HierarchyNet: Learning to Summarize Source Code with Heterogeneous Representations

Minh Huynh Nguyen[♣,*], **Nghi D. Q. Bui**[♦,*], **Truong Son Hy**[♠,♣], **Long Tran Thanh**[♥], **Tien N. Nguyen**[◇]

[♣] FPT Software AI Center, [♦]Department of Computer Science, Fulbright University, Viet Nam
[♠]Department of Mathematics and Computer Science, Indiana State University, USA
[♥]Department of Computer Science, University of Warwick, UK
[◇]Computer Science Department, The University of Texas at Dallas, USA [†]

Abstract

Code representation is important to machine learning models in the code-related applications. Existing code summarization approaches primarily leverage Abstract Syntax Trees (ASTs) and sequential information from source code to generate code summaries while often overlooking the critical consideration of the interplay of dependencies among code elements and code hierarchy. However, effective summarization necessitates a holistic analysis of code snippets from three distinct aspects: lexical, syntactic, and semantic information. In this paper, we propose a novel code summarization approach utilizing Heterogeneous Code Representations (HCRs) and our specially designed HIERARCHYNET. HCRs adeptly capture essential code features at lexical, syntactic, and semantic levels within a hierarchical structure. HIERARCHYNET processes each layer of the HCR separately, employing a Heterogeneous Graph Transformer, a Tree-based CNN, and a Transformer Encoder. In addition, HIERARCHYNET demonstrates superior performance compared to fine-tuned pre-trained models, including CodeT5, and CodeBERT, as well as large language models that employ zero/few-shot settings, such as CodeLlama, StarCoder, and CodeGen. Implementation details can be found at https://github.com/FSoft-AI4Code/HierarchyNet.

1 Introduction

Summarizing code is crucial for aiding developers in comprehending and maintaining source code. Yet, manual documentation is a laborious process. An automated method is required to craft comments efficiently. To generate precise summaries, a model should grasp lexical, syntax, and semantic aspects within the code. It's imperative to capture relationships such as data and control dependencies among program elements to enhance code representation learning for code summarization.

Early sequence-based techniques (Iyer et al., 2016; Ahmad et al., 2020) treated code as a sequence of texts, but they did not take into account the complex interdependence of program elements in syntax or semantics. Structured-based approaches (Alon et al., 2019a; LeClair et al., 2019; Shi et al., 2021; Chai and Li, 2022) were later proposed to better capture the syntactic information. The state-of-the-art approaches, such as CAST (Shi et al., 2021) and PA-Former (Chai and Li, 2022), leverage the idea of *hierarchically* splitting the AST into smaller parts based on its structure. CAST hierarchically splits the AST's code blocks based on certain attributes, while PA-Former works by treating statements as spans and splitting them into (sub)-tokens. These code-hierarchy approaches bring the benefits in terms of effective and affordable training of neural models. However, a common drawback is that they ignore the program dependencies in code representations. There are other lines of work leveraging graphs (LeClair et al., 2020; Fernandes et al., 2019; Hellendoorn et al., 2020a) that model program dependencies by adding edges to the AST, in which the edges are the dependencies derived from static analysis. However, these approaches do not take into account the code hierarchy as the previous line of work.

We propose a novel approach called *Heterogeneous Code Representation* (HCR) to overcome these limitations by combining the strengths of both methodologies. HCR excels in encapsulating crucial code attributes across lexical, syntactic, and semantic dimensions within a hierarchical structure. This structure organizes program elements based on their features: sequences for code tokens, AST subtrees for syntax, and graphs for dependencies. Significantly, we adeptly capture program depen-

[*]Equal contribution. Listing order is based on the alphabetical ordering of author surnames.

[†]Emails: minhnh46@fpt.com.vn, dqnbui.2016@smu.edu.sg, truongson.hy@indstate.edu, long.tran-thanh@warwick.ac.uk, tien.n.nguyen@utdallas.edu

Findings of the Association for Computational Linguistics: EACL 2024, pages 2355–2367

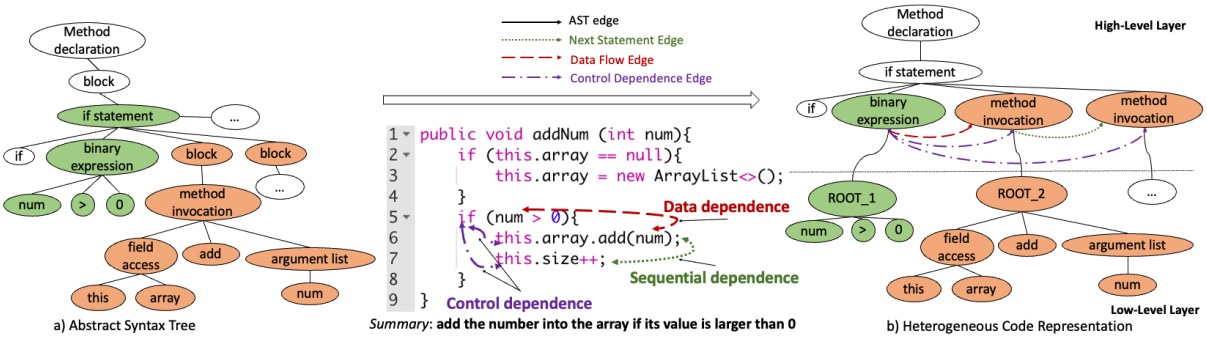

Figure 1: Motivating Example on Heterogeneous Code Representation

dencies by abstracting coarse-grained nodes into a higher-level layer and fine-grained nodes into a lower-level layer. This strategy enhances the generation of summaries as our model gains a more comprehensive understanding of the source code.

To process our representations, we introduce a heterogeneous architecture, called HIERARCHYNET, which comprises a Transformer Encoder for processing lexical information, a Tree-based Encoder for processing syntactic information, and a Graph-based Encoder for capturing program dependencies. These layers do not operate individually but hierarchically, which intuitively captures the relationships between program elements even better. Our comprehensive evaluation across diverse scenarios shows the effectiveness of our model in code summarization compared to state-of-the-art (SOTA) methods. Our model surpasses these methods in three distinct settings: (1) hierarchical neural networks (NNs) of code akin to us, such as PA-former and CAST; (2) fine-tuned SOTA pretrained language models of code, such as CodeT5 and CodeBERT; and (3) in-context learning of Large Code Language Models using zero-shot and few-shot settings, such as StarCoder and CodeGen.

To summarize, our key contributions include:

(1) *Heterogeneous Code Representation*: a novel code representation that incorporates sequences, trees, and graphs to effectively capture the lexical, syntactic, and semantic aspects of source code.

(2) HIERARCHYNET: a novel *hierarchical neural network architecture*, designed in a modular manner, where each module in the architecture is responsible for processing each layer in the *Heterogeneous Code Representation*. The key modules include the Transformer Encoder, Tree-based CNN, and Heterogeneous Graph Transformer, as well as a novel Hierarchy-Aware Cross Attention module for attending to information across layers.

(3) In our comprehensive evaluation on various established datasets for code summarization, including TL-CodeSum (Hu et al., 2018), DeepCom (Hu et al., 2020a), and FunCom (LeClair et al., 2019), HIERARCHYNET shows significantly superior performance compared to the baselines. In a variety of settings, HIERARCHYNET outperforms a wide range of models: (1) similar hierarchical NNs of code, such as PA-former and CAST; (2) fine-tuned SOTAs that are pretrained language models of code, such as CodeT5 and CodeBERT; and (3) in-context learning of Large Code Language Models using zero-shot and few-shot settings, such as CodeLlama, StarCoder, and CodeGen.

(4) We make our source code and implementation easy to reproduce via an anonymous link, allowing for future improvements for the research community: `https://github.com/FSoft-AI4Code/HierarchyNet`.

2 Related Work

Code Summarization Research in generating the descriptions for source code has evolved through various techniques. Initially, sequence-based methods treated code as text (Iyer et al., 2016; Ahmad et al., 2020; Wei et al., 2019), disregarding syntactic or semantic dependencies among program elements. For example, NeuralCodeSum (Ahmad et al., 2020) is a purely transformer-based approach that receives code tokens and generates summaries. Structure-based and tree-based approaches were also proposed to capture the syntax of source code (Tai et al., 2015; Mou et al., 2016a; Bui et al., 2021b; LeClair et al., 2019; Hu et al., 2020a; Peng et al., 2021b; Shi et al., 2021; Chai and Li, 2022). For instance, TreeLSTM (Tai et al., 2015) employs bottom-up node accumulation, while TPTrans (Peng et al., 2021b) integrates AST path information into the transformer.

CAST (Shi et al., 2021) and PA-former (Chai and Li, 2022) are currently the state-of-the-art methods with the same key idea of breaking the code into a structural hierarchy. Finally, graph-based techniques were used to capture code semantics by adding inductive bias into the AST through semantic edges, turning it into a graph (LeClair et al., 2020; Fernandes et al., 2019; Hellendoorn et al., 2020a). However, they still encounter challenges in representing code hierarchy and program dependencies, as well as neural networks to handle them.

Pretrained Language Models for Source Code Besides code summarization, language models of code generally support various code understanding tasks, such as code generation (Feng et al., 2020a; Wang et al., 2021b; Elnaggar et al., 2021), code completion (Feng et al., 2020a; Wang et al., 2021b; Peng et al., 2021a), program repair (Xia et al., 2022), etc. A large body of recent work employs language models from natural language processing for code (Feng et al., 2020a; Wang et al., 2021b; Guo et al., 2020; Ahmad et al., 2021; Bui et al., 2021a; Elnaggar et al., 2021; Peng et al., 2021a; Kanade et al., 2020; Chakraborty et al., 2022; Ahmed and Devanbu, 2022; Niu et al., 2022), applying similar pretraining strategies as used for natural languages. Despite their promising performance, these pretrained models have not been empirically demonstrated to effectively capture semantics in source code, such as data, control flows, and other program dependencies among code elements. In contrast, incorporating code-specific features into representations as inductive biases has been shown to increase the model's knowledge (Allamanis et al., 2018a; Hellendoorn et al., 2020b).

3 Motivation

Let us use an example to motivate and illustrate the key ideas of our solution. Figure 1 shows a code snippet and its corresponding summarization. The task is to collect the positive numbers into an array. To generate an accurate summary, a model needs to capture code features at the lexical, syntactic, and semantic levels. For example, at the lexical level, the sub-tokens `add`, `num`, and `array` should resemble words in the summary. The tokens `'>'` and `'0'` correspond to the texts `'larger than'` and `'0'` in the summary. At the syntactic level, the model should recognize code structures, such as the `if` statement at line 5 indicating a conditional sentence in the summary.

Importantly, the control and data dependencies among the statements could provide valuable insights into the intended execution order. Ignoring control dependencies hinders the model's ability to capture such intentions because the sequential order in the code may not reflect the execution order. For example, despite their sequential order, the execution of the statement at line 6 is not guaranteed to follow the statement at line 5, as it is dependent on the outcome of the *if condition* at line 5. Moreover, the data dependency via the variable `num` at line 5 and line 6 is also crucial for summarization as it indicates that only positive numbers are collected.

Previous approaches, such as those outlined in LeClair et al. (2020), Fernandes et al. (2019), and Hellendoorn et al. (2020a), utilize heuristics from static analysis to connect nodes in the AST to represent dependencies. However, the large size of the AST can pose challenges for a model to effectively capture dependencies among distant nodes (Alon and Yahav, 2021). In contrast, state-of-the-art approaches such as CAST (Shi et al., 2021) and PA-Former (Chai and Li, 2022) create a hierarchy among code elements by splitting the AST into smaller parts. However, these methods do not maintain program dependencies among the elements.

We propose the *Heterogeneous Code Representation* (Figure 1b) to restructure code into **hierarchical layers**, abstracting meaningful entities such as statements or expressions into single nodes in a higher layer. Importantly, HCR also enables the representation of **dependencies**, including data, control, sequential, and syntactic dependencies. We introduce a heterogeneous neural network utilizing an appropriate neural network at each level: a transformer encoder for code tokens, a tree-based encoder for AST subtrees, and a graph neural network for the coarse-grained dependencies. This approach reduces the computational workload and improves capturing the dependencies between distant nodes in an AST (an issue with the prior works).

4 Heterogeneous Code Representation

This section presents the Heterogeneous Code Representation (HCR) that integrates both hierarchical structure of source code as well as program dependencies among program elements. Figure 2 (left-side) displays the three layers of Heterogeneous Code Representation (HCR). The first layer, denoted by the "Linearized AST Sequence," is a sequence of nodes L from the serialization process

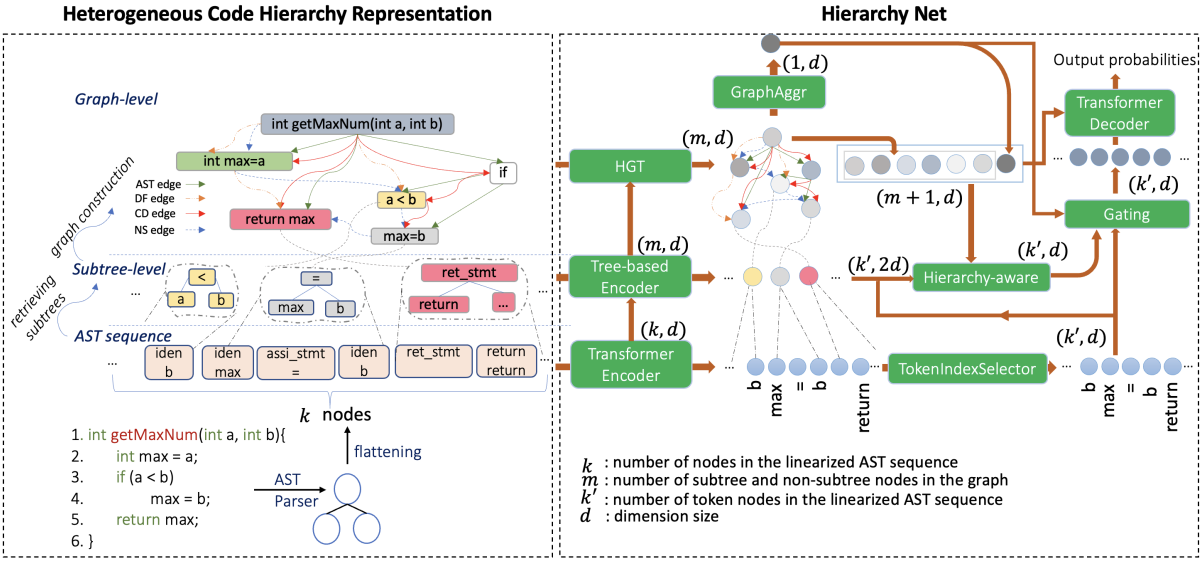

Figure 2: HIERARCHYNET Architecture

of the Abstract Syntax Tree (AST) of the given program. The second layer, the "Subtree-level," represents the statement and expression-level program elements, each represented by a significantly smaller subtree T' consisting of nodes from the original AST T. Finally, the last one is the highest-level and coarsest-grained layer, the "Graph level", which is represented by a graph G consisting of nodes from T', enriched by semantic edges such as control and data dependencies. Such dependencies are built from static program analysis. Next, we will present in details each layer in our model.

4.1 Serialized AST Sequence

We begin by parsing a program into an Abstract Syntax Tree (AST) T. Each token node contains a non-empty token, which is often made up of multiple sub-tokens. To incorporate these sub-tokens, we insert new sub-token nodes as children of the corresponding token node. The AST is then serialized to create a sequence of nodes L. Specifically, we convert the AST into a sequence of nodes by a traversal such that the original token order is maintained (Figure 2). Formally, the linearized AST sequence $L = [l_1, l_2, ..., l_k]$ (where k is the size of T) represents the lowest level of HCRs.

4.2 Syntactic Level

A function is usually a combination of many statements and expressions, each of which often represents a sufficient amount of information to understand how/what it does. We extract the AST subtrees corresponding to statements and expressions. These subtrees are then abstracted by replacing them with placeholder nodes in T, resulting in a smaller tree T'. This process is done through a depth-first traversal of the AST, where subtrees are replaced and further traversal is halted at the subtree's root node. This results in a new tree T' and a set of subtrees ST, with some nodes in T' pointing to elements in ST, which forms the second level in our HCRs. Note: some nodes in L do not belong to any subtrees (non-subtree nodes).

4.3 Semantic Level

We use the reduced AST T' and incorporate semantic edges among the nodes to create graph G (as depicted in Figure 2). Our graph includes four distinct edge types: AST edges, Data-flow (DF) edges, Control-dependence (CD) edges, and Next-subtree (NS) edges. These edges represent various forms of connections between program elements, such as code structures, data and control dependencies, and sibling statements in the source code.

5 Neural Network Architecture

This section explains the neural network architecture for our HIERARCHYNET method (Figure 2). Each node l_i in a sequence of nodes L has two attributes: *token* and *type*. The initial representation of each node l_i is computed by concatenating the embeddings of its *token* and its *type*. These embeddings can be looked up from two learnable embedding matrices (token and type). We denote s_i be the initial embedding of the node l_i, $i \in \mathbb{N}, 0 < i \leq k$ where k is the length of L.

The neural network architecture, HIERARCHYNET, consists of the following components.

5.1 Transformer Encoder

The Transformer Encoder encodes the linearized AST sequence L to capture lexical values. It takes initial embeddings $[s_1, s_2, ..., s_k]$ as input and produces the output $[h_1, h_2, ..., h_k]$.

5.2 Tree-based Encoder

This layer's primary role is to process the subtrees in the Subtree layer. Additionally, it also embeds non-subtree nodes in the L by applying a non-linear transformation. To model local patterns and hierarchical relations among nodes within the same subtree, all subtrees are passed through a Tree-based CNN (Mou et al., 2016b). An attention aggregation method (Alon et al., 2019b) is then employed to encode each subtree as an embedding vector, using a global attention vector α. The output of this layer are denoted as $\{\hat{t}_i\}_{i=1}^{m}$ where m is the number of subtrees and non-subtree nodes.

5.3 Heterogeneous Graph Transformer

After obtaining the embeddings of all the subtrees, we further encode the dependencies among the nodes in the heterogeneous graph G. We adapt the Heterogeneous Graph Transformer (HGT) (Hu et al., 2020b) to process the graph effectively. The outputs are the vectors $\{n_i\}_{i=1}^{m}$ that not only bring textual information (by Transformer Encoder and next-subtree edges) but also are contextualized by the locally hierarchical structures of the subtrees and dependence information that are unique characteristics in source code.

5.4 Graph Aggregation (GraphAggr)

Upon completion of the HGT processing, it is essential to aggregate the individual nodes within the graph into a vector that represents the graph. Similar to the tree aggregation technique employed in the Tree-based Encoder, an attention mechanism is utilized to aggregate the nodes and generate a graph embedding, denoted as g, by using the global attention vector β. This graph embedding g encapsulates the overall semantic meaning of the code.

5.5 Token Index Selector

The TokenIndexSelector layer utilizes the output of the Transformer Encoder as input and serves to retain the embeddings of nodes l_i that possess non-empty token attributes while discarding those that do not. The rationale is that the Transformer Encoder effectively encodes textual meaning but is inadequate in encoding syntax (as represented by the type attribute), which could potentially introduce noise to subsequent layers (such as the Gating Layer and Transformer Decoder). It is worth noting that the Subtree layer effectively encodes syntax information using Tree-based CNN. Formally, let H' be the sequence of the elements h_i such that l_i is a token node, for all $0 < i \leq k$. We denote the members of H' by $h'_1, h'_2, \ldots, h'_{k'}$ where k' is the number of token nodes in the L.

5.6 Hierarchy-Aware Cross Attention

Although information is gathered in a bottom-up manner, there may still be missing connections between layers. To address this issue, we introduce the Hierarchy-aware Cross Attention (HACA) layer, which enables the TokenIndexSelector layer to focus on the information from the HGT layer. This layer, depicted in Figure 2, calculates the attention of each token toward nodes in the structure (tree + graph). Keys K and values V are derived from the combination of the nodes' embeddings $\{n_i\}_{i=1}^{m}$ and the graph embedding g. Additionally, a token can occur multiple times in a code snippet, even with the use of positional encoding, the vectors of these tokens may be similar. To differentiate these occurrences, we concatenate their corresponding subtrees. For example, by examining the subtrees, we can discern the different roles of the variable a at lines 1 and 2. We enhance the distinctions by concatenating h'_i and $\hat{t}_i$ to create the vector query q_i; formally, $q_i = f_{ca}([h'_i, \hat{t}_i])$ where f_{ca} is a projection from $\mathbb{R}^{2d}$ to $\mathbb{R}^{d}$. Then the cross-attention is computed as usual, that is $\text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$ where d_k is the inner dimension size of each attention layer. This layer produces $\{c_i\}_{i=1}^{k'}$ where c_i is the fused hierarchical context dedicated to the token node corresponding to h'_i, for all $0 < i \leq k'$.

5.7 Gating Layer

The HACA layer is responsible for calculating attention scores across different layers, but it does not perform information integration. We introduce the Gating layer to combine the information across different layers in the hierarchy, serving as the input for the Transformer Decoder. The goal is to combine the outputs $\{c_i\}_{i=1}^{k'}$ of HACA with the lex-

Model	TL-CodeSum dataset				FunCom dataset			
	BLEU	Meteor	Rouge-L	Cider	BLEU	Meteor	Rouge-L	Cider
Training from scratch								
HDeepCom	23.32	13.76	33.94	1.74	25.71	15.59	36.07	1.42
ASTAttGru	30.78	17.35	39.94	2.31	28.17	18.43	39.56	1.90
NCS	40.63	24.86	52.00	3.47	29.18	19.94	40.09	2.15
CodeAstnn	41.08	24.95	51.67	3.49	28.27	18.86	40.34	1.94
CAST	45.19	27.88	55.08	3.95	31.55	21.10	42.71	2.31
PA-former	46.01	28.05	56.12	4.04	31.94	20.88	42.73	2.29
Fine-tuning								
CodeBERT-base	39.84	23.64	48.54	3.28	31.87	21.19	42.99	2.30
CodeT5-base	47.02	30.01	57.68	4.13	32.75	21.40	43.20	2.41
In-context Learning								
CodeGen-Multi 2B (zero-shot)	7.51	3.42	2.86	0.05	12.52	8.64	14.55	0.23
CodeGen-Multi 2B (one-shot)	11.62	7.59	17.21	0.37	21.65	14.30	29.51	1.14
CodeGen-Multi 2B (two-shot)	11.70	7.76	17.68	0.39	23.19	15.59	32.43	1.32
StarCoder (zero-shot)	13.12	12.55	24.01	0.58	19.05	16.72	28.65	0.81
StarCoder (one-shot)	14.41	11.36	24.46	0.65	23.04	15.93	32.51	1.35
StarCoder (two-shot)	15.66	12.10	26.32	0.74	24.21	16.65	34.35	1.48
HIERARCHYNET	**48.01**	**30.30**	**57.90**	**4.20**	**33.43**	**21.70**	**43.42**	**2.42**

Table 1: Comparative Code Summarization Performance on TL-CodeSum and FunCom Datasets (RQ1).

ical information of $\{h_i'\}_{i=1}^{k'}$. To balance the two sources of information, we propose to add a sufficient amount of context from c_i to h_i'. We take inspiration from the gating layer in Cho et al. (2014), and modify it to achieve this goal. Specifically, the ratio between the two sources is controlled by the graph embedding, as g is the highest level of abstraction and contains a global understanding of the code. Formally, the computation can be summarized as: $\lambda = \text{sigmoid}(Wg + b)$, where $W \in \mathbb{R}^{d \times d}$, $b \in \mathbb{R}^d$ or $W \in \mathbb{R}^d$, $b \in \mathbb{R}$ are learnable parameters, and d is the dimension of the vector g. We then apply a non-linear projection f_c to map c_i onto the space of h_i' and form the hierarchy-aware hidden state by: $e_i = \lambda f_c(c_i) + (1 - \lambda)h_i'$. Finally, $\{e_i\}_{i=1}^{k'}$ are final encoder hidden states.

5.8 Transformer Decoder

Unlike the vanilla Transformer Decoder (Vaswani et al., 2017), we need to combine the two sources, including hierarchy-aware textual information (the output of Gating layer) and the structural/semantic meaning (the output of HGT and GraphAggr).

Therefore, in HIERARCHYNET, we leverage the serial strategy (Libovický et al., 2018) in computing the encoder-decoder attention one by one for each input encoder. The key and value sets of the first cross-attention come from the output of HGT and GraphAggr. Those sets of the other cross-attention are from the output of Gating layer.

6 Empirical Evaluation

We have conducted several experiments to evaluate HIERARCHYNET. We seek to answer the following research questions:

1. RQ1. [**Automated Evaluation**]. How well does HIERARCHYNET perform in code summarization compared with the SOTA approaches?

2. RQ2. [**Human Evaluation**]. How well does HIERARCHYNET perform in code summarization in a human study with human evaluation?

3. RQ3. [**Ablation Study**]. How well do different components in HIERARCHYNET contribute to its overall code summarization performance?

6.1 Automated Evaluation (RQ1)

Datasets. To ensure a comprehensive comparison with several SOTA baselines, we considered multiple well-established datasets for code summarization, namely TL-CodeSum (Hu et al., 2018), DeepCom (Hu et al., 2020a), FunCom (LeClair et al., 2019), and FunCom-50 (Chai and Li, 2022). Note that different baselines use distinct datasets and achieve SOTA results. The FunCom-50 dataset was used by PA-Former (Chai and Li, 2022), but with a number of samples filtered out from FunCom, approximately 50% of the data. We followed the original dataset's partition in FunCom (LeClair et al., 2019) for training, testing, and validation.

Model	DeepCom				FunCom-50			
	BLEU	Meteor	Rouge-L	F1	BLEU	Meteor	Rouge-L	F1
Training from scratch								
HDeepCom	32.18	21.53	49.03	50.75	35.06	22.65	53.35	54.81
SiT	35.69	24.20	53.75	55.72	42.12	26.82	59.33	60.84
GREAT	36.38	24.18	53.61	55.46	43.29	27.44	60.36	61.83
NCS	37.13	25.05	54.80	56.68	43.36	27.54	60.41	61.86
TPTrans	37.25	25.02	55.00	56.88	43.45	27.61	60.57	62.03
CAST	38.03	25.27	54.95	56.83	43.58	27.67	60.52	61.98
PA-former	39.67	26.21	56.18	58.12	44.65	28.27	61.45	62.86
Fine-tuning								
CodeBERT-base	37.42	25.49	55.07	56.93	46.20	30.51	61.43	63.77
CodeT5-base	38.60	26.30	56.31	58.42	46.88	30.72	61.47	63.88
In-context Learning								
CodeGen-Multi 2B (zero-shot)	11.20	4.85	4.73	5.04	13.38	4.03	2.88	3.00
CodeGen-Multi 2B (one-shot)	17.12	13.09	23.21	24.49	21.08	14.29	25.68	26.56
CodeGen-Multi 2B (two-shot)	17.81	13.81	24.62	26.04	21.78	14.78	26.89	27.84
StarCoder (zero-shot)	16.03	15.34	24.55	26.27	19.22	18.65	29.74	31.17
StarCoder (one-shot)	18.78	15.68	27.33	28.95	23.93	17.97	31.25	32.13
StarCoder (two-shot)	19.29	16.07	28.09	29.68	25.18	18.45	32.59	33.68
CodeLlama 13B (zero-shot)	13.28	12.88	19.17	21.00	14.79	5.19	21.40	21.67
CodeLlama 13B (one-shot)	17.05	15.70	28.23	30.33	19.20	16.57	27.96	30.03
CodeLlama 13B (two-shot)	20.29	16.14	39.63	42.01	21.52	16.52	36.49	32.40
HIERARCHYNET	**43.64**	**29.22**	**59.00**	**60.53**	**51.12**	**34.13**	**65.43**	**66.64**

Table 2: Comparative Code Summarization Performance on DeepCom and FunCom-50 Datasets (RQ1).

Baselines. We compared HIERARCHYNET against three categories of baselines. The first category includes the baselines trained from scratch without utilizing pretrained checkpoints. Examples include CAST (Shi et al., 2021) and PA-Former (Chai and Li, 2022), which are consciously designed to incorporate code structures. Additional baselines in this category, grouped by code representation and neural architecture, including *sequence-based models* (NCS (Ahmad et al., 2020)), *structure-based and tree-based models* (ASTAttGru (LeClair et al., 2019), HDeepCom (Hu et al., 2020a), TPTrans (Peng et al., 2021b), TreeLSTM (Tai et al., 2015), CodeASTNN (Shi et al., 2021), SiT (Hongqiu et al., 2021)), and *graph-based models* (GREAT (Hellendoorn et al., 2020a)).

The second category comprises fine-tuned baselines for code summarization from well-known pretrained models. For representative models, we fine-tune CodeT5-base (Wang et al., 2021a) and CodeBERT-base (Feng et al., 2020b), considering CodeT5 as the state-of-the-art for code summarization and CodeBERT as a widely-used model.

The third category encompasses large language models that can perform in-context learning for code understanding tasks using zero-shot, one-shot, or few-shot learning approaches. For this category, we used CodeLlama 13B (Roziere et al., 2023), StarCoder (Li et al., 2023) and CodeGen-Multi 2B (Nijkamp et al., 2023).

Metrics. We employ BLEU (Papineni et al., 2002), Meteor (Banerjee and Lavie, 2005), Rouge-L (Lin, 2004), Cider (Vedantam et al., 2015) and F1-score, which are commonly used as the evaluation metrics for code summarization.

Results. The results shown in Table 1 and 2 indicate that HIERARCHYNET exhibits superior performance compared to the CAST and PA-former methods by a significant margin on the four datasets. Specifically, HIERARCHYNET achieves an average improvement of 4.46 and 3.48 BLEU scores over CAST and PA-former, respectively. Notably, PA-Former, which is currently considered the state-of-the-art baseline, only outperforms CAST by an average of 1 BLEU score. Furthermore, HIERARCHYNET also consistently surpasses CodeT5-base and CodeBERT-base and outperforms Large Language Models for code such as CodeLlama, StarCoder and CodeGen-Multi 2B in all three prompting scenarios (zero/one/two-shot) on the datasets.

In conclusion, the results show that HIERARCHYNET, which utilizes a hierarchical-based architecture and dependencies information, significantly improves code summarization performance.

ID	Tokens	Subtrees	Graph				BLEU	Meteor	Rouge-L	Cider
			AST edges	NS edges	CD edges	DF edges				
1	✓	-	-	-	-	-	40.63	24.86	52.00	3.47
2	✓	✓	-	-	-	-	44.16	28.19	55.48	3.77
3	✓	✓	✓	-	-	-	45.37	28.43	55.72	3.91
4	✓	✓	✓	-	✓	-	46.54	29.39	56.70	4.04
5	✓	✓	✓	-	-	✓	46.61	29.41	56.64	4.03
6	✓	✓	✓	-	✓	✓	47.46	30.15	57.63	4.14
7	✓	✓	✓	✓	-	-	45.44	28.24	54.72	3.89
8	✓	✓	✓	✓	✓	-	46.84	29.40	56.88	4.05
9	✓	✓	✓	✓	-	✓	47.26	30.10	57.64	4.12
10	✓	✓	✓	✓	✓	✓	**48.01**	**30.30**	**57.90**	**4.20**

Table 3: Results of Ablation Study on Heterogeneous Code Representation (RQ3)

6.2 Human Evaluation (RQ2)

In line with prior work on code summarization (Iyer et al., 2016; Shi et al., 2021; Chai and Li, 2022), we conducted a user study with the participation of five software development experts to examine the efficacy of our method in practice. We presented each participant with 100 random examples from the testing segment of the FunCom dataset, along with three respective summaries produced by HIERARCHYNET, PA-former, and CAST. In order to avoid potential biases, we do not provide the ground truth, and summaries of different methods are randomly tagged with placeholder names. Following Shi et al. (2021); Chai and Li (2022), we adopt two human evaluation criteria: 1) *naturalness:* grammar, fluency, and readability of generated summaries. 2) *usefulness:* to what extent generated summaries are useful to comprehend the code. Each aspect is divided into three standards rating from 1 to 3, with higher scores indicating better performance. The final score for each criterion is the average of all samples. As shown in Table 4, HIERARCHYNET outperforms both CAST and PA-former in terms of naturalness and usefulness.

Methods	Naturalness	Usefulness
CAST	2.76	2.48
PA-former	2.77	2.50
HIERARCHYNET	**2.81**	**2.52**

Table 4: Results of User Study (RQ2)

7 Model Analysis (RQ3)

7.1 Study on Heterogeneous Code Representation (HCR)

We investigated the influence of the HCR components on code summarization performance using the TL-CodeSum dataset, as shown in Table 3. Starting with only the AST-sequence layer resulted in suboptimal performance. Incorporating Subtree and Graph layers incrementally improved results. Our AST-edge-focused experiment at the Graph level surpassed CAST's performance (Table 1), suggesting our hierarchy's superiority. While the CD and DF edges notably impact performance, NS edges are less crucial. Still, excluding any edges reduces performance, indicating that the dependencies positively contributed to the performance.

7.2 Study on HierarchyNet

Method	BLEU	Meteor	Rouge-L	Cider
HIERARCHYNET	**48.01**	**30.30**	**57.90**	**4.20**
w/o Hierarchy-aware	46.63	29.49	56.63	4.03
w/o TokenIndexSelector	45.70	28.39	55.06	3.93

Table 5: Ablation Study of HIERARCHYNET (RQ3)

Decoding strategy	BLEU	Meteor	Rouge-L	Cider
serial decoding	**48.01**	**30.30**	**57.90**	**4.20**
only Gating layer's output	45.34	28.28	55.33	3.89
concat	47.22	29.41	56.45	4.10

Table 6: Ablation Study on Decoding Strategy (RQ3)

In addition, we aim to demonstrate the significance of our proposed layers in Hierarchy Net, including Hierarchy-Aware Cross-Attention (abbreviated as Hierarchy-Aware) and TokenIndexSelector, on the TL-CodeSum dataset. The result (Table 5) shows that the removal of any of these com-

ponents significantly degrades performance. This confirms that the Transformer architecture alone is not sufficient to encode both textual and structural/semantic meanings of code, thus highlighting the importance of explicitly integrating semantic and structural information using Hierarchy-Aware Attention. Additionally, we found that removing TokenIndexSelector has a negative impact on performance, which is likely due to the redundant information in the sequence fed to the Decoder.

To show the effectiveness of the serial decoding with the two consecutive cross attention in the Decoder, we compare to two alternatives that just use a cross attention in the Decoder. Specifically, the first option calls for utilizing the Gating layer's output. The other way is concatenating the TokenIndexSelector's output, HGT's output and GraphAggr's output into single extended sequences, which are then fed to the Decoder. As shown in Table 6, more information employed in the Decoder in the latter strategy leads to the better performance compared to only Gating layer's output. However, combining our proposed code hierarchy representation with the serial decoding achieves the highest results.

7.3 Comparison with LLMs

Model	Average word count
StarCoder (zero-shot)	10.64
StarCoder (one-shot)	7.59
StarCoder (two-shot)	8.12
CodeGen 2B (zero-shot)	4.95
CodeGen 2B (one-shot)	8.46
CodeGen 2B (two-shot)	8.49
References	9.97

Table 7: Comparative Results with LLMs regarding the Average Word Count of Summaries

Given that LLMs may potentially generate responses longer and more detailed than the ground truth, our objective is to thoroughly analyze and ensure the fairness of our evaluation. We present the average word count of summaries generated by LLMs compared to references on DeepCom in Table 7. Notably, LLMs like StarCoder and CodeGen 2B tend to produce shorter summaries than the ground truth. Although, in the zero-shot setting, StarCoder can generate slightly longer summaries, this difference is negligible. As a result, summaries generated by LLMs are considered to be of similar length to the references in the ground truth.

Moreover, the experimental results reveal a substantial performance disparity between our proposed method and large language models across all metrics. Specifically, in terms of Rouge-L, the gaps amount to approximately 10, 30, and 30 when compared to StarCoder on FunCom, DeepCom, and FunCom-50, respectively. Regarding Meteor, these are 5, 13, and 15, respectively. The study (Roy et al., 2021) shows that there is a statistically significant difference in performance between models whose performance difference is greater than 10 points. Furthermore, it finds that for the gaps exceeding 10 points, the metrics, like Rouge-L and Meteor, strongly agree with human assessment.

8 Conclusion

We introduce an innovative framework for code summarization that combines Heterogeneous Code Representations (HCRs) with HIERARCHYNET, a neural architecture tailored for processing HCRs. Our HCRs capture critical code attributes across lexical, syntactic, and semantic levels by organizing coarse-grained code elements into a higher-level layer while integrating fine-grained program elements into a lower-level layer. HIERARCHYNET is engineered to handle each layer of the HCR independently, enabling the representation of information gathered at the fine-grained level as input at the coarse-grained level. The core concept of HIERARCHYNET lies in integrating multi-level code representations and program dependencies. Our empirical evaluations demonstrate that our approach surpasses various state-of-the-art techniques across diverse settings, including structure-based models (CAST, PA-Former), fine-tuned pretrained models (CodeT5, CodeBERT), and in-context learning (CodeLlama, StarCoder, CodeGen). Our ablation study shows that all of the components in HIERARCHYNET contribute positively to its high performance. We also conducted a human study to evaluate the code summarization results produced by HIERARCHYNET. The results show that human subjects highly regarded the code summarizaiton results from our model.

Acknowledgments

The co-author, Tien N. Nguyen, was supported in part by the US National Science Foundations grant CNS-2120386 and the National Security Agency grant NCAE-C-002-2021.

Limitations

Our approach presents opportunities for improvement.

1. First, our Heterogeneous Code Representations (HCRs) with coarse-grained semantic edges have proven effective for code summarization. However, there may be potential for further enhancement by exploring alternative options for cross-layer semantic edges, such as connecting nodes at the fine-grained level with nodes at the coarse-grained level. This could be beneficial for other code representation learning tasks, such as variable name prediction (Allamanis et al., 2018b) and data flow analysis using neural models (Gu et al., 2021). Our next step is to conduct further research on extending HCRs to include these alternative options and evaluate their performance on other code representation learning tasks.

2. Second, while HIERARCHYNET effectively processes the HCRs, there is room for further optimization. We chose the layers in the HIERARCHYNET based on heuristics, resulting in the HGT being the best option for processing graphs. At the subtree-level, we chose the TBCNN as it is more computationally efficient compared to other state-of-the-art methods for processing ASTs, such as TreeCaps (Bui et al., 2021b). However, our approach can be considered a framework rather than a single neural model, so other advancements in tree- or graph-based models or sequence-based models can easily be incorporated to improve performance.

3. Finally, we did not provide any analysis on the explainability of our model. Explainability is an important aspect of code learning models (Bui et al., 2019; Bielik and Vechev, 2020; Zhang et al., 2020; Rabin et al., 2021), and is crucial for the real-world usage of practitioners in code summarization. Our current model design has the potential to support explainability in the future, as the inputs of the high-level layer are computed based on the attention aggregation mechanism, with each input being assigned an attention score. These attention scores can be used to visualize and explain the importance of code elements in a hierarchical way. We will explore this extension as a future work.

Ethics Statement

Our framework aims to revolutionize the way software is modeled by taking a new approach with a broader impact in the field. While language models for code have shown impressive performance and have the potential to boost developer productivity, they still face challenges with computational cost and memory consumption. For example, when modeling code and software at the repository level, such as on Github, the AI framework must consider the context of the current code being edited, as well as additional contexts from other files or API calls from external libraries. This is a dependency on a larger scale level in the context of software modeling. Currently, language models typically only model code within the scope of a function or within a single file for tasks such as code summarization or generation. However, this limitation may not be due to the language model itself, but rather the infrastructure of supported IDEs and the software modeling approach. We propose a more realistic way to represent programs as "repository=>file=>class=>function=>statement=>token." The simplest way to model such a hierarchy is to treat them all as a very large sequence and use Large Language Models to model it, but this results in large memory consumption and expensive computational costs. A more affordable approach is to represent large software as modules, where each module can be represented differently at each level. Each layer may require dependency analysis or not, depending on its characteristics. For example, the semantic edges used to connect components clearly differ at each level, requiring careful design of them. Existing approaches to representing the entire program as a graph will fail in this case because the set of semantic edges are designed the same for all nodes without treating them differently. Also, each of the modules can be preprocessed separately on different computing units and aggregated later to achieve efficient computation cost and save memory. We have already demonstrated efficacy when modeling the program at three levels: "function=>statement=>token" and plan to extend this further. Our natural way of structuring the source code hierarchically is also aligned well with the advances in programming language and software engineering research in program representations. We believe our solution can be viewed as a framework and opens up a new research direction for representing software.

References

Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2020. A transformer-based approach for source code summarization. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 4998–5007, Online. Association for Computational Linguistics.

Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified Pre-training for Program Understanding and Generation. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2021, Online, June 6-11, 2021*, pages 2655–2668. Association for Computational Linguistics.

Toufique Ahmed and Premkumar Devanbu. 2022. Multilingual training for software engineering. In *Proceedings of the 44th International Conference on Software Engineering*, pages 1443–1455.

Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. 2018a. Learning to represent programs with graphs. In *International Conference on Learning Representations*.

Miltiadis Allamanis et al. 2018b. Learning to represent programs with graphs. In *International Conference on Learning Representations*.

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

Uri Alon and Eran Yahav. 2021. On the bottleneck of graph neural networks and its practical implications. In *International Conference on Learning Representations*.

Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019b. Code2vec: Learning distributed representations of code. *Proc. ACM Programming Languages*, 3(POPL):40:1–40:29.

Satanjeev Banerjee and Alon Lavie. 2005. METEOR: An automatic metric for MT evaluation with improved correlation with human judgments. In *Proceedings of the ACL Workshop on Intrinsic and Extrinsic Evaluation Measures for Machine Translation and/or Summarization*, pages 65–72, Ann Arbor, Michigan. Association for Computational Linguistics.

Pavol Bielik and Martin Vechev. 2020. Adversarial robustness for code. *arXiv preprint arXiv:2002.04694*.

Nghi DQ Bui, Yijun Yu, and Lingxiao Jiang. 2019. Autofocus: interpreting attention-based neural networks by code perturbation. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 38–41. IEEE.

Nghi DQ Bui, Yijun Yu, and Lingxiao Jiang. 2021a. Self-supervised contrastive learning for code retrieval and summarization via semantic-preserving transformations. In *Proceedings of the 44th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 511–521.

Nghi DQ Bui, Yijun Yu, and Lingxiao Jiang. 2021b. Treecaps: Tree-based capsule networks for source code processing. In *Proceedings of the 35th AAAI Conference on Artificial Intelligence*.

Lei Chai and Ming Li. 2022. Pyramid attention for source code summarization. In *Advances in Neural Information Processing Systems*.

Saikat Chakraborty, Toufique Ahmed, Yangruibo Ding, Premkumar T Devanbu, and Baishakhi Ray. 2022. Natgen: generative pre-training by "naturalizing" source code. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 18–30.

Kyunghyun Cho, Bart van Merriënboer, Dzmitry Bahdanau, and Yoshua Bengio. 2014. On the properties of neural machine translation: Encoder–decoder approaches. In *Proceedings of SSST-8, Eighth Workshop on Syntax, Semantics and Structure in Statistical Translation*, pages 103–111, Doha, Qatar. Association for Computational Linguistics.

Ahmed Elnaggar, Wei Ding, Llion Jones, Tom Gibbs, Tamas Feher, Christoph Angerer, Silvia Severini, Florian Matthes, and Burkhard Rost. 2021. Codetrans: Towards cracking the language of silicon's code through self-supervised deep learning and high performance computing. *arXiv preprint arXiv:2104.02443*.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020a. Codebert: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020, Online Event, 16-20 November 2020*, volume EMNLP 2020 of *Findings of ACL*, pages 1536–1547. Association for Computational Linguistics.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020b. 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.

Patrick Fernandes, Miltiadis Allamanis, and Marc Brockschmidt. 2019. Structured neural summarization. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net.

Jiazhen Gu, Huanlin Xu, Haochuan Lu, Yangfan Zhou, and Xin Wang. 2021. Detecting deep neural network defects with data flow analysis. In *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*, pages 188–195. IEEE.

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

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

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

Wu Hongqiu, Zhao Hai, and Zhang Min. 2021. Code summarization with structure-induced transformer. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics (ACL)*.

Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2018. Deep code comment generation. In *Proceedings of the 26th Conference on Program Comprehension*, ICPC '18, page 200–210, New York, NY, USA. Association for Computing Machinery.

Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2020a. Deep code comment generation with hybrid lexical and syntactical information. *Empirical Software Engineering*, 25(3):2179–2217.

Ziniu Hu, Yuxiao Dong, Kuansan Wang, and Yizhou Sun. 2020b. Heterogeneous Graph Transformer. In *WWW '20: The Web Conference 2020, Taipei, Taiwan, April 20-24, 2020*, pages 2704–2710. ACM / IW3C2.

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 (Volume 1: Long Papers)*, pages 2073–2083, Berlin, Germany. Association for Computational Linguistics.

Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. 2020. Learning and evaluating contextual embedding of source code. In *International Conference on Machine Learning*, pages 5110–5121. PMLR.

Alexander LeClair, Sakib Haque, Lingfei Wu, and Collin McMillan. 2020. Improved code summarization via a graph neural network. In *Proceedings of the 28th international conference on program comprehension*, pages 184–195.

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 '19, page 795–806. IEEE Press.

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

Jindrich Libovický, Jindrich Helcl, and David Marecek. 2018. Input combination strategies for multi-source transformer decoder. In *Proceedings of the Third Conference on Machine Translation: Research Papers, WMT 2018, Belgium, Brussels, October 31 - November 1, 2018*, pages 253–260. Association for Computational Linguistics.

Chin-Yew Lin. 2004. ROUGE: A package for automatic evaluation of summaries. In *Text Summarization Branches Out*, pages 74–81, Barcelona, Spain. Association for Computational Linguistics.

Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. 2016a. Convolutional neural networks over tree structures for programming language processing. In *AAAI Conference on Artificial Intelligence*.

Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. 2016b. Convolutional neural networks over tree structures for programming language processing. In *AAAI Conference on Artificial Intelligence*.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2023. Codegen: An open large language model for code with multi-turn program synthesis. In *The Eleventh International Conference on Learning Representations*.

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. In *Proceedings of the 44th International Conference on Software Engineering*, pages 2006–2018.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: A method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*, ACL '02, page 311–318, USA. Association for Computational Linguistics.

Dinglan Peng, Shuxin Zheng, Yatao Li, Guolin Ke, Di He, and Tie-Yan Liu. 2021a. How could neural networks understand programs? In *International Conference on Machine Learning*, pages 8476–8486. PMLR.

Han Peng, Ge Li, Wenhan Wang, Yunfei Zhao, and Zhi Jin. 2021b. Integrating tree path in transformer for code representation. In *Advances in Neural Information Processing Systems*.

Md Rafiqul Islam Rabin, Nghi DQ Bui, Ke Wang, Yijun Yu, Lingxiao Jiang, and Mohammad Amin Alipour. 2021. On the generalizability of neural program models with respect to semantic-preserving program transformations. *Information and Software Technology*, 135:106552.

Devjeet Roy, Sarah Fakhoury, and Venera Arnaoudova. 2021. Reassessing automatic evaluation metrics for code summarization tasks. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2021, page 1105–1116, New York, NY, USA. Association for Computing Machinery.

Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*.

Ensheng Shi, Yanlin Wang, Lun Du, Hongyu Zhang, Shi Han, Dongmei Zhang, and Hongbin Sun. 2021. Cast: Enhancing code summarization with hierarchical splitting and reconstruction of abstract syntax trees. In *Conference on Empirical Methods in Natural Language Processing*.

Kai Sheng Tai, Richard Socher, and Christopher D. Manning. 2015. Improved semantic representations from tree-structured long short-term memory networks. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing of the Asian Federation of Natural Language Processing, ACL 2015, July 26-31, 2015, Beijing, China, Volume 1: Long Papers*, pages 1556–1566. The Association for Computer Linguistics.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc.

Ramakrishna Vedantam, C Lawrence Zitnick, and Devi Parikh. 2015. Cider: Consensus-based image description evaluation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4566–4575.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven C.H. Hoi. 2021a. CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 8696–8708, Online and Punta Cana, Dominican Republic. Association for Computational Linguistics.

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

Bolin Wei, Ge Li, Xin Xia, Zhiyi Fu, and Zhi Jin. 2019. Code generation as a dual task of code summarization. *Advances in neural information processing systems*, 32.

Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. 2022. Practical program repair in the era of large pre-trained language models. *arXiv preprint arXiv:2210.14179*.

Huangzhao Zhang, Zhuo Li, Ge Li, Lei Ma, Yang Liu, and Zhi Jin. 2020. Generating adversarial examples for holding robustness of source code processing models. In *34th AAAI Conference on Artificial Intelligence*.