Unsupervised Translation of Programming Languages

Marie-Anne Lachaux[*]
Facebook AI Research
malachaux@fb.com

Baptiste Roziere[*]
Facebook AI Research
Paris-Dauphine University
broz@fb.com

Lowik Chanussot
Facebook AI Research
lowik@fb.com

Guillaume Lample
Facebook AI Research
glample@fb.com

Abstract

A transcompiler, also known as source-to-source translator, is a system that converts source code from a high-level programming language (such as C++ or Python) to another. Transcompilers are primarily used for interoperability, and to port codebases written in an obsolete or deprecated language (e.g. COBOL, Python 2) to a modern one. They typically rely on handcrafted rewrite rules, applied to the source code abstract syntax tree. Unfortunately, the resulting translations often lack readability, fail to respect the target language conventions, and require manual modifications in order to work properly. The overall translation process is time-consuming and requires expertise in both the source and target languages, making code-translation projects expensive. Although neural models significantly outperform their rule-based counterparts in the context of natural language translation, their applications to transcompilation have been limited due to the scarcity of parallel data in this domain. In this paper, we propose to leverage recent approaches in unsupervised machine translation to train a fully unsupervised neural transcompiler. We train our model on source code from open source GitHub projects, and show that it can translate functions between C++, Java, and Python with high accuracy. Our method relies exclusively on monolingual source code, requires no expertise in the source or target languages, and can easily be generalized to other programming languages. We also build and release a test set composed of 852 parallel functions, along with unit tests to check the correctness of translations. We show that our model outperforms rule-based commercial baselines by a significant margin.

1 Introduction

A transcompiler, transpiler, or source-to-source compiler, is a translator which converts between programming languages that operate at a similar level of abstraction. Transcompilers differ from traditional compilers that translate source code from a high-level to a lower-level programming language (e.g. assembly language) to create an executable. Initially, transcompilers were developed to port source code between different platforms (e.g. convert source code designed for the Intel 8080 processor to make it compatible with the Intel 8086). More recently, new languages have been developed (e.g. CoffeeScript, TypeScript, Dart, Haxe) along with dedicated transcompilers that convert them into a popular or omnipresent language (e.g. JavaScript). These new languages address some shortcomings of the target language by providing new features such as list comprehension (CoffeeScript), object-oriented programming and type checking (TypeScript), while detecting errors and providing optimizations. Unlike traditional programming languages, these new languages are

[*]Equal contribution. The order was determined randomly.

designed to be translated with a perfect accuracy (i.e. the compiled language does not require manual adjustments to work properly). In this paper, we are more interested in the traditional type of transcompilers, where typical use cases are to translate an existing codebase written in an obsolete or deprecated language (e.g. COBOL, Python 2) to a recent one, or to integrate code written in a different language to an existing codebase.

Migrating an existing codebase to a modern or more efficient language like Java or C++ requires expertise in both the source and target languages, and is often costly. For instance, the Commonwealth Bank of Australia spent around $750 million and 5 years of work to convert its platform from COBOL to Java. Using a transcompiler and manually adjusting the output source code may be a faster and cheaper solution than rewriting the entire codebase from scratch. In natural language, recent advances in neural machine translation have been widely accepted, even among professional translators, who rely more and more on automated machine translation systems. A similar phenomenon could be observed in programming language translation in the future.

Translating source code from one Turing-complete language to another is always possible in theory. Unfortunately, building a translator is difficult in practice: different languages can have a different syntax and rely on different platform APIs and standard-library functions. Currently, the majority of transcompilation tools are rule-based; they essentially tokenize the input source code and convert it into an Abstract Syntax Tree (AST) on which they apply handcrafted rewrite rules. Creating them requires a lot of time, and advanced knowledge in both the source and target languages. Moreover, translating from a dynamically-typed language (e.g. Python) to a statically-typed language (e.g. Java) requires to infer the variable types which is difficult (and not always possible) in itself.

The applications of neural machine translation (NMT) to programming languages have been limited so far, mainly because of the lack of parallel resources available in this domain. In this paper, we propose to apply recent approaches in unsupervised machine translation, by leveraging large amount of monolingual source code from GitHub to train a model, TransCoder, to translate between three popular languages: C++, Java and Python. To evaluate our model, we create a test set of 852 parallel functions, along with associated unit tests. Although never provided with parallel data, the model manages to translate functions with a high accuracy, and to properly align functions from the standard library across the three languages, outperforming rule-based and commercial baselines by a significant margin. Our approach is simple, does not require any expertise in the source or target languages, and can easily be extended to most programming languages. Although not perfect, the model could help to reduce the amount of work and the level of expertise required to successfully translate a codebase. The main contributions of the paper are the following:

แปลง code →
- We introduce a new approach to translate functions from a programming language to another, that is purely based on monolingual source code.

เข้าใจ complex pattern →
- We show that TransCoder successfully manages to grasp complex patterns specific to each language, and to translate them to other languages.

Out →
- We show that a fully unsupervised method can outperform commercial systems that leverage rule-based methods and advanced programming knowledge.

Creat dataset →
- We build and release a validation and a test set composed of 852 parallel functions in 3 languages, along with unit tests to evaluate the correctness of generated translations.
- We will make our code and pretrained models publicly available.

pretrain

2 Related work

Source-to-source translation. Several studies have investigated the possibility to translate programming languages with machine translation. For instance, Nguyen et al. [36] trained a Phrase-Based Statistical Machine Translation (PBSMT) model, Moses [27], on a Java-C# parallel corpus. They created their dataset using the implementations of two open source projects, Lucene and db4o, developed in Java and ported to C#. Similarly, Karaivanov et al. [22] developed a tool to mine parallel datasets from ported open source projects. Aggarwal et al. [1] trained Moses on a Python 2 to Python 3 parallel corpus created with 2to3, a Python library [2] developed to port Python 2 code to Python 3. Chen et al. [12] used the Java-C# dataset of Nguyen et al. [36] to translate code with tree-to-tree neural networks.

[2] https://docs.python.org/2/library/2to3.html

They also use a transcompiler to create a parallel dataset CoffeeScript-Javascript. Unfortunately, all these approaches are supervised, and rely either on the existence of open source projects available in multiple languages, or on existing transcompilers, to create parallel data. Moreover, they essentially rely on BLEU score [38] to evaluate their translations [1, 10, 22, 36], which is not a reliable metric, as a generation can be a valid translation while being very different from the reference.

Translating from source code. Other studies have investigated the use of machine translation from source code. For instance, Oda et al. [37] trained a PBSMT model to generate pseudo-code. To create a training set, they hired programmers to write the pseudo-code of existing Python functions. Barone and Sennrich [10] built a corpus of Python functions with their docstrings from open source GitHub repositories. They showed that a neural machine translation model could be used to map functions to their associated docstrings, and vice versa. Similarly, Hu et al. [21] proposed a neural approach, DeepCom, to automatically generate code comments for Java methods.

Other applications. Another line of work studied the applications of neural networks to code suggestion [2, 11, 34], or error detection [13, 18, 47]. Recent approaches have also investigated the use of neural approaches for code decompilation [16, 24]. For instance, Katz et al. [23] propose a sequence-to-sequence model to predict the C code of binary programs. A common issue with standard seq2seq models, is that the generated functions are not guaranteed to compile, and even to be syntactically correct. To address this issue, several approaches proposed to use additional constraints on the decoder, to ensure that the generated functions respect the syntax of the target language [3, 4, 5, 40, 48]. Recently, Feng et al. [15] introduced Codebert, a transformer pretrained with a BERT-like objective [14] on open source GitHub repositories. They showed that pretraining improves the performance on several downstream tasks such as code documentation generation and code completion.

Unsupervised Machine Translation. The quality of NMT systems highly depends on the quality of the available parallel data. However, for the majority of languages, parallel resources are rare or nonexistent. Since creating parallel corpora for training is not realistic (creating a small parallel corpus for evaluation is already challenging [19]), some approaches have investigated the use of monolingual data to improve existing machine translation systems [17, 20, 41, 49]. More recently, several methods were proposed to train a machine translation system exclusively from monolingual corpora, using either neural models [30, 8] and statistical models [32, 7]. We describe now some of these methods and how they can be instantiated in the setting of unsupervised transcompilation.

3 Model

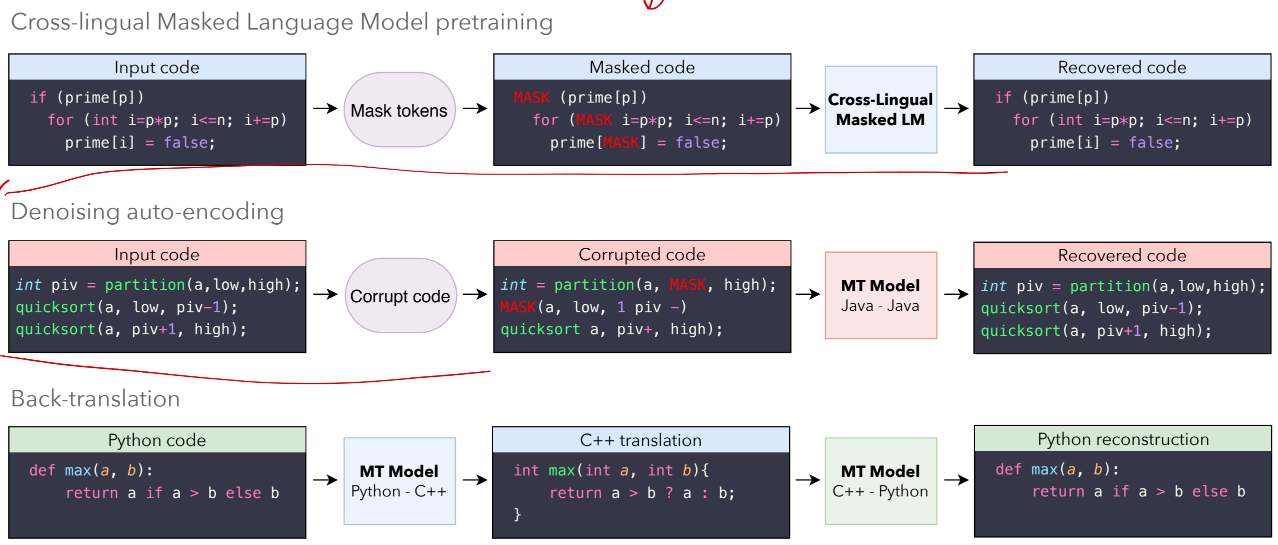
For TransCoder, we consider a sequence-to-sequence (seq2seq) model with attention [44, 9], composed of an encoder and a decoder with a transformer architecture [45]. We use a single shared model for all programming languages. We train it using the three principles of unsupervised machine translation identified in Lample et al. [32], namely initialization, language modeling, and back-translation. In this section, we summarize these principles and detail how we instantiate them to translate programming languages. An illustration of our approach is given in Figure 1.

3.1 Cross Programming Language Model pretraining

Pretraining is a key ingredient of unsupervised machine translation Lample et al. [32]. It ensures that sequences with a similar meaning are mapped to the same latent representation, regardless of their languages. Originally, pretraining was done by initializing the model with cross-lingual word representations [30, 8]. In the context of unsupervised English-French translation, the embedding of the word “cat” will be close to the embedding of its French translation “chat”. Cross-lingual word embeddings can be obtained by training monolingual word embeddings and aligning them in an unsupervised manner [31, 6].

Subsequent work showed that pretraining the entire model (and not only word representations) in a cross-lingual way could lead to significant improvements in unsupervised machine translation [29, 33, 43]. In particular, we follow the pretraining strategy of Lample and Conneau [29], where a Cross-lingual Language Model (XLM) is pretrained with a masked language modeling objective [14] on monolingual source code datasets.

Bz Mask ⊨ Bert



```
if (prime[p])
  for (int i=p*p; i<=n; i+=p)
    prime[i] = false;
```

```
MASK (prime[p])
  for (MASK i=p*p; i<=n; i+=p)
    prime[MASK] = false;
```

```
if (prime[p])
  for (int i=p*p; i<=n; i+=p)
    prime[i] = false;
```

recover

```
int piv = partition(a,low,high);
quicksort(a, low, piv-1);
quicksort(a, piv+1, high);
```

```
int = partition(a, MASK, high);
MASK(a, low, 1 piv -)
quicksort a, piv+, high);
```

```
int piv = partition(a,low,high);
quicksort(a, low, piv-1);
quicksort(a, piv+1, high);
```

```
def max(a, b):
    return a if a > b else b
```

```
int max(int a, int b){
    return a > b ? a : b;
}
```

```
def max(a, b):
    return a if a > b else b
```

Back MT

Figure 1: **Illustration of the three principles of unsupervised machine translation used by our approach.** The first principle initializes the model with cross-lingual masked language model pretraining. As a result, pieces of code that express the same instructions are mapped to the same representation, regardless of the programming language. Denoising auto-encoding, the second principle, trains the decoder to always generate valid sequences, even when fed with noisy data, and increases the encoder robustness to input noise. Back-translation, the last principle, allows the model to generate parallel data which can be used for training. Whenever the Python → C++ model becomes better, it generates more accurate data for the C++ → Python model, and vice versa. Figure 5 in the appendix provides a representation of the cross-lingual embeddings we obtain after training.

ชื่อเมือง, ชื่อคน

pretain model ที่ใช้กับ Anchor point

คำร่วม

The cross-lingual nature of the resulting model comes from the significant number of common tokens (anchor points) that exist across languages. In the context of English-French translation, the anchor points consists essentially of digits and city and people names. In programming languages, these anchor points come from common keywords (e.g. `for`, `while`, `if`, `try`), and also digits, mathematical operators, and English strings that appear in the source code. [3]

For the masked language modeling (MLM) objective, at each iteration we consider an input stream of source code sequences, randomly mask out some of the tokens, and train TransCoder to predict the tokens that have been masked out based on their contexts. We alternate between streams of batches of different languages. This allows the model to create high quality, cross-lingual sequence representations. An example of XLM pretraining is given on top of Figure 1.

3.2 Denoising auto-encoding

We initialize the encoder and decoder of the seq2seq model with the XLM model pretrained in Section 3.1. The initialization is straightforward for the encoder, as it has the same architecture as the XLM model. The transformer decoder, however, has extra parameters related to the source attention mechanism [45]. Following Lample and Conneau [29], we initialize these parameters randomly.

XLM pretraining allows the seq2seq model to generate high quality representations of input sequences. However, the decoder lacks the capacity to translate, as it has never been trained to decode a sequence based on a source representation. To address this issue, we train the model to encode and decode sequences with a Denoising Auto-Encoding (DAE) objective [46]. The DAE objective operates like a supervised machine translation algorithm, where the model is trained to predict a sequence of tokens given a corrupted version of that sequence. To corrupt a sequence, we use the same noise model as the one described in Lample et al. [30]. Namely, we randomly mask, remove and shuffle input tokens.

[3] In practice, the "cross-linguality" of the model highly depends on the amount of anchor points across languages. As a result, a XLM model trained on English-French will provide better cross-lingual representations than a model trained on English-Chinese, because of the different alphabet which reduces the number of anchor points. In programming languages, the majority of strings are composed of English words, which results in a fairly high number of anchor points, and the model *naturally* becomes cross-lingual.

The first symbol given as input to the decoder is a special token indicating the output programming language. At test time, a Python sequence can be encoded by the model, and decoded using the C++ start symbol to generate a C++ translation. The quality of the C++ translation will depend on the "cross-linguality" of the model: if the Python function and a valid C++ translation are mapped to the same latent representation by the encoder, the decoder will successfully generate this C++ translation.

The DAE objective also trains the "language modeling" aspect of the model, i.e. the decoder is always trained to generate a valid function, even when the encoder output is noisy. Moreover it also trains the encoder to be robust to input noise, which is helpful in the context of back-translation where the model is trained with noisy input sequences. DAE is illustrated in the middle of Figure 1.

3.3 Back-translation

In practice, XLM pretraining and denoising auto-encoding alone are enough to generate translations. However, the quality of these translations tends to be low, as the model is never trained to do what it is expected to do at test time, i.e. to translate functions from one language to another. To address this issue, we use back-translation, which is one of the most effective methods to leverage monolingual data in a weakly-supervised scenario. Initially introduced to improve the performance of machine translation in the supervised setting [41], back-translation turned out to be an important component of unsupervised machine translation [30, 32, 8].

In the unsupervised setting, a source-to-target model is coupled with a backward target-to-source model trained in parallel. The target-to-source model is used to translate target sequences into the source language, producing noisy source sequences corresponding to the ground truth target sequences. The source-to-target model is then trained in a weakly supervised manner to reconstruct the target sequences from the noisy source sequences generated by the target-to-source model, and vice versa. The two models are trained in parallel until convergence. An example of back-translation is illustrated in Figure 1.

4 Experiments

4.1 Training details

We use a transformer with 6 layers, 8 attention heads, and set the dimensionality of the model to 1024. We use a single encoder and a single decoder for all programming languages. During XLM pretraining, we alternate between batches of C++, Java, and Python, composed of 32 sequences of source code of 512 tokens. At training time, we alternate between the denoising auto-encoding and back-translation objectives, and use batches of around 6000 tokens. We optimize TransCoder with the Adam optimizer [25], a learning rate of 10^{-4}, and use the same learning rate scheduler as Vaswani et al. [45]. We implement our models in PyTorch [39] and train them on 32 V100 GPUs. We use float16 operations to speed up training and to reduce the memory usage of our models.

4.2 Training data

We download the GitHub public dataset available on Google BigQuery[4]. It contains more than 2.8 million open source GitHub repositories. We filter projects whose license explicitly permits the re-distribution of parts of the project, and select the C++, Java, and Python files within those projects. Ideally, a transcompiler should be able to translate whole projects. In this work, we decide to translate at function level. Unlike files or classes, functions are short enough to fit into a single batch, and working at function level allows for a simpler evaluation of the model with unit tests (c.f. Section 4.4). We pretrain TransCoder on all source code available, and train the denoising auto-encoding and back-translation objectives on functions only. Please refer to Section A.3 and Table 3 in the appendix for more details on how the functions are extracted, and for statistics about our training set. We carry out an ablation study to determine whether it is better to keep or remove comments from source code. Keeping comments in the source code increases the number of anchor points across languages, which results in a better overall performance (c.f. Table 6 in the appendix). Therefore, we keep them in our final datasets and experiments.

[4] `https://console.cloud.google.com/marketplace/details/github/github-repos`

4.3 Preprocessing

Recent approaches in multilingual natural language processing tend to use a common tokenizer [28], and a shared vocabulary for all languages. This reduces the overall vocabulary size, and maximizes the token overlap between languages, improving the cross-linguality of the model [14, 29]. In our case, a universal tokenizer would be suboptimal, as different languages use different patterns and keywords. The logical operators && and || exist in C++ where they should be tokenized as a single token, but not in Python. The indentations are critical in Python as they define the code structure, but have no meaning in languages like C++ or Java. We use the `javalang`[5] tokenizer for Java, the tokenizer of the standard library for Python[6], and the `clang`[7] tokenizer for C++. These tokenizers ensure that meaningless modifications in the code (e.g. adding extra new lines or spaces) do not have any impact on the tokenized sequence. An example of tokenized code is given in Figure 3 in the appendix. We learn BPE codes [42] on extracted tokens, and split tokens into subword units. The BPE codes are learned with fastBPE[8] on the concatenation of tokenized C++, Java, and Python files.

4.4 Evaluation

GeeksforGeeks is an online platform[9] with computer science and programming articles. It gathers many coding problems and presents solutions in several programming languages. From these solutions, we extract a set of parallel functions in C++, Java, and Python, to create our validation and test sets. These functions not only return the same output, but also compute the result with similar algorithm. In Figure 4 in the appendix, we show an example of C++-Java-Python parallel function that determines whether an integer represented by a string is divisible by 13.

The majority of studies in source code translation use the **BLEU** score to evaluate the quality of generated functions [1, 10, 22, 36], or other metrics based on the relative overlap between the tokens in the translation and in the reference. A simple metric is to compute the **reference match**, i.e. the percentage of translations that perfectly match the ground truth reference [12]. A limitation of these metrics is that they do not take into account the syntactic correctness of the generations. Two programs with small syntactic discrepancies will have a high BLEU score while they could lead to very different compilation and computation outputs. Conversely, semantically equivalent programs with different implementations will have low BLEU scores. Instead, we introduce a new metric, the **computational accuracy**, that evaluates whether the hypothesis function generates the same outputs as the reference when given the same inputs. We consider that the hypothesis is correct if it gives the same output as the reference for every input. Section B and Table 4 in the appendix present more details on how we create these unit tests, and give statistics about our validation and test sets.

At inference, TransCoder can generate multiple translations using beam search decoding [26]. In machine translation, the considered hypotheses are typically the ones with the highest log-probabilities in the beam. In our case, we have access to unit tests to verify the correctness of the generated hypotheses, so we report two sets of results for our computational accuracy metric: **Beam N**, the percentage of functions with at least one correct translation in the beam, and **Beam N - Top 1** the percentage of functions where the hypothesis in the beam with the highest log-probability is a correct translation. We select our best model using greedy decoding (Beam 1) for speed efficiency.

4.5 Results

We report the results on our test set in Table 1, using greedy decoding (beam size 1), for the three metrics presented in Section 4.4. In Table 2, we report our results with beam search decoding, and compare TransCoder to existing baselines. We give an example of unsupervised translation from Python to C++ in Figure 2.

Evaluation metric differences. In Table 1, we observe that a very large fraction of translations differ from the reference, and are considered as invalid by the reference match metric although they

[5] `https://github.com/c2nes/javalang`
[6] `https://docs.python.org/3/library/tokenize.html`
[7] `https://pypi.org/project/clang`
[8] `https://github.com/glample/fastBPE`
[9] `https://practice.geeksforgeeks.org`

Python input

```python
def SumOfKsubArray(arr, n, k):
  Sum = 0
  S = deque()
  G = deque()
  for i in range(k):
    while (len(S) > 0 and arr[S[-1]] >= arr[i]):
      S.pop()
    while (len(G) > 0 and arr[G[-1]] <= arr[i]):
      G.pop()
    G.append(i)
    S.append(i)
  for i in range(k, n):
    Sum += arr[S[0]] + arr[G[0]]
    while (len(S) > 0 and S[0] <= i - k):
      S.popleft()
    while (len(G) > 0 and G[0] <= i - k):
      G.popleft()
    while (len(S) > 0 and arr[S[-1]] >= arr[i]):
      S.pop()
    while (len(G) > 0 and arr[G[-1]] <= arr[i]):
      G.pop()
    G.append(i)
    S.append(i)
  Sum += arr[S[0]] + arr[G[0]]
  return Sum
```

Unsupervised C++ translation

```cpp
int SumOfKsubArray(int arr[], int n, int k){
  int Sum = 0;
  deque <int> S;
  deque <int> G;
  for(int i = 0; i < k; i ++){
    while((int) S.size() > 0 && arr[S.back()] >= arr[i])
      S.pop_back();
    while((int) G.size() > 0 && arr[G.back()] <= arr[i])
      G.pop_back();
    G.push_back(i);
    S.push_back(i);
  }
  for(int i = k; i < n; i ++){
    Sum += arr[S.front()] + arr[G.front()];
    while((int) S.size() > 0 && S.front() <= i - k)
      S.pop_front();
    while((int) G.size() > 0 && G.front() <= i - k)
      G.pop_front();
    while((int) S.size() > 0 && arr[S.back()] >= arr[i])
      S.pop_back();
    while((int) G.size() > 0 && arr[G.back()] <= arr[i])
      G.pop_back();
    G.push_back(i);
    S.push_back(i);
  }
  Sum += arr[S.front()] + arr[G.front()];
  return Sum;
}
```

Figure 2: **Example of unsupervised Python to C++ translation**. TransCoder successfully translates the Python input function `SumOfKsubArray` into C++. TransCoder infers the types of the arguments, of the variables, and the return type of the function. The model maps the Python `deque()` container, to the C++ implementation `deque<>`, and uses the associated `front`, `back`, `pop_back` and `push_back` methods to retrieve and insert elements into the `deque`, instead of the Python square brackets `[]`, `pop` and `append` methods. Moreover, it converts the Python `for` loop and `range` function properly.

successfully pass the unit tests. For instance, when translating from C++ to Java, only 3.1% of the generations are strictly identical to the ground truth reference, although 60.9% of them return the expected outputs. Moreover, the performance in terms of BLEU is relatively flat and does not correlate well with the computational accuracy. These results highlight the issues with the traditional reference match and BLEU metrics commonly used in the field.

Beam search decoding. In Table 2, we study the impact of beam search, either by considering all hypotheses in the beam that pass the unit tests (Beam N) or by only considering the ones with the highest log-probabilities (Beam N - Top 1). Compared to greedy decoding (Beam 1), beam search significantly improves the computational accuracy, by up to 33.7% in Java → Python with Beam 25. When the model only returns the hypothesis with the highest log-probability, the performance drops, indicating that TransCoder often finds a valid translation, although it sometimes gives a higher log-probability to incorrect hypotheses. More generally, beam search allows minor variations of the translations which can make the unit tests succeed, such as changing the return or variable types in Java and C++, or fixing small errors such as the use of / instead of the // operator in Python. More examples of errors corrected by beam search are presented in Figure 9 in the appendix.

In a real use-case, checking whether the generated functions are syntactically correct and compile, or creating unit tests from the input function would be better approaches than comparing log-probabilities in order to select an hypothesis from the beam. Table 5 in the appendix shows that many failures

Table 1: **Results of TransCoder on our test set with greedy decoding.** We evaluate TransCoder with different metrics: reference match, BLEU score, and computational accuracy. Only 3.1% of C++ to Java translations match the ground truth reference, although 60.9% of them successfully pass the unit tests, suggesting that reference match is not an accurate metric to evaluate the quality of translations. Similarly, the BLEU score does not correlate well with the computational accuracy.

	C++ → Java	C++ → Python	Java → C++	Java → Python	Python → C++	Python → Java
Reference Match	3.1	6.7	24.7	3.7	4.9	0.8
BLEU	85.4	70.1	97.0	68.1	65.4	64.6
Computational Accuracy	60.9	44.5	80.9	35.0	32.2	24.7

Table 2: **Computational accuracy with beam search decoding and comparison to baselines.** Increasing the beam size improves the performance by up to 33.7% in Java $\rightarrow$ Python. When the model only returns the hypothesis with the highest log-probability (Beam 10 - Top 1), the performance drops, indicating that the model often finds a correct translation, although it does not necessarily assign it with the highest probability. TransCoder significantly outperforms the Java $\rightarrow$ Python baseline (+30.4%) and the commercial C++ $\rightarrow$ Java baseline (+13.9%), although it is trained in a fully unsupervised manner and does not leverage human knowledge.

	C++ $\rightarrow$ Java	C++ $\rightarrow$ Python	Java $\rightarrow$ C++	Java $\rightarrow$ Python	Python $\rightarrow$ C++	Python $\rightarrow$ Java
Baselines	61.0	-	-	38.3	-	-
TransCoder Beam 1	60.9	44.5	80.9	35.0	32.2	24.7
TransCoder Beam 5	70.7	58.3	86.9	60.0	44.4	44.3
TransCoder Beam 10	73.4	62.0	89.3	64.4	49.6	51.1
TransCoder Beam 10 - Top 1	65.1	46.9	79.8	49.0	32.4	36.6
TransCoder Beam 25	74.8	67.2	91.6	68.7	57.3	56.1

come from compilation errors when the target language is Java or C++. It suggests that the "Beam N - Top 1" metric could easily be improved. We leave this to future work.

Comparison to existing baselines. We compare TransCoder with two existing approaches: j2py[10], a framework that translates from Java to Python, and a commercial solution from Tangible Software Solutions[11], that translates from C++ to Java. Both systems rely on rewrite rules manually built using expert knowledge. The latter handles the conversion of many elements, including core types, arrays, some collections (Vectors and Maps), and lambdas. In Table 2, we observe that TransCoder significantly outperforms both baselines in terms of computational accuracy, with 74.8% and 68.7% in the C++ $\rightarrow$ Java and Java $\rightarrow$ Python directions, compared to 61% and 38.3% for the baselines. TransCoder particularly shines when translating functions from the standard library. In rule-based transcompilers, rewrite rules need to be manually encoded for each standard library function, while TransCoder learns them in an unsupervised way. In Figure 10 of the appendix, we present several examples where TransCoder succeeds, while the baselines fail to generate correct translations.

4.6 Discussion - Analysis

In Figure 2, we give an example of TransCoder unsupervised translation from C++ to Java. Additional examples can be found in Figure 6 and Figure 7 of the appendix. We observe that TransCoder successfully understands the syntax specific to each language, learns data structures and their methods, and correctly aligns libraries across programming languages. For instance, it learns to translate the ternary operator "`X ? A : B`" in C++ or Java to "`if X then A else B`" in Python, in an unsupervised way. In Figure 5 of the appendix, we present a t-SNE [35] visualization of cross-lingual token embeddings learned by the model. TransCoder successfully map tokens with similar meaning to the same latent representation, regardless of their languages. Figure 8 of the appendix shows that TransCoder can adapt to small modifications. For instance, renaming a variable in the input may result in different translated types, still with valid translations. In Figure 11, we present some typical failure cases where TransCoder fails to account for the variable type during generation. For instance, it copies the C++ NOT operator `!` applied to an integer in Java, while it should be translated to `~`. It also translates the Python `min` function on lists to `Math.min` in Java, which is incorrect when applied to Java arrays. Finally, Table 5 gives detailed results on failure cases.

5 Conclusion

In this paper, we show that approaches of unsupervised machine translation can be applied to source code to create a transcompiler in a fully unsupervised way. TransCoder can easily be generalized to any programming language, does not require any expert knowledge, and outperforms commercial solutions by a large margin. Our results suggest that a lot of mistakes made by the model could easily be fixed by adding simple constraints to the decoder to ensure that the generated functions are syntactically correct, or by using dedicated architectures [12]. Leveraging the compiler output or other approaches such as iterative error correction [16] could also boost the performance.

[10]`https://github.com/natural/java2python`

[11]`https://www.tangiblesoftwaresolutions.com/`

References

[1] Karan Aggarwal, Mohammad Salameh, and Abram Hindle. Using machine translation for converting python 2 to python 3 code. Technical report, PeerJ PrePrints, 2015.

[2] Miltiadis Allamanis, Earl T Barr, Christian Bird, and Charles Sutton. Learning natural coding conventions. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 281–293, 2014.

[3] Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. code2seq: Generating sequences from structured representations of code. *ICLR*, 2019.

[4] Uri Alon, Roy Sadaka, Omer Levy, and Eran Yahav. Structural language models for any-code generation. *arXiv preprint arXiv:1910.00577*, 2019.

[5] Matthew Amodio, Swarat Chaudhuri, and Thomas Reps. Neural attribute machines for program generation. *arXiv preprint arXiv:1705.09231*, 2017.

[6] Mikel Artetxe, Gorka Labaka, and Eneko Agirre. Learning bilingual word embeddings with (almost) no bilingual data. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, volume 1, pages 451–462, 2017.

[7] Mikel Artetxe, Gorka Labaka, and Eneko Agirre. Unsupervised statistical machine translation. *arXiv preprint arXiv:1809.01272*, 2018.

[8] Mikel Artetxe, Gorka Labaka, Eneko Agirre, and Kyunghyun Cho. Unsupervised neural machine translation. In *International Conference on Learning Representations (ICLR)*, 2018.

[9] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

[10] Antonio Valerio Miceli Barone and Rico Sennrich. A parallel corpus of python functions and documentation strings for automated code documentation and code generation. *arXiv preprint arXiv:1707.02275*, 2017.

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

[12] Xinyun Chen, Chang Liu, and Dawn Song. Tree-to-tree neural networks for program translation. In *Advances in neural information processing systems*, pages 2547–2557, 2018.

[13] Zimin Chen, Steve James Kommrusch, Michele Tufano, Louis-Noël Pouchet, Denys Poshyvanyk, and Martin Monperrus. Sequencer: Sequence-to-sequence learning for end-to-end program repair. *IEEE Transactions on Software Engineering*, 2019.

[14] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *CoRR*, abs/1810.04805, 2018.

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

[16] Cheng Fu, Huili Chen, Haolan Liu, Xinyun Chen, Yuandong Tian, Farinaz Koushanfar, and Jishen Zhao. Coda: An end-to-end neural program decompiler. In *Advances in Neural Information Processing Systems*, pages 3703–3714, 2019.

[17] Caglar Gulcehre, Orhan Firat, Kelvin Xu, Kyunghyun Cho, Loic Barrault, Huei-Chi Lin, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. On using monolingual corpora in neural machine translation. *arXiv preprint arXiv:1503.03535*, 2015.

[18] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. Deepfix: Fixing common c language errors by deep learning. In *Thirty-First AAAI Conference on Artificial Intelligence*, 2017.

[19] Francisco Guzmán, Peng-Jen Chen, Myle Ott, Juan Pino, Guillaume Lample, Philipp Koehn, Vishrav Chaudhary, and Marc'Aurelio Ranzato. Two new evaluation datasets for low-resource machine translation: Nepali-english and sinhala-english. *arXiv preprint arXiv:1902.01382*, 2019.

[20] Di He, Yingce Xia, Tao Qin, Liwei Wang, Nenghai Yu, Tie-Yan Liu, and Wei-Ying Ma. Dual learning for machine translation. In *Advances in neural information processing systems*, pages 820–828, 2016.

[21] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. Deep code comment generation. In *Proceedings of the 26th Conference on Program Comprehension*, pages 200–210, 2018.

[22] Svetoslav Karaivanov, Veselin Raychev, and Martin Vechev. 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, 2014.

[23] Deborah S Katz, Jason Ruchti, and Eric Schulte. Using recurrent neural networks for decompilation. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 346–356. IEEE, 2018.

[24] Omer Katz, Yuval Olshaker, Yoav Goldberg, and Eran Yahav. Towards neural decompilation. *arXiv preprint arXiv:1905.08325*, 2019.

[25] Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[26] Philipp Koehn. Pharaoh: a beam search decoder for phrase-based statistical machine translation models. In *Conference of the Association for Machine Translation in the Americas*, pages 115–124. Springer, 2004.

[27] Philipp Koehn, Hieu Hoang, Alexandra Birch, Chris Callison-Burch, Marcello Federico, Nicola Bertoldi, Brooke Cowan, Wade Shen, Christine Moran, Richard Zens, Ondrej Bojar Chris Dyer, Alexandra Constantin, and Evan Herbst. Moses: Open source toolkit for statistical machine translation. In *Annual Meeting of the Association for Computational Linguistics (ACL), demo session*, 2007.

[28] Taku Kudo and John Richardson. Sentencepiece: A simple and language independent subword tokenizer and detokenizer for neural text processing. *arXiv preprint arXiv:1808.06226*, 2018.

[29] Guillaume Lample and Alexis Conneau. Cross-lingual language model pretraining. *arXiv preprint arXiv:1901.07291*, 2019.

[30] Guillaume Lample, Alexis Conneau, Ludovic Denoyer, and Marc'Aurelio Ranzato. Unsupervised machine translation using monolingual corpora only. *ICLR*, 2018.

[31] Guillaume Lample, Alexis Conneau, Marc'Aurelio Ranzato, Ludovic Denoyer, and Hervé Jégou. Word translation without parallel data. In *ICLR*, 2018.

[32] Guillaume Lample, Myle Ott, Alexis Conneau, Ludovic Denoyer, and Marc'Aurelio Ranzato. Phrase-based & neural unsupervised machine translation. In *EMNLP*, 2018.

[33] Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Ves Stoyanov, and Luke Zettlemoyer. Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. *arXiv preprint arXiv:1910.13461*, 2019.

[34] Jian Li, Yue Wang, Michael R Lyu, and Irwin King. Code completion with neural attention and pointer networks. *IJCAI*, 2018.

[35] Laurens van der Maaten and Geoffrey Hinton. Visualizing data using t-sne. *Journal of machine learning research*, 9(Nov):2579–2605, 2008.

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

[37] Yusuke Oda, Hiroyuki Fudaba, Graham Neubig, Hideaki Hata, Sakriani Sakti, Tomoki Toda, and Satoshi Nakamura. Learning to generate pseudo-code from source code using statistical machine translation (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 574–584. IEEE, 2015.

[38] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting on association for computational linguistics*, pages 311–318. Association for Computational Linguistics, 2002.

[39] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. *NIPS 2017 Autodiff Workshop*, 2017.

[40] Maxim Rabinovich, Mitchell Stern, and Dan Klein. Abstract syntax networks for code generation and semantic parsing. *arXiv preprint arXiv:1704.07535*, 2017.

[41] Rico Sennrich, Barry Haddow, and Alexandra Birch. Improving neural machine translation models with monolingual data. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics*, pages 86–96, 2015.

[42] Rico Sennrich, Barry Haddow, and Alexandra Birch. Neural machine translation of rare words with subword units. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics*, pages 1715–1725, 2015.

[43] Kaitao Song, Xu Tan, Tao Qin, Jianfeng Lu, and Tie-Yan Liu. Mass: Masked sequence to sequence pre-training for language generation. In *International Conference on Machine Learning*, pages 5926–5936, 2019.

[44] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pages 3104–3112, 2014.

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

[46] Pascal Vincent, Hugo Larochelle, Yoshua Bengio, and Pierre-Antoine Manzagol. Extracting and composing robust features with denoising autoencoders. In *Proceedings of the 25th international conference on Machine learning*, pages 1096–1103, 2008.

[47] Ke Wang, Rishabh Singh, and Zhendong Su. Dynamic neural program embedding for program repair. *arXiv preprint arXiv:1711.07163*, 2017.

[48] Pengcheng Yin and Graham Neubig. A syntactic neural model for general-purpose code generation. *arXiv preprint arXiv:1704.01696*, 2017.

[49] Hao Zheng, Yong Cheng, and Yang Liu. Maximum expected likelihood estimation for zero-resource neural machine translation. In *IJCAI*, 2017.

A Data and preprocessing

A.1 Training dataset

We tried removing and keeping the comments in the code from our training data. As shown in Table 6, keeping the comments gives better results overall. Thus, we decided to keep them in our final training data. Detailed statistics of the resulting dataset can be found in Table 3.

Table 3: **Statistics of our GitHub dataset.** We show the statistic for our entire github dataset (All) and for the extracted functions. We give the size in GigaBytes, the number of files and functions, and the number of tokens.

	C++	Java	Python
All - Size	168 GB	352 GB	224 GB
All - Nb of files	15 M	56 M	18 M
All - Nb of tokens	38 B	75 B	50 B
Functions - Size	93 GB	185 GB	152 GB
Functions - Nb of functions	120 M	402 M	217 M

A.2 Tokenization

Python function v1

```
def rm_file(path):
    try:
        os.remove(path)
        print("Deleted")
    except:
        print("Error while deleting file", path)
```

Python function v2

```
def rm_file(path):

    try:
        os.remove( path )
        print( "Deleted" )
    except  :
        print("Error while deleting file", path)
```

```
def rm_file ( path ) : NEWLINE try : NEWLINE INDENT os . remove (path) NEWLINE print ( " Deleted " )
DEDENT except : NEWLINE INDENT print ( " Error ▁ while ▁ deleting ▁ file " , path ) DEDENT
```

Figure 3: **Example of function tokenization.** We show two versions of the same Python function and their common tokenization. These function versions differ by extra spaces and one extra new line. Our Python tokenizer is robust to extra spaces and extra new lines except in strings. In strings, spaces are tokenized as ▁ (U+2581). Indentation is meaningful in Python: indented blocks are surrounded by INDENT DEDENT tokens.

A.3 Function extraction

We train and evaluate our translation model on functions only. We differentiate class functions and standalone functions. By standalone functions, we refer to functions that can be used without instantiating a class. In C++ and Python, this corresponds to static methods of classes, and functions outside classes. In Java, it only corresponds to static methods. In GeeksforGeeks, solutions are implemented with standalone functions, and our evaluation protocol only involves these functions. In Table 3, the functions statistics are given for all kind of functions. In C++ and Python, 50% of functions are standalone functions. In Java, standalone functions only represent 15% of the dataset. We tried to train our model on standalone functions only, and observed better results than when training on all functions. Thus, all the results in this work are given for models pretrained on all available data and trained on standalone functions only.

B Evaluation

GeeksforGeeks is an online platform with computer science and programming articles. It contains many coding problems and presents solutions in several programming languages. We gather all the problems for which some solutions are implemented in C++, Java, and Python. The parallel data for these problems is already enough to test a model using the BLEU score or the Reference Match score. However, we need to generate some unit tests to check that the function are semantically correct and to compute the Computational Accuracy.

These unit tests are contained in a script, which contains a reference function — named `f_gold` — from the parallel dataset, a commented `TOFILL` marker which is to be replaced with a generated function, and a main which runs both functions on a series of inputs and compares the behaviors of the two functions. We have one script per function and per programming language.

In order to generate these scripts, we extract the parameters and their types from the Java implementation of the solution. Then, we generate 10 random inputs for these types, which are hardcoded in the test script and used to test the function. We test the generated scripts by injecting the reference function a second time with the name `f_filled` instead of the `TOFILL` comment and running it. We keep only the scripts that return a perfect score in less than 10 seconds. As Python is dynamically typed, we need to infer the Python parameters types from the Java types, and to assume that the order and types of the parameters is the same in Java and Python. When this assumption happens to be wrong, the generated script fails the tests and is discarded. As this approach is quite effective, we generated the C++ scripts in a similar manner and barely use the C++ parameter types which can be extracted from the function definition.

Equality tests. We adapt the tests checking that the reference and gold function behave in the same way based on the output type of the function (extracted from its Java implementation). For instance, we test the equality of `int` outputs with `==`, while we use `equals` for `String` outputs and relative tests for `double` outputs. If the function is inplace (the output type is `void`), we check the side effects on all its mutable arguments instead.

Special cases for random input generation. The goal of our scripts is to decide whether a function is semantically equivalent to from the reference function, and the way we generate the random inputs is critical to how discriminative the script will be. For instance, if the input of the reference function is a string, a naive solution may be to generate strings of random length and with characters sampled randomly from the set of all characters. However, our dataset contains several functions such as `checkDivisibility` in Figure 4 which considers the string to be a representation of a long integer. This type of function could always return the same result (e.g. `False`) on inputs strings that do not contain only digits. As many functions in our dataset assume the input strings or characters to be representations of long integers or representations of integers in base 2, we alternate between sampling the characters from (i) the set of all lowercase and uppercase letters plus the space character, (ii) the set of all digits, and (iii) the set containing 0 and 1. For similar reasons, when there is an integer array in the function parameters, we alternate between the sets $\{0 \dots 100\}$, $\{-100 \dots 100\}$ and $\{0, 1\}$ to sample the integers inside the array. When the function takes no argument, we do not generate any input for it and only check that the output is the same for the reference function and the generated function.

Manual verifications. In order to ensure that our unit tests are appropriate, we manually check and modify the scripts when the output of the function is the same on all 10 inputs, when the function is inplace, or when the function contains prints. As we only check the side effects affecting the mutable arguments, we remove all the functions which mainly print or write to a file.

C++

```cpp
bool checkDivisibility(string num){
  int length = num.size();
  if(length == 1 && num[0] == '0')
    return true;
  if(length % 3 == 1){
    num += "00";
    length += 2;
  }
  else if(length % 3 == 2){
    num += '0';
    length += 1;
  }

  int sum = 0, p = 1;
  for(int i = length - 1;
              i >= 0; i--){
    int group = 0;
    group += num[i--] - '0';
    group += (num[i--] - '0') * 10;
    group += (num[i] - '0') * 100;
    sum = sum + group * p;
    p *= (-1);
  }

  sum = abs(sum);
  return (sum % 13 == 0);
}
```

Java

```java
static boolean checkDivisibility(
                String num){
  int length = num.length();
  if(length == 1 && num.charAt(0) == '0')
    return true;
  if(length % 3 == 1){
    num += "00";
    length += 2;
  }
  else if(length % 3 == 2){
    num += "0";
    length += 1;
  }

  int sum = 0, p = 1;
  for(int i = length - 1; i >= 0; i--){
    int group = 0;
    group += num.charAt(i--) - '0';
    group += (num.charAt(i--) - '0') * 10;
    group += (num.charAt(i) - '0') * 100;
    sum = sum + group * p;
    p *= (-1);
  }

  sum = Math.abs(sum);
  return (sum % 13 == 0);
}
```

Python

```python
def checkDivisibility(num):
  length = len(num)
  if(length == 1 and num[0] == '0'):
    return True
  if(length % 3 == 1):
    num = str(num) + "00"
    length += 2
  elif(length % 3 == 2):
    num = str(num) + "0"
    length += 1

  sum = 0
  p = 1
  for i in range(length - 1, -1, -1):
    group = 0
    group += ord(num[i]) - ord('0')
    i -= 1
    group += (ord(num[i]) - ord('0')) * 10
    i -= 1
    group += (ord(num[i]) - ord('0')) * 100
    sum = sum + group * p
    p *= (-1)

  sum = abs(sum)
  return (sum % 13 == 0)
```

Figure 4: **Example of parallel function from our test set.** We extracted parallel functions from GeeksforGeeks to create validation and test sets. Here, we have the parallel implementations in C++, Java, and Python of the checkDivisibility function, which determines whether a long integer represented as a string is divisible by 13.

Table 4: **Number of functions with unit tests for our validation and test sets.** We report the number of function with unit tests for C++, Java, and Python, for the validation and test sets. We also show the average number of tokens per function. A unit test checks whether a generated function is semantically equivalent to its reference. For each function, we have 10 unit tests, each testing it on a different input. As a result, the number of functions with unit tests per language gives the size of the validation and test sets of each pair of languages. For instance, we have 231 C++ functions with unit tests for the validation set, which means that we have a validation set of 231 functions for Java $\rightarrow$ C++ and Python $\rightarrow$ C++.

	C++	Java	Python
Nb of functions with unit tests - valid set	231	234	237
Nb of functions with unit tests - test set	466	481	463
Average #tokens per function	105.8	112.0	103.1

C Results

C.1 Detailed results

Table 5: **Detailed results for greedy decoding.** Many failures come from compilation errors when the target language is Java or C++. It suggests that our method could be improved by constraining the decoder to generate compilable code. Runtime errors mainly occur when translating from Java or C++ into Python. Since Python code is interpreted and not compiled, this category also includes syntax errors in Python. The majority of remaining errors are due to the program returning the wrong output on one or several of the unit tests. Timeout errors are generally caused by infinite loops and mainly occur in the Java $\leftrightarrow$ Python pair.

	#tests	Success	Compilation	Runtime	Wrong Output	Timeout
C++ $\rightarrow$ Java	481	60.9%	27.2%	4.4%	5.4%	2.1%
C++ $\rightarrow$ Python	463	44.5%	0.0%	36.5%	18.1%	0.9%
Java $\rightarrow$ C++	466	80.9%	10.3%	1.1%	7.5%	0.2%
Java $\rightarrow$ Python	463	35.0%	0.0%	31.8%	15.6%	17.7%
Python $\rightarrow$ C++	466	32.2%	29.0%	4.9%	32.6%	1.3%
Python $\rightarrow$ Java	481	24.7%	23.5%	12.5%	24.3%	15.0%

C.2 Ablation study

Table 6: **Training data ablation study - with and without code comments.** We compare the computational accuracy of TransCoder for different training sets, where we either keep or remove comments from source code training data. We give results for different beam sizes. When translating from C++ to Python, from Java to C++ and from Java to Python, keeping comments in the training set gives better results. In the other directions, keeping or removing comments does not have a significant impact on the performance.

	C++ $\rightarrow$ Java		C++ $\rightarrow$ Python		Java $\rightarrow$ C++		Java $\rightarrow$ Python		Python $\rightarrow$ C++		Python $\rightarrow$ Java	
With Comments	No	Yes	No	Yes	No	Yes	No	Yes	No	Yes	No	Yes
Beam 1	62.2	60.9	40.8	44.5	76.8	80.9	46.4	35.0	34.1	32.2	33.9	24.7
Beam 5	71.6	70.7	54.0	58.3	85.6	86.9	58.5	60.0	46.4	44.4	46.0	44.3
Beam 10	73.6	73.4	57.9	62.0	88.4	89.3	62.9	64.4	50.9	49.6	50.3	51.1
Beam 25	75.3	74.8	64.6	67.2	89.1	91.6	66.7	68.7	56.7	57.3	56.3	56.1

C.3 Cross-lingual token embedding space

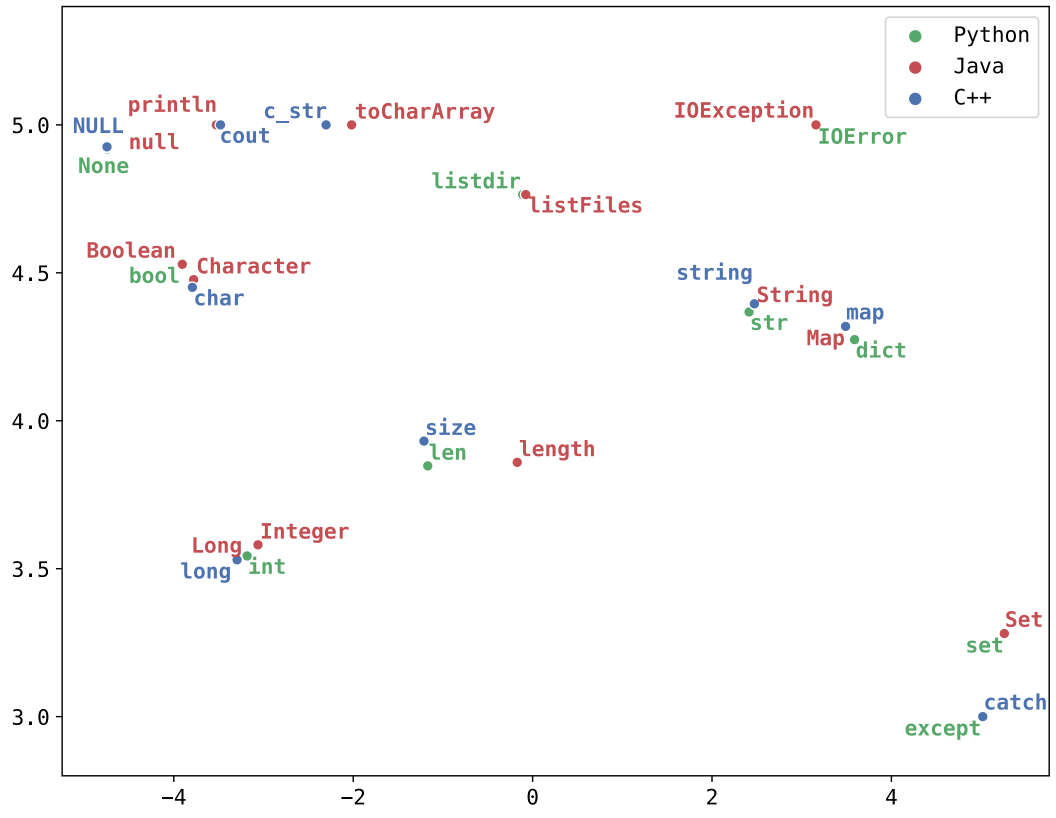


Figure 5: **Cross-lingual token embedding space.** We show a t-SNE visualization of our cross-lingual token embeddings. These embeddings are obtained by encoding programming language tokens into TransCoder's lookup table. We show the embeddings of C++, Java, and Python keywords. Keywords of different programming languages that are used in similar contexts are very close in the embedding space. For instance, `except` in Python and `catch` in Java and C++, which are both used to catch exceptions, are mapped to very similar embeddings. The same phenomenon is observed for implementations of maps (`Map`, `map` and `dict`), for `c_str` and `toCharArray` which are used to transform a string into a char array, and for similar primitive types (e.g. `Long`, `long`, `Integer`, and `int`).

C.4 Additional examples of translations

C++ Input

```
int maxLen(int arr[], int n){
  unordered_map<int, int> hM;
  int sum = 0;
  int max_len = 0;
  int ending_index = - 1;
  for(int i = 0; i < n; i++)
    arr[i] = (arr[i] == 0) ? -1 : 1;
  for(int i = 0; i < n; i++){
    sum += arr[i];
    if(sum == 0){
      max_len = i + 1;
      ending_index = i;
    }
    if(hM.find(sum + n) != hM.end()){
      if(max_len < i - hM[sum + n]){
        max_len = i - hM[sum + n];
        ending_index = i;
      }
    }
    else hM[sum + n] = i;
  }
  for(int i = 0; i < n; i++)
    arr[i] = (arr[i] == -1) ? 0 : 1;
  printf("%d to %d\n",
          ending_index - max_len + 1,
          ending_index);
  return max_len;
}
```

Unsupervised Java Translation

```
public static int maxLen(int arr[], int n){
  HashMap<Integer, Integer> hM
        = new HashMap<Integer, Integer>();
  int sum = 0;
  int maxLen = 0;
  int endingIndex = - 1;
  for(int i = 0; i < n; i++){
    arr[i] = (arr[i] == 0) ? - 1 : 1;
  }
  for(int i = 0; i < n; i++){
    sum += arr[i];
    if(sum == 0){
      maxLen = i + 1;
      endingIndex = i;
    }
    if(hM.containsKey(sum + n)){
      if(maxLen < i - hM.get(sum + n)){
        maxLen = i - hM.get(sum + n);
        endingIndex = i;
      }
    }else{
      hM.put(sum + n, i);
    }
  }
  for(int i = 0; i < n; i ++){
    arr[i] = (arr[i] == -1) ? 0 : 1;
  }
  System.out.println(endingIndex - maxLen + 1
                   + " to " + endingIndex);
  return maxLen;
}
```

```
std::unordered_set<int> uniqueElements(
                    const std::vector<int>& l){
  return std::unordered_set<int>(l.begin(), l.end());
 }
```

```
public static Set<Integer> uniqueElements(
                    Collection<Integer> l){
  return new HashSet<Integer>(l);
 }
```

```
std::vector<int> squareList(const std::vector<int>& l){
  std::vector<int> result = {};
  for(int elmt : l){
    result.push_back(elmt * elmt);
  }
  return result;
 }
```

```
public static List<Integer> squareList(
                    List<Integer> l){
  List<Integer> result = new ArrayList<>();
  for(Integer elmt : l) result.add(elmt*elmt);
  return result;
}
```

Figure 6: **Examples of correct translations using TransCoder**. In all these examples, TransCoder converts properly the primitive types, find the equivalent data structures and the corresponding methods, and libraries between languages. For instance, in the `maxLen` function, TransCoder converts the C++ `unordered_map` container into the Java implementation of the `Map` interface, `HashMap`, and properly uses the associated `get` and `put` methods to retrieve and insert keys and values into the map, instead of the C++ square brackets `[]`.

Input | TransCoder Translation

C++

```cpp
void worstFit(int blockSize[], int m, int processSize[], int n){
  int allocation[n];
  memset(allocation, -1, sizeof(allocation));
  for(int i = 0; i < n; i ++){
    int wstIdx = - 1;
    for(int j = 0; j < m; j++){
      if(blockSize[j] >= processSize[i]){
        if(wstIdx == - 1)
          wstIdx = j;
        else if(blockSize[wstIdx] < blockSize[j])
          wstIdx = j;
      }
    }
    if(wstIdx != -1){
      allocation[i] = wstIdx;
      blockSize[wstIdx] -= processSize[i];
    }
  }
  cout << "\nProcess No.\tProcess Size\tBlock no.\n";
  for(int i = 0; i < n; i++){
    cout << "   " << i+1 << "\t\t" << processSize[i] << "\t\t";
    if(allocation[i] != -1) cout << allocation[i] + 1;
    else
      cout << "Not Allocated";
  cout << endl;
  }
}
```

Python

```python
def worstFit(blockSize, m, processSize, n):
  allocation = [-1] * n
  for i in range(n):
    wstIdx = -1
    for j in range(m):
      if blockSize[j] >= processSize[i]:
        if wstIdx == -1:
          wstIdx = j
        elif blockSize[wstIdx] < blockSize[j]:
          wstIdx = j
    if wstIdx != -1:
      allocation[i] = wstIdx
      blockSize[wstIdx] -= processSize[i]
  print("\nProcess No.\tProcess Size\tBlock no.\n")
  for i in range(n):
    print("   " + str(i+1) + "\t\t" +
          str(processSize[i]) + "\t\t")
    if allocation[i] != -1:
        print(allocation[i] + 1)
    else:
        print("Not Allocated")
    print()
```

Java

```java
public static int max(int a, int b){
  return a > b ? a : b;
}

public static void createDirectory(Path path)
            throws IOException{
  if(!Files.exists(path)){
    Files.createDirectories(path);
  }
}
```

Python

```python
def max(a, b):
  return a if a > b else b


def create_directory(path):
  if not os.path.exists(path):
    os.makedirs(path)
```

Python

```python
def sum_elements(values):
    return sum(values)




def no_letters(s):
  return s.lower() == s.upper()



def get_env_variable(name):
  return os.environ[name]
```

C++

```cpp
template <typename T> T
sum_elements(const std::vector<T> &values){
  return std::accumulate(values.begin(), values.end(),
                          0);
}

static bool noLetters(const QString& str){
  return str.toLower() == str.toUpper();
}

const char* GetEnvVariable(const char* name){
  return getenv(name);
}
```

Python

```python
def calcMaxValue(str):
  res = ord(str[0]) - 48
  for i in range(1, len(str)):
    if(str[i] == '0' or str[i] == '1' or res < 2):
      res += ord(str[i]) - 48
    else:
      res *= ord(str[i]) - 48
  return res











def foo(x):
  return bar(x) + 1



def area(r):
  return 2 * PI * r ** 2
```

Java

```java
public static int calcMaxValue(String str){
  int res = (int)(str.charAt(0) - 48);
  for(int i = 1; i < str.length(); i++){
    if(str.charAt(i) == '0'
        || str.charAt(i) == '1'
        || res < 2){
      res += (int)(str.charAt(i) - 48);
    }
    else{
      res *= (int)(str.charAt(i) - 48);
    }
  }
  return res;
}

public static int foo(int x){
  return bar(x) + 1;
}

public static double area(double r){
  return 2 * PI * r * r;
}
```

Figure 7: **Examples of correct translations from or to Python using TransCoder**. When translating from Python, TransCoder successfully infers types. Here, TransCoder infers the Python `list` type and translates it into its C++ equivalent `std::vector`. The last two examples show that TransCoder does not modify the call to the non-standard function `bar` or the global variable `PI`.

C++ input

```cpp
int minPalPartion(char* str){
  int n = strlen(str);
  int C[n][n];
  bool P[n][n];
  int i, j, k, L;
  for(i = 0; i < n; i ++){
    P[i][i] = true;
    C[i][i] = 0;}
  for(L = 2; L <= n; L ++){
    for(i = 0; i < n - L + 1; i ++){
      j = i + L - 1;
      if(L == 2) P[i][j] = (str[i] == str[j]);
      else P[i][j] = (str[i] == str[j])
                    && P[i+1][j-1];
      if(P[i][j] == true) C[i][j] = 0;
      else{
        C[i][j] = INT_MAX;
        for(k = i; k <= j - 1; k ++)
            C[i][j] = min(C[i][j],
                        C[i][k]+C[k+1][j]+1);
      }
    }
  }
  return C[0][n - 1];
}
```

TransCoder Java translation

```java
public static int minPalPartion(String str){
  int n = str.length();
  int[][] C = new int[n][n];
  boolean[][] P = new boolean[n][n];
  int i , j , k , L;
  for(i = 0; i < n; i ++){
    P[i][i] = true;
    C[i][i] = 0;}
  for(L = 2; L <= n; L ++){
    for(i = 0; i < n - L + 1;  i ++){
      j = i + L - 1;
      if(L == 2){
        P[i][j] = (str.charAt(i) == str.charAt(j));}
      else{
        P[i][j] = (str.charAt(i) == str.charAt(j))
                    && P[i + 1][j - 1];}
      if(P[i][j] == true){
        C[i][j] = 0;}
      else{
        C[i][j] = Integer.MAX_VALUE;
        for(k = i; k <= j - 1; k ++){
          C[i][j] = Math.min(C[i][j],
                    C[i][k] + C[k + 1][j] + 1);}}}}
  return C[0][n - 1];
}
```

```cpp
int minPalPartion(char* arr){
  int n = strlen(arr);
  int C[n][n];
  bool P[n][n];
  int i, j, k, L;
  for(i = 0; i < n; i ++){
    P[i][i] = true;
    C[i][i] = 0;}
  for(L = 2; L <= n; L ++){
    for(i = 0; i < n - L + 1; i ++){
      j = i + L - 1;
      if(L == 2) P[i][j] = (arr[i] == arr[j]);
      else P[i][j]= (arr[i] == arr[j])
                    && P[i+1][j-1];
      if(P[i][j] == true) C[i][j] = 0;
      else{
        C[i][j] = INT_MAX;
        for(k = i; k <= j - 1; k ++)
            C[i][j] = min(C[i][j],
                        C[i][k]+C[k+1][j]+1);
      }
    }
  }
  return C[0][n - 1];
}
```

```java
public static int minPalPartion(char[] arr){
  int n = arr.length;
  int[][] C = new int[n][n];
  boolean[][] P = new boolean[n][n];
  int i , j , k , L;
  for(i = 0; i < n; i ++){
    P[i][i] = true;
    C[i][i] = 0;}
  for(L = 2; L <= n; L ++){
    for(i = 0; i < n - L + 1;  i ++){
      j = i + L - 1;
      if(L == 2){
        P[i][j] = (arr[i] == arr[j]);}
      else{
        P[i][j] = (arr[i] == arr[j])
                    && P[i + 1][j - 1];}
      if(P[i][j] == true){
        C[i][j] = 0;}
      else{
        C[i][j] = Integer.MAX_VALUE;
        for(k = i; k <= j - 1; k ++){
          C[i][j] = Math.min(C[i][j],
                    C[i][k] + C[k + 1][j] + 1);}}}}
  return C[0][n - 1];
}
```

```cpp
int minPalPartion(char* input){
  int n = strlen(input);
  int C[n][n];
  bool P[n][n];
  int i, j, k, L;
  for(i = 0; i < n; i ++){
    P[i][i] = true;
    C[i][i] = 0;}
  for(L = 2; L <= n; L ++){
    for(i = 0; i < n - L + 1; i ++){
      j = i + L - 1;
      if(L == 2) P[i][j] = (input[i] == input[j]);
      else P[i][j] = (input[i] == input[j])
                    && P[i+1][j-1];
      if(P[i][j] == true) C[i][j] = 0;
      else{
        C[i][j] = INT_MAX;
        for(k = i; k <= j - 1; k ++)
            C[i][j] = min(C[i][j],
                        C[i][k]+C[k+1][j]+1);
      }
    }
  }
  return C[0][n - 1];
}
```

```java
public static int minPalPartion(String input){
  int n = input.length();
  int[][] C = new int[n][n];
  boolean[][] P = new boolean[n][n];
  int i , j , k , L;
  for(i = 0; i < n; i ++){
    P[i][i] = true;
    C[i][i] = 0;}
  for(L = 2; L <= n; L ++){
    for(i = 0; i < n - L + 1;  i ++){
      j = i + L - 1;
      if(L == 2){
        P[i][j] = (input.charAt(i) == input.charAt(j));
        }
      else{
        P[i][j] = (input.charAt(i) == input.charAt(j))
                    && P[i + 1][j - 1];}
      if(P[i][j] == true){
        C[i][j] = 0;}
      else{
        C[i][j] = Integer.MAX_VALUE;
        for(k = i; k <= j - 1; k ++){
          C[i][j] = Math.min(C[i][j],
                    C[i][k] + C[k + 1][j] + 1);}}}}
  return C[0][n - 1];
}
```

Figure 8: **TransCoder robustness to variable names.** We take the C++ function `minPalPartion`, change the parameter name from `str` to `arr` and `input` and show the three corresponding TransCoder Java translations. All translations are correct. In the first and third cases, TransCoder translates `char* str` and `char* input` into Java `String` and uses the `charAt` method to retrieve elements. This shows that TransCoder is robust to variable name changes and that it remembers variable types along the whole translation to apply the appropriate methods. In the second case, TransCoder translates `char* arr` into Java `char[]` and uses `[]` to retrieve elements, showing that TransCoder can adjust its translations to parameter names while remaining accurate.

C++ input | TransCoder Python translation: Greedy decoding | Beam search

```cpp
int compute_average(int a, int b){
  return (a + b) / 2;
}
```

```python
def compute_average(a, b):
  return (a + b) / 2
```

```python
def compute_average(a, b):
  return (a + b) // 2
```

```cpp
bool isPalindrome(string str){
  int l = 0;
  int h = str.length() - 1;
  while(h > l)
    if(str[l++] != str[h--])
      return false;
  return true;
}
```

```python
def isPalindrome(str):
  l = 0
  h = len(str) - 1
  while h > l:
    if str[l] != str[h--]:
      return False
  return True
```

```python
def isPalindrome(str):
  l = 0
  h = len(str) - 1
  while h > l:
    if str[l] != str[h]:
      return False
    l += 1
    h -= 1
  return True
```

```cpp
int findRotations(string str){
  string tmp = str + str;
  int n = str.length();
  for(int i = 1; i <= n; i ++){
    string substring = tmp.substr(
                    i, str.size());
    if(str == substring) return i;
  }
  return n;
}
```

```python
def findRotations(str):
  tmp = str + str
  n = len(str)
  for i in range(1, n + 1):
    substring = tmp[i :
                    i + len(str)]
    if str == substring:
      return i
  return n
```

```python
def findRotations(str):
  tmp = str + str
  n = len(str)
  for i in range(1, n + 1):
    substring = tmp[i : len(str)]
    if str == substring:
      return i
  return n
```

Figure 9: **Examples of incorrect greedy decoding translations versus correct beam translations.** We take C++ inputs and translate them into Python with TransCoder. In the second column, we use greedy decoding and the translations are incorrect. In the third column, we use beam search and obtain accurate translations. A common error corrected by beam search for C++ $\rightarrow$ Python is the usage of the double instead of the integer division operator (first example). Beam search is also able to correct errors such as the $++$ and $--$ operators that do not exist in Python (second example) or incorrect bounds for a substring (`[i:i+len(str)]` instead of `[i:len(str)]` in the third example).

Input	Baseline translation	TransCoder translation
Java	**Python**	**Python**

`static String getEnvVariable(`
`        String name){`
`  return System.getenv(name);`
` }`
	`def getEnvVariable(name):`
	`  return System.getenv(name)`
		`def getenvh(name):`
		`  return os.environ.get(name, '')`

`static List<Integer> truncate(`
`    List<Integer> l,`
`    Integer e){`
`  if(l.contains(e)){`
`    return l.subList(0,`
`            l.indexOf(e));`
`  }`
`  else{`
`    return l;`
`  }`
` }`
	`def truncate(l, e):`
	`  if l.contains(e):`
	`    return l.subList(0,`
	`             l.indexOf(e))`
	`  else:`
	`    return l`
		`def truncate(l, e):`
		`  if l.count(e) > 0:`
		`    return l[ : l.index(e)]`
		`  else:`
		`    return l`

`static void deleteFile(Path path){`
`  try{`
`    Files.delete(path);`
`  }`
`  catch(Exception e){`
`    System.err.println(`
`    "Error deleting " + path);`
`  }`
`}`
	`def deleteFile(path):`
	`  try:`
	`    Files.delete(path)`
	`  except Exception as e:`
	`    System.err.println(`
	`      "Error deleting " + path)`
		`def delete_file(path):`
		`  try:`
		`    os.remove(path)`
		`  except:`
		`    sys.stderr.write(`
		`        "Error deleting %s\n"`
		`        % path`
		`    )`

C++	**Java**	**Java**

`memset(prime, 0, sizeof(prime));`	`memset(prime, 0,`
	`  (Integer.SIZE/Byte.SIZE));`
		`Arrays.fill(prime, 0);`

`sort(a, a + n);`	`sort(a, a + n);`	`Arrays.sort(a);`
`for(char ch : str)`	`for(char ch : str)`	`for(char ch : str.toCharArray())`

Figure 10: **Examples of incorrect baseline translations versus correct TransCoder translations.** When translating from Java to Python, the baseline fails to translate the `System.getenv`, `System.err.println`, and `Files.delete` functions from the standard library, and the `contains`, `subList`, and `IndexOf` methods of the Java `List` interface. Instead, it simply copies them, showing the limitations of a rule-based system. On the other hand, TransCoder converts properly all of these functions into their Python equivalents. In the C++ $\rightarrow$ Java direction, baseline translations are made at token-level, and are incorrect. For instance, the first example shows that the baseline tries to translate the `sizeof` function, and leaves `memset` unchanged although it does not exist in Java. Instead, TransCoder correctly uses `Arrays.fill` to fill the array `prime` with zeros.

Input	Java failed translations	Description

`bool isEven (int n){`
`  return (!(n & 1));`
`}`
	`static boolean isEven(int n){`
	`  return (!(n & 1));`
	`}`
		The `!` operator works on boolean and integers in C++ (it returns `true` if the integer is positive) but it only works on boolean in Java.

`int summingSeries(long n){`
`  return pow(n, 2);`
`}`
	`static int summingSeries(long n){`
	`  return Math.pow(n, 2);`
	`}`
		In Java, `Math.pow(n, 2)` returns a `double` which should be cast to `int` to match the function return type.

`def minSum(A):`
`    min_val = min(A)`
`    return min_val * (len(A) - 1)`
	`static double minSum(double[] A){`
	`  double minVal = Math.min(A);`
	`  return minVal*(A.length - 1);`
	`}`
		`Math.min` is a Java function but does not take as input a `double[]` array but a pair of `double`.

Figure 11: **Examples of failed TransCoder translations.** TransCoder fails to translate these C++ and Python functions into Java, showing its limitations. In these examples, it fails to account for the variable types when using a method or an operator. In particular, the NOT operator `!` in C++ should have been translated to ~ in Java, because it is applied to an integer. Similarly, the `Math.min` function in Java cannot be applied to arrays.