Syntax Errors Just Aren't Natural: Improving Error Reporting with Language Models

Joshua Charles Campbell, Abram Hindle, José Nelson Amaral
Department of Computing Science
University of Alberta
Edmonton, Canada
{joshua2, hindle1, amaral} @cs.ualberta.ca

***Abstract*—A frustrating aspect of software development is that compiler error messages often fail to locate the actual cause of a syntax error. An errant semicolon or brace can result in many errors reported throughout the file. We seek to find the actual source of these syntax errors by relying on the consistency of software: valid source code is usually repetitive and unsurprising. We exploit this consistency by constructing a simple N-gram language model of lexed source code tokens. We implemented an automatic Java syntax-error locator using the corpus of the project itself and evaluated its performance on mutated source code from several projects. Our tool, trained on the past versions of a project, can effectively augment the syntax errors produced by the native compiler. Thus we provide a methodology and tool that exploits the naturalness of software source code to detect syntax errors alongside the parser.**

I. Motivation

Syntax errors are a common annoyance in software engineering. Compilers often fail to accurately report the location or nature of a coding mistake, especially for syntax errors. As an example, consider the missing brace at the end of line 2 in the following Java code:

```
1  for (int i = 0; i < scorers.length; i++) {
2    if (scorers[i].nextDoc() == NO_MORE_DOCS)
5      lastDoc = NO_MORE_DOCS;
6      return;
7    }
8  }
```

This mistake, while easy for a human programmer to understand and fix if they know where to look, causes the Oracle Java compiler to report 50 error messages, including those in Figure 1, none of which are on the line with the mistake. This poor error reporting slows down the software development process because a human programmer must examine the source file or change set to locate the error, which can be a very time consuming process. Syntax errors and poor compiler/interpreter error messages have been found to be a major problem for inexperienced programmers [1].

Recent research by Hindle *et al.* [2] successfully leveraged applied statistical techniques from natural language processing (NLP) in order to predict likely next tokens for use in code suggestion and code completion. The approach used in this paper is based on the same n-gram language models, and has been shown to be even more effective on source code than it is on natural language text. The effectiveness of those techniques in this domain is due to the repetitive, and consistent, structure of code.

The idea behind the new method is to train a model on compilable source-code token sequences, and then evaluate on new code to see how often those sequences occur within the model. The main idea is that source code that does not compile should be surprising for an n-gram language model trained on source code that compiles. Intuitively, most available source code compiles. Projects and their source code are rarely released with syntax errors, although this property may depend on the project's software development process. Thus, existing software can act as a corpus of compilable and working software. Furthermore, whenever a source file successfully compiles it can automatically update the training corpus because the goal is to augment the compiler's error messages. Changes to the software might not compile. When that happens, locations in the source file that the model finds surprising should be prioritized as locations that a software engineer should examine.

The following sections of the paper first examine the feasibility of the new method by testing its ability to locate errors in a variety of situations, including situations much more adverse than are expected in practise. Then the paper presents an evaluation of the method's performance when integrated into the compilation process, as an augmentation of compiler error reporting.

The main contributions of this work include:

- A method of statistical, probabilistic syntax-error location detection that exploits n-gram language models.
- A prototype implementation of an n-gram language-model-based Java syntax error locator, UnnaturalCode,[1] that can be used with existing build systems and Java compilers to suggest locations that might contain syntax errors.
- A validation of the feasibility of the new syntax error location detection method.
- A validation of the integration of the new method with the compiler's own methods.

[1] UnnaturalCode is available at https://github.com/orezpraw/unnaturalcode

```
/home/joshua/projects/lucene-4.0.0/core/src/java/org/apache/lucene/search/ConjunctionScorer.java:56:
 error: <identifier> expected
    ArrayUtil.mergeSort(scorers, new Comparator<Scorer>() { // sort the array
                        ^
                                        ... 48 errors omitted ...
/home/joshua/projects/lucene-4.0.0/core/src/java/org/apache/lucene/search/ConjunctionScorer.java:152:
error: class, interface, or enum expected
  }
  ^
[javac] 50 errors
```

Figure 1. Oracle Java Error messages from the motivational example.

- A modified version of `MITLM`,[2] which has routines developed by the authors to calculate the entropy of short sentences with respect to a large corpus quickly.

II. Background

An n-gram language model at its lowest level is simply a collection of counts. These counts represent the number of times a phrase appears in a corpus. These phrases are referred to as n-grams because they consist of at most n words or tokens. These counts are then used to infer the probability of a phrase: the probability is simply the frequency of occurrence of the phrase in the original corpus. For example, if the phrase "I'm a little teapot" occurred 7 times in a corpus consisting of 700 4-grams, its probability would be .01. However, it is more useful to consider the probability of a word in its surrounding context. The probability of finding the word "little" given the context of "I'm a _____ teapot" is much higher because "little" may be the only word that shows up in that context — a probability of 1.

These n-gram models become more accurate as n increases because they can count longer, more specific phrases. However, this relationship between n and accuracy is also problematic because most n-grams will not exist in a corpus of human-generated text (or code). Therefore most n-grams would have a probability of zero for a large enough n. This issue can be addressed by using a smoothed model. Smoothing increases the accuracy of the model by estimating the probability of unseen n-grams from the probabilities of the largest m-grams (where $m < n$) that the n-gram consists of and that exist in the corpus. For example, if the corpus does not contain the phrase "I'm a little teapot" but it does contain the phrases "I'm a" and "little teapot" it would estimate the probability of "I'm a little teapot" using a function of the two probabilities it does know. In UnnaturalCode, however, the entropy is measured in bits. Entropy in bits is simply $S = -\log_2(p)$, where p is the probability. The higher the entropy, therefore, the lower the probability and the more surprising a token or sequence of tokens is.

Based on the entropy equation, as probability approaches 0, entropy approaches negative infinity. An uncounted n-gram could exist, which would have 0 probability effectively cancelling out all the other n-grams. Thus we rely on smoothing to address unseen n-grams.

[2]The modified `MITLM` package used in this paper is available at https://github.com/orezpraw/MIT-Language-Modeling-Toolkit

UnnaturalCode uses Modified Kneser-Ney smoothing as implemented by `MITLM`, the MIT Language Model package [3]. Modified Kneser-Ney smoothing is widely regarded as a good choice for a general smoothing algorithm. This smoothing method discounts the probability of n-grams based on how many m-grams (where $m < n$) it must use to estimate their probability. For example, a 7-gram who's probability is estimated from a 3- and a 4-gram will not be discounted as heavily as one whose probability is estimated from seven 1-grams. Modified Kneser-Ney smoothing is tunable: a parameter may be set for each discount. In UnnaturalCode these parameters are not modified from their default values in `MITLM`.

While `MITLM` and the entropy estimation techniques implemented within `MITLM` were designed for natural-language text, UnnaturalCode employs those techniques on code. Hindle *et al.* [2] have shown that code has an even lower entropy per token than English text does per word, as shown in Figure 2. That is to say, the same techniques that work for natural English language texts work even better on source code. Moreover, syntactically invalid source code will often have a higher cross-entropy than compilable source code given a corpus of only syntactically valid source code. Therefore, defective source code looks *unnatural* to a natural language model trained on compilable source.

Previous publications addressing this issue fall into two categories: *parser-based* or *type-based*. Burke's parse action deferral [4] is a *parser-based* technique that backs the parser down the parse stack when an error is encountered and then discards problematic tokens. Graham *et al.* [5] implemented a system that combined a number of heuristic and cost-based approaches including prioritizing which production rules will be resumed. Many modifications for particular parser algorithms have also been proposed to attempt to suppress spurious parse errors by repairing or resuming the parse after an error. Recent examples can be found by Kim *et al.* [6] who apply the k-nearest neighbour algorithm to search for repairs, or Corchuelo *et al.* [7] who present a modification that can be applied to parser generators and does not require user interaction. Other researchers have focused on *type-based* static analysis such as Heeren's Ph.D. thesis [8] which suggests implementing a constraint-based framework inside the compiler. Lerner *et al.* [9] use a corpus of compilable software to improve type error messages for statically typed languages. There have also been heuristic analyzers that work alongside the parser such as the one presented in Hristova *et al.* [10], but this approach is limited to predefined heuristic rules addressing a specific

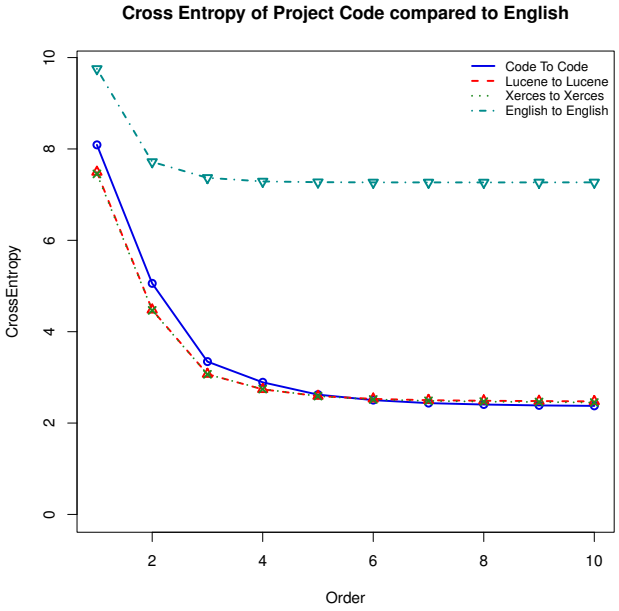

Figure 2. From Hindle *et al.* [2], a comparison of cross-entropy for English text and source code vs gram size, showing that English has much higher cross-entropy than code.

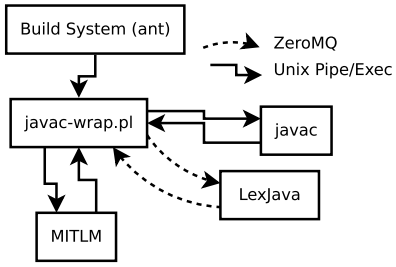

Figure 3. Data Flow of the Syntax-Error Detector UnnaturalCode.

selection of common mistakes. In comparison, UnnaturalCode requires no additional information about a language other than its lexemes, which may be extracted from its implementation or specification. The implementation presented here differs from those works in that it does not attempt to parse the source code. There is also a more recent publication by Weimer *et al.* [11] that uses Genetic Algorithms to mutate parse trees in an attempt to fix defects, but this requires syntactically valid source code.

III. A Prototype Implementation of UnnaturalCode

UnnaturalCode is designed to assist the programmer by locating the coding mistakes that caused a failed compilation. To this end, it only provides suggestions if the compile fails. If the compile succeeds it adds the error-free code to its corpus automatically. The data-flow diagram of UnnaturalCode is depicted in Figure 3.

The central component of the system is `javac-wrap.pl`, a wrapper script that is inserted in between the build system and `javac`. Our tests were run with the `ant` build system and the Oracle 1.7.1 JDK `javac`. Unfortunately, the package's `build.xml` file, which instructs `ant` how to build the package, must be modified to call `javac-wrap.pl` instead of merely instantiating the Java compiler classes from within `ant`. Furthermore, `javac` is called with a list of all java files to be compiled in a single go. This list often contains hundreds of source files; therefore, the first thing that `javac-wrap.pl` must do in the case of a failed compilation is to locate the file that failed to compile. It does this using heuristic regular expressions on `javac`'s error output.

Once `javac-wrap.pl` detects a failed compilation and locates the offending file, it must lexically analyze that file. No parsing is done, only lexical analysis is performed. In UnnaturalCode, comments are first removed from the input and then lexical analysis is performed by an ANTLR-produced Java lexical analyzer called `LexJavaMQ`.

In the case of a failed compilation, once `LexJavaMQ` returns the lexically analyzed source back to `javac-wrap.pl`, `javac-wrap.pl` starts an instance of `MITLM` that it communicates with via standard Unix pipes. It then sends `MITLM` a series of queries that are sequences of tokens from the lexically analyzed source file. Queries are generated by sliding a window of length $c \cdot n$ tokens. Therefore, there are at most $l - c \cdot n$ queries for a single file, where n is the gram size of the n-gram model being used in `MITLM`, and c is the chunk size used by `javac-wrap.pl`.

The prototype implementation of UnnaturalCode uses a modified version of the `MITLM` package that was first modified for use in Hindle *et al.* [2] and then further modified for the system presented here. It has been modified to compute the entropy of a small sample of text in relation to a corpus. It has also been modified to do so without requiring a restart and for higher performance.

Once `javac-wrap.pl` has finished computing results for all the queries, it ranks them by the entropy per token of each query, as in Figure 4. It then reports the top five strings with the highest entropy to the user as suggestions of where to look for mistakes in the code. The entropy is a measure of how unlikely the presented string of $c \cdot n$ tokens is, given the corpus. In this case $c = 2$ and $n = 10$. These constants were chosen for their high performance after some initial, informal testing. The entropy, S, is calculated by `MITLM` in bits as the negative logarithm of the probability. The higher the entropy score, the less likely a given string of tokens was found to be by `MITLM`. Figure 2 shows that entropy per token values are typically between 2 and 3 bits, compared to English text which typically has entropy near 7 bits per word.

The current implementation is fast enough to be used by a software engineer. Building the corpus takes much less time than compiling the same code, because only lexical analysis is performed. For the Lucene 4.0.0 corpus, results for a broken compile, in the form of suggestions, are found for a source file with over 1000 tokens in under 0.02 seconds on an Intel i7-3770 running Ubuntu 12.10 using only a single core and under 400MiB of memory. This is more than fast enough to be used interactively if `MITLM` is already running. However,

Check near == NO_MORE_DOCS) lastDoc = NO_MORE_DOCS; return
With entropy 4.552985
Check near () == NO_MORE_DOCS) lastDoc = NO_MORE_DOCS
With entropy 4.498802
Check near NO_MORE_DOCS) lastDoc = NO_MORE_DOCS; return;
With entropy 4.244520
Check near) lastDoc = NO_MORE_DOCS; return; }
With entropy 4.183379
Check near) == NO_MORE_DOCS) lastDoc = NO_MORE_DOCS;
With entropy 3.858807

Figure 4. Example of UnnaturalCode output showing an accurate error location from the motivational example.

n-gram order	10
chunk size in tokens	20
Files in Lucene 4.0.0	2 866
Files in Lucene 4.1.0	2 916
Files in Ant 1.7.0	1 113
Files in Ant 1.8.4	1 196
Files in Xerces Java 2.9.1	716
Files in Xerces Java 2.10.0	754
Files in Xerces Java 2.11.0	757
Lucene Diffs considered	827
Lucene Hunks >25 tokens	6 535
Lucene Total hunks and files	12 317
Types of mutation	3
Mutations per type per file/hunk	500
Feasibility Tests using Lucene 4.0.0 corpus	20 269 500
Feasibility Tests using Ant 1.7.0 corpus	3 463 500
Feasibility Tests using Xerces Java 2.9.1 corpus	3 340 500
Integration Tests using Lucene 4.0.0 corpus	2 626 940
Integration Tests using Ant 1.7.0 corpus	833 960
Integration Tests using XercesJ 2.9.1 corpus	1 547 180
Total Tests Performed	32 054 580

Table I
VALIDATION DATA SUMMARY STATISTICS

`MITLM` start-up can be quite slow depending on the size of the corpus. For the Lucene 4.0.0 corpus `MITLM` takes about 5 seconds to start on the same platform.

IV. FEASIBILITY VALIDATION METHOD

UnnaturalCode was tested primarily on three Apache Foundation projects: Lucene,[3] Ant,[4] and XercesJ.[5] UnnaturalCode was tested against two revisions of Lucene, which is a fast document-search engine intended for use in websites. Lucene is quite large, containing over 2800 individual Java source files.

For each project the training corpus consisted of every Java source file from the oldest version of the project tested (Lucene 4.0.0, Ant 1.7.0, XercesJ 2.9.1). These source files were compiled and added to the training corpus and this corpus was used for the duration of testing. No automatic updates to the training corpus were performed during testing. UnnaturalCode was also tested against the contiguous parts (hunks) of the diffs of the revisions of Lucene between 4.0.0 and 4.1.0. These hunks represent the changes that a product sees over time. Three different types of tests on four different kinds of input source file were performed.

[3]Apache Lucene is available at https://lucene.apache.org/
[4]Apache Ant is available at https://ant.apache.org/
[5]Apache XercesJ is available at https://xerces.apache.org/xerces-j/

The following mutations were applied to files and relevant *diff hunks* (contiguous lines added in a patch):

- **Random Deletion:** a token was chosen at random from the input source file and deleted. The file was then run through the querying and ranking process to determine where the first result with adjacent code appeared in the suggestions.
- **Random Replacement:** a token was chosen at random and replaced with the token "XXXXXXX."
- **Random Insertion:** a location in the source file was chosen at random and the token "XXXXXXX" was inserted there.

The modified code is never tested to see if it actually fails to compile, it is simply assumed to fail to compile. Each of these three tests were repeated on each input file 500 times, each time modifying a newly and randomly chosen location in the source file. For Lucene, all 3 tests were performed 500 times each on over 13,513 files and hunks. Thus, a total of over 20 million data points were collected as shown in Table I. 3 million tests were run on each Ant and XercesJ as well.

For Lucene, 4 different kinds of source-code inputs were tested. First, for the Lucene 4.0.0 test, source files were taken from the same exact package as the corpus and were modified by the above process and then tested. These source files exist unmodified in the corpus. Second, source files were taken from the next Lucene release, the 4.1.0 version, that had been modified by developers. Some of these source files exist in their 4.0.0 form in the corpus, but have been modified by developers and then by the above process. These files are listed in the results as the "Lucene 4.1.0 – Changed Files" test. Additionally, new source files were added to Lucene after the 4.0.0 release for 4.1.0. These new files do not exist in the corpus but are related to files which did. These are listed in the results as the "Lucene 4.1.0 – New Files" test. Finally, to test files completely external to the corpus, Java source files from Apache Ant 1.8.4 were tested. Not only do these files not exist in the corpus but they are not related to the files that do, except in that they are both Apache Foundation software packages.

In order to get the above results, the following steps were performed. First a corpus was created from the earliest release. This necessitates modifying Lucene's `build.xml` file in order to instruct `ant` to use a new `javac` wrapper to perform compilation steps instead of instantiating the compiler directly. Then, Lucene 4.0.0 was built, automatically adding all compilable source files to the corpus.

For Ant 1.8.4, Lucene 4.0.0, and Lucene 4.1.0, all of the Java source files were collected. For the diff test, diffs from the Lucene Subversion repository trunk between versions 4.0.0 and 4.1.0 were collected. These revisions span a range of dates from October 10th, 2012 to January 21st, 2013. Then these were broken into hunks of at least 25 tokens. Diff hunks less than 25 tokens were discarded, because there are only 5 possible results for a query on 25 tokens.

For Ant and XercesJ all Java source files from their milestone tarballs were used. The same procedure as used for the

Sources Tested	Corpus	Delete	Insert	Replace
Lucene 4.0.0	Lucene 4.0.0	.95	.94	.94
Lucene 4.1.0	Lucene 4.0.0	.86	.83	.83
Diff Hunks	Lucene 4.0.0	.69	.61	.69
Ant 1.8.4	Lucene 4.0.0	.24	.21	.23
Lucene 4.1.0 Only new files	Lucene 4.0.0	.31	.25	.26
Lucene 4.1.0 Only changed files	Lucene 4.0.0	.72	.69	.68
Ant 1.7.0	Ant 1.7.0	.95	.92	.91
Ant 1.8.4	Ant 1.7.0	.65	.60	.60
XercesJ 2.9.1	XercesJ 2.9.1	.97	.95	.95
XercesJ 2.10.0	XercesJ 2.9.1	.77	.72	.73
XercesJ 2.11.0	XercesJ 2.9.1	.76	.71	.71

Table II
CUMULATIVE MEAN RECIPROCAL RANKS (MEAN MRR)

Lucene milestones was used.

Then, for each file (or diff hunk) in these input sets, `javac-wrap.pl` was called in testing mode. In this mode, it lexically analyzes the file once and then retrieves query results for the presumed compilable input. Next, it runs 500 query tests. In each test it chooses a random token in the input file to mutate as described above, and queries `MITLM` with the new, mutated, file, and compares the query results to the query results obtained for the original, compilable file. For each test it records the rank of the first correct result, r_q, which is the first result that contains tokens adjacent to the location of the mutation and has a higher entropy than the result at the same rank from the original file.

The rankings are analyzed statistically using the reciprocal rank. The mean is reported as the mean reciprocal rank (MRR) [12]:

$$\mu = \frac{1}{|\mathcal{Q}|}\left[\sum_{q\in\mathcal{Q}}\frac{1}{r_q}\right].$$

$\mathcal{Q}$ is the set of all queries, and q is an individual query from that set. For example, $|\mathcal{Q}| = 500$ for an individual file and type of mutation. Using the MRR has several advantages: it differentiates the most among the first few (highest) ranks, and has a worst value of 0, whereas for UnnaturalCode the worst possible absolute rank depends on the length of the input file. This is important because the more results a programmer has to consider to find a syntax error, the less likely he or she is to consider them all.

MRR is a very unforgiving measure of the performance of a system that returns multiple sorted results. In order to achieve an MRR greater than 0.75, the correct result must be the first result presented to the user most of the time. For example, consider a random mutation of each type on a single file. If the correct result was ranked first for the deletion, second for the insertion, and third for the replacement, UnnaturalCode would only have achieved an MRR score of 0.61 for that file.

V. FEASIBILITY VALIDATION RESULTS

Figure 5 shows the distributions of the MRR scores of the files of versions of Lucene and Ant versus a Lucene trained corpus. The wider the shaded area is in these charts, the more

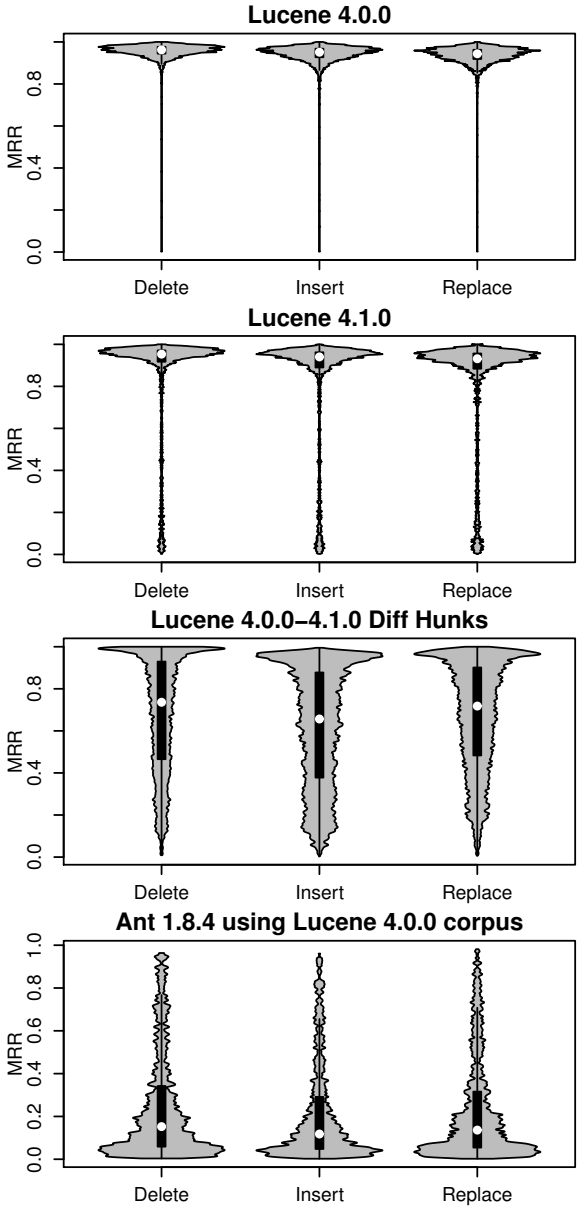

Figure 5. MRR Distributions of the files/hunks of Lucene 4.0.0, 4.1.0 and files of Ant 1.8.4 tested against a Lucene 4.0.0 corpus.

files had that MRR. These plots also show the 25th, 50th and 75th percentiles as the beginning of the black box, the white dot, and the end of the black box in the centre. Table II presents the cumulative MRRs for each data set and method.

UnnaturalCode performs very well at detecting mutations in code that it is familiar with. UnnaturalCode did very well with only the first Lucene 4.0.0 version in the corpus when tested against both Lucene 4.0.0 and Lucene 4.1.0. The MRR scores for diff hunks are more variable, however, and tend to be lower than the scores for the file to file comparisons. These diff hunks come entirely from changes made in between the two milestone releases. A test of Ant 1.8.4 against a foreign corpus (Lucene 4.0.0) results in poor performance.

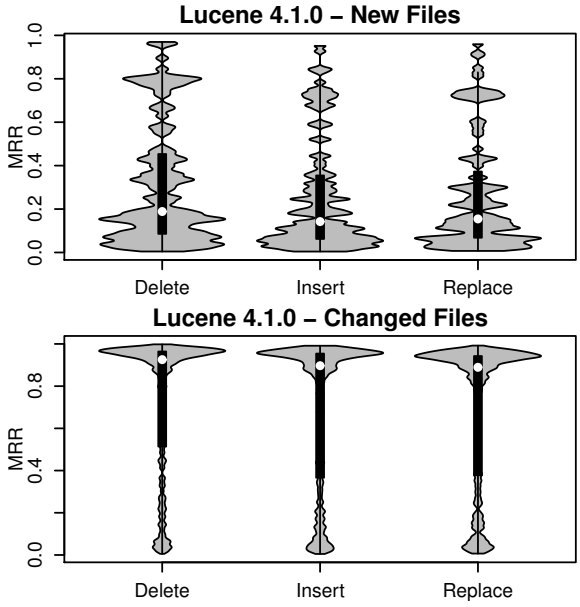

Figure 6. MRR Distributions of only new and changed files from Lucene 4.1.0

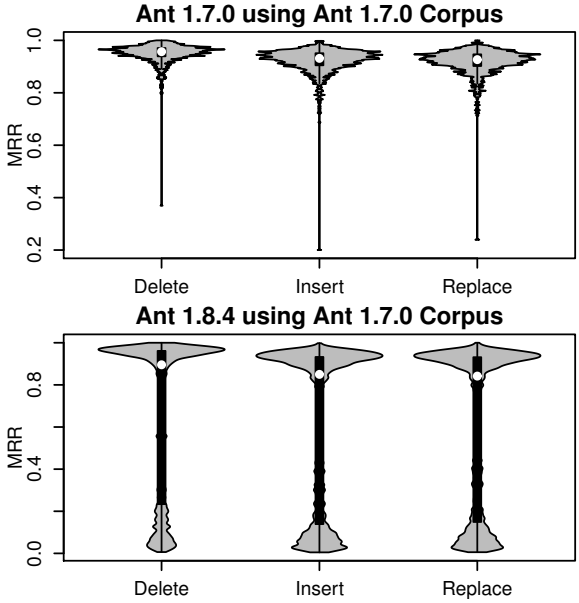

Figure 7. MRR Distributions of the files of Ant 1.7.0 and Ant 1.8.4 tested against the Ant 1.7.0 corpus.

Syntax error detection performance is best with a corpus trained on the same or earlier version of the system.

The scores and chart for the whole of Lucene 4.1.0 is not the entire story for that version. It contains three kinds of files: files unchanged from 4.0.0, changed files, and new files added since 4.0.0. Figure 6 clearly shows how these types files bring the MRR scores for 4.1.0 down from the scores for 4.0.0. The newly added files have very inconsistent performance with MRR scores near those of the scores for Ant's unrelated files, despite the fact that they are a part of the same project as the training corpus. Similarly, files changed for 4.1.0 present similar scores as diff hunks, even though they contain unmodified code as well as modifications. In comparison, the diff hunks only contain the changed and added lines. However, files changed have a much higher median MRR than diff hunks, implying that the correct result appears first in the results very often for this case.

Figure 7 compares the performance of Ant 1.7.0 versus itself and Ant 1.8.4. The MRR behaviour is similar to the Lucene plots in Figure 5 for Lucene versus Lucene tests, and the poor performance of Ant versus Lucene has been negated by using Ant code in the corpus. The median remains very high, implying that UnnaturalCode scores very well on most files.

Figure 8 tests 3 consecutive major releases of XercesJ against a corpus of XercesJ 2.9.1. In all cases the median MRR of these tests are above 0.8, which means that over 50% of the files have an MRR greater than 0.8. As with Ant, in the test on a past corpus (XercesJ 2.10.0 and 2.11.0) the bottom quartile extends further, but ends before an MRR score of 0.4, which implies that for 75% of the files tested, UnnaturalCode performed well.

Figure 9 investigates the relationship between file size and

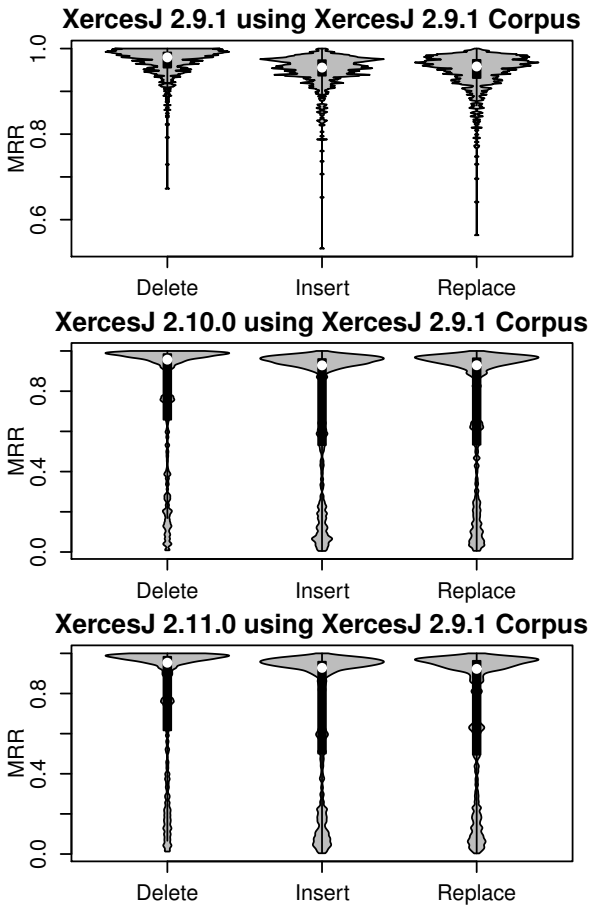

Figure 8. MRR Distributions of the files of XercesJ 2.9.1, 2.10.0, and 2.11.0 tested against the XercesJ 2.9.1 corpus.

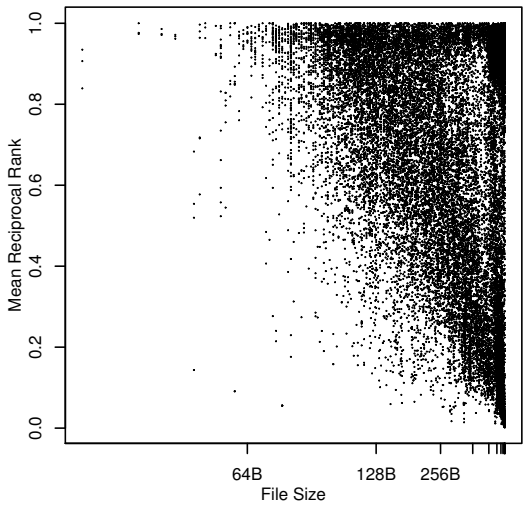

Figure 9. MRR vs file size plot showing how low file size can prevent a low MRR

minimum MRR. In this figure each dot represents the MRR of a single type of mutation on a single file or hunk that was used for the validation, and therefore each dot represents a mean of 500 reciprocal ranks. It clearly shows that UnnaturalCode is very likely to produce the correct result at a high rank when used on small files, confirming the intuition that errors are easier to find in smaller files.

An interesting question is whether there was a trend in the MRR of revisions over time. Would newer revisions have lower MRR scores because of a loss of context? However in an evaluation of UnnaturalCode on diff hunks, as Figure 11 shows, no such pattern is present in the data. The per-revision variance in MRR is quite high. While MRR averaged over each revision is 0.60, the standard deviation is 0.18.

Newer revisions seemed unaffected by the missing context of their past revision in the corpus (Figure 11).

VI. Compiler Integration Validation Method

The evaluation of the performance of UnnaturalCode when integrated with the compiler as a part of the compilation process repeats much of the same process as the feasibility validation. However, there are some important changes.

Test files were mutated in much the same way as the feasibility validation method. However, to provide more realistic results, insertion and replacement tests used tokens randomly chosen from the same file, instead of "XXXXXXX." So, for any single replacement or insertion mutation, the inserted or replaced token might be a brace, identifier, or other valid Java token.

The second important change is that these mutant files were actually compiled, and when compilation succeeded the mutant file was skipped. This had particularly dramatic results on the deletion tests, where 33% of the random token deletions resulted in a file that still compiled.

		Interleaving Pattern			
Sources Tested	Corpus	UUUU	JJJJ	JUJU	UJUJ
Lucene 4.0.0	Lucene 4.0.0	.950	.932	.959	.963
Lucene 4.1.0	Lucene 4.0.0	.865	.937	.960	.914
Ant 1.7.0	Ant 1.7.0	.940	.927	.956	.958
Ant 1.8.4	Ant 1.7.0	.681	.923	.945	.806
Ant 1.8.4	Lucene 4.0.0	.308	.921	.930	.600
XercesJ 2.9.1	XercesJ 2.9.1	.939	.895	.937	.955
XercesJ 2.10.0	XercesJ 2.9.1	.694	.889	.916	.796
XercesJ 2.11.0	XercesJ 2.9.1	.688	.884	.911	.791

Table III
Compiler Integration Mean Reciprocal Ranks (MRRs)

Third, the compiler's own error messages were considered. These error messages were also given an MRR score for each file. The compiler was scored in a similar fashion to UnnaturalCode: the first result produced by the compiler mentioning the correct line number was considered correct.

The fourth important change is that MRR scoring was implemented for two different sets of interleaved JavaC and UnnaturalCode results. These combined results consist of a JavaC result followed by an UnnaturalCode result, followed by a JavaC result, and so on. The two variations are: 1) returning a JavaC result first; and 2) returning an UnnaturalCode result first. These combined results represent the intended use of UnnaturalCode as a way to augment compiler errors. Finally, some bugs which negatively affected the accuracy of UnnaturalCode were fixed.

VII. Compiler Integration Validation Results

Table III shows the results of the compiler integration tests and Figure 10 shows the MRR distribution for one of these tests. In this table, the column heading describes the output interleaving pattern: "UUUU" gives MRR means for UnnaturalCode results, "JJJJ" gives MRR means for JavaC, "JUJU" gives MRR means for interleaved results with JavaC's first result first in the output, and "UJUJ" gives MRR means for interleaved results starting with UnnaturalCode's first result. All four plots come from the same set of randomly chosen mutations. Both JavaC and UnnaturalCode perform well on their own. UnnaturalCode performs worse than JavaC on the deletion test and better than JavaC on the insertion and replacement tests. However, interleaved results perform better than either system by itself: the best performing interleave depends on the file. Token deletions become more difficult for UnnaturalCode to locate when compilable deletions are not considered.

In the XercesJ tests, both UnnaturalCode and the Java compiler performed worse overall. However, the interleaved results still dramatically improve performance. Interleaving allows the XercesJ results to approach the performance acheived with Ant and Lucene.

VIII. Discussion

The n-gram language-model approach was capable of detecting all mutations: inserted tokens, missing tokens, and

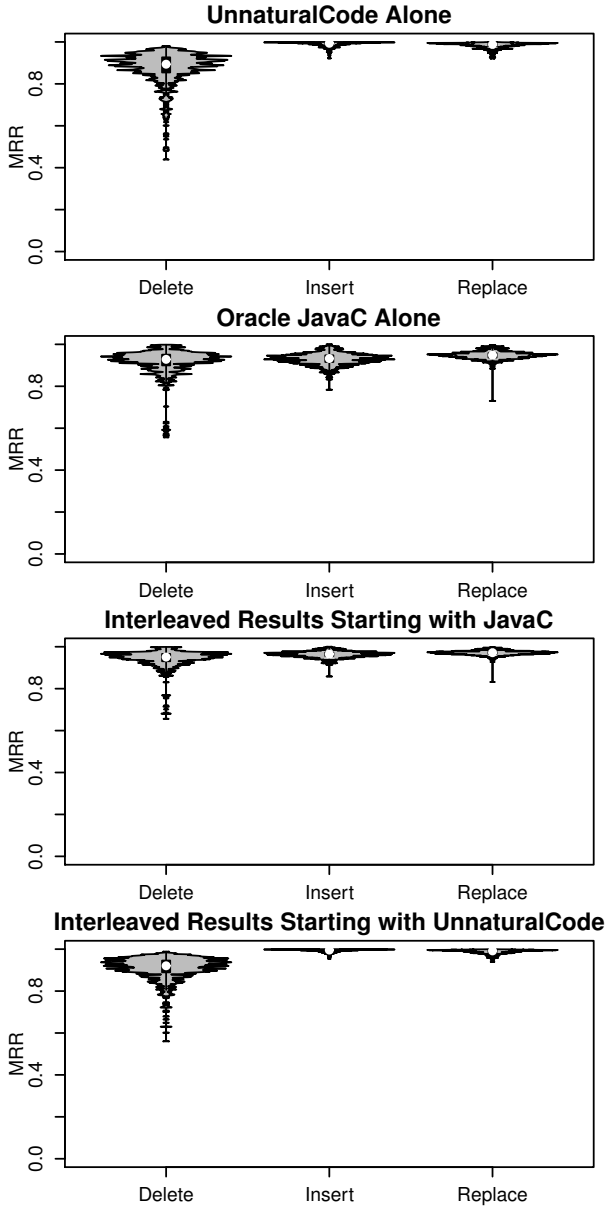

Figure 10. MRR Distributions of the files of Lucene 4.0.0 for the compiler integration test.

replaced tokens. This is because the sequence of tokens will not have been seen before by the language model, assuming it has been trained on compilable code.

A. Performance on Milestones

In the milestone tests, some files would get consistently wrong results because the top results would always be the same regardless of where the changes were made. In particular, files from outside the corpus that contained strings of new identifier tokens would consistently produce poor results with UnnaturalCode.

Unfortunately, with only a single project in the corpus, performance was sometimes very poor. This poor performance could be easily triggered by adding new identifiers that were not present in the corpus, since those new identifiers were labelled with high entropy by the model. Sometimes this behaviour is accurate, as in the case of a misspelled identifier, but sometimes it is inaccurate, as in the case of a newly added, but correctly spelled, identifier.

These results are highly encouraging, however. Even the worst MRR score of 0.21 implies that there are files in Apache Ant in which errors are locatable using only a corpus trained on Lucene. If one runs the tests as described above on broken Java code that imports classes from packages that do not exist in the training corpus, these correctly specified imports will almost always be the top five results. However, this behaviour only persists until they are added to the corpus, and this happens after the first successful compilation of those files. For example, even though class imports in Ant 1.8.4 share the same `org.apache` class path, they also contain new identifiers. When `ProjectHelperImpl.java` imports `org.apache.tools.ant.Target`, this string contains three identifiers, "tools," "ant," and "Target" which are not in the training corpus. This will cause the average entropy of any string of 20 tokens containing "`tools.ant.Target`" to contain at least three unseen tokens. In comparison, when the code is mutated for testing, at most one additional unseen token is introduced. One possible solution to the new identifier problem is to ignore identifier content and train on the type of the identifier lexeme itself.

> The syntax error detector can often misreport new identifiers, such as package names, as syntax errors.

B. Performance on Revisions

Figure 11 indicates that some revisions have much higher MRR than others. An examination of some of these changes reveals what may have caused their score to be particularly high or low.

Lucene revision 1408367 had a particularly low mean MRR score of 0.15. In this revision there was no Java code change hunk larger than the 25-token minimum in UnnaturalCode, and the remaining large hunks were English language comments. These comments were not removed because `diff` had removed the leading `/*`. Another low-scoring revision, number 1419892, with a mean MRR of 0.12, contained a hunk with a fragment of the Apache 2.0 License.

Thus, the actual performance on Java code from the diff hunks is likely higher than the mean presented in Table II because that result includes hunks of English text. The validation framework failed to remove these English texts from the hunks it considered. Sometimes diffs lack the context to resolve their content. Comments are often modified such that the code signalling the beginning of the comment is not recorded in the diff.

There are also high-MRR outliers in the set of revisions considered, such as revision number 1425817, in which the Lucene developers added new variables, "`CONN_LOSS`," and "`EXP`." However these new variables were added with common Java attributes and types: "`private static final boolean`." This common sequence occurs 20 times in the

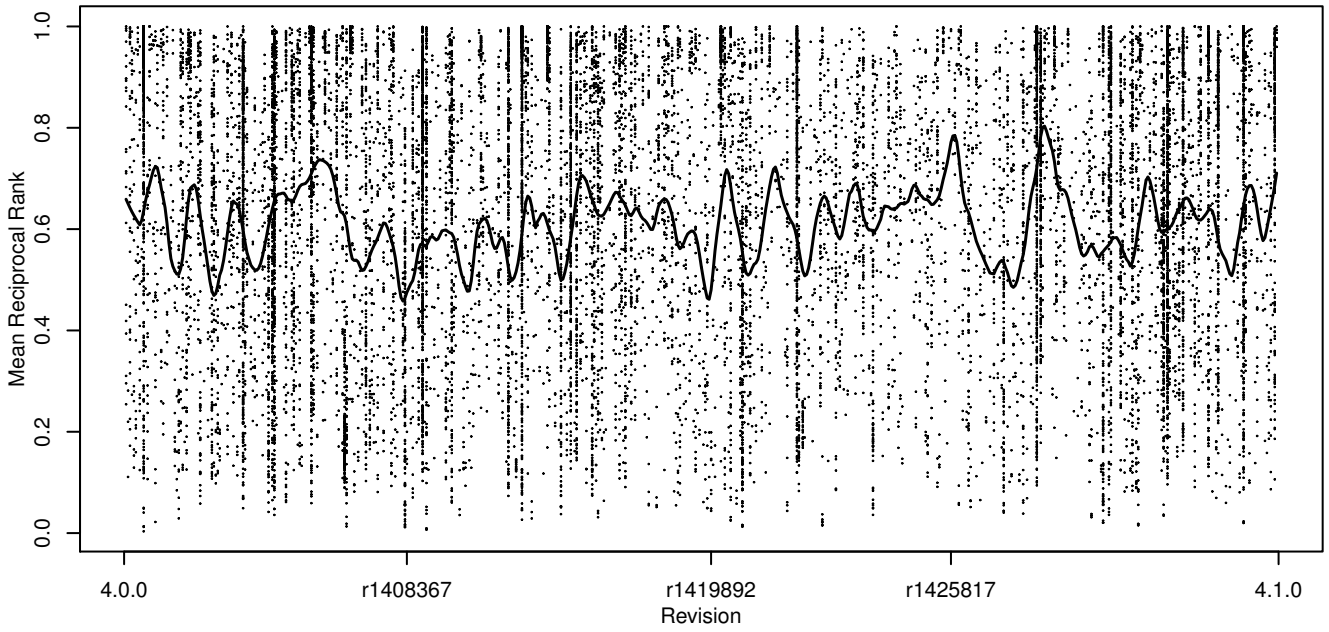

Figure 11. Per-hunk-and-mutation-type MRRs and per-diff mean MRRs of diff revisions over time for Lucene 4.0.0 to 4.1.0

training corpus. Therefore, they help keep the entropy of token sequences containing the new declarations low by averaging out the high entropy of "CONN_LOSS," and "EXP," that do not occur in the training corpus anywhere.

C. Performance with Compiler Integration

> UnnaturalCode performs comparably to JavaC despite not knowing the syntax of the Java language.

It achieves this performance while having only a single, related, compiling program code in its corpus. Additionally, UnnaturalCode was capable of augmenting the accuracy, in terms of MRR score, of the compiler itself when its results were interleaved with the compiler's own results. The interleaving improved the MRR score from .915, using just JavaC, to 0.943 using interleaved results for the deletion test on Lucene 4.0.0. Intuitively, this means that 66% of the time when JavaC's highly ranked results are wrong, UnnaturalCode has a highly ranked result that is correct. Results are presented to the user side-by-side because the best interleave choice is not consistent.

IX. Threats to Validity

Construct validity is affected by the assumption that single-token mutation is representative of syntax errors. This assumption may not be very representative of real-world changes a software engineer would make between compilation attempts. However, single-token mutations are the worst-case scenario and provide a lower bound on performance because any additional entropy they generate is averaged over $c \cdot n = 20$ tokens. Multi-token mutations would be easier for UnnaturalCode to detect.

Internal validity is hampered by using the MRR formula to score our ranking of the correct query results. These rankings are affected by the maximum gram length, n, where typically larger is better, and the number of total results. The number of total results is

$$\frac{l - c \cdot n}{s}$$

where l is the the length of the file, c is the number of n-grams considered at once, and s is the step size. Since correct results comprise at most $\frac{c \cdot n}{s}$ of the results, the chance of the correct result appearing in the top 5 if the results are sorted randomly is approximately proportional to $1/l$, as shown in Figure 9. In other words, the system will naturally perform better on short input files simply because there is less noise to consider. This is the reason diff hunks with less than 25 tokens are discarded. Since $\frac{25-2 \cdot 5}{1} = 5$, there are only 5 total query results and the correct one is always in the top 5. The revision tests can accidentally run on context-less Java comments, when the comment tokens ($/\star$, $\star/$ or $//$) are missing, which can have some effect on the MRR results.

External validity is affected by the choice of Java projects. The experimental evaluation covers 27 million tests of UnnaturalCode across 3 different medium-to-large Apache Foundation Java projects: Ant, XercesJ and Lucene. In addition, the evaluation ran 5 million tests with both UnnaturalCode and JavaC. JavaC compilation is much slower than querying UnnaturalCode alone.

Java has a syntax typical of many C-like languages. Thus, these tests are a good representative, and the results for these projects will generalize well to other projects. These results should generalize to other languages as well but this

assumption has not been tested yet. The evaluation on 32 million single-token mutation tests across 3 distinct corpuses is a fairly significant evaluation of the new method.

X. Future Directions

This technology can be very useful to software engineers who are actively developing a piece of software. All the engineer must do is instruct their build system to call the wrapper instead of the compiler directly, and the system will begin building a corpus; if the compile fails it will return side-by-side ranked results from both the compiler and from UnnaturalCode. The performance evaluation indicates that a system using the ideas presented in this paper will suggest the location of the fault in the top two results often enough to be useful as long as the corpus contains one successful compile of the same project.

The method proposed should be implemented not only as a compiler wrapper for general purpose command-line use, but also as a plug-in for an integrated development environment (IDE) such as Eclipse. In such an environment it could provide immediate and visual feedback after a failed compile of which lines were likely to cause problems, perhaps by colouring the background with a colour corresponding to the increase in entropy for those lines.

The generalization to other languages needs to be evaluated. One simply needs to replace the Java lexical analyzer used in UnnaturalCode with a lexical analyzer for the language they wish to use and modify the compiler wrapper to locate input file arguments and detect the failure of their compiler of choice. The n-gram model is flexible and will work with many programming languages [2].

Several changes could be made to enhance performance. The first is to build an n-gram model consisting only of the lexical types of each token, but not the token itself. The language model corpus and input would then consist of words like "identifier," "operator," "type," instead of "readByte", "+," and "int." Combining entropy estimates from that model with the current model could produce more accurate results when considering never-before-seen tokens.

The effect of a multi-project corpus on syntax error detection should be investigated. The idea is to explore what makes a good corpus for general purpose syntax error detection.

Statistical information on what typical coding mistakes look like is missing from UnnaturalCode, as it can only characterize correct code. Therefore, software repositories could be mined for patches that fix coding mistakes and syntax errors. That data could then be used to characterize those mistakes and fixes statistically. Source code could then be examined for code resembling the broken code from those defective commits.

Several extensions to the system could be implemented. For example, the combination of this method with token prediction could automatically repair source code by statistically directing most-likely syntax repair searches based on a dynamic and project-specific corpus instead of statically defined least-cost repair searches such as those presented in Corchuelo *et al.* [7] and Kim *et al.* [6]. This approach should be much more efficient than the approach applied in Weimer *et al.* [11]. It may also be interesting for developers or project managers to be able to see the entropy of each line or token of source code themselves because the entropy may correlate with other properties of the source that have not yet been considered.

XI. Conclusion

This paper presents a system to address the common issue of poor syntax-error location reporting by modern compilers. The system avoids parsing completely, instead relying on a statistical language model from the field of natural language processing. The choice of model was guided by the promising results in code completion. UnnaturalCode consists of a compiler wrapper, a lexical analyzer, and a language model that cooperate and are more than fast enough to be used interactively. UnnaturalCode is tested by inserting, deleting and replacing random tokens in files from the software that it was trained on, the next version of the software it was trained on, commits between the two versions, and a completely independent software package. The experimental evaluation shows that UnnaturalCode performed very well on the same version, well on the next version, and poorly on external software. When UnnaturalCode's results are combined with the compilers own results, the correct location is reported earlier than with the compiler's results alone.

This work helps bridge the gap between statistical language analysis tools and computer-language parsers. Bridging this gap solves a problem that has haunted the users of parsers since they were invented: accurately reporting syntax errors. This new approach is unique in that it employs a statistical model: all previous work in this area are based on parser modifications to repair or recover from syntax errors, static code analysis, predefined heuristics, or type analysis. Furthermore, the model can evolve with the changing contexts of individual engineers and projects as it continuously updates its training corpus.

> The n-gram language-model is capable of enhancing the compiler's ability to locate missing tokens, extra tokens, and replaced tokens.

Finally, when you build a language model of working source code, "syntax errors just aren't natural."

Acknowledgements

Abram Hindle is supported by an NSERC Discovery Grant. Joshua Charles Campbell is supported by a scholarship from the University of Alberta.

REFERENCES

[1] 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.

[2] A. Hindle, E. Barr, Z. Su, M. Gabel, and P. Devanbu, "On the naturalness of software," in *Software Engineering (ICSE), 2012 34th International Conference on*, June 2012, pp. 837–847.

[3] B. Hsu and J. Glass, "Iterative language model estimation: efficient data structure & algorithms," 2008.

[4] M. G. Burke and G. A. Fisher, "A practical method for LR and LL syntactic error diagnosis and recovery," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 9, no. 2, pp. 164–197, Mar. 1987.

[5] S. L. Graham, C. B. Haley, and W. N. Joy, "Practical LR error recovery," *SIGPLAN Not.*, vol. 14, no. 8, pp. 168–175, Aug. 1979.

[6] I.-S. Kim and K.-M. Choe, "Error repair with validation in LR-based parsing," *ACM Trans. Program. Lang. Syst.*, vol. 23, no. 4, pp. 451–471, Jul. 2001.

[7] R. Corchuelo, J. A. Pérez, A. Ruiz, and M. Toro, "Repairing syntax errors in LR parsers," *ACM Trans. Program. Lang. Syst.*, vol. 24, no. 6, pp. 698–710, Nov. 2002.

[8] B. J. Heeren, "Top quality type error messages," Ph.D. dissertation, Universiteit Utrecht, Nederlands, 2005.

[9] B. S. Lerner, M. Flower, D. Grossman, and C. Chambers, "Searching for type-error messages," in *Conference on Programming Language Design and Implementation (PLDI)*, San Diego, CA, USA, 2007, pp. 425–434.

[10] 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.

[11] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest, "Automatically finding patches using genetic programming," in *Proceedings of the 31st International Conference on Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 364–374.

[12] E. M. Voorhees *et al.*, "The TREC-8 question answering track report," in *Proceedings of TREC*, vol. 8, 1999, pp. 77–82.