Alignment-Guided Curriculum Learning for Semi-Supervised Code Translation

Anonymous ACL submission

Abstract

Neural code translation is the task of converting source code from one programming language to another. The scarcity of parallel code data impedes code translation models' ability to learn accurate cross-language alignment, thus restricting performance improvements. In this paper, we introduce **MIRACLE**, a semi-supervised approach that improves code translation through curriculum learning on code data with ascending alignment levels. It leverages static analysis and compilation to generate synthetic parallel code datasets with enhanced alignment to address the challenge of data scarcity. Extensive experiments show that MIRACLE significantly improves code translation performance on C++, Java, Python, and C, surpassing state-of-the-art models by substantial margins. Notably, it achieves up to a 43% improvement in C code translation with fewer than 150 annotated examples.

1 Introduction

Code translation is the task of converting source code written in one programming language (PL) to another. It is valuable for migrating existing code to other languages, and can significantly reduce the costs of legacy code maintenance and new platform development. One line of work in code translation follows the "pre-training - fine-tuning" approach (Ahmad et al., 2021a; Wang et al., 2021; Roziere et al., 2021a; Fried et al., 2022; Zheng et al., 2023). However, pre-training tasks such as masked language modeling (MLM) and auto-regressive language modeling (Devlin et al., 2019; Feng et al., 2020; Guo et al., 2020) are usually quite different from the downstream tasks such as code translation, and the performance on the latter is limited by the discrepancy. Another line of work takes an unsupervised learning approach for code translation. Established techniques from unsupervised neural machine translation (NMT) (Lample et al., 2017; Artetxe et al., 2017; Lample et al., 2018; Artetxe et al., 2019), such as back-translation and denoising auto-encoding, can be applied to code data effectively, achieving promising performances on code translation (Edunov et al., 2018; Roziere et al., 2020; Agarwal et al., 2021; Ahmad et al., 2022; Szafraniec et al., 2022; Kusum et al., 2022; Huang et al., 2023). However, unsupervised learning introduces significant noise in the training process, which is particularly harmful to code generation tasks that require precision.

Parallel code data refers to pairs of code snippets from different programming languages that are functionally equivalent and bug-free. It is essential for neural models to learn the correct alignment of data structures, APIs, and grammatical rules across different languages. However, existing parallel code data is limited in quantity and supported languages (Nguyen et al., 2013, 2015; Karaivanov et al., 2014; Lu et al., 2021; Chen et al., 2018; Ahmad et al., 2021b; Zhu et al., 2022; Zheng et al., 2023). Without training on sufficient parallel code, the models, especially self-supervised and unsupervised ones, can potentially learn incorrect mappings of syntax and data structures from one language to another (Pan et al., 2023). We refer to this issue as "shallow translation". Figure 1 illustrates an example of shallow translation.

Considering the limitations of existing methods, we argue that it is crucial to efficiently generate high-quality and well-aligned parallel code data to effectively learn cross-lingual alignment. In this paper, we propose a novel se**MI**-supe**R**vised p**A**rallel **C**ode a**L**ignm**E**nt approach, termed **MIRACLE**, that leverages static analysis and compilation to generate synthetic parallel code datasets with enhanced alignment. MIRACLE improves code translation through curriculum learning on code datasets with ascending alignment levels. The static analysis and compilation secure the syntactical correctness and alignment of the synthetic parallel code

Input (**Java**)	Transcoder-ST (**C++**)	MIRACLE (**C++**)

Input (Java)

```
Boolean areElementsContiguous(int arr[], int n) {
    HashSet<Integer> us = new HashSet<Integer>();
    for (int i = 0; i < n; i++)
        us.add(arr[i]);
    int count = 1;
    int curr_ele = arr[0] - 1;
    while (us.contains(curr_ele) == true) {
        count++;
        curr_ele--;
    }
    curr_ele = arr[0] + 1;
    while (us.contains(curr_ele) == true) {
        count++;
        curr_ele++;
    }
    return (count == (us.size()));
}
```

Transcoder-ST (C++)

```
Boolean areElementsContiguous (int arr [ ] , int n ) {
  HashSet < int > us ;
  for ( int i = 0 ; i < n ; i ++ ) {
    us . add ( arr [ i ] ) ;
  }
  int count = 1 ;
  int curr_ele = arr [ 0 ] - 1 ;
  while ( us . contains ( curr_ele ) == true ) {
    count ++ ;
    curr_ele -- ;
  }
  curr_ele = arr [ 0 ] + 1 ;
  while ( us . contains ( curr_ele ) == true ) {
    count ++ ;
    curr_ele ++ ;
  }
  return ( count == ( us . size ( ) ) ) ;
}
```

MIRACLE (C++)

```
bool areElementsContiguous ( int arr [ ] , int n ) {
  unordered_set < int > us ;
  for ( int i = 0 ; i < n ; i ++ )
    us . insert ( arr [ i ] ) ;
  int count = 1 ;
  int curr_ele = arr [ 0 ] - 1 ;
  while (us . find ( curr_ele ) != us . end ( ) ) {
    count ++ ;
    curr_ele -- ;
  }
  curr_ele = arr [ 0 ] + 1 ;
  while (us . find ( curr_ele ) != us . end ( ) ) {
    count ++ ;
    curr_ele ++ ;
  }
  return ( count == ( us . size ( ) ) ) ;
}
```

Figure 1: An example of the "Shallow Translation" problem, with the Java function shown in the first column as input, the C++ translations from baseline method TransCoder-ST, and our proposed method MIRACLE (with CodeT5 as generator). The highlighted parts show that TransCoder-ST's translation directly copied types, data structures, and statements from the input Java code, which are non-existent or grammatically incorrect in the target language C++, while MIRACLE was able to correctly convert them in the corresponding C++ grammar.

in a cost-efficient way. Moreover, the proposed alignment-ascending curriculum learning is robust to data noise, which effectively mitigates the shallow translation problem.

Our contributions can be summarized as follows: **(1)** We propose MIRACLE, a novel semi-supervised code translation method that leverages static analysis and compilation to generate synthetic parallel code with enhanced alignment in a scalable way. The proposed method can be generalized to multiple languages and various models with little overhead. **(2)** We introduce alignment-ascending curriculum learning, where the code translation model is trained on both synthetic parallel code and annotated parallel code, considering the alignment level, noise level, and quantity of each type of data. We demonstrate that curriculum learning improves the code translation model's performance and enhances alignment across different languages, resulting in more precise translations. **(3)** Extensive experiments show that MIRACLE successfully improves code translation performance by up to 30% on C++, Java, and Python, outperforming state-of-the-art baselines on translation between Python and C++ by 5.7%, C++ and Python by 6%, and Python and Java by 8% in execution-based evaluation (CA@1). Notably, our method improves C translations by up to 43% with less than 150 annotated training instances.

2 Method

The lack of parallel code data poses a challenge for training code translation models, which rely on large amounts of parallel data to achieve good performance. Semi-supervised methods can leverage monolingual data to generate synthetic parallel data but often struggle to maintain alignment quality between source and target languages. Therefore, we aim to efficiently generate synthetic parallel code with enhanced cross-lingual alignment through alignment-ascending curriculum learning. Our approach, MIRACLE, focuses on function-level code translation, as functions are the building blocks of programs. Figure 2 shows an overview of the proposed method.

2.1 Parallel Code Data Generation

To address the data scarcity challenge, we propose a parallel code generation method using semi-supervised learning. The method consists of two modules, a hypotheses generator f_G, and a selector f_D. The hypotheses generator f_G is sequence-to-sequence model that takes as input a code snippet x from the source language s and generates a set of hypothetical translations $\mathcal{Y}_h = \{y_h^{(1)}, y_h^{(2)}, .., y_h^{(M)}\}$ in the target language t. Here, $\mathcal{Y}_h$ consists of M translations (hypotheses) for the same input code snippet x. The generator f_G is trained on a limited amount of parallel code data (D_L, L is for labeled), and will be used to generate a large number of hypotheses for monolingual code data (D_U, U is for unlabeled). The selector f_D comprises a set of K filtering criteria $\mathcal{F} = \{F_k\}_{k=1}^{K}$ where $\widetilde{\mathcal{Y}}_{h,k} = F_k(\mathcal{Y}_h)$ takes $\mathcal{Y}_h$ as input and outputs the subset of hypotheses $\widetilde{\mathcal{Y}}_{h,k} \subset \mathcal{Y}_h$ that passes the criterion F_k.

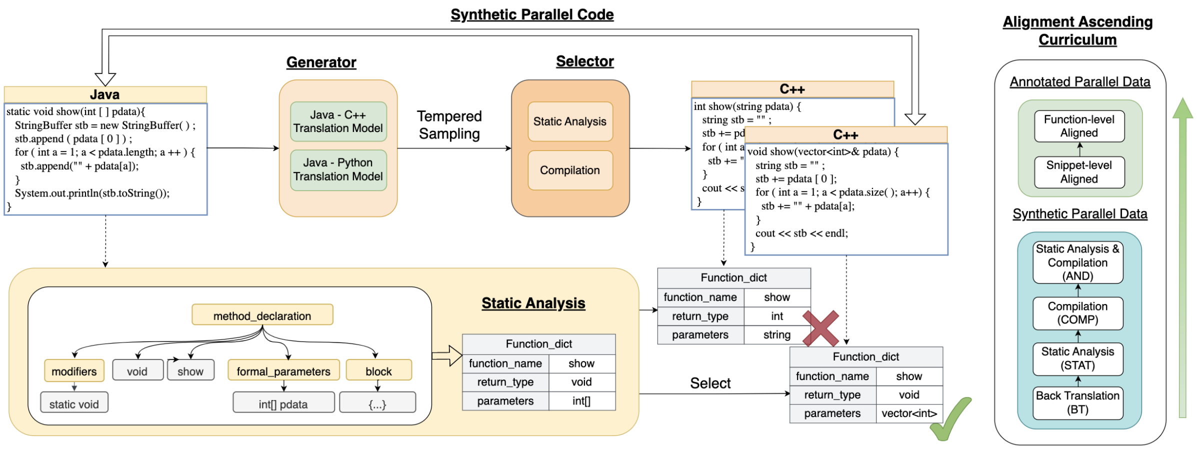

Figure 2: Overview of MIRACLE for Code Translation. MIRACLE utilizes a two-step process to generate high-quality translation hypotheses from monolingual code inputs. First, the generator produces multiple translation hypotheses using tempered sampling. Next, the selector applies static analysis and compilation techniques to select the most promising hypotheses. By employing various selection criteria, MIRACLE generates synthetic parallel code datasets with varying alignment levels and quality. These synthetic datasets, along with annotated parallel code datasets, are organized into a curriculum, where the alignment and quality gradually improve. The proposed curriculum-based approach enhances code translation performance.

2.1.1 Hypotheses Generation

The hypotheses generator f_G is initialized by training on a limited amount of parallel code data. This is to enable f_G with the ability to translate code from the source language s to the target language t. To further improve f_G's translation capability, we leverage the snippet training method from (Zhu et al., 2022), which matches code comments in parallel programs to get snippet-level parallel training data. A snippet usually consists of several lines of code and is not necessarily a complete function. We then use the trained f_G to generate hypotheses for a large amount of monolingual code.

Snippet Training. We use two small annotated parallel code datasets, $\mathcal{D}_{L_s}$ and $\mathcal{D}_L$, with different levels of alignment to train f_G. The parallel code data aligned at snippet-level is denoted as $\mathcal{D}_{L_s} = \{(x,y)^{(ls)}\}_{ls=1}^{|\mathcal{D}_{L_s}|}$, and the function-level parallel data is denoted as $\mathcal{D}_L = \{(x,y)^{(l)}\}_{l=1}^{|\mathcal{D}_L|}$. $\mathcal{D}_{L_s}$ can be constructed from $\mathcal{D}_L$ by matching code comments from the parallel programs (Zhu et al., 2022). We first train f_G on $\mathcal{D}_{L_s}$, and then continue the training on $\mathcal{D}_L$. We refer to this step as snippet training, which helps the generator to learn fine-grained alignment between different languages and substantially improves f_G's ability to generate hypotheses with better alignment to the input code. This step enables f_G to generate valid hypotheses with sufficient initial quality.

Tempered Sampling. Let $\mathcal{D}_U = \{x^{(i)}\}_{i=1}^{|\mathcal{D}_U|}$ be a monolingual dataset in source language s, where each $x^{(i)}$ is a function-level code block. With $\mathcal{D}_U$ as input, we can generate a set of translation hypotheses in the target language t with the trained f_G. To increase the diversity of the hypotheses and improve coverage for different possible translations, we employ tempered sampling to acquire M different hypotheses for each input code. Tempered sampling makes use of a tuned scaled softmax to control the degree of randomness (temperature) in the sampling process (Ackley et al., 1985; Hinton et al., 2015). We denote the hypotheses set as $\mathcal{H} = \{\mathcal{Y}_h{}^{(1)}, \mathcal{Y}_h{}^{(2)}, \ldots, \mathcal{Y}_h{}^{(i)}, \ldots, \mathcal{Y}_h{}^{|\mathcal{D}_U|}\}$, where $\mathcal{Y}_h{}^{(i)} = \{y_h{}^{(1)}, y_h{}^{(2)}, .., y_h{}^{(M)}\}$ is a set of different translations for x_i in target language t.

2.1.2 Hypotheses Selection

The selector f_D takes $\mathcal{H}$ as input and produces $\widetilde{\mathcal{H}} = \{\widetilde{\mathcal{Y}}_h^{(i)}\}_{i=1}^{|\mathcal{D}_U|}$, in which $\widetilde{\mathcal{Y}}_h^{(i)}$ is the subset of $\mathcal{Y}_h^{(i)}$ that passes the selection criteria $\mathcal{F}$, *i.e.*, $\widetilde{\mathcal{Y}}_h^{(i)} = \mathcal{F}(\mathcal{Y}_h^{(i)})$. If $\widetilde{\mathcal{Y}}_h^{(i)}$ contains more than one hypothesis, only one is kept, as our preliminary experiments confirm that keeping more than one hypothesis for each input does not yield improved performance [1]. We pair all the $y_h^{(i)}$ with the input corresponding input code $x^{(i)}$ to acquire pseudo parallel dataset $\mathcal{D}_S = \{(x, y_h)^{(l)}\}_{l=1}^{|\mathcal{D}_S|}$. In practice, we rely on cross-lingual static code analysis and compilation as selection criteria $\mathcal{F}$ for the hypotheses.

[1] If $\widetilde{\mathcal{Y}}_h^{(i)}$ is empty, it will be discarded.

Cross-Lingual Static Analysis. To ensure that the selected hypotheses have high alignment quality with the input code, we use cross-lingual static analysis to compare the key information of both the input code and all the hypotheses. Static code analysis is a technique used to analyze source code without executing the program. One way to perform static code analysis is through the use of an abstract syntax tree (AST). An AST is a tree-like data structure that represents the structure of a program's source code. It captures the high-level structure of the code and the relationships between its elements, enabling a deeper understanding of the code beyond the sequence level. Figure 2 shows an example AST generated from a Java function.

Specifically, we compare the number of functions, and after matching each pair of functions from the output with the input, we check whether the return types are equivalent, and if the parameter lists match in terms of the number of parameters and the type of each parameter. For non-typed languages such as Python, we skip the type part and only compare the number of functions and the parameter list of each function. Passing the cross-lingual static analysis is a strong indicator of the alignment quality of the hypotheses to the input, which helps in selecting the best hypotheses.

Compilation Filtering. We additionally leverage compilation to filter out hypotheses that may contain errors. Specifically, we compile the generated code using the target compiler and check for any compilation errors. Any hypothesis that fails to compile is discarded. This step further improves the quality of the selected hypotheses by ensuring that they are syntactically correct and can be compiled successfully.

2.2 Alignment-Ascending Curriculum Learning

By pairing the hypotheses with their corresponding inputs, we obtain multiple synthetic parallel code datasets at different stages of the generation process. Without the selector, the generation is reduced to plain back-translation. We denote the unfiltered synthetic parallel data from the unfiltered hypotheses, as BT data. Similarly, we denote the synthetic parallel data from cross-lingual static analysis and compilation filtering as STAT and COMP, respectively. In addition, we denote the subset of hypotheses that pass both criteria, static analysis and compilation, as AND data. We adopt a curriculum learning approach to train our code translation model, strategically leveraging the quality of the data at different stages. Our curriculum consists of multiple training phases, progressively incorporating different types of data. We first train with the unfiltered synthetic parallel data, allowing the model to grasp the basic translation patterns. Next, we introduce the cross-lingual static analysis filtered data, which helps refine the model's understanding of language-specific code idioms and improve translation accuracy. Subsequently, we integrate the compilation filtered data, which further enhances the model's ability to generate syntactically correct translations. The curriculum then advances to utilize the intersection of both filtered datasets, combining the benefits of both data sources. We then introduce snippet-level annotated data to enhance translation performance in specific code segments. Finally, we conclude by training with function-level annotated data, enabling the model to capture higher-level structural patterns and produce more coherent translations. By following this carefully designed curriculum, MIRACLE not only benefits from exposure to a diverse range of training data but also progressively refines its translation quality and alignment, leading to improved performance and robustness.

3 Experiments

Datasets. We make use of the annotated CoST dataset from (Zhu et al., 2022) to support snippet training and execution-based evaluation. The CoST dataset contains parallel code aligned at both program and snippet levels. To support execution-based evaluation, we execute all programs in CoST and remove the ones that throw run-time errors and the ones with empty execution output. We refer to the resulting dataset as ECoST (Execution-based CoST). ECoST has approximately $1,000$ function-level training instances for C++, Java, and Python, and 150 for C. We employ a train/validation/test split ratio of approximately 70:5:25. To support snippet and function-level training, we extract the functions from ECoST through AST parsing[2] to get both snippet-level and function-level parallel data ($\mathcal{D}_{L_s}$ and $\mathcal{D}_L$), which we refer to as ECoST-snippet and ECoST-function.

Synthetic Parallel Code Generation. We use the CodeNet dataset (Puri et al., 2021) as the monolingual code data ($\mathcal{D}_U$) for parallel code generation.

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

Input (Java)	Synthetic Output (C++)	Input (Python)	Synthetic Output (C++)

static void show(int [ ] pdata){
StringBuffer stb = new StringBuffer( ) ;
stb.append (pdata [0]) ;
for (int a = 1; a < pdata.length; a ++) {
stb.append("" + pdata[a]);
}
System.out.println(stb.toString());
}
	void show(vector<int>& pdata) {
	string stb = "" ;
	stb += pdata [0];
	for (int a = 1; a < pdata.size( ); a++) {
	stb += "" + pdata[a];
	}
	cout << stb << endl;
	}
		def f (a , b) :
		p , q = a , b
		while q != 0 :
		p , q = q , p % q
		return a / p * b
			float f (float a , float b) {
			int p = a , q = b ;
			while (q != 0) {
			p = q ;
			q = p % q ;
			}
			return a / p * b ;
			}

public static long gcd (long a , long b){
long c = a % b ;
if (c == 0) {
return b ;
}
else {
return gcd (b , c) ;
}
}
	long gcd (long a , long b) {
	long c = a % b ;
	if (c == 07)
	return b ;
	else
	return gcd (b , c) ;
	}
		def check (n , array) :
		if 1.1 <= n :
		array [0] += 1
		elif 0.6 <= n < 1.1 :
		array [1] += 1
		elif 0.2 <= n < 0.6 :
		array [2] += 1
		else :
		array [3] += 1
			void check (int n , int array [ ]) {
			if (1.0 <= n)
			array [0] ++ ;
			else if (1.0 <= n)
			array [1] ++ ;
			else if (0.2 <= n)
			array [2] ++ ;
			else array [3] ++ ;
			}

Figure 3: Synthetic parallel code examples, with PLBART (Ahmad et al., 2021a) as generator. The synthetic parallel data demonstrates great alignment quality, with minor noise in some cases.

CODENET is a large-scale dataset containing 13M programs spanning 55 languages. The programs in CODENET originate from code submissions to online judge of programming problems. We select the "Accepted" submissions (*i.e.*, submissions that pass the online judge) in 4 languages, C++, Java, Python and C, from around $1,600$ problems. After some quality filtering, we get approximately $87,000$ examples. We experiment with two different models as the generator model, PLBART (Ahmad et al., 2021a) and CodeT5 (Wang et al., 2021). The monolingual CODENET data are used as inputs to the generators to obtain the hypotheses through tempered sampling with a temperature of 0.5 and sample size M set to 10. We then get the synthetic parallel code through selection by static analysis and compilation ($\mathcal{F}$).

Baselines and Evaluation Metrics. We compare against five advanced code translation models. CodeBERT (Feng et al., 2020), PLBART (Ahmad et al., 2021a), and CodeT5 (Wang et al., 2021) are programming language models pre-trained with self-supervised learning techniques on large-scale open-source code datasets. These models can perform code translation as a downstream task after fine-tuning on parallel code data. TransCoder (Roziere et al., 2020) is an unsupervised code translation model that relied on back-translation for data augmentation. TransCoder-ST (Roziere et al., 2021b) improves TransCoder by leveraging unit testing to generate parallel code data. After generating the synthetic parallel code, we train code translation models using the generated data and evaluate their performances. CodeBERT, PLBART and CodeT5 need fine-tuning to perform code translation, therefore they are fine-tuned on ECOST with both snippet-level and function-level data. On the other hand, TransCoder and TransCoder-ST do not need fine-tuning as they are unsupervised methods. All models are evaluated on ECOST test set. CodeBLEU(Ren et al., 2020) is a weighted sum of n-gram matching, AST matching, and data flow matching between source and target programs. Computation Accuracy (CA) (Roziere et al., 2020) is a new metric introduced in TransCoder that measures if the hypothesis has the same execution output as the reference. We use CA@1 for all the evaluations. Model training details are included in the Appendix A.

4 Results and Analysis

We evaluate two variations of our method, MIRACLE-PLBART and MIRACLE-CodeT5, by performing parallel code generation with PLBART and CodeT5 as generators and curriculum learning with their generated data respectively. The generated parallel code data is referred to as MIRACLE-function. We focus on two aspects, generated data quality and improvements in code translation performance.

4.1 Quality of the Synthetic Parallel Code

Statistics of MIRACLE-function. With $86,972$ monolingual code as input, we manage to generate $516,142$ and $529,108$ synthetic parallel code pairs in 6 language pairs from PLBART and CodeT5, respectively. Table 1 shows the statistics of the synthetic parallel code data generated by PLBART. Note that the datasets resulting from static analysis and compilation are not subsets of back-translation, because for back-translation we randomly pick a hypothesis from the 10 sampled hypotheses, and

PLBART	Number of Pairs						Selection Rate					
Selector	C++ – Java	C++ – Py	C++ – C	Java – Py	Java – C	Py – C	C++ – Java	C++ – Py	C++ – C	Java – Py	Java – C	Py – C
Back Translation (BT)	47540	63637	49550	37233	22919	39231	1	1	1	1	1	1
Static Analysis (STAT)	25211	58157	14945	31228	13059	33882	0.53	0.91	0.30	0.84	0.57	0.86
Compilation (COMP)	15258	36224	1893	13525	1562	11088	0.32	0.57	0.04	0.36	0.07	0.28
SA & Compilation (AND)	9278	34733	1200	12104	1313	10730	0.20	0.55	0.02	0.33	0.06	0.27

Table 1: Statistics of MIRACLE-function, with PLBART (Ahmad et al., 2021a) as generator. Due to page limit, statistics for CodeT5 (Wang et al., 2021) generated data are included in the Appendix A. SA & Compilation refers to the intersection of the Static Analysis and Compilation selections.

for static analysis and compilation we select the hypothesis from the ones that pass the selection criteria. From the selection rate, we can observe that static analysis is the most lenient to Python, as it is a weakly-typed language. Compilation has the least selection rate on C. This is due to data scarcity as the generator has poor performance on C due to being trained with less than 150 examples.

Qualitative Analysis. We further perform qualitative analysis and manually inspect samples of the generated data. Table 3 illustrates four examples from the synthetic parallel code, with two in Java – C++, and two in Python – C++. The Java and Python codes are the monolingual input from CodeNet, and the C++ codes are the synthetic codes. The generated code snippets are in good alignment with their corresponding inputs, with correct mapping of types, data structures, and syntax. Note that the synthetic codes still contain some noise. However, Table 2 and 3 results indicate that it does not impede the effectiveness of the synthetic code in improving code translation performance.

4.2 Performances in Code Translation

Comparison with Baseline Models. Table 2 shows the CodeBLEU and Computation Accuracy performance on C++, Java, and Python of the baseline models and MIRACLE-PLBART and MIRACLE-CodeT5. In terms of CodeBLEU, both MIRACLE models outperform all baselines, with MIRACLE-CodeT5 surpassing the best baseline performance by 8% on Python – C++ and Java – Python translation. In terms of Computation Accuracy, MIRACLE-CodeT5 outperforms the best baseline performance by 5% on Python – C++ and C++ – Java, 6% on C++-Python, and 8% on Python-Java. Moreover, both MIRACLE models outperform their respective generator models on all the language pairs and both metrics by a wide margin. Compared to CodeT5, MIRACLE-CodeT5's Computation Accuracy on Python – C++ and Python – Java improves by 20%, and on Java – Python and C++ – Python the improvements are 25% and 30%, respectively.

Performance on Low-resource Languages. In ECoST, C only has less than 150 parallel code pairs with each language, making it suitable for evaluating in more challenging low-resource language settings. As shown in Table 1, the compilation rate is the lowest when C is involved, as the generator is not able to generate high-quality data when the training data of C is significantly less. Table 3 shows the performance of the two implementations of MIRACLE and their respective generators. For PLBART, MIRACLE improves the CodeBLEU by up to 40% and improves the Computation Accuracy (CA@1) by up to 43%. This shows that the augmentation of parallel code generation works well in low-resource language settings, where the generator's performance is weak. For CodeT5, the improvement in CA@1 is up to 23%.

Analysis of Alignment-Ascending Curriculum Table 5 presents the datasets employed in curriculum learning and their acquisition methods. To assess the impact of the quality, volume, and order of the datasets in the alignment-ascending curriculum, we train models with different variations of the curriculum and compare their Computation Accuracy, as detailed in Table 4. Initially, a base model is trained solely on the annotated dataset ECoST-function, where its modest size yields limited performances. Incorporating ECoST-snippet markedly enhances model performance, underscoring the value of snippet-based training. Adding the high-quality synthetic data, AND, further improves the performance. Similarly, the integration of unfiltered noisy data, BT, also boosts the performance. However, neither AND nor BT alone reaches the efficacy of MIRACLE, highlighting the critical role of both data quality and volume. Reversing the order of the alignment-ascending curriculum to AND+COMP+STAT+BT+Snippet+Function causes the performance to drop significantly compared to MIRACLE, emphasizing the importance

	CodeBLEU						Computation Accuracy					
Model	Java – C++	Py – C++	C++-Java	Py – Java	C++ – Py	Java – Py	Java – C++	Py – C++	C++ – Java	Py – Java	C++ –Py	Java – Py
CodeBERT	61.75	50.18	29.71	42.21	46.99	46.69	13.44	4.82	10.22	3.93	6.33	5.74
PLBART	71.39	66.62	71.27	64.76	62.05	60.62	25.54	24.40	27.15	23.87	32.23	32.33
CodeT5	72.76	64.99	72.13	64.26	59.16	61.25	37.63	19.28	41.13	23.87	20.78	24.77
Trancoder	72.54	66.47	70.36	63.61	56.29	55.29	49.73	25.60	40.86	22.36	41.87	46.22
Trancoder-ST	71.47	61.28	70.96	64.81	58.85	57.70	**51.08**	36.14	44.09	35.35	43.98	**51.96**
MIRACLE-PLBART	74.55	68.43	72.90	67.14	63.09	63.47	41.94	35.24	40.05	33.84	38.55	41.09
MIRACLE-CodeT5	**74.94**	**69.25**	**74.85**	**69.64**	**65.10**	**65.95**	**51.08**	**41.87**	**49.19**	**43.20**	**50.00**	49.55

Table 2: Performance comparison of two implementations of MIRACLE with PLBART and CodeT5 against baseline approaches. The metrics used for comparison are CodeBLEU and Computation Accuracy (CA@1). Across both measures, MIRACLE outperforms the baseline approaches, demonstrating its effectiveness in code translation.

	CodeBLEU						Computation Accuracy					
Model	C++ – C	Java–C	Python – C	C – C++	C–Java	C – Python	C++ – C	Java – C	Python – C	C – C++	C – Java	C – Python
PLBART	40.66	56.85	43.66	42.77	32.49	52.98	2.60	0	1.56	5.19	0	14.06
MIRACLE-PLBART	**79.08**	**72.37**	**61.73**	**80.34**	**68.79**	**61.92**	**33.77**	**28.77**	**17.19**	**48.05**	**23.29**	**28.12**
CodeT5	82.06	74.16	62.25	80.04	71.25	61.06	66.23	47.95	25.00	64.94	39.73	28.12
MIRACLE-CodeT5	**82.26**	**74.59**	**63.87**	**81.24**	**74.21**	**66.65**	**68.83**	**56.16**	**31.25**	**64.94**	**45.21**	**51.56**

Table 3: Performance comparison before and after applying MIRACLE on low-resource language C. The results show substantial performance improvements across all measures after the application of our method, indicating the effectiveness of MIRACLE on low-resource languages.

Curriculum	**Data Volume**	Java – C++	Py – C++	C++ – Java	Py – Java	C++ – Py	Java – Py
Function	3,326	0.81	4.52	1.88	3.63	16.87	16.62
Snippet+Function	35,144	25.54	24.4	27.15	23.87	32.23	32.33
AND+Snippet+Function	104,502	34.68	34.64	33.06	32.93	36.45	37.16
BT+Snippet+Function	295,254	38.98	34.94	37.1	30.21	35.54	39.58
AND+COMP+STAT+BT+Snippet+Function	551,286	38.98	32.23	37.63	**33.84**	35.84	39.58
BT+STAT+COMP+AND+Snippet+Function (**MIRACLE**)	551,286	**41.94**	**35.24**	**40.05**	**33.84**	**38.55**	**41.09**

Table 4: Comparison of variations of curriculum. Data Volume refers to the number of parallel codes. The base model is PLBART. All results are measured in Computation Accuracy. Results demonstrate the effectiveness of alignment-enhancing curriculum learning.

Data	**Type**	**Volume**	**Source**
BT	Synthetic	260110	Back Translation
STAT	Synthetic	176482	Static Analysis
COMP	Synthetic	79550	Compilation
AND	Synthetic	69358	Static Analysis & Compilation
Snippet	Annotated	31818	ECoST
Function	Annotated	3326	ECoST

Table 5: Datasets for Alignment-Ascending curriculum learning. Volume refers to number of parallel codes.

of the order of the curriculum. Interestingly, this inverted curriculum aligns closely in performance with BT+Snippet+Function, likely due to the larger volume of the BT dataset overpowering the effect of the previous datasets.

Qualitative Analysis. Figure 4 shows examples of various model translations and their execution outputs given the same input code. The first column corresponds to the code used as input in the source language, and the last column corresponds to the ground truth translation in the target language. All examples are from the ECoST test set. We compare MIRACLE-CodeT5 with two other baselines, TransCoder-ST and CodeT5. In the first two examples, we observe that both baselines demonstrate the "shallow translation" problem. In the C++ – Python example, both TransCoder-ST and CodeT5 directly copy from the input code. While `min_element` is a valid built-in function defined in header `<algorithm>` in C++, it does not exist in Python, resulting in compilation errors for both baselines. TransCoder-ST also exhibits an inability to translate multiple functions at once. In the Python – Java example, both TransCoder-ST and CodeT5 translate the keyword `"not"` in Python to `"!"` in Java. However, the operator `"!"` cannot be used when the operand is an integer. By translating at the token level, these baselines fail to take context into consideration, causing run-time errors. In both cases, MIRACLE-CodeT5 can translate the function calls and statements from the source lan-

Input (C++)	MIRACLE (Python)	Transcoder-ST (Python)	CodeT5 (Python)	Groundtruth (Python)

int getMin(int arr[], int n) {
return *min_element(arr, arr + n); }
int getMax(int arr[], int n) {
return *max_element(arr, arr + n); }
	def getMin (arr , n) :
	return min (arr)
	def getMax (arr , n) :
	return max (arr)
		def getMin (arr , n) :
		return * min_element (arr , n)
			def getMin (arr , n) :
			return min_element (arr, arr + n)
			def getMax (arr , n) :
			return max_element (arr, arr + n
			)
				def getMin(arr,n):
				return min(arr)
				def getMax(arr,n):
				return max(arr)

Test Code (Python)	**Execution Output**			
arr = [12,1234,45,67,1] n = len(arr) getMin(arr, n); getMax(arr, n)	1 1234	SyntaxError: can't use starred expression here	NameError: name 'min_element' is not defined	1 1234

Input (Python)	MIRACLE (Java)	Transcoder-ST (Java)	CodeT5 (Java)	Groundtruth (Java)

def nextPowerOf2(n):
count = 0;
if (n and not (n & (n - 1))):
return n
while(n != 0):
n >>= 1
count += 1
return 1 << count;
	static int nextPowerOf2 (int n) {
	int count = 0 ;
	if (n != 0 && (n & (n - 1)) == 0)
	return n ;
	while (n != 0) {
	n >>= 1 ;
	count ++ ;
	}
	return (1 << count) ;
	}
		public static int nextPowerOf2 (int n
		) {
		int count = 0 ;
		if ((n != 0) && (! (n & (n - 1))
		)) {
		return n ; }
		while ((n != 0)) {
		n >>= 1 ;
		count ++ ;
		}
		return 1 << count ;
		}
			static int nextPowerOf2 (int n) {
			int count = 0 ;
			if (n != 0 && ! (n & (n - 1)))
			return n ;
			while (n != 0) {
			n >>= 1 ;
			count ++ ;
			}
			return (1 << count) ;
			}
				static int nextPowerOf2(int n) {
				int count = 0;
				if (n > 0 && (n & (n - 1)) == 0)
				return n;
				while(n != 0) {
				n >>= 1;
				count += 1;
				}
				return 1 << count;
				}

Test Code (Java)	**Execution Output**			

public static void main(String args[]) {
int n = 0;
System.out.println(nextPowerOf2(n));
}
	1	error: bad operand type int for unary operator '!'	error: bad operand type int for unary operator '!'	1

Input (Java)	MIRACLE (Python)	Transcoder-ST (Python)	CodeT5 (Python)	Groundtruth (Python)

static int findTrailingZeros(int n) {
int count = 0;
for (int i = 5; n / i >= 1; i *= 5)
count += n / i;
return count;
}
	def findTrailingZeros (n) :
	count = 0
	i = 5
	while n // i >= 1 :
	count += n // i
	i *= 5
	return count
		def findTrailingZeros (n) :
		count = 0
		for i in range (5 , n // i + 1 , 5) :
		count += n // i
		return count
			def findTrailingZeros (n) :
			count = 0
			for i in range (5) :
			count += n // i
			return count
				def findTrailingZeros(n):
				count = 0
				while(n >= 5):
				n //= 5
				count += n
				return count

Test Code (Python)	**Execution Output**			
n = 100 print("Count of trailing 0s " + "in 100! is", findTrailingZeros(n))	Count of trailing 0s in 100! is 24	UnboundLocalError: local variable 'i' referenced before assignment	ZeroDivisionError: integer division or modulo by zero	Count of trailing 0s in 100! is 24

Figure 4: Qualitative translation results from MIRACLE and baseline methods with the same input. In all three examples, the baselines' results exhibit the "Shallow Translation" problem, where code snippets are directly copied or translated token by token from the source language, causing compilation and run-time errors in the target language. MIRACLE's translation shows its strong ability to correctly align the syntax and APIs across different languages.

guage to the target language correctly. In the Java – Python example, both baselines fail at translating a complex `for` loop, while MIRACLE correctly translates this in a different way from the ground truth, showing a strong capability of understanding the input code and mapping it into a different language.

5 Conclusion

In this paper, we introduce MIRACLE, a semi-supervised approach utilizes static analysis and compilation to generate synthetic parallel code datasets with enhanced alignment, and improves code translation through curriculum learning on code datasets with ascending alignment levels. We evaluate the performance of MIRACLE through extensive experiments conducted on multiple languages and models. The proposed alignment-ascending curriculum learning significantly improves the computation accuracy of code translation, outperforming state-of-the-art baselines by a significant margin. Notably, our method achieves remarkable gains in C translations even with a limited number of annotated training instances. Our work showcases the importance of parallel code data with good alignment quality and the effectiveness of alignment-ascending curriculum learning in enhancing code translation capabilities. Future work can extend to more tasks that benefit from large amount of parallel data.

6 Limitations and Future Work

Despite the promising results and contributions, MIRACLE relies heavily on the generation of parallel code data and does not take into account other types of information that may be useful for code translation, such as comments or documentation. Incorporating such information into the generation process could potentially further improve the quality of the generated data. Moreover, our evaluation is mainly focused on execution-based metrics, which measure the quality of the generated code based on its ability to execute correctly. While these metrics are important, they do not capture other aspects of code quality, such as readability, maintainability, or style. Future work could explore the development of metrics that capture these aspects of code quality.

References

David H Ackley, Geoffrey E Hinton, and Terrence J Sejnowski. 1985. A learning algorithm for boltzmann machines. *Cognitive science*, 9(1):147–169.

Mayank Agarwal, Kartik Talamadupula, Fernando Martinez, Stephanie Houde, Michael Muller, John Richards, Steven I Ross, and Justin D Weisz. 2021. Using document similarity methods to create parallel datasets for code translation. *arXiv preprint arXiv:2110.05423*.

Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021a. 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*, pages 2655–2668.

Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2022. Summarize and generate to back-translate: Unsupervised translation of programming languages. *arXiv preprint arXiv:2205.11116*.

Wasi Uddin Ahmad, Md Golam Rahman Tushar, Saikat Chakraborty, and Kai-Wei Chang. 2021b. Avatar: A parallel corpus for java-python program translation. *arXiv preprint arXiv:2108.11590*.

Mikel Artetxe, Gorka Labaka, and Eneko Agirre. 2019. An effective approach to unsupervised machine translation. *arXiv preprint arXiv:1902.01313*.

Mikel Artetxe, Gorka Labaka, Eneko Agirre, and Kyunghyun Cho. 2017. Unsupervised neural machine translation. *arXiv preprint arXiv:1710.11041*.

Xinyun Chen, Chang Liu, and Dawn Song. 2018. Tree-to-tree neural networks for program translation. In *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. Bert: Pre-training of deep bidirectional transformers for language understanding. In *NAACL-HLT (1)*.

Sergey Edunov, Myle Ott, Michael Auli, and David Grangier. 2018. Understanding back-translation at scale. *arXiv preprint arXiv:1808.09381*.

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

Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. 2022. Incoder: A generative model for code infilling and synthesis. *arXiv preprint arXiv:2204.05999*.

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*.

Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. 2015. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*.

Yufan Huang, Mengnan Qi, Yongqiang Yao, Maoquan Wang, Bin Gu, Colin Clement, and Neel Sundaresan. 2023. Program translation via code distillation. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 10903–10914.

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

Svetoslav Karaivanov, Veselin Raychev, and Martin Vechev. 2014. Phrase-based statistical translation of programming languages. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, pages 173–184.

Kusum Kusum, Abrar Ahmed, Bhuvana C, and V. Vivek. 2022. Unsupervised translation of programming language - a survey paper. In *2022 4th International Conference on Advances in Computing, Communication Control and Networking (ICAC3N)*, pages 384–388.

Guillaume Lample and Alexis Conneau. 2019. Cross-lingual language model pretraining. *arXiv e-prints*, pages arXiv–1901.

Guillaume Lample, Alexis Conneau, Ludovic Denoyer, and Marc'Aurelio Ranzato. 2017. Unsupervised machine translation using monolingual corpora only. *arXiv preprint arXiv:1711.00043*.

Guillaume Lample, Alexis Conneau, Ludovic Denoyer, and Marc'Aurelio Ranzato. 2018. Unsupervised machine translation using monolingual corpora only. In *International Conference on Learning Representations*.

Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Veselin Stoyanov, and Luke Zettlemoyer. 2020. Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. In *Proceedings of the 58th Annual Meeting of* *the Association for Computational Linguistics*, pages 7871–7880.

Yinhan Liu, Jiatao Gu, Naman Goyal, Xian Li, Sergey Edunov, Marjan Ghazvininejad, Mike Lewis, and Luke Zettlemoyer. 2020. Multilingual denoising pre-training for neural machine translation. *Transactions of the Association for Computational Linguistics*, 8:726–742.

Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. Roberta: A robustly optimized bert pretraining approach.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. 2021. Codexglue: A machine learning benchmark dataset for code understanding and generation. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 1)*.

Anh Tuan Nguyen, Tung Thanh Nguyen, and Tien N Nguyen. 2013. Lexical statistical machine translation for language migration. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 651–654.

Anh Tuan Nguyen, Tung Thanh Nguyen, and Tien N Nguyen. 2015. Divide-and-conquer approach for multi-phase statistical migration for source code (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 585–596. IEEE.

Rangeet Pan, Ali Reza Ibrahimzada, Rahul Krishna, Divya Sankar, Lambert Pouguem Wassi, Michele Merler, Boris Sobolev, Raju Pavuluri, Saurabh Sinha, and Reyhaneh Jabbarvand. 2023. Understanding the effectiveness of large language models in code translation. *arXiv preprint arXiv:2308.03109*.

Ruchir Puri, David S Kung, Geert Janssen, Wei Zhang, Giacomo Domeniconi, Vladmir Zolotov, Julian Dolby, Jie Chen, Mihir Choudhury, Lindsey Decker, et al. 2021. Project codenet: A large-scale ai for code dataset for learning a diversity of coding tasks. *arXiv preprint arXiv:2105.12655*.

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. 2020. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of Machine Learning Research*, 21(140):1–67.

Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. 2020. Codebleu: a method for automatic evaluation of code synthesis. *arXiv preprint arXiv:2009.10297*.

Baptiste Roziere, Marie-Anne Lachaux, Lowik Chanussot, and Guillaume Lample. 2020. Unsupervised translation of programming languages. In *NeurIPS*.

Baptiste Roziere, Marie-Anne Lachaux, Marc Szafraniec, and Guillaume Lample. 2021a. Dobf: A deobfuscation pre-training objective for programming languages. *arXiv preprint arXiv:2102.07492*.

Baptiste Roziere, Jie Zhang, Francois Charton, Mark Harman, Gabriel Synnaeve, and Guillaume Lample. 2021b. Leveraging automated unit tests for unsupervised code translation. In *International Conference on Learning Representations*.

Vikash Sehwag, Saeed Mahloujifar, Tinashe Handina, Sihui Dai, Chong Xiang, Mung Chiang, and Prateek Mittal. Robust learning meets generative models: Can proxy distributions improve adversarial robustness? In *International Conference on Learning Representations*.

Marc Szafraniec, Baptiste Roziere, Hugh Leather Francois Charton, Patrick Labatut, and Gabriel Synnaeve. 2022. Code translation with compiler representations. *arXiv preprint arXiv:2207.03578*.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 8696–8708.

Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Lei Shen, Zihan Wang, Andi Wang, Yang Li, et al. 2023. Codegeex: A pre-trained model for code generation with multilingual benchmarking on humaneval-x. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 5673–5684.

Ming Zhu, Karthik Suresh, and Chandan K Reddy. 2022. Multilingual code snippets training for program translation. In *36th AAAI Conference on Artificial Intelligence (AAAI)*.

A Appendix

A.1 Related Work

Parallel Code Data. Parallel code data refers to code pairs from different programming languages that are functionally equivalent and bug-free. Existing datasets are characterized by relatively high alignment but are limited in size and supported languages. For example, CodeXGLUE (Lu et al., 2021) constructed a Java – C# translation dataset by matching function names from open-source repositories. MuST-PT (Zhu et al., 2022) introduced a program translation dataset CoST, with snippet-level alignment that supports 7 programming languages. CoST was collected from the coding tutorial website GeeksforGeeks[3], where each coding problem is provided with solutions in up to 7 languages, with each in similar structure and comments. AVATAR (Ahmad et al., 2021b) only supports the translation between Java and Python. Other kinds of datasets are usually significantly larger and support a wider range of languages, but the alignment quality is low. These are usually collected from competitive online code judgments. Given a coding problem, users can submit their solutions in various supported languages and get judged based on online tests. The user-contributed solutions to the same problems are collected as parallel code in different languages. For example, Google Code Jam and Project CodeNet (Puri et al., 2021) were both collected in this manner. However, due to the diverse backgrounds and the large number of users, the solutions for the same problem have wide discrepancies in distribution across different languages, which lowers alignment quality.

Neural Code Translation. Recent advances in machine learning, especially in self-supervised learning techniques, have benefited a wide range of tasks (Vaswani et al., 2017; Liu et al., 2019; Lample and Conneau, 2019; Liu et al., 2020; Sehwag et al.). Some techniques from NLP were transferred to programming languages and have achieved great success. Similar to BERT (Devlin et al., 2019), CodeBERT (Feng et al., 2020) is a code language model pre-trained on CodeSearchNet (Husain et al., 2019) with Masked Language Modeling (MLM). PLBART (Ahmad et al., 2021a) is pre-trained the same way as BART (Lewis et al., 2020), with Denoising Auto-Encoding (DAE) (Lample et al., 2018) on GitHub data. Although CodeBERT and PLBART are pre-trained on code, they model code the same way as natural language sequences without considering code-specific features. Inspired by T5 (Raffel et al., 2020), CodeT5 (Wang et al., 2021) is pre-trained on CodeSearchNet but with an identifier-aware objective to align more with programming language distributions. All three models use general pre-training to gain programming language intelligence, without optimizing for any specific tasks. They require fine-tuning on task-specific data to perform downstream tasks. TransCoder (Roziere et al., 2020) is an unsupervised code translation model that relies on back-translation to generate pseudo-parallel code data during training. However, back-translation introduces noisy code into the training process, compromising the model's ability to generate high-quality translations. TransCoder-ST (Roziere et al., 2021b) improves TransCoder by adding automated unit tests to filter out invalid translations and reduce noise from the back-translation process. However, obtaining unit tests for different languages is expensive, and running unit tests is unscalable for a large amount of code data. MuST-PT (Zhu et al., 2022) leverages snippet-level DAE and translations for pre-training before fine-tuning on program-level data, which improves code translation performance. However, MuST-PT is less scalable, as it relies solely on a limited amount of finely aligned parallel code for training without utilizing widely available non-parallel code.

[3]https://www.geeksforgeeks.org/

A.2 Implementation Details

All models are trained with a batch size of 16 for 10 epochs, with a learning rate of $5e-5$. Experiments are performed on one NVIDIA A100 GPU with 80G memory. For tempered sampling, we use a sample size of 10 with a fixed temperature of 0.5. For evaluation, we use beam search with a beam size of 5. We use a max sequence length of 200 tokens for both the inputs and outputs.

Preprocessing. For all the program data, we first remove all the comments, docstrings, and empty lines. New lines are replaced with special token `NEW_LINE`. For pre-tokenization, Python is pre-tokenized with a TreeSitter-based tokenizer from TransCoder(Roziere et al., 2020), for better handling of indentations. Other languages are not pre-tokenized. When running experiments, the data will be tokenized again using the corresponding tokenizer of each model.

Function Info Extraction. We rely on AST parsing to extract function information from programs, which are further used for static analysis and execution-based evaluation. An AST is a tree-like data structure that represents the structure of a program's source code. It captures the high-level structure of the code and the relationships between its elements, enabling a deeper understanding of the code beyond the sequence level. To create an AST, the source code is first parsed to identify its syntactic elements, such as keywords, operators, and identifiers. The parser then constructs the AST by assigning each syntactic element to a node in

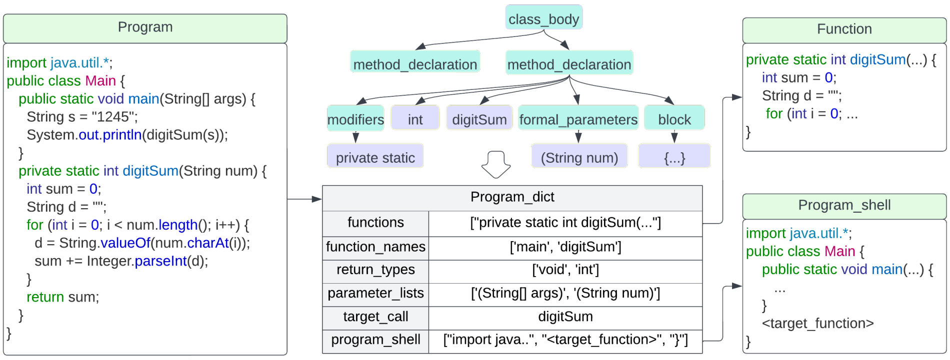

Figure 5: An illustration of function info extraction through AST parsing. Given an input program, we first generate its corresponding AST, and then extract function-related information from AST into program_dict. The tree in the top middle shows an example of AST. After the functions are extracted, the leftover part of the program is called `program_shell`, which can be used for execution-based evaluation later.

the tree. An AST consists of terminal and non-terminal nodes. Terminal nodes are leaf nodes in AST and are part of the source code. Non-terminal nodes are not part of the source code. With the help of AST, we can extract function-related information by matching the corresponding non-terminal nodes in that language, such as `method_declaration`, `method_invocation`, `formal_parameters` etc. One of the most widely used open-source AST parsing tools is TreeSitter[4]. It supports most of the commonly used programming languages. Figure 5 shows an example of a Java program and its AST (parsed by TreeSitter). The blue nodes are non-terminal and the purple nodes are terminal.

Sourcing of Monolingual Code Inputs. CODENET (Puri et al., 2021) is a huge dataset containing 13 million of programs in 55 languages. The programs in CODENET are from code submissions to online judge websites of programming problems. We use CODENET as a source of monolingual code inputs for parallel code generation. We select the "Accepted" submissions (submissions that pass the prescribed tests) in 4 languages, C++, Java, Python, and C, from around 1600 problems, which gives us approximately 1 million programs. To ensure the quality of the input data, we set two filtering criteria: (1) the program should be modularized, which means it should contain at least one function (other than `main()` or `Main()` function), and (2) the program should be bug-free, which means it can be compiled without errors. After applying the two steps of filtering, only around 8% of the programs remain, approximately 87k.

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

Parallel Code Generation. We experiment with two different models as the generator model, PLBART (Ahmad et al., 2021a) and CodeT5 (Wang et al., 2021). The generator models are initialized by first training on the snippet-level data, and then the function-level data from ECOST. We then utilize the monolingual CODENET data as inputs and acquire the hypotheses from the generators through tempered sampling. For cross-lingual static analysis, we extract the function information of both the monolingual inputs and all the hypotheses and compare them. For compilation, we use the compiler of each language to compile all the hypotheses. Since the hypotheses are functions not programs, we pair each of them with a set of common imports in the corresponding language before compilation to avoid dependency errors. For Python, we first try with python2, and subsequently with python3 if python2 returns with an error. The statistics of the selected hypotheses generated by MIRACLE-CodeT5 can be found in Table 6.

Execution-Based Evaluation. ECOST test set is used for all the evaluations. ECOST train set and generated parallel data are used for model training. The train/valid/test split of ECOST is 70:5:25, and the generated parallel dataset is 85:5:10. The statistics of ECOST are shown in Table 7. To evaluate the quality of the generated hypotheses, we employ an execution-based evaluation strategy. By inserting the generated hypothesis of an input function into the `program_shell` of the ground truth program, we execute the modified program and compare its output against the original output. This process allows us to verify whether the hypothesis successfully passes the built-in test cases, thus eval-

CodeT5	Number of Pairs						Selection Rate					
Selector	C++-Java	C++-Py	C++-C	Java-Py	Java-C	Py-C	C++-Java	C++-Py	C++-C	Java-Py	Java-C	Py-C
Back Translation (BT)	47637	64037	49550	37422	22935	39335	1	1	1	1	1	1
Static Analysis (STAT)	25211	58663	14945	31379	13059	34072	0.53	0.92	0.30	0.84	0.57	0.87
Compilation (COMP)	17373	36544	2290	16888	3821	13947	0.36	0.57	0.05	0.45	0.17	0.35
SA & Compilation (AND)	10811	35457	1325	15256	2731	13309	0.23	0.55	0.03	0.41	0.12	0.34

Table 6: Statistics of CODENET-MIRACLE, with CodeT5 (Wang et al., 2021) as generator. SA & Compilation refers to the intersection of the Static Analysis and Compilation selections.

	Function-Level						Snippet-Level					
CoST	C++-Java	C++-Py	C++-C	Java-Py	Java-C	Py-C	C++-Java	C++-Py	C++-C	Java-Py	Java-C	Py-C
Train	1014	947	138	947	146	134	10472	8893	1358	8716	1305	1074
Val	51	46	14	47	14	14	417	324	78	340	78	69
Test	372	332	77	331	73	64	2493	1991	450	1964	422	313

Table 7: Data split and number of parallel code pairs in ECoST.

uating its correctness and suitability. However, the function names in the generated hypotheses might not match the function calls in `program_shell`, causing execution errors. Therefore, through function information extraction, we replace the function name of the hypotheses with the corresponding ground truth function name before each evaluation.

A.3 Broader Impacts

The ability to automatically translate code between programming languages can help software developers port existing codebases from one language to another, allowing them to work with a wider range of tools and frameworks. It can also facilitate collaboration between developers who work with different programming languages. In addition, our work has the potential to reduce the barriers to entry for new developers who want to learn a new programming language. By enabling them to translate code from a language they are familiar with to a new language, they can quickly learn the connections and differences between the two languages, and start working on projects in the new language. Moreover, it also has the potential to create more inclusive software engineering learning environments, which makes computer science more accessible for learners from various backgrounds. However, there are also potential negative impacts of this work, such as the possibility of automated code translation leading to loss of jobs for software developers or increased reliance on automated tools in the software development process.