocked; keyword unpackings
# that precede iterable unpackings are blocked; etc.
argument: ( test [comp_for] |
           test ':=' test |
           test '=' test |
           '**' test |
           '*' test )

comp_iter: comp_for | comp_if
sync_comp_for: 'for' exprlist 'in' or_test [comp_iter]
comp_for: [ASYNC] sync_comp_for
comp_if: 'if' test_nocond [comp_iter]

# not used in grammar, but may appear in "node" passed from Parser to
    Compiler
```

```
encoding_decl: NAME

yield_expr: 'yield' [yield_arg]
yield_arg: 'from' test | testlist_star_expr

# the TYPE_COMMENT in suites is only parsed for funcdefs,
# but can't go elsewhere due to ambiguity
func_body_suite: simple_stmt | NEWLINE [TYPE_COMMENT NEWLINE] INDENT stmt+
    DEDENT

func_type_input: func_type NEWLINE* ENDMARKER
func_type: '(' [typelist] ')' '->' test
# typelist is a modified typedargslist (see above)
typelist: (test (',' test)* [','
       ['*' [test] (',' test)* [',' '**' test] | '**' test]]
     |  '*' [test] (',' test)* [',' '**' test] | '**' test)
```