Disentangled Code Representation Learning for Multiple Programming Languages

Jingfeng Zhang[1] **Haiwen Hong**[1] **Yin Zhang**[1*] **Yao Wan**[2] **Ye Liu**[3] **Yulei Sui**[4]
[1] College of Computer Science and Technology, Zhejiang University, Hangzhou, China
[2]School of Computer Sci. & Tech., Huazhong University of Science and Technology, China
[3] Department of Computer Science, University of Illinois at Chicago, IL, USA
[4] School of Computer Science, University of Technology Sydney, Australia
{zhjf, honghaiwen, zhangyin98}@zju.edu.cn, wanyao@hust.edu.cn,
yliu279@uic.edu, yulei.sui@uts.edu.au

Abstract

Developing effective distributed representations of source code is fundamental yet challenging for many software engineering tasks such as code clone detection, code search, code translation and transformation. However, current code embedding approaches that represent the semantic and syntax of code in a mixed way are less interpretable and the resulting embedding can not be easily generalized across programming languages. In this paper, we propose a disentangled code representation learning approach to separate the semantic from the syntax of source code under a multi-programming-language setting, obtaining better interpretability and generalizability. Specially, we design three losses dedicated to the characteristics of source code to enforce the disentanglement effectively. We conduct comprehensive experiments on a real-world dataset composed of programming exercises implemented by multiple solutions that are semantically identical but grammatically distinguished. The experimental results validate the superiority of our proposed disentangled code representation, compared to several baselines, across three types of downstream tasks, i.e., code clone detection, code translation, and code-to-code search.

1 Introduction

Code representation learning has become an essential technique to support various software engineering tasks. Most of previous code representation learning approaches (Chen and Zhou, 2018; Jain et al., 2020; Nie et al., 2020) focus on a particular programming language, while learning code representations for multiple programming languages, though challenging, is an important step towards more generalizable and interpretable code embeddings. In principle, code snippets can be seen as code token sequences where their structural information often manifests as tree or graph data structures like AST (Abstract Syntax Tree). The downstream tasks often take full advantage of different code modalities (DQ et al., 2019) (e.g., structural information and textual tokens in the form of natural languages) to achieve better performance.

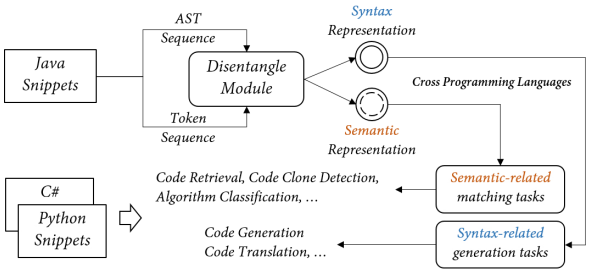

Figure 1: We disentangle the code representation into semantic and syntactic parts. The semantic part, which is relevant to code functionality but is often independent of a specific language, can be reused for semantic-related tasks across programming languages. The syntax part, which is related to a particular language but does not represent the code functionality, can be reused to control syntactic transformations for cross-programming language generation tasks.

It is noteworthy that syntax-level noise is an important issue in cross-language semantic-related tasks. Simply mixing textual token information and structural information of code (e.g., ASTs) often can not boost the performance on the cross-language code tasks. In this paper, we investigate a new approach that disassociates the latent semantic and syntactic representations of multi-lingual code snippets. The semantic representation excluding syntax information is more suitable for cross-language semantic-related code tasks as shown in Figure 1. We therefore study a new multi-lingual AST-guided code disentanglement technique called CODEDISEN, in order to provide a disentangled representation of code snippets that separates the la-

*Corresponding author: Yin Zhang

Findings of the Association for Computational Linguistics: ACL-IJCNLP 2021, pages 4454–4466
August 1–6, 2021.

tent syntax representations for masked ASTs from its latent semantic counterpart for solving a particular programming exercise. Our new multi-lingual code representation disentanglement approach effectively utilises available linguistic resources, i.e., ASTs and textual code tokens. Overall, the main contributions of this paper are as follows:

• To the best of our knowledge, it is the first time that we formulate the code representation learning problem from the perspective of disentangling code semantics and syntax information across multiple programming languages.

• We propose an AST-guided disentangled code representation learning approach for multiple programming languages. We employ masked AST information to guide the disentanglement of code semantics and syntax, and design a cross-language reconstruction loss and a posterior distribution loss for modeling the fact that programs written in different languages for the same problem can share the similar program semantic. Furthermore, attentive code position loss can effectively fuse AST information into an effective code representation.

• To validate the effectiveness of our approach, we have conducted extensive experiments on three downstream tasks (i.e., code-to-code search, code translation and clone detection). Experimental results show that the latent semantic and syntax representation learned by our approach are nearly orthogonal, and the learnt disentangled semantic representation can significantly boost the performance of the downstream cross-language tasks.

2 Preliminaries

2.1 Code Syntax and Semantics

The Abstract Syntax Tree (AST) is an abstract representation of the syntax structure of source code. As shown in Figure 2, compilation nodes, e.g. `augment_list`, represent syntactic information, and leaf variable nodes, e.g. `range`, represent semantic information. In this paper, we parse the source codes into ASTs by using tree-sitter[1], an open source syntax parser, which supports multiple programming languages. We traverse the nodes of an AST based on the depth first algorithm, and consider the traversed paths as syntax representation of the code snippets. Using AST paths can significantly reduce the learning effort to extract grammatical information of code. *To restrict the AST paths to syntax information only, we masked leaf nodes during the traversal because the semantic information of the code snippets often comes from the leaf variable nodes.* Introducing masked AST paths to CODEDISEN ensures that our approach can take some semantic meanings from the textual tokens in code snippets rather than ASTs, which can be used to learn the general syntax representation adhering to a specific language.

[1] https://tree-sitter.github.io

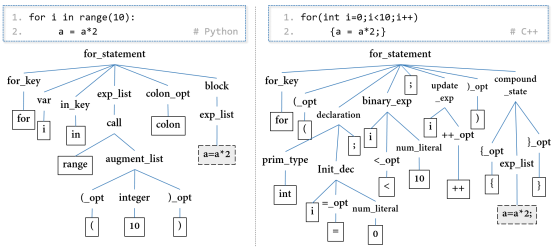

Figure 2: Python and C++ code snippets with their ASTs for the same problem. The solid boxes represent the leaf nodes. Note that the compilation nodes for `a=a*2` are almost identical.

2.2 Problem Statement

We denote code snippets for solving the same programming problem j as $\{\langle x_1, \ldots, x_n\rangle | x_i \in P_j\}$, where x_i is the solution of programming language i. In our experiments, we tested Java, Python, C++ and C#, thus the number of languages is $n = 4$. For each code snippet x_i, we construct a raw representation vector $\langle x_i, x_i^{ast}\rangle$, where x_i denotes a sequence of tokens, and x_i^{ast} represents syntax information derived from the abstract syntax tree of code snippet x_i.

For the same problem j, the code snippets $x_1, \ldots, x_n$ of multiple languages share the same semantic, although they have different programming language syntax z_i. Variants of Variational AutoEncoders (VAE) have been proposed to encode the raw vector $\langle x_i, x_i^{ast}\rangle$ into the latent representation. We aim to disentangle latent representation into two untangled parts: semantic y and syntax z latent representation of code. Formally, the objective of encoding is $\langle x_i, x_i^{ast}\rangle \rightarrow \langle y_i, z_i\rangle$ for each code snippet. For that purpose, we have to add multiple additional losses to the VAE architecture to enforce the effective disentanglement of code semantic and syntax. Next, we will introduce the design of multiple additional losses to effectively enforce disentanglement for code representation learning under the multi-lingual setting.

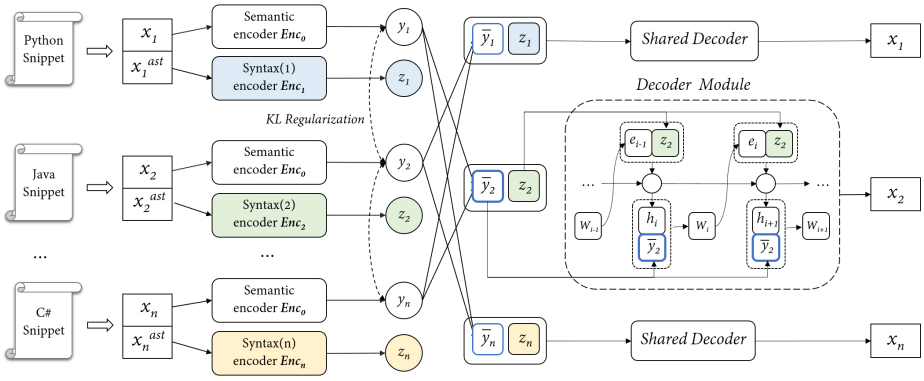

Figure 3: The reconstruction of code snippets written in multiple programming languages for the same problem. Enc_0 corresponds to the Semantic (Y) Encoder shared parameters among all programming languages. Enc_i corresponds to the Syntax (Z) Encoder dedicated to programming language i with independent parameters (*IZ*). The *KL divergence term* is imposed to the semantic latent variable y to enforce the alignment of different code snippets in the semantic space. The *decoder* also shares the parameters among all programming languages.

3 CODEDISEN Approach

Our approach is a variant of notable VAE architecture. In this paper, we start from the vMF-Gaussian Variational Autoencoder (VGVAE) model (Chen et al., 2019b), which is proposed to disentangle textual semantics from language syntax within the same human language. Our problem setting differs from disentanglement setting of human language in that: (1) We focus on dealing with multiple programming languages, instead of a single language. (2) Unlike human languages, programming language is a formal symbol system and has much stricter syntax rules than human languages, so we can make use of the AST information derived from a code snippet. To handle multiple programming languages, we propose CODEDISEN which adds more inputs and multiple losses to the VGVAE to effectively enforce the disentanglement of code semantics and syntax.

Figure 3 shows the overall architecture of our CODEDISEN approach. Unlike the unsupervised VGVAE, our approach introduces the masked AST x_i^{ast} that just retains the syntax information and removes almost all semantic information by masking leaf nodes. x_i^{ast} provides a strong supervision signal for disentangling code semantics y_i from code syntax z_i. For brevity, we will describe the factorization process from perspective of single code snippet. Following the conditional independence assumption in the graphical model, the joint probability $p_\theta(x, x^{ast}, y, z)$ can be factorized as:

$$p_\theta(x, x^{ast}, y, z) = p_\theta(y)p_\theta(z)\prod_{t=1}^{T} p_\theta(w_t|w_{1:t-1}, y, z)p(x^{ast}|x), \quad (1)$$

where w_t is the t-th word of x and $p_\theta(w_t|w_{1:t-1}, y, z)$ is given by a softmax over a vocabulary $\mathcal{V}$, $p(x^{ast}|x)$ is a deterministic transformation process. Different from the VGVAE variants (Chen et al., 2019b), we propose the following factorization $q_\varphi(y, z|x, x^{ast}) = q_\varphi(y|x)q_\varphi(z|x^{ast})$ to approximate the posterior when applying neural variational inference, since x^{ast} just retains the syntax information, and x contains more semantic information that is missing in compilation nodes. The objective of VAE is to maximize a lower bound of marginal log-likelihood, thus the basic loss $\mathcal{L}_0$ is written as:

$$\mathcal{L}_0 = -\mathop{\mathbb{E}}_{y,z\sim\, q_\theta(y|x), q_\theta(z|x^{ast})} \log p_\theta(x|y, z) + \mathrm{KL}\left(q_\theta(y|x) \parallel p_\theta(y)\right) + \mathrm{KL}\left(q_\theta(z|x^{ast}) \parallel p_\theta(z)\right). \quad (2)$$

3.1 Encoders and Decoder

In this paper, we assume that $q_\theta(y|x)$ follows a vMF distribution (Chen et al., 2019b) and $p_\theta(y)$ follows the uniform distribution $vMF(\cdot; 0)$. Similarly, we assume that $q_\theta(z|x^{ast})$ follows a Gaussian distribution $\mathcal{N}(\mu_\beta(x^{ast}), diag(\sigma_\beta(x^{ast})))$ and that the prior $p_\theta(z)$ is $\mathcal{N}(0; I_d)$, where I_d is a $d \times d$ identity matrix. Concretely, we implement the *semantic encoder* Enc_0 (i.e., $q_\varphi(y|x)$) *shared among all languages* as a bidirectional long short-term memory network (BiLSTM) followed by a 3-layer feedforward neural network. Similarly, we adopt an independent BiLSTM model followed by a 3-layer feedforward network for each syntax encoder Enc_i adhering to programming language i (i.e., $q_{\varphi_i}(z_i|x_i^{ast})$). We also select LSTM model as the shared decoder of our generative model. As shown

in Figure 3, at the decoding stage, we concatenate the syntactic variable z with the previous word's embedding as the input to calculate hidden state h' since grammatical information is more influenced by code token positions. Furthermore, we concatenate the semantic variable y with hidden state h' to predict the code token at each step, which could make full use of semantic information.

3.2 Losses for Disentanglement

In order to effectively enforce the disentanglement of code semantics and syntax, we design three additional loss terms, in addition to the loss $\mathcal{L}_0$.

Cross-Language Reconstruction Loss Since the code snippets $\{\langle x_0, x_1, \ldots, x_n\rangle | x_i \in P_j\}$ solve the same problem of P_j, they should share the same program semantics, inducing the cross-language reconstruction loss. Concretely, we hope that code snippet x_i can be reconstructed from its own syntax representation z_i and semantic representation $\overline{y}_i$. $\overline{y}_i$ is derived from latent semantic representations $\{y_k|k \neq i\}$ of code snippets $\{x_k|k \neq i\}$ that do not use language i. Formally, $\langle \overline{y}_i, z_i\rangle \rightarrow \langle x_i\rangle$. If we can regenerate x_i successfully, it means that $\{y_i\}$ share the almost same program semantic for problem P_j, and z_i encodes language-specific syntax information of programming language i.

As shown in Figure 3, at each step, we input $X = \{x_1, x_2, \ldots, x_n\}$, which is a set of code snippets that have the same program semantics p_j for problem j and are written in distinct programming languages. Formally, the cross-language reconstruction loss can be formulated as:

$$\mathcal{L}_{rec} = -\sum_{i=1}^{n} \mathbb{E}\left[\log p_\theta(x_i|\overline{y}_i, z_i)\right], \tag{3}$$

where $\overline{y}_i$ is calculated as:

$$\overline{y}_i = F_{Linear}(f_{cat}(\overline{Y}_i)), \tag{4}$$

where $\overline{Y}_i$ represents all the latent semantic variables except y_i, i.e., $\overline{Y}_i = \{y_k|k \neq i\}$, $f_{cat}(\cdot)$ is the function of concatenation, and F_{Linear} aims to fuse the concatenated vector to the same dimension as y_i, through a linear layer.

Posterior Distribution Loss Since all the code snippets of n programming languages for the same problem j share the same program semantics p_j, we expect that the posterior distribution

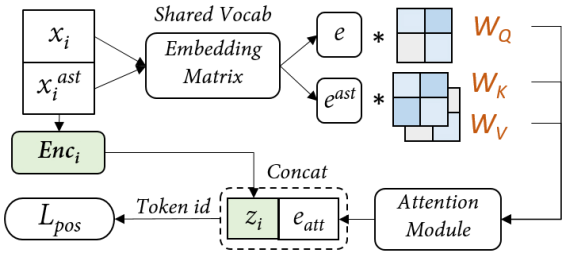

Figure 4: An illustration for the attentive position loss of language i at encoding stage.

$q_{\varphi_i}(y_i|x_i)$ of code snippets of programming language i should be close to the mean posterior distribution $q_m(y_m|x_m)$ of code snippets of all programming languages.

Concretely, we employ KL terms (Chen and Zhou, 2018) to constrain the distribution discrepancy between $q_{\varphi_i}(y_i|x_i)$ and $q_m(y_m|x_m)$ in the latent space. Formally, The posterior distribution constraint loss for programming language i is defined as:

$$\begin{aligned} \mathcal{L}_{dist(i)} &= \mathrm{KL}(q_{\varphi_i}(y_i|x_i) \parallel q_m(y_m|x_m)), \\ q_m(y_m||x_m) &= vMF(\frac{\sum_{i=0}^{n}\mu(x_i)}{n}, \frac{\sum_{i=0}^{n}\kappa(x_i)}{n^2}), \end{aligned} \tag{5}$$

where vMF is the same definition as in (Chen et al., 2019b). The whole posterior distribution loss function is defined as:

$$\mathcal{L}_{dist} = \sum_{i=1}^{n} \mathcal{L}_{dist(i)}. \tag{6}$$

Attentive Code Position Loss As observed in (Chen et al., 2019b), the position information of code token x_i has a significant impact on its syntax, such as `import` is always at position `0` in Python. To better utilise ASTs to represent syntactic information, we introduce an *Token2AST* attention-based code position loss ($\mathcal{L}_{pos}$) to predict positions of code tokens based on the embedding e_i of x_i and the embedding e_i^{ast} of x_i^{ast}. We map the e_i^{ast} to the token side e_i via attention mechanism as shown in Figure 4. Firstly, there is a correlation between the AST nodes and the tokens, e.g., variable `a` is expected to have a higher weight with *identifier* in Figure 2. Secondly, the length of AST sequences is often much longer than that of the tokens in the code snippet. Therefore the attentive code position loss can fuse AST and token information to better extract syntactic features.

We implement $\mathcal{L}_{pos}$ for both encoder and decoder to predict code tokens position i, which consists of a 3-layer feedforward neural network $f(\cdot)$

with the input from the concatenation of the samples of the syntactic variable z and the attention embedding vector e_{att} at input position i. The $\mathcal{L}_{pos}$ and e_{att} are defined as:

$$e_{att} = \text{softmax}(\frac{(e_i \cdot W_q) \cdot (e_i^{ast} \cdot W_k^T)}{\sqrt{d}}) \cdot (e_i^{ast} \cdot W_v)$$
$$\mathcal{L}_{pos} = - \mathop{\mathbb{E}}_{z \sim q_\varphi(z|x)} \left[\sum_i \log \text{softmax}(f([e_{att}; z]))_t \right], \quad (7)$$

where d is the dimension of the embedding, to increase the training stability, $\text{softmax}(\cdot)_t$ indicates the probability of a code token at position t, and W_q, W_k, and W_v respectively denotes the query, key and value matrix in the attention mechanism.

Overall Objective We subsequently define the overall objective as the combination of the aforementioned basic loss and three additional losses for disentanglement. The total loss function is formulated as follows:

$$\mathcal{L} = \mathcal{L}_0 + \alpha \cdot \mathcal{L}_{rec} + \beta \cdot \mathcal{L}_{dist} + \gamma \cdot \mathcal{L}_{pos}. \quad (8)$$

4 Experiment and Analysis

In this section, we aim to address the following research questions: (1) *Can the program semantics and syntax be successfully disentangled by our proposed* CODEDISEN? (2) *Will disentangled semantics indeed improve the performance of downstream tasks*? and (3) *What is the generalizability of disentangled code representation across different programming languages*? We also perform ablation analysis to investigate the effect of each module of the model, as well as a qualitative analysis of detailed examples.

To answer the above questions, our experiments will validate the following two principles: ***(1) Equivalence of Semantics.*** Given a sequence of semantically identical code snippets $x_1, \ldots, x_n$ and their corresponding masked ASTs $x_1^{ast}, \ldots, x_n^{ast}$, CODEDISEN will yield y_i and $\overline{y}_i$ (see Eq. (4)), if $x_i = Decode(y_i, z_i) = Decode(\overline{y_i}, z_i)$, then we have $y_i = \overline{y_i}$, which means the shared semantic encoder Enc_0 extracts the same features for those code snippets. Further, given $x_i \neq x_j$ and $y_i = \overline{y}_i = y_j$, we have $z_i \neq z_j$, which means $Enc_{1\ldots n}$ yields the respective syntax representations adhering to programming language $1 \ldots n$. ***(2) Orthogonality of Semantic and Syntax Vector.*** If the semantic vector y is completely disentangled from the syntax vector z, just applying y to downstream tasks will improve the performance, compared to applying x to downstream tasks. However, just applying z will perform poorly.

Implementation Details As for building vocabulary, we observe that more than 95% of the vocabulary of multi-lingual code snippets are user-defined variable names, with a tiny percentage of keywords and compilation nodes with respect to each programming language. Additionally, variable names in different code snippets for the same problem are likely to share semantics, which can facilitate implicit alignment of the semantics of code snippets of different languages. Hence, we resort to constructing a *shared vocabulary* for all code snippets and ASTs of all programming languages. More implementation details are referred to the Appendix A.1.

4.1 Dataset and Downstream Tasks

For multi-lingual cross-training, we use the CLCDSA dataset (Nafi et al., 2019), which is composed of 26,000 code snippets across four programming languages (i.e., Java, Python, C# and C++). This dataset is collected from three open source programming contest sites (i.e., AtCoder[2], Google CodeJam[3] and CoderByte[4]). All solutions in this dataset are functionally similar but written in different programming languages. In our experiments, we choose Java, Python, C# and C++ as the target languages, and limit the maximum code tokens length to 128. Consequently, we obtain a training dataset containing $2,500$ samples per language, and 500 samples for both validation and testing.

Code Clone Detection Code cloning across languages, which reuses a fragment of source code via copy-paste-modify, is a common way for code reuse and software prototyping. We treat the solutions belonging to different languages for the same problem as positive samples and the other random solution combinations in each batch as negative samples. We control the number of positive/negatives samples are balanced. We set the threshold as 0.8, which means that the cross-language input code pairs are semantic identical if the cosine similarity between them is greater than 0.8. For evaluation, we select LSTM (Sundermeyer et al., 2012), Tree-LSTM (Shido et al., 2019), TBCNN (Mou et al., 2016) and GraphCodeBERT (Guo et al., 2020) models as baselines.

[2]https://atcoder.jp/
[3]https://codingcompetitions.withgoogle.com/codejam
[4]https://coderbyte.com/

Table 1: The performance of CODEDISEN on code reconstruction of Python w.r.t different data size and training languages.

Method	*Size*	*Languages in Training*	BLEU-1	CIDER	ROUGE-L
VGVAE	1k	*Java/Python*	29.033	36.372	49.953
CODEDISEN	1k	*Java/Python*	29.750	40.102	50.079
	1k	*Java/Python/C#/C++*	34.861	47.455	55.616
CODEDISEN	2.5k	*Java/Python*	34.519	48.039	51.802
	2.5k	*Java/Python/C#/C++*	**44.765**	**116.90**	**59.159**

Code-to-Code Search During software development process, developers often look for code snippets that offer similar functionality (Kim et al., 2018). Our goal is to search the code snippet of other programming languages with the same functionality based on the current code snippet. To be more challenging, there is only one code snippet matching the query in the queried collection. We compare the code snippet in the source language with all code snippets in the target language to calculate their cosine similarity. For evaluation, we select BiLSTM (Linhares Pontes et al., 2018), Tree-LSTM (Shido et al., 2019), TBCNN (Mou et al., 2016) and GraphCodeBERT (Guo et al., 2020) models as baselines, and we adopt Accuracy, MRR and NDCG as evaluation metrics.

Code Translation In cross-language reconstruction, we know that x_i and x_j are source and target code fragments, which are semantically identical and belong to the same problem P_j. However, in cross-language code translation, we do not know x_j and have to sample a random code snippet $x_j^{'}$ in language j to obtain syntactic features $z_j^{'}$. We use this task to demonstrate that our model extracts non-zero and identical syntactic features for the same programming language. In addition, we use Tree-LSTM and VGVAE as baselines, to demonstrate the superior performance of our model on cross-programming language tasks.

4.2 Disentangled or Not? (RQ1)

To check the equivalence of semantics, we conduct experiment of reconstruction on Python code snippets. For a given Python code snippet x_i in the test set, our CODEDISEN yields the *aggregated semantic vector* $\overline{y}_i$ from semantically identical code snippets of other programming languages, i.e. Java or Java/C#/C++, as well as the *syntax vector* z_i from the Python code snippet x_i. Then $\overline{y}_i$ and z_i are jointly used to reconstruct the Python code snippet $\overline{x}_i$. We adopt the BLEU (Papineni et al., 2002), CIDER (Vedantam et al., 2015) and ROUGE (Lin, 2004) to measure the quality of reconstructed text $\overline{x}_i$ from x_i.

Table 1 shows the reconstruction performance of our CODEDISEN under various multi-lingual settings with different sizes of dataset. From this table, we observe that our model which is trained using AST information of 1,000 ($1k$) Java and 1,000 ($1k$) Python programs significantly outperforms vanilla VGVAE. It is also interesting to find that our model achieves a significant performance improvement when (1) we increase the training data from $1k$ to $2.5k$ samples for each language and (2) expand the bi-lingual model to a multi-lingual architecture (Java/Python/C#/C++). Furthermore, when comparing with VGVAE, CODEDISEN achieves 15.7%, 80.6% and 9.2% performance gains in terms of BLEU-1, CIDER and ROUGE-L, respectively. It is worth noting that CODEDISEN when trained using $1k$ samples still outperforms the bi-lingual model trained using $2.5k$ samples. The total dataset sizes used for training CODEDISEN are $1k \times 4 = 4k$ and $2.5k \times 2 = 5k$. This indicates that the multi-lingual architecture is good at dealing with more languages in training, since variable names may share similar semantics across different programming languages.

To further check the orthogonality of semantic and syntax vectors, we conduct experiments using the shared semantic vector and language-specific syntax vectors on downstream tasks. As shown in Table 2, CODEDISEN (Y) denotes only using the output y of the shared semantic encoder Enc_0 in code-to-code search. The performance has a significant improvement compared to BiLSTM without Enc_0. CODEDISEN (Z) means only using the output z of the syntax encoder, whose performance even has a dramatic drop. This indicates that the hidden vector y contains rich semantic information, while the hidden vector z rarely contains semantic information. As shown in Table 3, CODEDISEN (R) means randomly sampling a code snippet $x_j^{'}$ in the training set to extract syntactic feature $z_j^{'}$ for reconstruction. We can observe that little degradation in model performance indicates that the syntax vector of randomly sampled $x_j^{'}$ is almost the same as that of the original code snippet x_j of the same language j. When we set the variable z to zero tensor, we find that the model performance drops significantly. It confirms that the syntax vector z is critical in the reconstruction process and z is almost identical within the same programming language.

Table 2: Effectiveness of shared semantic encoder of our model in code-to-code search (2.5k/Java/Python).

Method	ACC	MRR	NDCG
BiLSTM	0.166	0.275	0.413
Tree-LSTM	0.081	0.162	0.293
TBCNN	0.007	0.024	0.169
GraphCodeBERT	0.246	0.314	0.343
CODEDISEN (Y)	**0.316**	**0.436**	**0.551**
CODEDISEN (Z)	0.004	0.021	0.157

Table 3: Effectiveness of the syntax encoder of CODEDISEN in code translation (1k/Java/Python).

Method	BLEU-1	ROUGE-L	CIDER
Tree-LSTM	25.84	37.53	36.96
VGVAE	29.03	49.95	36.37
VGVAE (R)	24.94	36.09	25.09
CODEDISEN	**29.75**	**50.08**	**40.10**
CODEDISEN (R)	29.51	48.64	38.92
CODEDISEN (0)	13.82	15.55	1.73

4.3 Downstream Task Performance (RQ2)

To better verify whether the disentangled multi-lingual code semantic representation can boost the performance of downstream tasks, we fine-tune the model on the downstream tasks of code translation, code-to-code search and code clone detection under the cross-language setting.

As shown in Table 4, CODEDISEN (Y) that only considers the semantics of code achieves the best performance, significantly outperforming the performance of counterpart CODEDISEN (Z) that only considers the syntax of code. We set the threshold value to 0.8 according to the testing performance on code clone detection task. When we set the threshold to 0.5, the performances of Tree-LSTM and CODEDISEN are 0.576/0.954/0.718, and 0.724/0.992/0.837, in terms of Precision, Recall and F1, respectively. This is because that if we set the threshold to a lower value, more code snippets may be classified as duplicates, thus the recall increases while the precision decreases. Therefore, we choose a threshold of 0.8 to better compare the differences in performance between models.

It is noteworthy that the models such as Tree-LSTM and TBCNN, which accept ASTs of a program as their inputs can obtain high recall but low precision. This indicates that if the ASTs are same, to a large extent, the two programs can be considered as semantically identical, so the recall is high. However, the ASTs of different programming languages vary greatly and generate many temporal variables during compilation, thus introducing noise nodes, so the precision can be low. Our approach combines the advantages of token and AST features while obtaining high precision and recall on cross-programming language tasks. GraphCodeBERT, a pre-trained model on the code corpora of multi-programming language, is suitable for fine-tuning on specific task of a programming language. For the task of code clone detection, we simply fine-tune the model based on the released checkpoint of GraphCodeBERT, under the setting of our scenario. For the task of code-to-code search, we extract the last layer of GraphCodeBERT output and take the average value as the feature of the code segment, and calculate the cosine similarity to select the target from candidates, as described in the Appendix A.3. As shown in Table 2 and Table 4 , we can find that GraphCodeBERT does not adapt well to cross-programming language semantic matching related tasks.

Table 4: Effectiveness of semantic encoder of our model in code clone detection (2.5k/Java/Python).

Method	Precision	Recall	F1
LSTM	0.85	0.75	0.79
Tree-LSTM	0.78	0.84	0.81
TBCNN	0.50	**0.99**	0.66
GraphCodeBERT	0.56	0.54	0.50
CODEDISEN (Y)	**0.88**	0.93	**0.90**
CODEDISEN (Z)	0.50	0.33	0.38

Table 3 shows that although Tree-LSTM is more suitable for encoding the structure information of code than our LSTM-based model, our CODEDISEN (R) still outperforms Tree-LSTM in BLEU-1, ROUGE-L, CIDER by 3.7%, 11.1%, 2.0%, respectively. By introducing AST information, our CODEDISEN (R) also has a significant improvement when compared to VGVAE. Table 2 shows that Tree-LSTM and TBCNN perform poorly for cross-language code search tasks. The main reason is that both Tree-LSTM and TBCNN are based only on the input representation of an AST. However, the ASTs of two semantically equivalent programs written in two different languages (e.g., Java and Python) can be generated quite differently by the compilers of these two languages, hence introducing syntax-level noise.

4.4 Generalizability of CODEDISEN (RQ3)

To investigate the generalizability of our semantic module across languages, we evaluate our model on unseen datasets in different languages. In addition, we compare the performance when combining different languages in the code-to-code search task to demonstrate the superiority of multi-lingual structures. From Table 5, we observe that when

Table 5: Generalization ability of semantic encoder.

Languages (Testing)	ACC	MRR	NDCG	Languages(Training)
Java-Python	0.316	0.436	0.551	*Java/Python*
Java-C#	0.298	0.420	0.538	*Java/Python*
C++-C#	0.234	0.351	0.481	*Java/Python*
Java-Python	**0.330**	**0.452**	**0.564**	*Java/Python/C#/C++*
C++-Python	0.255	0.375	0.499	*Java/Python*
C++/Python	0.279	0.402	0.519	*Java/Python/C#*

Table 6: Ablation study of CODEDISEN, where *IZ* denotes independent syntax encoders $Enc_{1\sim n}$ and *KL* denotes semantic KL term.

Method	BLEU-1	ROUGE-L	CIDER
CODEDISEN	44.77	59.16	116.9
-IZ	32.12	45.82	27.40
-KL	36.71	53.89	67.06
$-L_{pos}$	41.19	55.05	110.1
L_{pos}-att	41.93	56.90	102.6

we use the shared semantic encoder trained on the Java/Python dataset, our model still achieves good results on C++-C# and Java-C# code-to-code search tasks after fine-tuning. Note that C++-C# data are not there when training CODEDISEN , and our model keeps most of its performance on Java-Python dataset in Table 2. This is a good evidence that the semantic representations extracted by our model are generalizable across languages.

For Java-Python code search, the code semantic encoder trained on four languages (Java/Python/C#/C++) performs better than the one trained on two language (Java/Python). For C++-Python code search, we ensure the training dataset free of C++ code snippets. We find that the code semantic encoder trained on Java/Python/C# performs better than the one trained on Java/Python. These indicate that our multi-lingual architecture can further utilise the samples of more programming languages to train a better semantic encoder, and is extensible to train more language-specific syntax encoders.

4.5 Ablation Study

We conduct ablation analysis to understand the performance contribution from different component in our model. As shown in Table 6, we choose the model trained on four languages as the baseline (CODEDISEN). In fact, we find that the independent *Syntax* encoders (IZ) and KL term (KL) have a significant impact on the multi-lingual model. When we remove these components, the BLEU-1 scores of our model drop by 12.65% and 8.06%. This suggests that the design of implicit semantic alignment and syntactic independence between multiple programming languages is effective.

We also explore the role of attention in the code position loss, while AST sequences are usually much longer and more complicated than code tokens. The results show that when we use the code position loss without *Token2AST* attention (L_{pos}-att), performance of (L_{pos}-att) is close to that of (-L_{pos}) removing code position loss. It means our *Token2AST* attention mechanism could merge the syntactic AST features and the semantic features of tokens to handle the long sequence dependence.

4.6 Qualitative Analysis

We conduct case study to further investigate results of the semantic extraction in code refactoring and abstract syntax representation, as shown in Table 7. From *Case 1*, it is clear to see that the variable names in the generated snippets are consistent with the semantic input Java snippets. Then we compare the semantic information between the generated and the input semantic code pairs. As shown in *Case 2*, the syntax input does not have "`Yes`" or "`No`" at all, but our generated snippet extracts this from the semantic input very well. In addition, we have rewritten the complex multivariate input form of Java into the simple map input of Python, which demonstrates that our model can extract semantics well. On the other hand, we find that the generated snippets are compliant with Python syntax. In conjunction with the random syntax sampling discussed earlier, we can further show that the syntax variables we extracted abstractly represent the syntax of specific programming language.

5 Related Work

Deep Code Representation The existing code representation works represent code snippets in three ways, i.e., token-based representation, AST-based representation, and graph-based representation. As for token-based representation (Hindle et al., 2012; Bhoopchand et al., 2016), code snippets are tokenized into token sequences and each code token is represented as a real-valued vector. As for AST-based representation, one line of work is to directly represent the tree structure via Tree-CNN (Mou et al., 2016) or Tree-LSTM (Chen et al., 2018) . Another line of work is to indirectly represent the AST by linearizing the AST into a sequence of nodes (Hu et al., 2018; Alon et al., 2019; Liu et al., 2020) via traversing. Wan et al. (Wan

Table 7: Example of generated results by code translation (*Java/Python*).

	Case 1
Semantic	Scanner sc = new Scanner(System.in); int a = sc.nextInt(); int b = sc.nextInt(); if(a < b) System.out.println (b); else System.out.println (a);
Syntax	h1 = int (input ()) h2 = int (input ()) print (h1 - h2)
Reference	a , b = map (int ,input ().split ()) print (max(a , b))
Generated	a , b = map (int ,input ().split ()) print (a + b)
	Case 2
Semantic	Scanner sc = new Scanner (System.in); int A = sc.nextInt(); int B = sc.nextInt(); int C = sc.nextInt(); if(C <= A + B) System.out.println (" Yes "); else System.out.println (" No ");
Syntax	n = int (input ()) if n == 12 : print (1) else : print (n + 1)
Reference	A , B , C = map (int ,input ().split ()) if A + B < C : print (" No ") else : print (" Yes ")
Generated	A , B = map (int ,input ().split ()) if A == B : print (" YES ") else : print (" NO ")

et al., 2018; Wang et al., 2020b; Wan et al., 2019; Hua et al., 2021) propose to integrate the semantics of code from different views (e.g., the tokens, AST and control-flow graph) into a hybrid feature space, and put forward a hybrid representation approach, for the task of code summarization, code search and code clone detection. As for graph-based representations several works resort to parse the program into a graph (e.g., augmented AST, control-flow graph, and data-flow graph) (Li et al., 2015; LeClair et al., 2020; Wan et al., 2019; Sui et al., 2020; Guo et al., 2020). Benefiting from the strong power of pre-training technique in natural language processing, recently, several works (Kanade et al., 2020; Feng et al., 2020; Guo et al., 2020) propose to pre-train a masked language model on the large-scale of code corpora, like BERT (Devlin et al., 2019). CodeBERT (Feng et al., 2020) pre-trains a language model on the source codes and natural language descriptions, and significantly boosts performance on code search. GraphCodeBERT (Guo et al., 2020) advances the CodeBERT by incorporating the data-flow information among variables into pre-training.

Multilingual Knowledge Transfer For multilingual tasks, if we treat word embedding spaces isomorphic between different languages, which has been shown not to hold in practice (Søgaard et al., 2018), and fundamentally limits their performance. Sabet et al. (2019) train a bilingual model on bilingual corpora by introducing a cross-lingual loss in addition to the monolingual loss. The model learns to translate on each other by inputting parallel data sets at one step simultaneously. This ensures that the word and n-gram embeddings of both languages lie in the same space. Our approach is primarily referenced to text-controlled generation, which transfers the knowledge by dissociating tangled representations. Cross-training disentangling methods (Chen et al., 2019a,b; Wang et al., 2020a) on the controlled text generation task, which are implemented in a VGVAE framework and guided by paraphrase reconstruction loss have inspired us a lot. In particular, the syntax input of the code can be conveyed via AST. Code syntax regularity can be well exploited in multilingual architectures to achieve semantic alignment in dissociated latent spaces to improve the quality of representations with desirable generalizability.

6 Conclusion

In this paper, we propose a novel disentangled code representation learning approach under multilingual setting. We introduce three dedicated losses to enforce the disentanglement of code semantics and syntax. Comprehensive experiments on the three downstream tasks validate the effectiveness of our disentangled semantic and syntax representation. In the future, we will devise more effective disentanglement models for code representation learning. Another line is to extend the proposed approach to cross-lingual customer service robots, where answers of different languages for the same question share the same semantic information.

Acknowledgement

We thank the anonymous reviewers for their helpful comments. This work is supported by National Key R&D Program of China (No. 2018AAA0101900), the NSFC projects (No. 62072399, No. U19B2042, No. 61402403), Chinese Knowledge Center for Engineering Sciences and Technology, MoE Engineering Research Center of Digital Library, National Engineering Research Center for Big Data Technology and System, the Fundamental Research Funds for the Central Universities, and partially supported by Australian Research Grant DP21010134.

References

Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–29.

Avishkar Bhoopchand, Tim Rocktäschel, Earl Barr, and Sebastian Riedel. 2016. Learning python code suggestion with a sparse pointer network. *arXiv preprint arXiv:1611.08307*.

Mingda Chen, Qingming Tang, Sam Wiseman, and Kevin Gimpel. 2019a. Controllable paraphrase generation with a syntactic exemplar. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 5972–5984, Florence, Italy. Association for Computational Linguistics.

Mingda Chen, Qingming Tang, Sam Wiseman, and Kevin Gimpel. 2019b. A multi-task approach for disentangling syntax and semantics in sentence representations. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 2453–2464, Minneapolis, Minnesota. Association for Computational Linguistics.

Qingying Chen and Minghui Zhou. 2018. A neural framework for retrieval and summarization of source code. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 826–831. IEEE.

Xinyun Chen, Chang Liu, and Dawn Song. 2018. Tree-to-tree neural networks for program translation. In *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*, pages 2552–2562.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, Minneapolis, Minnesota. Association for Computational Linguistics.

Bui Nghi DQ, Yijun Yu, and Lingxiao Jiang. 2019. Bilateral dependency neural networks for cross-language algorithm classification. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 422–433. IEEE.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*.

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*.

Abram Hindle, Earl T Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. 2012. On the naturalness of software. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 837–847. IEEE.

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)*, pages 200–20010. IEEE.

Wei Hua, Yulei Sui, Yao Wan, Guangzhong Liu, and Guandong Xu. 2021. Fcca: Hybrid code representation for functional clone detection using attention networks. *IEEE Transactions on Reliability*, 70(1):304–318.

Paras Jain, Ajay Jain, Tianjun Zhang, Pieter Abbeel, Joseph E Gonzalez, and Ion Stoica. 2020. Contrastive code representation learning. *arXiv preprint arXiv:2007.04973*.

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.

Kisub Kim, Dongsun Kim, Tegawendé F Bissyandé, Eunjong Choi, Li Li, Jacques Klein, and Yves Le Traon. 2018. Facoy: a code-to-code search engine. In *Proceedings of the 40th International Conference on Software Engineering*, pages 946–957.

Alexander LeClair, Sakib Haque, Linfgei Wu, and Collin McMillan. 2020. Improved code summarization via a graph neural network. *arXiv preprint arXiv:2004.02843*.

Y. Li, D. Tarlow, M. Brockschmidt, and R. Zemel. 2015. Gated graph sequence neural networks. *arXiv preprint arXiv:1511.05493*.

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.

Elvys Linhares Pontes, Stéphane Huet, Andréa Carneiro Linhares, and Juan-Manuel Torres-Moreno. 2018. Predicting the semantic textual similarity with Siamese CNN and LSTM. In *Actes de la Conférence TALN. Volume 1 - Articles longs, articles courts de TALN*, pages 311–320, Rennes, France. ATALA.

Fang Liu, Lu Zhang, and Zhi Jin. 2020. Modeling programs hierarchically with stack-augmented lstm. *Journal of Systems and Software*, 164:110547.

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, February 12-17, 2016, Phoenix, Arizona, USA*, pages 1287–1293. AAAI Press.

Kawser Wazed Nafi, Tonny Shekha Kar, Banani Roy, Chanchal K Roy, and Kevin A Schneider. 2019. Clcdsa: cross language code clone detection using syntactical features and api documentation. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1026–1037. IEEE.

Lun Yiu Nie, Cuiyun Gao, Zhicong Zhong, Wai Lam, Yang Liu, and Zenglin Xu. 2020. Contextualized code representation learning for commit message generation. *arXiv preprint arXiv:2007.06934*.

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 of the Association for Computational Linguistics*, pages 311–318, Philadelphia, Pennsylvania, USA. Association for Computational Linguistics.

Ali Sabet, Prakhar Gupta, Jean-Baptiste Cordonnier, Robert West, and Martin Jaggi. 2019. Robust cross-lingual embeddings from parallel sentences. *arXiv preprint arXiv:1912.12481*.

Yusuke Shido, Yasuaki Kobayashi, Akihiro Yamamoto, Atsushi Miyamoto, and Tadayuki Matsumura. 2019. Automatic source code summarization with extended tree-lstm. In *2019 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8. IEEE.

Anders Søgaard, Sebastian Ruder, and Ivan Vulić. 2018. On the limitations of unsupervised bilingual dictionary induction. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 778–788, Melbourne, Australia. Association for Computational Linguistics.

Yulei Sui, Xiao Cheng, Guanqin Zhang, and Haoyu Wang. 2020. Flow2vec: value-flow-based precise code embedding. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–27.

Martin Sundermeyer, Ralf Schlüter, and Hermann Ney. 2012. Lstm neural networks for language modeling. In *Thirteenth annual conference of the international speech communication association*.

Ramakrishna Vedantam, C. Lawrence Zitnick, and Devi Parikh. 2015. Cider: Consensus-based image description evaluation. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2015, Boston, MA, USA, June 7-12, 2015*, pages 4566–4575. IEEE Computer Society.

Yao Wan, Jingdong Shu, Yulei Sui, Guandong Xu, Zhou Zhao, Jian Wu, and Philip S Yu. 2019. Multi-modal attention network learning for semantic source code retrieval. *arXiv preprint arXiv:1909.13516*.

Yao Wan, Zhou Zhao, Min Yang, Guandong Xu, Haochao Ying, Jian Wu, and Philip S Yu. 2018. Improving automatic source code summarization via deep reinforcement learning. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 397–407.

Shaonan Wang, Jiajun Zhang, Nan Lin, and Chengqing Zong. 2020a. Probing brain activation patterns by dissociating semantics and syntax in sentences. In *AAAI*, pages 9201–9208.

Wenhua Wang, Yuqun Zhang, Yulei Sui, Yao Wan, Zhou Zhao, Jian Wu, Philip Yu, and Guandong Xu. 2020b. Reinforcement-learning-guided source code summarization via hierarchical attention. *IEEE Transactions on Software Engineering*, pages 1–1.

A Appendices

A.1 Training Details

All the experiments are conducted on 2 Geforce GTX 1080Ti GPUs. It tasks about 4 hours to train CODEDISEN . For encoder and decoder networks of CODEDISEN, we use the same BiLSTM model structure. The embedding size is 100 and the hidden size is 100. The dimensionality of *Semantic* latent variable vector is 50. The dimensionality of *Syntax* latent variable vector is 50. Specially, the hidden size in feed-forward network and attention mechanism is also 100. The coefficient α of the cross-language reconstruction loss $\mathcal{L}_{rec}$ is 1.0. When calculating the KL divergence term, the coefficient β of the Posterior distribution loss $\mathcal{L}_{dist}$ is 0.1, the coefficient of the $vMF(\cdot)$ KL divergence is $1e$-4 and the coefficient of the $Gaussian(\cdot)$ KL divergence is $1e$-3. The coefficient γ of the attentive code position loss $\mathcal{L}_{pos}$ is 1.0. We train each model for 60 epochs and the batch size is 10 for each programming language.

A.2 Case Study

As shown in Table 8, the semantic inputs and reference code snippets are semantically identical yet grammatically different. Based on the semantic information extracted from a Java code snippet and the syntax information extracted from a random selected Python code snippet, our approach can generate a Python snippet similar to the reference Python snippet which is semantically similar to Java input. We find that the semantics of the snippet we generated and the reference snippet are very similar, especially the content of printed string, such as "`YES`" or "`NO`", "`Even`" or "`Odd`", even the rare words ("`Christmas Eve...`"). At the same time, the generated code snippets are completely unaffected by randomly sampled syntax input. This means that our semantic and syntactic disentanglement modules perform well in extracting shared semantic information from code snippets for the same programming exercise and general syntactic features belonging to specific programming language.

Note that our generative model will be deficient in reconstructing mathematical expressions. For example, the reference snippet is "`a%2==0`" or "`b%2==0`" and the generated is "`a%b`". The main reason is that the specific content of mathematical expressions is less weighty in the semantic expression of a code snippet, and our model tends to focus more on generating an expression rather than on the content of expressions. Another drawback is that our model can not generate long code snippets well, e.g., "`'')`" is missing after the "`print(``Christmas Eve ... Eve`" in the third example. In the future, we will replace the original mathematical expressions with word descriptions of longer token length to increase the weight in reconstruction loss function. In addition, we will use tree-structured decoders to guarantee the executability of the generated code so as to increase long dependencies.

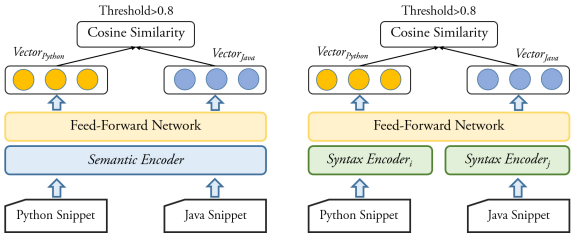

Figure 5: The framework of code clone detection. The left shows training on the semantic module and the right shows training on the specific syntax modules.

A.3 Architecture of Downstream Tasks

In this section, we detail the model architecture of cross-language code clone detection and code-to-code search tasks. The model for the code translation is identical to the cross-language reconstruction model used for the disentanglement training, except that the code snippets from which the syntactic latent variables are extracted are randomly sampled.

The key component of the proposed downstream tasks flow is the Bi-NN. It is modeled as two underlying subnetworks followed by a classification layer. In our work, the underlying subnetworks are semantic and syntax modules and other baseline networks such as BiLSTM. The classifier we defined as a 2-layer shared feed-forward network and calculate the cosine similarity of the input cross-language samples.

A.3.1 Code Clone Detection

Code cloning across languages, which reuses a fragment of source code via copy-paste-modify, is a common way for code reuse and software prototyping. We train and test the code clone detection task on *Java/Python*, *Python/C++*, *C++/C#* and *C#/Java* language pairs. In particular, we calculate the metric scores on average, as shown in Figure 5. We treat the solutions belong to different

Table 8: More examples of reconstructed results by random syntax (Java/Python).

	Code Snippets
Semantic	import java . util .*; public class Main public static void main (String [] args) Scanner sc = new Scanner (System . in); String [] line = sc . next Line (). split (“ ”); int r = Integer . parse Int (line [0]) * 100 ; int g = Integer . parse Int (line [1]) * 10 ; int b = Integer . parse Int (line [2]); int result = r + g + b ; if (result % 4 == 0) System . out . println (“ YES ”); return ; System . out . println (“ NO ”);
Syntax	s = input () num = “ ” for i in range (len (s)): if s [i] in “ 0123456789 ”: num += s [i] print (num)
Reference	x , y , z = input (). split () a = int (x + y + z) if a % 4 == 0 : print (‘ YES ’) else : print (‘ NO ’)
Generated	a , b , c = map (int , input (). split ()) if a % b == 0 : print (“ Yes ”) else : print (“ No ”)
Semantic	import java . util .*; public class Main public static void main (String [] args) Scanner sc = new Scanner (System . in); int a = sc . next Int (); int b = sc . next Int (); if (a % 2 == 0 || b % 2 == 0) System . out . print (“ Even ”); else System . out . print (“ Odd ”); sc . close ();
Syntax	N = int (input ()) ans = 0 for i in range (N): l , r = map (int , input (). split ()) ans += r - l print (ans + N)
Reference	a , b = map (int , input (). split ()) if a % 2 == 0 or b % 2 == 0 : print (‘ Even ’) else : print (‘ Odd ’)
Generated	a , b = map (int , input (). split ()) if a % b == 0 : print (“ Even ”) else : print (“ Odd ”)
Semantic	import java . io .*; import java . util .*; public class Main public static void main (String [] args) try Scanner sc = new Scanner (System . in); int d ; d = Integer . parse Int (sc . next ()); System . out . print (“ Christmas ”); for (int i = 0 ; i < 25 - d ; i ++) System . out . print (“ Eve ”); System . out . println (“”); catch (Exception e) System . out . println (“ out ”);
Syntax	n = int (input ()) k = int (input ()) if n > 2 * k : ans = “ YES ” else : ans = “ NO ” print (ans)
Reference	D = int (input ()) if D == 25 : print (“ Christmas ”) else : if D == 24 : print (“ Christmas Eve ”) else : if D == 23 : print (“ Christmas Eve Eve ”) else : print (“ Christmas Eve Eve Eve ”)
Generated	A , B = map (int , input (). split ()) if A == B : print (“ Christmas Eve Eve Eve Eve ”) elif D == 23 : print (“ Christmas Eve Eve Eve Eve Eve Eve Eve

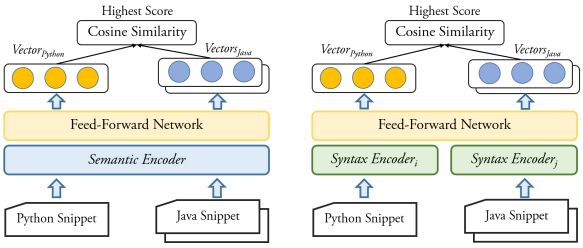

Figure 6: The framework of code-to-code search. Given a query code snippet written in Python as well as a series of candidate code snippets written in Java, the goal of code-to-code search is to retrieve the most relevant Java snippets based on cosine similarity.

languages for the same problem as positive samples and the other random solution combinations in each batch as negative samples. To be more challenging, we extracted 350 programming problems from CLCDSA dataset such that each problem has only one solution per language for evaluation. We control the number of positive/negatives samples are balanced. We set the threshold as 0.8(@80). It means that if the cosine similarity of cross-language input code pairs is greater than 80%, we consider them as semantic clone pairs. In addition, we use the semantic module and the syntax modules compared to baselines in Table 4 to validate that extracted semantics features could improve the performance and our syntax modules may perform poorly because of missing semantic information.

A.3.2 Code-to-Code Search

The training language pair combinations and dataset construction are the same as the code clone detection task. We control that each programming language has only one unique solution for each programming problem. When evaluating our model, we compare the code snippet in source query language to the all code snippets in target language, calculating their cosine similarity. Then we predict the type of algorithm by greedy choosing the highest score sample as shown in Figure 6. In contrast to the usual algorithm classification of one-hot tags, we chose to compare the similarity with all samples of the target domain to do code-to-code search. This makes the more difficult and convincing task to validate the quality of the semantic representation of the code.