Semi-Supervised Code Translation Overcoming the Scarcity of Parallel Code Data

Ming Zhu[1], Mohimenul Karim[1], Ismini Lourentzou[1,2], Danfeng (Daphne) Yao[1]
mingzhu@vt.edu,mohimenul@vt.edu,lourent2@illinois.edu,danfeng@vt.edu
[1]Sanghani Center for AI and Data Analytics, Virginia Tech
Blacksburg, VA, USA
[2] University of Illinois at Urbana-Champaign
Champaign, IL, USA

ABSTRACT

Neural code translation is the task of converting source code from one programming language to another. One of the main challenges is the scarcity of parallel code data, which hinders the ability of translation models to learn accurate cross-language alignments. In this paper, we introduce **MIRACLE**, a semi-supervised approach that improves code translation through synthesizing high-quality parallel code data and curriculum learning on code data with ascending alignment levels. MIRACLE leverages static analysis and compilation to generate synthetic parallel code datasets with enhanced quality and alignment to address the challenge of data scarcity. We evaluate the proposed method along with strong baselines including instruction-tuned Large Language Models (LLMs) for code. Our analysis reveals that LLMs pre-trained on open-source code data, regardless of their size, suffer from the "shallow translation" problem. This issue arises when translated code copies keywords, statements, and even code blocks from the source language, leading to compilation and runtime errors. Extensive experiments demonstrate that our method significantly mitigates this issue, enhancing code translation performance across multiple models in C++, Java, Python, and C. Remarkably, MIRACLE outperforms code LLMs that are ten times larger in size. MIRACLE also achieves up to a 43% improvement in C code translation with fewer than 150 annotated examples.

CCS CONCEPTS

• **Computing methodologies → Machine learning**; • **Software and its engineering**;

KEYWORDS

Neural Code Translation, Cross-Language Code Alignment, Semi-Supervised Learning, Curriculum Learning, Static Analysis

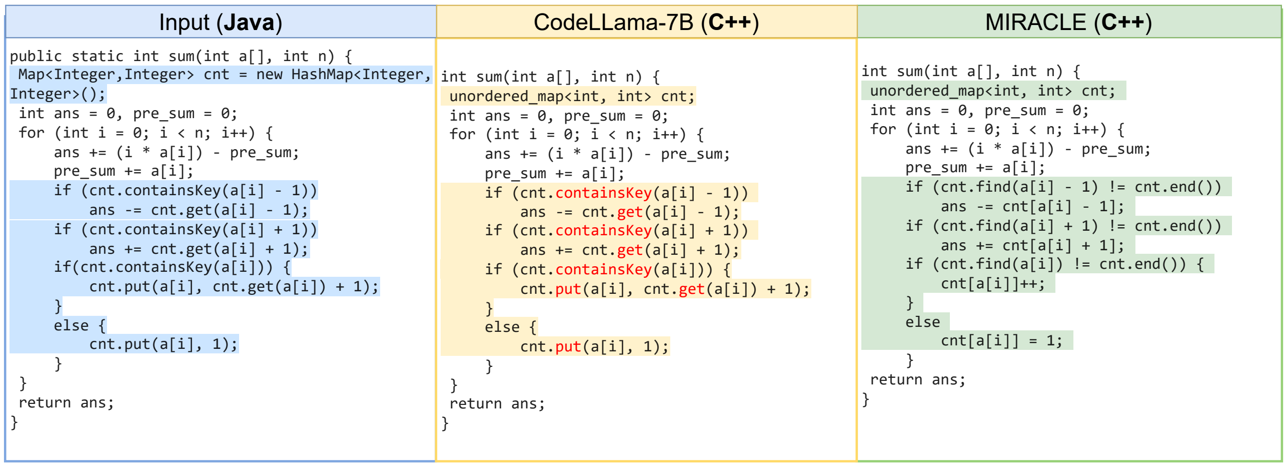
ACM Reference Format:
Ming Zhu[1], Mohimenul Karim[1], Ismini Lourentzou[1,2], Danfeng (Daphne) Yao[1]. 2024. Semi-Supervised Code Translation Overcoming the Scarcity of Parallel Code Data. In *39th IEEE/ACM International Conference on Automated Software Engineering (ASE '24), October 27-November 1, 2024, Sacramento, CA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3691620.3695524

1 INTRODUCTION

Code translation, which involves converting source code written in one programming language (PL) to another, 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 [4, 13, 36, 42, 45]. However, pre-training tasks such as masked language modeling (MLM) and auto-regressive language modeling [10, 12, 14] focus on internal language consistency by predicting missing or subsequent tokens, while code translation demands a deeper semantic understanding of programming language structures. This discrepancy can limit the effectiveness of traditional pre-training in code translation. Large language models (LLMs) have shown significant promise in code translation tasks due to their extensive pre-training on diverse code corpora [34, 41]. However, performance on code tasks varies significantly across LLMs, due to several factors, including the quality of the training data, the inherent complexity of code translation and synthesis tasks, and the lack of proper handling language-specific features such as method overloading and annotation [28].

A critical component in improving code translation is therefore the availability of high-quality parallel code data. 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 [6, 9, 18, 25–27, 45, 46]. To reduce the dependency on parallel data, a separate line of code translation works has explored unsupervised learning. Established techniques from unsupervised neural machine translation (NMT) [7, 8, 21], such as back-translation and denoising auto-encoding, can be applied to code data effectively, achieving promising performances on code translation [3, 5, 11, 16, 19, 35, 38].

Despite the potential to improve generalization, unsupervised learning introduces significant noise, which is particularly harmful to code generation tasks that require precision. A core challenge of unsupervised methods is that they often fail to learn correct mappings of syntax and data structures from one language to another [30], and hence cannot understand functional equivalences between different programming languages, resulting in "shallow translations" where code appears superficially correct but doesn't function as intended. Figure 1 illustrates an example of shallow translation. Moreover, weak translation may lead to cryptographic API, requiring further identification of language-specific irrelevant elements [33].



```java
public static int sum(int a[], int n) {
 Map<Integer,Integer> cnt = new HashMap<Integer,
Integer>();
 int ans = 0, pre_sum = 0;
 for (int i = 0; i < n; i++) {
     ans += (i * a[i]) - pre_sum;
     pre_sum += a[i];
     if (cnt.containsKey(a[i] - 1))
         ans -= cnt.get(a[i] - 1);
     if (cnt.containsKey(a[i] + 1))
         ans += cnt.get(a[i] + 1);
     if(cnt.containsKey(a[i])) {
         cnt.put(a[i], cnt.get(a[i]) + 1);
     }
     else {
         cnt.put(a[i], 1);
     }
 }
 return ans;
}
```

```cpp
int sum(int a[], int n) {
 unordered_map<int, int> cnt;
 int ans = 0, pre_sum = 0;
 for (int i = 0; i < n; i++) {
     ans += (i * a[i]) - pre_sum;
     pre_sum += a[i];
     if (cnt.containsKey(a[i] - 1))
         ans -= cnt.get(a[i] - 1);
     if (cnt.containsKey(a[i] + 1))
         ans += cnt.get(a[i] + 1);
     if (cnt.containsKey(a[i])) {
         cnt.put(a[i], cnt.get(a[i]) + 1);
     }
     else {
         cnt.put(a[i], 1);
     }
 }
 return ans;
}
```

```cpp
int sum(int a[], int n) {
 unordered_map<int, int> cnt;
 int ans = 0, pre_sum = 0;
 for (int i = 0; i < n; i++) {
     ans += (i * a[i]) - pre_sum;
     pre_sum += a[i];
     if (cnt.find(a[i] - 1) != cnt.end())
         ans -= cnt[a[i] - 1];
     if (cnt.find(a[i] + 1) != cnt.end())
         ans += cnt[a[i] + 1];
     if (cnt.find(a[i]) != cnt.end()) {
         cnt[a[i]]++;
     }
     else
         cnt[a[i]] = 1;
     }
 return ans;
}
```

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 CodeLLama-7B-Instruct, and our proposed method MIRACLE (with CodeT5+ of size 770M as generator). The highlighted parts show that CodeLLama-7B-Instruct's translation directly copied code blocks from the input Java code, which contains non-existent methods or grammatically incorrect statements in the target language C++, while MIRACLE was able to correctly convert them in the corresponding C++ grammar.

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. Static analysis and compilation ensure the syntactical correctness and alignment of the synthetic parallel code in a cost-efficient way. Moreover, the proposed alignment-ascending curriculum learning is robust to data noise, effectively mitigating 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.

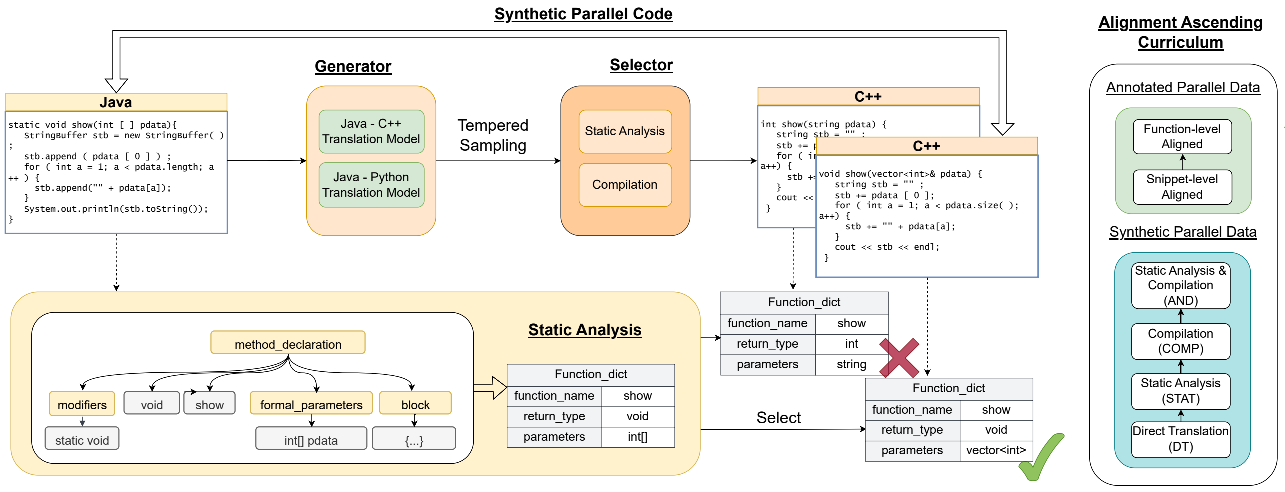
(3) Extensive experiments show that MIRACLE successfully improves code translation performance by up to 30% on C++, Java, and Python, outperforming state-of-the-art baselines on translation between Python and C++ by 5.7%, C++ and Python by 6%, and Python and Java by 8% in execution-based evaluation (CA@1). Notably, our method improves C translations by up to 43% with less than 150 annotated training instances.

(4) We additionally evaluate several large language models and their capability to translate code through instruction. We conduct case studies of both closed-sourced LLMs such as GPT-3.5 [29] and GPT-4 [1], and open-source models such as CodeLLama-Instruct [34] and LLama-2 [39]. Our experiments showcase that open-source LLMs struggle with code involving built-in data structures or bitwise operations.

2 RELATED WORK

Parallel Code Data. Parallel code data refers to code pairs from different programming languages that are functionally equivalent and bug-free. One type of existing datasets is characterized by relatively high alignment but is limited in size and supported languages. For example, CodeXGLUE [25] constructed a Java – C# translation dataset by matching function names from open-source repositories. MuST-PT [46] introduced a program translation dataset CoST, with snippet-level alignment that supports 7 programming languages. CoST was collected from the coding tutorial website GeeksforGeeks[1], where each coding problem is provided with solutions in up to 7 languages, with each in similar structure and comments. AVATAR [6] only supports the translation between Java and Python. Another type of datasets is usually significantly larger in size and supports a wider range of languages, but the alignment quality is low. They are usually collected from competitive online code judgments. Given a coding problem, users can submit their own 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 [31] 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 the quality of the alignment.

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



```
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());
}
```

```
int show(string pdata) {
   string stb = "" ;
   stb += p
   for ( in
a++) {
     stb +=
   }
   cout <<
 }
```

```
void show(vector<int>& pdata) {
   string stb = "" ;
   stb += pdata [ 0 ];
   for ( int a = 1; a < pdata.size( );
a++) {
     stb += "" + pdata[a];
   }
   cout << stb << endl;
 }
```

Function_dict	
function_name	show
return_type	void
parameters	int[]

Function_dict	
function_name	show
return_type	int
parameters	string

Function_dict	
function_name	show
return_type	void
parameters	vector<int>

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.

Neural Code Translation. Recent advances in machine learning, especially in self-supervised learning techniques, have benefited a wide range of tasks [20, 23, 24, 40]. Neural networks were applied for application programming interface (API) completion tasks where next code sequence are given to predict the following API method [43, 44]. Some techniques from NLP were transferred to programming languages and have achieved great success. Similar to BERT [10], CodeBERT [12] is a code language model pre-trained on CodeSearchNet [17] with Masked Language Modeling (MLM). PLBART [4] is pre-trained the same way as BART [22], with Denoising Auto-Encoding (DAE) [21] 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 [32], CodeT5 [42] and CodeT5+ [41] are both pre-trained on CodeSearchNet but with an identifier-aware objective to align more with programming language distributions. CodeT5+ [41] is a family of encoder-decoder large language models (LLMs) for code understanding and generation tasks. These 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 [35] 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 [37] 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 [46] leverages snippet-level DAE and translations for pre-training before fine-tuning on program-level data, which improves code translation performance. Nevertheless, 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. Recent advancements in large language models (LLMs) such as GPT-3.5 [29] and GPT-4 [1], have shown significantly improved capabilities of code translation. These models are not only capable of generating high-quality translations between programming languages but also excel at following instructions, which enhances their usability in various applications. CodeLLama [34] is a family of open-source large language models for code generation and infilling. The models are based on Llama 2 [39] and reaches state-of-the-art performance on several code benchmarks. The instruction-tuned CodeLLama can perform code translation tasks through prompting.

3 METHOD

Code translation models rely on large amounts of parallel data to achieve good performance. Semi-supervised methods generate synthetic parallel data from monolingual data sources but often struggle to maintain alignment quality between source and target languages. To address the data scarcity challenge, our work aims 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 presents an overview of the proposed method.

3.1 Parallel Code Data Generation

MIRACLE 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 and learns to generate a large number of hypotheses for monolingual code data D_U. 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.

3.1.1 Hypotheses Generation. The hypotheses generator f_G is initialized by training on a limited amount of parallel code data, 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 [46], 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. The trained f_G then generates 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 [46]. 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 valid hypotheses with better alignment to the input code and 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 [2, 15]. 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.

3.1.2 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 [2]. 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.

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), a tree-like data structure that represents the structure of a program's source code. AST 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 ensures that selected hypotheses are syntactically correct and can be compiled successfully.

3.2 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 direct translation. We denote the unfiltered synthetic parallel data from the unfiltered hypotheses, as DT (Direct Translation) 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.

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

4 EXPERIMENTS

4.1 Datasets

We make use of the annotated CoST dataset from [46] 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[3] to get both snippet-level and function-level parallel data, which we refer to as ECoST-snippet and ECoST-function.

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[35], 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 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. It supports most of the commonly used programming languages. Figure 3 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.

Execution-Based Evaluation ECoST test set is used for all the evaluations. 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 evaluating 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.

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

4.2 Synthetic Parallel Code Generation

We use the CodeNet dataset [31] as the monolingual code data ($\mathcal{D}_U$) for parallel code generation. 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 quality filtering, we get approximately 87, 000 examples. We experiment with three different models as the generator model, PLBART [4], CodeT5 [42] and CodeT5+ [41]. 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 ($\mathcal{F}$).

4.3 Baselines and Evaluation Metrics

We compare against advanced code translation models of different sizes. PLBART (139M) [4], CodeT5 (220M) [42] and CodeT5+ (770M) [41] are programming language models pre-trained with self-supervised learning techniques on large-scale open-source code datasets. TransCoder (110M) [35] and TransCoder-ST (110M) [37] are unsupervised code translation models. We also compare our method with prompting an open-source instruction-tuned Code Large Language Model (LLM), CodeLLama-7B-Instruct [34].After generating the synthetic parallel code, we train code translation models using the generated data and evaluate their performances. PLBART, CodeT5, and CodeT5+ require fine-tuning to perform code translation, therefore they are fine-tuned on ECoST with both snippet-level and function-level data. TransCoder and TransCoder-ST do not need fine-tuning as they are unsupervised methods. CodeLLama performs the translation task through prompting. All models are evaluated on ECoST test set.

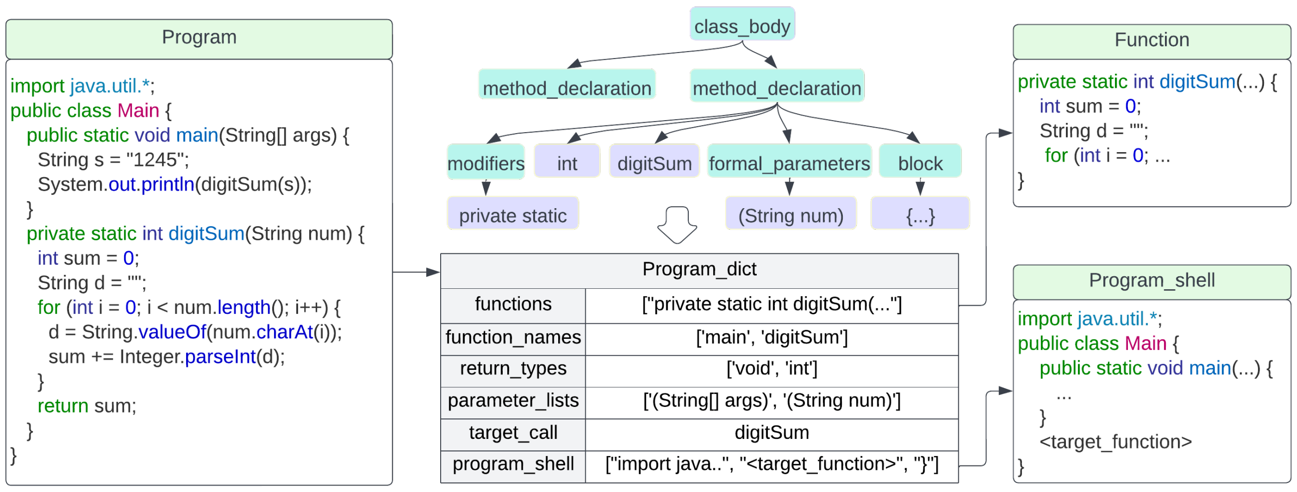
Computation Accuracy (CA) [35] is an execution-based evaluation metric for synthetic code that measures if the hypothesis has the same execution output as the reference. We use CA@1 for all the evaluations.

4.4 Implementation Details

All MIRACLE models are trained with a batch size of 16 for 10 epochs, with a learning rate of $5e-5$. Experiments are performed on 16 NVIDIA A100 GPUs with 80G memory on each. 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 400 tokens for both the inputs and outputs.

5 RESULTS AND ANALYSIS

We evaluate three MIRACLE variations with varying model sizes, MIRACLE-PLBART, MIRACLE-CodeT5 and MIRACLE-CodeT5+, by performing parallel code generation with PLBART, CodeT5, and CodeT5+ (770M) as generators and curriculum learning with their



Program

```
import java.util.*;
public class Main {
  public static void main(String[] args) {
    String s = "1245";
    System.out.println(digitSum(s));
  }
  private static int digitSum(String num) {
    int sum = 0;
    String d = "";
    for (int i = 0; i < num.length(); i++) {
      d = String.valueOf(num.charAt(i));
      sum += Integer.parseInt(d);
    }
    return sum;
  }
}
```

Function

```
private static int digitSum(...) {
  int sum = 0;
  String d = "";
   for (int i = 0; ...
}
```

Program_dict	
functions	["private static int digitSum(..."]
function_names	['main', 'digitSum']
return_types	['void', 'int']
parameter_lists	['(String[] args)', '(String num)']
target_call	digitSum
program_shell	["import java..", "<target_function>", "}"]

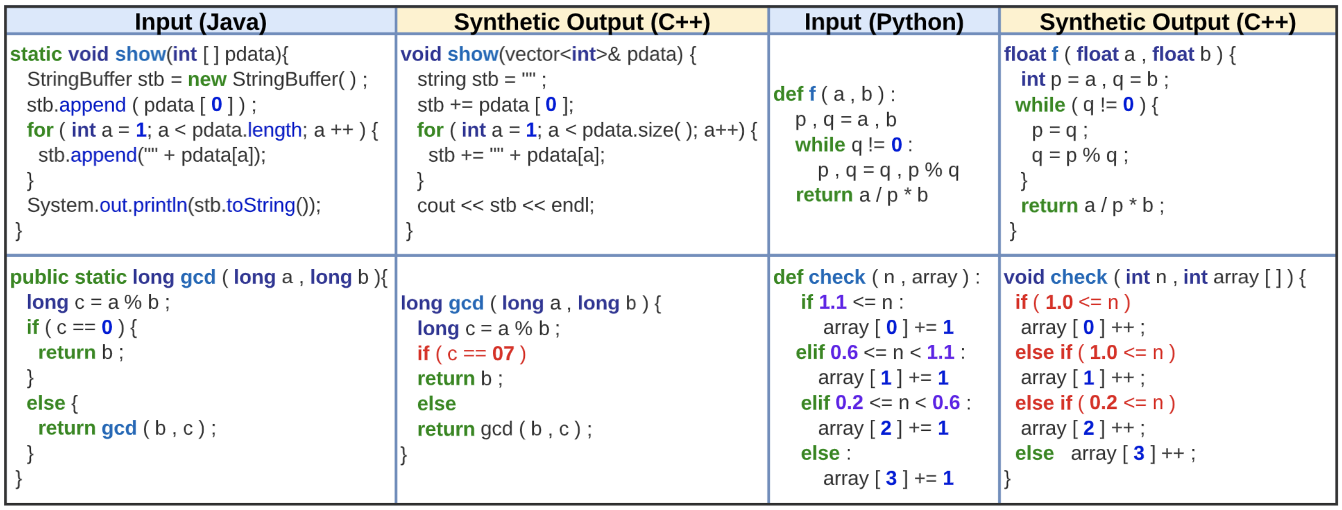
Program_shell

```
import java.util.*;
public class Main {
  public static void main(...) {
    ...
  }
  <target_function>
}
```

Figure 3: Illustration of function info extraction through AST parsing. Given an input program, MIRACLE first generates its corresponding AST and then extracts function-related information from AST into program_dict. The top middle tree shows an example of AST. After the functions are extracted, the leftover part of the program is called `program_shell`, which can later be used for execution-based evaluation.

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
Direct Translation (DT)	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 [4] as generator. SA & Compilation refers to the intersection of the Static Analysis and Compilation selections.


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

Input (Java)

```
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());
}
```

Synthetic Output (C++)

```
void show(vector<int>& pdata) {
  string stb = "" ;
  stb += pdata [ 0 ];
  for ( int a = 1; a < pdata.size( ); a++) {
    stb += "" + pdata[a];
  }
  cout << stb << endl;
}
```

Input (Python)

```
def f ( a , b ) :
  p , q = a , b
  while q != 0 :
    p , q = q , p % q
  return a / p * b
```

Synthetic Output (C++)

```
float f ( float a , float b ) {
  int p = a , q = b ;
  while ( q != 0 ) {
    p = q ;
    q = p % q ;
  }
  return a / p * b ;
}
```

Input (Java)

```
public static long gcd ( long a , long b ){
  long c = a % b ;
  if ( c == 0 ) {
    return b ;
  }
  else {
    return gcd ( b , c ) ;
  }
}
```

Synthetic Output (C++)

```
long gcd ( long a , long b ) {
  long c = a % b ;
  if ( c == 07 )
  return b ;
  else
  return gcd ( b , c ) ;
}
```

Input (Python)

```
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
```

Synthetic Output (C++)

```
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 4: Synthetic parallel code examples, with PLBART [4] as generator. The synthetic parallel data demonstrates great alignment quality, with minor noise in some cases.

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.

5.1 Quality of the Synthetic Parallel Code

Statistics of MIRACLE-function. With 86, 972 monolingual code as input, we generate approximately 1.5 million synthetic parallel code pairs in 6 language pairs from PLBART, CodeT5, and CodeT5+.

Model	Java – C++	Py – C++	C++ – Java	Py – Java	C++ – Py	Java – Py	C++ – C	Java – C	Py – C	C – C++	C –Java	C – Py
PLBART	25.54	24.4	27.15	23.87	32.23	32.33	2.6	0	1.56	5.19	0	14.06
CodeT5	37.63	19.28	41.13	23.87	20.78	24.77	66.23	47.95	25	64.94	39.73	28.12
CodeT5+	63.71	54.22	62.1	50.15	32.83	36.25	84.42	61.64	46.88	84.42	38.36	29.69
Trancoder	49.73	25.60	40.86	22.36	41.87	46.22	-	-	-	-	-	-
Trancoder-ST	51.08	36.14	44.09	35.35	43.98	**51.96**	-	-	-	-	-	-
CodeLLama-7B	47.04	49.7	30.11	46.53	38.86	42.6	62.34	45.21	**51.56**	58.44	42.47	34.38
MIRACLE-PLBART	41.94	35.24	40.05	33.84	38.55	41.09	33.77	28.77	17.19	48.05	23.29	28.12
MIRACLE-CodeT5	51.08	41.87	49.19	43.2	**50**	49.55	68.83	56.16	31.25	64.94	**45.21**	**51.56**
MIRACLE-CodeT5+	**69.35**	**61.75**	**62.37**	**57.1**	41.87	41.39	**85.71**	**73.97**	45.31	**84.42**	43.84	31.25

Table 2: Performance comparison of three implementations of MIRACLE with PLBART, CodeT5 and CodeT5+ against baseline approaches. The metric used for comparison is Computation Accuracy (CA@1). The Transcoder models do not support translation with C language, therefore the results are not included.

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
DT+Snippet+Function	295,254	38.98	34.94	37.1	30.21	35.54	39.58
AND+COMP+STAT+DT+Snippet+Function	551,286	38.98	32.23	37.63	**33.84**	35.84	39.58
DT+STAT+COMP+AND+Snippet+Function (**MIRACLE**)	551,286	**41.94**	**35.24**	**40.05**	**33.84**	**38.55**	**41.09**

Table 3: 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.

Table 1 shows the statistics of the synthetic parallel code data generated by PLBART as an example. Note that the datasets resulting from static analysis and compilation are not subsets of direct translation, because for direct translation we randomly pick a hypothesis from the 10 sampled hypotheses, and 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. Due to the generator being trained with less than 150 examples on C, compilation has the lowest selection rate on this programming language.

Qualitative Analysis. We further perform qualitative analysis and manually inspect samples of the generated data. Figure 4 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 results indicate that it does not impede the effectiveness of the synthetic code in improving code translation performance.

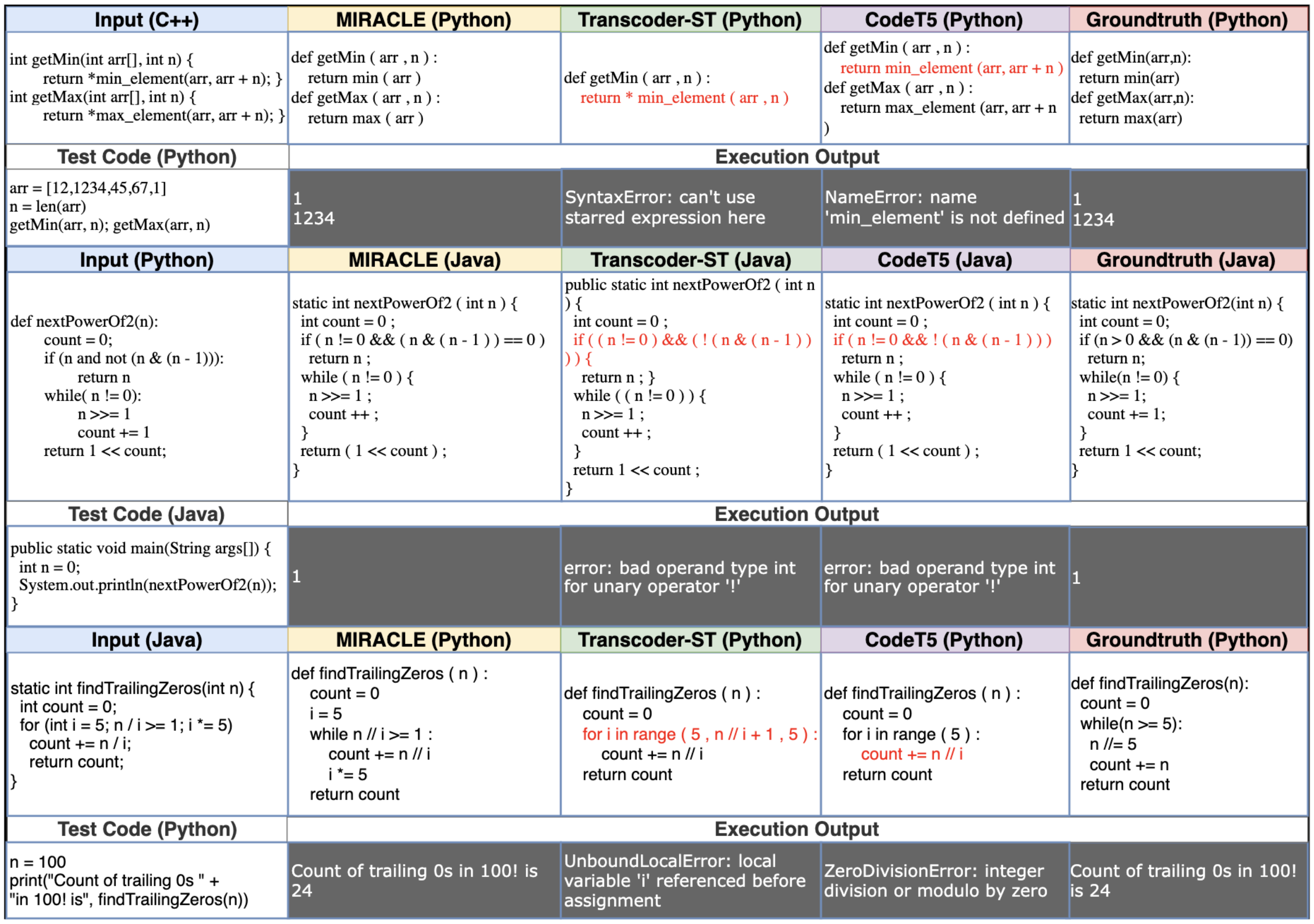
5.2 Code Translation Performance

To evaluate the improvement in translation performance using the generated data, we performed parallel code generation with three models, PLBART, CodeT5, and CodeT5+. We then trained them using their respective generated data. We compared their performances with and without the generated data. We also compared them against several state-of-the-art baselines, including unsupervised code translation models TransCoder and TransCoder-ST and open-source Code LLM, CodeLLama-7B-Instruct.

Comparison with Baseline Models. Table 2 shows the Computation Accuracy performance on C++, Java, Python, and C of the baseline models and the MIRACLE models. MIRACLE-CodeT5 and MIRACLE-CodeT5+ outperforms the best open-source models' performance on all of the language pairs except for Java – Python and Python – C, surpassing LLMs of tens of times the size (MIRACLE-CodeT5 has 220M parameters, MIRACLE-CodeT5+ has 770M parameters, while CodeLLama-7B-Instruct has 7B parameters). Specifically, MIRACLE-CodeT5+ outperforms CodeLLama-7B o C++ – Java by 32%. The experimental results demonstrate that smaller open-source models trained on high-quality synthetic parallel data can achieve comparable code translation performance of LLMs of much larger size and extensive pre-training.

Notably, all three MIRACLE models beats their corresponding generator models on almost all of the language pairs. Compared to PLBART, MIRACLE-PLBART has a 16% improvement on Java – C++; MIRACLE-CodeT5 outperforms CodeT5 on C++ – Python with a 29% margin; MIRACLE-CodeT5+ improves CodeT5+'s performance on Python – C++ by 7%.

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 2 shows the performance of the three MIRACLE variants and their respective



Input (C++)

```
int getMin(int arr[], int n) {
      return *min_element(arr, arr + n); }
int getMax(int arr[], int n) {
      return *max_element(arr, arr + n); }
```

MIRACLE (Python)

```
def getMin ( arr , n ) :
  return min ( arr )
def getMax ( arr , n ) :
  return max ( arr )
```

Transcoder-ST (Python)

```
def getMin ( arr , n ) :
  return * min_element ( arr , n )
```

CodeT5 (Python)

```
def getMin ( arr , n ) :
  return min_element (arr, arr + n )
def getMax ( arr , n ) :
  return max_element (arr, arr + n
)
```

Groundtruth (Python)

```
def getMin(arr,n):
  return min(arr)
def getMax(arr,n):
  return max(arr)
```

Test Code (Python)

```
arr = [12,1234,45,67,1]
n = len(arr)
getMin(arr, n); getMax(arr, n)
```

Execution Output

MIRACLE:

```
1
1234
```

Transcoder-ST:

```
SyntaxError: can't use
starred expression here
```

CodeT5:

```
NameError: name
'min_element' is not defined
```

Groundtruth:

```
1
1234
```

Input (Python)

```
def nextPowerOf2(n):
    count = 0;
    if (n and not (n & (n - 1))):
        return n
    while( n != 0):
        n >>= 1
        count += 1
    return 1 << count;
```

MIRACLE (Java)

```
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 ) ;
}
```

Transcoder-ST (Java)

```
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 ;
}
```

CodeT5 (Java)

```
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 ) ;
}
```

Groundtruth (Java)

```
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)

```
public static void main(String args[]) {
  int n = 0;
  System.out.println(nextPowerOf2(n));
}
```

Execution Output

MIRACLE:

```
1
```

Transcoder-ST:

```
error: bad operand type int
for unary operator '!'
```

CodeT5:

```
error: bad operand type int
for unary operator '!'
```

Groundtruth:

```
1
```

Input (Java)

```
static int findTrailingZeros(int n) {
  int count = 0;
  for (int i = 5; n / i >= 1; i *= 5)
    count += n / i;
    return count;
}
```

MIRACLE (Python)

```
def findTrailingZeros ( n ) :
  count = 0
  i = 5
  while n // i >= 1 :
    count += n // i
    i *= 5
  return count
```

Transcoder-ST (Python)

```
def findTrailingZeros ( n ) :
  count = 0
  for i in range ( 5 , n // i + 1 , 5 ) :
    count += n // i
  return count
```

CodeT5 (Python)

```
def findTrailingZeros ( n ) :
  count = 0
  for i in range ( 5 ) :
    count += n // i
  return count
```

Groundtruth (Python)

```
def findTrailingZeros(n):
  count = 0
  while(n >= 5):
    n //= 5
    count += n
  return count
```

Test Code (Python)

```
n = 100
print("Count of trailing 0s " +
"in 100! is", findTrailingZeros(n))
```

Execution Output

MIRACLE:

```
Count of trailing 0s in 100! is
24
```

Transcoder-ST:

```
UnboundLocalError: local
variable 'i' referenced before
assignment
```

CodeT5:

```
ZeroDivisionError: integer
division or modulo by zero
```

Groundtruth:

```
Count of trailing 0s in 100!
is 24
```

Figure 5: Comparison of models of similar sizes, including MIRACLE-CodeT5, CodeT5 and Transcoder-ST. 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.

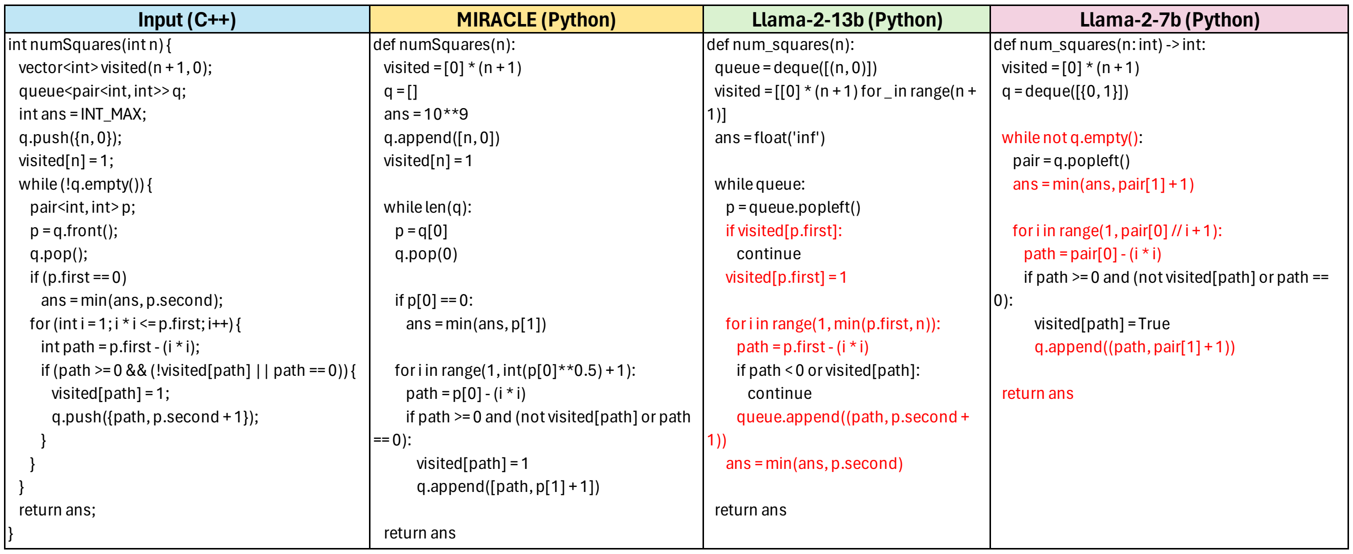
generators. For PLBART, MIRACLE improves Computation Accuracy (CA@1) by up to 43% on C – C++. Results demonstrate that the augmentation of parallel code generation works well in low-resource language settings, where the generator’s performance is weak. For CodeT5, the improvement in CA@1 is up to 23% on C – Python. For CodeT5+, the improvement is up to 12% on Java – C. We also observed that larger models tend to perform better with low resource language, as the improvement margin of MIRACLE compared to the generator model decreases as the model sizes get larger.

Analysis of Alignment-Ascending Curriculum Table 4 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 3. 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

Data	Type	Volume	Source
DT	Synthetic	260,110	Direct Translation
STAT	Synthetic	176,482	Static Analysis
COMP	Synthetic	79,550	Compilation
AND	Synthetic	69,358	Static Analysis & Compilation
Snippet	Annotated	31,818	ECoST
Function	Annotated	3,326	ECoST

Table 4: Datasets for Alignment-Ascending Curriculum Learning. Volume refers to the number of parallel codes.

synthetic data, AND, further improves the performance. Similarly, the integration of unfiltered noisy data, DT, also boosts the performance. However, neither AND nor DT 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+DT+Snippet+Function causes the performance to drop significantly compared to MIRACLE, emphasizing the importance of the order of the curriculum. Interestingly, this inverted curriculum aligns closely in performance with



Input (C++)
```
int numSquares(int n) {
  vector<int> visited(n + 1, 0);
  queue<pair<int, int>> q;
  int ans = INT_MAX;
  q.push({n, 0});
  visited[n] = 1;
  while (!q.empty()) {
    pair<int, int> p;
    p = q.front();
    q.pop();
    if (p.first == 0)
      ans = min(ans, p.second);
    for (int i = 1; i * i <= p.first; i++) {
      int path = p.first - (i * i);
      if (path >= 0 && (!visited[path] | | path == 0)) {
        visited[path] = 1;
        q.push({path, p.second + 1});
      }
    }
  }
  return ans;
}
```

MIRACLE (Python)
```
def numSquares(n):
  visited = [0] * (n + 1)
  q = []
  ans = 10**9
  q.append([n, 0])
  visited[n] = 1

  while len(q):
    p = q[0]
    q.pop(0)

    if p[0] == 0:
      ans = min(ans, p[1])

    for i in range(1, int(p[0]**0.5) + 1):
      path = p[0] - (i * i)
      if path >= 0 and (not visited[path] or path
== 0):
        visited[path] = 1
        q.append([path, p[1] + 1])

  return ans
```

Llama-2-13b (Python)
```
def num_squares(n):
  queue = deque([(n, 0)])
  visited = [[0] * (n + 1) for _ in range(n +
1)]
  ans = float('inf')

  while queue:
    p = queue.popleft()
    if visited[p.first]:
      continue
    visited[p.first] = 1

    for i in range(1, min(p.first, n)):
      path = p.first - (i * i)
      if path < 0 or visited[path]:
        continue
      queue.append((path, p.second +
1))
    ans = min(ans, p.second)

  return ans
```

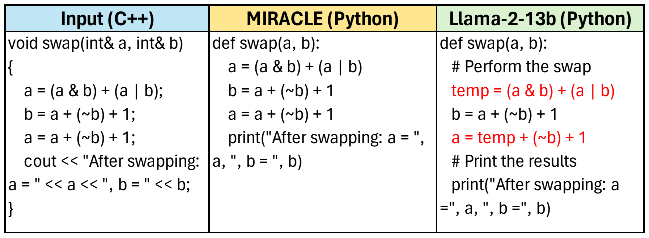
Llama-2-7b (Python)
```
def num_squares(n: int) -> int:
  visited = [0] * (n + 1)
  q = deque([{0, 1}])

  while not q.empty():
    pair = q.popleft()
    ans = min(ans, pair[1] + 1)

    for i in range(1, pair[0] // i + 1):
      path = pair[0] - (i * i)
      if path >= 0 and (not visited[path] or path ==
0):
        visited[path] = True
        q.append((path, pair[1] + 1))

  return ans
```

Figure 6: Examples of wrong translations from Llama-2 models that try to mimic the input code while translating from C++ to Python. In the Llama-2-13b translation, p.first and p.second do not work in Python similarly as in Java code. For Llama-2-7b, p[0] and p[1] does not work for set elements. Also, deque in Python does not have an empty method whereas it works for Java.



Input (C++)
```
void swap(int& a, int& b)
{
  a = (a & b) + (a | b);
  b = a + (~b) + 1;
  a = a + (~b) + 1;
  cout << "After swapping:
a = " << a << ", b = " << b;
}
```

MIRACLE (Python)
```
def swap(a, b):
  a = (a & b) + (a | b)
  b = a + (~b) + 1
  a = a + (~b) + 1
  print("After swapping: a = ",
a, ", b = ", b)
```

Llama-2-13b (Python)
```
def swap(a, b):
  # Perform the swap
  temp = (a & b) + (a | b)
  b = a + (~b) + 1
  a = temp + (~b) + 1
  # Print the results
  print("After swapping: a
=", a, ", b =", b)
```

Figure 7: An example of wrong translation from Llama-2-13b for C++ to Python code translation. The bitwise swap operation does not need any extra temporary variable but the model introduced a temp variable instead of using a. This translation generates wrong output.

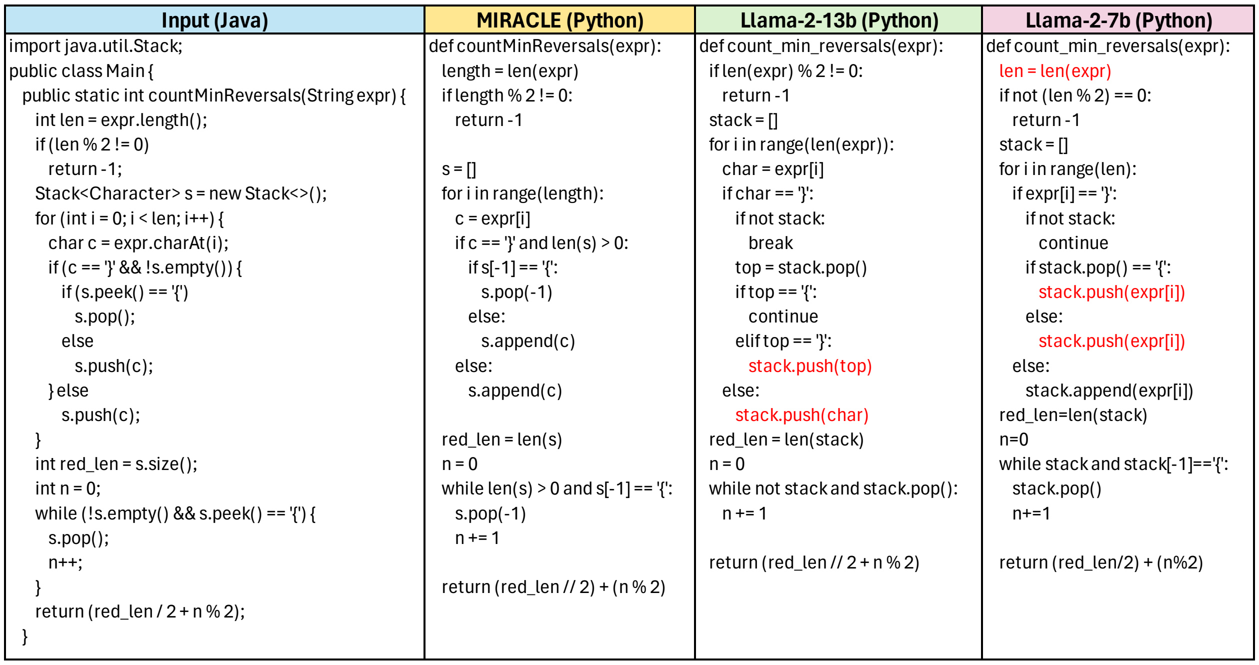
DT+Snippet+Function, likely due to the larger volume of the DT dataset overpowering the effect of the previous datasets.

Qualitative Analysis of Similar Size Models. To compare the performance of models of similar sizes, in Figure 5 we show examples translations of MIRACLE-CodeT5 (220M) with two other baselines, TransCoder-ST (110M) and CodeT5 (220M). We also include 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. 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 language 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.

5.3 LLM Code Translation Capabilities

In addition to our existing experiments, we assess the code translation capability of GPT-3.5, GPT-4 and two more LLMs, Llama-2-13b and Llama-2-7b, through prompts with 18 function-level instances from the ECoST test set. We manually analyze several different language pairs, including C++-Python, Java-Python, C++-C, and Java-C. With the 18 instances tested, GPT-3.5 and GPT-4 both achieved perfect accuracy, while Llama-2-13b and Llama-2-7b failed to correctly translate 6 and 5 instances, achieving accuracy of 66.67% and 72.22%, respectively. Notably, our MIRACLE-CodeT5+ also achieved 100% accuracy for those cases, demonstrating the efficacy of the proposed approach.

In Figure 6, both Llama-2 models failed to translate from C++ to Python. Both models mimicked the input code in the translation. Llama-2-13b called `p.first` and `p.second` in the translation, which causes an AttributeError in Python ('tuple' object has no attribute 'first'). Llama-2-7b initialized q as a set but tried to access it like a tuple or list which throws an error stating the set object is not subscriptable in Python. Additionally, it tried to check whether the queue is empty by calling an empty function mimicking the input code, however, deque does not have an empty method. Another type of error that results from the tendency to replicate the input code includes defining variable names without ensuring the name is not



Input (Java)

```
import java.util.Stack;
public class Main {
  public static int countMinReversals(String expr) {
    int len = expr.length();
    if (len % 2 != 0)
      return -1;
    Stack<Character> s = new Stack<>();
    for (int i = 0; i < len; i++) {
      char c = expr.charAt(i);
      if (c == '}' && !s.empty()) {
        if (s.peek() == '{')
          s.pop();
        else
          s.push(c);
      } else
        s.push(c);
    }
    int red_len = s.size();
    int n = 0;
    while (!s.empty() && s.peek() == '{') {
      s.pop();
      n++;
    }
    return (red_len / 2 + n % 2);
  }
```

MIRACLE (Python)

```
def countMinReversals(expr):
  length = len(expr)
  if length % 2 != 0:
    return -1

  s = []
  for i in range(length):
    c = expr[i]
    if c == '}' and len(s) > 0:
      if s[-1] == '{':
        s.pop(-1)
      else:
        s.append(c)
    else:
      s.append(c)

  red_len = len(s)
  n = 0
  while len(s) > 0 and s[-1] == '{':
    s.pop(-1)
    n += 1

  return (red_len // 2) + (n % 2)
```

Llama-2-13b (Python)

```
def count_min_reversals(expr):
  if len(expr) % 2 != 0:
    return -1
  stack = []
  for i in range(len(expr)):
    char = expr[i]
    if char == '}':
      if not stack:
        break
      top = stack.pop()
      if top == '{':
        continue
      elif top == '}':
        stack.push(top)
    else:
      stack.push(char)
  red_len = len(stack)
  n = 0
  while not stack and stack.pop():
    n += 1

  return (red_len // 2 + n % 2)
```

Llama-2-7b (Python)

```
def count_min_reversals(expr):
  len = len(expr)
  if not (len % 2) == 0:
    return -1
  stack = []
  for i in range(len):
    if expr[i] == '}':
      if not stack:
        continue
      if stack.pop() == '{':
        stack.push(expr[i])
      else:
        stack.push(expr[i])
    else:
      stack.append(expr[i])
  red_len=len(stack)
  n=0
  while stack and stack[-1]=='{':
    stack.pop()
    n+=1

  return (red_len/2) + (n%2)
```

Figure 8: An example of wrong translation from Llama-2 models for Java to Python code translation. Both of the models mimicked the input code by calling push for list. Llama-2-7b declared a variable named 'len' and called the built-in function len that causes the UnboundLocalError.

as same as a built-in function name in the target language. We found such a translation from Llama-2-7b that throws UnboundLocalError. Figure 8 demonstrates one example, where both of the Llama-2 models replicated the Java input code to the Python translation. Additionally, Llama-2-7b defined a variable name as `len` while calling the built-in function 'len' in the same line. This causes the UnboundLocalError as mentioned earlier. A similar type of error is seen in the translation when a local variable is not defined earlier. In Figure 7, the input C++ code swaps values of two variables through bitwise operations without using a third temporary variable. While other models translated to Python code correctly, Llama-2-13b introduced a third variable `temp`. This might happen in the context of a swap operation that utilizes a temporary variable. However, bitwise swap operation does not need such a variable and therefore, the translated code generates a wrong output. Nonetheless, the translations are error-free when the input codes are straightforward with regular variable declarations, arithmetic operations, conditional statements, and loops.

6 THREATS TO VALIDITY

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.

7 CONCLUSION

In this paper, we introduce MIRACLE, a semi-supervised approach that 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 amounts of parallel data.

8 ACKNOWLEDGEMENT

This work has been supported by the Virginia Commonwealth Cyber Initiative (CCI) and the Office of Naval Research under Grant N00014-22-1-2057; and National Science Foundation under Grant No. CNS-1929701.

REFERENCES

[1] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. Gpt-4 Technical Report. *arXiv:2303.08774* (2023).

[2] David H Ackley, Geoffrey E Hinton, and Terrence J Sejnowski. 1985. A Learning Algorithm for Boltzmann Machines. *Cognitive science* 9, 1 (1985), 147–169.

[3] 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:2110.05423* (2021).

[4] Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified Pre-training for Program Understanding and Generation. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. 2655–2668.

[5] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2022. Summarize and Generate to Back-translate: Unsupervised Translation of Programming Languages. *arXiv:2205.11116* (2022).

[6] Wasi Uddin Ahmad, Md Golam Rahman Tushar, Saikat Chakraborty, and Kai-Wei Chang. 2021. AVATAR: A Parallel Corpus for Java-Python Program Translation. *arXiv:2108.11590* (2021).

[7] Mikel Artetxe, Gorka Labaka, and Eneko Agirre. 2019. An Effective Approach to Unsupervised Machine Translation. *arXiv:1902.01313* (2019).

[8] Mikel Artetxe, Gorka Labaka, Eneko Agirre, and Kyunghyun Cho. 2017. Unsupervised Neural Machine Translation. *arXiv:1710.11041* (2017).

[9] Xinyun Chen, Chang Liu, and Dawn Song. 2018. Tree-to-tree Neural Networks for Program Translation. In *Advances in Neural Information Processing Systems*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett (Eds.), Vol. 31. Curran Associates, Inc. https://proceedings.neurips.cc/paper/2018/file/d759175de8ea5b1d9a2660e45554894f-Paper.pdf

[10] 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)*, Jill Burstein, Christy Doran, and Thamar Solorio (Eds.). Association for Computational Linguistics, Minneapolis, Minnesota, 4171–4186. https://doi.org/10.18653/v1/N19-1423

[11] Sergey Edunov, Myle Ott, Michael Auli, and David Grangier. 2018. Understanding Back-Translation at Scale. *arXiv:1808.09381* (2018).

[12] 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*. Association for Computational Linguistics, Online, 1536–1547. https://doi.org/10.18653/v1/2020.findings-emnlp.139

[13] 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:2204.05999* (2022).

[14] 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:2009.08366* (2020).

[15] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. 2015. Distilling the Knowledge in a Neural Network. *arXiv:1503.02531* (2015).

[16] 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*. 10903–10914.

[17] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. Codesearchnet Challenge: Evaluating the State of Semantic Code Search. *arXiv:1909.09436* (2019).

[18] 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*. 173–184.

[19] 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)*. 384–388. https://doi.org/10.1109/ICAC3N56670.2022.10074182

[20] Guillaume Lample and Alexis Conneau. 2019. Cross-lingual Language Model Pretraining. *arXiv e-prints* (2019), arXiv–1901.

[21] Guillaume Lample, Alexis Conneau, Ludovic Denoyer, and Marc'Aurelio Ranzato. 2018. Unsupervised Machine Translation Using Monolingual Corpora Only. In *International Conference on Learning Representations*.

[22] 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*. 7871–7880.

[23] 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 (2020), 726–742.

[24] 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. (2019).

[25] 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)*.

[26] 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*. 651–654.

[27] 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)*. IEEE, 585–596.

[28] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis. In *The Eleventh International Conference on Learning Representations*.

[29] Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. 2022. Training Language Models to Follow Instructions with Human Feedback. *Advances in neural information processing systems* 35 (2022), 27730–27744.

[30] 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:2308.03109* (2023).

[31] 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:2105.12655* (2021).

[32] 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 (2020), 1–67.

[33] Sazzadur Rahaman, Ya Xiao, Sharmin Afrose, Fahad Shaon, Ke Tian, Miles Frantz, Murat Kantarcioglu, and Danfeng Yao. 2019. Cryptoguard: High Precision Detection of Cryptographic Vulnerabilities in Massive-sized Java Projects. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 2455–2472.

[34] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. 2023. Code Llama: Open Foundation Models for Code. *arXiv:2308.12950* (2023).

[35] Baptiste Roziere, Marie-Anne Lachaux, Lowik Chanussot, and Guillaume Lample. 2020. Unsupervised Translation of Programming Languages.. In *NeurIPS*.

[36] Baptiste Roziere, Marie-Anne Lachaux, Marc Szafraniec, and Guillaume Lample. 2021. DOBF: A Deobfuscation Pre-Training Objective for Programming Languages. *arXiv:2102.07492* (2021).

[37] Baptiste Roziere, Jie Zhang, Francois Charton, Mark Harman, Gabriel Synnaeve, and Guillaume Lample. 2021. Leveraging Automated Unit Tests for Unsupervised Code Translation. In *International Conference on Learning Representations*.

[38] Marc Szafraniec, Baptiste Roziere, Hugh Leather Francois Charton, Patrick Labatut, and Gabriel Synnaeve. 2022. Code Translation with Compiler Representations. *arXiv:2207.03578* (2022).

[39] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. 2023. Llama 2: Open Foundation and Fine-tuned Chat Models. *arXiv:2307.09288* (2023).

[40] 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*. 5998–6008.

[41] Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi DQ Bui, Junnan Li, and Steven CH Hoi. 2023. CodeT5+: Open Code Large Language Models for Code Understanding and Generation. *arXiv:2305.07922* (2023).

[42] 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*. 8696–8708.

[43] Ya Xiao, Wenjia Song, Salman Ahmed, Xinyang Ge, Bimal Viswanath, Na Meng, and Danfeng Yao. 2024. Measurement of Embedding Choices on Cryptographic API Completion Tasks. *ACM Transactions on Software Engineering and Methodology* 33, 3 (2024), 1–30.

[44] Ya Xiao, Wenjia Song, Jingyuan Qi, Bimal Viswanath, Patrick McDaniel, and Danfeng Yao. 2023. Specializing Neural Networks for Cryptographic Code

Completion Applications. *IEEE Transactions on Software Engineering* 49, 6 (2023), 3524–3535.

[45] 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*. 5673–5684.

[46] Ming Zhu, Karthik Suresh, and Chandan K Reddy. 2022. Multilingual Code Snippets Training for Program Translation. In *36th AAAI Conference on Artificial Intelligence (AAAI)*.