Syntax and *Sensibility*: Using language models to detect and correct syntax errors

Eddie Antonio Santos, Joshua Charles Campbell, Dhvani Patel, Abram Hindle, and José Nelson Amaral
Department of Computing Science
University of Alberta
Edmonton, Alberta, Canada
{easantos,joshua2,dhvani,hindle1,jamaral}@ualberta.ca

***Abstract*—Syntax errors are made by novice and experienced programmers alike; however, novice programmers lack the years of experience that help them quickly resolve these frustrating errors. Standard LR parsers are of little help, typically resolving syntax errors and their precise location poorly. We propose a methodology that locates where syntax errors occur, and suggests possible changes to the token stream that can fix the error identified. This methodology finds syntax errors by using language models trained on correct source code to find tokens that seem out of place. Fixes are synthesized by consulting the language models to determine what tokens are more likely at the estimated error location. We compare n-gram and LSTM (long short-term memory) language models for this task, each trained on a large corpus of Java code collected from GitHub. Unlike prior work, our methodology does not rely that the problem source code comes from the same domain as the training data. We evaluated against a repository of real student mistakes. Our tools are able to find a syntactically-valid fix within its top-2 suggestions, often producing the exact fix that the student used to resolve the error. The results show that this tool and methodology can locate and suggest corrections for syntax errors. Our methodology is of practical use to all programmers, but will be especially useful to novices frustrated with incomprehensible syntax errors.**

I. Introduction

Computer program source code is often expressed in plain text files. Plain text is a simple, flexible medium that has been preferred by programmers and their tools for decades. Yet, plain text can be a major hurdle for novices learning how to code [1, 2, 3]. Not only do novices have to learn the semantics of a programming language, but they also have to learn how to place arcane symbols in the right order for the computer to understand their intent. The positioning of symbols in just the right way is called *syntax*, and sometimes humans—especially novices—get it wrong [1].

The tools meant for interpreting the human-written source code, *parsers*, are often made such that they excel in understanding well-structured input; however, if they are given an input with so much as one mistake, they can fail catastrophically. What's worse, the parser may come up with a misleading conclusion as to where the actual error is. Consider the Java source code in Listing 1. A single *token*—an open brace ({) at the end of line 3—in the input is different from that of the correct file that the programmer intended to write. Give this input to a Java 8 compiler such as OpenJDK's `javac` [4], and it reports that there is an error with the source file, but it overwhelms the user with misleading error messages to identify the location of the mistake made by the programmer.

Listing 1: Syntactically invalid Java code. An open brace ({) is missing at the end of line 3.

```
1 public class A {
2     public static void main(String args[]) {
3         if (args.length < 2)
4             System.out.println("Not enough args!");
5             System.exit(1);
6         }
7         System.out.println("Hello, world!");
8     }
9 }
```

```
A.java:7: error: <identifier> expected
        System.out.println("Hello, world!");
                           ^
A.java:7: error: illegal start of type
        System.out.println("Hello, world!");
                            ^
A.java:9: error: class, interface, or enum expected
}
^
3 errors
```

Imagine a novice programmer writing this simple program for the first time, being greeted with three error messages, all of which include strange jargon such as `error: illegal start of type`. The compiler identified the problem as being *at least* on line 7 when the mistake is four lines up, on line 3. However, an experienced programmer could look at the source code, ponder, and exclaim: "Ah! There is a missing open brace ({) at the end of line 3!" Such mistakes involving unbalanced braces are the most frequent errors among new programmers [5], yet the compiler offers little help in resolving them.

In this paper, we present *Sensibility*, which finds and fixes **single token syntax errors**. We address the following problem:

> Given a source code file with a syntax error, how can one accurately pinpoint its location and produce a single token suggestion that will fix it?

We compare the use of n-gram models and long short-term memory neural network (LSTM) models for modelling source code for the purposes of correcting syntax errors. All models

were trained on a large corpus of hand-written Java source code collected from GitHub. Whereas prior works offer solutions that are limited to fixing syntax errors within the same domain as the training data—such as only fixing errors within the same source repository [6], or within the same introductory programming assignment [7, 8]—*Sensibility* imposes no such restriction. As such, we evaluated against a corpus of real syntax errors collected from the Blackbox repository of novice programmers' activity [9].

In this paper, we focus on correcting syntax errors at the token level. We do not consider *semantic errors* that can occur given a valid parse tree such as type mismatches, and—in our abstract models (Section III-B)—misspelled variable names. These errors are already handled by other tools given a source file with valid syntax. For example, the Clang C++ compiler [10] can detect misspelled variable names and—since it has a valid parse tree—Clang can suggest which variable in scope may be intended. As such, we focus our attention in this paper to errors that occur *before* a compiler can reach this stage—namely, errors that prevent a valid parse of a source code file. In particular, we consider syntax errors that are non-trivial to detect using simple handwritten rules, as used in some parsers [11, 12].

Our contributions include:

- Showing that language models can successfully locate and fix syntax errors in human-written code without parsing.
- Comparing three different language models including two n-gram models and one deep neural network.
- Evaluating how all three models perform on a corpus of real-world syntax errors with known fixes provided by students.

II. Prior work

Prior work relevant to this research addresses repositories of source code and repositories of mistakes, past attempts at locating and fixing syntax errors in and out of the parser, methods for preventing syntax errors, deep learning, and applying deep learning to syntax error locating and fixing.

a) Data-sources and repositories: are needed to train models and evaluate the effectiveness of techniques on syntax errors. Brown et al. [9, 13] created the Blackbox repository of programmers' activity collected from the BlueJ Java Integrated Development Environment (IDE) [14], which is used internationally in introductory computer science classes. Upon installation, BlueJ asks the user whether they consent to anonymized data collection of editor events. The dataset—which is continually updated—enables a wealth of analyses concerning how novices program Java. Our empirical evaluation, described in Section VI, is one such analysis using the Blackbox data.

b) Syntax errors: are errors whereby the developer wrote code that cannot be recognized by the language rules of the parser. It can be as simple as missing a semi-colon at the end of a statement. The long history of work on syntax error messages is often motivated by the need to better serve novice programmers [1, 2, 3, 15, 16, 17, 18, 19, 20]. Denny et al. [21] categorized common syntax errors that novices make by the error messages the compiler generates and how long it takes for the programmer to fix them. This research shows that novices and advanced programmers alike struggle with syntax errors and their accompanying error messages. Dy and Rodrigo [22] developed a system that detects compiler error messages that do not indicate the actual fault, which they name "non-literal error messages", and assists students in solving them. Nienaltowski et al. [23] found that more detailed compiler error messages do not necessarily help students avoid being confused by error messages. They also found that students presented with error messages only including the file, line, and a short message did better at identifying the actual error in the code more often for some types of errors. Marceau et al. [24] developed a rubric for grading the error messages produced by compilers. Becker's dissertation [25] attempted to enhance compiler error messages in order to improve student performance. Barik et al. [26] studied how developers visualize compilation errors. They motivate their research with an example of a compiler misreporting the location of a fault. Pritchard [27] shows that the most common type of errors novices make in Python are syntax errors. Brown et al. [5] investigated the opinions of educators regarding what they believe are the most common Java programming mistakes made by novices, and contrasted it with mistakes mined from the Blackbox dataset. They found that educators share a weak consensus of what errors are most frequent, and furthermore show that across $14{,}235{,}239$ compilation events, educators' opinion demonstrates a low level of agreement against the actual data mined. The most common error witnessed was unbalanced parenthesis and brackets, which was ranked the eleventh most common on average by educators. The other most common errors were semantic and type errors.

Earlier research attempted to tackle errors at the parsing stage. In 1972, Aho [28] introduced an algorithm to attempt to repair parse failures by minimizing the number of errors that were generated. In 1976, Thompson [29] provided a theoretical basis for error-correcting probabilistic compiler parsers. He also criticized the lack of probabilistic error correction in then-modern compilers. However, this trend continues to this day.

Parr et al. [30] discusses the strategy used in ANTLR, a popular LL(*) parser-generator. The parser attempts to repair the code when it encounters an error using the context around the error so it can continue parsing. This strategy allows ANTLR parsers to detect multiple problems instead of stopping on the first error. Jeffery [11] created Merr, an extension of the Bison parser generator, which allows the grammar writer to provide examples of expected syntax errors that may occur in practice, accompanied with a custom error message. In contrast to Merr, our work does not require any hand-written rules in order to provide a suggestion for a syntax error, and is not reliant on the parser state, which may be oblivious to the actual location of the syntax error.

Recent research has applied language models to syntax error detection and correction. Campbell et al. [6] created UnnaturalCode which leverages n-gram language models to locate syntax errors in Java source code. UnnaturalCode wraps the invocation of the Java compiler. Every time a program

is syntactically-valid, it augments the existing n-gram model. When a syntax error is detected, UnnaturalCode calculates the *entropy* of each token. The token sequence with the highest entropy with respect to the language model would be the likely location of the true syntax error, in contrast to the location where the parser may report the syntax error. Using code-mutation evaluation, the authors were able to find that a combination of UnnaturalCode's reported error location and the Java compiler's reported error locations would yield the best mean-reciprocal rank for the true error location. Using a conceptually similar technique to UnnaturalCode, our work detects the location of syntax errors; unlike UnnaturalCode, our work can also suggest the token that will fix the syntax error.

c) Preventing syntax errors: is the goal of research that wishes to reduce syntax errors or make syntax errors impossible to make, even in text-based languages. This is often accomplished by blurring the line between the program editor and the language itself and engaging in tree edits, such as the tree edit states of Omar et al. [31]'s proposed Hazel, or Project Lambdu [32] where an AST is modified within the editor instead of text.

Numerous compilers have placed a focus on more user-friendly error messages that explain the error and provide solutions. Among these are Clang [10], Rust [33], Scala [34], and Elm [35].

d) Deep learning: is a technique to model token distributions on software text [36, 37]. Recurrent neural networks (RNNs), unlike feedforward neural networks, have links between layers that form a directed cycle, effectively creating a temporal memory. RNNs have been successful in speech recognition [38]. Long short-term memory (LSTM) neural networks, extend RNNs by protecting its internal memory cells from being affected by the "squashing" effect of the logistic activation functions across recurrent links. Typically neural networks' weights and parameters are trained by some form of stochastic gradient descent, a gradient descent that shuffles inputs each round. RNNs and LSTMs have shown value in processing natural language texts and programming language texts. Hellendoorn and Devanbu [39] discuss the effectiveness of deep learning (specifically RNNs and LSTMs) on the task of suggesting the next token in a file. The authors compare deep learning against a n-gram/cache language model that dynamically changes based on what file and packages are in scope in an editing session within a IDE. They find that combining deep learning and the aforementioned cache language model achieves very low entropy. The disadvantage is that deep learning often requires keeping a closed vocabulary, making deep learning unsuitable for predicting novel identifiers and literals in ever-changing scopes. The LSTM model we present in this paper maintains a closed vocabulary; for the purpose of detecting syntax errors, the exact value of identifiers and literals is irrelevant. Others have applied recurrent neural networks (RNNs) to source code. White et al. [37] trained RNNs on source code and showed their practicality in code completion. Similarly, Raychev et al. [36] used RNNs in code completion to synthesize method call chains in Java code. Dam et al. [40] provides an overview of LSTMs instead of ordinary RNNs as language models for code. Our work is similar to code completion, in that given a file with one token missing, *Sensibility* may suggest how to complete it; however, our focus is on syntax errors, rather than helping complete code as it is being written.

e) Deep learning and syntax error fixing: has already been attempted by a few researchers with different degrees of success and treatment. Bhatia and Singh [8] present SynFix, which uses RNN and LSTM networks (collectively, RNNs) to automatically correct syntax errors in student assignments. The RNNs are trained on syntactically-correct student submissions, one model per programming assignment. RNNs takes a 9-token window from the input file and is trained to return a 9-token window shifted one token to the right. In other words, it outputs the next overlapping sliding window. Given an invalid file, SynFix naïvely generates a sequence of fixes at the error location as returned by the parser. The LSTM approach described in this paper is quite similar to SynFix, however it varies in a few important ways: we use two LSTM models, whereas SynFix uses only one; SynFix is trained on a corpus comprised entirely of syntactically-valid submissions of one particular assignment, whereas our model is trained on a large corpus of syntactically-valid source code collected from GitHub. SynFix uses a thresholding method to rename identifiers, whereas we abstract identifiers (n-grams and LSTMs) and maintain all unique identifiers in the corpus (n-grams). Finally, our method of generating fixes does not rely on the parser's conception of where the syntax error is, as the parser can be quite unreliable in this regard [6]. SynFix was able to completely fix 31.69% of student submissions, and partially fix 6.39% more files.

Gupta et al. [7] describe DeepFix, which models C source code with "common programming errors" (including syntax errors) as a noisy channel. The authors employed with a sequence-to-sequence Gated Recurrent Units (GRU)—a type of recurrent neural network—to decode erroneous C code to compilable C code. They trained their multilayer GRU models on 100–400 token long student-submitted C source code for which they have mutated to introduce synthesised errors. DeepFix is able to completely fix 27% of all erroneous student submissions, and partially fixed an additional 19% of student submissions.

III. Methodology

In order to suggest a fix for a syntax error, first we must find the error. For both finding errors and fixing syntax errors, it is useful to have a function that determines the likelihood of the adjacent token in the token stream given some *context* from the incorrect source code file (Equation 1).

$$P\left(\textit{adjacent-token}|\textit{context}\right) \tag{1}$$

We estimated smoothed n-gram models to approximate the function in Equation 1 (Section IV). We also trained long short-term memory (LSTM) recurrent neural networks in a similar

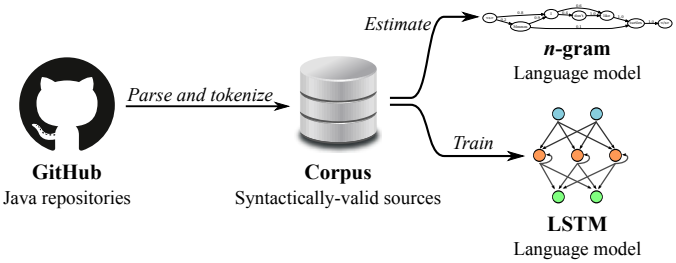

Figure 1: Methodology for training language models of code.

manner (Section V). In order to train the models, we needed a vast corpus of positive examples. For this, we mined over nine thousand of the most popular open source Java repositories from GitHub (Section III-A). This source code was tokenized (Section III-B) such that it could be used to train the Java language models. Finally, we used the approximated functions expressed in Equation 1 to detect a syntax error in a file and suggest a plausible fix. Figure 1 summarizes this process.

A. Mining GitHub for syntactically-valid training examples

To obtain the training data, we downloaded Java source code from GitHub. Since we required Java tokens from each file, other GitHub mining resources such as Boa [41] and GHTorrent [42] were insufficient. Thus, at the end of June 2017, we downloaded the top 10,000 Java repositories by stars (as an analog of popularity). Since GitHub's search application programming interface (API) outputs a total of 1000 search results per query, we had to perform 10 separate queries, each time using the previous least popular repository as the upper bound of stars per repository for the next query. In total, we successfully downloaded 9993 Java repositories.

For each repository, we downloaded an archive containing the latest snapshot of the Git repository's default branch (usually named `master`). We extracted every file whose filename ended with `.java`, storing a SHA-256 hash to avoid storing byte-for-byte duplicate files. We used `javac`'s scanner (tokenizer) and parser as implemented in OpenJDK 8 [4] to tokenize and parse every Java file downloaded. Parsing each source file allowed us to filter only Java source files that were syntactically-valid according to the Java Platform Standard Edition 8 [43], more commonly known as Java 8. In total, we tokenized 2,322,481 syntactically-valid Java files out of 2,346,323 total `.java` files downloaded. All data—repository metadata, source code, and repository licenses—were stored in an SQLite3 database.[1]

B. Tokenization

A *token* is the smallest meaningful unit of source code, and usually consists of one or more characters. For example, a semicolon is a token that indicates the end of a statement.

The set of all possible **unique** tokens tracked by a language model is called the *vocabulary*. Each new source file will likely contain novel variable names or string literals that have never been seen before. This complicates the creation of language models [39], since every novel file presented to the model will likely contain out-of-vocabulary tokens. For the purposes of learning the syntax of the language, we deem these problematic tokens to be irrelevant to the task. Thus, to keep a generally small, bounded vocabulary that has enough unique tokens to faithfully represent the syntax and regularity of handwritten Java, we *abstracted* certain tokens.

Table I: Token kinds according to the Java SE 8 Specification [44], and whether we abstracted them or used them verbatim.

Token kind	Action	Examples
Keyword	Verbatim	`if`, `else`, `for`, `class`, `strictfp`, `int`, `char`, `const`, `goto`, …
Keyword literal	Verbatim	`true`, `false`, `null`
Separators	Verbatim	`(`, `)`,`{`, `}`,`[ ]`,`;`, `,`, `.`, `...`, `@`, `::`
Operators	Verbatim	`+`, `=`, `::`, `>>>=`, `->`, …
Identifier	Abstracted	`AbstractSingletonFactory`, `$ecret`, `buñuelo`
Numeric literal	Abstracted	`4_2L`, `0xC0FFEE`, `0755` `0b101010`, `.3e-02d`, `0xFFp+12f`, `'™'`
String literal	Abstracted	`"hello, world"`

A key insight is that, in order to model the *syntax* of a programming language, it is unnecessary to model precise variable names and literal values. Thus, when creating a fixed vocabulary, we abstracted certain tokens that vary between files, and kept all other tokens verbatim (Table I). The result was 110 unique tokens for the Java 8 standard. In addition to these tokens, we added the synthetic `<unk>` token to encode out-of-vocabulary tokens, and `<s>` and `</s>` tokens such that we could encode beyond the start and end of a source file, respectively [45]. Thus, the total size of our vocabulary was 113 unique tokens for the abstract models.

C. Tokenization pipeline

Table II: The series of transformations from source code to vectors suitable for training the language models. The simplified vocabulary indices are "=" = 0, ";" = 1, "String" = 2, and "Identifier" = 3.

Original source code	`greeting = "hello";`
Tokenization	Identifier(`"greeting"`), Operator(`"="`), String(`"hello"`), Separator(`";"`)
Vocabulary abstraction	Identifier = String ;
Vectorization	[3 0 2 1]

To convert a source file to a form that is suitable for training the models, we performed a series of transformations (Table II). The raw source code was tokenized in a form that is suitable as input for the Java parser. As mentioned in Section III-A, this was done for each Java file using `javac`.

Then, we normalized the token stream such that each token is an entry of the abstracted vocabulary. The exact text of tokens belonging to open classes was discarded for training, except in the case of the 10-gram Concrete model (Section IV). Each token in the vocabulary is assigned a non-negative integer index for the LSTM model or a text name for the 10-gram Abstract model.

[1] Available: https://archive.org/details/sensibility-saner2018

IV. Training n-gram models for syntax error correction

Inspired by the prior work by Campbell et al. [6], we implemented two separate 10-gram models for syntax error correction. The models work by first estimating a Modified Kneser-Ney smoothed 10-gram model on the valid code in the training corpus. Since the 10-gram model expects a corpus of space-separated words, each token in the training source code file was converted into a single "word". Unlike LSTMs, the n-gram model can easily handle arbitrary extensions to its vocabulary, so we created both an *abstract* model—like the LSTM models—and a *concrete* model, where token text was ingested verbatim for all tokens.

A. Detecting syntax errors with n-gram models

When presented with a syntactically invalid file which needs correction, the tool breaks the source code into 21-token-long windows, which are compared against the model for their cross-entropy. Equivalently, this is the negative logarithm of the probability of encountering that specific 21-token window according to the Kneser-Ney smoothed 10-gram model. The model in this case is either the abstract model, containing token types, or the concrete model, containing the actual text of each token.

Then, the cross-entropy is converted to a specific value for each token. Each token is assigned a score equal to the average entropy of every window which it was a member of. Since the window length is 21, each token is a member of 21 windows, and so its score is the mean entropy of those 21 windows.

The window length, 21 was chosen to match the LSTM window length in the next section and so that there is at least one 10-gram before each token and one 10-gram after each token being examined. This takes advantage of the fact that a single token can affect the entropy of a window ending with that token, the entropy of a window beginning with that token, and windows at every position in between.

The tool considers the token with the highest score to be the most likely location of a code mistake, since it contributed the most entropy to the 21 windows it was in.

B. Fixing syntax errors with n-gram models

The top-10 scoring tokens are then considered in turn for correction. This is limited to 10 to limit runtime. For each of the top-10 scoring tokens, the model attempts to delete, insert a token before, or substitute that token.

For deletion, there is only one option for each of the top-10 scoring tokens: try deleting that token. For insertion and substitution, the model has many options. It can insert any token it has seen before at that location, or it can substitute the token at that location with any token it has seen before. The tool tries any token it has seen in the training corpus with frequency ≥ 1000. This lower limit on token frequency is also imposed to limit the runtime of the tool.

The above process produces many possible fixes. For each possible fix, the tool uses the model to compute the entropy of the 21-token window with the deleted, inserted, or substituted token at the center. This entropy is subtracted from the original entropy, before the suggestion was applied. The resulting value is how much the cross-entropy of the erroneous file was decreased (or increased) by applying a possible fix.

If the entropy has decreased, this means the probability of this code being observed before has increased, which means, according to the model, the file (with the possible fix applied) is more likely to be syntactically correct. The fixes are ranked based on how much the entropy decreased after the fix suggestion was applied. The suggestion which causes the greatest decrease in entropy when applied is reported first.

The search process described above takes less than two seconds on a modern CPU (Intel® Core™ i7-3700K) to produce a list of fixes and check them for syntactic-validity.

V. Training LSTMs for syntax error correction

To train the LSTMs, each source file was converted into a vector, representing the tokens of an entire file. The vector is constructed by substituting each token with its corresponding numeric index in the abstracted vocabulary (Section III-B). Finally, each vector was converted into a *one-hot encoded* matrix (also known as *one-of-k encoding*). In a one-hot encoding, exactly one item in each column is given the value one; the rest of the values in the column are zero. The one-hot bit in this encoding represents the index in the vocabulary. Thus, the matrix has as many columns as tokens in the file and has as many rows as entries in the abstracted vocabulary. Each column corresponds to a token at that position in the file, and has a single one bit assigned to the row corresponding to its (zero-indexed) entry in the vocabulary.

The modelling goal is to approximate a function that, given a context from source code, determines the likelihood of the adjacent token. If this function judges the token as unlikely, it indicates a syntax error. This function solves the first problem: finding the location of a syntax error, as demonstrated by Campbell et al. [6]. However, to fix errors, it is also necessary to know what tokens actually *are* likely for each given context. We rephrase Equation 1 such that, instead of predicting the likelihood solely of the adjacent token, it returns the likelihood of *every entry in the vocabulary*. In other words, we want a function that returns a *categorical distribution* (Equation 2).

$$\mathit{adjacent}\,[\mathit{context}] = \begin{cases} P(\texttt{if}|\mathit{context}) \\ P(\texttt{else}|\mathit{context}) \\ P(\text{Identifer}|\mathit{context}) \\ P(\text{String}|\mathit{context}) \\ \dots \\ P(\texttt{\}}|\mathit{context}) \end{cases} \tag{2}$$

The categorical distribution can also be seen as a vector where each index corresponds to the likelihood of an entry in the vocabulary being the adjacent token. Being a probability distribution, the sum of the elements in this vector add up to 1.0. The probability distribution works double duty—because it outputs probabilities, it can determine what the most probable

Listing 2: Syntactically-valid Java

```
1 if (activity == null) {
2     this.active = false;
3 }
```

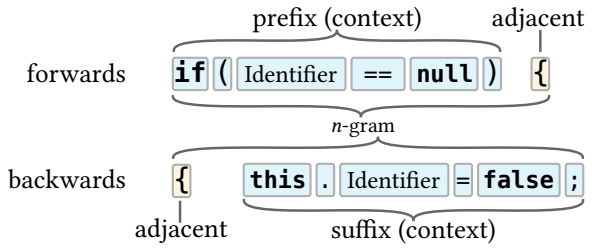

Figure 2: The relationship between an n-gram, the contexts, and the adjacent token. In this diagram, $n = 7$, and the adjacent token is `{` in both cases. Thus, the contexts are $n - 1$ or 6 tokens long.

adjacent token *should be*; hence, it can be used to determine possible fixes (discussed in Section V-B).

To approximate such a function, we used deep learning to map contexts to categorical distributions of the adjacent token. For this task, we employed long short-term memory (LSTM) recurrent neural networks, as they were successfully used by prior work in predicting tokens from source **two** models—the *forwards* model, given a *prefix* context and code [36, 37]. Unlike prior work, we have trained returning the distribution of the next token; and the *backwards* model, given a *suffix* context and returning the distribution of the previous token. Using recurrent neural networks in two different directions was used successfully in speech recognition [38], but has yet to be applied to source code.

Our insight is that models with opposite viewpoints (that is, different contexts for the same adjacent token) may return *different* categorical distributions. That is, whilst the forwards model may declare that the keyword `this` is likely for the next token, the backwards model may declare that an open brace (`{`) is far more likely than the keyword `this`. With this formulation, we are able to both detect the location of syntax errors and produce possible fixes.

As an example, consider the syntactically-valid Java snippet in Listing 2. Figure 2 illustrates the contexts—both prefix and suffix—when estimating the likelihood of the open brace (`{`) on the first line of Listing 2.

As training input, we iterated over each token in each file, moving a *sliding window* over the tokens of the source code file. Based on empirical work done by White et al. [37] we chose a context length τ of 20 tokens. This corresponds to an n-gram length of 21 tokens, as an n-gram includes both the context and the adjacent token. The prefix was provided as the example input to the forwards model, and the suffix was provided to the backwards model. Both contexts were provided as a one-hot matrix. As example output to both models, we provided the adjacent token as a one-hot vector. To handle tokens whose contexts extend beyond the start and end of the file, we inserted synthetic `<s>` and `</s>` tokens, collectively called *padding* tokens [45]. This means that the first prefix context in the file is comprised entirely of 20 `<s>` tokens; likewise, the last suffix context in the file is comprised of 20 `</s>` tokens.

Table III: Summary of the neural network architecture we trained. $|V| = 113$ is the size of the vocabulary (Section III-B) and $\tau = 20$ is the length of each context in number of tokens.

Input	One-hot matrix, dimensions = $\tau \cdot \lvert V \rvert$		
	Type	Parameters	Activation
	LSTM	300 hidden units	tanh; recur.: hard sigmoid
	Dense		softmax
Output	Categorical distribution, size = $\lvert V \rvert$		
Loss	Categorical cross-entropy		
Optimizer	RMSprop, initial learning rate = 0.001		

We used Keras 2.0.8 [46], a Python deep neural network framework, to define our model architecture, using the Theano 0.8.2 [47] backend to train the models proper. A summary of the precise architecture (hyperparameters) that we used is given in Table III. The LSTM layer has 300 hidden units, based on the observation by White et al. [37] that recurrent neural networks with 300 outputs with 20 tokens of context or 400 outputs with 5 tokens of context have the lowest perplexity with respect to the corpus of source code. We used the RMSprop [48] gradient descent optimizer with an initial learning rate of 0.001, optimizing to minimize categorical cross-entropy. We ran a variable number of *epochs*—full iterations of the training examples—to train the models, using *early stopping* to determine when to stop training. Early stopping was configured to stop upon detecting three consecutive epochs (its *patience* parameter) that yield no improvement to the categorical cross-entropy with respect to the validation examples. Later tuning (Section VIII-A) revealed that shuffling samples within files, patience of 10, and the Adam optimizer had far better results.

Once each model was trained, it was serialized in Hierarchical Data Format 5 (HDF5). In total, the weights and biases of each individual model resulted in 6.1 MiBs of data.[2] Each model individually took between 2½ and 11½ days to train; up to six models were trained simultaneously on two Nvidia® GeForce® GTX 1070 GPUs. Section VII-A discusses how files were chosen for the training, validation, and testing sets.

A. Detecting syntax errors with dual LSTM models

We used the output of the models trained in Sections IV and V to find the likely location of the syntax error. Given a file with one syntax error, we tokenized it using `javac`. `javac`'s tokenizer is able to tolerate erroneous input in the tokenization stage, which produces a token stream (as opposed to the parsing stage, which produces an abstract syntax tree).

Recall that each model outputs the probability of the adjacent token given a *context* from the token stream, but each model differs in where the context is located relative to the adjacent token. We used the two independent probability distributions to determine which tokens are quantifiably unlikely, or "unnatural" [6].

[2] Available: https://archive.org/details/sensibility-replication-package

To calculate "naturalness", we summed the cross-entropy of each probability model with respect to the erroneous source file. Once the naturalness of every single token in the file is calculated, we return a sorted list of the least likely tokens, or, in other words, the locations that are most likely to be a syntax error.

For a discrete distribution $p(x)$ and a distribution $\hat{q}(x)$ that models $p(x)$, the cross-entropy is computed as follows

$$H(p, \hat{q}) = -\sum_{x \in X} p(x) \log_2 \hat{q}(x)$$

This returns a value in $[0, \infty)$, where 0 indicates that $\hat{q}$ models p perfectly; as the cross-entropy increases, this increases the amount of information required to model $p(x)$. Hence, a value closer to 0 is more "natural", whereas larger values indicates a more "unnatural" event.

For each token position in the erroneous source file, we calculate the cross-entropy with respect to the current token in the file, for both the forwards and backwards models. That is, we let $p(x)$ equal 1 *iff* x is equal to the token at the current position in the file.

$$p(x) = \begin{cases} 1, x = \text{actual token} \\ 0, \text{otherwise} \end{cases}$$

We then use $\hat{q}_f(x|\text{prefix})$ and $\hat{q}_b(x|\text{suffix})$ obtained from consulting the forwards and backwards models with their corresponding context, respectively. We then combine the two cross-entropies by summing. Thus, the "unnaturalness" of a token t from the source file is:

$$unnaturalness = H(p, \hat{q}_f) + H(p, \hat{q}_b)$$

To obtain a ranked list of possible syntax error locations, we sort the list of tokens in the file in descending order of unnaturalness. That is, the positions with the highest summed cross-entropy are the most likely location of the syntax error.

B. Fixing syntax errors with dual LSTM models

Given the top-k most likely syntax error locations (as calculated in Section V-A), we use a naïve "guess-and-check" heuristic to produce and test a small set of possible fixes. Using the categorical distributions produced by the models, we obtain the top-j most likely adjacent tokens (according to each model) which may be inserted or substituted at the estimated location of the fault. Each fix is tested to see if, once applied, it produces a syntactically-valid file. Finally, we output the valid fixes.

For a given syntax error location, we consult each model for the top-j most likely tokens at that position. The models often produce similar suggestions, hence we take the union of the two sets.

$$suggestions = \text{top}_j\left(\hat{q_f}\right) \cup \text{top}_j\left(\hat{q_b}\right)$$

We then try the following edits, each time applying them to the syntactically-invalid file, and parsing the result with `javac` to check if the edit produces a syntactically-valid file. If it does, we consider the edit to be a *valid fix*. We use the following strategies to produce fixes:

1) Assume a token at this location was erroneously **deleted**. For each t in the set of suggestions, insert t at this location.
2) Assume the token at this location was erroneously **inserted**. Delete this token.
3) Assume the token at this location was erroneously **substituted** for another token. Substitute it with t, for each t in the set of suggestions.

We repeat this process for the top-k most likely syntax error locations. We let $k = 3$ to limit runtime. j was calibrated according to the *perplexity* of the estimated probability distribution. Perplexity is, roughly speaking, the expected number of choices that an estimated distribution $\hat{q}$ requires to model events from the true distribution p. It is directly related to cross-entropy $H(p, \hat{q})$ by $2^{H(p,\hat{q})}$. To determine j, we used the ceiling of the highest validation cross-entropy obtained while training. The highest cross-entropy was 1.5608, or a perplexity of just under 3; therefore, we let $j = 3$.

Finally, all valid fixes are output to the user. The LSTM model takes less than three seconds on a modern CPU (Intel® Core™ i5-3230M) to produce a list of suggested fixes for a single file. Every single fix suggested this way is guaranteed to produce a syntactically-valid file.

VI. Mining Blackbox for novice mistakes

Both the n-gram and LSTM model presented in this paper are trained on data from professional code available on GitHub. In order to properly evaluate these models for their intended purpose—detecting and correcting syntax errors in novices' code—we would ideally have access to a repository of syntax errors made by novices.

Blackbox [9] is a continually-updated repository of events from the BlueJ Java IDE [14]—an IDE aimed primarily at novices learning Java in introductory computer science classes. Blackbox has been used to analyze novice programmers before [5, 13, 49, 50]. The data contains over four years of IDE events collected from users around the world who have opted-in to anonymously share edit events. The IDE events includes the start and end of an editing session, the invocation of the compiler, whether a compilation succeeded or failed, and edits to lines of code. Importantly, one can reconstruct the source code text at the time of any compilation event.

We mined Blackbox in order to find realistic data to evaluate our syntax-correction algorithms. By using the IDE event data in Blackbox, we collected syntax mistakes "in the wild", which are accompanied by the *true* resolution to that mistake. The intended audience of *Sensibility* is novices, thus it is important to evaluate our syntax error correction methods against actual code that a novice would write.

A. Retrieving invalid and fixed source code

For the purposes of the evaluation, we sought to retrieve pairs of revisions to a source code file: the revision directly prior to a compilation that failed due to a syntax error, and the revision immediately following which compiled successfully.

Table IV: Edit distance of collected syntax errors

Edit Distance	Instances	Percentage (%)
0	10,562	0.62
1	984,471	57.39
2	248,388	14.48
3	93,931	5.48
4	54,932	3.20
5 or more	323,028	18.83
Total	1,715,312	

Table V: Summary of single token syntax-errors

Edit Operation	Instances	Percentage (%)
Insertion	223,948	22.75
Substitution	77,846	7.91
Deletion	682,677	69.34

In order to collect such pairs of revisions, we iterated through every editing session from the beginning of data collection up to midnight, July 1, 2017, UTC. For each session, we iterated through each pair of compilation events and filtered the pairs wherein the former compilation had failed (the "before" revision) and the compilation immediately following had succeeded (the "after" revision). We retrieved the complete Java source code at each revision of the pair and kept only those pairs wherein the source code of the "before" revision was syntactically-incorrect (verified using `javac`).

For the purposes of our analyses, we calculated the *token-wise edit distance* between the before-and-after revisions of source code. We used Levenshtein distance [51], wherein two strings of tokens are compared for the minimum amount of single token *edits* (insertions, deletions, or substitutions) that are required to transform one string of tokens into the other. Thus, edit distance indicates how many edits, at minimum, a syntax error correcter must suggest to transform an invalid source file to a syntactically-valid source file.

B. Findings

In total, we collected 1,715,312 before-and-after pairs matching our criteria. Of these pairs, 984,471 (57.39 %) were single-token syntax errors (Table IV)[3]; This means that, even if a tool accounts *only* for single-token mistakes, the tool accounts for the majority of syntax errors.

> The majority of syntax errors we found differed from the fixed source file by a **single** edit.

We further studied the single-token syntax errors. Table V breaks down the errors into each edit operation. The majority of single-token syntax errors are deletions—such as when a programmer misses a token such as a semi-colon, or a brace. Insertions account for almost one-quarter of errors, while substitutions are the least common.

[3]Syntax errors caused by invalid escape sequences within string literals were considered to have an edit distance of zero, as only the contents of the string must change, but not the actual token type.

Table VI: Number of tokens between partitions

	Mean	S.D.	Median	Min	Max
Train	8.16 M	596,019.30	8.06 M	7.45 M	9.00 M
Validation	3.80 M	365,337.76	3.68 M	3.47 M	4.34 M

In Section VII, we describe how we used the single-token syntax errors we mined to evaluate LSTM and n-gram methods for syntax error correction and detection.

VII. Evaluation

To determine the practical usefulness of *Sensibility*, we ask the following questions:

1) How well does *Sensibility* find syntax errors?
2) How often does *Sensibility* produce a valid fix?
3) If a fix is produced, how often is it the same fix used by the student to fix their own code?

To answer these questions, we created three "qualifications" for a suggested fix. The first qualification is that the fix suggestion must be at the exact *location* of the mistake in the code, down to the individual token.

To judge how well *Sensibility* produces syntactically-valid fixes, we created a second qualification, *valid fix*, which is stricter: the fix suggestion must be at the exact location of the mistake in the code, and applying the fix suggestion must yield a syntactically valid source file.

The third qualification is the most strict: the fix suggestion must be exactly the same token (abstracted or concrete, depending on the model) that the student used to fix their own code. A fix suggestion that matches the student's own fix is called a *true fix*. A true fix precisely reverses the mistake that introduced the error.

Then we found the highest ranking fix suggestion produced by the tool using various models and training partitions that met the above qualifications.

A. Partitioning the data

To empirically evaluate *Sensibility*, we repeated our experiments five times on mutually-exclusive subsets of source code files called *partitions*. Each of the five partitions are subdivided further into two mutually-exclusive sets: the **train set**, and the **validation set**, thus resulting in 10 sets total (described in Table VI). The validation set was used exclusively for the LSTM training procedure described in Section V, but not used when estimating n-gram models.

We populated every set with Java source code from the GitHub corpus collected in Section III-A. We split our training and validation data into five partitions to demonstrate how the performance of *Sensibility* changes when trained on completely different data. Keeping each corresponding partition the same size facilitates the comparison of results between partitions. This also represents the expected use case of an end-user who will never retrain *Sensibility*.

When assigning source code files to partitions, we imposed the following constraint to ensure the independence of training partitions: The source code files of a single repository cannot

be distributed over partitions; in other words, every source code file in a given repository must be assigned to one and only one partition. The constraint's purpose is to make the evaluation more realistic, considering the expected use case of *Sensibility*. If a user is trying to figure out a syntax error in their own hand-written source code, it is likely that their model was trained on whole projects at once.

The test files did not come from the GitHub corpus. Instead, they came from the Blackbox repository of novice programmers' activity [9]. The same set of test files were used to evaluate each partition; Only the training and validation files changed between partitions.

B. Finding the syntax error

To quantify *Sensibility*'s accuracy in finding the error, we calculated the *mean reciprocal rank* (MRR) of the ranked syntax error location suggestions. Reciprocal rank is the inverse of the rank of the first correct location found in an ordered list of syntax error locations for a file q. Mean reciprocal rank is the average of the reciprocal rank for every file q in the set of total solutions attempted Q:

$$MRR = \frac{1}{|Q|} \sum_{q \in Q} \frac{1}{rank_q}$$

MRR is always a value in $[0, 1]$, where an MRR of 1 is the best possible score, obtained when the first suggestion is always the correct location of the error. Conversely, an MRR of 0 means that the correct location number is never found in the ranked list of suggestions. For example, if for one mistake, the correct token was given first, for another student mistake, the correct token was given third, and for yet another mistake the correct token was never found, the token-based MRR would be $\frac{1}{3}\left(\frac{1}{1} + \frac{1}{3} + 0\right) = 0.44$. MRR is quite conservative: in the case that the correct result is ranked first half of the time and ranked second the rest of the time, the MRR is only 0.75. To quantify *Sensibility*'s ability to find syntax errors, we determined how often it finds the *exact location* of the erroneous token.

C. Fixing the syntax error

To evaluate the effectiveness of *Sensibility* to fix syntax errors, we measured how often the models produces an edit that, when applied, produces a syntactically-valid file. We report this as a *valid fix*. A stricter measure of success is how often *Sensibility* produces the *true fix*. The true fix is defined as the exact same fix that the student applied to their own code. For *abstract* tokens, the true fix requires that the tool applied the correct operation (insertion, deletion, substitution) at the correct location, and with the correct token type. For *concrete* tokens, the token must also match exactly, not just its type, to count as a true fix. Thus, for the 10-gram Concrete to produce a true fix it must produce the exact identifier, number literal, string, etc. that the student used to fix their code. So, if the student's fix was to insert the identifier `a`, the 10-gram Concrete model must also insert the identifier `a`, while the abstract models must only suggest inserting `identifier` at that same location.

Table VII: MRRs of n-gram and LSTM model performance

Model	Qualification	1	2	3	4	5	All
10-gram Abstract	Location	.41	.42	.41	.40	.40	.41
	Valid Fix	.39	.39	.39	.38	.38	.39
	True Fix	.36	.36	.36	.35	.35	.36
10-gram Concrete	Location	.07	.07	.07	.08	.07	.07
	Valid Fix	.06	.06	.06	.07	.06	.06
	True Fix	.04	.04	.04	.04	.04	.04
LSTM 1 (RMSProp) (no reshuffling)	Location	.06	.05	.05	.05	.05	.05
	Valid Fix	.06	.05	.05	.05	.05	.05
	True Fix	.05	.04	.04	.04	.04	.04
LSTM 2 (Adam) (reshuffling)	Location	.52	.53	.53	.50	.50	.52
	Valid Fix	.52	.53	.52	.49	.50	.51
	True Fix	.46	.46	.46	.44	.44	.46

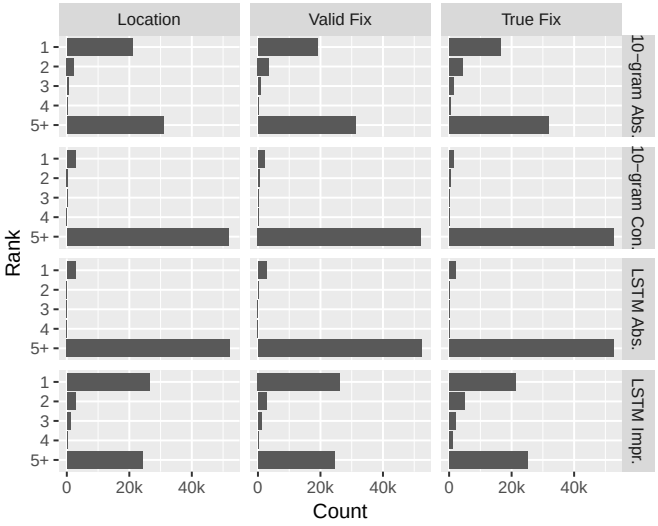

Figure 3: The mean reciprocal ranks of determining the exact location, a valid fix, and the true fix of student mistakes for all three models.

VIII. Results

Performance of finding syntax errors is measured by mean reciprocal rank. Table VII lists the MRR obtained when locating the exact token of the syntax error, generating a valid fix, and generating the true fix for each tool. As can be seen from the table, results are very consistent across partitions.

The values in Table VII are all well above what would be expected from guessing by chance alone. If a file had 100 lines and 10 tokens per line, location MRR would be 0.002 just by guessing. Since fix MRRs depend on determining the location first, just by guessing, they would be even lower than 0.002.

Figure 3 is a series of histograms visualizing the ranks of each qualification. The width of each bar displays the number of observations that have a rank at that value. Figure 3 shows that the MRR values in Table VII are dominated by either rank 1 results, that is, the tools first result was qualifying, or rank $5+$ results, that is the qualifying result was far down the suggestion list, or the tools were unable to produce a qualifying result. In this case, qualifying means the result is the correct location, the result was the first valid fix, or the result that was the true fix.

Performance of fixing syntax errors In all cases, there is

a clustering of reciprocal ranks at 1.0, meaning that the 10-gram Abstract model tool can suggest the true fix as its first suggestion 30.22% of the time. The 10-gram Concrete model tool can suggest the true fix as its first suggestion 3.52% of the time. The LSTM 1 tool can suggest the true fix as its first suggestion 3.84% of the time. LSTM 2 is described in Section VIII-A.

A 10-gram Abstract model can produce a fix for syntactically-invalid code by suggesting the correct token type, operation, and location about one third of the time.

A. Improving the performance of the LSTMs

The lacklustre results of LSTM 1 were surprising; thus, we engaged in a *hyperparameter search*. We tested 985 different configurations, varying the hidden layer size (50, 100, 200, 300, 400, 1000), the context size (5, 10, 15, 20), and optimizer (RMSprop, Adam), among other hyperparameters (refer to Section V). After evaluating each configuration, we found that an architecture identical to the one presented in Section V performed exceptionally well, with three critical differences:

- We used the Adam optimizer [52] instead of RMSprop;
- We trained longer, increasing the patience to 10 epochs;
- We shuffled samples *within* each file prior to constructing mini-batches.

The results of applying the same evaluation methodology presented in Section VII is at the end of Table VII. The improved LSTM configuration considerably outperformed all models in all qualifications. With an MRR of .46 for true fix, this improved LSTM model was capable of fixing nearly half of all single-token syntax errors in the corpus.

IX. Discussion

Using *Sensibility* to correct multiple mistakes at once is theoretically possible but due to the fact that, at best, its first fix suggestion is only the true fix approximately half of the time, more advanced search strategies may be required that try fixes in different combinations. Alternatively more generic forms of search can be exploited, such as SMT solvers or heuristic search to help produce more parsable and safe outputs. In this work we only explore code as a one-dimensional token stream, but hand-written code is often laid out in two dimensions on the user's screen. Spatial relationships or positioning could prove useful in future error locators and fixers.

Our work is complementary to other syntax error detection tools such as Merr [11] and Clang's "fix it" hints [10]. This work can be integrated with other tools to squash syntax errors. *Sensibility* is only one step in the hunt for typos. It attempts to handle any typos that would produce a definite syntax error. However, it does not address misspelled variable names, or type errors. *Sensibility* can help other tools that require a valid abstract syntax tree—such as type checkers—that cannot work on code with invalid syntax. Localness [53], online n-gram models, or other search-based models might be feasible candidates to help resolve identifiers. Further investigation is required into applying ensembles of learners to combine the strengths of deep learning, smoothed n-gram models, and other probabilistic models to achieve high precision in detecting and fixing syntax errors.

A. Threats to validity

a) Construct validity: is threatened by the abstraction of identifiers. By suggesting that an identifier be inserted or modified, the choice is still up to the programmer since the code may parse, but may not necessarily be compilable.

b) Internal validity: is threatened by our reliance on the Blackbox data set [9, 13]. There is potential bias from the curators of the dataset, and a self selection bias as the syntax errors are submitted with the consent of the students. Internal validity could be harmed by using directly adjacent compilations: the first for erroneous code and the second for the fixed source code. This could miss situations where a syntax error is made, doesn't compile, and several fixes are attempted by a human. In this case the erroneous source code we used wouldn't match the original erroneous source code a human had written, that is, their first attempt.

c) External validity: is addressed by the use of a large number of Java source files; however these source files were only collected from GitHub, thus are not necessarily representative of all Java code or code of other languages.

X. Conclusions

We have described a method of exploiting n-gram and LSTM language models to locate syntax errors and to suggest fixes for them. Typically a fix is found about half of the time. The fix is often suggested based on abstract tokens, so often the end-user needs to fill in the appropriate identifier.

Abstract vocabularies overcomes two limitations of software engineering deep learning problems: vocabulary size and unseen vocabulary, and cost of training for end-users. Both n-gram and LSTM models require extensive training; we present a method for training on large corpora before deployment such that end-users of the model never have to locally train.

This work demonstrates that search aided by naturalness—language models applied to source code—can produce results with appropriate accuracy and runtime performance (two to three seconds on a modern CPU).

In summary by training language models on error-free code, we enable them to locate errors through measures of perplexity, cross-entropy, or confidence. We can then exploit the same language models bi-directionally to suggest the appropriate fixes for the discovered syntax errors. This relatively simple method can be used to aide novice learners avoid syntax-driven pitfalls while learning how to program.

Acknowledgments

We thank Neil C. C. Brown, Ian Utting, and the entire Blackbox administration team for the creation of the Blackbox repository. and the continual support and community that they foster. We thank Julian Dolby, for offering a new perspective on the problem. We would also like to thank Nvidia Corporation for their GPU grant. This work was funded by a MITACS Accelerate with BioWare and a NSERC Discovery Grant.

References

[1] E. S. Tabanao, M. M. T. Rodrigo, and M. C. Jadud, "Identifying at-risk novice Java programmers through the analysis of online protocols," in *Philippine Computing Science Congress*, 2008.

[2] M. C. Jadud, "A first look at novice compilation behaviour using BlueJ," *Computer Science Education*, vol. 15, no. 1, pp. 25–40, 2005.

[3] E. S. Tabanao, M. M. T. Rodrigo, and M. C. Jadud, "Predicting at-risk novice Java programmers through the analysis of online protocols," in *Proceedings of the seventh international workshop on Computing education research*, ser. ICER '11. New York, NY, USA: ACM, 2011, pp. 85–92. [Online]. Available: http://doi.acm.org/10.1145/2016911.2016930

[4] Oracle Corporation, "The Java programming language compiler group," http://openjdk.java.net/groups/compiler/, 2017, (Accessed on 08/04/2017).

[5] N. C. Brown and A. Altadmri, "Investigating novice programming mistakes: Educator beliefs vs. student data," in *Proceedings of the tenth annual conference on International computing education research*. ACM, 2014, pp. 43–50.

[6] J. C. Campbell, A. Hindle, and J. N. Amaral, "Syntax errors just aren't natural: Improving error reporting with language models," in *Proceedings of the 11th Working Conference on Mining Software Repositories*. ACM Press, 2014, pp. 252–261, available: http://dl.acm.org/citation.cfm?doid=2597073.2597102.

[7] R. Gupta, S. Pal, A. Kanade, and S. Shevade, "DeepFix: Fixing common C language errors by deep learning." in *AAAI*, 2017, pp. 1345–1351.

[8] S. Bhatia and R. Singh, "Automated correction for syntax errors in programming assignments using recurrent neural networks," 2016, available: http://arxiv.org/abs/1603.06129.

[9] N. C. C. Brown, M. Kölling, D. McCall, and I. Utting, "Blackbox: a large scale repository of novice programmers' activity," in *Proceedings of the 45th ACM technical symposium on Computer Science Education*. ACM, 2014, pp. 223–228.

[10] "Clang—Expressive Diagnostics," October 2016, available: http://clang.llvm.org/diagnostics.html.

[11] C. L. Jeffery, "Generating LR Syntax Error Messages from Examples," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 25, no. 5, pp. 631–640, 2003.

[12] The Eclipse Foundation, "4.7 - Eclipse project downloads," http://download.eclipse.org/eclipse/downloads/drops4/R-4.7-201706120950/#JDTCORE, 2017, (Accessed on 08/09/2017).

[13] A. Altadmri and N. C. Brown, "37 million compilations: Investigating novice programming mistakes in large-scale student data," in *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*. ACM, 2015, pp. 522–527.

[14] M. Kölling, B. Quig, A. Patterson, and J. Rosenberg, "The BlueJ system and its pedagogy," *Computer Science Education*, vol. 13, no. 4, pp. 249–268, 2003. [Online]. Available: http://www.tandfonline.com/doi/abs/10.1076/csed.13.4.249.17496

[15] M. C. Jadud, "Methods and tools for exploring novice compilation behaviour," in *Proceedings of the second international workshop on Computing education research*. ACM, 2006, paper http://www.jadud.com/people/mcj/files/2006-icer-jadud.pdf, pp. 73–84.

[16] J. Jackson, M. J. Cobb, and C. Carver, "Identifying top Java errors for novice programmers," in *Frontiers in Education Conference*, vol. 35. STIPES, 2005, p. T4C.

[17] S. Garner, P. Haden, and A. Robins, "My program is correct but it doesn't run: a preliminary investigation of novice programmers' problems," in *Proceedings of the 7th Australasian conference on Computing education-Volume 42*. Australian Computer Society, Inc., 2005, pp. 173–180.

[18] L. McIver, "The effect of programming language on error rates of novice programmers," in *12th Annual Workshop of the Psychology of Programming Interest Group*. Citeseer, 2000, pp. 181–192.

[19] S. K. Kummerfeld and J. Kay, "The neglected battle fields of syntax errors," in *Proceedings of the fifth Australasian conference on Computing education-Volume 20*. Australian Computer Society, Inc., 2003, pp. 105–111.

[20] M. Hristova, A. Misra, M. Rutter, and R. Mercuri, "Identifying and correcting Java programming errors for introductory computer science students," *ACM SIGCSE Bulletin*, vol. 35, no. 1, pp. 153–156, 2003.

[21] P. Denny, A. Luxton-Reilly, and E. Tempero, "All syntax errors are not equal," in *Proceedings of the 17th ACM annual conference on Innovation and technology in computer science education*. ACM, 2012, pp. 75–80.

[22] T. Dy and M. M. Rodrigo, "A detector for non-literal Java errors," in *Proceedings of the 10th Koli Calling International Conference on Computing Education Research*, ser. Koli Calling '10. New York, NY, USA: ACM, 2010, pp. 118–122. [Online]. Available: http://doi.acm.org/10.1145/1930464.1930485

[23] M.-H. Nienaltowski, M. Pedroni, and B. Meyer, "Compiler error messages: What can help novices?" *SIGCSE Bull.*, vol. 40, no. 1, pp. 168–172, Mar. 2008. [Online]. Available: http://doi.acm.org/10.1145/1352322.1352192

[24] G. Marceau, K. Fisler, and S. Krishnamurthi, "Measuring the effectiveness of error messages designed for novice programmers," in *Proceedings of the 42nd ACM technical symposium on Computer science education*. ACM, 2011, pp. 499–504.

[25] B. A. Becker, "An exploration of the effects of enhanced compiler error messages for computer programming

novices," Master's thesis, Dublin Institute of Technology, 2015.

[26] T. Barik, K. Lubick, S. Christie, and E. Murphy-Hill, "How developers visualize compiler messages: A foundational approach to notification construction," in *Software Visualization (VISSOFT), 2014 Second IEEE Working Conference on*. IEEE, 2014, pp. 87–96.

[27] D. Pritchard, "Frequency distribution of error messages," in *Proceedings of the 6th Workshop on Evaluation and Usability of Programming Languages and Tools*. ACM, 2015, pp. 1–8.

[28] A. V. Aho and T. G. Peterson, "A minimum distance error-correcting parser for context-free languages," *SIAM Journal on Computing*, vol. 1, no. 4, pp. 305–312, 1972.

[29] R. A. Thompson, "Language correction using probabilistic grammars," *IEEE Transactions on Computers*, vol. 100, no. 3, pp. 275–286, 1976.

[30] T. Parr and K. Fisher, "LL(*): The foundation of the ANTLR parser generator," in *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '11. New York, NY, USA: ACM, 2011, pp. 425–436. [Online]. Available: http://doi.acm.org/10.1145/1993498.1993548

[31] C. Omar, I. Voysey, M. Hilton, J. Sunshine, C. Le Goues, J. Aldrich, and M. A. Hammer, "Toward semantic foundations for program editors," 2017, preprint.

[32] "Lamdu," http://www.lamdu.org/, (Accessed on 09/26/2017).

[33] J. Turner, "Shape of errors to come—the Rust programming language blog," August 2016, available: https://blog.rust-lang.org/2016/08/10/Shape-of-errors-to-come.html.

[34] F. Mulder, "Awesome error messages for Dotty," October 2016, available: http://scala-lang.org/blog/2016/10/14/dotty-errors.html.

[35] E. Czaplicki, "Compiler errors for humans," http://elm-lang.org/blog/compiler-errors-for-humans, 2015.

[36] V. Raychev, M. Vechev, and E. Yahav, "Code Completion with Statistical Language Models," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '14. ACM, 2014, pp. 419–428.

[37] M. White, C. Vendome, M. Linares-Vasquez, and D. Poshyvanyk, "Toward Deep Learning Software Repositories," in *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, 2015, pp. 334–345.

[38] A. Y. Hannun, C. Case, J. Casper, B. Catanzaro, G. Diamos, E. Elsen, R. Prenger, S. Satheesh, S. Sengupta, A. Coates, and A. Y. Ng, "Deep Speech: Scaling up end-to-end speech recognition," 2014, preprint. [Online]. Available: http://arxiv.org/abs/1412.5567

[39] V. J. Hellendoorn and P. Devanbu, "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, 2017, pp. 763–773.

[40] H. K. Dam, T. Tran, and T. Pham, "A deep language model for software code," 2016, preprint. [Online]. Available: http://arxiv.org/abs/1608.02715

[41] R. Dyer, H. A. Nguyen, H. Rajan, and T. N. Nguyen, "Boa: A Language and Infrastructure for Analyzing Ultra-Large-Scale Software Repositories," in *35th International Conference on Software Engineering*, ser. ICSE 2013, 2013, pp. 422–431.

[42] G. Gousios, "The GHTorrent dataset and tool suite," in *Proceedings of the 10th Working Conference on Mining Software Repositories*, ser. MSR '13. IEEE Press, 2013, pp. 233–236. [Online]. Available: http://dl.acm.org/citation.cfm?id=2487085.2487132

[43] J. Gosling, B. Joy, G. Steele, G. Bracha, and A. Buckley, *The Java Language Specification*, 8th ed. Oracle Corporation, February 2015, available: https://docs.oracle.com/javase/specs/jls/se8/jls8.pdf.

[44] ——, *Lexical Structure*, 8th ed. Oracle Corporation, February 2015, ch. 3, available: https://docs.oracle.com/javase/specs/jls/se8/html/jls-3.html.

[45] C. D. Manning and H. Schütze, *Foundations of statistical natural language processing*, 1st ed. Canbridge, MA: The MIT Press, May 1999.

[46] F. Chollet *et al.*, "Keras," https://github.com/fchollet/keras, 2015.

[47] Theano Development Team, "Theano: A Python framework for fast computation of mathematical expressions," May 2016, preprint. [Online]. Available: http://arxiv.org/abs/1605.02688

[48] T. Tieleman and G. Hinton, "RMSprop gradient optimization," April 2014, course slides. [Online]. Available: http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf

[49] S. D. Smith, N. Zemljic, and A. Petersen, "Modern goto: Novice programmer usage of non-standard control flow," in *Proceedings of the 15th Koli Calling Conference on Computing Education Research*, ser. Koli Calling '15. New York, NY, USA: ACM, 2015, pp. 171–172. [Online]. Available: http://doi.acm.org/10.1145/2828959.2828980

[50] A. Altadmri, M. Kölling, and N. C. Brown, "The cost of syntax and how to avoid it: Text versus frame-based editing," in *2016 IEEE 40th Annual Computer Software and Applications Conference (COMPSAC)*, 2016.

[51] V. I. Levenshtein, "Binary codes capable of correcting deletions, insertions, and reversals," in *Soviet physics doklady*, vol. 10, no. 8, 1966, pp. 707–710.

[52] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," 2014. [Online]. Available: http://arxiv.org/abs/1412.6980

[53] Z. Tu, Z. Su, and P. Devanbu, "On the localness of software," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2014, pp. 269–280.

[54] O. Tange, "GNU parallel—the command-line power tool," *;login: The USENIX Magazine*, vol. 36, no. 1, pp. 42–47, Feb 2011. [Online]. Available: http://www.gnu.org/s/parallel