SPE Attention: Making Attention Equivariant to Semantic-Preserving Permutation for Code Processing

Chengyu Jiao[1], **Shuhao Chen**[1,2], **Yu Zhang**[1]*
[1]Southern University of Science and Technology
[2]The Hong Kong University of Science and Technology

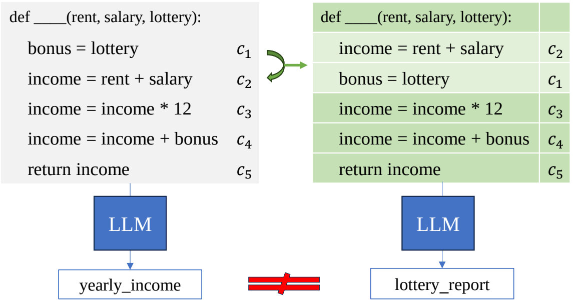
Abstract

Codes serve as the fundamental language for human to communicate with machines, and various Transformer-based models are trained to process codes in recent advancements. A unique symmetry of code is its semantic-preserving permutation, which allows certain lines to be rearranged without altering the overall meaning. To capture such symmetry, we propose a novel attention mechanism that incorporates semantic-preserving permutation equivariance, called the SPE attention. By leveraging the symmetry relationships within code, we introduce a directed layered graph to represent the code structure, and this graph is then summarized into a symmetry mask. The SPE attention integrates those symmetry masks, granting semantic-preserving permutations equivariance to the model. Experiments on various code related tasks, including code summarization and error detection, demonstrate the effectiveness of the proposed SPE attention.

1 Introduction

The attention mechanism (Vaswani et al., 2017) have demonstrated significant success in many transformer-based models (Yenduri et al., 2023; Touvron et al., 2023) on processing natural languages. As a specialized form of language, code is frequently analyzed by these models in various research studies. Many large language models (LLMs) have fine-tuned versions specifically designed for code related tasks, such as Code-Llama (Rozière et al., 2024) from the Llama family (Touvron et al., 2023) and Codex (Chen et al., 2021) from OpenAI's GPT family (Yenduri et al., 2023). Those models have numerous applications, including automated code generation, debugging, and code optimization. Such tools can drastically improve the efficiency of programming.

From the perspective of geometric deep learning, there is one key difference between a paragraph of natural language and a code block: Certain lines of code can be permuted without altering their semantics, forming a unique symmetry as the *semantic-preserving permutation*, which generally do not exist in natural languages. Therefore, it is natural to make use of such permutation information into model designs to make models equivariant to semantic-preserving permutation, where the *equivariance* refers to the model capacity to incorporate and preserve certain symmetries.



```
def ____(rent, salary, lottery):
    bonus = lottery                  c1
    income = rent + salary           c2
    income = income * 12             c3
    income = income + bonus          c4
    return income                    c5
```

```
def ____(rent, salary, lottery):
    income = rent + salary           c2
    bonus = lottery                  c1
    income = income * 12             c3
    income = income + bonus          c4
    return income                    c5
```

Figure 1: For two function that have same semantics, the predicted function names by the CodeLlama are different due to the lack of the permutation equivariance.

However, most attention-based models, including LLMs, fail to possess such semantic-persevering permutation equivariance. To see that, as illustrated in Figure 1. we have a code block C that computes the yearly income. It is easy to see that swapping the first two lines in C does not change the semantics. However, the predicted function names by the CodeLlama for the two code snippets are different, potentially leading to errors in some cases.

The lack of equivariance in current attention-based models is largely due to how positional encoding is integrated (Vaswani et al., 2017). When two embeddings are swapped, the corresponding positional encodings remain unchanged. Therefore, the sum of the embeddings and positional encodings cannot be the equivariant.

*Corresponding author (yu.zhang.ust@gmail.com)

Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing, pages 6555–6568
November 4-9, 2025 ©2025 Association for Computational Linguistics

However, directly eliminating positional encoding is not a viable solution, as it would render the attention mechanism equivariant to all the permutations, including those that alter the underlying semantics of codes. Consequently, achieving the equivariance within the existing network architectures presents a significant challenge.

To address this challenge, in this paper, we propose a Semantic-preserving Permutation Equivariant (SPE) attention to empower attention layers with the equivariance to semantic-preserving permutations, while maintaining positional information. In the proposed SPE attention, we first introduce a novel representation of code: the *directed layered graph*. This representation is specifically crafted to capture semantic-preserving permutations, enabling us to easily distinguish between permutations that preserve symmetries and those that do not. Additionally, such directed layered graph are more efficient to compute comparing to previous representation (Pei et al., 2023). We further summarize the directed layered graph of a code block into a symmetry mask. The symmetry mask has an important property that it is equivariant only to the semantic-preserving permutations. Then the proposed SPE attention combines such masks with the attention score and we prove that the SPE attention is indeed equivariant to semantic-preserving permutations.

To employ such SPE Attention, the model takes the tokenized text of the code along with this precalculated symmetry mask. After embedding layer and resetting the positional encoding at the beginning of each line, we utilize the SPE attention to process the embedded text of the code, granting semantic-preserving permutation equivariance to output features. We repeat the process for all SPE attention layers. After the last SPE attention layer, pooling is performed, and the final output is invariant for classification.

To summarize, our contributions are three-fold.

1. We introduce a novel SPE attention mechanism by integrating attention scores with a symmetry mask that captures the symmetry of code blocks.

2. We derive such symmetry mask from a novel representation of code, directed layered graph. This representation is closely tied to the semantically preserving permutations, and the symmetry mask is equivariant to such permutations.

3. Empirical evaluations on several code related tasks, including code summarization and error detection, demonstrate that the proposed SPE attention could help improve the model performance.

2 Related Works

2.1 Equivariant Neural Networks

When known symmetries are present in the data, incorporating equivariance into models often yields better performance improvements than relying solely on data augmentation (Gerken et al., 2022) Group Equivariant Convolutional Neural Networks (Cohen and Welling, 2016a) (G-CNNs) and follow-up methods (Cohen and Welling, 2016b; Cohen et al., 2018) are widely used for processing images that exhibit rotation and reflection symmetries, and showed siginificant improvements over non-equivariant convolutional networks. When processing graphs, Graph Neural Networks (GNN) can take advantages of node permutation equivariance (Satorras et al., 2021). Neural Functional (Zhou et al., 2023) aims to directly learn from the weights of neural networks.

2.2 Code Processing

There are some large language models (Rozière et al., 2024; Chen et al., 2021) specifically finetuned for code-related tasks. Other than large language models, several models are also designed specifically for these tasks, such as code2vec (Alon et al., 2018) and DOBF (Rozière et al., 2021). However, for the utilization of the equivariance for code processing, to the best of our knowledge, SYMC (Pei et al., 2023) is the only one related work to ours. The SYMC method gathers information from the dependency relationship of codes, and formulate a *permutation bias* term. Adding such permutation bias to the attention layer grants equivariance to the code. The SYMC method achieves better performance over non-equivariant models that have much larger model size. Despite sharing the same initiative, there are some key differences between the SYMC method and the our proposed SPE attention, including different representation of codes.

2.3 Representation of Codes

The representation of codes has gained significant attention in recent years, driven by the need to leverage structural information for various software analysis tasks. Early approaches primarily utilize abstract syntax trees (ASTs) (Song et al., 2024) to represent code structures, allowing the extraction of syntactic features and enabling tasks such as code similarity evaluation. Recent advancements have expanded the scope of graph-based representations to larger programs composed of several code blocks, by incorporating control flow graphs (CFGs) (Huang et al., 2023). This representation captures the dynamic relationships between code blocks within a program. In terms of equivariance, the program interpretation graph (PIGs) (Pei et al., 2023) is used in the SYMC method, where dependency pairs are computed by considering all possible permutation. As discussed above, such representation is very inefficient. The SYMC method avoids this issue by relaxing the definition and using the program dependency graph (Ferrante et al., 1987) instead. Different from those works, in this paper we propose a more efficient directed layered graph, strongly related to semantic-preserving permutations.

3 Preliminary

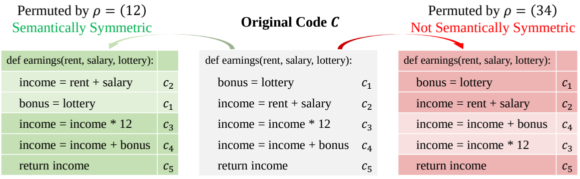
3.1 Permutation Group

In this section, we provide a brief definition of the permutation groups. A more detailed and formal explanation can be found in Appendix A.2.

Abstractly, groups capture mathematical symmetries: Each element of a group can transform an object in a structured way. In this work, we focus on the *permutation group* S_n, which consists of all possible ways to reorder n elements.

To define S_n, we first introduce the concept of cycle notation. A cycle c is represented as $c = (a_1\ a_2\ \ldots\ a_k)$, which maps each element to the next element in a circular fashion. Specifically, for i such that $1 \leq i \leq k-1$, we have $c(a_i) = a_{i+1}$, and for the last element, $c(a_k) = a_1$. A permutation $\rho \in S_n$ can be uniquely represented in disjoint cycle notations, and acts on a sequence of code lines by rearranging their positions.

For example, consider three code lines C = (`a=1', `b=2', `b=a+3'). The first example permutation $\rho = (132)$ acts on $C$ by sending position 1 to 3, 3 to 2, and 2 to 1. The result is $\rho C$ = (`b=2', `b=a+3', `a=1'). A second example is $\varphi = (13)(2)$. It acts on C by swapping the first and third lines while leaving the second line unchanged, resulting in φC = (`b=a+3', `b=2', `a=1'). The unmoved action, such as the (2) in φ can often be abbreviated, resulted in $\varphi = (13)$.


Permuted by $\rho = (12)$
Semantically Symmetric
Original Code C
Permuted by $\rho = (34)$
Not Semantically Symmetric

```
def earnings(rent, salary, lottery):
    income = rent + salary         c2
    bonus = lottery                c1
    income = income * 12           c3
    income = income + bonus        c4
    return income                  c5
```

```
def earnings(rent, salary, lottery):
    bonus = lottery                c1
    income = rent + salary         c2
    income = income * 12           c3
    income = income + bonus        c4
    return income                  c5
```

```
def earnings(rent, salary, lottery):
    bonus = lottery                c1
    income = rent + salary         c2
    income = income + bonus        c4
    income = income * 12           c3
    return income                  c5
```

Figure 2: The original function in the middle computes the yearly earnings. On the left, we apply $\rho_1 = (12)$, switching line 1 and line 2. On the right, we apply $\rho_2 = (34)$. The first permutation preserve sematic while the second one do not.

Unlike many equivariant models (Cohen and Welling, 2016a,b), general permutation equivariance is undesired here. For example, if an error detection model M is equivariant to all permutations, M would make the same prediction for C and $\rho(C)$, destined to make mistake if ρ changes the semantic of C. Therefore, we only need the equivariance to "nice" permutations, which do not alter semantics of codes. In the next section, we formulate our interested subset of permutations, namely semantic-preserving permutations.

3.2 Semantic-Preserving Permutation of Codes

Given a code block C with n lines, we denote the code in the i-th line by c_i, and C can be viewed as a ordered set $C = \{c_i\}_{i=1}^n$. Permutation on C can be viewed as acting on the indices of lines, i.e., $\rho C = \{c_{\rho i}\}_{i=1}^n$.

Due to the nature of codes, certain lines can be permuted without changing the semantic of this code block C. The main criterion for identifying semantics is the dependency relationship among all lines, easily accessed using various parsing tools, such as `JavaLang`[1] for Java codes and the `ast` package[2] for Python codes. If line c_m depends on line c_n, we denote the dependence relationship by (n, m), and the collection of all such pairs by $C_{de} = \{(n, m) | c_m \text{ depends on } c_n\}$. The permutation ρ acts on C_{de} by permuting indices of each dependency pairs, i.e., $\rho C_{de} = \{(\rho(n), \rho(m)) | (n, m) \in C_{de}\}$.

It is crucial to distinguish between ρC_{de} and $(\rho C)_{de}$. The expression $(\rho C)_{de}$ means permuting the code first based on the permutation, then extracting the dependency pairs from the resulted code block ρC. It is possible that some depen-

[1] https://pypi.org/project/javalang/

[2] https://docs.python.org/3/library/ast

dency pairs are swapped, removed, or added after applying permutations to the code. Therefore, for a general $\rho \in S_n$, ρC_{de} and $(\rho C)_{de}$ are not always equal to each other.

Definition 3.1. Two blocks of codes C and D, both with n lines, are *semantically symmetric* if there is a unique permutation ρ such that $D = \rho C$ and $D_{de} = \rho C_{de}$. Such permutation ρ is called a *semantic-preserving permutation.*

In essence, two code blocks, C and D, are semantically symmetric if D is a permuted version of C and all the dependency pairs are preserved by such permutations. We provide an example in Figure 2. For the code C, permuting by $\rho_1 = (12)$ preserved all dependency pairs, thus it is a semantic-preserving permutation. On the other hand, the permutation $\rho_2 = (34)$ alters the semantics, as line c_4 depends on line c_3 before the permutation, but this dependency is reversed afterwards.

4 Methodology

In this section, we present the proposed SPE attention for achieving semantic-preserving permutation equivariance in attention layers. Without loss of generality, we consider a code block $C = \{c_i\}_{i=1}^n$ where each line consists of a single token for simplicity. In practice, all operations described below can be expanded to match the full token sequence length of each line.

4.1 Semantic-Preserving Permutation Equivariance

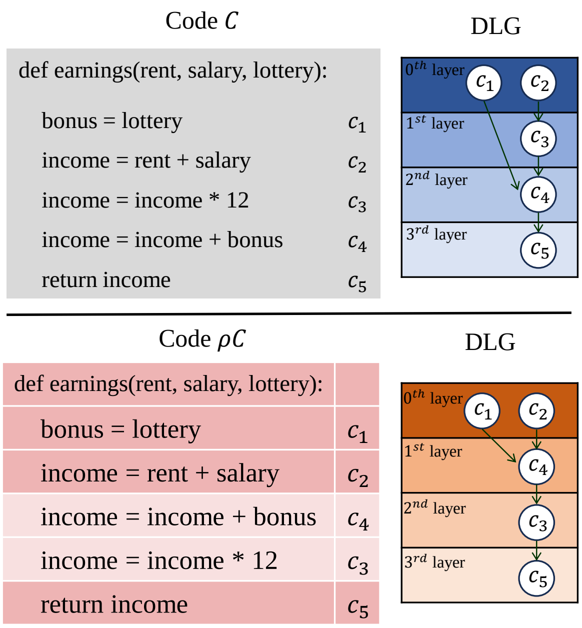
In this section, we aim to define the desired property of equivariance. As the only symmetry we are interested is permutation, we first define the general S_n-equivariance.

Definition 4.1. A neural network f is S_n-equivariant if

$$f(\rho x) = \rho f(x)$$

for all ρ in the permutation group S_n. For the special case of $f(\rho x) = f(x)$, we say that f is invariant.

It is well established that the standard self-attention mechanism is S_n-equivariant (Pei et al., 2023). Given a code block C, we denote by P_n the subset of S_n that preserves semantics. It is important that we only need the equivariance to the permutations that preserve semantics, which is defined as follows.



```
def earnings(rent, salary, lottery):
    bonus = lottery              c1
    income = rent + salary       c2
    income = income * 12         c3
    income = income + bonus      c4
    return income                c5
```

```
def earnings(rent, salary, lottery):
    bonus = lottery              c1
    income = rent + salary       c2
    income = income + bonus      c4
    income = income * 12         c3
    return income                c5
```

Figure 3: For two code blocks, we generate directed layered graph accordingly based on the dependency pairs. Comparing to the DLG on the top, the DLG on the bottom swaps c_3 and c_4.

Definition 4.2. A model f is semantic-preserving permutation equivariant or equivalently P_n-equivariant if the following two conditions hold.

1. For all $\rho \in P_n$, $f(\rho x) = \rho f(x)$.
2. For $\rho \in S_n/P_n$, $f(\rho x) \neq \rho f(x)$.

Similarly, a model f is P_n-invariant if the following two conditions are satisfied.

1. For all $\rho \in P_n$, $f(\rho x) = f(x)$.
2. For $\rho \in S_n/P_n$, $f(\rho x) \neq f(x)$.

We expect our model to be P_n-equivariant for reasons discussed in Sec. 3.2. Thus, we omit P_n and only say equivariant/invariant for the rest of the paper, unless we want to emphasize the difference between P_n and S_n.

To address code related tasks such as error detection and function name prediction, the model is designed to be invariant in their final output and equivariant before the last layer. This approach has proven to be effective in several equivariant neural networks (Cohen and Welling, 2016a; Zhou et al., 2023). As long as the intermediate features are equivariant, the invariance can be achieved by simply performing the average or max pooling on the equivariant features.

4.2 Directed Layered Graph

Given a code block C, there are currently many tools to extract the dependency pair between each line. Based on the collection of all dependency pairs, we can construct a *directed layered graph* (DLG) from them as follows.

1. For all the lines c_i that do not depend on any other lines, they are put in the the zeroth layer of the DLG. These are commonly referred to as the source nodes.

2. If c_j depends on c_i, there is a directed edge (i, j) from c_i to c_j in DLG.

3. For any other lines c_i, we compute the longest path from c_i to *any* source nodes, denoted by q. We put c_i in the q-th layer. We denote the layer of each c_i by $L(c_i)$.

For a permutation $\rho \in S_n$, ρ can also act on nodes and edges by permuting index. That is , $\rho c_i = c_{\rho(i)}$ and $\rho(i, j) = (\rho i, \rho j)$.

As shown in the following theorem,[3] the DLG has a strong correspondence to the permutation equivariance of C.

Theorem 4.3. *Given a code block C, ρ is a semantic-preserving permutation for C if and only if c_i and ρc_i are on the same layer for all $c_i \leq n$.*

Therefore, determining whether ρ is a permutation that preserves the symmetries of C is straightforward. One can simply examine the DLG of C and ρC to see if any of the nodes moves to a different layer. If such movement happens anywhere, then $\rho \notin P_n$.

According to the DLGs shown in Figure 3 that is based on Figure 2, we can see that these two blocks of codes are not equivariant by simply observing that c_4 and c_3 are moved to different layers. Among all the possible permutations, permutations that do not alter semantics are $P_n = \{(12), (132)\}$.

4.3 Symmetry Masks and SPE Attention

Given the DLG, we aim to design the SPE attention based on a proposed mask called *symmetry mask*, which is defined as follows.

Definition 4.4. Based on the DLG, the symmetry mask $M_C \in \mathbb{R}^{n \times n}$ of a code block C with n lines is defined as

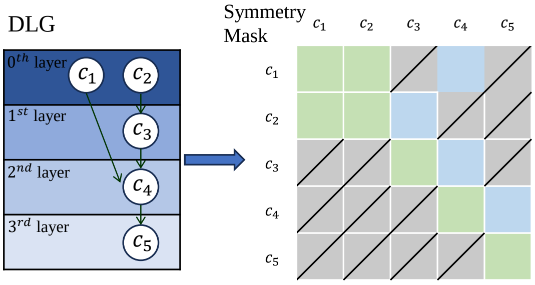


Figure 4: From the DLG, we generate corresponding symmetry mask. The crossed-out gray cell stands for zero. The green cells is kept based on the first principle in Def. 4.4, and the blue cells is kept based on the second principle in Def. 4.4.

1. If line i and j belong to the same layer in DLG, the (i, j)-th and (j, i)-th entries of M_C are set to 1.

2. If there is a directed arrow from i to j in DLG, the (i, j)-th entry of M_C is set to 1.

3. All other entries in M_C are set to zero.

The design of the symmetry mask is fairly straightforward to interpret. That is, we only keep the entries that are on the same layer, or have direct dependency relationships, while all the other entries are masked. An example of generating the symmetric mask is shown in Figure 4. For the symmetry mask, we have the following property.

Proposition 4.5. *The symmetry mask is equivariant to semantic-preserving permutations. That is, for any $\rho \in P_n$,*

$$\rho M_C = M_{\rho C},$$

where $M_{\rho C}$ is to perform ρ on the square matrix M_C by permuting rows and columns simultaneously.

Given a code block C with the symmetry mask M_C, the SPE attention is formulated as

$$\text{SPE-Att}(C) = \text{softmax}\left(\frac{(Q_C K_C^\top) \odot M_C}{\sqrt{d_k}}\right) V_C,$$

where Q_C, K_C, and V_C denote the query, key, and value of C, respectively, d_k is the dimension of K_C, and $\odot$ denotes the element-wise product or Hadamard product between two matrices.

For the SPE attention, we have the following properties.

[3] All proofs are presented in Appendix D.

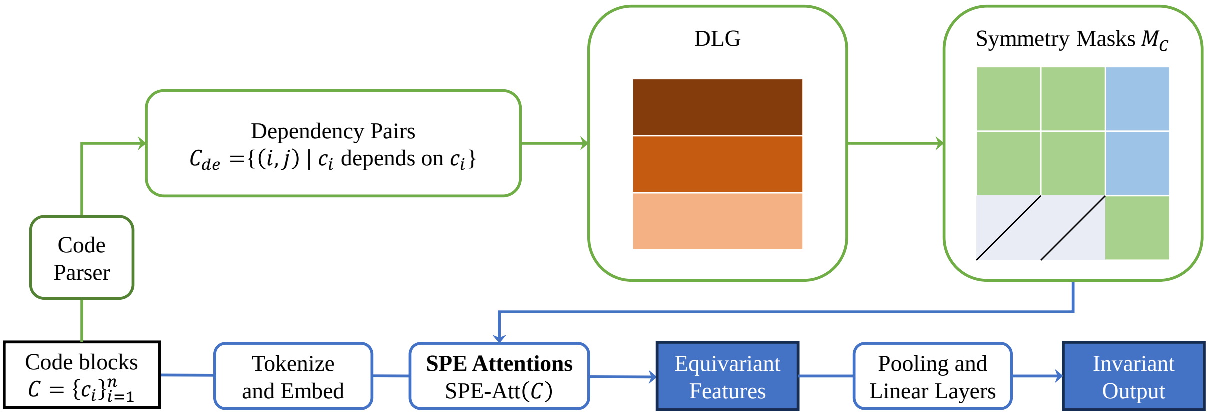

DLG
Symmetry Masks M_C
Dependency Pairs
$C_{de} = \{(i,j) \mid c_i \text{ depends on } c_j\}$
Code Parser
Code blocks $C = \{c_i\}_{i=1}^n$
Tokenize and Embed
SPE Attentions SPE-Att(C)
Equivariant Features
Pooling and Linear Layers
Invariant Output

Figure 5: An illustration of the proposed SPE attention.

Theorem 4.6. *Given a code block C and its corresponding masks M_C, we have the following properties.*

1. *SPE-Att(C) is equivariant to P_n. That is, for any $\rho \in P_n$, ρSPE-Att(C) = SPE-Att(ρC).*
2. *It is not equivariant to S_n. That is, for any $\rho \in S_n/P_n$, ρSPE-Att$(C) \neq$ SPE-Att(ρC).*

Thm. 4.6 shows that the proposed SPE attention is only equivariant to the semantic-preserving permutations. Note that applying the transpose of the symmetry mask also grants desired equivariance as in Thm. 4.6.

In the proof of Thm. 4.6, we use a fact from (Pei et al., 2023) that the conventional attention mechanism is permutation equivariant. However, positional encoding will break the equivariance property when added or multiplied to the embedding of C.[4] To mitigate this issue, we reset the positional encoding at the beginning of each line of the code (i.e., c_i). This approach maintains the positional information for each individual line. At the same time, it does not compromise the equivariance property between lines. This balance is crucial for effectively capturing both the structure and symmetry of the code, which enhances the overall performance. The complete framework of the proposed SPE attention network is illustrated in Figure 5.

The proposed SPE attention mechanism can be seamlessly integrated into a multi-head attention framework. For each head, we have the option to apply the symmetry mask, the transpose of the symmetry mask, or no masks at all. To ensure the equivariance to semantic-preserving permutations, it is sufficient for at least one head to apply either the symmetry mask or its transpose.

5 Experiment

In this section, we empirically evaluate the proposed SPE attention.

5.1 Experimental Setup

Tasks. Following (Zhang et al., 2024; Pei et al., 2023; Jin et al., 2022), we evaluate SPE attention on two types of tasks: (i) ***Error detection***, which aims to determine whether the provided code block contains a bug. We follow (Pei et al., 2023) to use Defect4J (Just et al., 2014), a collection of over 170 types of bugs across 17 open-source projects, for evaluation. We label each function based on the existence of bugs and split all these functions into training (70%) and testing (30%) split with a strict non-overlapping. (ii) ***Function name prediction***, which is a multiple binary classification task that predicts the summarization of the function behavior. Specifically, we tokenize the function names and formulate the function name prediction as a multi-label classification problem, as the strategies in (Jin et al., 2022). We use datasets from various code languages, including Open Source Java (Allamanis et al., 2016) for Java and Python 150K (Raychev et al., 2016) for Python. For Python 150K, we adopt its default training and testing splits. As Open Source Java doesn't have a default split, we split it randomly into training (70%) and testing (30%).

Baseline and Evaluation Metrics. We compare SPE with several open-source large language models specialized for code, including: CodeBERT (Feng et al., 2020), CodeLlama (Rozière et al.,

[4] We prove it in Appendix D.

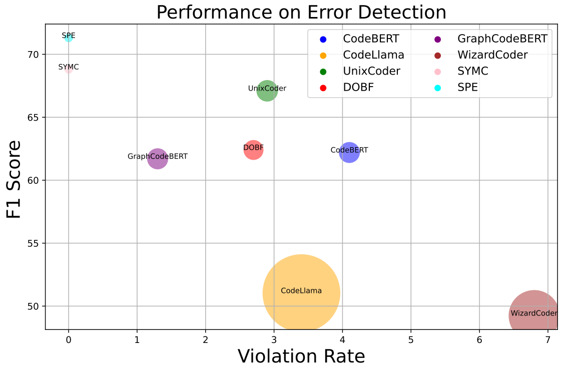


Figure 6: Performance comparison on the Java Error Detection task of various models based on F1 Score and Violation Rate. The area of each circle corresponds to the model size.

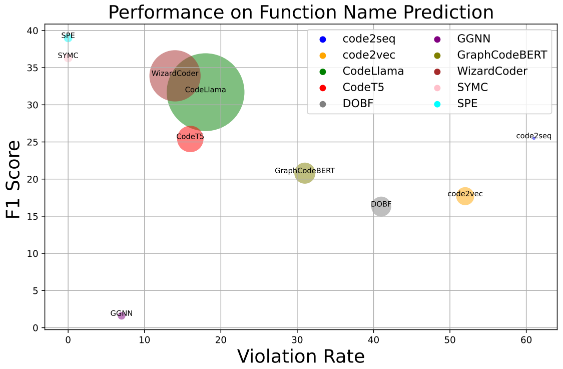


Figure 7: Performance comparison on the Java Function Name Detection task of various models based on F1 Score and Violation Rate. The area of each circle corresponds to the model size.

2024), code2seq (Alon et al., 2018), code2vec (Alon et al., 2018), CodeT5 (Wang et al., 2021), DOBF (Rozière et al., 2021), GraphCodeBERT (Guo et al., 2020), UnixCoder (Guo et al., 2022), and WizardCoder (Luo et al., 2023). For LLMs, we generate answers based on the fixed prompt template with further details provided in Appendix B. Moreover, SPE is compared with SYMC (Pei et al., 2023), which adds a bias term to the attention based on the dependency of the code, as discussed in Sec. 2.2. To ensure a fair evaluation, we adopt *F1 score* used in (Jin et al., 2022) as the evaluation metric. Specifically, let W be the ground truth token set, W' be the predicted token set, we compute the F1 score by the harmonic mean of precision $\frac{W \cap W'}{|W'|}$ and recall $\frac{W \cap W'}{|W|}$. Additionally, we evaluate equivariance by calculating the violation rate. This metric measures the model's consistency under input permutations. For each sample in the test set, we randomly apply a semantic-preserving permutation to the input and generate predictions for both the original and permuted code. A violation is recorded if the outputs differ after the permutations.

Implementation Details. We construct the SPE model with eight multi-head SPE attention layers and a two-layer linear classifier. To encourage variation among attention heads, for each layer, we apply the symmetry masks to six heads, the transpose symmetry mask to three heads, and standard attention to the rest three heads. After the attention layers, average pooling is performed, followed by two linear layers. We collect all tokens $\{t_i\}_{i=0}^{k}$ of function names in the training set, and set the length of the final output features as k. The further details of the model and training are provided in Appendix B.

5.2 Java Error Detection

Figure 6 shows the comparison of F1 score and violation rate for the error detection task on the Defect4J dataset. As can be seen, SPE shows a much higher F1 score than previous non-equivariant models with a relatively small trainable parameter count. For example, SPE outperforms CodeLlama by 20.27, but only requires 67.7M (1% of 7B) parameters, indicating the effectiveness and parameter efficiency of SPE Attention. Moreover, SPE shows a 0% violation, indicating a strict equivariance of the SPE model. Based on Figure 6, models with a lower violation rate tend to achieve a higher F1 score. This observation aligns with findings from equivariant studies (Gerken et al., 2022; Cohen and Welling, 2016a), further highlighting the effectiveness of equivariance. Compared with SYMC, SPE achieves an improvement with the same number of parameters in the F1 score by a large margin of $+2.5$. This significant improvement validates the effectiveness of incorporating a symmetry mask in the SPE attention.

5.3 Java Function Name Prediction

Figure 7 shows the comparison of F1 score and violation rate for the function name prediction task on the Open Source Java dataset. As can be seen, SPE demonstrates superior performance, achieving the highest F1 score among the models evaluated, despite having a relatively small trainable parameter count of 68.4M. Notably, SPE outperforms CodeLlama by 7.3 points while using only 1% of CodeLlama's 7B parameters, validating its efficiency and effectiveness. Moreover, SPE maintains a 0% vi-

Table 1: Comparison of F1 score for the function name prediction task on the Python 150K dataset.

Model	#Parameters	F1 Score
code2seq	6.3 M	27.3
CodeLlama	7 B	32.9
SYMC	72.9M	33.2
SPE	72.9M	37.1

olation rate, indicating strict equivariance, which is crucial for maintaining consistent predictions. Similar to the error detection task, models with a lower violation rate tend to achieve a higher F1 score. This further emphasizes the benefits of implementing equivariance in code processing models. Compared to SYMC, SPE shows a significant improvement in F1 score, with an increase of 2.7 points, using the same number of parameters. This substantial enhancement underscores the advantage of incorporating a symmetry mask in the SPE model's attention mechanism.

5.4 Python Function Name Prediction

Table 1 shows the comparison of F1 scores for the function name prediction task on the Python 150K dataset. As shown, SPE brings significant improvement to SYMC by a large margin of 3.9 in F1 score, validating the effectiveness of SPE Attention. Compared to larger models (i.e., CodeLlama with 7B parameters), SPE achieves a higher performance (+4.2) with significantly fewer parameters. Overall, the results on the Python task align well with those in Java, showing a superior performance of SPE across various languages and tasks.

5.5 Graph Computation Time

To compare computational efficiency, we collected average computation times for the Program Interpretation Graph (PIG), and Program Dependency Graph (PDG) used in SYMC (Pei et al., 2023) with the Python 150K dataset. The times were averaged over fifty file reads.

The PIG is generated by considering all possible execution paths within a code block, recording an edge whenever two lines are dependent in any execution path. As shown in Table 2, collecting the PIG requires 750 times more time than collecting the DPG. During implementation, SYMC was compelled to use the PDG as an over-approximation of equivariance (Pei et al., 2023). In contrast, our approach does not necessitate such a theoretical relaxation, and our graph is as efficient as the PDG.

Table 2: Average computation time for obtaining the Directed Layered Graph (DLG) in our work, compared to the Program Interpretation Graph (PIG) and the Program Dependency Graph (PDG) used in SYMC.

Graph Representation	DLG	PIG	PDG
Computation Time	0.0554 s	42.0709 s	0.0553 s

Table 3: F1 Score of SPE and two variants of SPE on the Python150K dataset.

Models	Performance
SPE (all)	38.2
SPE (*w.o.* pe)	22.5
SPE	39.0

5.6 Ablation Study

We conduct experiments for the function name prediction task to investigate the effect of two components of our model: (i) SPE (all), which applies the symmetry masks to all of the attention heads. (ii) SPE (*w.o.* pe), where we train the model by using the input without this sentence-level positional encoding. Table 3 shows the F1 Score comparison between SPE and the two variants. As can be seen, SPE performs slightly better than SPE (all), indicating that increasing the diversity of heads can boost performance. Moreover, the performance of SPE (*w.o.* pe) drops significantly compared to SPE, showing that sentence-level position information is still necessary in the SPE model.

6 Conclusion

In this paper, we highlight a unique symmetry in codes: the semantic-preserving permutations. We incorporate semantic-preserving permutations equivariance to an attention computation by utilizing symmetry masks from a novel directed layered graph. We show that such directed layered graph has a strong connection with the possible semantic-preserving permutations on a code block. We provide a theoretical proof demonstrating that this SPE attention mechanism is equivariant to semantic-preserving permutations, and empirically show its superiority over existing methods. Furthermore, we establish that the directed layered graph offers better computational efficiency.

Acknowledgement

This work was supported by NSFC grant under grant number 62136005.

Limitations

While the proposed SPE attention is promising for processing code, currently it cannot be integrated into generative models. This limitation arises because the symmetry mask can be obtained once the code is complete. Furthermore, for newer programming languages where parsing tools are unavailable, the proposed SPE attention is unable to acquire the required symmetry mask.

Ethic Statement

There is no ethical problem in our study.

References

Miltiadis Allamanis, Hao Peng, and Charles Sutton. 2016. A convolutional attention network for extreme summarization of source code. *CoRR*, abs/1602.03001.

Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. 2018. code2seq: Generating Sequences from Structured Representations of Code. *arXiv e-prints*, arXiv:1808.01400.

Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2018. code2vec: Learning distributed representations of code. *Preprint*, arXiv:1803.09473.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé 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, and 39 others. 2021. Evaluating large language models trained on code. *CoRR*, abs/2107.03374.

Taco S. Cohen, Mario Geiger, Jonas Köhler, and Max Welling. 2018. Spherical cnns. *CoRR*, abs/1801.10130.

Taco S. Cohen and Max Welling. 2016a. Group equivariant convolutional networks. *CoRR*, abs/1602.07576.

Taco S. Cohen and Max Welling. 2016b. Steerable cnns. *CoRR*, abs/1612.08498.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1536–1547, Online. Association for Computational Linguistics.

Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. 1987. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349.

Jan E. Gerken, Oscar Carlsson, Hampus Linander, Fredrik Ohlsson, Christoffer Petersson, and Daniel Persson. 2022. Equivariance versus augmentation for spherical images. *CoRR*, abs/2202.03990.

Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. Unixcoder: Unified cross-modal pre-training for code representation. *arXiv preprint arXiv:2203.03850*.

Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, and 1 others. 2020. Graphcodebert: Pre-training code representations with data flow. *arXiv preprint arXiv:2009.08366*.

Qing Huang, Zhou Zou, Zhenchang Xing, Zhenkang Zuo, Xiwei Xu, and Qinghua Lu. 2023. Ai chain on large language model for unsupervised control flow graph generation for statically-typed partial code. *Preprint*, arXiv:2306.00757.

Xin Jin, Kexin Pei, Jun Yeon Won, and Zhiqiang Lin. 2022. Symlm: Predicting function names in stripped binaries via context-sensitive execution-aware code embeddings. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, CCS '22, page 1631–1645, New York, NY, USA. Association for Computing Machinery.

René Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4j: a database of existing faults to enable controlled testing studies for java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, page 437–440, New York, NY, USA. Association for Computing Machinery.

Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. 2023. Wizardcoder: Empowering code large language models with evol-instruct. *arXiv preprint arXiv:2306.08568*.

Kexin Pei, Weichen Li, Qirui Jin, Shuyang Liu, Scott Geng, Lorenzo Cavallaro, Junfeng Yang, and Suman Sekhar Jana. 2023. Exploiting code symmetries for learning program semantics. In *International Conference on Machine Learning*.

Veselin Raychev, Pavol Bielik, and Martin Vechev. 2016. Probabilistic model for code with decision trees. *SIGPLAN Not.*, 51(10):731–747.

Baptiste Rozière, Marie-Anne Lachaux, Marc Szafraniec, and Guillaume Lample. 2021. DOBF: A deobfuscation pre-training objective for programming languages. *CoRR*, abs/2102.07492.

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, and 7 others. 2024. Code llama: Open foundation models for code. *Preprint*, arXiv:2308.12950.

Victor Garcia Satorras, Emiel Hoogeboom, and Max Welling. 2021. E(n) equivariant graph neural networks. *CoRR*, abs/2102.09844.

Yewei Song, Cedric Lothritz, Daniel Tang, Tegawendé F. Bissyandé, and Jacques Klein. 2024. Revisiting code similarity evaluation with abstract syntax tree edit distance. *Preprint*, arXiv:2404.08817.

Hugo Touvron, Louis Martin, Kevin R. Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Daniel M. Bikel, Lukas Blecher, Cristian Cantón Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, and 49 others. 2023. Llama 2: Open foundation and fine-tuned chat models. *ArXiv*, abs/2307.09288.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *CoRR*, abs/1706.03762.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859*.

Gokul Yenduri, Manju Ramalingam, Govardanan Chemmalar Selvi, Y. Supriya, Gautam Srivastava, Praveen Kumar Reddy Maddikunta, G. Deepti Raj, Rutvij H. Jhaveri, B. Prabadevi, Weizheng Wang, Athanasios V. Vasilakos, and Thippa Reddy Gadekallu. 2023. Gpt (generative pre-trained transformer)— a comprehensive review on enabling technologies, potential applications, emerging challenges, and future directions. *IEEE Access*, 12:54608–54649.

Xiaoling Zhang, Zhengzi Xu, Shouguo Yang, Zhi Li, Zhiqiang Shi, and Limin Sun. 2024. Enhancing function name prediction using votes-based name tokenization and multi-task learning. *Proc. ACM Softw. Eng.*, 1(FSE).

Allan Zhou, Kaien Yang, Kaylee Burns, Adriano Cardace, Yiding Jiang, Samuel Sokota, J. Zico Kolter, and Chelsea Finn. 2023. Permutation equivariant neural functionals. *Preprint*, arXiv:2302.14040.

Yongshuo Zong, Tingyang Yu, Ruchika Chavhan, Bingchen Zhao, and Timothy Hospedales. 2024. Fool your (Vision and) language model with embarrassingly simple permutations. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pages 62892–62913. PMLR.

Appendix

A Details of Definitions

A.1 Groups and representations

Definition A.1. Let G be a set and $*$ be an operation. Then, $(G, *)$ is a *group* if the following holds:

1. There exists a $e \in G$, such that for any $g \in G$, $g * e = e * g = g$. We call this e the identity.
2. For any element $g \in G$, there exists $h \in G$ such that $g * h = h * g = e$. We call this the inverse element, and denote it as $-h$ or h^{-1} depending on the context of the operation.
3. G is closed under this operation. That is, for any $g_1, g_2 \in G$, $g_1 * g_2$ is always in G.
4. For any $g, h, k \in G$, $g * (h * k) = (g * h) * k$.

After defining a group $(G, *)$, it is common to simply refer to it as G when the operation is clear from the context.

Definition A.2. Given a group $(G, *)$, a *representation* or a *group action* of G on a vector space V is a map ρ with inputs in G. For any $g \in G$, $\rho(g)$ is a *linear map* on V. Furthermore, $\rho(g_1 g_2) = \rho(g_1)\rho(g_2)$. We denote the representation by (ρ, V) or simply ρ.

As mention in Sec. 3, we focus on permutation group $(S_n, *)$. Here we take S_3 as an example

A.2 Example Permutation Group: S_3

The group S_3 has $3! = 6$ element. Each element can be thought as an unique permutation on 3 elements. For example, (123) means 1 moves 2, 2 moves 3 and 3 moves to 1. $(12)(3)$ or simply (12) means 1 goes to 2, 2 goes to 1, and 3 stays unchanged. The identity elements is e, keeping everything unchanged. Multiplication of two permutation means doing both permutation consecutively, performing the one on the right first. For example, $(123) * (12)$ means 1 goes to 2 by the right permutation, and 2 goes to 1 by the left permutation. 2 goes to 1 by the right permutation, and 2 goes to 3 by the left permutation. 3 goes to 3 by the right permutation, and 3 goes to 1 by the left permutation. Thus, the result permutation $(123) * (12) = (13)$. Here, we present a complete multiplication table (also known as the Cayley table) for S_3.

$*$	e	(12)	(23)	(13)	(123)	(132)
e	e	(12)	(23)	(13)	(123)	(132)
(12)	(12)	e	(123)	(132)	(23)	(13)
(23)	(23)	(132)	e	(123)	(13)	(12)
(13)	(13)	(123)	(132)	e	(12)	(23)
(123)	(123)	(13)	(12)	(23)	(132)	e
(132)	(132)	(23)	(13)	(12)	e	(123)

B Additional Model Details

We use the tokenizer of CodeLlama, and the SPE model is trained from scratch, composed of 8 SPE attention layers, with 12 heads. The attention embedding dimension is 768. We trained on the training set for 20 epochs, with the Adam optimizer and learning rate of 1e-4.

For generative models, we use a fixed prompt: I have a segment of Python function code and need help determining a suitable name for it. Analyze the function's logic, inputs/outputs, and purpose, then suggest a suitable full name. Then, generate a common abbreviation (if applicable) based on the full name. Return the format: [full_name] [abbreviation], with each enclosed in separate brackets. Only provide the two names, no additional text. Here is the code: TEXT OF CODE.

Table 4: The performance of different models in terms of F1 score and violation rate for the error detection task on the Defect4J dataset. A violation of 0% indicates equivariance.

Model	#Parameters	F1 Score	Violation (%)
CodeBERT	476M	62.2	4.1
CodeLlama	7B	51.03	3.4
CodeT5	770M	63.3	6
DOBF	428M	62.4	2.7
GPT-4	N/A	51.56	13.5
GraphCodeBERT	481M	61.7	1.3
UnixCoder	504M	67.1	2.9
WizardCoder	3B	49.24	6.8
SYMC	67.7M	68.8	0
SPE	67.7M	71.3	0

C Additional Experiment Results

In Sec.5, we provided visualizations of models performance and violation rate in Figure 6 and 7. In the table below, we provide numerical values, as well as performance on GPT-4.

C.1 Performance under the permutation attack

On top of the violation rate, we adapt the ”permutation attack“(Zong et al., 2024) in Table 6: we apply random semantic-preserving permutations to the codes and record the lowest F1 Score amoung all permutations. We only consider four random permutations for efficiency. According to the results shown in the table below, we can see that the higher the "violation rate" is, the more vulnerable the model is to permutation attacks. Due to the permutation equivariance, the proposed SPE attention is robust to such attack, and the predictions are very consistent.

C.2 Finetuned Models

The pretrained models are not finetuned, as they have already been tuned on general code datasets. To evaluate whether finetuning would help reducing the violation rate, we further fine-tuned the code llama model for comparison. As shown in the following table, our SPE attention still outperforms the finetuned version of code llama, and the Violation Rate of code llama is minimally affected by fine-tuning.

D Proofs

In this section, we provide a complete proof for our theorems in the main paper.

Proposition D.1. *Adding traditional positional encoding is not equivariant. Resetting the positional encoding at the beginning for each line is equivariant.*

Proof. The proof of the first claim is rather straight-forward. We denote the input embedding as x and the positional encoding as e. We denote the i-th embedding or positional encoding as $x(i)$ or $e(i)$ respectively. The design of positional encoding assures that $e(i) = e(j)$ if and only if $i = j$. For any permutation ρ, given a index i, we observe that $x(\rho(i)) = \rho(x(i))$, but $e(\rho(i)) \neq \rho(e(i))$. Therefore the addition is not equivariant.

To prove the second claim, assume the positional encoding reset at the beginning of each options. Then, if a option is permuted to a new place, its corresponding positional encoding is the same. Additionally, it is trivial that the embedding of options is not changed by the permutation either. Therefore, the sum of reset positional encoding and input is equivariant. □

Table 5: The performance of different models in terms of F1 score and violation rate for the function name prediction task on the Open Source Java dataset.

Model	#Parameters	F1 Score	Violation (%)
code2seq	6.3 M	25.5	61
code2vec	348 M	17.7	52
CodeLlama	7 B	31.7	18
CodeT5	770 M	25.4	16
DOBF	428 M	16.3	41
GGNN	53 M	1.6	7
GPT-4	N/A	30.3	43
GraphCodeBERT	481 M	20.8	31
WizardCoder	3B	33.9	14
SYMC	68.4M	36.3	0
SPE	68.4M	39.0	0

Table 6: The performance (%) of different models under permutation attack

Tasks	**code2seq**	**CodeLlama**	**SPE**
Java Name Prediction	13.4 (-12.1)	26.2 (-5.5)	39.0 (-0.0)
Python Name Prediction	14.9 (-12.4)	26.5 (-6.4)	37.1 (-0.0)

Table 7: The F1-Score of different tasks, with the violation rate (%) in parentheses

Tasks	**Original CodeLlama**	**CodeLlama Finetuned**	**SPE Attention**
Java Error Detection	51.0 (3.4)	55.4 (3.2)	71.3 (0)
Java Name Prediction	31.7 (18)	34.4 (16.8)	39.0 (0)
Python Name Prediction	32.9 (14.2)	35.8 (16.1)	37.1 (0)

Theorem D.2. *(Same as Theorem 4.3)*
A permutation ρ of C is semantic-preserving if and only if c_i and ρc_i are on the same layer for all $c_i \leq n$.

Proof. We show the if statement first. Let ρ be a semantic-preserving permutation. Then, by definition, all dependency pairs are preserved by this permutation. Based on the construction of the DLG, since all dependency pairs is unchanged, we still have the same source nodes, and the longest distance is unaffected. Therefore, the DLG is the same. The only possible difference is that the same node might be put into the same layer in different order, thus the permutation on that layer.

To show the only if statement, we instead show the contra-positive. If ρ is not a semantic-preserving permutation. Then, one of the following three cases will happen after permutation: 1. a source node depends on other nodes. 2. A node previously dependent to other nodes becomes a source node. 3. The longest path to a specific code is changed, due to the change in dependency pairs. Regardless of which case, the directed layered graph is always changed, and affected nodes are moved to a different layer. □

Now, we prove Theorem 4.6. First, we need the following lemma.

Lemma D.3. *Our symmetry mask is equivariant only to P_n.*

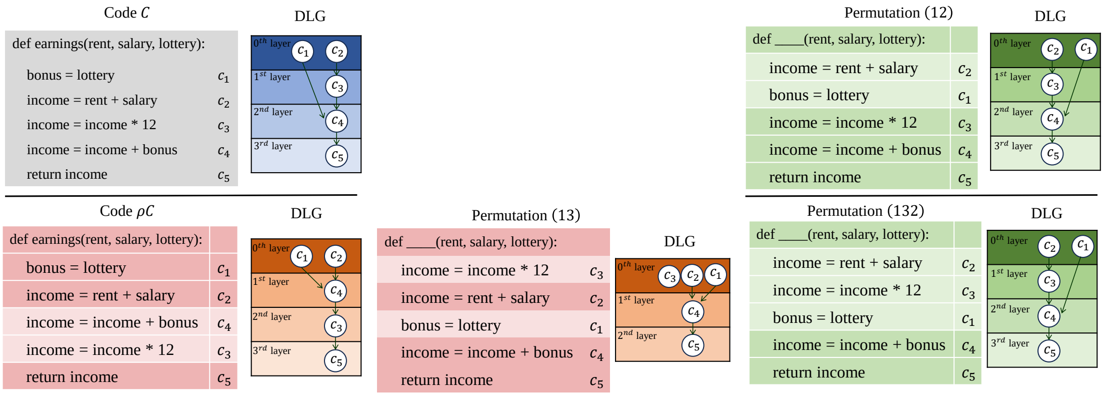
Proof. The construction of such symmetry mask is a one-to-one correspondence with the DLG. Combining with the previous theorem, this proof is trivial. □

Theorem D.4. *(Same as theorem 4.6)*
The combination of the symmetry mask and the attention is equivariant only to P_n.

Proof. The previous theorem already showed that the symmetry mask is equivariant only to P_n. Combined with the fact that attention is equivariant to S_n, it is easy to see that the combination of these two is equivariant to P_n. □

E Visualization of DLG

In this section, based on the example in Figure 3, we provide more visualization of the DLG in Figure 8. The only two permutations that preserved semantics are the permutation (12) and (132). Their DLGs differ from the original code's DLG only by permuting c_1 and c_2 in the first layer. All other permutations change the semantics of the code. For example, we visualize the permutation (13) in Figure 8 by swapping c_1 and c_3. In the original C, $(2, 3) \in C_{de}$. However, this dependency vanishes after permutation, as c_2 and c_3 no longer depend on each other. The resulting DLG is completely different from the original DLG.



Code C

```
def earnings(rent, salary, lottery):
    bonus = lottery               c1
    income = rent + salary        c2
    income = income * 12          c3
    income = income + bonus       c4
    return income                 c5
```

Permutation (12)

```
def ____(rent, salary, lottery):
    income = rent + salary        c2
    bonus = lottery               c1
    income = income * 12          c3
    income = income + bonus       c4
    return income                 c5
```

Code ρC

```
def earnings(rent, salary, lottery):
    bonus = lottery               c1
    income = rent + salary        c2
    income = income + bonus       c4
    income = income * 12          c3
    return income                 c5
```

Permutation (13)

```
def ____(rent, salary, lottery):
    income = income * 12          c3
    income = rent + salary        c2
    bonus = lottery               c1
    income = income + bonus       c4
    return income                 c5
```

Permutation (132)

```
def ____(rent, salary, lottery):
    income = rent + salary        c2
    income = income * 12          c3
    bonus = lottery               c1
    income = income + bonus       c4
    return income                 c5
```

Figure 8: DLGs of the original code and its permuted versions. Green DLGs are different with the blue DLG with permutation on the 0-th layer, while red DLGs are different with permutations between layers.