# 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 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, $\sigma_n$.

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 using language intersection techniques. 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, or continue to edit the code manually.

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 find 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 example, where the grammar is $S \to$ `( )` $\mid$ `(` $S$ `)` $\mid SS$ and the broken code is `( ) )`.

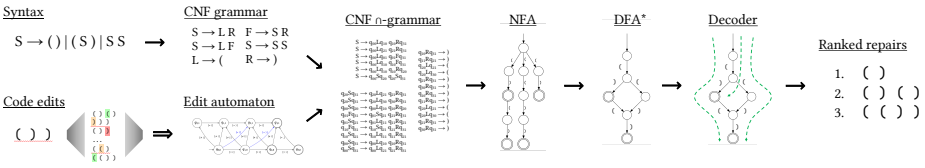


**Fig. 1.** Simplified dataflow. Given a grammar and broken code fragment, we create an automaton generating the language of small edits, then intersect it with the grammar to produce an intersection grammar, which can be simplified to a DFA and decoded.

To generate the repairs, we first construct an automaton that recognizes every string within a certain edit distance of the input. We then construct an intersection grammar, which recognizes all and only valid sequences within a certain edit distance of the input. 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 formalize the problem and describe the implementation details in the following sections.

## 2 Problem statement

Source code in a programming language can be treated as a string over a finite alphabet, $\Sigma$. We use a lexical alphabet for convenience. The language has a syntax, $\ell \subset \Sigma^*$, containing every acceptable program. A syntax error is an unacceptable string, $\underset{\sim}{\sigma} \notin \ell$. We can model syntax repair as a language intersection between a context-free language (CFL) and a regular language. Henceforth, $\underset{\sim}{\sigma}$ will always and only be used to denote a syntactically invalid string whose target language is known.

**Definition 1 (Bounded Levenshtein-CFL reachability).** *Given a CFL, $\ell$, and an invalid string, $\underset{\sim}{\sigma} : \bar{\ell}$, find every valid string reachable within $d$ edits of $\underset{\sim}{\sigma}$, i.e., letting $\Delta$ be the Levenshtein metric and $L(\underset{\sim}{\sigma}, d) = \{\sigma' \mid \Delta(\underset{\sim}{\sigma}, \sigma') \leq d\}$ be the Levenshtein $d$-ball, we seek to find $\ell_\cap = L(\underset{\sim}{\sigma}, d) \cap \ell$.*

As the admissible set $\ell_\cap$ is typically under-constrained, we want a procedure which surfaces natural and valid repairs over unnatural but valid repairs:

**Definition 2 (Ranked repair).** *Given a finite language $\ell_\cap = L(\underset{\sim}{\sigma}, d) \cap \ell$ and a probabilistic language model $P_\theta : \Sigma^* \to [0, 1] \subset \mathbb{R}$, the ranked repair problem is to find the top-$k$ maximum probability repairs under the language model, i.e.,*

$$R(\ell_\cap, P_\theta) = \underset{\sigma \subseteq \ell_\cap, |\sigma| \leq k}{\operatorname{argmax}} \sum_{\sigma \in \sigma} P_\theta(\sigma) \tag{1}$$

Assuming we have a grammar that recognizes the Levenshtein-CFL intersection, the question then becomes how to maximize the number of unique valid sentences in a given number of samples. Top-down incremental sampling with replacement eventually converges to the language, but does so superlinearly [**?**]. Due to practical considerations including latency, we require the sampler to converge linearly, ensuring with much higher probability that natural repairs are retrieved in a timely manner. This motivates the need for a specialized generating function. More precisely,

**Definition 3 (Linear convergence).** *Given a finite CFL, $\ell$, we want a randomized generating function, $\varphi : \mathbb{N}_{\leq |\ell|} \to 2^\ell$, whose rate of convergence is linear in expectation, i.e., $\mathbb{E}_{i \in [1,n]} |\varphi(i)| \propto n$.*

This will ensure that if $|\ell_\cap|$ is sufficiently small and enough samples are drawn, $\varphi$ is sure to include a representative subset, and additionally, will terminate after exhausting all valid repairs.

To satisfy Def. 3, we can construct a bijection from syntax trees to integers (§ **??**), sample integers uniformly without replacement, then decode them as trees. This will produce a set of unique trees, and each tree, assuming grammatical unambiguity, will correspond to a unique sentence in the language. Finally, sentences can be scored and ranked by likelihood under a language model.