# NatGen: Generative Pre-training by “Naturalizing” Source Code

Saikat Chakraborty
Columbia University
New York, NY, USA
saikatc@cs.columbia.edu

Toufique Ahmed
University of California, Davis
Davis, CA, USA
tfahmed@ucdavis.edu

Yangruibo Ding
Columbia University
New York, NY, USA
yrbding@cs.columbia.edu

Premkumar T. Devanbu
University of California, Davis
Davis, CA, USA
ptdevanbu@ucdavis.edu

Baishakhi Ray
Columbia University
New York, NY, USA
rayb@cs.columbia.edu

## ABSTRACT

Pre-trained Generative Language models (*e.g.,* PLBART, CodeT5, SPT-Code) for source code yielded strong results on several tasks in the past few years, including code generation and translation. These models have adopted varying pre-training objectives to learn statistics of code construction from very large-scale corpora in a self-supervised fashion; the success of pre-trained models largely hinges on these pre-training objectives. This paper proposes a new pre-training objective, “Naturalizing” of source code, exploiting code’s bimodal, dual-channel (formal & natural channels) nature. Unlike natural language, code’s bimodal, dual-channel nature allows us to generate semantically equivalent code at scale. We introduce six classes of semantic preserving transformations to introduce un-natural forms of code, and then force our model to produce more natural original programs written by developers. Learning to generate equivalent, but more natural code, at scale, over large corpora of open-source code, without explicit manual supervision, helps the model learn to both ingest & generate code. We fine-tune our model in three generative Software Engineering tasks: code generation, code translation, and code refinement with limited human-curated labeled data and achieve state-of-the-art performance rivaling CodeT5. We show that our pre-trained model is especially competitive at zero-shot and few-shot learning, and better at learning code properties (e.g., syntax, data flow).

## CCS CONCEPTS

• **Software and its engineering** → *Language features*; • **Computing methodologies** → **Knowledge representation and reasoning**.

## KEYWORDS

Source Code Pre-training, Neural Network, Source Code Transformer, Semantic Preserving Transformation

**ACM Reference Format:**
Saikat Chakraborty, Toufique Ahmed, Yangruibo Ding, Premkumar T. Devanbu, and Baishakhi Ray. 2022. NatGen: Generative Pre-training by “Naturalizing” Source Code. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE ’22), November 14–18, 2022, Singapore, Singapore.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3540250.3549162

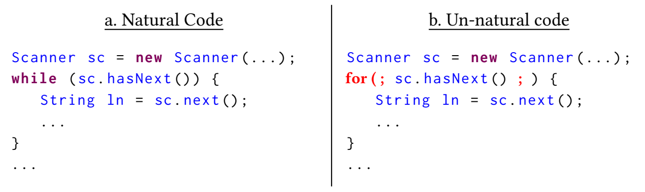
## 1 INTRODUCTION

Statistical models of the “naturalness" of code [33] have proven useful for a range of Software Engineering tasks [7, 50], including code generation [10], repair [15, 61], summarization [40], retrieval [47], and clone detection [20, 66]. The earlier work in this area trained models directly on tasks, including the early work on type recovery [8, 31], de-obfuscation [55, 62], repair [30], and summarization [2, 35]. Training on-task requires a lot of labeled data. While labeled data is abundant for tasks like code completion (where the corpus inherently provides supervision), other tasks like code generation, translation, summarization, repair, etc., require well-curated, high-quality data. Simply grabbing data from Github might yield poor-quality [27], highly-duplicated data [5]. With increasing model capacity (hundreds of millions, even billions of parameters, are pretty common; larger models tend to perform better [17, 64]), this unacceptable disparity between vast model capacity and the limited availability of well-curated, high-quality, labeled data has increased and will likely worsen.

This shortage of high-quality labeled data for on-task training is not unique to Software Engineering (SE), although it is complicated here by the increased, specialized skill required for labeling SE data. To address the issue of training large models in the presence of data scarcity, such models are often pre-trained on some generic tasks, which relate to actual downstream tasks. For example, consider two SE tasks: code generation and code translation. Both tasks require ML models to learn how to generate natural, syntactically, and semantically correct code. This commonality across tasks motivates a quest for better pre-trained models, using a self- (or un-) supervised task which transfers well to other downstream tasks. Such pre-trained models can also learn a generic representation of the input data, which, in turn, transfers to diverse downstream tasks.

A popular approach for dealing with this problem involves derivatives of BERT style models, *e.g.,* CodeBERT [22], GraphCodeBERT [28], etc. These models are good at capturing generic code representations. For code generation tasks, GPT-3 or BART-style models (*e.g.,* Codex, CodeT5, PLBART, SPTCode, etc. [3, 17, 46, 64]) are popular. The important insight here is that independent of final tasks, when *very* high capacity models are trained with huge code corpora to learn simple, self-supervised, "busy work", they still *learn general syntactic and semantic constraints of writing code.* Different approaches adopt different techniques to train the model to write code. For instance, GPT-style models (*e.g.,* Codex) learn to generate code sequentially, mimicking the left-to-right language model. CodeT5 masks out some tokens and asks the model to generate *only* those masked tokens. On the other hand, PLBART and SPT-Code present the model with erroneous code (with deleted or masked tokens) and ask the model to generate the corrected, complete code. The models' ability to generate code depends mainly on the pre-training objective that the model is optimized for.

We propose a novel pre-training task: we ask the model to "naturalize" code, *i.e.,* take "weird", synthetic code as input and output semantic equivalent, "natural" code that a human developer would have written. This is a very demanding pre-training task—the model has to learn both code naturalness *and* code semantics. We were inspired by noting the work of human Editors (of books, journals, newspapers): they digest imperfectly written but mostly correct text, understand the intent, and then produce more perfect text with pretty much the same meaning. Editing is *hard*: a skilled Editor has to have very high levels of language comprehension, to understand given, potentially badly-written text, *and then* deploy very high-level writing skills to generate well-formed text. If Editing could be used as an at-scale pre-training task, the learned model would presumably have excellent language comprehension and also generate excellent text. However, it's not obvious how to generate at-scale training data for this "Editing" task, say, for English.


a. Natural Code

```
Scanner sc = new Scanner(...);
while (sc.hasNext()) {
    String ln = sc.next();
    ...
}
...
```

b. Un-natural code

```
Scanner sc = new Scanner(...);
for(; sc.hasNext() ; ) {
    String ln = sc.next();
    ...
}
...
```

**Figure 1: Example of a natural code fragment written by developers and its 'un-naturally' transformed counterpart. If the `initialization` and `update` part of the `for` loop were to left empty, developers would write the `while` loop.**

But our concern here is code, not natural language. We start with the argument that, because of the bimodal, dual-channel nature of code [12], it is indeed possible to generate at-scale training data for the Editing task (a.k.a. refactoring in Software Engineering terminology). Code has a formal channel, with well-defined semantics; because of this, it's possible to transform code into endless forms, all *meaning-equivalent.* Essentially, we can deploy a set of meaning preserving transformations to *rewrite* existing code from widely-used GitHub projects (which presumably have good-quality code that has passed human code review). These rewrites, (*e.g.,* Figure 1), preserve meaning but will make the code into an artificial, often unnatural form[1].

[1]Studies, with human-subjects [13, 14] suggest that humans find such rewritten but semantically identical forms harder to read and understand.

Nevertheless, we now have a matched pair of two semantically equivalent forms of code: a "de-naturalized" form and the original "natural" form. Furthermore, we can produce these pairs at-scale, and then pre-train on a code "Naturalization" task. By analogy with human Editors as described above, such pre-training forces the model to learn two hard things: 1) capture the meaning of the input code, and 2) generate an output that more closely resembles human-written code. We hypothesize that the resulting model will both learn better meaning representations, *and also* generate better code.

To this end, we pre-trained our NatGen model, using "Code Naturalizing" task. NatGen is based on a transformer-based sequence-to-sequence model, and learns to "naturalize" artificially generated "de-naturalized" code back into the form originally written by developers. We *emphasize* that NatGen learns to generate the whole code; this learned skill transfers to downstream fine-tuning tasks that require code generation. We show that our pre-training objective helps model generate more natural code (complete code, with high syntactic and semantic similarity with the original human-written code). With proper fine-tuning, NatGen achieves state-of-the-art performance in various downstream fine-tuning tasks, including code generation, code translation, bug fix, that demand code generation. We also show that NatGen is specially effective when labelled data is scarce.

We summarize our main contributions.

(1) We introduce the idea of "Code naturalization" as a pre-training task.
(2) Using code from Github, and custom tooling, we have generated and released a large dataset for pre-training models on the Naturalization task.
(3) We have built and released a large Sequence-to-Sequence model pre-trained on Naturalization.
(4) We show that (when appropriately fine-tuned) NatGen outperforms SOTA on several settings.

We publish our source code and data download script for pre-training NatGen anonymously in https://github.com/saikat107/NatGen. We also share the pre-trained model in https://bit.ly/natgen-pre-trained-models and all the finetuned model in https://bit.ly/natgen-fine-tuned-models.

## 2 BACKGROUND & PROBLEM FORMULATION

This section presents the relevant technical background that leads to this work and an overview of the main research questions.

### 2.1 The Dual Channels of Code

Humans can read and write both natural languages and code. However, unlike natural language, source code involves *two* channels of information: formal & natural [14]. The formal channel, unique to code, affords precise, formal semantics; interpreters, compilers, etc., use this channel. On the other hand, the natural channel (perhaps more probabilistic and noisy) relies on variable names, comments, etc., and is commonly used by humans for code comprehension and communication [13, 14]. The formal channel's precision enables semantic preserving code transformation, which supports static analysis, optimization, obfuscation, *etc.* For instance, major refactoring of a source code may drastically change the syntactic structure while preserving the semantics [20, 23]. However, not all the semantically equivalent code is "natural" [32]—the usual way developers write code and thus, amenable to statistical models [32]. In fact, deviation from such "naturalness" may lead to unintended bugs [54], and increase difficulty of human comprehension [13, 14].

We leverage the natural/formal duality for our pre-training objective in this work. We keep the formal channel constant (not changing the meaning) for a given code and modify the syntax by creating "unnatural" code. Then we train the model to take the "unnatural" code as input and do what a human Editor does with natural language text: understand the "unnatural" code and generate more natural code that a developer would write. Thus, the model simultaneously learns to both comprehend code, *and* generate "natural" code.

## 2.2 "Naturalizing" *vs.* De-noising

Naturalizing pre-training essentially follows in the tradition of *denoising pre-training*, although, arguably, the former is more subtle and challenging. Denoising pre-training [3, 38, 39] is a well-established pre-training strategy for encoder-decoder models: the encoder is presented with a noised-up input, and the decoder is asked to generate the original, noise-free input. By training the model to identify & remove "noise" in a noisy output, (in theory) one teaches it to reason about and correctly generate text. Exactly what a model learns largely depends on the noise types. For instance, PLBART [3] uses syntactic noise[2](*i.e.,* token masking, token deletion, etc.). Thus, denoising pre-training enables PLBART to learn both about the syntax of input source code, *and* learn to generate syntactically correct code. Naturalizing pre-training, on the other hand, begins with syntactically correct but artificially-created *unnatural* source code and forces the model to generate correct *semantically equivalent natural* code that is just what a human originally wrote. Such pre-training requires more subtle changes to the code. We hypothesize that this provides a more demanding pre-training setting, which will lead to better on-task code generation performance.

## 2.3 Research Questions

Our hypothesis is that our *naturalizing* task (see Section 3.1) endows our pre-trained model with the ability to generate syntactically and semantically correct, *and* natural code. This leads to several RQs.

**RQ1. Does "Naturalization" help to improve code generation?**

In contrast to existing de-noising techniques [3] that help the model learn lexical & syntactic structure, the naturalizing task, which is arguably more demanding than de-noising, forces NatGen generating better code with higher syntactic and semantic correctness.

The pre-training data we use (in NatGen) challenges the model to naturalize code that was "de-naturalized" in several ways, such as dead-code inserted, variable renamed, etc. We investigate the relative performance under different naturalization challenges.

[2]Noise that breaks the syntax structure of code

**RQ2. How do different components in NatGen contribute to code generation?**

We evaluate the performance under different challenges on a held-out validation dataset. This dataset is sampled with the same distribution of de-naturalizing transforms as the training dataset ($\mathcal{D}_t$); on this set, the model to reconstruct the original code. Our exploratory investigation reveals that Variable Renaming is the hardest transformation to undo: the model reconstructs original code with only 40% accuracy. Dead Code, on the other hand, is the easiest with 99% accuracy.

We further investigate NatGen's performance for downstream source code generation tasks.

**RQ3. How effective is NatGen when fine-tuned for different generative tasks in source code?**

We fine-tune the pre-trained NatGen on task-specific training dataset for a certain time budget and evaluate the fine-tuned model on the benchmark testing dataset for corresponding task. These tasks include source code (java) generation from text, code translation (from Java to C# and C# to Java), and Bug fixing. After fine-tuning, NatGen achieves the state-of-the-art performance in all these tasks. In addition, we also discover that, code generated by NatGen are syntactically and semantically more closer to the expected code.

We observe that training a model for a complex task requires sufficient labeled data. However, for most software engineering tasks, finding labeled data is a significant challenge [4]. We investigate potential scenario where size of the training data is extremely small.

**RQ4. How well does NatGen's pre-training help in tasks where labelled data is scarce?**

We simulate training data scarcity in two different ways – *Zero-shot learning*, and *Few-shot learning*. For "Zero-shot" learning, we evaluate the pre-trained NatGen in different tasks *without* any task specific fine-tuning. For "few-shot" setting, we simulate training data scarcity by sub-sampling the benchmark training datasets. We fine-tune the pre-trained NatGen on these limited training examples and measure the performance. We observe that NatGen is very efficient in low-data training. Since NatGen learns to generate syntactically and semantically correct code as part of pre-training, it faces less burden while learning in low-data training.

## 3 METHODOLOGY

Our approach comprises three steps: (i) "De-Naturalize" source code to accumulate pre-training data for NatGen (§3.1); (ii) pre-train NatGen using this data for naturalization task (§3.2); (iii) Fine-tune pre-trained NatGen with task specific dataset (§3.3).

## 3.1 De-naturalizing Source Code

For the first step above, we use six rules to transform a natural code into its unnatural counterpart. These transformations are semantic-preserving but rewrite an original, natural, (human-) written code to an artificial form. Given a natural code element, we deploy an

```
1   int search(int[] arr, int key, int low, int high){
2       while (low <= high) {
3           int mid = low  + ((high - low) / 2);
4           if(arr[mid] == key)  { return mid; }
5           else { high = mid + 1; }
6       }
7       return -1;
8   }
```

**(a) Original Code**

```
1   int search(int[] arr, int key, int low, int high){
2       for ( ; low <= high ; ) {
3           int mid = low  + ((high - low) / 2);
4           if(arr[mid] == key) { return mid; }
5           else { high = mid + 1; }
6       }
7       return -1;
8   }
```

**(b) Loop Transformation**

```
1   int search(int[] arr, int key, int low, int high){
2       while (low <= high) {
3           int mid = low  + ((high - low) / 2);
4           while ( i < i ) {
5               high = mid + 1;
6           }
7           // ... Rest of the Code
8       }
9       return -1;
10  }
```

**(c) DeadCode Insertion**

```
1   int search(int[] arr, int key, int low, int high){
2       while ( high >= low ) {
3           int mid = low  + ((high - low) / 2);
4           if( arr[mid] != key )  {
5               high = mid + 1;
6           }
7           else { return mid; }
8       }
9       return -1;
10  }
```

**(d) Block and Operand Swap**

```
1   int search(int[] arr, int key, int low, int high){
2       while (low <= high) {
3           int mid = low  + ((high - low) / 2);
4           if(arr[mid] == key) { return mid; }
5           else {
6               high = mid++ ;
7           }
8       }
9       return -1;
10  }
```

**(e) Inserting confusing code element**

```
1   int search(int[] var_1 , int key, int low, int var_2 ){
2       while (low <= var_2 ) {
3           int mid = low  + (( var_2  - low) / 2);
4           if( var_1 [mid] == key) { return mid; }
5           else { var_2  = mid + 1; }
6       }
7       return -1;
8   }
```

**(f) Variable Renaming**

**Figure 2: Semantic preserving transformation used to prepare the pre-training data for NatGen.**

appropriate transformation, based on its AST structure and rewrite the code to "de-naturalize" it.

*3.1.1 Designing Transformation Rules.* We use six classes of de-naturalizing transformations. These transformations are motivated by prior work on functional reasoning about source code [20, 25, 26] and semantic bug-seeding [48]. Figure 2 show the details.

**Loop Transformation (Figure 2b).** This rule modifies `for` loops into equivalent `while` loop and vice-versa. We rewrite a `while` loop of the form `while ( condition ){ loop-body }` into a `for` loop as `for ( ; condition ; ){ loop-body }`. Likewise, to transform a `for` loop into a `while` loop, we move the initializer of the `for` (if any) before the loop, and the update *expression* (if any) of the `for` loop as the last *statement* in the loop. We also add this update statement before any loop breaking statement (*i.e.,* `break`, `continue`). For example, we transform "`for(int i = 0; i < 10; i++){ if(i){ foo(); continue;} bar(); }`" as "`int i = 0; while(i < 10){ if(i){ foo(); i++; continue;} bar(); i++;}`".

**Dead Code Injection (Figure 2c).** We inject blocks of dead code at random positions in the original code. By "dead code" we mean code that appears in the source but is never executed. In Figure 2c, we inject the code block `high = mid + 1;` at line 4 of the original code (Figure 2a). To add challenge to the model, we transplant these inserted statements from the same input code. To ensure the "death" of inserted code, we put the inserted statements in a block headed by either a loop or a branch, guarded by a unsatisfiable condition so that the code inside the block will never execute. In Figure 2c, the condition `i < i` is always **`false`** ; and the code in line 5 is quite dead.

**Block Swap (Figure 2d).** Here we swap the "then" block of a chosen `if` statement with the corresponding `else` block. To preserve semantic equivalence, we negate the original branching condition. For instance, Figure 2d replaces the `if` block (line 4 in Figure 2a) with the `else` block (line 5 in Figure 2a). We negate the original condition (`arr[mid] == key` ) as (`arr[mid] != key` ).

**Operand Swap (Figure 2d).** Here, we swap the operands of binary logical operations. For instance, we change the expression `low <= high` with `high >= low` in line 2 in Figure 2d. When swapping the operands of a logical operator, we change the operator to make sure the modified expression is the logical equivalent to the one before modification. In case of asymmetric inequality operators

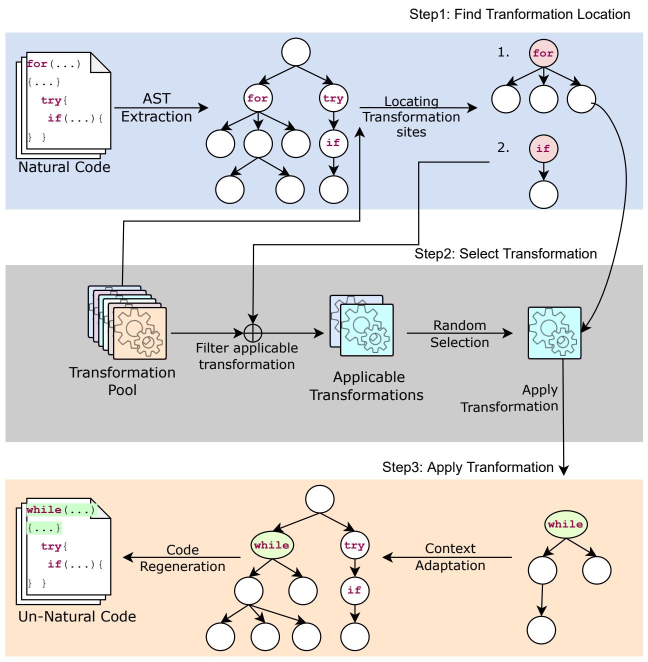

Step1: Find Tranformation Location
Natural Code
AST Extraction
Locating Transformation sites
Step2: Select Transformation
Transformation Pool
Filter applicable transformation
Applicable Transformations
Random Selection
Apply Transformation
Step3: Apply Tranformation
Un-Natural Code
Code Regeneration
Context Adaptation

**Figure 3: "De-Naturalization" workflow in NatGen.**

(>, <, >=, <=), we change the direction – keep as is for symmetric operators (*i.e.,* ==, ! =).

**Confusing Code Insertion (Figure 2e).** We introduce confusing code patterns in the code as outlined by Gopstein et al. [25, 26]. In particular, we introduce two forms of confusing code. First, we modify the of the form `{i = j; j += 1;}` to `i = j++;` . Second, we introduce ternary operator as applicable. For example, we transform the code `if (x != 0){y = p;} else {y = q;}` to `y = (x != 0)? p : q;` .

**Variable Renaming (Figure 2f).** We rename some variables to `VAR_i`. While renaming a variable, we analyze the dataflow of that variable and rename all occurrences of that variable in the entire code. From all the variables used in the code, we change just a certain percentage. For instance, in Figure 2f, we renamed variable `arr` to `var_1` , and variable `high` to `var_2` , leaving all other variables unchanged. Note that, unlike other transformations, variable renaming does not create AST of Dataflow graph difference. However, this challenging task [9] forces the model to learn to generate natural variable names. This resembles the de-obfuscation pre-training task of [58].

#### 3.1.2 *Applying Transformation.*

Assume a set of transformation rules $\Phi = \{\phi_1, \phi_2, \phi_3, ...\}$. Given original code $c_i$, $\phi_j(c_i)$ transforms the code, changing the structure while preserving semantics. Figure 3 shows how to apply such transformation to $c_i$. It works in three steps:

- *Find Transformation Location.* Given a piece of source code ($c_i$), we first use tree-sitter[3] to parse out the AST ($T_{c_i}$). From the AST, we extract potential locations for de-naturalization. These locations are nodes ($n_k$) in $T_{c_i}$. While choosing location $n_k$ from $T_{c_i}$, we consult $\Phi$ – we extract the nodes where at least one of $\phi_j \in \Phi$ is applicable.
- *Select Transformation Rule.* Once we have a set of such nodes, we filter out the transformation rules that cannot be applied to any node of in $T_{c_i}$. After such a filtration, we have a set of transformations $\Phi_a \subseteq \Phi$. At this stage, we randomly select one transformation pattern $\phi_j \in \Phi_a$ to apply at an application location (AST node) $n_k$.
- *Apply Transformation.* We apply $\phi_j$ to $n_k$ to get the transformed node $n'_k$. We then structurally match $n'_k$ with the original AST $T_{c_i}$, specifically $n_k$. We adapt the context of $n_k$ to the transformed node's ($n'_k$) context. In that way, we get the transformed AST ($T'_{c_i}$), which we then translate to get the transformed code $c'_i$.

We designed the transformation function $\phi_j$ and subsequent context adaptation in such a way that preserves the meaning or functionality of the original code. We use AST analysis and (approximated) data flow analysis on code AST.

### 3.2 Pre-training

Once we have a pool of "unnatural" code using the transformation in Section 3.1 (*i.e.,* transform code $c_i$ as 'un-natural' code $\phi_j(c_i)$), we use a neural sequence-to-sequence translation model ($\mathcal{M}$) to reconstruct $c_i$ from $\phi(c_i)$, *i.e.,* we want $\mathcal{M}(\phi_j(c_i))$ to approximate $c_i$ . In particular, given a training dataset $\mathcal{D}_t = \{c_1, c_2, ...\}$ consisting of developers written code, set of "de-naturalizing" transformations $\Phi = \{\phi_1, \phi_2, \phi_3, ...\}$, we optimize the following function to learn $\mathcal{M}$'s optimal parameter $\Theta$.

$$\Theta = \arg\min_{\theta} \sum_{c_i \in \mathcal{D}_t} CrossEntropy\left(\mathcal{M}\left(\phi_j\left(c_i\right)\right), c_i\right) \tag{1}$$

### 3.3 Fine-Tuning

The objective of our pre-training is to learn to both comprehend and generate general-purpose source code. However, different tasks related to source code generation (*e.g.,* text to code generation, code to code translation, bug fixing) call for task-specific training of the pre-trained model. This training phase on a pre-trained model is known as fine-tuning [? ]. We consider the fine-tuning in NatGen as a translation task and follow the standard transformer based-machine translation procedure [63]. First, the encoder generates the encoded representation $R(X)$ given the input $X = [x_1, x_2, ..., x_n]$. The decoder then sequentially generates the output $Y = [y_1, y_2, ..., y_m]$. While encoding an input token $x_k$, the encoder learns the attention matrix *w.r.t.* every token in the input, including $x_k$. Such attention matrix is known as *self-attention.* While generating an output token $y_m$, the decoder learns the attention matrix with all previously generated tokens $[y_1, y_2, ..., y_{m-1}]$ through *self-attention* and the encoder generated representation $R(X)$ through *cross-attention.* We refer to Vaswani et al. [63] for more detail about transformer-based translation.

## 4 EXPERIMENTAL SETUP

This section details the experimental design of NatGen.

*Pre-training data.* Following prior works [22, 28, 64], we primarily use CodeSearchNet [34] dataset for the pre-training purpose. CodeSerachNet is a publicly available dataset with six languages:

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

Java, Python, Go, JavaScript, Ruby, and PHP. In addition to CodeSearchNet, CodeT5 uses additional data for C and C#. We also use 1M functions each for C and C#. For these two additional languages, we collected 5000 active projects from GitHub and randomly selected 1M functions considering the maximum sequence length of the model.

**Table 1: Statistics of fine-tuning datasets.**

| | Task | Dataset | Train# | Dev# | Test# |
|---|---|---|---|---|---|
| Text → Code | Generation [36] | Concode | 100000 | 2000 | 2000 |
| Code → Code | Translation [42] | CodeXGLUE | 10300 | 500 | 1000 |
| Text+code → Code | BugFix [60] | Small | 46628 | 5828 | 5831 |
| | | Medium | 53324 | 6542 | 6538 |

*Fine-tuning data.* We evaluate different variations of three benchmark tasks related to source code generation. The first task is *Text to Code generation*, where the input is an NL description of a Java method, and the output is the code. The second task is *Code Translation* between *Java to C#* and *C# to Java*. For this task, we evaluate Java-C# parallel dataset proposed by Lu et al. [42]. The third and final task is *Bug Fix*, where the given a buggy code and a summary of the fix model generates the fixed code. For this task, we used the two different versions of the dataset (small, with less than 50 tokens and medium with up to 100 tokens) proposed by Tufano et al. [60]. Note that, similar to MODIT [16], we evaluate on *concrete* version of the refinement datasets.Table 1 shows the datasets and their statistics. For Text to Code Generation and Code Translation, we reuse the same split from CodeXGLUE [42], and for Bug Fix, we reuse the same split as MODIT.

*Pre-training Model Configurations.* We use 12 layer transformers with 12 attention heads on both encoder and decoder following the CodeT5 [64] architecture. As discussed in Section 3, we use de-naturalization generative objectives for pre-training. We initialize our model with CodeT5's [64] released parameters. In particular, we initialize NatGen with "CodeT5-base" model. We pre-train NatGen on 2 Nvidia GeForce RTX 3090 GPUs for 25K steps, maintaining the effective batch size at 1080 with learning rate 5e-5. We train NatGen for approximately 168 hours.

*Evaluation Metric.* Throughout the experiments in this work, we evaluate accuracies w.r.t. exact match (EM), Syntax match (SM), Dataflow match (DM), and CodeBLEU (CB) [56]. SM is the proportion of matching subtrees between output code and tadget code's ASTs *w.r.t.* number of all possible subtrees in the target code's AST. DM is the percentage of matched (with target code) anonymized dataflow edge (def-use edge) of output code *w.r.t.* all dataflow edges in the target code. Note that, both the SM and DM are components of CB. We explicitly evaluate these for understanding the syntactic and semantic correctness of generated code. We reuse Microsoft CodeXGLUE tool [44] to compute SM, DM, and CB.

*Baselines.* While comparing the evaluation results for different tasks, we compare with large scale pre-trained models, including GPT-2 [51], CodeGPT [42], PLBART [3], SPT-Code [46] and CodeT5 [64]. Most of our fine-tuning evaluation is on benchmarked dataset; thus, we report the available results from CodeXGLUE leaderboard [45]. There are some task specific baselines, which we discuss while describing corresponding task.

## 5 EMPIRICAL RESULTS

We evaluate NatGen on (i) pre-training and (ii) three fine-tuning tasks. We also check NatGen's effectiveness in zero-shot and few-shot settings.

### 5.1 NatGen's Effectiveness on Pre-training

RQ1. Does "Naturalization" help to improve code generation?

*Motivation.* We investigate whether pre-training on naturalizing task helps the model generate correct and natural code (code that is syntactically and semantically similar to the original code).

*Experimental Setup.* We compare three large scale pre-trained models: (i) CodeT5 [64], (ii) PLBART [3], and (iii) NatGen. Note that, since PLBART is only pre-trained on Java and Python, we compare PLBART only for those languages, with the corresponding results of other models. We ask each of these models to reconstruct developers' written code from its de-naturalized (but semantically identical, see §3.1 & §3.1.1) variants. We use the held-out validation data from our training procedure for this evaluation. We evaluate the models for generating the Exact Match (EM), Syntax Match (SM) and Dataflow Match (DM).

**Table 2: Evaluation of NatGen for code generation task. CS is the percentage of examples where output is directly copied from source, and ED is the median edit distance between input code and output code.**

| Eval Data | Model | EM | SM | DM | CB | CS | ED |
|---|---|---|---|---|---|---|---|
| Full | CodeT5 | 0 | 13.93 | 19.86 | 9.74 | 0% | 60 |
| | NatGen | **70.39** | **98.78** | **97.69** | **97.31** | 0.01% | 8 |
| Java & Py | CodeT5 | 0 | 13.83 | 23.67 | 10.87 | 0% | 65 |
| | PLBART | 0 | 73.17 | 75.95 | 74.56 | 7.05% | 3 |
| | NatGen | 64.13 | 98.16 | 96.85 | 96.82 | 0.01% | 10 |

*Results.* Table 2 shows the evaluation results.

• *Syntax Match.* We find that the code generated by PLBART and NatGen are mostly syntactically correct. However, CodeT5's does not always generate syntactically valid code, suggesting an advantage for naturalization pre-training. For instance, Figure 4 shows code generated by different models from the given input. As we can see, CodeT5 generates a syntactically erroneous fragment. In contrast, PLBART made a minor edit on the input code, just removing the `protected` keyword. Both PLBART and NatGen are pre-trained to generate complete code rather than fragments (which is the case of CodeT5 [52]); thus, the former two generally do better at generating syntactically correct code.

• *Semantic Match.* NatGen is effective at recovering developers' written code from its de-naturalized semantic variants—around 70% of the generated code (CodeBlue = 97%) *exactly matches* the original code. PLBART, which deploys syntactic denoising, is at the second position in terms of CodeBlue.

1. Input

```
protected SDV iam(SDV in,...){
    if(i < i){
        return new IAM(...);
    }
    return new IAM(...);
}
```

2. PLBART output

```
SDV iam(SDV in, ...){
    if(i < i){
        return new IAM(...);
    }
    return new IAM(...);
}
```

3. NatGen output

```
protected SDV iam(SDV in,...){
    return new IAM(...);
}
```

4. CodeT5 output

```
if (in) {
      return
}  }
```

**Figure 4: Example of input generated code by different pre-trained models (slightly simplified).**

NatGen also dominates the other two models in generating syntactically (SM) & semantically (DM) valid code. While PLBART appears to generate syntactically correct code, it mostly copies code from the input—median edit distance from PLBART’s input and the generated code is 3 (see Table 2). In fact, in 7.05% of cases, PLBART just copies the input! By contrast, NatGen learns to generate *variants* of the input code, with only 0.01% direct copy and a median edit distance of 10. Since PLBART is trained to remove syntax errors from the input, we conjecture that it does not inherently learn the semantic variation of the code. By contrast, we expose NatGen to semantic code variations, forcing it to learn to generate code that is both more natural <u>*and*</u> semantically equivalent.

• *Closer look into CodeT5.* Unlike NatGen and PLBART, CodeT5 is not explicitly trained to generate complete code. During pre-training, CodeT5 learned to “unmask” masked token sequences. Thus, to better measure CodeT5’s generation capacity, we conduct another experiment where we replaced all occurrences of some of the variable names in code with a special `MASK1`, `MASK2` tokens and asked CodeT5 to generate. This is one of the objectives (masked identifiers prediction) CodeT5 is pre-trained to optimize. We take the CodeT5’s output and identify all potential identifiers [4]. Surprisingly, in *only* 0.27% of the cases, could CodeT5 generate *all* the variables, and in 0.61% of cases *half* of the masked variables., while NatGen successfully translates 40.45% of those examples back to its original code, including correctly predicting the replaced variable names. In addition, CodeT5’s generated token sequence contained a lot of other tokens than the variable names (Figure 4.4, for example).

**Result 1:** *Naturalization enables NatGen to reason about code semantics and thus help generate more natural code variants than existing pre-training models and pre-training objectives.*

We also did an ablation study evaluating the effect of NatGen’s different components on the results.

RQ2. How do different components in NatGen contribute to code generation?

*Motivation.* In this RQ, we study how different transformation rules (see §3.1)contribute to learn generating natural code from different semantic variants . We also evaluate how well NatGen learns that in different programming languages over training time.

[4]we use regex `"[A-Za-z_]+[A-Za-z0-9_]*"` to find identifiers.

*Experimental Setup.* While pre-training, we checkpoint the NatGen model every 1k training steps, for a full run of 25k steps. At each checkpoint, we evaluate the naturalization task performance. Before training, we held out 0.1% of the total data as validation data. Note that, since our goal in this experiment is to understand NatGen’s pre-training better, we “de-naturalized" the validation data using the same training data distribution. This setting gives us a controlled environment for experimentation.

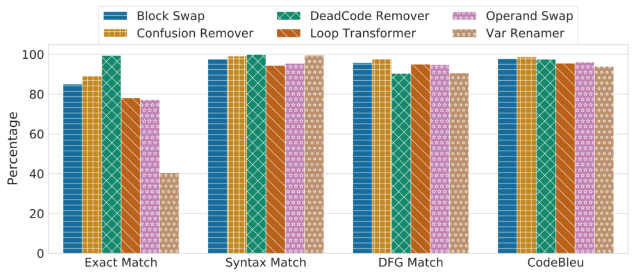


**Figure 5: Performance of NatGen pre-trained model under different code transformations.**

*Results.* Figure 5 shows NatGen’s performance under different types of semantic variants. Results show that NatGen has most trouble recreating the original code (just 40% Exact Match) with the variable renaming task. Variable renaming is challenging even for human developers [6]—different developers may propose different names for the same object. Nevertheless, on this task, NatGen achieves good syntax and dataflow match (99% and 92% respectively), indicating that NatGen preserves syntax or semantics in most cases while generating code with renamed variables.

On the other hand, NatGen can eliminate Dead Code with 99% accuracy. This result may be an artifact of our specific implementation of this transformation. Our dead-code insertion rule is simple, and formulaic; so the NatGen quickly learns to identify and remove such dead code. A more complex pattern of dead code may challenge the model more, and help make it more robust; we leave this for future work. For naturalizing other transformations, NatGen achieves more than 80% exact match accuracy for Block swap and Confusion removing, and more than 75% exact match accuracy for the rest. In all cases, syntax match, dataflow match, and CodeBLEU are well above 90%.

Figure 6 shows how validation performance improves for different languages, with more training steps. Across all the languages the performance rapidly increases over the first few thousand training steps. In fact, at the beginning of (step 0) of NatGen’s pre-training, the overall exact match is 0, syntax match is 13.93%, dataflow match is 19.86% and CodeBLEU is 9.74% (see Table 2 for details[5]). However, after just 1000 steps of training, the exact match rises to 61%, syntax match to 97%, dataflow match to 94%, and CodeBLEU to 95%. These metrics continue improving as training progresses. These results confirm that across all the languages NatGen gradually learns to generate more natural code from semantic variants.

[5]NatGen’s pre-training start from CodeT5-base. Thus, CodeT5-base is NatGen’s checkpoint at step 0.

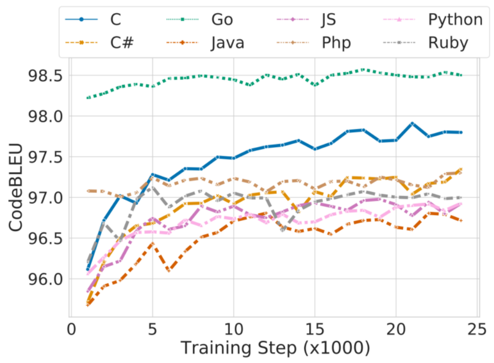


**Figure 6: Progression of CodeBLEU of different language in Validation dataset over number pre-training steps.**

**Result 2:** *pre-training performance depends on the types of semantic variants—while variable renaming seems the most difficult (∼40% accuracy), dead-code elimination appears to be an easier task (∼99% accuracy) to learn.*

## 5.2 NatGen's Effectiveness on Fine-Tuning Tasks

This section evaluates NatGen's performance on three benchmark source code generative tasks.

RQ3. How effective is NatGen when fine-tuned for different generative tasks in source code?

**Table 3: Results of Text to Code Generation. '-' implies that those results are not reported by corresponding approaches. $\mathcal{M}_{last}$ is the model after completing the fintuning, and $\mathcal{M}_{best}$ is the intermediate model with best validation performance.**

| Approach | | EM | SM | DM | CB |
|---|---|---|---|---|---|
| Seq2Seq | | 3.05 | - | - | 26.39 |
| Guo et al. [29] | | 10.05 | - | - | 29.46 |
| Iyer et al. [36] | | 12.20 | - | - | - |
| GPT-2 | | 17.30 | - | - | 29.69 |
| CodeGPT | | 20.10 | - | - | 35.98 |
| PLBART | | 18.75 | - | - | 38.52 |
| CodeT5-base (reported) | | **22.30** | - | - | 43.20 |
| CodeT5* | $\mathcal{M}_{last}$ | 21.85 | 44.34 | 44.52 | 41.75 |
| | $\mathcal{M}_{best}$ | 21.55 | 41.08 | 43.71 | 38.30 |
| NatGen | $\mathcal{M}_{last}$ | 22.25 | **45.59** | **46.87** | **43.73** |
| | $\mathcal{M}_{best}$ | **22.30** | 44.38 | 45.64 | 42.44 |

* Our reproduced result using CodeT5's publicly available pre-trained model.

*Baselines.* In addition to the baselines discussed in Section 4, for the *Text to Java Code generation* task, we compare with a group of baselines with no pre-training involved. These baselines include LSTM based Sequence to sequence models, Guo et al. [29]'s, and Iyer et al. [36]'s proposed techniques. We also report our reproduced version of CodeT5 results in different tasks, slightly different from what they reported. For both the *Bug Fix* task, we compare with the reported results of MODIT [16] and our reproduced CodeT5 result.

*Results.*

**Text to Code Generation.** Table 3 shows evaluation results for text to code generation. We trained for 30 epochs. We stopped the training is the validation performance does not increase for more than three(3) consecutive epochs. For both CodeT5 and NatGen, we report the performance of final model after the fine-tuning terminated ($\mathcal{M}_{last}$) and the performance of the model with best validation perfomance ($\mathcal{M}_{best}$). Interestingly, for both CodeT5 and NatGen, the $\mathcal{M}_{last}$ model performs better than the corresponding $\mathcal{M}_{best}$ model. The result shows that NatGen's generated code are more syntactically and semantically closer to the target code. The $\mathcal{M}_{last}$ model of NatGen outperforms CodeT5's $\mathcal{M}_{last}$ model by 2.8% in SM, 5.28% in DM and 4.74% in CB. We conjecture that NatGen's pre-training with "naturalization" help generate more natural code.

**Table 4: Code Translation results. '-' implies that those results are not reported by corresponding approaches.**

| Approach | Java→C# | | | | C#→Java | | | |
|---|---|---|---|---|---|---|---|---|
| | EM | SM | DM | CB | EM | SM | DM | CB |
| PBSTM | 12.5 | - | - | 42.7 | 16.1 | - | - | 43.5 |
| CodeBERT | 59.0 | - | - | 85.1 | 58.8 | - | - | 79.4 |
| SPT-Code | 64.1 | - | - | - | 60.2 | - | - | - |
| PLBART | 64.6 | - | - | 87.9 | 65.0 | - | - | **85.3** |
| CodeT5 (reported) | 65.9 | - | - | - | 66.9 | - | - | - |
| CodeT5* | 65.9 | 90.4 | 91.9 | 87.8 | 66.0 | 90.4 | 88.9 | 84.4 |
| NatGen | **66.2** | **91.0** | **92.0** | **88.1** | **67.3** | **91.0** | **89.8** | 85.2 |

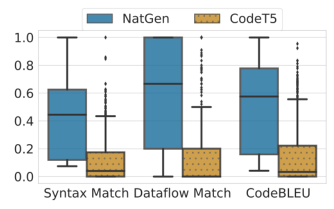
* Our reproduced result using CodeT5's publicly available pre-trained model.

**Code Translation.** Table 4 shows the results of NatGen and different baselines for Code Translation. For Java to C# translation, NatGen achieves exact match accuracy of 66.2% while CodeT5's accuracy is 65.9%. In C# to Java translation, NatGen achieves 67.3% exact match accuracy, which CodeT5 achieves 66.0%. In addition, the syntactic match (SM), Dataflow match, and CodeBLEU are also higher than that of CodeT5.

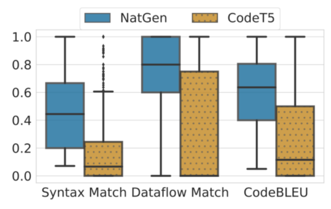
**Table 5: Result of Bug fix (Top 1 fix accuracy).**

| Approach | $\text{BugFix}_{small}$ | | $\text{BugFix}_{medium}$ | |
|---|---|---|---|---|
| | Unimodal | Multimodal | Unimodal | Multimodal |
| MODIT | 20.35 | 21.57 | 8.35 | 13.18 |
| CodeT5 | 21.79 | 22.97 | 12.59 | **14.94** |
| NatGen | **22.26** | **23.43** | **13.32** | 14.93 |

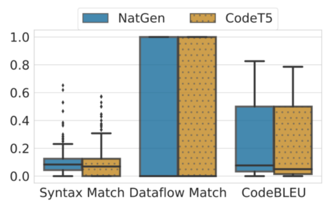
**Bug Fix.** Similar to MODIT, we evaluate the top-1 accuracy of the generated fixed code. We also evaluate uni-modal settings, where the fix description is unavailable, and multi-modal settings, where we have access to the fix description. Table 5 shows the results of Bug Fix. For the $\text{BugFix}_{small}$ dataset, NatGen outperforms both CodeT5 and MODIT in both unimodal and multi-modal settings.



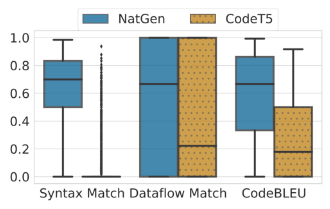
**(a) Java to C# Translation**



**(b) C# to Java Translation**

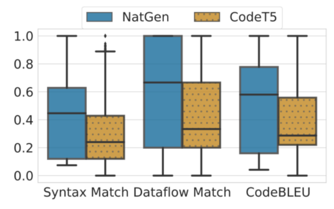


**(c) Text to Code Generation**

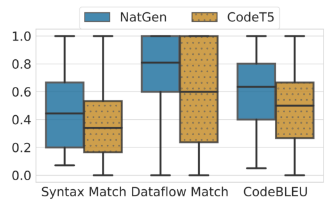


**(d) Bug Fix (small, multimodal)**

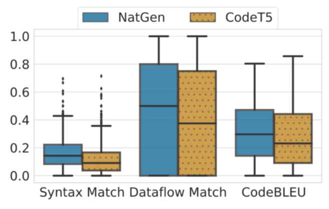
**Figure 7: Zero-shot transfer learning capability of NatGen in for different tasks.**



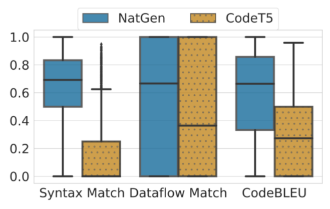
**(a) Java to C# Translation**



**(b) C# to Java Translation**

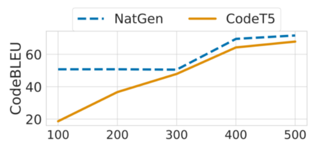


**(c) Text to Code Generation**

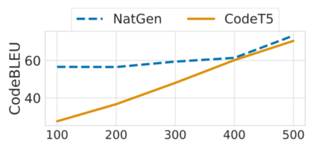


**(d) Bug Fix (small, multimodal).**

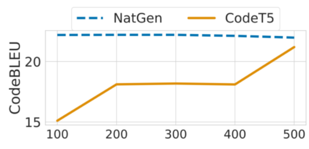
**Figure 8: Few shot Learning evaluation of NatGen. In each case, the pre-trained model is fine-tuned on 200 training examples for 10 epoch and the result is on the full test set.**



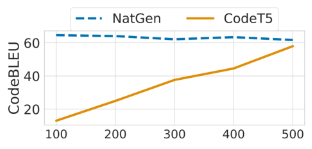
**(a) Java to C# Translation.**



**(b) C# to Java Translation.**



**(c) Text to Code Generation.**



**(d) Bug Fix (small, multimodal).**

**Figure 9: NatGen’s results on different tasks with Few shot settings. X-axis shows number of training examples.**

For For the BugFix$_{medium}$ dataset, NatGen performs better than CodeT5 and MODIT in unimodal setting and slightly worse than CodeT5 in the multi-modal setting.

**Result 3:** *NatGen performs better than most of the existing baselines. NatGen’s improvement in Syntax match and Dataflow match signifies NatGen’s ability to generate code syntactically and semantically closer to target code.*

Finally, we evaluate NatGen’s performance in the presence of data scarcity.

RQ4. How well does NatGen’s pre-training help in tasks where labelled data is scarce?

*Motivation.* Learning to generate code usually requires a large amount of annotated training data. A lot of time and effort goes into curating high-quality training data [4, 38]. Unsupervised pre-training endows machine learning models with necessary domain knowledge about the task [21]. In practice, this knowledge appears to transfer across multiple tasks. Such pre-training reduces the effort to learn each different task. We therefore study the effectiveness of NatGen’s domain knowledge about source code syntax and semantics. In particular, we *stress test* whether the knowledge NatGen learned during pre-training is useful for downstream tasks, by limiting available task-specific training data.

*Experimental Setup.* We evaluate NatGen’s over several data-limited tasks: *Text to Code generation*, *Code Translation*, and *Bug Fix*. We consider two different settings. First, we consider zero-shot [57, 67] evaluation. Here we evaluate different pre-trained models *without* any task-specific training. Naturally, we don’t see good performance in this setting. Nevertheless, this stress-test measures the code generation ability of models. Second, we try few-shot learning [53, 59, 65]. We randomly choose a few training examples for each task and fine-tune the pre-trained models on those examples, and evaluate their performance. We gradually increase the number of training examples over several few-shot settings.

*Results.* Figure 7 shows the NatGen’s and CodeT5’s zero-shot performance. Lacking task-specific training, we can see here how much transferable knowledge each model learned just during pre-training. There are large differences in all the tasks between NatGen and CodeT5 across Syntax Match and Dataflow Match. It signifies NatGen learns to generate both syntactically and semantically correct code during pre-training, which CodeT5 rarely can do. Figure 8 shows the performance of NatGen and CodeT5 when trained on 200 training examples. NatGen also has an advantage over CodeT5 here.

We note a larger performance gap in the Translation tasks (Figure 7a & 7b) and Bug Fix (Figure 7d) tasks, compared to Text to Code Generation task (Figure 7c) in both the zero-shot and the few shot (Figure 8) experiments. We conjecture that such discrepancy is the artifact of the nature of the tasks. The cross-lingual alignment between NL and Java code is the key factor in generating text to code. In contrast, both the input and output are the programming language in the translation and bug fix task. Thus, we hypothesize that NatGen leverages its shared knowledge across different programming languages learned during the pre-training.

We further stress test NatGen's with few-shot learning; we gradually increased the number of training examples and trained both CodeT5 and NatGen. Figure 9 shows the performance progress as the number of training examples increase. For all four tasks, NatGen significantly improves over CodeT5 when the number of training examples is minimal. With increasing training examples, the performance gap gradually decreases. Arguably, with enough *labeled* data and enough resources, all high-capacity models will get better at generating source code. Nevertheless, we learn two critical lessons from NatGen's better performance in zero-shot and few-shot learning. First, NatGen's better performance across all tasks suggests that that the coding knowledge it learns from the naturalization task is more generic and transferable. Second, for any pre-trained model to be effective in code generation, especially in a limited training data scenario, the pre-training should explicitly teach the model how to write code. Otherwise, we hypothesize that a big chunk of fine-tuning resources will be spent on the models' learning to write code.

**Result 4:** *NatGen is very effective in source code generative tasks when minimal training resource is available. Since NatGen explicitly learns to generate code during pre-training, it can avoid learning such during fine-tuning saving fine-tuning resource.*

## 6 LIMITATIONS & THREATS

***Bias introduced by 'de-naturalizing' transformations.*** In Section 3.1, we described our six transformations to "de-naturalize" source code. The NatGen model learns to revert one transformation at a time. In fact, we found empirically that, when given code with more than one 'de-naturalization' transformation applied, the model reverses only one of them. There is thus a threat our limited application of de-naturalization limits the ability of our NatGen. Regardless, we consider NatGen as a proof-of-concept and the first work towards teaching a model to write natural code. We leave the investigation more natural code patterns and their effect on code generation as a potential future work.

**Table 6: NatGen's performance in Code summarization**

| Approach | Go | Java | JS | Python | Php | Ruby | Overall |
|---|---|---|---|---|---|---|---|
| PLBART | 18.91 | 18.45 | 15.56 | 19.30 | 23.58 | 14.11 | 18.32 |
| CodeT5 | **19.56** | 20.31 | **16.16** | 20.01 | **26.03** | 15.24 | 19.55 |
| NatGen | 19.43 | **20.38** | 16.00 | **20.09** | 26.00 | **15.38** | 19.55 |

***Knowledge retention from CodeT5.*** As mentioned in Section 4, we start NatGen's pre-training from CodeT5-base model [64]. Starting further pre-training from an existing pre-trained checkpoint is very common in large-scale pre-training. For instance, GraphCodeBERT [28] is pre-trained based on CodeBERT [22] model, which was pre-trained based on RoBERTa [41] model. Both the Open AI-CodeX [17] and GitHub Copilot [24] models are further pre-trained in OpenAI-GPT3 [11]. Nevertheless, when we further train a pre-trained model on different tasks, it is subject to "*catastrophic forgetting*" [37] of the knowledge learned in the base model. In order to test whether NatGen is forgetting CodeT5's knowledge about natural language generation, we also evaluate NatGen for Code summarization. Here the input is source code, and the output is Natural language. After fine-tuning NatGen's overall BLEU in 19.547 while CodeT5's was 19.551, suggesting that NatGen mostly retains CodeT5's capacity to generate NL (see Table 6 for detailed results).

***Fair Comparison with CodeT5.*** We initialize NatGen with pre-trained checkpoint from CodeT5 (already pre-trained 75K steps with their objective) and train NatGen for 25K steps with 'natural-code' writing objective. A skeptic reader would want to know what happens when we pre-train CodeT5 for 25K more steps with their training objective. We argue that since the pre-training objective does not explicitly account for generating code (See section 3.2 of CodeT5's original paper), further training with the CodeT5 objective does not necessarily increase its code generation capacity. We do acknowledge CodeT5's ability to understand and reason about *input*. Since the pre-training large model is extremely expensive (§4)[6]; we leverage such knowledge by initializing NatGen from CodeT5's publicly available pre-trained model. Moreover, CodeT5 release neither their code for pre-training (only for fine-tuning), nor any earlier or later checkpoints for us to carry out further investigation.

***"Naturalization" with program-analysis.*** NatGen is a prototype of a generative pre-trained model with "Naturalization" task, trained to revert six classes of de-naturalization transformations (see Figure 2). However, perfect performance *w.r.t.* these transformation is **not** the main objective of this research. Tools to accomplish "naturalization" could surely be built using traditional refactoring methods; however, our goal is to train NatGen so that it learns to generate natural code with the help of this "Naturalization" task.

***NatGen as "Code-Refactoring" tool.*** NatGen suggests the promise of neural transformers to build meaning-preserving code-refactoring tools. However, to realize a more accurate and powerful neural re-factoring tool, more training data, with a larger variety of transformations, would be required. We leave this as future work.

## 7 RELATED WORKS

The approach of pre-training large Transformers without human labels started in NLP domain with BERT [? ], which introduces two pre-training objectives (i.e., Mask Language Modeling and Next Sentence Prediction). Later, Liu et al. show that RoBERTa [41] outperforms BERT only using Mask Language Modeling (MLM) with new training strategies and hyper-parameter tuning. MLM is

[6]CodeT5 was pre-trained on 16 NVIDIA A100s, with 40G memory each, for 12 days! One might reasonably assume it was already well-trained on the original objective

a self-supervised task that the model randomly masks or modifies a certain number of tokens and tries to recover them.

Following the success of the pre-trained model in the NLP domain, researchers applied these models to code related tasks. CodeBERT is one of the earliest that was specially trained for code and relevant natural language descriptions. It is pre-trained with two objectives (i.e., MLM and Replaced Token Detection [18]) and demonstrated pre-training's effectiveness for code. Later, an architecturally equivalent model, GraphCodeBERT, was introduced; it improved over CodeBERT on most tasks by incorporating data-flow information.

Though CodeBERT [22] & GraphCodeBERT [28], DietCodeBERT [68] do well at code understanding tasks, these models are not as good at generative tasks. Both models are encoder-only and have to start with an untrained decoder in fine-tuning for generative tasks, such as code repair, code generation, code summarization, and code translation. To address this limitation, Ahmad et al. introduced PLBART [3], pre-trained as a generative denoising autoencoder. A specific set of noises is introduced to code and relevant natural language description and used as the input to the model. The model's objective is to encode the noisy input in the encoder and generate noise-free code or text in the decoder. PLBART (builds on BART [39]) outperforms both CodeBERT [22] and GraphCodeBERT [28] on both understanding and generative tasks with a pre-trained encoder and decoder [3]. DOBF [58] uses de-obfuscation (recovering variable names) as their pre-training task; however, rather than generating code, they just generate a dictionary of recovered names.

CodeT5 [64] (based T5 [52]) is the latest denoising model. CodeT5 uses the developer-assigned identifiers in code, adding two code-specific pre-training objectives to the original T5, identifier tagging and masked identifier prediction. CodeT5 is an encoder-decoder model and excels at both understanding and generative tasks compared to other models. Similar to CodeT5, [43, 49] are also built based on T5 architecture and perform reasonably well in the different downstream tasks. NatGen has a similar architecture to CodeT5; but rather than CodeT5's pre-training objectives, we "denaturalize" code, using the formal channel of code to inject meaning-preserving transforms, and then force NatGen to recreate, the original, "natural" code. Rewriting semantically equivalent code requires semantic understanding, and that can be applied to code only because of its dual-channel nature. Our evaluation shows that rewriting semantically equivalent programs in the pre-training stage results in performance gains in at least three popular Software Engineering tasks.

## 8 CONCLUSION

We introduce the "Code-Naturalization" pre-training objective for generative models of code. As proof-of-concept we pre-trained our NatGen to write 'natural' source code from 'un-natural' counterpart. With this pre-training, NatGen learns to write code syntactically and semantically closer to developers' written code. We "denaturalize" existing developers' code, using six kinds of "semantic-preserving" transformations. We further fine-tune the NatGen on different variations of three downstream tasks that require code generation. NatGen achieves state-of-the-art performance in these downstream tasks, and NatGen's generated code are syntactically and semantically closer to the target code. Our pre-training on the 'naturalizing' task is especially effective in resource-constrained setting *i.e.,* zero-shot, and few-shot transfer learning.

## 9 DATA AVAILABILITY STATEMENT

We publicly code, and all processing scripts of NatGen's pre-training [1]. NatGen pre-trained model is also available through https://huggingface.co/saikatc/NatGen.

## ACKNOWLEDGMENTS

This work is supported in part by NSF grants CCF-2107405, SHF-1845893, SHF-1414172, SHF-2107592, IIS-2221943, and IBM faculty award. Any opinions, findings, conclusions, or recommendations expressed herein are those of the authors and do not necessarily reflect those of the US Government, NSF, or IBM. Ahmed was also supported by UC Davis College of Engineering Dean's Distinguished Fellowship.

## REFERENCES

[1] 2022. NatGen: Generative Pre-training by "Naturalizing" Source Code - Code and scripts for Pre-Training. https://doi.org/10.5281/zenodo.6977595
[2] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2020. A Transformer-based Approach for Source Code Summarization. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics (ACL).*
[3] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified Pre-training for Program Understanding and Generation. In *2021 Annual Conference of the North American Chapter of the Association for Computational Linguistics (NAACL).*
[4] 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 [cs.SE]
[5] Miltiadis Allamanis. 2019. The adverse effects of code duplication in machine learning models of code. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software.* 143–153.
[6] Miltiadis Allamanis, Earl T Barr, Christian Bird, and Charles Sutton. 2015. Suggesting accurate method and class names. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering.* 38–49.
[7] Miltiadis Allamanis, Earl T Barr, Premkumar Devanbu, and Charles Sutton. 2018. A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)* 51, 4 (2018), 1–37.
[8] Miltiadis Allamanis, Earl T Barr, Soline Ducousso, and Zheng Gao. 2020. Typilus: Neural type hints. In *Proceedings of the 41st acm sigplan conference on programming language design and implementation.* 91–105.
[9] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. 2017. Learning to represent programs with graphs. *arXiv preprint arXiv:1711.00740* (2017).
[10] Matthew Amodio, Swarat Chaudhuri, and Thomas W Reps. 2017. Neural attribute machines for program generation. *arXiv preprint arXiv:1705.09231* (2017).
[11] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. arXiv:2005.14165 [cs.CL]
[12] Casey Casalnuovo, Earl T Barr, Santanu Kumar Dash, Prem Devanbu, and Emily Morgan. 2020. A theory of dual channel constraints. In *2020 IEEE/ACM 42nd International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER).* IEEE, 25–28.
[13] Casey Casalnuovo, Kevin Lee, Hulin Wang, Prem Devanbu, and Emily Morgan. 2020. Do programmers prefer predictable expressions in code? *Cognitive science* 44, 12 (2020), e12921.
[14] Casey Casalnuovo, E Morgan, and P Devanbu. 2020. Does surprisal predict code comprehension difficulty. In *Proceedings of the 42nd Annual Meeting of the Cognitive Science Society.* Cognitive Science Society Toronto, Canada.
[15] Saikat Chakraborty, Yangruibo Ding, Miltiadis Allamanis, and Baishakhi Ray. 2020. CODIT: Code Editing with Tree-Based Neural Models. *IEEE Transactions on Software Engineering* 1 (2020), 1–1.

[16] Saikat Chakraborty and Baishakhi Ray. 2021. On Multi-Modal Learning of Editing Source Code. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 443–455. https://doi.org/10.1109/ASE51524.2021.9678559

[17] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. arXiv:2107.03374 [cs.LG]

[18] Kevin Clark, Minh-Thang Luong, Quoc V. Le, and Christopher D. Manning. 2020. ELECTRA: Pre-training Text Encoders as Discriminators Rather Than Generators. In *International Conference on Learning Representations*. https://openreview.net/pdf?id=r1xMH1BtvB

[19] ]devlin2018bert Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. [n. d.]. 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)*.

[20] Yangruibo Ding, Luca Buratti, Saurabh Pujar, Alessandro Morari, Baishakhi Ray, and Saikat Chakraborty. 2022. Towards Learning (Dis)-Similarity of Source Code from Program Contrasts. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 6300–6312.

[21] Dumitru Erhan, Aaron Courville, Yoshua Bengio, and Pascal Vincent. 2010. Why does unsupervised pre-training help deep learning?. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*. JMLR Workshop and Conference Proceedings, 201–208.

[22] 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*. 1536–1547.

[23] Martin Fowler. 2018. *Refactoring: improving the design of existing code*. Addison-Wesley Professional.

[24] GitHub. 2022. GitHub Copilot (https://copilot.github.com/).

[25] Dan Gopstein, Anne-Laure Fayard, Sven Apel, and Justin Cappos. 2020. Thinking aloud about confusing code: A qualitative investigation of program comprehension and atoms of confusion. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 605–616.

[26] Dan Gopstein, Hongwei Henry Zhou, Phyllis Frankl, and Justin Cappos. 2018. Prevalence of confusing code in software projects: Atoms of confusion in the wild. In *Proceedings of the 15th International Conference on Mining Software Repositories*. 281–291.

[27] David Gros, Hariharan Sezhiyan, Prem Devanbu, and Zhou Yu. 2020. Code to Comment "Translation": Data, Metrics, Baselining & Evaluation. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 746–757.

[28] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Jian Yin, Daxin Jiang, et al. 2021. GraphCodeBERT: Pre-training Code Representations with Data Flow. In *International Conference on Learning Representations*.

[29] Daya Guo, Duyu Tang, Nan Duan, Ming Zhou, and Jian Yin. 2019. Coupling Retrieval and Meta-Learning for Context-Dependent Semantic Parsing. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics, Florence, Italy, 855–866. https://doi.org/10.18653/v1/P19-1082

[30] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. 2017. DeepFix: Fixing Common C Language Errors by Deep Learning.. In *AAAI*. 1345–1351.

[31] Vincent J Hellendoorn and Premkumar Devanbu. 2017. Are deep neural networks the best choice for modeling source code?. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 763–773.

[32] Abram Hindle, Earl T Barr, Mark Gabel, Zhendong Su, and Premkumar Devanbu. 2016. On the naturalness of software. *Commun. ACM* 59, 5 (2016), 122–131.

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

[34] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. Codesearchnet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436* (2019). https://arxiv.org/abs/1909.09436

[35] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2016. Summarizing Source Code using a Neural Attention Model. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 2073–2083. https://doi.org/10.18653/v1/P16-1195

[36] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2018. Mapping language to code in programmatic context. *arXiv preprint arXiv:1808.09588* (2018).

[37] James Kirkpatrick, Razvan Pascanu, Neil Rabinowitz, Joel Veness, Guillaume Desjardins, Andrei A Rusu, Kieran Milan, John Quan, Tiago Ramalho, Agnieszka Grabska-Barwinska, et al. 2017. Overcoming catastrophic forgetting in neural networks. *Proceedings of the national academy of sciences* 114, 13 (2017), 3521–3526.

[38] Marie-Anne Lachaux, Baptiste Roziere, Lowik Chanussot, and Guillaume Lample. 2020. Unsupervised Translation of Programming Languages. *arXiv preprint arXiv:2006.03511* (2020).

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

[40] Shangqing Liu, Yu Chen, Xiaofei Xie, Jingkai Siow, and Yang Liu. 2020. Retrieval-augmented generation for code summarization via hybrid gnn. *arXiv preprint arXiv:2006.05405* (2020).

[41] 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. *arXiv preprint arXiv:1907.11692* (2019). https://arxiv.org/abs/1907.11692

[42] 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. *arXiv preprint arXiv:2102.04664* (2021). https://arxiv.org/abs/2102.04664

[43] Antonio Mastropaolo, Simone Scalabrino, Nathan Cooper, David Nader Palacio, Denys Poshyvanyk, Rocco Oliveto, and Gabriele Bavota. 2021. Studying the usage of text-to-text transfer transformer to support code-related tasks. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 336–347.

[44] Microsoft. 2021. CodeBLEU calculator (https://github.com/microsoft/CodeXGLUE/tree/main/Code-Code/code-to-code-trans/evaluator/CodeBLEU).

[45] Microsoft. 2022. CodeXGLUE Leaderboard (https://microsoft.github.io/CodeXGLUE/).

[46] Changan Niu, Chuanyi Li, Vincent Ng, Jidong Ge, Liguo Huang, and Bin Luo. 2022. SPT-Code: Sequence-to-Sequence Pre-Training for Learning the Representation of Source Code. *arXiv preprint arXiv:2201.01549* (2022).

[47] Md Rizwan Parvez, Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Retrieval Augmented Code Generation and Summarization. *arXiv preprint arXiv:2108.11601* (2021).

[48] Jibesh Patra and Michael Pradel. 2021. Semantic bug seeding: a learning-based approach for creating realistic bugs. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 906–918.

[49] Long Phan, Hieu Tran, Daniel Le, Hieu Nguyen, James Anibal, Alec Peltekian, and Yanfang Ye. 2021. CoTexT: Multi-task Learning with Code-Text Transformer. *arXiv preprint arXiv:2105.08645* (2021).

[50] Michael Pradel and Satish Chandra. 2021. Neural software analysis. *Commun. ACM* 65, 1 (2021), 86–96.

[51] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. Language models are unsupervised multitask learners. *OpenAI blog* 1, 8 (2019), 9.

[52] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. 2019. Exploring the limits of transfer learning with a unified text-to-text transformer. *arXiv preprint arXiv:1910.10683* (2019).

[53] Sachin Ravi and Hugo Larochelle. 2016. Optimization as a model for few-shot learning. (2016).

[54] Baishakhi Ray, Vincent Hellendoorn, Saheel Godhane, Zhaopeng Tu, Alberto Bacchelli, and Premkumar Devanbu. 2016. On the" naturalness" of buggy code. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. IEEE, 428–439.

[55] Veselin Raychev, Martin Vechev, and Eran Yahav. 2014. Code completion with statistical language models. In *Acm Sigplan Notices*, Vol. 49. ACM, 419–428.

[56] Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Ming Zhou, Ambrosio Blanco, and Shuai Ma. 2020. CodeBLEU: a Method for Automatic Evaluation of Code Synthesis. *arXiv preprint arXiv:2009.10297* (2020). https://arxiv.org/abs/2009.10297

[57] Bernardino Romera-Paredes and Philip Torr. 2015. An embarrassingly simple approach to zero-shot learning. In *International conference on machine learning*. PMLR, 2152–2161.

[58] Baptiste Roziere, Marie-Anne Lachaux, Marc Szafraniec, and Guillaume Lample. 2021. DOBF: A deobfuscation pre-training objective for programming languages.

*arXiv preprint arXiv:2102.07492* (2021).

[59] Qianru Sun, Yaoyao Liu, Tat-Seng Chua, and Bernt Schiele. 2019. Meta-transfer learning for few-shot learning. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 403–412.

[60] Michele Tufano, Jevgenija Pantiuchina, Cody Watson, Gabriele Bavota, and Denys Poshyvanyk. 2019. On Learning Meaningful Code Changes via Neural Machine Translation. *arXiv preprint arXiv:1901.09102* (2019).

[61] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. 2019. An empirical study on learning bug-fixing patches in the wild via neural machine translation. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 28, 4 (2019), 1–29.

[62] Bogdan Vasilescu, Casey Casalnuovo, and Premkumar Devanbu. 2017. Recovering clear, natural identifiers from obfuscated JS names. In *Proceedings of the 2017 11th joint meeting on foundations of software engineering*. 683–693.

[63] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *Advances in Neural Information Processing Systems 30*. 5998–6008.

[64] 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. *arXiv preprint arXiv:2109.00859* (2021).

[65] Yaqing Wang, Quanming Yao, James T Kwok, and Lionel M Ni. 2020. Generalizing from a few examples: A survey on few-shot learning. *ACM computing surveys (csur)* 53, 3 (2020), 1–34.

[66] Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. 2016. Deep learning code fragments for code clone detection. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ACM, 87–98.

[67] Yongqin Xian, Christoph H Lampert, Bernt Schiele, and Zeynep Akata. 2018. Zero-shot learning—a comprehensive evaluation of the good, the bad and the ugly. *IEEE transactions on pattern analysis and machine intelligence* 41, 9 (2018), 2251–2265.

[68] Zhaowei Zhang, Hongyu Zhang, Beijun Shen, and Xiaodong Gu. 2022. Diet Code is Healthy: Simplifying Programs for Pre-Trained Models of Code. In *Proceedings of the 2022 The ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Singapore, Singapore) *(ESEC/FSE 2022)*.