Noname manuscript No.
(will be inserted by the editor)

Learning Lenient Parsing & Typing via Indirect Supervision

Toufique Ahmed · **Premkumar Devanbu** · **Vincent Hellendoorn**

Received: date / Accepted: date

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 could somehow be parsed & typed; this makes such code 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 and type imperfect code requires a large training set including many 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 performs well on shorter student error program and achieves best-in-class performance on longer programs that have more than 400 tokens. We also show that by blending Deepfix and our tool, we could achieve 77% accuracy, which outperforms all previously reported student error correction tools.

Keywords Program Repair · Naturalness · Deep Learning

Toufique Ahmed
Department of Computer Science, University of California, Davis, CA, USA
E-mail: tfahmed@ucdavis.edu

Premkumar Devanbu
Department of Computer Science, University of California, Davis, CA, USA
E-mail: ptdevanbu@ucdavis.edu

Vincent Hellendoorn
Department of Computer Science, University of California, Davis, CA, USA
E-mail: vhellendoorn@ucdavis.edu

1 Introduction

Most of the development tools require syntactically correct, well-typed code; the rest will usually be rejected in some fashion by static or dynamic checks within the tool. However, developers 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 and correct. Likewise, learners struggle with syntax (McCracken et al., 2001), and frequently make mistakes; the diagnosis and repair of syntax errors can be quite a challenge, especially for beginners. TAs and professors then have to expend valuable contact hours 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 (Wang et al., 2018; Gupta et al., 2017).

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 training, using only (mostly) correct code taken from Github. This code (mostly) compiles and is thus usually syntactically correct, and is well-typed. This code is easily processed by (*e.g.*) Eclipse JDT to yield massive volumes of matched tuples 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, while leaving the (desired) ASTs and types alone. 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 and 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 GitHub code repos to create massive amounts of "incorrect-fixed" training pairs, without relying on human annotation. We use this data to train high-capacity, efficient neural Transformer architectures, to leniently fix, parse and type fragmentary and incorrect code.
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 typer 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 and manual protocols, and demonstrate that we achieve good performance on the novel typing task, and improve upon a prior baseline for repairing student code, for longer fragments. We explicitly compare our tool with DeepFix (Gupta et al., 2017) and the tool proposed by Santos *et al.* (Santos et al., 2018).
5. We have released our data, to the extent permissible (for student data)[1], 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.

The remainder of the paper is organized as follows. Section 2 presents the motivation and background of our research. Section 3 discusses the technical approach of the paper. Section 4 presents the results of this research. Section 5 describes some prior research relevant to our work. Section 6 discusses the implications of the work and provides some future direction. Finally, Section 7 concludes the paper.

2 Motivation & Background

StackOverflow is now the preferred source of coding examples for developers. Given any coding quandary about core language features, or specific APIs, one can find answers, with illustrative code examples, on StackOverflow. However, the code examples are often fragmentary: just a few stand-alone lines of code, which are not complete, parseable units of Java code. If these fragments could be parsed into an AST[2] form, and also 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):

```
textView.setTypeface(textView.getTypeface(), Typeface.BOLD);
```

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 *"wrap-and-parse"* trick is simple and appealing, we estimate based on a manual examination of 200 randomly sampled fragments, that 28% of StackOverflow fragments[3] are not parseable due to various kinds of coding errors. Such fragments are quite common on StackOverflow, often missing delimiters, including ellipses, and missing declarations; the following example from Stackoverflow is fairly typical [4], and cannot be parsed by Eclipse JDT.

[1] It's possible for others to license the data, however, as did we.

[2] AST = Abstract Syntax Tree

[3] 28% is a point estimate. The 95% Wald confidence interval, on a binomial estimator with a sample size of 200, is 22-35%.

[4] https://stackoverflow.com/a/54596387

```
Optional<String> getIfExists() {
    ...
    return Optional.empty();
}
```

These fragments resist processing via the simple *"wrap-and-parse"* trick, and require more intelligent handling. Table 4 presents a few more StackOverflow fragments (along with this example) and the parse trees generated by our approach.

A more intelligent, lenient parsing approach could have benefits beyond StackOverflow. Student code, for example is rife with similar, syntactic errors; automatically fixing these could be very helpful for teachers and learners. In our experiment, we use the Blackbox dataset (Brown et al., 2014) which contains millions of examples of student submissions from around a million users. Fig 1 is an example from this dataset. Both Eclipse and IntelliJ IDEs fail to repair this example because none of the IDEs have any hand-crafted rule to solve this problem.

```
1  public class Modulus
2  {
3    public static void main(String[] args){
4
5        for(int i = 0; i < 12; i = i + 2){
6            System.out.println("i = " + i);
7            System.out.println("3 - i (" + (i) + ") = " (3 - i));
8        }
9
10   }
11 }
```

Fig. 1 Incorrect (verbatim) student code sample

Note the missing "+" on line 7 before "$(3-i)$". Simple syntax errors challenge and frustrate beginners (McCracken et al., 2001). A lenient parser pipeline could 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 "+" is needed. Our overarching research goal is a kind *lenient program analysis* which exploits powerful deep neural network models of code to enable IDEs to work more intelligently with malformed fragments by guessing their intended structure; in this paper, we focus on *lenient parsing and typing* specifically aimed at managing the vagaries of StackOverflow fragments and student code. StackOverflow fragments, thus rendering them a greater proportion of them more usable within an IDE.

3 Technical Approach

For the lenient parser, we use a pipeline with two learned deep-neural network (DNN) stages. The first DNN stage learns to repair errors in nested block structure ("BlockFix"), and the second stage learns repair and parse syntactically incorrect 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 corpora have 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 a completely data-driven DNN architecture (Vinyals et al., 2015) was shown to be remarkably effective at 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 could *learn to effectively parse* from aligned pairs of sentences and associated (serialized) parse trees.

To our knowledge, this learned parsing-as-translation approach has not been used for fragmented, noisy code, but it is *prima facie* well-suited. Unlike with NL, where parse trees (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, the fragments of code contained therein have regularities (thanks to the well-known naturalness (Hindle et al., 2012) 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 (Vaswani et al., 2017) 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 readily produce large volumes of training pairs of code + AST using compilers; however, the code must necessarily be correct and complete, to be compilable. We therefore have to artificially fragment and noise-up this code, to train our learned parser to robustly deal with such problems. Our training approach resembles that of Pradel *et al.* (Pradel and Sen, 2018). However, their goal was to *detect* (not fix) specific types of bugs (e.g., accidentally swapped function arguments, incorrect binary operators, and incorrect operands in binary operations) in *otherwise syntactically correct code*; our goal here is to leniently parse (thus, also repairing) and type syntactically malformed code.

Second, the vocabulary in code tends to be much larger (Hellendoorn and Devanbu, 2017; Karampatsis et al., 2020) than natural language, thanks to identifiers. Normally, larger vocabularies would present a challenge for learning to translate (Hellendoorn and Devanbu, 2017). However, for our purposes, 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 and 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 fragme
simple

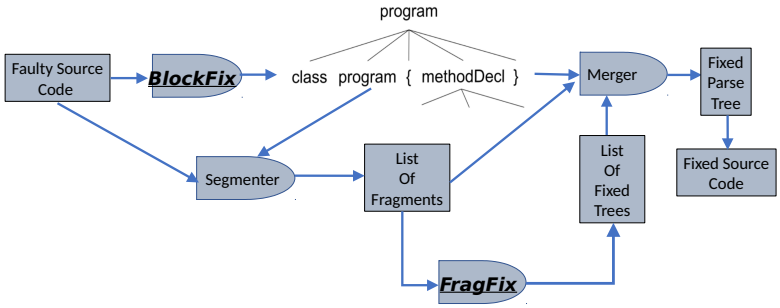


Fig. 2 Full pipeline. BLOCKFIX and FRAGFIX are learned transformer models. For the STACKOVERFLOW parsing task we stop with the "Fixed Parse Tree". For the Student code correction task we generate the fixed source code.

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 (Allamanis and Sutton, 2013).[5] 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 data, as described below, to make our parsing and typing models robust to fragmentation and noise. Thus our approach relies on *indirect supervision*, since the training data originates from a different setting than the tasks.

[5] Allamanis and Sutton define popularity as the number of forks plus the number of watchers.

3.3 Transfomers are all we need

The "Transformer" is a DNN architecture originally developed for language modeling and translation. It relies exclusively on attention (Vaswani et al., 2017) to model sequential dependency in language. Prior approaches to translation modeled this dependence using recurrent architectures, *viz.,* long short-term memory (Hochreiter and Schmidhuber, 1997), and gated recurrent units (Chung et al., 2014). These recurrent neural network (RNN) models require "back-propagation through time" (BPTT) to recursively propagate loss values over gradients, within the same recurrent units, so that long-distance dependencies could be captured from the training set. While this approach is quite effective, the serial nature of BPTT greatly inhibits parallelism during training and use. RNNs also suffer from memory-loss when dealing with longer dependencies. Bahdanau *et al.* introduced *attention*, wherein a fixed-length vector is used to identify and relate relevant part of the input to the target output (Bahdanau et al., 2014). This mechanism supplemented the recurrent structure and improved performance by enabling more direct calculation of dependencies.

Standard, basic transformers essentially take attention to the next level; they outperform all older models on various tasks.[6] They eliminate all recurrence, but multiply the attention mechanisms. We use transformers for all 3 of our tasks: lenient StackOverflow parsing and typing, and syntax error correction. As with older recurrent (RNN) models, like RNNs, Transformers use input embedding layers to convert discrete sequential input tokens into sequences of continuous vector embeddings, and softmax output layers to convert internal vector representations into output symbols. The output and input vocabularies are usually limited to control the input and output layer sizes. But the resemblance to RNN models ends here. Transformers primarily rely on layers of multi-headed attention, which can attend to multiple parts of the same sequence to help calculate a representation of the full sequence. These attention layers are interspersed with fully-connected feedforward layers that process the output from the attention layer. The non-linearities in the feedforward layer allow powerful and flexible combination of the elements upon which the incoming attention layer is focused. Both the encoder and decoder part of the translation model use many such levels, each consisting of pairs of stacked self-attention and fully-connected feedforward sub-levels. 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 allow 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 up 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. [7]

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

[6] See `https://ai.googleblog.com/2017/08/transformer-novel-neural-network.html`

[7] See `https://github.com/Lsdefine/attention-is-all-you-need-keras`

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 learns to model common-block structuring patterns, and repair nesting structure if necessary. The repaired structure 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.

```
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 }
```

Fig. 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 and 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 discernment to fix 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

Since BLOCKFIX has the sole task of modeling and repairing block nesting structure, it is trained with input-output pairs that just reflect this task. Consider the code below in figure 4.

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

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

Almost everything is removed from the input, except curly braces and keywords; we abstracted out all other identifiers, constants, expressions, and delimiters. 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,

```
public class TableListWriter extends HTMLListWriter{
    public TableListWriter(File outputDir){
        super("Current Tables", "currenttables.html", "tables", outputDir);
        if(ListClosing()) {
          WriteCloseMarkup()
          }
        else {
          CloseList();
        }
    }
}
```

Fig. 4 Example Java program for abstraction.

split randomly between open and close curly braces for this noising step.[8] Training on many 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 removed 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 and track where these fit into the abstracted input, 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 is trained with files whose abstracted versions are no more than 500 tokens; the dataset includes around 1.5M such files. The 500-token limit fits inside the Transformer's attention window, on a NVidia GeForce RTX 2080 Ti, without exceeding GPU memory; as explained below in subsection 3.5 this is adequate for our purposes. 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 layer, 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 (Vaswani et al., 2017). 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.* (Vaswani et al., 2017). 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 and repairing nesting structures, which helps the performance.

Sometimes it is not clear exactly where a close curly should be added. Consider the example in Table 1 from student error code dataset. Should the close "}" be added before,

[8] Anecdotally, 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.

Table 1 BLOCKFIX fixing nesting error

Version with Error

```
public class ImageBasics
{
  public static void main(String[] args)
  {
    APImage cv = new APImage(255,255);
    for(int x = 0; x<255; x++)
    {
      for(int y = 0; y<255; y++)
      {
        Pixel p = cv.getPixel(x,y);
        p.setRed(x);
      }
    canvas.draw();
  }
}
```

Actual Fixed Error

```
public class ImageBasics
{
  public static void main(String[] args)
  {
    APImage cv = new APImage(255,255);
    for(int x = 0; x<255; x++)
    {
      for(int y = 0; y<255; y++)
      {
        Pixel p = cv.getPixel(x,y);
        p.setRed(x);
      }
    }
    canvas.draw();
  }
}
```

or after the canvas.draw()? Both would lead to correct parse. In this case, BLOCKFIX learns from the training set that nested loops are usually closed all at the same time, and proposes this fix. Both Eclipse and IntelliJ IDEs recommend adding the close "}" after canvas.draw(), which is not the correct fix here.

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 fewer than 10,000 tokens and produced the ASTs using Eclipse JDT. This limit ensures that the abstracted versions of these files are small enough to be processed by BLOCKFIX; in practice, this limit works well. Most Stackoverflow fragments are much smaller than 10,000 tokens, and only 0.6% of the student programs exceed this limit. 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.* (Santos et al., 2018) find that, in BlackBox, most (57.4%) of syntax errors arise from single-token errors (extra, missing, or wrong tokens). Extra tokens accounted for about 23% 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 types of brackets ("`[{()}]`"), and the string separator "+". To gain a better understanding, we examined the errors from 200 randomly selected student programs. We found nine significant categories. The errors involving a specific category includes deletion, insertion or misplacement of that particular separator. Table 2 presents our findings. The other category includes cases with spaces between variable names, missing quotation marks, and some misplaced symbol which can not be covered by eight major classes. Of our 2M fragments, we mutated approximately

half. We sampled locations within fragments uniformly at random to inject these mutations. In the majority of cases, each fragment gets one mutation because our primary goal is to solve a single token error. However, to make the model robust, we sometimes injected two mutations (30% of the mutated fragments have two mutations).

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 corrupted inputs were paired with the original, correct AST, which indicated the location of the error, and it's repair, as illustrated below:

Code: `int 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 prioritize learning 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

Table 2 Categories of student code error

Category	Total count	In percentage (%)
Semicolon	73	36.5
Curly brace	39	19.5
Parenthesis	28	14.0
Arithmetic operator	20	10.0
Keyword	9	4.5
Comma	7	3.5
Missing datatype	6	3.0
Bracket	2	1.0
Others	16	8.0

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. The STACKOVERFLOW parsing is slightly different, and is explained afterwards. For student code: 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 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 $\mathscr{P}$
1 $abs \leftarrow FindBraces(P)$;
2 **if** *NotBalancedBraces(abs)* **then**
3 $\quad abs \leftarrow BlockFix(P)$;
4 **end**
5 $segs \leftarrow segment(abs, P)$;
6 $frags \leftarrow splitProgram(segs, P)$;
7 $AST \leftarrow initializeAST(abs)$;
8 **for** $frag \in frags$ $(in\ order)$ **do**
9 $\quad fragAST \leftarrow FragFix(frag)$;
10 $\quad AST \leftarrow Ins(fragAST, abs, segs, AST)$;
11 **end**
12 $\mathscr{P} \leftarrow GenerateCodeFrom(AST)$;

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 (Raychev et al., 2015) showed that it is possible to guess and type annotations for gradually typed languages such as Typescript. Hellendoorn *et al.* (Hellendoorn et al., 2018; Malik et al., 2019) 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 (Hellendoorn et al., 2018), 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 (Hellendoorn et al., 2018), 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:

```
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.* (Vaswani et al., 2017). 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 (Loshchilov and Hutter, 2016) 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. Table 3 presents the detailed evaluation plan of our research.

Table 3 Detailed evaluation plan

Tasks	Models	Dataset	Baseline	Evaluation Approach
Lenient Parsing	FRAGFIX	Synthetic	N/A	Quantative
	BLOCKFIX	Synthetic	N/A	Quantative
	BLOCKFIX + FRAGFIX	STACKOVERFLOW	Wrap & Parse	Quantative-manual
Lenient Typing	TYPEFIX	Synthetic	N/A	Quantative
		STACKOVERFLOW	N/A	Quantative-manual
Student Code Fixing	BLOCKFIX + FRAGFIX	Blackbox (Brown et al., 2014)	DeepFix (Gupta et al., 2017)	Quantative
			Santos *et al.* (Santos et al., 2018)	Quantative (reported from paper)

First, to evaluate lenient parsing of STACKOVERFLOW fragments, we used a combination of automated and manual methods. We have two different datasets for the evaluation. For BLOCKFIX and FRAGFIX, we use a synthetic dataset, which includes mutated (and untouched) fragments from GitHub; the "golden" parsed AST is produced by Eclipse, and is compared to our lenient parser. Note here, however, that the lenient parser's task is to process mutated *and* untouched fragments to yield an AST; this AST is being compared against the AST produced by Eclipse from the *non-mutated*, original, code. In this way, we evaluate the power of the lenient parse components to parse both untouched and synthetically mutated code fragments. We note here that BLOCKFIX and FRAGFIX phases are unique to our approach, and do not have any comparable baselines.

Second, for parsing the STACKOVERFLOW fragments, we used simple *wrap & parse* trick on STACKOVERFLOW fragments to baseline our model: wrap the code in a function skeleton, and process with Eclipse JDT parser. If the AST generated by our lenient parser matches precisely to the AST generated by wrap & parse, we considered them correct. For the STACKOVERFLOW fragments on which the wrap & parse trick failed, we had to check by hand.

Third, for the lenient typing task in STACKOVERFLOW, we used only a manual evaluation. We are not aware of any previous approach for finding the types in fragments of code, that would be suitable as a baseline. We have a synthetic dataset also for the evaluation of the model, as with parsing: the lenient typer is asked to type mutated GitHub fragments which are evaluated against types produced by the Eclipse JDT from the original, un-mutated code.

Finally, for the student code correction task, we used a fully automated evaluation. We explicitly compare our tool with DeepFix (Gupta et al., 2017) and the tool proposed by Santos *et al.* (Santos et al., 2018). The rationale and results are presented separately in sections below.

4.1 Performance of Lenient Parsing

As clarified in Section 3.5, the lenient parser comprises two models (i.e., FRAGFIX and BLOCKFIX). FRAGFIX was trained with 2M fragments collected from GitHub. After training, we first evaluate it on 50,000 test fragments, consisting of about 29,300 mutated and 20,700 untouched fragments. from GitHub. These test fragments originated from correct code that could be parsed automatically by Eclipse JDT to create ASTs as a "golden" benchmark. We ensured that the test data didn't have any fragments duplicated from the training data. With NVidia GeForce RTX 2080 Ti GPU, we could able to train one epoch in about two hours using the transformer-based model. For *mutated* held-out fragments, we found that 94.5% of the outputs from the trained lenient parser accurately (exactly) matched the

output from Eclipse JDT. For the untouched fragments, we achieved an accuracy of 95.7%. This supports the claim that the lenient parser is able to overcome fragmentation & mutation. 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. To evaluate our BLOCKFIX model, we also used examples from GitHub. In this case, however, since segmenting correct code is trivial, we only evaluated on incorrect code. We trained BLOCKFIX with 1.5M training examples and achieved around 76% accuracy on our test set, again using a transformer-based model.

Parsing performance on STACKOVERFLOW A key goal of lenient parsing is to correctly process malformed STACKOVERFLOW fragments. These, by definition, could not be automatically parsed and so required manual checking. Our evaluation of STACKOVERFLOW test samples is limited by required human effort. Still, we sought a a sufficiently large & representative sample to get a good estimate of the performance. We collected the STACKOVERFLOW fragments from the public Google BigQuery dataset. [9] 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 an automatic 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 designed to capture cases which allow easy, automated 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. Consider the following fragment. Though it's a field declaration, FRAGFIX predicts it as a variable declaration.

```
List <Class<?>> defaultGroupSequence = new ArrayList <Class<?>> ( ) ;
```

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` *`classname`* `{ ...}`) or method (`void` *`methodname`* `() { ...}`) 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.

[9] See `https://cloud.google.com/bigquery/public-data/`

Table 4 StackOverflow fragments parseable by our approach but not by JDT

Code Fragments	By BlockFix + FragFix	Comments
`Optional<String> getIfExists() { ... return Optional.empty(); }`	(MethodDeclaration ... (Block ... ExtraPunctuation(s) * (ReturnStatement ...)...))	Unwanted ellipses in the fragment
`case 3: if (price > 75 ) { totalPrice = price; } else { totalprice = 5.95 + price ; } break;`	(SwitchCase ... (IfStatement ... (InfixExpression ... (Block ...) (BreakStatement)	Missing initial part of switch statement
`str = str.replaceAll("0+$", "")`	(ExpressionStatement ... (MissingSemicolon))	Syntactical error because of missing semicolon
`getClass().getClassLoader() .getResource("/ resources")))`	(ExpressionStatemen ... (ExtraPuntuation(s) * ... (MissingSemicolon))	Syntactical error because of missing semicolon and extra parentheses
`public BankAccount(double b , String n) { double balance = b;`	(MethodDeclaration ...(MissingCloseCurly))	Incompleteness of the block

Of the 123 JDT-parseable fragments, the lenient parser got 90 *correct*s, no *autofixed*s, 27 *partial*s, 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, the lenient parser yielded 30 *correct*, 6 *autofix*, 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 *fully automated* hybrid approach allows for parsing a total of 89.3%, (123+36 = 159/178) of fragments in our sample (Wald confidence interval 85-94%). A binomial test of difference of proportions yields a p-value < 0.00001 (n=178, "heads" = 159 *vs.* 123) for the null hypothesis that the observed difference (between the combination approach and simple wrapping+JDT parsing) is due to random sampling error. Table 4 presents some StackOverflow fragments along with the ASTs that are parseable by BlockFix + FragFix but not by Eclipse JDT. Note that we present a concise form of the ASTs in the table. The complete ASTs are similar to the example presented in Section 3.5.

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. We achieved 95% top-1 and 99% top-5 accuracy using transformer-based approach. For the top 1,000 most frequently-used types in our data, the top-1 and the top-5 accuracy are 97% 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* (Qiu et al., 2016)) Java classes:

1. `java.lang.String`
2. `java.lang.Override`
3. `java.util.List`
4. `java.lang.Exception`
5. `java.lang.Object`

b) *Core Java types* consisting of any fragments tagged with only "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 5 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. 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).

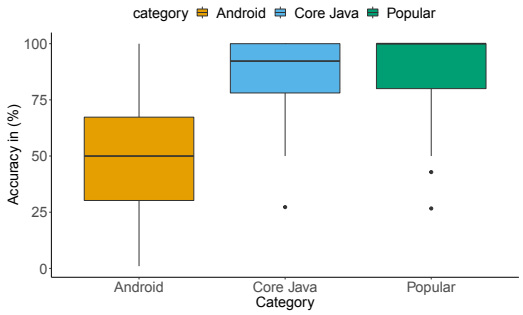


Fig. 5 Performance of DNN typer on different categories of STACKOVERFLOW fragments

4.3 Evaluation of Student Code Fixing

To evaluate the performance of the repair tool, we need realistic student programs with syntax errors, along with human-produced fixed versions. We used the dataset used from the Blackbox (Brown et al., 2014) repository, as used in Santos *et al.* (Santos et al., 2018); their work makes for a good baseline because this dataset is both very large and diverse. This Blackbox repository collects students' coding activity directly from the BlueJ Java IDE, which is designed to help beginners learn Java in programming classes (Kölling et al., 2003). After obtaining permissions, we evaluated our tool on this dataset. The dataset contains about 1.7M pairs of incorrect versions and the corresponding fixes. Santos *et al.* found that many syntax errors (57%, or *circa* 1M pairs) can be corrected by a single token edit; they report their tool's performance only on files with just a single token error. They use the mean reciprocal rank (MRR) as a performance metric, which tracks the average of the inverse (reciprocal) of the rank of the correct solution found in an ordered list of solutions (i.e., repairs). We report both MRR and top-1 accuracy, for comparison.

Our general pipline for fixing student programs was described on page 12. If the final FRAGFIX stage signals missing/extra tokens, we use it to sample upto 5 most likely ASTs, to measure the MRR of the correct answer. For each AST, we make the appropriate repairs, and check if any of those exactly match the correct repair given in the dataset. If so, the reciprocal of the rank of the correct answer is recorded; otherwise, we record zero. We also record the proportion of top-1 correct answers.

Performance of Student Code Fixer on Blackbox Dataset To get a performance estimate with tight bounds, we chose a very large random sample of 200,000 files out of the *circa* 1M files with a single syntactical error. Our models achieve a 56.4% top-1 accuracy rate and 0.58

MRR for the true fixes which is substantially higher than the reported MRR (0.46) by Santos *et al.* (Santos et al., 2018). Most of the time, the correct fix appears at the first rank; if not, we rarely see the fix in the top 5.

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 (Santos et al., 2018), 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* (Santos et al., 2018), Table 1), of which we can fix 56.4% perfectly (top-1 correction), yielding an estimated top-1 fix rate for all files with syntax errors in the Blackbox of about 32%. 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 35% 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 3,253 (66.25%) 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 0.58 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 6 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 63% accuracy for files with less than 300 tokens, 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 58%, which compares favourably with other approaches.

To observe the performance of BLOCKFIX with increased file length, we prepared ten buckets with different ranges of token counts. Each bucket consists of 10K examples. Table 5 shows that the accuracy of BLOCKFIX significantly declines with file length. We also report the MRR if we do not apply BLOCKFIX. The MRR does not change much with increased file length. Therefore, we can infer that our model works consistently on all the errors except the one due to imbalanced braces.

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 7. Note that these are scatter plots that additionally signal the prevalence of datapoint buckets with their color gradient.

Table 5 Performance of BLOCKFIX vs. file length

Token Range	MRR (overall)	Block Error	Solved by BLOCKFIX	Accuracy (in %) of BLOCKFIX	MRR (without BLOCKFIX)
1-100	0.66	2426	2124	87.55	0.45
101-200	0.62	2403	1787	74.36	0.44
201-300	0.56	2491	1536	61.66	0.41
301-400	0.52	2417	1200	49.64	0.40
401-500	0.47	2492	1024	41.09	0.37
501-600	0.44	2448	737	30.10	0.36
601-700	0.39	2343	519	22.15	0.34
701-800	0.36	2178	359	16.48	0.32
801-900	0.33	2362	284	12.02	0.31
901-1000	0.31	2183	149	6.08	0.29

Note: * we have 9,130 examples for 701-800 tokens and 9,052 for 901-1000 tokens.

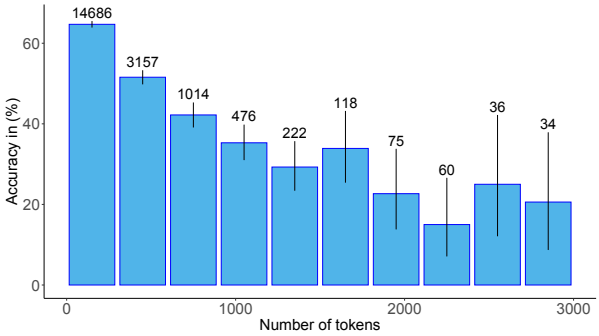
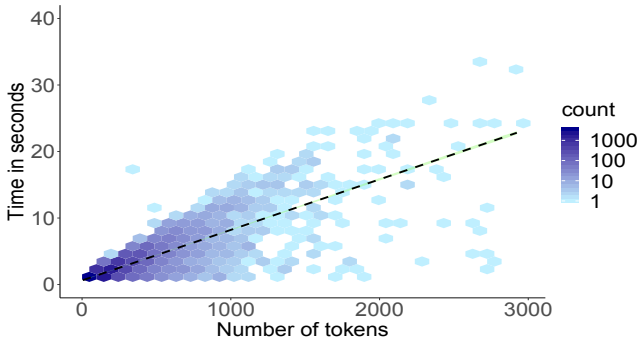


Fig. 6 Performance of code correct with increasing number of tokens in the file with 95% confidence interval

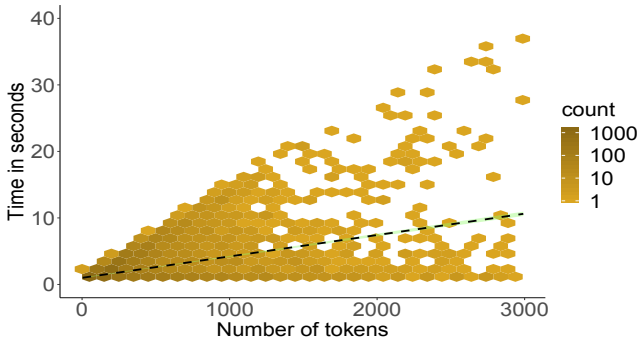
Immediately evident is the flatter slope for the failing cases, with many failing quickly: these usually fail earlier in our pipeline—either BLOCKFIX fails to properly balance and nest the input source code, or there other errors that inhibit fragmenting of the code, so we abort before getting to FRAGFIX. Figure 7 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 time reported in the Blackbox dataset (gathered from actual human-generated fixes, see Table 1 (Brown and Altadmri, 2017)), suggesting that we could provide timely help to students quite often. In all, we process 95% of files in under 10 seconds.

Comparison with DeepFix (Gupta et al., 2017) DeepFix, which uses an RNN seq2seq (with a single attention head) translation model, also works on student programs and repairs syntactic errors in C with 27% top-1 accuracy (Gupta et al., 2017). The programs in Deepfix's dataset range in size from 100-400 tokens, making them substantially much smaller than those in Blackbox.

Typically, DeepFix considers an erroneous program of a few hundred tokens; precisely predicting the (fixed) target sequence this long is very challenging for neural networks. Natural language translation (which was the original intended application of seq2seq neural models) rarely need to handle such long sequences, since they work a sentence at a time. To combat this problem, they encode the problem representation with line number. That means



(a) Fixable



(b) Non-Fixable

Fig. 7 Processing time vs. number of tokens in the file

instead of generating all the program tokens, their model outputs the defective statement with the line number, which is an excellent idea of representing the error in the program. However, there is no straight forward way to directly compare our work with DeepFix's current learned model, because of 3 reasons.

1. Deepfix is trained on C programs, which has a different syntax. Therefore, the reported accuracy is not directly comparable. Therefore, we trained the (publicly available) DeepFix implementation[10] on our Java dataset.
2. In DeepFix, the authors applied mutations to generate training examples because of limited programs from 93 different problems. They introduced five mutations on each program and applied five-fold cross-validation. In each fold, they have an average of 200K mutations on their training set. They iteratively resolve the errors from the program one at a time. Since we have a 200K real student program with a single error, we train DeepFix with real data instead of mutated ones.
3. As mentioned earlier, the length of the programs ranges from 100-400 tokens for DeepFix. Since DeepFix is trained on all the tokens of a program, there must be some upper

[10] See `https://bitbucket.org/iiscseal/deepfix/src/master/`

limit on token counts. This limitation is fundamental to the structure of all state-of-the-art machine translation models. Moreover, the training typically gets harder for longer sequences. There are a significant amount of Java programs with more than 400 tokens. It would be challenging for a model trained with limited tokens and explicit line number to repair the error in longer sequences. Therefore, we trained two DeepFix models: DeepFix Short (DF Short) and DeepFix Long (DF Long) to compare with our one. DF Shorts is trained on examples up to 400 tokens long, and DF Long is trained with examples up to 800 tokens long. We also prepared ten buckets of programs with token counts upto 1000 tokens for validation. We ensured that none of the examples from the test set is present in the training set. Finally, we evaluate three models (BLOCKFIX + FRAGFIX, DF Short, and DF Long) with the test set. Note that we applied these models on files with single syntactical error.

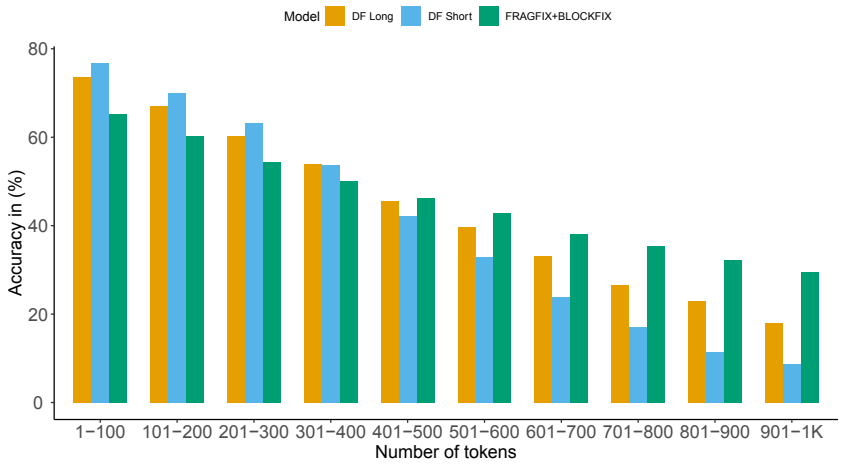


Fig. 8 Performance of three models on programs with different ranges of token-count

Figure 8 shows that DF Short and DF Long perform a bit better than our model (BLOCKFIX + FRAGFIX) until 400 tokens. After 400 tokens, our model surpasses the other two models and then gradually becomes dominant. The accuracy of all the model decrease with increased file length. DF Short worked extremely well with the shorter programs (around 76% top-1 accuracy for 1-100 tokens) but fails poorly with longer ones (8.8% for 901-1,000 tokens). It is quite obvious that if the error occurs after 400 tokens, the corresponding line number may not be in the output vocabulary. Without the line number, it is not possible the locate the error correctly. DF Long performs almost similar to DF Short; however, it performs well after 400 tokens and achieves 17.4% top-1 accuracy for 901-1,000 tokens. We can conclude that DeepFix models will fail to resolve minor or prevalent errors if the error located outside the token-range of training data. Besides, locating an error in the longer program is inherently difficult than to locate in shorter programs.

Though our model is not the best one for shorter programs (around 65% top-1 accuracy for 1-100 tokens), it achieves higher accuracy on the longer ones (30% top-1 accuracy for 901-1,000 tokens). We applied our model on even longer sequences (10,000 examples of

1,000-3,000 tokens) and we still achieves 26% top-1 accuracy. As discussed earlier, in our approach, we divide the program into smaller fragments and try to repair each error locally. Therefore, we could solve more trivial errors in the longer program compared to other models, and even for those errors, there is no need to retrain FRAGFIX or BLOCKFIX. However, there are several reasons for getting lower accuracies in the shorter programs. We generate AST on the FRAGFIX and introduced about 20 common mutations. Moreover, because of segmentation, we lose some information (i.e., the position of the token in the real program). DeepFix model performs better at finding misspelled keywords, and it extracts more than 20 mutations from the training data. On the other hand, our model does not work with misspelled keywords. Since we do not have positional information, it is not very easy for our model to work on this. Consider the following statement with a syntactic error from the student program.

```
imkport slp.core.util.Pair;
```

Let us consider "imkport" is not present in the input vocabulary of both FRAGFIX and DeepFix model. Therefore, both models will encode it with "<unk>". Now, FRAGFIX does not have any positional information; it assumes two identifiers (all "<unk>" are identifiers) are placed side by side. In that case, our model may try to separate two identifiers by placing a period between two identifiers. Moreover, FRAGFIX fails to identify the root node of the AST: "ImportDeclaration".

```
imkport. slp.core.util.Pair;
```

On the other hand, DeepFix can fix this. In most of the cases, the first few lines of java programs start with keyword "import". DeepFix quickly learns this pattern using the line number.

```
line-number1 import slp.core.util.Pair;
```

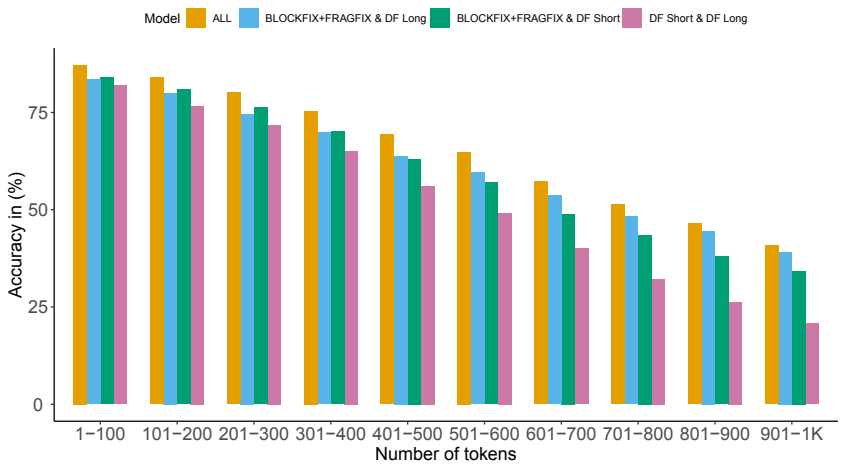


Fig. 9 Performance of blended models on programs with different ranges of token-count

DeepFix (Gupta et al., 2017) is also reasonably language agnostic; on the other hand, while it can repair code, it's not a "lenient" parser that can parse fragmentary, and wrong

code. We can produce a tree for downstream use, e.g., from a fragment of case statement or an statement with ellipses ("...").

Blending BLOCKFIX + FRAGFIX *and DeepFix (Gupta et al., 2017)* Finally, we explore a blended strategy. Do DeepFix and BLOCKFIX + FRAGFIX are repairing the same files? Since we have a fixed validation set, we try to blend three models to see how much top-1 accuracies we could achieve. We apply a simple heuristics to report the blending accuracy. If an error is corrected by at least one of the candidate models, we would consider that as a success. On the contrary, if all the candidate models fail to correct one specific error, we marked that as failure by the blending model.

Figure 9 presents the accuracies if we blend the models. If we consider all the three models, the top-1 accuracies range from 87%-40% for different token ranges. On the full dataset, we can achieve 77% overall top-1 accuracy, which outperforms the individual performance of all the models. For any given erroneous code, we get 3 guesses, one from each model; using the compiler we can pick a good fix. If the compiler rejects the result from a certain model, we can try another one. We also note that mixing DF Short and DF Long (prior work) doesn't provide the best results. Therefore, we can infer that there are a significant amount of errors (10% of Blackbox dataset) are solved by our model, which the DeepFix approach cannot handle.

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 (Van Deursen and Kuipers, 1999; Moonen, 2001). Synytskyy (Synytskyy et al., 2003) 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) (Bacchelli et al., 2017). 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 (Rigby and Robillard, 2013). 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 (Bacchelli et al., 2017), §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 (Rountev et al., 1999; Dagenais and Hendren, 2008). 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[11] 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

[11] `http://www.sable.mcgill.ca/ppa/`

web (Holmes et al., 2005; Thummalapenta and Xie, 2007; Nasehi et al., 2012; Ponzanelli et al., 2014). 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 (Dagenais and Hendren, 2008). Alexandru *et al.* propose that DNN is not very good at tokenizing source code, but it is highly capable of recognizing token types and their relative locations in a parse tree (Alexandru et al., 2017). Raychev *et al.* applied statistical inference model for inferring types for JavaScript (Raychev et al., 2015). Hellendoorn *et al.* (Hellendoorn et al., 2018) 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) (Malik et al., 2019). 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 Mesbah *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 (Mesbah et al., 2019), which we do not.

Gupta *et al.* applied reinforcement learning to a very similar dataset (Gupta et al., 2018), 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) (Bhatia and Singh, 2016). 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 (Bhatia and Singh, 2016) 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 (Santos et al., 2018); 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 (Tufano et al., 2019; Chen et al., 2019; Li et al., 2020; Ding et al., 2020; Lutellier et al., 2020), or to find relevant repairs for patching (White et al., 2019). Translation models that use tree to tree (rather than seq2seq) have also been proposed (Chakraborty et al., 2018a,b). These approaches are mostly not aimed at syntax errors, but rather at semantic errors exposed by failing tests. However, Deepfix (Gupta et al., 2018) is relevant to our approach, and we have done an extensive comparison above.

All the above mentioned Automatic Program Repair (APR) tasks have a fault localization step. APR is inherently a harder problem because it is often unclear where the semantic error is. The success of these tools depends a lot on the fault-localizer. Different APR techniques use different fault-localizers. Most of the papers use the perfect fault-localizer to have a fair comparison. We can infer that the reported performance will not be the same if the authors would use some real-world, imperfect fault-localizer. For our task, we do not need any fault-localizer. BLOCKFIX & FRAGFIX are together capable of localizing and fixing the error.

We note that many of these "semantic" models are not applicable to our setting. For one, most APR approaches, including GenProg (Le Goues et al., 2011), Prophet (Long and Rinard, 2016), as well as tree to tree neural models (Chakraborty et al., 2018a,b), assume that

the "old" (buggy) version of the code is at least syntactically valid and can be parsed/compiled/executed, and even subjected to static analysis. Needless to say, this is inapplicable to our setting. Furthermore, most deep learning-based work are specifically trained to repair semantic errors (Tufano et al., 2019; Chen et al., 2019; Li et al., 2020; Ding et al., 2020; Lutellier et al., 2020), or to find relevant repairs for patching (White et al., 2019). Without training on syntactic errors, such models are highly unlikely to be useful on the latter. DeepFix (Gupta et al., 2018), however, does target syntactic errors and is thus precisely relevant to our approach, and correspondingly we compared extensively with that tool.

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 56% 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 at least 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.* (Malik et al., 2019) quite a bit of type information is carried in the identifier names; so we believe that approaches like (Babii et al., 2019; Karampatsis et al., 2020), 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 (Brown et al., 2014).

References

Alexandru CV, Panichella S, Gall HC (2017) Replicating parser behavior using neural machine translation. In: 2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC), IEEE, pp 316–319

Allamanis M, Sutton C (2013) Mining Source Code Repositories at Massive Scale using Language Modeling. In: The 10th Working Conference on Mining Software Repositories, IEEE, pp 207–216

Babii H, Janes A, Robbes R (2019) Modeling vocabulary for big code machine learning. arXiv preprint arXiv:190401873

Bacchelli A, Mocci A, Cleve A, Lanza M (2017) Mining structured data in natural language artifacts with island parsing. Science of Computer Programming 150:31–55

Bahdanau D, Cho K, Bengio Y (2014) Neural machine translation by jointly learning to align and translate. arXiv preprint arXiv:14090473

Bhatia S, Singh R (2016) Automated correction for syntax errors in programming assignments using recurrent neural networks. arXiv preprint arXiv:160306129

Brown NC, Altadmri A (2017) Novice java programming mistakes: Large-scale data vs. educator beliefs. ACM Transactions on Computing Education (TOCE) 17(2):7

Brown NCC, Kölling M, McCall D, Utting I (2014) Blackbox: a large scale repository of novice programmers' activity. In: Proceedings of the 45th ACM technical symposium on Computer science education, ACM, pp 223–228

Chakraborty S, Allamanis M, Ray B (2018a) Tree2tree neural translation model for learning source code changes. arXiv preprint arXiv:181000314

Chakraborty S, Allamanis M, Ray B (2018b) Tree2tree neural translation model for learning source code changes. CoRR abs/1810.00314, URL `http://arxiv.org/abs/1810.00314`, `1810.00314`

Chen Z, Kommrusch SJ, Tufano M, Pouchet LN, Poshyvanyk D, Monperrus M (2019) Sequencer: Sequence-to-sequence learning for end-to-end program repair. IEEE Transactions on Software Engineering

Chung J, Gulcehre C, Cho K, Bengio Y (2014) Empirical evaluation of gated recurrent neural networks on sequence modeling. arXiv preprint arXiv:14123555

Dagenais B, Hendren L (2008) Enabling static analysis for partial java programs. In: ACM Sigplan Notices, ACM, vol 43, pp 313–328

Ding Y, Ray B, Devanbu P, Hellendoorn VJ (2020) Patching as translation: the data and the metaphor. 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)

Gupta R, Pal S, Kanade A, Shevade S (2017) Deepfix: Fixing common c language errors by deep learning. In: Thirty-First AAAI Conference on Artificial Intelligence

Gupta R, Kanade A, Shevade S (2018) Deep reinforcement learning for programming language correction. arXiv preprint arXiv:180110467

Hellendoorn VJ, Devanbu P (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, pp 763–773

Hellendoorn VJ, Bird C, Barr ET, Allamanis M (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, pp 152–162

Hindle A, Barr ET, Su Z, Gabel M, Devanbu P (2012) On the naturalness of software. In: 2012 34th International Conference on Software Engineering (ICSE), IEEE, pp 837–847

Hochreiter S, Schmidhuber J (1997) Long short-term memory. Neural computation 9(8):1735–1780

Holmes R, Walker RJ, Murphy GC (2005) Strathcona example recommendation tool. In: ACM SIGSOFT Software Engineering Notes, ACM, vol 30, pp 237–240

Karampatsis RM, Babii H, Robbes R, Sutton C, Janes A (2020) Big code != big vocabulary: Open-vocabulary models for source code. In: International Conference on Software Engineering (ICSE)

Kölling M, Quig B, Patterson A, Rosenberg J (2003) The bluej system and its pedagogy. Computer Science Education 13(4):249–268

Le Goues C, Nguyen T, Forrest S, Weimer W (2011) Genprog: A generic method for automatic software repair. Ieee transactions on software engineering 38(1):54–72

Li Y, Wang S, Nguyen TN (2020) Dlfix: Context-based code transformation learning for automated program repair. In: 2020 42th International Conference on Software Engineering (ICSE)

Long F, Rinard M (2016) Automatic patch generation by learning correct code. In: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp 298–312

Loshchilov I, Hutter F (2016) Sgdr: Stochastic gradient descent with warm restarts. arXiv preprint arXiv:160803983

Lutellier T, Pham HV, Pang L, Li Y, Wei M, Tan L (2020) Coconut: combining context-aware neural translation models using ensemble for program repair. In: Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, pp 101–114

Malik RS, Patra J, Pradel M (2019) Nl2type: inferring javascript function types from natural language information. In: Proceedings of the 41st International Conference on Software Engineering, IEEE Press, pp 304–315

McCracken M, Almstrum V, Diaz D, Guzdial M, Hagan D, Kolikant YBD, Laxer C, Thomas L, Utting I, Wilusz T (2001) A multi-national, multi-institutional study of assessment of programming skills of first-year cs students. In: Working group reports from ITiCSE on Innovation and technology in computer science education, ACM, pp 125–180

Mesbah A, Rice A, Johnston E, Glorioso N, Aftandilian E (2019) Deepdelta: learning to repair compilation errors

Moonen L (2001) Generating robust parsers using island grammars. In: Proceedings Eighth Working Conference on Reverse Engineering, IEEE, pp 13–22

Nasehi SM, Sillito J, Maurer F, Burns C (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, pp 25–34

Ponzanelli L, Bavota G, Di Penta M, Oliveto R, Lanza M (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, pp 102–111

Pradel M, Sen K (2018) Deepbugs: A learning approach to name-based bug detection. Proceedings of the ACM on Programming Languages 2(OOPSLA):1–25

Qiu D, Li B, Leung H (2016) Understanding the api usage in java. Information and software technology 73:81–100

Raychev V, Vechev M, Krause A (2015) Predicting program properties from big code. In: ACM SIGPLAN Notices, ACM, vol 50, pp 111–124

Rigby PC, Robillard MP (2013) Discovering essential code elements in informal documentation. In: 2013 35th International Conference on Software Engineering (ICSE), IEEE, pp 832–841

Rountev A, Ryder BG, Landi W (1999) Data-flow analysis of program fragments. In: Software Engineering?ESEC/FSE?99, Springer, pp 235–252

Santos EA, Campbell JC, Patel D, Hindle A, Amaral JN (2018) Syntax and sensibility: Using language models to detect and correct syntax errors. In: 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER), IEEE, pp 311–322

Synytskyy N, Cordy JR, Dean TR (2003) Robust multilingual parsing using island grammars. In: Proceedings of the 2003 conference of the Centre for Advanced Studies on Collaborative research, IBM Press, pp 266–278

Thummalapenta S, Xie T (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, pp 204–213

Tufano M, Pantiuchina J, Watson C, Bavota G, Poshyvanyk D (2019) On learning meaningful code changes via neural machine translation. In: Proceedings of the 41st International

Conference on Software Engineering, IEEE Press, pp 25–36
Van Deursen A, Kuipers T (1999) Building documentation generators. In: Proceedings IEEE International Conference on Software Maintenance-1999 (ICSM'99).'Software Maintenance for Business Change'(Cat. No. 99CB36360), IEEE, pp 40–49
Vaswani A, Shazeer N, Parmar N, Uszkoreit J, Jones L, Gomez AN, Kaiser Ł, Polosukhin I (2017) Attention is all you need. In: Advances in neural information processing systems, pp 5998–6008
Vinyals O, Kaiser Ł, Koo T, Petrov S, Sutskever I, Hinton G (2015) Grammar as a foreign language. In: Advances in neural information processing systems, pp 2773–2781
Wang K, Singh R, Su Z (2018) Search, align, and repair: Data-driven feedback generation for introductory programming exercises. SIGPLAN Not 53(4):481–495, DOI 10.1145/3296979.3192384, URL http://doi.acm.org/10.1145/3296979.3192384
White M, Tufano M, Martinez M, Monperrus M, Poshyvanyk D (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, pp 479–490