Tidyparse: A Tool for Realtime Syntax Repair

Breandan Considine[1], Jin Guo[1], and Xujie Si[2]

[1] McGill University, Montréal, QC H2R 2Z4, Canada
{breandan.considine@mail, jguo@cs}.mcgill.ca
[2] University of Toronto, Toronto, ON, M5S 1A1 Canada
six@utoronto.ca

Abstract. We describe the implementation of a tool for real-time syntax correction in an IDE. Upon activation, our tool takes a syntactically invalid source code fragment around the caret position, and produces a small set of suggested repairs. We model the problem of syntax repair as a structured prediction task, whose goal is to generate the most likely valid repair in a small edit distance of the invalid code fragment.

Keywords: Error correction · CFL reachability · Langauge games.

1 Introduction

Syntax errors are a familiar nuisance for software developers. Whenever a syntax error is detected, the IDE typically flags the offending code fragment, but offers little guidance on how it should be fixed. The developer must inspect the code and manually apply the appropriate fix through a process of trial and error. This process can be distracting and time-consuming, especially for novice developers. In this paper, we describe a tool for automatic syntax repair in an IDE.

We propose a new approach to syntax repair and accompanying tool, called *Tidyparse* that suggests a small set of repairs to the user, which are guaranteed to be valid, minimal and natural. Our repair tool is a fusion of two widely available components: grammars and language models. At first glance, these two models are not obviously synergistic: the grammar is a deterministic, formal model of the language, while the language model is only an approximate generator of linguistic patterns. However, we show that by carefully integrating them, it is possible to generate repairs that are always correct and highly natural.

Language models are statistical models that generate natural sequences of text, however, these models make no guarantees about the validity of the generated text. Given a sequence of previous tokens, $\sigma_0, \ldots, \sigma_{n-1}$, an autoregressive language model outputs a distribution over the next most likely token, σ_n.

Almost every programming language ever developed is syntactically context-free, meaning the syntax of the language can be expressed as a context-free grammar (CFG). This grammar can be used to recognize the validity of a given input sequence, or force an autoregressive language model to generate only syntactically valid sequences by blocking out invalid tokens during inference.

Likewise, this grammar can be also used to construct a synthetic grammar, recognizing all and only valid sequences within a certain edit distance of a broken source code fragment. Our approach uses a pretrained language model to sample repair candidates from this synthetic grammar. We rank the results by negative log likelihood under the language model, and present the top k candidates to the user. The user can then select the most appropriate repair from the list.

Let us consider a simplified example. Suppose the user has just written the following code fragment: `v = df.iloc(5:, 2:)`. Given an alphabet of just a few dozen lexical tokens, this tiny statement has millions of possible two-token edits, yet only six of those possibilities are accepted by the Python parser:

(1) `v = df.iloc(5:, 2,)` (3) `v = df.iloc(5[:, 2:])` (5) `v = df.iloc[5:, 2:]`

(2) `v = df.iloc(5), 2()` (4) `v = df.iloc(5:, 2:)` (6) `v = df.iloc(5[:, 2])`

The color green denotes an insertion, orange is substitution and red is deletion. To generate these repairs, we start by lexicalizing the input sequence as follows:

```
v    = df   . iloc ( 5      : , 2      : )
NAME = NAME . NAME ( NUMBER : , NUMBER : )
```

Next, we construct an automaton that recognizes every string within a certain edit distance of the input. We will depict this process for a simpler language, where the grammar is $S \to$ `( )` $\mid$ `(` S `)` $\mid SS$ and the broken code is `( ) )`.

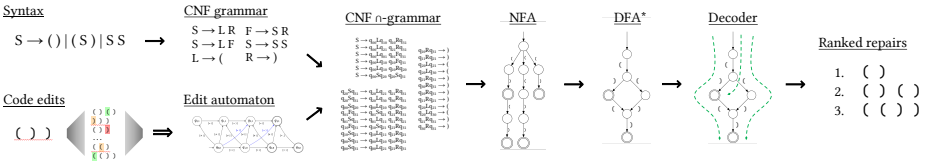


Fig. 1. Simplified dataflow. Given a grammar and broken code fragment, we create an automaton generating the language of small edits, then intersect it with the grammar to produce an intersection grammar, which can be simplified to a DFA and decoded.

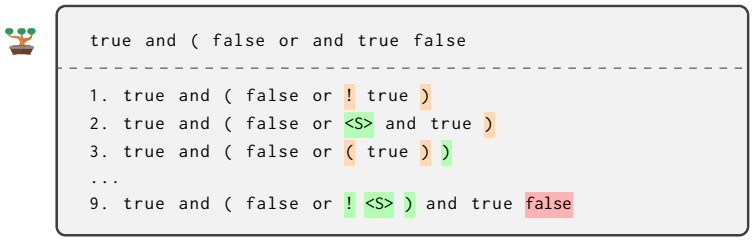
The programming language syntax is represented by a context-free grammar, which we first reduce the grammar to Chomsky normal form (CNF). Then, using a modified construction from Bar-Hillel [1], we construct an intersection grammar, which recognizes all and only valid sequences recognized by both the grammar and edit automaton. This grammar is known to be non-recursive, and can be simplified to a deterministic finite automaton (DFA) using standard techniques. Finally, we decode the DFA to produce a list of repair candidates, which are reranked by a scoring function, e.g., the log likelihood of a language model.

Now that we have a high-level overview of our approach, we will demonstrate a few of the capabilities of our tool by means of some usage examples.

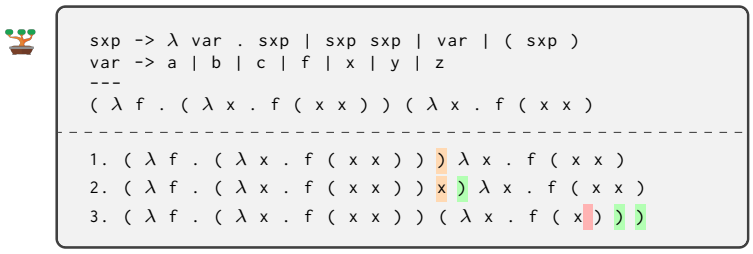
2 Usage examples

Tidyparse offers a convenient user interface featuring a text editor, a grammar editor and a parse tree viewer for interactive prototyping. All tokens are delimited by whitespace. For example, suppose we have the following grammar:

Syntax repair is the primary intended use case of our tool. If given an unparsable string, it will return an ordered set of suggestions how to fix it:



For simplicity, it is also possible to define a grammar and string side-by-side from the same editor, as shown in the untyped λ-calculus example below:



By default, Tidyparse samples the finite intersection language uniformly without replacement, then sorts the results by Levenshtein distance. Customizing the ranking order is possible using a programmatic interface. For the Python language, we use a small dataset of n-grams to rank the repairs by naturalness.

3 Related work

Many methods to sample sequences from grammars and language models have been proposed. Some of these guarantee that all samples will be grammatically valid. Others guarantee that all grammatically valid sentences are generable. The difficulty is not just generating valid sentences, but doing so in a parallel communication-free manner, without compromising soundness or completeness. The goal should be to massively parallelize a replacement-free discrete sampler.

In general, the problem of program induction from input-output examples is not well-posed, so specialized solvers that can make stronger assumptions will usually have an advantage on domain-specific benchmarks. Most existing

program synthesizers do not satisfy all of our desiderata, e.g., soundness, completeness, naturalness, and parallel sampling. We compare our approach against Seq2Parse and BIFI, the leading neural syntax repair models in the literature.

		Sound	Complete	Natural	Theory	Parallel	Tool
Repair	Tidyparse [2]	✓	✓	✓	$\text{CFG}_\cap$	✓	IDE-ready
	Seq2Parse [4]	✓[1]	✗	✓	CFG	✗	Python
	BIFI [8]	✗	✗	✓	Σ^*	✗	Python
	OrdinalFix [9]	✓	✗	✗	CFG/MJ	✗	Rust
	Don't panic! [3]	✓	✗	✗	CFG/LR	✗	Python
SyGuS	Outlines [6]	✓[1]	✓[1]	✓	CFG	✗	Python
	SynCode [5]	✓	✓	✓	CFG	✗	Python
	Guidance	✓	?	✓	CFG	✗	Python
	llama.cpp	✓	?	✓	CFG	✗	Python

More generally, there exists a broad spectrum of LLM-powered syntax-guided synthesis (SuGuS) tools that do not specifically target syntax repair, but nonetheless constrain the output of the model to only produce syntactically valid text. We mention the latter tools for their relevance to grammar-constrained decoding, and otherwise omit constrained decoding methods without a dedicated tool.

4 Evaluation

We use syntax errors and fixes from the Python language to validate our approach. Python code fragments are abstracted as a sequence of lexical tokens, homogenizing literals and identifiers, but retaining all other keywords. Accuracy is evaluated by checking for lexical equivalence with the ground-truth human repair, following Sakkas et al. (2022) [4]. Specifically, we use the Precision@k statistic, which counts how often the true repair is contained within the top-k results. Given a repair model, $R : \Sigma^* \times \mathbb{N} \to \Sigma^*$ and a test set $\mathcal{D}_{\text{test}}$ of pairwise-aligned errors ($\sigma^\dagger$) and fixes (σ'), we define Precision@k as:

$$\text{Precision@k}(R) = \frac{1}{|\mathcal{D}_{\text{test}}|} \sum_{\langle \sigma^\dagger, \sigma' \rangle \in \mathcal{D}_{\text{test}}} \mathbb{1}\left[\sigma' \in \left\{R(\sigma^\dagger, i)\right\}_{i=0}^{k}\right] \tag{1}$$

We compare our method against two external baselines, Seq2Parse and Break-It-Fix-It (BIFI) [8] on a single test set. This dataset [7] consists of 20k naturally-occurring pairs of Python errors and their corresponding human fixes from StackOverflow, and is used to compare the precision of each method at blind recovery of the ground truth repair across varying edit distances, snippet lengths and latency cutoffs. We preprocess all source code by filtering for broken-fixed snippet pairs shorter than 80 tokens and fewer than five Levenshtein edits apart, whose broken and fixed form is accepted and rejected, respectively, by the Python 3.8.11 parser. We then balance the dataset by sampling an equal number of repairs from each length interval and Levenshtein edit distance.

[1] Claimed by authors, but published tool is known to exhibit counterexamples.

The Seq2Parse and BIFI experiments were conducted on a single Nvidia V100 GPU with 32 GB of RAM. For Seq2Parse, we use the default pretrained model provided in commit `7ae0681`[3]. Since it was unclear how to extract multiple repairs, we only measure Precision@1. For BIFI, we use the Round 2 breaker and fixer from commit `ee2a68c`[4], the highest-performing model reported by the authors, with a variable-width beam search to control the number of predictions, and let the BIFI fixer model predict the top-k repairs, for $k = \{1, 5, 10, 2 \times 10^4\}$.

The Tidyparse experiments were conducted on a server containing 40 Intel Skylake cores running at 2.4 GHz, with 150 GB of RAM, executing bytecode compiled for JVM 17.0.2. For the decoder, we use an order-5 Markov chain trained on 55 million BIFI tokens, which takes roughly 10 minutes. Sequences are scored using negative log likelihood with Laplace smoothing and our evaluation measures the Precision@{1, 5, 10, All} for varying latency cutoffs up to 90s, although the decoder often exhausts the search space and halts before timeout.

4.1 StackOverflow evaluation

For our first experiment, we measure the precision of Tidyparse on repairs of varying snippet lengths and edit distances. We compare the Precision@1 of our method against Seq2Parse, vanilla BIFI and BIFI with a beam size and precision at 2×10^4 distinct samples. The results are depicted in Fig. 2.

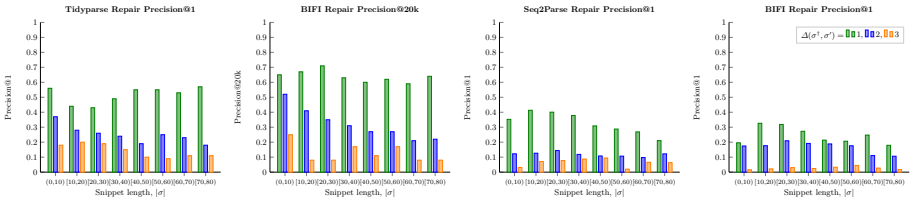


Fig. 2. Repair precision across varying snippet lengths and edit distances.

Tidyparse exhibits a highly competitive top-1 precision versus Seq2Parse and BIFI across all lengths and edit distances, and attains a significant advantage in the few-edit regime. The Precision@1 of our method is even competitive with BIFI's Precision@20k while requiring only a fraction of the data and compute.

For the second experiment, we measure the precision at various Precision@k thresholds and wall-clock timeouts. These results are shown in Fig. 3. Our method attains the same precision as Seq2Parse and BIFI for 1-edit repairs at comparable latency, however Tidyparse takes longer to attain the same precision for 2- and 3-edit repairs. BIFI and Seq2Parse both have subsecond single-shot response times but are neural models trained on a much larger dataset.

[3] https://github.com/gsakkas/seq2parse/tree/7ae0681f1139cb873868727f035c1b7a369c3eb9
[4] https://github.com/michiyasunaga/BIFI/tree/ee2a68cff8dbe88d2a2b2b5feabc7311d5f8338b

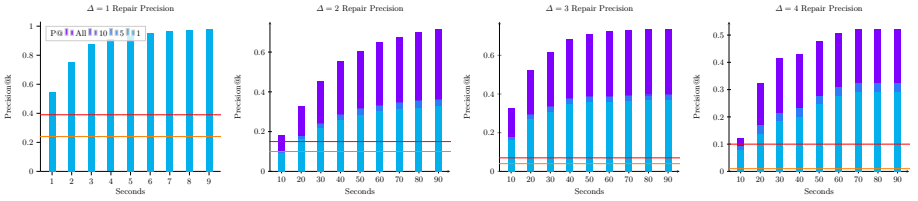


Fig. 3. Human repair benchmarks. Note the y-axis across different edit distance plots has varying ranges. The red line indicates Seq2Parse and the orange line indicates BIFI's Precision@1 on the same repairs, which both have subsecond repair latency.

The main lesson we draw from our experiments is that by constraining and parallelizing the decoder, it is possible to leverage compute to compete with large language models on practical program repair tasks. Though sample-efficient, their size comes at the cost of expensive training, and domain adaptation requires fine-tuning or retraining on pairwise repairs. Our method uses a small grammar and a cheap ranking metric (n-gram perplexity) to achieve significantly higher precision on natural repair. This approach allows Tidyparse to trade additional time for increased precision and handle languages with very little training data, offering far more flexibility and steer-ability during the repair process.

Our primary insight leading to state-of-the-art precision is that repairs are typically concentrated near the center of a small edit ball, and by enumerating or sampling the language of nearby valid snippets, then reranking generated repairs by naturalness, one can achieve significantly higher precision than one-shot neural repair. This is especially true for small-radii edit balls, where the admissible set is small enough to be completely enumerated and ranked.

5 Conclusion

From a usability standpoint, we argue that syntax repair tools should be as user-friendly and widely accessible as autocorrection tools in word processors. We argue that it is possible to reduce disruption from manual syntax repair and improve the efficiency of working programmers by driving down the latency needed to synthesize an acceptable repair. In contrast with program synthesizers that require intermediate editor states to be syntactically well-formed, our tool does not impose any constraints on the code itself being written and is possible to use in an interactive programming setting such as an IDE or text editor.

We have implemented our approach and demonstrated its viability as a tool for syntax assistance in real-world programming languages. Tidyparse [5] is capable of generating repairs for invalid source code in a range of practical languages with a context-free grammar and little to no training data required. We plan to continue expanding the autocorrection functionality and hope to conduct a user study to validate its effectiveness in practical programming scenarios.

[5] Artifact is available as a web application or IDE plugin.

References

1. Bar-Hillel, Y., Perles, M., Shamir, E.: On formal properties of simple phrase structure grammars. Sprachtypologie und Universalienforschung **14**, 143–172 (1961)
2. Considine, B.: A pragmatic approach to syntax repair. In: Companion Proceedings of the 2023 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity. pp. 19–21 (2023)
3. Diekmann, L., Tratt, L.: Don't panic! better, fewer, syntax errors for lr parsers. arXiv preprint arXiv:1804.07133 (2018)
4. Sakkas, G., Endres, M., Guo, P.J., Weimer, W., Jhala, R.: Seq2parse: neurosymbolic parse error repair. Proceedings of the ACM on Programming Languages **6**(OOPSLA2), 1180–1206 (2022)
5. Ugare, S., Suresh, T., Kang, H., Misailovic, S., Singh, G.: Improving LLM code generation with grammar augmentation. arXiv preprint arXiv:2403.01632 (2024)
6. Willard, B.T., Louf, R.: Efficient guided generation for LLMs. arXiv preprint arXiv:2307.09702 (2023)
7. Wong, A.W., Salimi, A., Chowdhury, S., Hindle, A.: Syntax and stack overflow: A methodology for extracting a corpus of syntax errors and fixes. In: 2019 IEEE International Conference on Software Maintenance and Evolution (ICSME). pp. 318–322. IEEE (2019)
8. Yasunaga, M., Liang, P.: Break-it-fix-it: Unsupervised learning for program repair. In: International Conference on Machine Learning. pp. 11941–11952. PMLR (2021)
9. Zhang, W., Wang, G., Chen, J., Xiong, Y., Liu, Y., Zhang, L.: Ordinalfix: Fixing compilation errors via shortest-path cfl reachability. arXiv preprint arXiv:2309.06771 (2023)