Learning Lenient Parsing & Typing via Indirect Supervision

Toufique Ahmed, Vincent Hellendoorn, Premkumar Devanbu
University of California, Davis
Davis, California, USA
tfahmed,vhellendoorn,ptdevanbu@ucdavis.edu

ABSTRACT
Both professional coders and teachers frequently deal with imperfect (fragmentary, incomplete, ill-formed) code. Such fragments are common in StackOverflow; students also frequently produce ill-formed code, for which instructors, TAs (or students themselves) must find repairs. In either case, the developer experience could be greatly improved if such code somehow could be parsed & typed; this makes them more amenable to use within IDEs and allows early detection and repair of potential errors. We introduce a lenient parser, which can parse & type fragments, even ones with simple errors. Training a machine learner to leniently parse & type imperfect code requires a large training set of pairs of imperfect code and its repair (and/or type information); such training sets are limited by human effort and curation. In this paper, we present a novel indirectly supervised approach to train a lenient parser, without access to such human-curated training data. We leverage the huge corpus of mostly correct code available on Github, and the massive, efficient learning capacity of Transformer-based NN architectures. Using Github data, we first create a large dataset of fragments of code and corresponding tree fragments and type annotations; we then randomly corrupt the input fragments (while requiring correct output) by seeding errors that mimic corruptions found in StackOverflow and student data. Using this data, we train high-capacity transformer models to overcome both fragmentation and corruption. With this novel approach, we can achieve reasonable performance on parsing & typing StackOverflow fragments; we also demonstrate that our approach achieves best-in-class performance on a large dataset of student errors.

KEYWORDS
program repair, naturalness, deep learning

1 INTRODUCTION
Most of the development tools we use require syntactically correct, well-typed code; the rest will usually be rejected in some fashion by static or dynamic checks within the tool. However, we often have to confront and work with fragmentary, malformed code. Two prominent settings of concern are a) partial, or flawed, code fragments from StackOverflow, and b) malformed code in student assignments. StackOverflow fragments are often useful, but may not be syntactically complete & correct. Likewise, learners struggle with syntax [23], and frequently make mistakes; the diagnosis and repair of syntax errors can be quite a challenge, especially for beginners. Much instructional time is spent by TAs and professors helping students repair such mistakes. Yet such fragmentary code is often already “mostly correct”, requiring at most a few corrections; hence, it is not unrealistic to consider automating this process [13, 39].

Given the demonstrated success of machine learning at similar tasks in other domains (e.g., fixing errors in writing) there is good prior motivation to attempt several relevant tasks here: a) Leniently parse StackOverflow fragments, so that properly-constructed abstract syntax tree (AST) fragment can be created, even from malformed/partial fragments, and made available for use by an IDE, b) Leniently parse malformed student code, while locating and fixing errors therein. c) Leniently type-annotate such problematic code fragments, providing further information to IDEs to add necessary glue code (declarations, imports, etc). To our knowledge this final lenient typing task above has not been previously attempted, for malformed code fragments.

We develop a novel approach to parsing and typing that relies on indirect-supervision parsing, solely using the (enormous) amounts of (mostly) correct code in Github, which (mostly) compiles and is thus (mostly) syntactically correct, and is (mostly) well-typed. This code is easily processed by (e.g.) Eclipse JDT to yield massive volumes of matched sets of source, ASTs, and type annotations. We take this matched data, and abuse the input source code in various ways to create challenging training data. First, we chop it up randomly to create fragments (with matching types and ASTs) that mimic the kinds of fragments found in StackOverflow. Second, we randomly corrupt it (while retaining correct AST and types on the desired output) to reflect the repair of typical errors found in StackOverflow fragments and in student code. We use this challenging data to train a high-capacity neural network to leniently parse & type imperfect, fragmentary code, by forcing it to minimize its loss against the desired, correct output. To summarize:

1) We use an indirect-supervision approach, which leverages massive GitHub code repos, and high-capacity, efficient neural Transformer architectures, to learn how to leniently fix, parse & type fragmentary and incorrect code without relying on human annotations to create training data.

2) We use a 2-stage approach, with two different neural networks, one of which learns to model (and fix) block nesting structure, and the other which learns to model (and fix) fragments of code. This combination allows us to deal with very long-distance syntactic dependencies within a sequence-based neural network, and thus improve performance on our parsing tasks.

3) Compared to earlier algorithmic work on robust parsers, our approach is fairly language-agnostic: we make minimal assumptions about the language, except for the existence of a parser and static type to create training data. To port to another language also requires identification of block delimiters, expression delimiters, and statement delimiters. (respectively, '{ }', '(' ' '); as will be clear below.

4) We have evaluated our approach using a combination of automated & manual protocols, and demonstrate that we
achieve good performance on the novel typing task, and improve upon a prior baseline for student code repair.

(5) We have released our data, to the extent permissible (for student data), and made our implementation available.

We also point out that our approach could be used as a pre-training adjoint to existing translation based approaches, which rely on human-created datasets; thus in addition to improving on prior performance, our indirect supervision approach could be supplemented with direct supervision to yield further improvements.

2 MOTIVATION & BACKGROUND

StackOverflow is now the preferred source of coding examples for developers. Given any coding quandary, whether it be related to core language features, or specific APIs, one can find answers, with illustrative code examples, on StackOverflow. However, many of the code examples are fragmentary, consisting of a few stand-alone lines of code, which do not represent complete, parseable units of Java code. If these fragments could be parsed into an AST form, and typed, then it would be much easier to paste them into an IDE: the IDE could assist by adding import statements to import packages relevant to the types used in the fragments, adding declarations for needed variables, suggest renaming of variables occurring in the fragment to relevant variables of corresponding types currently in scope, and so on.

But how can ASTs and types be obtained for partial fragments of code? Typing fragments is rarely possible, as they usually don’t provide the necessary import statements and declarations to allow the types of variables in code fragments to be inferred. Parsing fragments to derive ASTs is non-trivial as well. Consider an otherwise correct fragment (from the Android section of StackOverflow):

```
System.out.println(greeting.replace("H","I");
```

For such a well-constructed fragment, one can simply wrap the fragment in a dummy method, and invoke a parser, which would provide an AST for the entire dummy method, from which one can easily extract the parse for just the fragment. Although this is quite effective, we estimate based on a manual examination of a 200 fragment random sample, that ca. 28% of StackOverflow fragments of fragments are not parseable due to various kinds of coding errors. Such fragments are quite common on StackOverflow, often missing semicolons or intentionally including ellipses, e.g.:

```
byte[] bytes = { ... }
```

String `str` = new String(bytes, "UTF-8");

These fragments resist processing via the simple “wrap-and-parse” trick, and require more intelligent handling. Similar parsing headaches abound in student code. In our experiment, we use the Blackbox dataset [7] which contains millions of examples of student submissions from around a million users. Fig 1 is an example from this dataset. Note the missing close paren on line 6, and the extra curly brace on line 9. Simple syntax errors challenge and frustrate beginners [23]. Our lenient parser pipeline can deal with these: it can fix the error, and in the case of the student program, provide a full parse tree that indicates the context where the missing close paren and the extra brace are needed. More generally, our goal is a lenient parser & typer, which can handle a large proportion of such problematic StackOverflow fragments, thus rendering them a greater proportion of them more usable within an IDE.

```
public class StringDemo
{
    public static void main(String[] args)
    {
        String greeting = "Hello";
        System.out.println(greeting.replace("H","I");
        System.out.println(greeting);
    }
}
```

Figure 1: Incorrect (verbatim) student code sample

3 TECHNICAL APPROACH

For the lenient parser, we use a pipeline with two learned DNN stages. The first DNN stage learns to repair a potentially broken nested block structure (“BlockFix”), and the second stage learns repair & parse noisy fragments (“FragFix”). This two-stage approach is needed to handle long-range dependencies, as we discuss below. The lenient typer (TypeFix) is a single-stage learned model. All of the 3 learned models are built using Transformer-based architectures, which are explained below.

3.1 Overall Architecture

We begin with the intuition that Natural Language (NL) parsing is a helpful platform to build learned models to process malformed code. NL is complex, ambiguous, and challenging to parse. Syntactic (“constituency”) parsing is a core NLP problem, that has been refined over decades. Given that code has been shown to be “natural”, NLP parsing technology holds promise for lenient parsing of code.

Traditionally, however, effective NL parsers were tricky beasts that required a lot of algorithm engineering. This approach changed substantially when Vinyals et al. [38] introduced a completely data-driven DNN architecture for parsing. Rather than using a pre-conceived formalism (e.g., probabilistic context-free grammars) with associated algorithms, they render parsing as translation. Just as DNNs could learn to translate from English to German from large datasets of aligned English-German sentence pairs, they argued that DNNs could learn to parse from aligned pairs of sentences and their associated (serialized) parse trees. This remarkable approach worked very well indeed.

To our knowledge, this learned parsing-as-translation approach has not been used for fragmented, noisy code, but it appears well-suited. Unlike with NL, where parses (for training) must be hand-constructed by experts, large amounts of parsed code can be freely harvested by compiling complete projects from GitHub. Our core idea is this: while parsing complete files requires correct code and correct build set-up, we believe that the fragments of code contained therein have regularities (thanks to the well-known naturalness [16] phenomenon) that will allow a well-trained high-capacity DNN to learn to parse most commonly-occurring fragments of code, even wrong ones, without the benefit of build and parsing context. For greater capacity, while Vinyals et al. used an older, Sequence-to-sequence (seq2seq, with attention) recurrent-neural-network (RNN) approach, we use a newer transformer-based model [37] which is
known to outperform older seq2seq approaches. Even so, there are several novel issues that arise when trying to use NL parsers for the task of parsing fragmented and noisy code.

We can happily produce large volumes of training pairs of code + AST using compilers; however, the code must necessarily be correct & complete, to be compilable. We therefore have to artificially fragment and noise-up this code to train our learned parser to deal with such vagaries. Second, the vocabulary in code tends to be much larger [2, 15] than natural language, thanks to identifiers. Normally, larger vocabularies would present a challenge for learning to translate: they would make input embedding and output softmax layers enormous [15], and make translation costlier to learn, and less data-efficient. However, identifiers fall into specific syntactic classes (variables, method names, type names etc.), and can be abstracted out into categories to simplify the parsing task, while keeping vocabulary requirements modest, and brought back in later. Finally, input code fragments (whether from StackOverflow or from student programs) skew much longer than natural language sentences. As a result, syntax errors may arise from inter-related tokens hundreds or even thousands of tokens away from each other, e.g.: one might forget the last closing curly brace ("})" of a very long while loop. To handle the long-dependency problem, we use a two-stage pipeline, where both stages are trained to deal with improper syntax. The first stage, BlockFix, learns to identify and fix common patterns of block structuring in code. The output of this stage, is intended to clearly delineate the beginning and end of blocks, allowing easy segmenting of the code into statement-level fragments, which are typically 50-100 tokens in length. The next stage, FragFix, learns to parse statement-level fragments, fixing any simple syntax errors in the process.

The details are in figure 2. Given a (potentially faulty) code fragment, the learned BlockFix model first identifies the proper nesting structure of the blocks (details below) performing repairs as needed. Using this repaired nesting, the segmenter simply splits the code using block delimiters ("{"}) and statement delimiters (";", and linespacing) to split the input code into fragments, while retaining their point of origin within the block structure. The learned FragFix model repairs & parses each fragment into a fixed tree. These fixed trees are then merged into the original block structure predicted by the the BlockFix model. If repaired code is desired, the tree is “unparsed” to the fixed code.

For the task of lenient typing (viz., the TypeFix model), we use the transformer architecture again, adapting the methods used in gradual typing applications of complete/correct fragments in Javascript [14, 29]. TypeFix has a simpler task, requiring just a single-stage model as described below.

### 3.2 Training Data

Our training data consists of mostly clean, correct code from GitHub. We used code from 50 most popular projects from the 14,785-project dataset published by Allamanis and Sutton [1]. Most of this code is professionally crafted, with complete build environments. Thus it can be easily compiled to produce ASTs and types. We use these to create training data. We also fragmented and added noise to this

---

3Allamanis & Sutton define popularity as the number of forks plus the number of watchers.

4See https://ai.googleblog.com/2017/08/transformer-novel-neural-network.html

---
in the feedforward layer allow powerful & flexible combination of the elements upon which the attention layer that feeds it is focused. Both the encoder and decoder part of the translation model use multiple such levels, each consisting of pairs of stacked self-attention and fully-connected feedforward sublayers. The encoder attends just to its own state (self-attention); the decoder attends both to any previously decoded symbols and to the encoded input at every layer in this stack. This structure allows the model to attend to various parts of an input sequence, while eschewing a recurrent architecture (and the attendant inherently non-parallel BPTT during training), which allows greatly increased model capacity, and more parallelism (and thus speed) during training. The number of sets of sub-layers in the encoder and decoder, as well as the number of heads is configurable depending on the task at hand. The original Vaswani et al paper used stacks of 6 such sets, and 8 heads for attention in every layer. We use several different configurations for our various tasks, as described below; all are based on a configurable, open-source Transformer implementation freely available on GitHub. 5

3.4 Recovering Block Structure: BlockFix

In programs with complicated nested blocks, code tokens can have very long-range syntax dependencies. The length of such dependencies can run into hundreds of tokens; if there are errors, even very powerful DNN models can struggle to identify and repair them. However, if the block structure were correct, it is possible to break code into statement-level segments.

BlockFix has the task of recovering the block structure: it is trained to model common-block structuring patterns, and repair nesting structure if necessary. The repaired structured allows the code to be fragmented into forms that can be passed on to FragFix for repair, and then recombined into a whole AST. We illustrate with some examples.

```java
1 public class Batch {
2     public static String subjects = "English, Maths, Science"
3     public static void main(String args[]) {
4         if (subjects.length() > 50) {
5             System.out.println("subjects too long")
6         }
7     }
8 }
```

**Figure 3: Example Java program for segmentation.**

Consider the code snippet in Figure 3. Lines 2 and 5 are fragments that are syntactically independent of each other; despite missing the ‘‘‘ —our FragFix can fix & parse each separately. Now consider the fragment from line 4-6. To generate the AST for this fragment, we can produce the ASTs for the statements of line 5 ("System . . . ”) and "if" clause ("if ( . . . ) { ”) separately. Now we know exactly where the AST of the former should go: between the two curly braces in the corresponding AST of the latter. If there were multiple statements, we would need to place the ASTs of all the statements sequentially in the block structure recovered by BlockFix.

The BlockFix model requires knowledge of typical block nesting structures, and the wisdom to forgive errors therein. If there are unbalanced curly braces, this model repairs the code by inserting or removing braces where they would be expected to appear. This is based on the assumption that the syntactic structures commonly used in code are natural, with repeating patterns of nesting usage, which can be learned. If we can learn a model which knows these common patterns, it can also be trained to be "forgiving" when curly braces are misused.

**Training BlockFix** We train a separate transformer to learn the common patterns of code nesting. To build this model, we collected one million random files from our dataset. For efficiency, we only used files with less than 1000 tokens for training. Each of these files is parsed by a JDT Parse, resulting in full ASTs on the output. These pairs of files and full AST is our starting point to create training data.

But BlockFix has the sole task of modeling & repairing block nesting structure. Therefore it is trained with input-output pairs that just reflect this task. Consider the code below:

```java
1 public class TableListWriter extends HTMLListWriter {
2     super("Current Tables", "currenttables.html", "tables",
3             outputDir);
4     if (ListClosing()) {
5         WriteCloseMarkup();
6         return;
7     }
8     else {
9         CloseList();
10         return;
11     }
```

We first abstract the input source code into an abstract form, like below:

```java
public class simple_name extends simple_name {
    public simple_name ( public simple_name paren_expression {
        expression if paren_expression { expression } else
        { expression } }
```  

We remove essentially everything from the input, except curly braces and keywords, and abstract out all other identifiers, constants, expressions, and delimiters. Consecutive sequences of expressions are collapsed into one abstract expression. Identifiers are also abstracted. We can then simulate common structure-related syntax errors by corrupting this abstraction slightly and tasking the model with reproducing the original, uncorrupted (abstracted) code. Specifically, we add additional braces into half of the examples, while dropping some from the remaining half, split randomly between open and close curly braces for this noising step. Training on lots of such abstracted pairs allows the BlockFix model to learn how syntactic constructs are most frequently nested in code. In all cases, the desired output shows where the mistakes are inserted, so the segmenter learns to be both lenient, and provide the correct fix if an error is detected.

The above process transforms a program with potential syntax errors into an abstracted structure with placeholders for all abstracted expressions. What happens to these bits that are abstracted out when actually doing the task? These removed bits constitute the fragments that are sent along to FragFix for the next stage. Of course, we record & track these bits that fit into the abstracted input,

---

5See https://github.com/Lsde/define/attention-is-all-you-need-keras

Anecdotal: additional braces are often next to existing braces; we therefore simulate this in 70% of cases while inserting them in another random location for the rest.
so we can reassemble the ASTs produced by FragFix from these code bits into the abstract block-structure produced by BlockFix.

Using this abstracted, noised segment training data, we train a transformer (translation) model. This model uses a stack of two layers in both the encoder and the decoder, which we found was sufficient for our training setup. Each layer includes multi-head self-attention and position-wise fully connected feed-forward layers, and in the case of the decoder also multi-headed input (encoder) attention. Because of our chosen input/output abstractions, the input and output vocabulary size is limited to 54. As in the original Transformer architecture, all our layers use 512-dimensional states, which is split across 8 parallel heads for attention, and projected into 2048 dimensions (and back) in each pair of feed-forward layers. Except for the number of layers \( N \) (we use \( N = 2 \) instead of \( N = 6 \), we replicate all the hyper-parameters described in [37], We use an Adam optimizer as a learner with 4000 warm-up steps. We apply layer normalization after each sublayer. To prevent overfitting, we employ residual dropout (0.1) for regularization. We also add positional encoding to the inputs and vary the learning rate following the recommendation of Vaswani et al. [37]. We trained our model for 10 epochs with a batch size of 64 fragments. The limited vocabulary allows most of the model capacity to be used for learning & repairing nesting structures, and we get good performance, as explained below.

### 3.5 Recovering ASTs from fragments: FragFix

The next step is to train a lenient fragment parser that can fix & parse fragments. We first gathered all the Java files with less than 10,000 tokens and produced the ASTs using Eclipse JDT. These (mostly) correct, well-structured programs are easily broken into fragments using the semicolon (\( ; \)) and curly braces (\{ \} and \( \) ) as breaking-points. We tried to keep the fragment-length roughly uniform, with limited variance, and removed duplicates. Our belief was that these fragments, despite their disparate origins would nevertheless have repeating syntactic patterns that FragFix would learn to capture, even out of context. We collect 2M such fragments.

Our next task was to train FragFix to be lenient with respect to common errors. Santos et al. [32] find that, in BlackBox, most (57.4%) of syntax errors arise from single-token errors (extra, missing, or wrong tokens). Extra tokens account for about 25% of single-token errors; missing tokens for about 69%, and substitutions for about 8%. Based on a manual examination of a small sample (around 1,000 examples) of the data, we noted that the vast majority of these single-token errors centered around certain tokens we may call separators; tokens such as commas, semicolons, periods, all of these single-token errors centered around certain tokens we may call separators; tokens such as commas, semicolons, periods, all types of brackets (\( \{ \}, \{ \} \), \{ \}) and the string separator “+”.

Based on the commonality of these errors, we sought to teach FragFix to robustly recover from them. We therefore inject occurrences of these errors into the input source code in our training data (details of error-injection below). These “sinful” inputs were paired with a “redeemed” AST, which indicated the location of the error, and it’s repair, as illustrated below:

```
Code: tot x = 0
AST: (#VariableDeclarationStatement (#PrimitiveType )
(#VariableDeclarationFragment (#SimpleName) (#PunctTerminal) (#NumberLiteral))
#missing-semicolon)
```

The “redeemed” AST in this training sample clearly signals the #missing-semicolon in the code fragment, which can be used to repair the code. Also note that we drop concrete code tokens from the desired output, retaining just the AST nodes; this reduces not just the size of the output sequence (the serialized “parse”) but also of the output vocabulary, simplifying the learning task, and allowing us to better leverage the capacity of the DNN learner. During actual parsing, we know the true input tokens, and can reproduce the full ASTs by inserting the tokens into the output in the same order. We also abstract all the numeric values to 0 and all strings and characters to their empty values ("", "), since these values tend to increase the vocabulary size without contributing to the structure of the AST.

Regarding simulating typical errors, we observed from an examination of the data (both StackOverflow and student code), that erroneous inserts of separators do not occur uniformly at random locations; instead, they predominantly occur next to other separators. So, we often see “stutter” errors of the form “math. log(35.0)” or “\( x = 0 \) \{ \}”, with repeated separators, but rarely ones of the form “math. log(35.0)” or “\( x = 0 \) \{ \}”. To learn to “forgive” such errors, we prepared training data similarly biased towards stutters. To mimic these errors, we randomly choose separators within fragment as described above, and with 70% probability repeat that separator, while the remaining 30% are randomly inserted elsewhere in the code. These errors are paired with the “redeemed” AST indicating the position of the extra token. This data trains our parser to produce an AST that both indicates what was wrong, where, and how to fix it.

The transformer architecture for FragFix is identical to that used for BlockFix; only the input and output configurations are slightly different. Our output now includes just the vocabulary of possible AST nodes in the tree, and excludes all input tokens. For Java, this means that size of our output vocabulary is just 95 tokens. That simplifies the translation task greatly; we found that transformer with 2 layers is sufficient. Our encoder input does include regular code tokens, which can be highly diverse; thus, we create a limited input vocabulary of the 64,833 most common tokens by discarding tokens which appear less than 12 times in the training corpus. We use the same training regime here as for BlockFix.

### 3.6 Final Lenient Parsing Pipeline

We summarize the entire pipeline using the algorithm below, specifically for processing student code; note that the StackOverflow parsing is slightly different, as we explain below. First, we check if the input code fragment has balanced braces, using a simple counter-based algorithm (line 1.2). If not, (line 3) it is is sent through BlockFix which fixes the block structure. Next (5), any block delimiters, semi-colons, and linefeeds are used as markers to identify locations where the input source code can be split into fragments. These markers are also used later to reassemble the fragments. Line 6 splits the input code using these markers into a list of fragments. Each fragment is then parsed (leniently) by FragFix (lines 8,9) and then the resulting fragments are used to re-assemble the full AST (line 10). Finally, using the indicated errors (missing/extra operators, delimiters etc), the repaired source code is generated in line 12.
For the StackOverflow parsing task, there are two differences. First, many StackOverflow fragments are quite short. Since FragFix can manage fragments shorter than 40 tokens, we just skip the BlockFix phase for these. Second, since we only need the AST, we skip the code generation step on line 12.

```
input: Code fragment $P$
output: Fixed-up Code Fragment $P$

1. $abs \leftarrow \text{FindBraces}(P)$;
2. if NotBalancedBraces($abs$) then
   3. $abs \leftarrow \text{BlockFix}(P)$;
   4. end
5. $segs \leftarrow \text{segment}(abs, P)$;
6. $frags \leftarrow \text{splitProgram}(segs, P)$;
7. $AST \leftarrow \text{initializeAST}(abs)$;
8. for $frag \in frags$ (in order) do
   9. $fragAST \leftarrow \text{FragFix}(frag)$;
   10. $AST \leftarrow \text{Inst}(fragAST, abs, segs, AST)$;
end
12. $P \leftarrow \text{GenerateCodeFrom}(AST)$;

Algorithm 1: Steps Followed for Student Code Correction
```

### 3.7 Lenient Typing

Many StackOverflow fragments omit declarations or imports. Therefore, using even a simple fragment is challenging, since identifier types cannot be easily derived. Prior work [29] showed that it is possible to guess and type annotations for gradually typed languages such as Typescript. Hellendoorn et al. [14, 22] use DNNs to predict types, formulating this task as a sequence tagging problem because there is a one to one mapping between the input token and types [14]. They used an RNN architecture. With non-identifiers receiving an empty annotation. None of these approaches have been applied to Java StackOverflow fragments that lack imports and declarations, yet having type information for a fragment may enable a downstream IDE to suggest declarations, imports etc (or even renamings for variables in the fragment to variables of the same type that are available, and in scope) when re-using that fragment.

We followed an approach similar to [14], except using the transformer-based model instead of an RNN. Our training data consists of the same projects as before; we used Eclipse JDT to derive the types for all identifiers in these Java files, while marking non-identifiers (e.g., keywords, operators, delimiters) with a special ’no-type’ symbol (in the following example, we use a special symbol “~”). After generating types for every token (in all complete files), we created random (cross-project) fragments for training data, as we did for the parsing task, with corresponding types as derived by JDT. In total, we extract ca. 2 million fragments from the projects with the desired types, all similar to this pair below:

```java
if { something } { Object o = new Object ( ) ; }
~ ~ boolean ~ ~ ~ java.lang.Object ~ ~ ~ ~ ~ ~ ~
```

Out of 14 tokens in this fragment, 2 are identifiers for which types are provided; one primitive and one fully quantified. The other tokens are deterministically tagged with “~” to simplify the model’s task.

### Training Transformer Model for Lenient Typing

As before, we used a transformer-based model for typing of Java fragments from StackOverflow. However, we formulate the typing task as a sequence-tagging problem (similar to part-of-speech tagging, Named Entity Recognition etc.) since the input and output lengths are always identical, unlike with the translation task. Also, the output vocabulary (the set of possible types) is much larger. Therefore, the translation model used for parsing is not directly applicable. Sequence tagging is in some ways an easier task than translation: we do not need to digest the full input sequence. Types can mostly be assigned based on local information, so there is no need for a full encoder mechanism to encode the full input; the task can be performed with a single “decoder” element. In the absence of the encoder element, the decoder simply attends (using multiple heads) to various tokens of the input sequence, as it generates tags (types) on the output.

For this task, the hyper-parameters are set as recommended by Vaswani et al. [37]. For the single “decoder” element, we use 6 layers (each consisting of multi-head attention + feedforward) instead of 2 to provide enough capacity to model the much larger input and output vocabulary. We keep all other hyperparameters unchanged except for the learning rate and warm-up steps. We set the initial learning rate as 0.2 and warm-up steps at 1000. We also use a warm restart for the learning rate [21] by resetting the learning rate to its initial value after each epoch. Note that because of the one-to-one mapping, the length of the input and output sequence must be same. We include a token into the input vocabulary if the token appears at least 35 times in the training corpus; the cutoff value for output (type) vocabulary is 50, making the size of the input and output vocabulary 40, 316 and 18, 673 respectively. We prevent gradient updating for the non-type token to simplify the learning process.

We trained the model for 10 epochs with batch size of 4000 tokens.

### 4 Evaluation & Results

We used a mixed methods approach to evaluating our 3 tasks: StackOverflow parsing, student code correction, and StackOverflow typing, based on the characteristics of each task. For the lenient parsing of StackOverflow fragments, we used a combination of automated and manual methods. For the lenient typing in StackOverflow, we used a manual evaluation. Finally, for the student code correction task, we used fully automated evaluation. The rationale and results are presented separately in sections below.

#### 4.1 Performance of Lenient Parsing

As mentioned in Section 3.5, our lenient parser was trained with 2M fragments collected from GitHub. After training, we first ensure that our model has the capacity to parse by evaluating it on 85,000 held-out fragments from the same data source. These test fragments mostly originated from correct code, which could be parsed automatically by Eclipse JDT to create ASTs as a ”golden” benchmark. For these 85,000, we found that 97% of the outputs from the trained lenient parser exactly matched the output from Eclipse JDT, thus confirming the capacity of the transformer model to perform “parsing as translation”. We also observed that FragFix’s accuracy decreased with fragment length, with performance sharply decreasing above a 40-token length. Of course, our main
interest is the performance on actual StackOverflow fragments, which may be syntactically erroneous, and thus impossible to parse directly with Eclipse JDT; our approach to this is described next. Parsing performance on StackOverflow Malformed fragments, by definition, could not be automatically parsed in general, and so required manual checking. Therefore the number of StackOverflow test samples is limited by required human effort. Still, we sought a sufficiently large & representative sample to get a good estimate of the performance. We collected the StackOverflow fragments from the public Google BigQuery dataset. For this experiment, we collected the answers for questions tagged with "Java." After that, we isolated the fragments using "<code>" tag used for presenting code snippet. We randomly chose a total of 200 fragments with various lengths for evaluation.

Our goal here is to measure how often the lenient parser produces an AST that could easily be used by downstream tools, such as IDEs. For this reason, we believe the standard BLEU-score measure used for translation-based tools is unsuitable. Instead we used a repeatable, objective 4 class categorization of outputs: a) Correct: the output AST exactly matched the correct AST. b) Autofixed: the model’s output matched the correct AST after a small post-processing step of adding or removing close parens, ‘), at the very end of the output to balance all open ‘( parens. No other change is allowed. c) Partial: the output AST only matched the top-level node of the correct AST, and d) Incorrect: none of the above. The Correct and Autofixed classes are chosen to capture cases which allow easy, automatable downstream IDE use.

One additional caveat: in the absence of context, it’s virtually impossible to distinguish between field (class member) declarations and variable (local variable) declarations in small fragments. When pasting in a parsed AST fragment, it should be quite possible for an IDE to adapt the declaration form as needed; so in our evaluations, we ignored this distinction. Either was considered correct.

Given a StackOverflow fragment, we used a two-stage scheme for checking ASTs produced by the lenient parser. First, we attempted to embed the fragment within a class (class class_name { . . . }) or method (void method_name () { . . . }) wrapper, thus turning it into a unit potentially parseable by JDT. If the JDT would parse the fragment within such a wrapping, we had the exact AST for the fragment, and use that as the correct baseline. If such a wrapper could not be found, we manually evaluated the lenient parser output. Of the 200 fragments, 123 could be parsed after wrapping by JDT, and 55 could not. The remaining 22 fragments were not Java, but XML, Gradle, data etc. The outputs from the 178 Java fragments were categorized as above; the correct category was checked automatically whenever we had “Golden” results from JDT. The rest were manually checked by the two authors independently, strictly following the protocol laid out above.

Of the 123 JDT-parseable fragments, the lenient parser got 90 corrects, no autofixeds, 27 partials, and 6 wrong. Overall, the lenient parser, by itself, could produce ASTs in 126/178 cases (or roughly 71% of cases) in a form that was easily usable by downstream tools. This may seem like only a slight improvement on the 123 of the simple approach of wrapping and parsing with JDT, but the models did not actually solve the same fragments. Instead, on the 55

fragments on which JDT wouldn’t work, we produced an 30 correct, 6 autofixes, 16 partial, and only 3 wrong ASTs. In other words, while simple wrapping and then parsing works in about 69.1% (123/178) of cases, fragments that resist parsing with this trick can then be fed to our approach; this hybrid approach allows for parsing a total of 89.3%, (123+36 = 159/178) of fragments in our sample (Wald confidence interval 85-94%).

4.2 Performance of Lenient Typer

Our lenient typer was trained on about 2M training instances (49M tokens). To first get a sense of the performance potential, we turned again to our 82K held-out fragments from the same data source, with their “golden” types from the JDT. On this set, we achieved 95.56% top-1 and 99.44% top-5 accuracy. For the top 1,000 most frequently-used types in our data, the top-1 and the top-5 accuracy are 97.10% and 100%; for primitive types in particular, TypeFix is virtually infallible in this automatically created dataset. This makes sense given that Java is a statically typed language and these files contain no syntax errors; it implies that our model has accurately learned the distribution of types given tokens. The real test will be the actual StackOverflow fragments, where we need to manually check the predicted types.

Typing Performance on StackOverflow As before, we collected StackOverflow fragments from the public Google BigQuery dataset and processed them using our learned lenient typer. The outputs in this case, however, have to be checked entirely by hand, since most fragments lack the necessary build environment information (e.g., CLASSPATH, imports) and cannot be automatically processed to get “Golden types”. We therefore selected 75 code fragments from highly rated answers (1000-3500 net positive votes). To get a broader diversity of samples, we collected these from 3 categories (25 from each): a) Popular types consisting of the 5 most popular (as identified by Qiu et al [28]) Java classes: (java.lang.String, java.lang.Override, java.util.List, java.lang.Exception, & java.lang.Object), b) Core Java types consisting of any fragments tagged with just “Java” in StackOverflow, and c) Android types consisting of types that occur in the Android API, that don’t fall to the other two categories. The Android category, in particular, can inform how the amount of available data affects the performance of our tool; Android API classes (though clearly important) were found in only 12 projects in our dataset, which accounted for about 4.5M tokens out of a total of corpus size of 52M tokens in all the projects. Therefore TypeFix has a more limited exposure to Android API types during training.

The other two categories were well represented. We report our evaluation based on the proportion of identifiers in each fragment that were correctly typed. If used downstream in an IDE, the incorrect identifiers would have to be fixed manually. This number is shown on the y-axis of Fig 4 as a percentage. If all the identifiers in a fragment were labeled correctly, the sample would score at 100%. We break the scores into 3 groups by Category, and show a boxplot for each. As can be seen, there appears to be a correlation between the amount of training data and performance.

Footnotes:

7See https://cloud.google.com/bigquery/public-data/
We see the best performance for the Popular category (median 100%) and Core (median 90%), and a lower median for Android, around 50%. These results suggest that training TypeFix on even larger datasets could further improve performance; we discuss additional approaches to improve performance later (§ 6).

We also applied our approach to files with 2 and 3 syntactical errors. Out of the remaining 700,000 pairs, there are approximately 248K examples with two syntax errors [32], and 94K files with 3 errors. To estimate performance in these two categories, we chose 50K files with 2 errors, and 50K with 3 errors. In the two error category, we measured 19% top-1 accuracy, and for the 3 error files, we noted 9% top-1 accuracy. We note that Santos et al do not consider files with more than 1 error.

In summary, out of all 1.7M programs with syntax errors, 57.4% are single token errors (as per Santos et al [32], Table 1), of which we can fix 54.1% perfectly (top-1 correction), yielding an estimated top-1 fix rate for all files with syntax errors in the Blackbox of about 31%. If we consider the top-1 accuracy for up 3 syntactical errors, we estimate (using Santos’ Table 1 estimates of proportions) that we could fix approximately 34% of these files. In the remaining part of this section, we discuss various aspects of our model’s performance.

Ablation: The BlockFix’s role We used BlockFix to help fragment the code, since all DNNs (LSTMs or Transformers) struggle with long-range syntax dependencies. So how much does it actually help? We used 20,000 randomly chosen files to measure this effect. We found quite a large number, 4,925 (24.62%) of files with unbalanced curly braces. Of these, our complete pipeline could fix 2,865 (58.15%) cases, yet FragFix per se, without BlockFix, could only fix 36 (0.9%)! For the remaining 15,075 files, FragFix did still work fairly well, incurring an overall MRR drop from 0.56 to 0.42. Thus, we believe that BlockFix plays a useful and complementary role.

Performance vs. file length Most student programs are less than 1K tokens in length (though some are much longer). It is reasonable to expect performance to decrease with (much) larger files; indeed, even the BlockFix could struggle with large files, since even abstracted version of these can have hundreds of tokens. Figure 5 shows how (Top-1 accuracy) performance decreases with length. For simplicity, we bucketed the samples by length, and show average performance and confidence intervals for each bucket. We can see that our pipeline achieves a peak performance of around 65% accuracy for files with less than 300 token, while accuracy decreases to ca. 20% around 3000 tokens. Note that the confidence interval increases with length, because there are fewer and fewer samples in our data (bucket size is indicated above each bar). It should be noted that files under 1000 tokens, our top-1 accuracy is around 56%, which compares favourably with previous approaches.

Time vs. Length. Our biggest performance overhead is the DNN computation time, especially since we use two separate models; we can expect our pipeline to take longer for bigger files. To assess this, we measured performance on a random sample of 20,000 files, and show separate plots for cases where we succeeded and failed in Figure 6. Note that these are scatter plots that additionally signal the prevalence of datapoint buckets with their color gradient. Immediately evident is the flatter slope for the failing cases, with many failing quickly: these usually fail earlier in our pipeline—either BlockFix fails to properly balance and nest the input source code, or there other errors that inhibit fragmenting of the code, so we abort before getting to FragFix. Figure 6 also shows that the processing time generally increases with the number of tokens. Even so, most files are processed fairly quickly. Our median repair time is around 1.5 seconds, which is about 10% of the median repair
Figure 5: Performance of code correct with increasing number of tokens in the file with confidence interval

Figure 6: Processing time vs. number of tokens in the file

time reported in the BlackBox dataset (gathered from actual human-generated fixes, see Table 1 [6]), suggesting that we could provide timely help to students quite often. In all, we process 95% of files in under 10 seconds.

5 RELATED WORK

Island Parsing The main objective of the island parsing problem is to find "islands" of structured content (e.g., code snippets) from "water" of unstructured data (e.g., English descriptions). Since useful code snippets are often found in mixed English-code corpora in manuals, web sites, etc, island parsing can help programmers by carving out useful bits of code. Moonen and Van Deursen introduced a grammar-based approach to solving island parsing problem [25, 36]. Synytskyy [33] demonstrate the use of this approach for dealing with ASP fragments, which mix comments, HTML, and Visual Basic. Bacchelli et al. applied two approaches to solve island parsing problem: generalized LR (SGLR) and Parsing Expression Grammars (PEGs) [3]. Rigby et al. did not separate the code snippet from StackOverflow fragments; instead, they applied a set of regular expressions to approximate the Java construct, e.g., qualified terms, package names, variable declarations, qualified variables, method chains, and class definitions [30]. While this is a powerful approach, current methods depend on hand-crafted grammars. Our approach is rather more general, requiring just the availability of a parser that can produce ASTs. Though Island parsers might (See [3], §6.3) be applicable to code with syntax errors, we are not aware of any prior work or benchmark where they were used to correct student code to compare our work against.

A related line of work is partial program analysis, which attempts to derive types and data-flow facts from incomplete programs [11, 31]. Most of these works in the area of partial program analysis consider "fragments" to be either complete files, or complete procedures, rather than the kinds of noisy bits we consider. The one available tool, PPA\(^8\) only works for Java 1.4 or 1.5. Our test set (and training corpora) include features from later releases (such as Collections). We also note a considerable body of prior work in finding, using, and mining code examples from the web [18, 26, 27, 34]. Our work is generally complementary to this line of work.

Predicting the Type of Identifiers Dagenais proposed some predefined strategies to infer the types of identifiers, e.g., by using the type of the identifier on the other side of assignment operator [11]. Raychev et al. applied statistical inference model for inferring for JavaScript [29]. Hellendoorn et al. [14] use DNN to predict the types. While these works use the implementation to infer types, Malik et al. extract type information from natural language descriptions (comments, identifiers) [22]. However, none of these machine learning-based approaches were applied specifically to inferring the type of incomplete code fragments; they were trained and tested on complete source code files.

Fixing Compile Errors Meshah et al. describe DeepDelta, which fixes mostly identifier name related errors, not syntax errors. DeepDelta was developed and tested on code that led to build errors, all from professional developers at Google. The authors also assume that precise knowledge of the location to be fixed is available [24], which we do not.

DeepFix, which uses a seq2seq (with single attention) translation model, also works on student programs and repairs syntactic errors in C with 27% top-1 accuracy [13]. The programs in Deepfix’s dataset range in size from 100-400 tokens, making them substantially much smaller than those in BlackBox. Gupta et al. applied reinforcement learning to a very similar dataset [12], reporting 26.6% accuracy of their tool (RLAssist). Bhatia et al. achieved slightly higher accuracy than Deepfix and RLAssist on repairing student code (31.69% accuracy) [5]. However, these numbers are difficult to compare across datasets; e.g., Bhatia et al.’s dataset consists of solutions to just 5 different programs, which is considerably less diverse than BlackBox (which collects data from all users of BlueJ, not just ones doing particular homeworks). The programs in [5] are also relatively small, ranging from ca. 40 to 100 tokens. Santos et al. used the BlackBox dataset, and were able to fix almost half of instances of student code with single syntax errors [32]; as reported earlier, we exceed their MRR performance, and can also fix programs with more than one error.

Deep-learning for Code Repair There is considerable interest in applying deep learning to the problem of code repair. Typically, such work uses a large dataset of bug-fixing commits to train seq2seq type models [35], or to find relevant repairs for patching [40]. Translation models that use tree to tree (rather than seq2seq) have also been proposed [8, 9]. These approaches are mostly not aimed at syntax errors, but rather at semantic errors exposed by failing tests.

\(^8\)http://www.sable.mcgill.ca/PPA/
6 DISCUSSION & FUTURE WORK

6.1 Threats & Caveats

Despite the observed performance, some caveats apply. For the StackOverflow parsing task, even with nearly 90% accuracy for the combined approach, developers will still have to deal with erroneous repairs. Although we did not run experiments with developers, we can expect that, if used within an IDE, features like syntax-directed indenting should make it fairly easy for developers to assert whether the pasted-in AST is indeed correct. Our manual assessment relied on a random sample; the confidence interval reported (§ 4.1) gives a sense of how the actual performance might vary.

For the student code syntax correction task, our top-1 accuracy estimate (matching the exact fix produced by the student) is based on a very large random sample, and is thus likely to be close to the true value. Although higher than previous work, we still reach only 54% top-1 accuracy; thus, suggested repairs may still be incorrect, either syntactically or semantically. To ensure that the fix is good syntactically, it would be prudent to apply the fix and run the compiler or a parser, as a check (which can be done automatically) before offering the suggested fix to the user. Semantic correctness of the suggest repair (or atleast equivalence to what student intended) is much more problematic to determine, and can only be assessed with test cases or invariants provided by the instructor.

Our performance on the lenient typing task is good for popular types, but clearly declines with decreased training data availability. While we hope to improve the performance in future work (see below), the type annotations especially for less common types would need review by the developer if used in an IDE.

6.2 Future Work

There are several interesting directions for extensions of this work. Our lenient parser uses indirect supervision on noised data and was not trained on student or StackOverflow data. However, there is a lot of student data available, which should provide a more precise signal, if relatively less training data. In that light, it is entirely reasonable to see our current setting as a form of pretraining and additionally fine-tune our model on e.g. real student data, using the true fixes as targets. This might improve performance in ways attenuated to student data specifically.

Since our lenient parser provides an actual repaired parse tree, and not just a suggested edit, there is also an opportunity for each suggested fix to provide some pedagogical value and/or explanation as to why some token(s) should be added, removed, or changed. This is a promising direction we hope to pursue.

Lenient typing performance is currently constrained, first because of limited data, and secondly because the vocabulary is limited for the input embedding layer. As noted by Malik et al. [22] quite a bit of type information is carried in the identifier names; so we believe that approaches like [2, 19], which intelligently decompose identifiers into constituent sub-tokens based on co-occurrence frequencies, can enhance performance for the typing task, since the compositions of identifiers can be used predict their types. This will require recasting the typing task as translation (rather than tagging) since input and output lengths won’t match anymore. Finally, we also believe that both lenient parsers and types for domain-specific IDEs (such as Android Studio for Android, or Visual Studio for .NET) could benefit from training on large volumes of code rich in specific APIs of interest to the target audience.

7 CONCLUSION

We have described an approach to processing (parsing & typing) incomplete and erroneous code, from students and StackOverflow. We generate large volumes of training data for parsing & typing erroneous code by starting with code which syntactically correct, and well-typed, which can be parsed and typed with a standard parser, and then fragmenting and injecting noise into this data to train a lenient parser and typer. We use a parsing-as-translation approach, based on the state-of-the-art Transformer model, while using a tagging approach for typing. To deal with the long-distance dependencies of source code, we first segment the code into fragments, using statement delimiters and nesting via curly braces. Since code could have improper nesting, we train a separate model to fix missing or extra nesting structures. This pipeline, consisting of BlockFix and FragFix, performs better on the large and diverse BlackBox dataset than previous work. It also performs well for StackOverflow fragment parsing, and has some degree of success on the typing task. In future work, we hope to pursue further improvements on the typing task, and seek integration with an IDE, to help fix errors, and also to help paste-in code from StackOverflow.

Finally, we will make our implementation, and some of the data available at https://doi.org/10.5281/zenodo.3374019. The BlackBox data is not redistributable, and must be explicitly requested from the authors [7].

REFERENCES

[1] Miltiadis Allamanis and Charles Sutton. 2013. Mining Source Code Repositories at Massive Scale using Language Modeling. In The 10th Working Conference on Mining Software Repositories. IEEE, 207–216.
[2] Hlib Babi, Andrea Janes, and Romain Bobbes. 2019. Modeling vocabulary for big code machine learning. arXiv preprint arXiv:1904.01873 (2019).
[3] Alberto Bacchelli, Andrea Mocci, Anthony Cleve, and Michele Lanza. 2017. Mining structured data in natural language artifacts with island parsing. Science of Computer Programming 150 (2017), 31–55.
[4] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2014. Neural machine translation by jointly learning to align and translate. arXiv preprint arXiv:1409.0475 (2014).
[5] Sahil Bhatia and Rishabh Singh. 2016. Automated correction for syntax errors in programming assignments using recurrent neural networks. arXiv preprint arXiv:1603.06129 (2016).
[6] Neil CC Brown and Amjad Altadmi. 2017. Novel Java programming mistakes: Large-scale data vs. educator beliefs. ACM Transactions on Computing Education (TOCE) 17, 2 (2017), 7.
[7] Neil Christopher Charles Brown, Michael Kolling, Davin McCall, and Ian Utting. 2014. Blackbox: a large scale repository of novice programmers’ activity. In Proceedings of the 45th ACM technical symposium on Computer science education. ACM, 223–228.
[8] Saikat Chakraborty, Miltiadis Allamanis, and Baishakhi Ray. 2018. Tree2Tree neural translation model for learning source code changes. arXiv preprint arXiv:1810.00314 (2018).
[9] Saikat Chakraborty, Miltiadis Allamanis, and Baishakhi Ray. 2018. Tree2Tree Neural Translation Model for Learning Source Code Changes. CoRR abs/1810.00314 (2018). arXiv:1810.00314 http://arxiv.org/abs/1810.00314
[10] Junyoung Chung, Caglar Gulcehre, Kyunghyun Cho, and Yoshua Bengio. 2014. Empirical evaluation of gated recurrent neural networks on sequence modeling. arXiv preprint arXiv:1412.3555 (2014).
[11] Barthelemy Dagenais and Laurier Hendren. 2008. Enabling static analysis for large code bases. In ACM Sigplan Notices, Vol. 43. ACM, 313–328.
[12] Rahul Gupta, Aditya Kanade, and Shirish Shevade. 2018. Deep reinforcement learning for programming language correction. arXiv preprint arXiv:1801.10467 (2018).
[13] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirin Shevade. 2017. Deepfix: Fixing common C language errors by deep learning. In *Thirty-First AAAI Conference on Artificial Intelligence*.

[14] Vincent J Hellendoorn, Christian Bird, Earl T Barr, and Miltiadis Allamanis. 2018. Deep learning type inference. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ACM, 152–162*.

[15] 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*.

[16] 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*.

[17] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation, 9*, 2 (1997), 1735–1780.

[18] Reid Holmes, Robert J Walker, and Gail C Murphy. 2005. Strathcona example recommendation tool. In *ACM SIGSOFT Software Engineering Notes, Vol. 30, ACM, 237–240*.

[19] Rafael-Michael Karampatsis and Charles Sutton. 2019. Maybe Deep Neural Networks are the Best Choice for Modeling Source Code. *arXiv preprint arXiv:1903.05734* (2019).

[20] Michael Kölling, Bruce Quig, Andrew Patterson, and John Rosenberg. 2003. The Blue system and its pedagogy. *Computer Science Education 13, 4 (2003), 249–268*.

[21] Ilya Loshchilov and Frank Hutter. 2016. Sgdr: Stochastic gradient descent with warm restarts. *arXiv preprint arXiv:1608.03983* (2016).

[22] Rabea Sohail Malik, Jibesh Patra, and Michael Fradel. 2019. NL2Type: inferring JavaScript function types from natural language information. In *Proceedings of the 41st International Conference on Software Engineering, IEEE Press, 304–315*.

[23] Michael McCracken, Vicki Almstrum, Danny Diaz, Mark Guzdial, Dianne Hagan, Yifat Ben-David Kolikant, Cary laser, Lynda Thomas, Ian Utting, and Tadeusz Wilusz. 2001. A multi-national, multi-institutional study of assessment of programming skills of first-year CS students. In *Working group reports from ITiCSE 2001*, IBM Press, 266–278.

[24] Ali Mesbah, Andrew Rice, Emily Johnston, Nick Glorioso, and Eddie Aftandilian. 2019. DeepDelta: learning to repair compilation errors. (2019).

[25] Leon Moonen. 2001. Generating robust parsers using island grammars. In *Proceedings Eighth Working Conference on Reverse Engineering, IEEE, 13–22*.

[26] Seyed Mehdi Nasehi, Jonathan Silitto, Frank Maurer, and Chris Burns. 2012. What makes a good code example?: A study of programming Q&A in StackOverflow. In *2012 28th IEEE International Conference on Software Maintenance (ICSM), IEEE, 25–34*.

[27] Luca Ponzanelli, Gabriele Batova, Massimiliano Di Penta, Rocco Oliveira, and Michele Lanza. 2014. Mining StackOverflow to turn the IDE into a self-confident programming prompter. In *Proceedings of the 11th Working Conference on Mining Software Repositories, ACM, 102–111*.

[28] Dong Qiu, Bixin Li, and Haretom Leung. 2016. Understanding the API usage in Java. *Information and software technology 73* (2016), 81–100.

[29] Veselin Raychev, Martin Vechev, and Andreas Krause. 2015. Predicting program properties from big code. In *ACM SIGPLAN Notices, Vol. 50, ACM, 111–124*.

[30] Peter C Rigby and Martin P Robillard. 2013. Discovering essential code elements in informal documentation. In *2013 35th International Conference on Software Engineering (ICSE), IEEE, 832–841*.

[31] Atanas Rountev, Barbara G Ryder, and William Landi. 1999. Data-flow analysis of program fragments. In *Software Engineering/ICSE ’99, Springer, 235–252*.

[32] Eddie Antonio Santos, Joshua Charles Campbell, Dhvanii Patel, Abram Hindle, and José Nelson Amaral. 2018. Syntax and sensibility: Using language models to detect and correct syntax errors. In *2018 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER), IEEE, 311–322*.

[33] Nikita Synytskyy, James R Cordy, and Thomas R Dean. 2003. Robust multilingual parsing using island grammars. In *Proceedings of the 2003 conference of the Centre for Advanced Studies on Collaborative research, IBM Press, 266–278*.

[34] Suresh Thummalapenta and Tao Xie. 2007. Parseweb: a programmer assistant for reusing open source code on the web. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering, ACM, 204–213*.

[35] Michele Tufano, Jevgenija Pantuchchina, Cody Watson, Gabriele Batova, and Denys Poshyvanyk. 2019. On learning meaningful code changes via neural machine translation. In *Proceedings of 41st International Conference on Software Engineering, IEEE Press, 25–36*.

[36] Arne Van Deursen and Tobias Kuipers. 1999. Building documentation generators. In *Proceedings IEEE International Conference on Software Maintenance-1999 (ICSM’99) Software Maintenance for Business Change*(Cat. No. 99CB36360), IEEE, 49–49.

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

[38] Oriol Vinyals, Lukasz Kaiser, Terry Koo, Slav Petrov, Ilya Sutskever, and Geoffrey Hinton. 2015. Neural machine translation without alignment. In *Advances in neural information processing systems*. 5998–6008.

[39] Ke Wang, Rishabh Singh, and Zhendong Su. 2018. Search, Align, and Repair: Data-driven Feedback Generation for Introductory Programming Exercises. *SIGPLAN Not. 53, 4 (June 2018), 481–495. https://doi.org/10.1145/3296979.3192384*.

[40] Martin White, Michele Tufano, Matias Martinez, Martin Monperrus, and Denys Poshyvanyk. 2019. Sorting and transforming program repair ingredients via deep learning code similarities. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER), IEEE, 479–490*.