Realtime syntax repair with resource constraints

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, which means 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 an example. Suppose the user has written the following code fragment: `v = df.iloc(5:, 2:)`. Assuming an alphabet of just a hundred 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])`

To generate these repairs, we first lexicalize the input as follows:

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

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

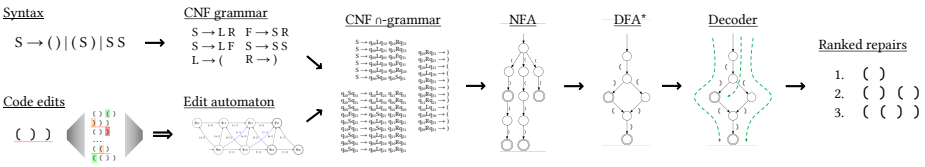


Fig. 1. Simplified dataflow. Given a grammar and broken code fragment, we create an automaton generating the language of small edits, then intersect it with the grammar to produce an intersection grammar, which can be simplified to a DFA and decoded.

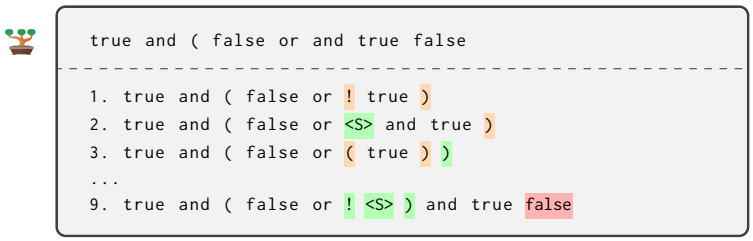
We first construct an automaton that recognizes every string within a certain edit distance of the input. We then reduce the grammar to Chomsky normal form (CNF), and using a modified construction from Bar-Hillel [2], construct an intersection grammar, which recognizes all and only valid sequences recognized by 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 we rank by negative log likelihood under the 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.

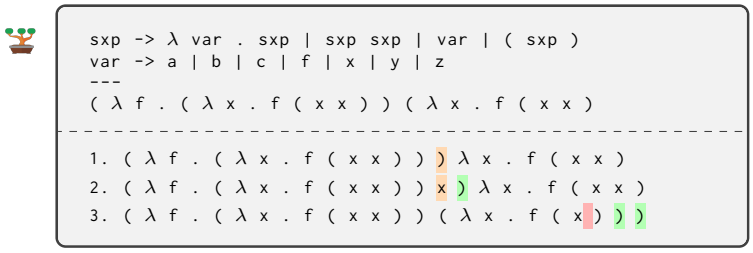
2 Usage examples

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, highlighted with colors, where green is insertion, orange is substitution and red is deletion.



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 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 samples are generable. The difficulty is not just synthesizing valid functions, but doing so in a parallel communication-free manner, without compromising soundness or completeness. The goal should be to massively scale up 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 these desiderata, e.g., soundness , completeness , naturalness, and parallel sampling. We compare our approach against Seq2Parse and BIFI, the leading neural program repair models.

	Sound	Complete	Natural	Theory	Parallel	Tool
Tidyparse [5]	✓	✓	✓	$\text{CFG}_\cap$	✓	IDE-ready
Seq2Parse [10]	✓†	✗	✓	CFG	✗	Python
BIFI [13]	✗	✗	✓	Σ^*	✗	Python
OrdinalFix [14]	✓	✗	✗	CFG+	✗	Rust
Aho & Peterson [1]	✓	✗	✗	CFG	✗	None
Diekmann & Tratt [6]	✓	✗	✗	LR	✗	Python

4 Evaluation

We use syntax errors and fixes from the Python language to validate our approach. Python source code fragments are abstracted as a sequence of lexical tokens using the official Python lexer, erasing numbers and identifiers, but retaining all other keywords. Accuracy is evaluated across a test set by checking for lexical equivalence with the ground-truth repair, following Sakkas et al. (2022) [10].

To evaluate accuracy, we use the Precision@k statistic, which measures the frequency of repairs in the top-k results matching the true repair. Specifically, given a repair model, $R : \Sigma^* \to 2^{\Sigma^*}$ and a test set $\mathcal{D}_{\text{test}}$ of pairwise aligned errors ($\sigma^\dagger$) and fixes (σ'), we define Precision@k as:

$$\text{Precision@k}(R) = \frac{1}{|\mathcal{D}_{\text{test}}|} \sum_{\langle \sigma^\dagger, \sigma' \rangle \in \mathcal{D}_{\text{test}}} 1\Big[\sigma' \in \underset{\sigma \subseteq R(\sigma^\dagger), |\sigma| \leq k}{\text{argmax}} \sum_{\sigma \in \sigma} \text{Score}(\sigma)\Big] \quad (1)$$

We compare our method against two external baselines, Seq2Parse and Break-It-Fix-It (BIFI) [13] on a single test set. This dataset [**?**] 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 and Levenshtein edit distance.

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 from their model, we only take a single repair prediction. For BIFI, we use the Round 2 breaker and fixer from commit `ee2a68c`[4], the highest-performing

[3] https://github.com/gsakkas/seq2parse/tree/7ae0681f1139cb873868727f035c1b7a369c3eb9

[4] https://github.com/michiyasunaga/BIFI/tree/ee2a68cff8dbe88d2a2b2b5feabc7311d5f8338b

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 language intersection experiments were conducted on 40 Intel Skylake cores running at 2.4 GHz, with 150 GB of RAM, executing bytecode compiled for JVM 17.0.2. To train our scoring function, we use an order-5 Markov chain trained on 55 million BIFI tokens. Training takes roughly 10 minutes, after which re-ranking is nearly instantaneous. Sequences are scored using NLL with Laplace smoothing and our evaluation measures the Precision@{1, 5, 10, All} for varying latency cutoffs up to 90s, although it often exhausts the search space and halts before timeout.

4.1 StackOverflow evaluation

For our first experiment, we measure the precision of our repair procedure at various lengths and Levenshtein distances. We rebalance the StackOverflow dataset across each length interval and edit distance, sample uniformly from each category and compare Precision@1 of our method against Seq2Parse, vanilla BIFI and BIFI with a beam size and precision at 2×10^4 distinct samples.

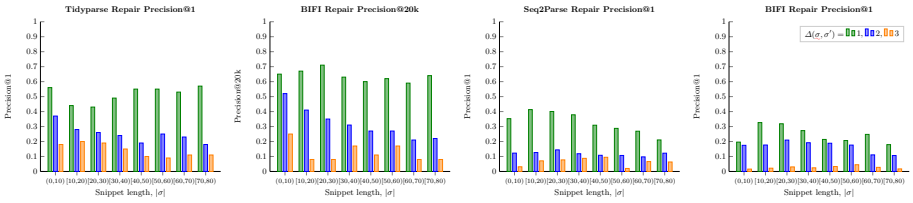


Fig. 2. Tidyparse, Seq2Parse and BIFI repair precision at various lengths and Levenshtein distances.

As we can see, Tidyparse has 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, whereas our Precision@All is Pareto-dominant across all lengths and edit distances, while requiring only a fraction of the data and compute.

Next, we measure the precision at various ranking cutoffs and wall-clock timeouts. 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 latency but are neural models trained on a much larger dataset.

The main lesson we draw from our experiments is that it is possible to leverage compute to compete with large language models on practical program repair

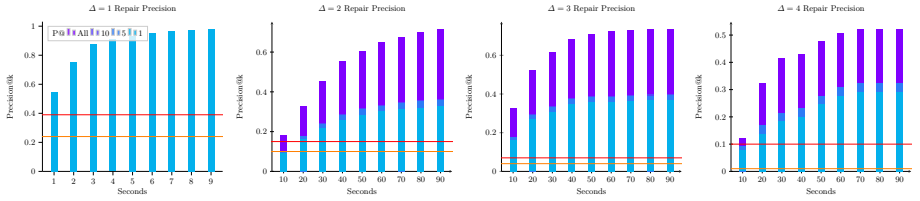


Fig. 3. Human repair benchmarks. Note the y-axis across different edit distance plots has varying ranges. The red line indicates Seq2Parse and the orange line indicates BIFI's Precision@1 on the same repairs.

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 approach uses a small grammar and a relatively cheap ranking metric to achieve significantly higher precision. This allows us to repair errors in languages with little to no training data and provides far more flexibility and controllability 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 Levenshtein ball, and by enumerating or sampling it carefully, then reranking repairs by naturalness, one can achieve significantly higher precision than one-shot neural repair. This is especially true for small-radii Levenshtein balls, where the admissible set is small enough to be completely enumerated and ranked. For larger radii, we can still achieve competitive precision by using an efficient decoder to sample the admissible set.

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 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 well-formed, our synthesizer does not impose any constraints on the code itself being written and is possible to use in an interactive programming setting.

We have implemented our approach and demonstrated its viability as a tool for syntax assistance in real-world programming languages. Tidyparse is capable of generating repairs for invalid source code in a range of practical languages with little to no data required. We plan to continue expanding the prototype's autocorrection functionality to cover a broader range of languages and hope to conduct a more thorough user study to validate its effectiveness in practical programming scenarios.

References

1. Aho, A.V., Peterson, T.G.: A minimum distance error-correcting parser for context-free languages. SIAM Journal on Computing **1**(4), 305–312 (1972)
2. Bar-Hillel, Y., Perles, M., Shamir, E.: On formal properties of simple phrase structure grammars. Sprachtypologie und Universalienforschung **14**, 143–172 (1961)
3. Bendkowski, M.: Automatic compile-time synthesis of entropy-optimal boltzmann samplers. arXiv preprint arXiv:2206.06668 (2022)
4. Beurer-Kellner, L., Fischer, M., Vechev, M.: Guiding LLMs the right way: Fast, non-invasive constrained generation. arXiv preprint arXiv:2403.06988 (2024)
5. 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)
6. Diekmann, L., Tratt, L.: Don't panic! better, fewer, syntax errors for lr parsers. arXiv preprint arXiv:1804.07133 (2018)
7. Fu, Y., Baker, E., Chen, Y.: Constrained decoding for secure code generation. arXiv preprint arXiv:2405.00218 (2024)
8. Park, K., Wang, J., Berg-Kirkpatrick, T., Polikarpova, N., D'Antoni, L.: Grammar-aligned decoding. arXiv preprint arXiv:2405.21047 (2024)
9. Roy, S., Sengupta, S., Bonadiman, D., Mansour, S., Gupta, A.: Flap: Flow adhering planning with constrained decoding in LLMs. arXiv preprint arXiv:2403.05766 (2024)
10. 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)
11. Ugare, S., Suresh, T., Kang, H., Misailovic, S., Singh, G.: Improving LLM code generation with grammar augmentation. arXiv preprint arXiv:2403.01632 (2024)
12. Willard, B.T., Louf, R.: Efficient guided generation for LLMs. arXiv preprint arXiv:2307.09702 (2023)
13. Yasunaga, M., Liang, P.: Break-it-fix-it: Unsupervised learning for program repair. In: International Conference on Machine Learning. pp. 11941–11952. PMLR (2021)
14. Zhang, W., Wang, G., Chen, J., Xiong, Y., Liu, Y., Zhang, L.: Ordinal-fix: Fixing compilation errors via shortest-path cfl reachability. arXiv preprint arXiv:2309.06771 (2023)