Implant Global and Local Hierarchy Information to Sequence based Code Representation Models

Kechi Zhang[†]
Key Lab of High Confidence Software
Technology, MoE (Peking University)
Beijing, China
zhangkechi@pku.edu.cn

Zhuo Li[†]
Key Lab of High Confidence Software
Technology, MoE (Peking University)
Beijing, China
lizhmq@pku.edu.cn

Zhi Jin*
Key Lab of High Confidence Software
Technology, MoE (Peking University)
Beijing, China
zhijin@pku.edu.cn

Ge Li*
Key Lab of High Confidence Software
Technology, MoE (Peking University)
Beijing, China
lige@pku.edu.cn

Abstract—Source code representation with deep learning techniques is an important research field. There have been many studies that learn sequential or structural information for code representation. But sequence-based models and non-sequence-models both have their limitations. Researchers attempt to incorporate structural information to sequence-based models, but they only mine part of token-level hierarchical structure information. In this paper, we analyze how the complete hierarchical structure influences the tokens in code sequences and abstract this influence as a property of code tokens called hierarchical embedding. The hierarchical embedding is further divided into statement-level global hierarchy and token-level local hierarchy. Furthermore, we propose the Hierarchy Transformer (HiT), a simple but effective sequence model to incorporate the complete hierarchical embeddings of source code into a Transformer model. We demonstrate the effectiveness of hierarchical embedding on learning code structure with an experiment on variable scope detection task. Further evaluation shows that HiT outperforms SOTA baseline models and show stable training efficiency on three source code-related tasks involving classification and generation tasks across 8 different datasets.

Index Terms—Code Representation, Code Summarization, Code Classification, Clone Detection

I. INTRODUCTION

Code representation is a hot research topic in software engineering (SE) and machine learning (ML) fields. Machine learning for code representation learning aims to convert programs of different formats (sequential formats such as token sequences, structural formats such as abstract syntax trees, dependency graphs, etc.) into vectorized semantic embeddings. These representation vectors can be applied on many downstream tasks, such as code classification [31], type inference [3], code summarization [6, 9, 19, 21], etc.

Most existing code representation methods can be divided into two categories: Sequence-based models [2, 12] are skilled
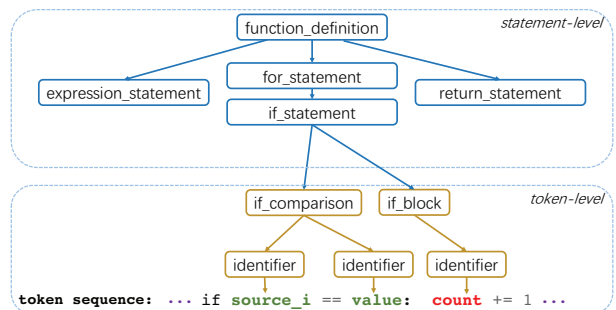
---

† The two authors share equal contribution.
* Corresponding authors

(a) Hierarchical location affects the operational semantics of a statement

(b) Hierarchy information within a statement affects the semantics of a token.

Fig. 1. An illustrative example of the hierarchy information in source code. The hierarchy of a token refers to the hierarchical location of its statement (*global hierarchy*) and the local component of the token in the statement (*local hierarchy*). The token `source_i` and `value` are both `identifier` in `if_comparison` marked in green in (b) and they are more closely related.

at processing sequence order information with long-term dependency. But they are sub-optimal for capturing structural information [31, 47]. Non-sequence-based models, such as tree-based (ASTNN [47], TBCNN [31]) or graph-based models (GGNN [25]), focus on encoding structural information, but sacrifice the advantages of sequence models or suffer from narrow receptive fields. These methods predominantly leverage either sequential or structural information of source code, and ignore the combination of the two modal information.

Recently, some studies jointly learn both sequential and

structural information for code representation in sequence-based models. Hellendoorn et al. [17] proposed *GREAT*, a Transformer model using the relative positional encoding to bias the attention using edges of the data flow graph and control flow graph. Zügner et al. [49] proposed *CodeTransformer*, which combines distances computed on the AST in the self-attention operation. However, these methods use limited information of code structure, such as node distances on the program tree or graph. They focus on modeling structure as a relation between tokens with attention mechanism and overlook the full impact of hierarchical structure information, which weakens the structural information of source code.

In this work, we take a step to explore how the complete hierarchical structure influence the tokens in source code sequence and abstract this influence as a property of code tokens called hierarchical embedding. We further divide this influence into two aspects: ❶ hierarchical embedding of statements (named as **global hierarchy**). The hierarchical embedding will affect the operational semantics of a statement. For example, in Figure 1, the statement `count += 1` is written in the `for-block` and possibly to be executed more than once, while the statement `count = 0` will be executed only once in each function call to `countOccurrences`. ❷ hierarchical embedding of tokens within a statement (named as **local hierarchy**). Tokens are in different components in a statement, which will affect the semantics of each token. *e.g.*, as shown in Figure 1(b), the three identifiers, *source_i*, *value*, and *count*, in the `if-statement`, are references to variables in the function. However, the first two identifiers within `if-comparison` component (marked in green color in the figure) have more similar semantics than *count* because they appear in the same comparison expression.

To validate our intuition of modeling the global and local hierarchies with code sequences, we propose to unify hierarchical structure into the code sequence in a concise Transformer format. We name the network **Hi**erarchy **T**ransformer (HiT), a simple but effective model that can achieve a preferable balance between efficiency and effectiveness. The HiT model consists of two key components: a Transformer-based hierarchy encoder that learns the representation of the hierarchy information, and a Transformer-based sequence encoder that fuses the hierarchy information and token sequence information. Specifically, the hierarchy information is represented by the root-to-leaf paths in the code syntax tree and encoded by the Transformer-based hierarchy encoder.

We conduct an empirical study to investigate the impact of the global and local hierarchy. The experiment proves the effectiveness of the two different types of hierarchical embeddings. It shows that with the cost of a small number of additional parameters, our approach significantly enhances the performance of sequence-based code representation models and achieves stable training efficiency. We also design a variable scope detection task to show our model can well learn the scope information in global hierarchy and represent the relationship between sequence and structure information of source code. We further evaluate our approach on three

tasks: code classification, clone detection, and method name prediction, with 8 different datasets from different domains. These tasks include classification and generation. The results show that our approach outperforms existing code representation models and other state-of-the-art models. It indicates the benefits of our approach for aligning the complete hierarchical embedding with code tokens for source code understanding.

The contributions of this paper are summarized as follows.

- We analyze how the complete hierarchical structure influences tokens in code sequences and abstract this influence as a property of code tokens called hierarchical embedding.
- We propose HiT, a simple but effective model to incorporate the hierarchical embedding into Transformer. Through experimental analysis, we demonstrate our approach can well represent the scope information in global hierarchy. Our empirical study shows that global and local hierarchical information are essential for code representation models, while existing jointly learning models ignore the former.
- We evaluate our approach on 3 source code-related tasks with 8 different datasets, involving classification tasks and generation tasks on source code. Our experimental results prove that aligning the complete hierarchical embedding with code tokens is effective for learning representations for programs with stable training efficiency. [1]

II. RELATED WORK

A. Sequence-based Code Representation

Code representation learning is a hot research topic in software engineering and machine learning fields. Among various representation approaches, sequence models are the most mainstream code representation models, based on the concept of "naturalness" [2, 12, 18, 43], which argues that programming languages are usually simple, repetitive, and can be understood through the same approaches used in natural language processing. Sequence models are efficient and effective in processing the code token sequence, and have been applied across many SE tasks [1, 5, 13, 21, 27, 28, 33, 39]. Recently, sequence-based pre-trained models [14, 26, 29], such as CodeBERT, have achieved success in SE tasks, demonstrating the power of sequence-based models. Due to their large-scale parameters and massive pre-training data, We did not consider pre-trained models as baseline models in this work.

There are also researchers trying to encode structural information with sequence models [6, 9, 19, 24, 34]. Some leverage the program AST to model the structure in source code and represent source code as a set of leaf-to-leaf paths over ASTs [6]. Others use the flattened (AST) node sequence as input to model the structure in the source code. However, leaf-to-leaf paths would scrap the code sequence information. Using the flattened node sequences to encode tree structures makes the entire sequence representation significantly longer, and code tokens are interspersed with other non-terminal tree nodes. These approaches weaken the "naturalness" of the source

---

[1]Code and data are open-sourced on Github: https://github.com/zkcpku/HiT-hierarchy-transformer

code context. We declare that combining the "naturalness" and hierarchical structure information in the code sequence is essential. In this paper, we follow the research line of incorporating hierarchical structure into the code sequence representation model.

B. Non-Sequence-based Code Representation

Programs contain extensive structure information. Therefore, recent studies explore using tree-based [7, 11, 31, 41, 47] and graph-based [4, 15, 40, 42, 45, 48] models for code representation models. Although tree-based and graph-based models can directly capture structural information of source code, they are generally less efficient than sequence models. They require complex data preprocessing designed for particular languages. In the input tree or graph, the number of nodes in the receptive field of each node grows exponentially, which leads to models not being able to understand the complete sequence information well [8]. In our experiments, we include graph-based and tree-based models as our baselines.

Recently, some studies have also integrated structure information of source code into sequence-based models [16, 17, 22, 23, 49]. Some of these studies are designed for code representation tasks: Hellendoorn et al. [17] proposed Graph Relation Embedding Attention Transformer (GREAT), which biases Transformer with relational information from graph edge types, and achieves good performances on variable misuse task. Zügner et al. [49] proposed CodeTransformer, which computes pairwise distances on AST and integrates multiple relations into the attention module. However, these methods jointly learn source code's sequential and structural information with a simplified token-level hierarchical structure, such as node distances on the program tree or graph. They overlook the full impact of hierarchical structure information on the code sequence, and we will give a deep analysis of the hierarchy information contained in the code structure.

III. ANALYSIS OF HIERARCHY INFORMATION

In this section, we analyze how the complete hierarchy information influences the tokens in the source code sequence and motivating examples. Generally, for programs, we tokenize the source code to get the token sequences and encode them with a sequence model with hierarchical embeddings. Hierarchical embedding can be regarded as a semantic property of each token in the sequence. We further divide the semantics in the hierarchical embedding into two levels: hierarchical embedding of tokens within a statement (named as **local hierarchy**) and hierarchical embedding of statements (named as **global hierarchy**). We analyze the token-level and statement-level semantics introduced by each part of the hierarchical embedding.

A. Understanding Token-level Semantics with Local Hierarchy

The semantics of the token is related to the local hierarchy. Simply encoding a token with only the token embedding will lose local structural information. Specifically, the same tokens may have different semantics, but given the same token
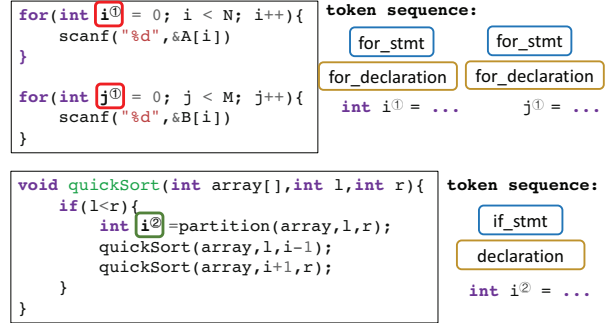

Fig. 2. An illustrative example of thse tokens in different contexts. The same token may have different semantics ($i^{①}$ and $i^{②}$ in two programs), and different tokens with similar context may have similar semantics ($i^{①}$ and $j^{①}$ in the first program)
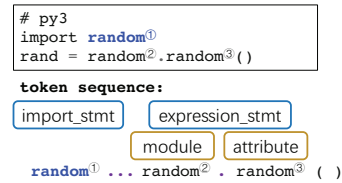

Fig. 3. An illustrative example of the repeated tokens in different statements. The repeated token random represents different semantics.

embedding. On the contrary, different tokens that appear in similar contexts may have similar semantics.

For example, in Figure 2, the variable i in the first program is used as a loop variable. In contrast, the variable i in the quick sort program represents the partition index. Although both programs define and use the variable i, they express different meanings with it. Furthermore, in the first program in Figure 2, there is another variable j used as a loop variable. We can discover that the variable i and j in the first program have similar semantics, which is different from the meaning of variable i in the second program. However, the embedding layer gives the same representation to i and a different representation to j. Another example is shown in Figure 3. In these two lines of python code, the token appears three times, indicating a module import, a module reference, and a module attribute, respectively. It is hard to exploit this implicit semantic difference for a traditional sequence model. Thus we believe that it is essential to model the structure of the local hierarchy to understand the meaning of tokens.

We revisit existing joint learning models for code representation [17, 49] and find that most of them focus on encoding relations between tokens in code sequences, such as node distances on the program tree/graph. They are skilled at encoding the token-level semantics and yield good performance. But our further analysis shows that hierarchical structural information contains more than token-level information.

B. Understanding Statement-level Semantics with Global Hierarchy

To illustrate the statement-level semantics contained in the global hierarchy, we show two classical examples: ❶ The semantics of statements is related to the global hierarchy.

159

```
row = len(M)③
for i in range(row):
    column = len(M[i])②
    for j in range(column):
        sum += M[i][j]①
        ①
②       ?   print(sum)
③
```

token sequence:

```
... M ... M[i] ... M[i][j] ...
```

expr for_stmt for_stmt
      expr   for_stmt
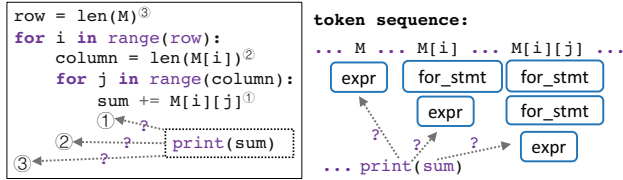   expr

```
... print(sum)
```

Fig. 4. An illustrative example of the implicit block structure ignored in token sequences. We can place the statement `print(sum)` in three places, where the token sequences are almost the same, but the semantics are different.


```
while True:
    n, x = map(int, input().split())
    if (n == x == 0):
        break
    count = 0
    for a in range(1, x // 3):
        for b in range(a + 1, x // 2):
            for c in range(b + 1, n + 1):
                Sum = a + b + c
                if (Sum == x):
                    count += 1
```

global hierarchy

while_stmt
for_stmt
for_stmt
for_stmt
if_stmt

> 91.7%
for *p02412*

Fig. 5. An illustrative solution example from problem id *p02412* in CodeNet Python800. There are 121 programs that have such a hierarchical position in the dataset. 111 of the 121 programs (more than 91.7%) are written for the problem *p02412*.

Most programming languages permit the creation of blocks and nested blocks. The block structure is fundamental for creating control flow and defining the scopes of variables. Thus modeling block structure is helpful for understanding control flow and variable scope. The control flow will influence the effects of a statement. Figure 4 shows an example about semantics of the statement in the block structure. The statement `print(sum)` can be placed at any of the three marked positions: in the inner `for` loop, in the outer `for` loop, and outside of the outer `for` loop. A small change in the statement's position will affect the program's output. Locating the statement in the block structure will help the model better determine the function of the source code. ❷ We also observe that the functionality of the program is closely related to the global hierarchy. Source code with similar functionalities tends to have similar global hierarchies. This unique global hierarchy can help the model distinguish the functionality and semantics of the program. To better confirm our observation, we conducted a simple statistical experiment on the code classification task on **Python800** dataset from the CodeNet project [35]. Figure 5 gives an example solution with problem id *p02412* in CodeNet. This program counts the number of triplets of numbers satisfying two requirements: each number is less than n and they sum up to k. There is a `if` statement in four levels of nesting while/for loop. We traverse the dataset and find that there are 121 programs that have a `while-for-for-for-if` hierarchical position. We surprisingly find that 111 of the 121 (about 91.7%) programs are written to solve problem *p02412*. And there are 300 programs in total for solving this problem, which means about 37% of the programs solving *p02412* use the special structure mentioned above. Through these statistics, we claim that the global hierarchy of source code is strongly related to the functionality and semantics of the program.

Through our analysis, we show that global and local hierarchical structures are essential for code representation models, while existing joint learning models ignore the former. We will conduct an empirical study to prove our point experimentally in Section VI-A.

IV. PROPOSED MODEL

A. Overview

In this work, we propose a source code representation model, HiT, to encode the code sequence and the hierarchy information simultaneously. The entire pipeline of our approach is show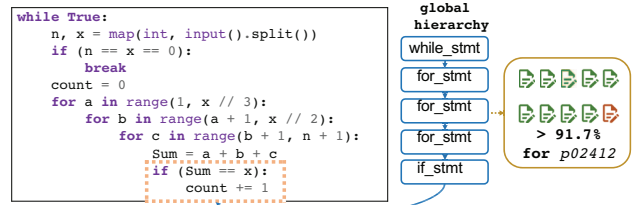n in Figure 6. To get the hierarchical position of the source code, we first parse the source code to the concrete syntax tree and extract paths from it. Each path is fed into a Transformer-based hierarchy encoder to embed the hierarchy information to hidden vectors. We then concatenate the hierarchy vector representation with the token embedding and use another Transformer-based sequence encoder to learn the final code representation. Finally, the representation vector can be fed into a linear classifier or a decoder for various downstream tasks.

B. Hierarchy Extraction

Hierarchical embeddings in source code are extracted from the concrete syntax tree. Parsing trees of source code contains rich structural information of source code. Concrete syntax tree (CST) and abstract syntax tree (AST) are parsing trees. The concrete syntax tree reflects the exact syntax of the source code, where each leaf node corresponds to a source code token. In theory, a CST can be converted to an AST equivalently. However, ASTs do not represent every detail appearing in the real syntax. For instance, the braces, semicolons, and parentheses are discarded in ASTs, making it hard to align the structural information with these code tokens. To extract the hierarchical embedding of each source code token, we use CSTs in this work.

As stated in Section III, we expect to model the hierarchical embedding of source code to better understand the semantics of tokens and statements. To get the hierarchical embedding, we extract all root-to-leaf paths from the CST. Considering that each leaf node in the CST corresponds to a source code token, the extracted root-to-leaf paths can be aligned to each token. The set of paths expresses the hierarchy of the program. Intuitively, we can further divide a path into two parts: the root-to-statement path and the statement-to-leaf path. The root-to-statement path represents the surrounding block structure of a statement. We name the structure *global hierarchy* which reflects the position of a statement in the program. The statement-to-leaf path represents the structure of the local context of a source code token. We name the structure *local hierarchy* which reflects the position of a token within the statement. *Step 1* in Figure 6 gives an illustration of different parts of the tree path. The *global hierarchy* and *local hierarchy* are included in the root-to-leaf path. In our main experiments, we use the root-to-leaf path in HiT model. To study the contribution of *global hierarchy* and *local hierarchy*,
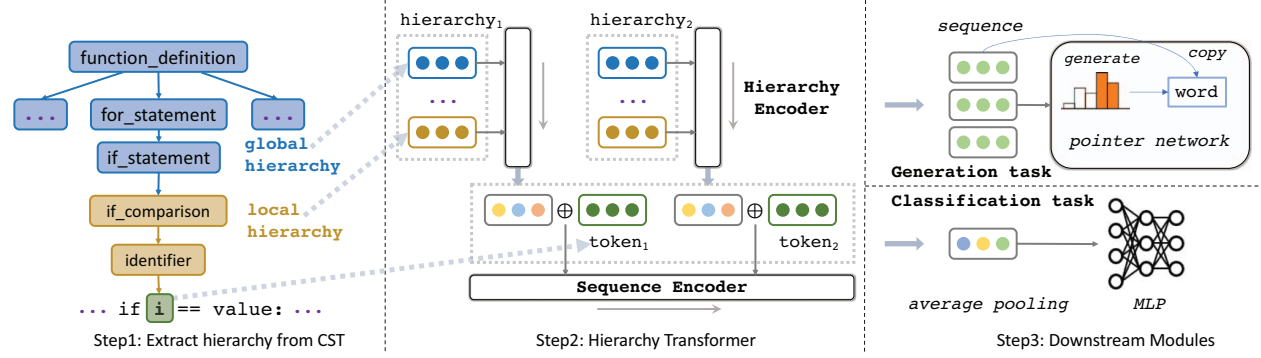
160

Fig. 6. The pipeline of our approach

respectively, we perform an empirical study where only one part of the path is used in RQ1 in Section VI-A.

C. Hierarchy Transformer

We propose a new sequence model, Hierarchy Transformer (HiT), which uses a combination of token sequences and hierarchy information for code representation. Given a token sequence with hierarchical path $s = [(\mathbf{n}_1, t_1), \cdots, (\mathbf{n}_l, t_l)]$, where l is the length of the token sequence, $(t_1, \cdots, t_l)$ indicate token sequence of the source code, and $\mathbf{n} = (n_1, \cdots, n_{l'})$ is a tree path (sequence of tree nodes). To process both token sequences and hierarchy information, we use a small Transformer to encode the paths $\mathbf{n}_k$. We then concatenate the representation of hierarchy information and token embedding and feed them into a larger Transformer model. We refer to the two transformers as Hierarchy Encoder and Sequence Encoder.

(i) Hierarchy Encoder. The hierarchy encoder is designed to process the hierarchy into a vector representation. For each tree path, we first embed the types of tree nodes and feed them into the Transformer model. Then we perform a mean pooling to get the representation of the whole path.

$$e_1, e_2, \cdots, e_{l'} = Embed(n_1, n_2, \cdots, n_{l'}) \quad (1)$$

$$h_1, h_2, \cdots, h_{l'} = Transformer_{hie}(e_1, e_2, \cdots, e_{l'}) \quad (2)$$

$$p = MeanPooling(h_1, h_2, \cdots, h_{l'}) \quad (3)$$

(ii) Sequence Encoder. The sequence encoder is designed to combine the hierarchy representation with tokens and process the resulting new sequence into the final code representation vector. We first concatenate the hierarchy representations p_i with the token embeddings e_i. Then we use another Transformer as the sequence encoder to get the final code representation.

$$E_1, E_2, \cdots, E_l = Embed(t_1, t_2, \cdots, t_l) \quad (4)$$

$$\mathbf{X} = (p_1||E_1, p_2||E_2, \cdots, p_l||E_l) \quad (5)$$

$$H_1, H_2, \cdots, H_l = Transformer_{seq}(\mathbf{X}) \quad (6)$$

D. Downstream Modules

The hierarchy encoder transforms the code sequence with hierarchy information into a vectorized representation. Most program processing tasks can be categorized into classification tasks and generation tasks. To apply our HiT on downstream tasks, we use different downstream modules according to the type of tasks.

(i) Classification. For classification tasks, models are required to classify programs based on the functionalities or other properties they implement. We first apply average pooling over the HiT output and get the global representation vector v. After getting v, we apply a 2-layer MLP as the classifier to get the classification result. The probability of the output label is then calculated with a softmax layer:

$$v = MeanPooling(H_1, H_2, \cdots, H_l)$$

$$o = g(W_2 \cdot f(W_1 \cdot v + b_1) + b_2)$$

$$P_i = \frac{exp(o_i)}{\sum_i exp(o_i)}$$

We use the standard cross entropy loss to train our model:

$$\mathcal{L} = -\sum_{i=1}^{|\mathcal{Y}|} \mathbb{1}_{y==i} \log P_i, \quad (7)$$

where $\mathbb{1}$ is the indicator function.

(ii) Generation. For generation tasks, models are required to generate the target sequence conditioned on the encoder output, such as method name prediction and code summarization. We pass all encoder output as a sequence to a transformer decoder. To generate out-of-vocabulary (OOV) tokens, we adopt a pointer network [38] based on the Transformer decoder model. The pointer model first attends to the encoder's output at each timestep and gets a hidden vector h_t^*.

$$e_i^t = W_3^T \tanh(W_1 H_i + W_2 s_t + b)$$

$$a^t = \text{softmax}(e^t)$$

$$h_t^* = \sum_i a_i^t H_i,$$

where W_1, W_2, W_3, b are learnable parameters, s_t represents the output of the transformer decoder at timestep t. After

obtaining the context vector, the model produces the vocabulary distribution and the copy probability $p_{copy} \in [0, 1]$ with H_i and s_t at this timestep. The p_c denotes the probability of copying tokens from the input sequence. On the contrary, $p_{gen} = 1 - p_{copy}$ indicates the probability of generate a token from the vocabulary. The probability of predicting the token w is calculated as follows:

$$P_v = \text{softmax}(W_4 h_t^* + b_1)$$
$$P_{copy} = \text{sigmoid}(W_5 h_t^* + b_2)$$
$$P(w) = p_{gen} P_v(w) + p_{copy} \sum_{i:w_i=w} a_i^t.$$

The copying mechanism enables the model to enhance its predictions by pointing at positions in the input sequence. During training, the loss for the output sequence is calculated as the average loss over the negative log likelihood of each target token w_t:

$$\mathcal{L} = \frac{1}{T} \sum_{t=0}^{T} -\log \sum_{\tilde{w}_t \in vocab} \mathbb{1}_{\tilde{w}_t == w_t} P(\tilde{w}_t) \qquad (8)$$

V. EXPERIMENTAL SETUP

With the extracted hierarchy information in the code sequence, we adopt HiT and perform extensive evaluation upon three code understanding tasks involving classification and generation tasks across 8 different datasets. We aim to investigate five research questions:

RQ1. Global Hierarchy & Local Hierarchy. What is the impact of the global and local hierarchy on code representation models? To what extent do the different types of hierarchy improve the performance of HiT?

RQ2. Scope Information In Global Hierarchy. Can HiT learn the scope information in global hierarchy? Is it important for the code representation model to focus on the global hierarchy?

RQ3. HiT vs. Transformer on Performance and Efficiency. Is the hierarchy information in HiT helpful for sequence models? What is the parameter cost and training efficiency of HiT compared with the vanilla Transformer?

RQ4. Performance on Classification Tasks. How does HiT perform compared with the SOTA models on the code classification and clone detection tasks?

RQ5. Performance on Generation Tasks. How does HiT perform on generation tasks? Can HiT produce better results than SOTA models on the method name prediction task?

A. Subject tasks and Datasets

Our experiments are conducted upon two representative source code classification tasks (*i.e.*, code classification and clone detection) and one generation task (*i.e.*, method name prediction). We evaluate 8 widely used datasets in total, and the compared baseline models are among the classical models or the SOTA models. The statistics of these datasets are summarized in Tabel I.

1) **Code Classification:** In the code classification task, the model needs to predict the category of the given code snippet based on the semantics. In particular, for selected datasets, solutions under each question correspond to a category. We consider using Project CodeNet [35] and POJ-104 [31]. Project CodeNet contains over 14M code samples from two open judge platforms AIZU and AtCoder. It provides four large and challenging datasets for the code classification task, including **Java250**, **Python800**, **C++1000** and **C++1400**. **POJ-104** is collected from another pedagogical online judge system with 104 programming problems. It has been used by many previous studies in code classification and clone detection tasks.

2) **Clone Detection:** In the clone detection task, the model needs to detect whether two pieces of code implement the same functionality. We adopt the **POJ-Clone** and follow the previous task settings [29]. It aims to retrieve other programs that solve the same problem given a program. To test the generalization ability of different approaches, the training/validation/test is split based on the problems.

3) **Method Name Prediction:** In method name prediction, a method with its name masked is fed into the model, and the model needs to predict the original method name based on the given method body. We experiment on two datasets introduced in the CodeSearchNet (CSN) Challenge [20, 49]: **CSN-Python** and **CSN-Ruby**. The datasets are obtained by scraping from public repositories across the most popular projects on GitHub. We follow the work of Husain et al. [20] which splits the data based on the source repositories.

B. Evaluation Metrics

For code classification task, we adopt the measure in [35] and use the accuracy for this multiclass classification task.

For clone detection task, we follow Lu et al. [29] and use the $MAP@R$ [32] score for evalutation. $MAP@R$ is defined as the mean of the average precision scores, each of which is evaluated to retrieve the most similar R samples given a query. For a code (query), R is the number of other codes in the same class and $R = 499$ in the POJ-Clone dataset.

For method name prediction task, we adopted metrics in previous studies[6, 9, 49], which measure *precision*, *recall*, and *f1* on subtokens of generated method names.

C. Implementation Details

Hierarchy Extraction Parser. We extract the hierarchy information from the concrete syntax tree with *Tree-sitter*, a parser generator tool. Based on this tool, our approach is general and dependency-free enough to parse any programming language and extract the hierarchy information. In our experiments, we have selected datasets in *C++, Java, Python, Ruby* for evaluation, showing the generality and effectiveness.

Model Implementation. Our model is implemented based on the Pytorch framework. We conduct all experiments on a Tesla V100S GPU with 32GB of memory. Each experiment is run five times with random seeds and then averaged for final results. We set the embedding size and the hidden size to 256, and employ 8 heads in each transformer layer. For the code

TABLE I
STATISTICS OF DATASETS

		Code Classification					Clone Detection		Method Name Prediction	
		Java250	Python800	C++1000	C++1400	POJ-104	POJ-Clone		CSN-Ruby	CSN-Python
							Examples	Problems		
Size	Train	45,000	144,000	300,000	252,000	36,400	32,000	64	48,791	412,178
	Valid	15,000	48,000	100,000	84,000	5,200	8,000	16	2,209	23,107
	Test	15,000	48,000	100,000	84,000	10,400	12,000	24	2,279	22,176
Avg. Length	Token Seq	228.02	125.18	270.96	334.89	246.96	246.96		79.18	131.59
	Complete Hierarchy	9.26	7.66	7.14	7.62	9.15	9.15		6.89	9.12
	Global Hierarchy	7.15	4.05	4.75	5.06	6.22	6.22		5.62	5.69
	Local Hierarchy	2.11	3.61	2.39	2.56	2.92	2.92		1.26	3.42
	Target Seq	-	-	-	-	-	-		2.23	2.25

TABLE II
PERFORMANCE OF HiT VS. TRANSFORMER, AND HiT WITH DIFFERENT
TYPES OF HIERARCHY FOR ALL THREE TASKS.

(a) On Code Classification and Clone Detection Tasks

	Para	Code Classification (Accuracy)					POJ-Clone (MAP@R)
		Java250	Py800	C++1000	C++1400	POJ	
HiT	4.55M	**94.81**	**95.97**	**95.05**	**93.27**	**97.08**	**80.46**
Trans	4.50M	93.49	93.99	89.93	67.87	88.13	67.15
global	4.55M	93.79	94.84	91.35	83.90	94.56	74.85
local	4.55M	93.95	95.42	92.64	90.45	96.37	75.88

(b) On Method Name Prediciton Task

	Para	CSN-Ruby			CSN-Python		
		P	R	F1	P	R	F1
HiT	36.75M	**30.70**	**27.58**	**29.06**	**37.25**	**33.75**	**35.41**
Trans	34.89M	24.26	19.66	21.71	32.71	27.63	29.96
global	36.75M	24.90	22.89	23.87	34.26	29.29	31.58
local	36.75M	28.69	25.58	27.05	35.34	30.50	32.74

classification task and the clone detection task, the hierarchy encoder consists of 2 layers, and the sequence encoder consists of 6 layers. For the method name prediction task, the number of sequence encoder layers and decoder layers are set to 4 and 2. We use AdamW with a learning rate of $1e^{-4}$ and weight decay. We use spaces as the separator for tokenizer, and set the vocabulary size between 5000-8000 according to different tasks. For all baseline models, we retrain on the given datasets to get more reliable results. We try to keep the hyper-parameters the same as baseline models for a fair comparison.

VI. EXPERIMENTAL RESULTS

A. RQ1: Global vs. Local Hierarchy

To investigate the impact of different types of hierarchy information, we conduct an empirical study and feed HiT with the global hierarchy and local hierarchy, respectively. We perform the analysis on all datasets of three tasks. The experimental results are listed in Table II. The row *global* and *local* refers to the two types of hierarchy.

On the basic program understanding task of the code classification task, we observe that the local hierarchy outperforms the global hierarchy by 0.16%, 0.58%, 1.29%, 6.55%, and 1.81% in the five datasets. It shows that the local hierarchy is comparable and sometimes more effective than the global hierarchy for code classification. The *MAP@R* of the local

hierarchy on the clone detection task is improved by 1.03 compared with the global hierarchy. For the method name prediction task, the local hierarchy outperforms the global hierarchy by 3.18 and 1.16 in F1-score on two datasets.

Experiments show that both the global and local hierarchy outperform the original sequence model, indicating that they are helpful for the sequence model to distinguish the semantics and functionalities of the programs. The global hierarchy provides the surrounding block structure of a statement, which contains the nested structure and the global position of a statement in the code. And the local hierarchy provides the structure of the local context of a source code, which contains the type information and the local position of a token in the statement. Experimental results show that the type information and the local structure play significant roles in the code representation, which also confirms that the existing models can achieve good results based on token-level local hierarchy.

Furthermore, We also notice that the performance of using global hierarchy and local hierarchy separately is worse than using the complete hierarchy. **Therefore, we can show that both global and local hierarchical embeddings are essential for code representation.** We use the complete hierarchy to feed HiT in our follow-up experiments.

B. RQ2: Scope Information in Global Hierarchy

To investigate the scope information in global hierarchy learned by HiT, we design a new task called **Variable Scope Detection**. Given two variables in a program, the model needs to detect whether these two variables are in a scope. [2] Formally, given the representation vector of the two variables h_A, h_B, the probability $p_{scope_{A,B}}$ of variable A and B in the same scope is calculated by dot product following a sigmoid function:

$$p_{scope_{(A,B)}} = \text{sigmoid}(h_A W_s h_B) \tag{9}$$

where W_s is a learnable parameter. We provide examples of the task in Table III. Considering that the variable scope is a mapping of the hierarchical structure information on variable tokens, this task requires the model to learn the accurate global hierarchical block structure for sequence tokens.

In our experiment, we adopt Python800 and C++1400 datasets in Project CodeNet. We sample variable pairs and

---

[2]The scope of a variable is a block structure in the entire program where the variable is declared, used, and can be modified.

163

TABLE III
EXAMPLES FOR VARIABLE SCOPE
DETECTION IN C++ DATASET

Example Program	Variable Pair
if (i % 2 == 0)	*(C, B)*
C -= B;	*Same scope*
else	*(C, A)*
A -= D;	*Different scope*

TABLE IV
PERFORMANCE OF MODELS ON
VARIABLE SCOPE DETECTION
(ACCURACY)

Model	Python	C++
HiT	**80.27**	**76.12**
Transformer	77.18	63.77
GGNN	79.81	68.94
GREAT	79.39	69.38

(a) Python800   (b) C++1400

Fig. 7. Training efficiency of *HiT* and *Transformer* on Python800 and C++1400 datasets. In 7(a) We plot a line to show that *HiT* can achieve comparable results to *Transformer* with less training cost. In 7(b) it indicates that our model can achieve more stable training process on difficult datasets.

TABLE V
PERFORMANCE OF MODELS ON CODE CLASSIFICATION TASK

Model	Java250	Python800	C++1000	C++1400	POJ-104
RGCN	91.93	91.60	92.73	92.34	95.57
GGNN	93.64	92.23	91.72	92.48	94.80
TBCNN	92.84	93.17	94.77	88.29	96.20
ASTNN	92.86	93.80	94.61	90.17	96.79
TreeCaps	93.07	94.35	94.92	90.16	96.81
SBT	65.64	71.69	65.05	56.33	89.58
X-SBT	83.31	89.10	66.79	67.00	94.58
Transformer	93.49	93.99	89.93	67.87	88.13
GREAT	93.36	93.27	92.76	92.50	90.33
HiT	**94.81**	**95.97**	**95.05**	**93.27**	**97.08**
CodeBERT[†]	*96.47*	*97.41*	*86.13*	*83.05*	*98.40*

[†] CodeBERT is a pre-trained model with significantly more parameters than our model, with a parameter count nearly 27 times that of HiT.

extract about *7 million* pairs for Python and *65 million* pairs for C++ with balanced labels. The division of the dataset follows the settings suggested by CodeNet in the classification task. We extract the variable representation vector from the encoder output of trained models in classification task. For sequence models, we extract the corresponding token representation. For graph-based models, we extract from node representations. We compare with the vanilla Transformer, GGNN and GREAT. The selection details of baselines are discussed in Section VII-A. Experiment results are shown in Table IV.

Results show that with the hierarchical embedding, our model can understand the hierarchical block structure better and outperforms the vanilla Transformer and GGNN on the variable scope detection task. We can also observe that HiT performs even better on the C++ dataset by at least 6.7%. Our in-depth investigation on the dataset reveals that the programs in the C++ dataset are much longer and the relationship between variables is more complex compared with the Python dataset. The program graphs in the C++ dataset for tree-based and graph-based models are large. The number of nodes in the receptive field of each node grows exponentially, which leads to these models not being able to understand the complete sequence information well. **Our model retains the advantage of the sequence models to capture long-term semantic dependency, and shows the importance of learning the scope information in global hierarchy.**

C. RQ3: HiT vs. Transformer on Performance and Efficiency

To answer RQ3, we compare the hierarchy transformer with the vanilla Transformer. We list the results of our HiT and the vanilla transformer on all three tasks in Table II. We also show the parameter cost under different task settings. The column *Para* refers to the total number of parameters for different models. In the following experiments we will continue to use the same amount of parameters.

For the classification tasks, we find that on C++1400 dataset, adding the hierarchy information would cause the classification accuracy to improve by more than 25.4%. On C++1000 dataset, this improvement is more than 5.12%. It shows that the hierarchy information can significantly improve the stability and performance of sequence models, especially for those datasets with complex semantics and long programs. For the clone detection task, we observe that HiT outperforms the Transformer by 13% on *MAP@R*. Considering our experimental setup that training and testing sets are split according to OJ problems, HiT is more generalizable on the unseen
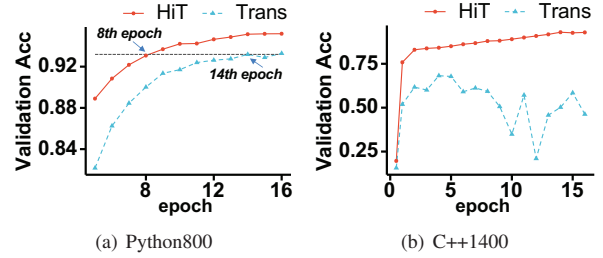
problems than the original sequence model. For the method name prediction task, upon the two datasets CSN-Ruby and CSN-Python, the F1-score improves by about 7.35% and 5.45%. This improvement is significant in both classification and generation tasks over 8 different datasets. Hence, the hierarchy information is essential for sequence models to generate accurate code representations for various tasks. Our model significantly enhances the original sequence model on different tasks, at a minimal extra parameter cost of 1%-5%.

To evaluate the training efficiency, We recorded the training process of the two models in Figure 7. We selected two representative datasets: Python800 and C++1400. The training process of our model is faster and more stable. On Python800 dataset, HiT can achieve comparable results to Transformer with less training epochs and time. For difficult datasets such as C++1400, HiT shows more stable learning ability, making it perform more efficiently on such datasets.

D. RQ4: On Classification Task

1) **RQ4.1: Code Classification:** We compare with the following state-of-the-art code representation models:
(1) Graph Neural Networks. We include RGCN ([36]) and GGNN ([25]) as baseline models.
(2) Tree-structured Neural Networks. We include TBCNN [31], ASTNN [47] and TreeCaps [11] as baseline models.
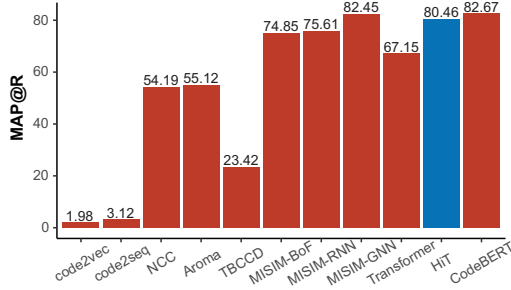
164

Fig. 8. Performance of models on POJ-Clone.[3]

Model	CSN-Ruby			CSN-Python		
	P	R	F1	P	R	F1
Code2seq	23.23	10.31	14.28	35.79	24.85	29.34
GGNN	19.15	14.11	16.24	24.07	19.09	21.29
SBT	19.84	12.22	15.12	30.92	18.32	23.01
X-SBT	22.82	13.04	16.60	34.58	20.69	25.89
Transformer	24.26	19.66	21.71	32.71	27.63	29.96
GREAT	24.66	22.25	23.39	35.09	31.62	33.26
CodeTrans	**31.46**	24.50	27.55	36.41	33.68	34.99
GTNM	24.59	20.11	22.13	32.98	27.73	30.13
HiT	30.70	**27.58**	**29.06**	**37.25**	**33.75**	**35.41**

(3) Traversal Sequences of ASTs. We compare our model with SBT [19] and XSBT [34]. SBT[19] represents the tree nodes into a token sequence with brackets to denote hierarchies. X-SBT [34] simplifies SBT by using an XML-like form.

(4) Transformer-based Models. In addition to the vanilla transformer, we also compare our model with GREAT[17]. For comparison, we also include results for CodeBERT, a widely used pre-trained model that has a much larger number of parameters than HiT. Specifically, CodeBERT has 125 million parameters, nearly 27 times the number in our model.

The results in Table V show that our model achieves the best performance on the code classification task over all baseline models across different program languages. The overall average accuracy on four datasets in Project CodeNet is at least 1.65% higher than other baseline models, and the accuracy on POJ-104 dataset is increased by at least 0.27% compared with the SOTA models. We observe that the graph-based or tree-based models are more effective than the vanilla Transformer. It proves that using only the token sequence, the sequence model cannot learn code representation well. Our proposed HiT addresses this issue for sequential models. With the enhancement of hierarchy information, HiT outperforms those graph/tree-based models. We also notice that directly feeding the traversal sequences of ASTs in the Transformer performs poorly. The flattened AST node sequence impairs the sequential information of the context in the source code. In comparison, our approach is much more efficient and effective in combining naturalness and hierarchy for code representation. Even compared to large-scale models such as CodeBERT, our model is still competitive. HiT achieves comparable performance and even outperforms on C++1000 and C++1400 datasets by 8.92%, 10.22%. The overall average accuracy for Project CodeNet of HiT is 4.01% higher than CodeBERT. We must be aware that CodeBERT requires a large model size and pre-training on over 8 million data samples (while our model can be trained on a single GPU and does not require pre-training). We also notice sequence models often struggle on C++1000 and C++1400 datasets, including pre-trained models. Further investigations are conducted in Section VII-B.

2) **RQ4.2: Clone Detection:** To further verify the generalization ability of the model in distinguishing program

[3]The results of baselines are from the CodeXGLUE benchmark.

semantics, we choose to evaluate our model on a clone detection dataset (POJ-Clone) for further evaluation. As we mentioned, the clone detection dataset is partitioned into training, validation, and test sets with different OJ problems. We compare our model with several state-of-the-art methods specially designed for code clone detection related tasks.

(1) Code2vec/Code2seq [6, 9] uses the attention-based method with leaf-to-leaf paths of AST to learn embeddings of codes.

(2) NCC [10] encodes programs by leveraging both the underlying data flow and control flow of the programs with LSTM to build a code similarity system.

(3) Aroma [30] is a code recommendation engine with the simplified parse tree (SPT).

(4) TBCCD [46] is a clone detection model with tree-based convolution networks. It achieves state-of-the-art performance on a simplified version dataset of clone detection. [4]

(5) MISIM [44] is a code clone detection system that incorporates the context-aware semantics structure (CASS) in its design. This structure has been carefully tailored to support the analysis of specific programming languages.

We list the results in Figure 8. In most cases, our HiT outperforms among baseline models, which improves by at least 25.34 in *MAP@R* except MISIM models. The MISIM models need to be evaluated on every possible combination of manually designed configurations [44], thus, the preprocessing process is complex. Our method is simple and easy to use, with comparable results with the best MISIM-GNN among those models. When conducted on sequence models, our performance is even better than MISIM-RNN.

We observe that in our challenging experimental setting, TBCCD performs poorly. We also notice that NCC, Aroma and MISIM both require complex preprocessing designed for particular languages. Our model is based on CSTs and is more generalizable and practical. Even compared with the extremely large pretrained models, our model achieves the comparable performance on the clone detection task.

E. RQ5: On Generation Task

To answer this RQ, in addition to the baselines mentioned, we also compare with the SOTA models on the method name prediction task, including CodeTransformer [49] and GTNM [27]. *(1) CodeTransformer* learns structure and context jointly, and achieves state-of-the-art performance on CSN datasets. *(2) GTNM* is a latest transformer-based model for method name prediction which extracts local contexts and project-level contexts and incorporates them into the sequence model. The original paper of GTNM uses contexts designed for Java that are not available for CSN-Ruby and CSN-Python. We use a variant of GTNM that only considers the local context for fairly comparison. In this experimental setup, we did not cover the tree models included in Table V, as we found that they did not perform well on this task. We use the same copy mechanism of HiT for all baseline models.

We list the results of our HiT with the baselines upon CSN-Ruby and CSN-Python in Table VI. The experimental results show that our HiT outperforms the baseline models significantly upon CSN-Ruby and CSN-Python. Specifically, our model outperforms CodeTransformer by 1.5071, 0.4216 in F1-score, respectively. And our model also significantly outperforms GTNM by 6.93, 5.28.

VII. DISCUSSIONS

A. Comparison with existing sequence/structure-based models

Due to the design of the model structure, the existing sequence model (*e.g.*, the vanilla Transformer [37]) or structural-based model (*e.g.*, ASTNN [47], TBCNN [31], GGNN [25]) is usually better at capturing the information of a certain modal. Some studies jointly learn both sequential and structural information for code representation in sequence-based models such as GREAT [17]. They focus on modeling structure as a relation between tokens with attention mechanism and overlook the full impact of hierarchical structure, especially for the global hierarchy. In this paper, we comprehensively investigate the impact of different types of hierarchy information on code representation tasks. In Section VI-B, we design a Variable Scope Detection task to check the ability of different models to capture the global hierarchy information. We choose one of the sequence models, structural-based models and jointly learning models as the baseline. Experiments show that our model can better identify the information brought in the hierarchy. Our model retains the advantage of the sequence models to capture long-term semantic dependency, and shows the importance of implanting the full hierarchy information.

B. Analysis of C++ datasets in Project CodeNet

We further investgate the reason why the sequence model does not 9perform well on the C++ dataset. Analysis of the code sequences in these C++ datasets revealed that they are longer (as shown in Table I). Upon further investigation, we

---

treat the programs from the same class as a single text snippet by concatenating them and calculate the TF-IDF cosine similarity between every two classes [41]. The average similarity for C++1000 and C++1400 is 0.787 and 0.754, while the average score for Java250 and Python800 is only 0.722 and 0.420. This suggests that code sequences in C++ datasets are highly similar even in different classes, making it difficult for sequence models to accurately classify them. HiT shows strong training stability, especially on such difficult datasets.

C. Time Efficiency of HiT

In our experiments, when training HiT on the Python800 dataset, the additional pre-processing step only required < 5 minutes to process 144000 samples, and HiT achieved the best performance after 122 minutes. In comparison, the vanilla transformer required 169 minutes. During inference, both models required approximately 30 seconds. This demonstrates the efficiency of HiT, especially at training time.

D. Threats to Validity

Threats to internal validity relate to the roles of the model architecture and hyper-parameters setting. In our experiments, we do a small-range grid search on learning rate and batch size settings. Another threat comes from the implementation of GTNM. We do not use additional project and document-level context for method name prediction as the original paper [27] because they are not available in CSN-Ruby/Python.

Threats to external validity mainly relate to tasks and datasets we choose. We counter this by evaluating our model on 8 different datasets of three tasks, including classification and generation tasks across 4 programming languages.

Threats to construct validity include the evaluation metrics we used in this work. These metrics are adequate for corresponding tasks and have been adopted by many previous studies [6, 9, 29, 32, 35, 49].

VIII. CONCLUSION

In this paper, we analyze how the complete hierarchical structure influences tokens in code sequence representation and put forward the property of hierarchical embedding, including statement-level global hierarchy and token-level local hierarchy. We propose HiT, a practical approach to incorporate hierarchical embedding into Transformer. We investigate the effectiveness of the global and local hierarchy with a detailed empirical study. The results show both hierarchies are essential for code representation models while existing joint learning models ignore the former. Our in-depth evaluations demonstrate that HiT can generate accurate and delicate representations and outperforms the SOTA baselines for classification and generation tasks on 8 challenging datasets.

IX. ACKNOWLEDGEMENT

---

[4] In the authors' paper, they evaluate their model on a clone detection dataset that is not partitioned based on OJ problems. The program semantics in the training and test sets are the same. However, we think this setting weakens the generalization ability of the model.

REFERENCES

[1] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2020. A Transformer-based Approach for Source Code Summarization. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics*.

[2] Miltiadis Allamanis, Earl T. Barr, Premkumar T. Devanbu, and Charles Sutton. 2018. A Survey of Machine Learning for Big Code and Naturalness. *ACM Comput. Surv.* 51, 4 (2018).

[3] Miltiadis Allamanis, Earl T Barr, Soline Ducousso, and Zheng Gao. 2020. Typilus: neural type hints. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*.

[4] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. 2018. Learning to Represent Programs with Graphs. In *International Conference on Learning Representations*.

[5] Miltiadis Allamanis, Hao Peng, and Charles Sutton. 2016. A convolutional attention network for extreme summarization of source code. In *International conference on machine learning*.

[6] Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. 2019. code2seq: Generating Sequences from Structured Representations of Code. In *International Conference on Learning Representations*.

[7] Uri Alon, Roy Sadaka, Omer Levy, and Eran Yahav. 2020. Structural Language Models of Code. In *Proceedings of the International Conference on Machine Learning*.

[8] Uri Alon and Eran Yahav. 2021. On the Bottleneck of Graph Neural Networks and its Practical Implications. In *the International Conference on Learning Representations*.

[9] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. code2vec: learning distributed representations of code. *Proc. ACM Program. Lang.* 3, POPL (2019).

[10] Tal Ben-Nun, Alice Shoshana Jakobovits, and Torsten Hoefler. 2018. Neural Code Comprehension: A Learnable Representation of Code Semantics. In *Advances in Neural Information Processing Systems*.

[11] Nghi D. Q. Bui, Yijun Yu, and Lingxiao Jiang. 2021. TreeCaps: Tree-Based Capsule Networks for Source Code Processing. In *Thirty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2021*.

[12] Luca Buratti, Saurabh Pujar, Mihaela A. Bornea, J. Scott McCarley, Yunhui Zheng, Gaetano Rossiello, Alessandro Morari, Jim Laredo, Veronika Thost, Yufan Zhuang, and Giacomo Domeniconi. 2020. Exploring Software Naturalness through Neural Language Models. *CoRR* abs/2006.12641 (2020). arXiv:2006.12641

[13] Ruichu Cai, Zhihao Liang, Boyan Xu, Zijian Li, Yuexing Hao, and Yao Chen. 2020. TAG : Type Auxiliary Guiding for Code Comment Generation. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*.

[14] 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*.

[15] Patrick Fernandes, Miltiadis Allamanis, and Marc Brockschmidt. 2019. Structured Neural Summarization. In *International Conference on Learning Representations*.

[16] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin B. Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. 2021. Graph-CodeBERT: Pre-training Code Representations with Data Flow. In *9th International Conference on Learning Representations, ICLR 2021*.

[17] Vincent J Hellendoorn, Charles Sutton, Rishabh Singh, Petros Maniatis, and David Bieber. 2020. Global relational models of source code. In *International Conference on Learning Representations*.

[18] Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, and Premkumar T. Devanbu. 2012. On the naturalness of software. In *34th International Conference on Software Engineering, ICSE 2012*.

[19] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2018. Deep code comment generation. In *2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC)*. IEEE.

[20] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. CodeSearchNet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436* (2019).

[21] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2016. Summarizing Source Code using a Neural Attention Model. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, ACL 2016*.

[22] Seohyun Kim, Jinman Zhao, Yuchi Tian, and Satish Chandra. 2021. Code Prediction by Feeding Trees to Transformers. In *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021*.

[23] Alexander LeClair, Siyuan Jiang, and Collin McMillan. 2019. A neural model for generating natural language summaries of program subroutines. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019*.

[24] Jia Li, Yongmin Li, Ge Li, Zhi Jin, Yiyang Hao, and Xing Hu. 2023. SkCoder: A Sketch-based Approach for Automatic Code Generation. *arXiv preprint arXiv:2302.06144* (2023).

[25] Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard S. Zemel. 2016. Gated Graph Sequence Neural Networks. In *International Conference on Learning*

Representations.

[26] Zhiyu Li, Shuai Lu, Daya Guo, Nan Duan, Shailesh Jannu, Grant Jenks, Deep Majumder, Jared Green, Alexey Svyatkovskiy, Shengyu Fu, and Neel Sundaresan. 2022. Automating Code Review Activities by Large-Scale Pre-Training *(ESEC/FSE 2022)*.

[27] Fang Liu, Ge Li, Zhiyi Fu, Shuai Lu, Yiyang Hao, and Zhi Jin. 2022. Learning to Recommend Method Names with Global Context. *CoRR* abs/2201.10705 (2022).

[28] Kui Liu, Dongsun Kim, Tegawendé F. Bissyandé, Tae-young Kim, Kisub Kim, Anil Koyuncu, Suntae Kim, and Yves Le Traon. 2019. Learning to spot and refactor inconsistent method names. In *ICSE 2019*.

[29] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin B. Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Li-dong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. 2021. CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation. In *NeurIPS Datasets and Benchmarks 2021*.

[30] Sifei Luan, Di Yang, Celeste Barnaby, Koushik Sen, and Satish Chandra. 2019. Aroma: code recommendation via structural code search. OOPSLA (2019).

[31] Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. 2016. Convolutional Neural Networks over Tree Structures for Programming Language Processing. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*.

[32] Kevin Musgrave, Serge J. Belongie, and Ser-Nam Lim. [n. d.]. A Metric Learning Reality Check. In *Computer Vision - ECCV 2020 - 16th European Conference*.

[33] Son Nguyen, Hung Phan, Trinh Le, and Tien N. Nguyen. 2020. Suggesting natural method names to check name consistencies. In *ICSE '20: 42nd International Conference on Software Engineering*.

[34] Changan Niu, Chuanyi Li, Vincent Ng, Jidong Ge, Liguo Huang, and Bin Luo. 2022. SPT-Code: Sequence-to-Sequence Pre-Training for Learning Source Code Representations. *CoRR* abs/2201.01549 (2022).

[35] Ruchir Puri, David S Kung, Geert Janssen, Wei Zhang, Giacomo Domeniconi, Vladimir Zolotov, Julian Dolby, Jie Chen, Mihir Choudhury, Lindsey Decker, Veronika Thost, Luca Buratti, Saurabh Pujar, Shyam Ramji, Ulrich Finkler, Susan Malaika, and Frederick Reiss. 2021. CodeNet: A Large-Scale AI for Code Dataset for Learning a Diversity of Coding Tasks. In *Neural Information Processing Systems Datasets and Benchmarks Track*.

[36] Michael Sejr Schlichtkrull, Thomas N. Kipf, Peter Bloem, Rianne van den Berg, Ivan Titov, and Max Welling. [n. d.]. Modeling Relational Data with Graph Convolutional Networks. In *ESWC 2018*.

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

[38] Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. 2015. Pointer networks. In *Advances in neural information processing systems*.

[39] Shangwen Wang, Ming Wen, Bo Lin, and Xiaoguang Mao. [n. d.]. Lightweight global and local contexts guided method name recommendation with prior knowledge. In *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*.

[40] Wenhan Wang, Ge Li, Bo Ma, Xin Xia, and Zhi Jin. 2020. Detecting Code Clones with Graph Neural Network and Flow-Augmented Abstract Syntax Tree. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE.

[41] Wenhan Wang, Kechi Zhang, Ge Li, Shangqing Liu, Anran Li, Zhi Jin, and Yang Liu. 2022. Learning Program Representations with a Tree-Structured Transformer.

[42] Yu Wang, Ke Wang, Fengjuan Gao, and Linzhang Wang. 2020. Learning semantic program embeddings with graph interval neural network. *Proc. ACM Program. Lang.* 4, OOPSLA (2020).

[43] Mengfei Yang, Bin Gu, Zhenhua Duan, Zhi Jin, Naijun Zhan, and Yunwei Dong. 2022. Intelligent program synthesis framework and key scientific problems for embedded software. *Chinese Space Science and Technology* 42, 4 (2022), 1.

[44] Fangke Ye, Shengtian Zhou, Anand Venkat, Ryan Marcus, Nesime Tatbul, Jesmin Jahan Tithi, Paul Petersen, Timothy G. Mattson, Tim Kraska, Pradeep Dubey, Vivek Sarkar, and Justin Gottschlich. 2020. MISIM: An End-to-End Neural Code Similarity System. *CoRR* abs/2006.05265 (2020). arXiv:2006.05265

[45] Pengcheng Yin, Graham Neubig, Miltiadis Allamanis, Marc Brockschmidt, and Alexander L Gaunt. 2019. Learning to Represent Edits. In *International Conference on Learning Representations*.

[46] Hao Yu, Wing Lam, Long Chen, Ge Li, Tao Xie, and Qianxiang Wang. [n. d.]. Neural detection of semantic code clones via tree-based convolution. In *Proceedings of the 27th International Conference on Program Comprehension, ICPC 2019*.

[47] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, and Xudong Liu. 2019. A novel neural source code representation based on abstract syntax tree. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019*.

[48] Kechi Zhang, Wenhan Wang, Huangzhao Zhang, Ge Li, and Zhi Jin. 2022. Learning to Represent Programs with Heterogeneous Graphs. In *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension (ICPC '22)*. 12 pages.

[49] Daniel Zügner, Tobias Kirschstein, Michele Catasta, Jure Leskovec, and Stephan Günnemann. [n. d.]. Language-Agnostic Representation Learning of Source Code from Structure and Context. In *9th International Conference on Learning Representations, ICLR 2021*.