Syntax Repair as Language Intersection

ANONYMOUS AUTHOR(S)

We introduce a new technique for correcting syntax errors in arbitrary context-free languages. Our work stems from the observation that syntax errors with a small repair typically have very few unique small repairs, which can usually be enumerated up to a small edit distance then quickly reranked. We place a heavy emphasis on precision: the enumerated set must contain every possible repair within a few edits and no invalid repairs. To do so, we model error correction as a language intersection problem between a Levenshtein automaton and a context-free grammar. To extract the repairs, we sample trees from the intersection grammar, yielding all valid repairs within a certain Levenshtein distance. Finally, we rank those repairs by n-gram likelihood.

1 INTRODUCTION

Syntax errors are a familiar nuisance for programmers, arising due to a variety of factors, from inexperience, typographic mistakes, to cognitive load. Often the mistake itself is simple to fix, but manual correction can disrupt concentration, a developer's most precious and fickle resource. Syntax repair attempts to automate the correction process by modifying a syntactically invalid program so that it conforms to the grammar, saving time and attention.

Prior work on syntax repair generally falls into two high-level categories: (1) *formal methods*, which use handcrafted rules to fix common syntax errors (e.g., [2]), and (2) *machine learning*, which use neural language models to repair code (e.g. [33, 38]). The former can be effective but typically produces a single repair, while the latter generates more natural edits, but is costly to train, prone to misgeneralization, and difficult to incorporate new constraints thereafter.

In this work, we take a pragmatic approach to syntax repair by embracing the problem's inherent ambiguity. We demonstrate it is possible to attain a significant advantage over state-of-the-art neural repair techniques by exhaustively retrieving every nearby repair and scoring it. Not only does this approach guarantee soundness, completeness and perfect length-generalization, it also improves precision when ranking by naturalness. Our proposed technique is straightforward:

(1) We model syntax repair as a language intersection problem between the Levenshtein ball and a context-free language, then materialize the grammar using a specialized version of the Bar-Hillel construction to Levenshtein intersections. (§ 4.3)
(2) We construct a data structure via idempotent matrix completion that compactly represents parse forests in context-free languages. This data structure is used to index syntax trees, significantly reducing the size of the intersection grammar. (§ 4.4, 4.5)
(3) To extract the repairs, we sample trees without replacement by constructing a bijection between syntax trees and integers, sample integers uniformly without replacement from a finite range, then decode them as trees. (§ 4.5)
(4) Finally, we rerank all repairs found before a fixed timeout by n-gram likelihood. (§ 4.6)

Our primary technical contributions are threefold: (1) the adaptation of the Levenshtein automaton and Bar-Hillel construction to syntax repair (2) a theoretical connection between idempotent matrix completion and CFL parsing with holes, and (3) an algebraic datatype and integer bijection for enumerating or sampling valid sentences in context-free languages. The efficacy of our technique owes to the fact it does not synthesize probable edits, but unique, fully formed repairs within a certain edit distance. This enables us to suggest correct and natural repairs with far less compute and data than would otherwise be required by a large language model to attain the same precision.

SPLASH'24, October 22-27, 2024, Pasadena, California, United States
2024. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

2 EXAMPLE

Consider the following Python snippet, which contains a small syntax error:

```
def prepend(i, k, L=[]) n and [prepend(i - 1, k, [b] + L) for b in range(k)]
```

We can fix it by inserting a colon after the function definition, yielding:

```
def prepend(i, k, L=[]): n and [prepend(i - 1, k, [b] + L) for b in range(k)]
```

A careful observer will note that there is only one way to repair this Python snippet by making a single edit. In fact, many programming languages share this curious property: syntax errors with a small repair have few uniquely small repairs. Valid sentences corrupted by a few small errors rarely have many small corrections. We call such sentences *metastable*, since they are relatively stable to small perturbations, as likely to be incurred by a careless typist or novice programmer.

Let us consider a slightly more ambiguous error: `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])`

With some typing information, we could easily narrow the results, but even in the absence of semantic constraints, one can probably rule out (2, 3, 6) given that `5[` and `2(` are rare bigrams in Python, and knowing `df.iloc` is often followed by `[`, determine (5) is most natural. This is the key insight behind our approach: we can usually locate the intended fix by exhaustively searching small repairs. As the set of small repairs is itself often small, if only we had some procedure to distinguish valid from invalid patches, the resulting solutions could be simply ranked by naturalness.

The trouble is that any such procedure must be highly sample-efficient. We cannot afford to sample the universe of possible d-token edits, then reject invalid ones – assuming it takes just 10ms to generate and check each sample, (1-6) could take 24+ hours to find. The hardness of brute-force search grows superpolynomially with edit distance, sentence length and alphabet size. We need a more efficient procedure for sampling all and only small valid repairs.

3 PROBLEM

We can model syntax repair as a language intersection problem between a context-free language (CFL) and a regular language. Henceforth, $\underset{\sim}{\sigma}$ will be used to denote a syntactically invalid string.

Definition 3.1 (Bounded Levenshtein-CFL reachability). Given a CFL ℓ and an invalid string, $\underset{\sim}{\sigma} : \ell^{\complement}$, find every valid string reachable within d edits of $\underset{\sim}{\sigma}$, i.e., letting Δ 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 $A = L(\underset{\sim}{\sigma}, d) \cap \ell$.

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

Definition 3.2 (Ranked repair). Given a finite language $A = L(\underset{\sim}{\sigma}, d) \cap \ell$ and a probabilistic language model $\text{P}_\theta : \Sigma^* \rightarrow [0, 1] \subset \mathbb{R}$, the ranked repair problem is to find the top-k maximum likelihood repairs under the language model. That is,

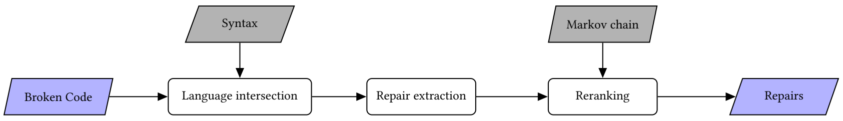
$$R(A, P_\theta) = \underset{\boldsymbol{\sigma} \subseteq A, |\boldsymbol{\sigma}| \leq k}{\text{argmax}} \sum_{\sigma \in \boldsymbol{\sigma}} \text{P}_\theta(\sigma \mid \underset{\sim}{\sigma}) \quad (1)$$

The problem of ranked repair can be solved by learning a distribution over strings, however this approach is highly sample-inefficient and generalizes poorly to new languages. Representing the problem as a distribution over Σ^* forces the language model to jointly learn syntax and stylometry. As we will demonstrate, this problem can be factorized into a bilevel objective: first retrieval, then ranking. By ensuring retrieval is sufficiently precise and exhaustive, maximizing likelihood over the admissible set can be achieved with a much simpler, syntax-oblivious language model.

Even with an extremely efficient approximate sampler for $\sigma \sim \ell_\cap$, due to the size of ℓ and $L(\underset{\sim}{\sigma}, d)$, it would be intractable to sample either ℓ or $L(\underset{\sim}{\sigma}, d)$, then reject invalid ($\sigma \notin \ell$) or unreachable ($\sigma \notin L(\underset{\sim}{\sigma}, d)$) edits, and completely out of the question to sample $\sigma \sim \Sigma^*$ as do many large language models. Instead, we will explicitly construct a grammar generating $\ell \cap L(\underset{\sim}{\sigma}, d)$, sample from it, then rerank all repairs after a fixed timeout. So long as $|\ell_\cap|$ is sufficiently small and recognizes all and only small repairs, our sampler is sure to retrieve the most natural repair and terminate quickly.

4 METHOD

At a very high level, the method we describe in this paper takes as input the invalid source code, and returns a set of plausible repairs. We assume the target syntax and a small distribution of lexical n-grams is provided, and optionally, a PCFG for improved efficiency. Our method can decomposed into three primary steps: (1) language intersection, (2) repair extraction, and (3) reranking.



First, we generate a synthetic grammar representing the intersection between the syntax and the Levenshtein ball around the source code. During extraction, we retrieve as many repairs as possible from the intersection grammar via sampling or enumeration. Finally, in the reranking step, we rank all repairs by n-gram likelihood. More specifically, we depict this as a flowchart in Fig. 1.

Since the syntax of most programming languages is context-free, we first construct a context-free grammar (CFG), $G_\cap$, that represents the intersection between the language's syntax (G) and an automaton recognizing the Levenshtein ball of a given radius, $L(\underset{\sim}{\sigma}, d)$. As CFLs are closed under intersection with regular languages, this is admissible. Three outcomes are possible:

(1) $G_\cap$ is empty, in which case there is no repair within the given radius. In this case, we simply increase the radius and try again.
(2) $\mathcal{L}(G_\cap)$ is small, in which case we enumerate all possible repairs. Complete enumeration is tractable for the majority of syntax repairs.
(3) $\mathcal{L}(G_\cap)$ is too large to enumerate, so we sample from the intersection grammar $G_\cap$. Sampling is necessary for a minority of remaining cases.

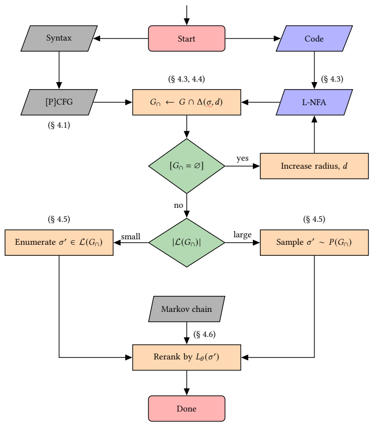


Fig. 1. Flowchart of our proposed method.

As long as we have done our job correctly, the intersection language should contain nearly every repair within a certain Levenshtein distance, and no invalid repairs. We will first describe how to generate the intersection grammar (§ 4.2, 4.3), then, describe a data structure that compactly represents its language, allowing us to efficiently extract all repairs contained within (§ 4.5). Finally, we use an n-gram model to rank and return the top-k results by likelihood (§ 4.6).

4.1 Preliminaries

Recall that a CFG, $\mathcal{G} = \langle \Sigma, V, P, S \rangle$, is a quadruple consisting of terminals (Σ), nonterminals (V), productions ($P\colon V \longrightarrow (V \mid \Sigma)^*$), and a start symbol, ($S$). Every CFG is reducible to so-called *Chomsky Normal Form*, $P'\colon V \longrightarrow (V^2 \mid \Sigma)$, where every production is either (1) a binary production $w \longrightarrow xz$, or (2) a unit production $w \longrightarrow t$, where $w, x, z : V$ and $t : \Sigma$. For example:

$$G = \left\{\, S \rightarrow S\, S \mid (\, S\,) \mid (\,) \,\right\} \Longrightarrow G' = \left\{\, S \rightarrow Q\, R \mid S\, S \mid L\, R, \quad R \rightarrow), \quad L \rightarrow (, \quad Q \rightarrow L\, S \,\right\}$$

Likewise, a finite state automaton is a quintuple $\mathcal{A} = \langle Q, \Sigma, \delta, I, F \rangle$, where Q is a finite set of states, Σ is a finite alphabet, $\delta \subseteq Q \times \Sigma \times Q$ is the transition function, and $I, F \subseteq Q$ are the set of initial and final states, respectively. We will adhere to this notation in the following sections.

4.2 Modeling code edits with the Levenshtein automaton

Levenshtein edits are recognized by an automaton known as the Levenshtein automaton. As the original construction defined by Schultz and Mihov [34] contains cycles and ε-transitions, we propose a variant which is ε-free and acyclic. Furthermore, we adopt a nominal form which supports infinite alphabets and considerably simplifies the language intersection to follow. Illustrated in Fig. 2 is an example of a small Levenshtein automaton recognizing $L(\sigma : \Sigma^5, 3)$. Unlabeled arcs accept any terminal from the alphabet, Σ. Equivalently, this transition system can be viewed as a kind of proof system in an unlabeled lattice. The following construction is equivalent to Schultz and Mihov's original Levenshtein automaton, but is more amenable to our purposes as it does not any contain ε-arcs, and instead uses skip connections to recognize consecutive deletions of varying lengths.

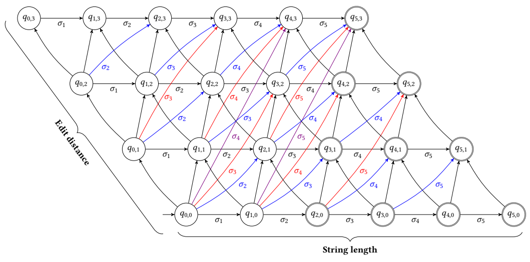


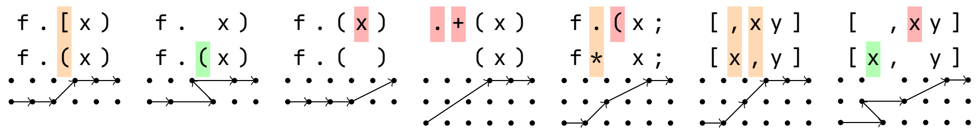
Fig. 2. NFA recognizing Levenshtein $L(\sigma : \Sigma^5, 3)$.

$$\frac{s \in \Sigma \quad i \in [0, n] \quad j \in [1, d_{\max}]}{(q_{i,j-1} \xrightarrow{s} q_{i,j}) \in \delta} \nwarrow \qquad \frac{s \in \Sigma \quad i \in [1, n] \quad j \in [1, d_{\max}]}{(q_{i-1,j-1} \xrightarrow{s} q_{i,j}) \in \delta} \nearrow$$

$$\frac{i \in [1, n] \quad j \in [0, d_{\max}]}{(q_{i-1,j} \xrightarrow{\sigma_i} q_{i,j}) \in \delta} \rightarrow \qquad \frac{d \in [1, d_{\max}] \quad i \in [d+1, n] \quad j \in [d, d_{\max}]}{(q_{i-d-1,j-d} \xrightarrow{\sigma_i} q_{i,j}) \in \delta} \nearrow$$

$$\frac{}{q_{0,0} \in I} \textsc{Init} \qquad \frac{q_{i,j} \qquad |n - i + j| \leq d_{\max}}{q_{i,j} \in F} \textsc{Done}$$

Each arc plays a specific role. $\nwarrow$ handles insertions, $\nearrow$ handles substitutions and $\nearrow$ handles deletions of one or more terminals. Let us consider some illustrative cases.



```
f . [ x )   f .   x )   f . ( x )   . + ( x )   f . ( x ;   [ , x y ]   [   , x y ]
f . ( x )   f . ( x )   f . (   )       ( x )   f *   x ;   [ x , y ]   [ x ,   y ]
```

Note that the same patch can have multiple Levenshtein alignments. Done constructs the final states, which are all states accepting strings σ' whose Levenshtein distance $\Delta(\sigma, \sigma') \leq d_{\max}$.

To avoid creating a parallel bundle of arcs for each insertion and substitution point, we instead decorate each arc with a nominal predicate, accepting or rejecting σ_i. To distinguish this nominal variant from the original construction, we highlight the modified rules in orange below.

$$\frac{i \in [0, n] \quad j \in [1, k]}{(q_{i,j-1} \xrightarrow{[\neq \sigma_i]} q_{i,j}) \in \delta} \nwarrow \qquad \frac{i \in [1, n] \quad j \in [1, k]}{(q_{i-1,j-1} \xrightarrow{[\neq \sigma_i]} q_{i,j}) \in \delta} \nearrow$$

$$\frac{i \in [1, n] \quad j \in [0, k]}{(q_{i-1,j} \xrightarrow{[=\sigma_i]} q_{i,j}) \in \delta} \rightarrow \qquad \frac{d \in [1, d_{\max}] \quad i \in [d+1, n] \quad j \in [d, k]}{(q_{i-d-1,j-d} \xrightarrow{[=\sigma_i]} q_{i,j}) \in \delta} \nearrow$$

Nominalizing the NFA eliminates the creation of $e = 2(|\Sigma| - 1) \cdot |\sigma| \cdot d_{\max}$ unnecessary arcs over the entire Levenshtein automaton and drastically reduces the size of the construction to follow, but does not affect the underlying semantics. Thus, it is essential to first nominalize the automaton before proceeding to avoid a large blowup in the intermediate grammar.

4.3 Recognizing syntactically valid code changes via language intersection

We now describe the Bar-Hillel construction, which generates a grammar recognizing the intersection between a regular and a context-free language, then specialize it to Levenshtein intersections.

Lemma 4.1. *For any context-free language ℓ and finite state automaton α, there exists a context-free grammar $G_\cap$ such that $\mathcal{L}(G_\cap) = \ell \cap \mathcal{L}(\alpha)$. See Bar-Hillel [5].*

Although Bar-Hillel [5] lacks an explicit construction, Beigel and Gasarch [8] construct $G_\cap$ like so:

$$\frac{q \in I \quad r \in F}{\big(S \to qSr\big) \in P_\cap} \qquad \frac{(A \to a) \in P \qquad (q \overset{a}{\to} r) \in \delta}{\big(qAr \to a\big) \in P_\cap} \qquad \frac{(w \to xz) \in P \qquad p, q, r \in Q}{\big(pwr \to (pxq)(qzr)\big) \in P_\cap} \bowtie$$

This, now standard, Bar-Hillel construction applies to any CFL and REG language intersection, but generates a grammar whose cardinality is approximately $|P_\cap| = |I|\cdot|F|+|P|\cdot|\Sigma|\cdot|\sigma|\cdot 2d_{\max}+|P|\cdot|Q|^3$. Applying the BH construction directly to practical languages and code snippets can generate hundreds of trillions of productions for even modestly-sized grammars and Levenshtein automata. Instead, we will describe a kind of reachability analysis that elides many superfluous productions in the case of Levenshtein intersection, greatly reducing the size of the intersection grammar, $G_\cap$.

Consider $\bowtie$, the most expensive rule. What $\bowtie$ tells us is each nonterminal in the intersection grammar matches a substring simultaneously recognized by (1) a pair of states in the original NFA and (2) a nonterminal in the original CFG. A key observation is that $\bowtie$ generates the Cartesian product of every such triple, but this is a gross overapproximation for most NFAs and CFGs, as the vast majority of all state pairs and nonterminals recognize no strings in common.

To identify these superfluous triples, we define an interval domain that soundly overapproximates the Parikh image, encoding the minimum and maximum number of terminals each nonterminal can generate. Since some intervals may be right-unbounded, we write $\mathbb{N}^* = \mathbb{N} \cup \{\infty\}$ to denote the upper bound, and $\Pi = \{[a, b] \in \mathbb{N} \times \mathbb{N}^* \mid a \leq b\}^{|\Sigma|}$ to denote the Parikh image of all terminals.

Definition 4.2 (Parikh mapping of a nonterminal). Let $p : \Sigma^* \to \mathbb{N}^{|\Sigma|}$ be the Parikh operator [31], which counts the frequency of terminals in a string. We define the Parikh map, $\pi : V \to \Pi$, as a function returning the smallest interval such that $\forall \sigma : \Sigma^*, \forall v : V, v \Rightarrow^* \sigma \vdash p(\sigma) \in \pi(v)$.

In other words, the Parikh mapping computes the greatest lower and least upper bound of the Parikh image over all strings in the language of a nonterminal. The infimum of a nonterminal's Parikh interval tells us how many of each terminal a nonterminal *must* generate, and the supremum tells us how many it *can* generate. Likewise, we define a similar relation over NFA state pairs:

Definition 4.3 (Parikh mapping of NFA states). We define $\pi : Q \times Q \to \Pi$ as returning the smallest interval such that $\forall \sigma : \Sigma^*, \forall q, q' : Q, q \overset{\sigma}{\Longrightarrow} q' \vdash p(\sigma) \in \pi(q, q')$.

Next, we will define a measure on Parikh intervals representing the minimum total edits required to transform a string in one Parikh interval to a string in another, across all such pairings.

Definition 4.4 (Parikh divergence). Given two Parikh intervals $\pi, \pi' : \Pi$, we define the divergence between them as $\pi \parallel \pi' = \sum_{n=1}^{|\Sigma|} \min_{(i,i') \in \pi[n] \times \pi'[n]} |i - i'|$.

Now, we know that if the Parikh divergence between two intervals exceeds the Levenshtein margin between two states in a Lev-NFA, those intervals must be incompatible as no two strings, one from each Parikh interval, can be transformed into the other with fewer than $\pi \parallel \pi'$ edits.

Definition 4.5 (Levenshtein-Parikh compatibility). Let $q = q_{h,i}, q' = q_{j,k}$ be two states in a Lev-NFA and V be a CFG nonterminal. We say that $(q, v, q') : Q \times V \times Q$ are compatible iff the Parikh divergence is bounded by the Levenshtein margin $k - i$, i.e., $v \triangleleft qq' \iff (\pi(v) \parallel \pi(q, q')) \leq k - i$.

Finally, we define the modified Bar-Hillel construction for nominal Levenshtein automata as:

$$\frac{(A \to a) \in P \quad S.a \quad (q \overset{S}{\to} r) \in \delta}{(qAr \to a) \in P_\cap} \quad \frac{w \lhd pr \quad x \lhd pq \quad z \lhd qr \quad (w \to xz) \in P \quad p, q, r \in Q}{(pwr \to (pxq)(qzr)) \in P_\cap} \bowtie$$

After $G_\cap$ is constructed, we then renormalize it by removing all unreachable and non-generating productions following [20] to obtain $G_\cap^*$, which is usually several orders of magnitude smaller.

Now that we have a language to recognize nearby repairs, we will need a method to generate the repairs themselves. We impose specific criteria on such a procedure: it must generate only valid repairs and all repairs in the language if possible, otherwise as many as can be sampled in an arbitrary but fixed timeout. In the following sections, we will describe a constructor (§ 4.4) for a data structure (§ 4.5) representing parse forests in a length-bounded CFL. Among other features, this data structure provides an explicit way to construct the Parikh map, and a method for sampling parse trees from the language with or without replacement.

4.4 Repair as idempotent matrix completion

In this section, we will introduce the porous completion problem and show how it can be translated to a kind of idempotent matrix completion, whose roots are valid strings in a context-free language. This technique is convenient for its geometric interpretability, parallelizability, and generalizability to any CFG, regardless of finitude or ambiguity. We will see how, by redefining the algebraic operations $\oplus, \otimes$ over different carrier sets, one can obtain a recognizer, parser, generator, Parikh map and other convenient structures for working with CFLs.

Given a CFG, $G' : \mathcal{G}$ in Chomsky Normal Form (CNF), we can construct a recognizer $R : \mathcal{G} \to \Sigma^n \to \mathbb{B}$ for strings $\sigma : \Sigma^n$ as follows. Let 2^V be our domain, 0 be $\varnothing$, $\oplus$ be $\cup$, and $\otimes$ be defined as:

$$X \otimes Z = \Big\{ \ w \mid \langle x, z \rangle \in X \times Z, (w \to xz) \in P \ \Big\} \tag{2}$$

If we define $\hat{\sigma}_r = \{w \mid (w \to \sigma_r) \in P\}$, then construct a matrix with nonterminals on the superdiagonal representing each token, $M_0[r + 1 = c](G', \sigma) = \hat{\sigma}_r$, the fixpoint $M_{i+1} = M_i + M_i^2$ is uniquely determined by the superdiagonal entries, which are computed as follows:

$$M_0 = \begin{pmatrix} \varnothing & \hat{\sigma}_1 & \varnothing & \cdots & \varnothing \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ & & & \ddots & \varnothing \\ & & & \ddots & \hat{\sigma}_n \\ \varnothing & \cdots & \cdots & \cdots & \varnothing \end{pmatrix} \Rightarrow \begin{pmatrix} \varnothing & \hat{\sigma}_1 & \Lambda & \cdots & \varnothing \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ & & & \ddots & \Lambda \\ & & & \ddots & \hat{\sigma}_n \\ \varnothing & \cdots & \cdots & \cdots & \varnothing \end{pmatrix} \Rightarrow \ldots \Rightarrow M_\infty = \begin{pmatrix} \varnothing & \hat{\sigma}_1 & \Lambda & \cdots & \Lambda_\sigma^* \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ & & & \ddots & \Lambda \\ & & & \ddots & \hat{\sigma}_n \\ \varnothing & \cdots & \cdots & \cdots & \varnothing \end{pmatrix}$$

The northeasternmost entry Λ_σ^* gives us the recognizer $R(G', \sigma) = [S \in \Lambda_\sigma^*] \Leftrightarrow [\sigma \in \mathcal{L}(G)]$ [1].

This procedure can be lifted to the domain of strings containing free variables, which we call the *porous completion problem*. In this case, the fixpoint is characterized by a system of language equations, whose solutions are the set of all sentences consistent with the template.

Definition 4.6 (Porous completion). Let $\underline{\Sigma} = \Sigma \cup \{\_\}$, where $\_$ denotes a hole. We denote $\sqsubseteq : \Sigma^n \times \underline{\Sigma}^n$ as the relation $\{\langle \sigma', \sigma \rangle \mid \sigma_i \in \Sigma \implies \sigma_i' = \sigma_i\}$ and the set of all inhabitants $\{\sigma' : \Sigma^+ \mid \sigma' \sqsubseteq \sigma\}$ as $\text{H}(\sigma)$. Given a *porous string*, $\sigma : \underline{\Sigma}^*$ we seek all syntactically valid inhabitants, i.e., $A(\sigma) = \text{H}(\sigma) \cap \ell$.

Let us consider an example with two holes, $\sigma = 1\ \_\ \_$, and the grammar being $G = \{S \to NON, O \to + \mid \times, N \to 0 \mid 1\}$. This can be rewritten into CNF as $G' = \{S \to NL, N \to 0 \mid 1, O \to \times \mid +, L \to ON\}$. Using the algebra where $\oplus = \cup$, $X \otimes Z = \Big\{ \ w \mid \langle x, z \rangle \in X \times Z, (w \to xz) \in P \ \Big\}$, the fixpoint $M' = M + M^2$ can be computed as follows, shown in the leftmost column:

[1]Hereinafter, we use Iverson brackets to denote the indicator function of a predicate with free variables, i.e., $[P] \Leftrightarrow \mathbb{1}(P)$.

	2^V	$\mathbb{Z}_2^{\lvert V\rvert}$	$\mathbb{Z}_2^{\lvert V\rvert} \rightarrow \mathbb{Z}_2^{\lvert V\rvert}$
M_0	$\begin{pmatrix} \{N\} & & \\ & \{N,O\} & \\ & & \{N,O\} \end{pmatrix}$	$\begin{pmatrix} \square\blacksquare\square\square & & \\ & \square\blacksquare\blacksquare\square & \\ & & \square\blacksquare\blacksquare\square \end{pmatrix}$	$\begin{pmatrix} V_{0,1} & & \\ & V_{1,2} & \\ & & V_{2,3} \end{pmatrix}$
M_1	$\begin{pmatrix} \{N\} & \varnothing & \\ & \{N,O\} & \{L\} \\ & & \{N,O\} \end{pmatrix}$	$\begin{pmatrix} \square\blacksquare\square\square & \square\square\square\square & \\ & \square\blacksquare\blacksquare\square & \blacksquare\square\square\square \\ & & \square\blacksquare\blacksquare\square \end{pmatrix}$	$\begin{pmatrix} V_{0,1} & V_{0,2} & \\ & V_{1,2} & V_{1,3} \\ & & V_{2,3} \end{pmatrix}$
$M_2 = M_\infty$	$\begin{pmatrix} \{N\} & \varnothing & \{S\} \\ & \{N,O\} & \{L\} \\ & & \{N,O\} \end{pmatrix}$	$\begin{pmatrix} \square\blacksquare\square\square & \square\square\square\square & \square\square\square\blacksquare \\ & \square\blacksquare\blacksquare\square & \blacksquare\square\square\square \\ & & \square\blacksquare\blacksquare\square \end{pmatrix}$	$\begin{pmatrix} V_{0,1} & V_{0,2} & V_{0,3} \\ & V_{1,2} & V_{1,3} \\ & & V_{2,3} \end{pmatrix}$

The same procedure can be translated, without loss of generality, into the bit domain ($\mathbb{Z}_2^{|V|}$) using a lexicographic ordering, however M_∞ in both 2^V and $\mathbb{Z}_2^{|V|}$ represents a decision procedure, i.e., $[S \in V_{0,3}] \Leftrightarrow [V_{0,3,3} = \blacksquare] \Leftrightarrow [A(\sigma) \neq \varnothing]$. Since $V_{0,3} = \{S\}$, we know there exists at least one $\sigma' \in A$, but M_∞ does not reveal its identity.

In order to extract the inhabitants, we can translate the bitwise procedure into an equation with free variables. Here, we can encode the idempotency constraint directly as $M = M^2$. We first define $X \boxtimes Z = [X_2 \land Z_1, \bot, \bot, X_1 \land Z_0]$ and $X \boxplus Z = [X_i \lor Z_i]_{i \in [0,|V|)}$. Since the unit nonterminals O, N can only occur on the superdiagonal, they may be safely ignored by $\boxtimes$. To solve for M_∞, we proceed by first computing $V_{0,2}, V_{1,3}$ as follows:

$$\begin{aligned} V_{0,2} &= V_{0,j} \cdot V_{j,2} = V_{0,1} \boxtimes V_{1,2} \\ &= [L \in V_{0,2}, \bot, \bot, S \in V_{0,2}] \\ &= [O \in V_{0,1} \land N \in V_{1,2}, \bot, \bot, N \in V_{0,1} \land L \in V_{1,2}] \\ &= [V_{0,1,2} \land V_{1,2,1}, \bot, \bot, V_{0,1,1} \land V_{1,2,0}] \end{aligned}$$

$$\begin{aligned} V_{1,3} &= V_{1,j} \cdot V_{j,3} = V_{1,2} \boxtimes V_{2,3} \\ &= [L \in V_{1,3}, \bot, \bot, S \in V_{1,3}] \\ &= [O \in V_{1,2} \land N \in V_{2,3}, \bot, \bot, N \in V_{1,2} \land L \in V_{2,3}] \\ &= [V_{1,2,2} \land V_{2,3,1}, \bot, \bot, V_{1,2,1} \land V_{2,3,0}] \end{aligned}$$

Now we solve for the corner entry $V_{0,3}$ by taking the bitwise dot product between the first row and last column, yielding:

$$\begin{aligned} V_{0,3} &= V_{0,j} \cdot V_{j,3} = V_{0,1} \boxtimes V_{1,3} \boxplus V_{0,2} \boxtimes V_{2,3} \\ &= [V_{0,1,2} \land V_{1,3,1} \lor V_{0,2,2} \land V_{2,3,1}, \bot, \bot, V_{0,1,1} \land V_{1,3,0} \lor V_{0,2,1} \land V_{2,3,0}] \end{aligned}$$

Since we only care about $V_{0,3,3} \Leftrightarrow [S \in V_{0,3}]$, so we can ignore the first three entries and solve for:

$$\begin{aligned} V_{0,3,3} &= V_{0,1,1} \land V_{1,3,0} \lor V_{0,2,1} \land V_{2,3,0} \\ &= V_{0,1,1} \land (V_{1,2,2} \land V_{2,3,1}) \lor V_{0,2,1} \land \bot \\ &= V_{0,1,1} \land V_{1,2,2} \land V_{2,3,1} \\ &= [N \in V_{0,1}] \land [O \in V_{1,2}] \land [N \in V_{2,3}] \end{aligned}$$

Now we know that $\sigma = 1\ \underline{\mathrm{O}}\ \underline{\mathrm{N}}$ is a valid solution, and we can take the product $\{1\} \times \hat{\sigma}_2^{-1}(O) \times \hat{\sigma}_3^{-1}(N)$ to recover the admissible set, yielding $A = \{1{+}0, 1{+}1, 1{\times}0, 1{\times}1\}$. In this case, since G is unambiguous, there is only one parse tree satisfying $V_{0,|\sigma|,3}$, but in general, there can be multiple valid parse trees.

4.5 An algebraic datatype for context-free parse forests

The procedure described in § 4.4 generates solutions satisfying the matrix fixpoint, but forgets provenance. The question naturally arises, is there a way to solve for the parse trees directly? This would allow us to handle ambiguous grammars, whilst preserving the natural treelike structure.

We will now describe a datatype for compactly representing CFL parse forests, then redefine the semiring algebra over this domain. This construction is especially convenient for tracking provenance under ambiguity, Parikh mapping, counting the size of a finite CFL, and sampling trees either with replacement using a PCFG, or without replacement using an integer pairing function.

We first define a datatype $\mathbb{T}_3 = (V \cup \Sigma) \rightharpoonup \mathbb{T}_2$ where $\mathbb{T}_2 = (V \cup \Sigma) \times (\mathbb{N} \rightharpoonup \mathbb{T}_2 \times \mathbb{T}_2)$[2]. Morally, we can think of $\mathbb{T}_2$ as an implicit set of possible trees that can be generated by a CFG in CNF, consistent with a finite-length porous string. Structurally, we may interpret $\mathbb{T}_2$ as an algebraic data type corresponding to the fixpoints of the following recurrence, which tells us each $\mathbb{T}_2$ can be a terminal, nonterminal, or a nonterminal and a sequence of nonterminal pairs and their two children:

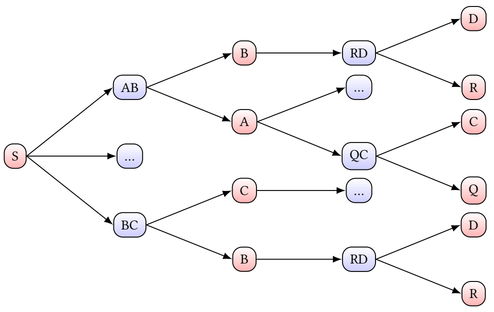


Fig. 3. A partial $\mathbb{T}_2$ corresponding to the grammar $\{S \to BC \mid \ldots \mid AB, B \to RD \mid \ldots, A \to QC \mid \ldots\}$.

$$L(p) = 1 + pL(p) \qquad\qquad P(a) = \Sigma + V + VL\big(V^2P(a)^2\big) \tag{3}$$

Given a $\sigma : \underline{\Sigma}$, we construct $\mathbb{T}_2$ from the bottom-up, and sample from the top-down. Depicted in Fig. 3 is a partial $\mathbb{T}_2$, where red nodes are `roots` and blue nodes are `children`.

We construct the first upper diagonal $\hat{\sigma}_r = \Lambda(\sigma_r)$ as follows:

$$\Lambda(s : \underline{\Sigma}) \mapsto \begin{cases} \bigoplus_{s' \in \Sigma} \Lambda(s') & \text{if } s \text{ is a hole,} \\ \big\{\mathbb{T}_2\big(w, \big[\langle \mathbb{T}_2(s), \mathbb{T}_2(\varepsilon)\rangle\big]\big) \mid (w \to s) \in P\big\} & \text{otherwise.} \end{cases} \tag{4}$$

This initializes the superdiagonal entries, enabling us to compute the fixpoint M_∞ in the same manner described in § 4.4 by redefining $\oplus, \otimes : \mathbb{T}_3 \times \mathbb{T}_3 \to \mathbb{T}_3$ as:

$$X \oplus Z \mapsto \bigcup_{k \in \pi_1(X \cup Z)} \Big\{k \Rightarrow \mathbb{T}_2(k, x \cup z) \mid x \in \pi_2(X \circ k), z \in \pi_2(Z \circ k)\Big\} \tag{5}$$

$$X \otimes Z \mapsto \bigoplus_{(w \to xz) \in P} \Big\{\mathbb{T}_2\Big(w, \big[\langle X \circ x, Z \circ z\rangle\big]\Big) \mid x \in \pi_1(X), z \in \pi_1(Z)\Big\} \tag{6}$$

These operators group subtrees by their root nonterminal, then aggregate their children. Instead of tracking sets, each Λ now becomes a dictionary of $\mathbb{T}_2$, indexed by their root nonterminals.

$\mathbb{T}_2$ is a convenient datatype for many operations involving CFGs. We can use it to approximate the Parikh image, compute the size of a finite CFG, and sample parse trees with or without replacement. For example, to obtain the Parikh map (Def. 4.2), we may use the following recurrence,

$$\pi(T : \mathbb{T}_2) \mapsto \begin{cases} \Big[[1,1] \text{ if } \texttt{root}(T) = s \text{ else } [0,0]\Big]_{s \in \Sigma} & \text{if } T \text{ is a leaf,} \\ \bigoplus_{\langle T_1, T_2\rangle \in \texttt{children}(T)} \pi(T_1) \otimes \pi(T_2) & \text{otherwise.} \end{cases} \tag{7}$$

where the operations over Parikh maps $\oplus, \otimes : \Pi \times \Pi \to \Pi$ are defined respectively as follows:

[2]Given a $T : \mathbb{T}_2$, we may also refer to $\pi_1(T), \pi_2(T)$ as `root`(T) and `children`(T) respectively.

$$X \oplus Z \mapsto \big[[\min(X_s \cup Z_s), \max(X_s \cup Z_s)]\big]_{s\in\Sigma} \tag{8}$$
$$X \otimes Z \mapsto \big[[\min(X_s) + \min(Z_s), \max(X_s) + \max(Z_s)]\big]_{s\in\Sigma} \tag{9}$$

To obtain the full Parikh map of a length-bounded CFG, we abstractly parse the porous string and take the union of all intervals, which subsumes every repair in the Levenshtein ball. Given a specific programming language syntax, G, this function can be precomputed and cached for all $v : V$, and small values of $n, d : \mathbb{N}$ for the sake of efficiency, then used to retrieve the Levenshtein-d Parikh-$\langle v, n\rangle$ map for any invalid string $\underset{\sim}{\sigma}$ of length n in constant time:

$$\pi(G : \mathcal{G}, v : V, n : \mathbb{N}, d : \mathbb{N}) : \Pi = \bigoplus_{i\in[n-d,n+d]} \pi\big(\Lambda^*(\{\_\}^i) \circ v\big) \tag{10}$$

$\mathbb{T}_2$ also allows us to sample whole parse trees by constructing $(\Lambda^*(\sigma) \circ S) : \mathbb{T}_2$ from the bottom-up and sampling top-down. Given a PCFG whose productions indexed by each nonterminal are decorated with a probability vector $\mathbf{p}$ (uniform in the non-probabilistic case), we define a tree sampler $\Gamma : (\mathbb{T}_2 \mid \mathbb{T}_2^2) \rightsquigarrow \mathbb{T}$ which recursively draws children according to a Multinoulli distribution:

$$\Gamma(T) \mapsto \begin{cases} \texttt{BTree}\Big(\texttt{root}(T), \Gamma\big(\text{Multi}(\texttt{children}(T), \mathbf{p})\big)\Big) & \text{if } T : \mathbb{T}_2 \\ \Big\langle \Gamma\big(\pi_1(T)\big), \Gamma\big(\pi_2(T)\big)\Big\rangle & \text{if } T : \mathbb{T}_2 \times \mathbb{T}_2 \end{cases} \tag{11}$$

This method is closely related to the generating function for the ordinary Boltzmann sampler,

$$\Gamma C(x) \mapsto \begin{cases} \text{Bern}\left(\frac{A(x)}{A(x)+B(x)}\right) \rightarrow \Gamma A(x) \mid \Gamma B(x) & \text{if } \mathcal{C} = \mathcal{A} + \mathcal{B} \\ \big\langle \Gamma A(x), \Gamma B(x)\big\rangle & \text{if } \mathcal{C} = \mathcal{A} \times \mathcal{B} \end{cases} \tag{12}$$

from analytic combinatorics, however unlike Duchon et al. [18], our work does not depend on rejection to guarantee exact-size sampling, as all trees from $\mathbb{T}_2$ will necessarily be the same width.

The number of binary trees inhabiting a single instance of $\mathbb{T}_2$ is sensititive to the number of nonterminals and rule expansions in the grammar. To obtain the total number of trees with breadth n, we abstractly parse the porous string, letting $T = \Lambda^*(\{\_\}^n) \circ S$, then use the recurrence below to compute the total number of unique trees in the language:

$$|T : \mathbb{T}_2| \mapsto \begin{cases} 1 & \text{if } T \text{ is a leaf,} \\ \sum_{\langle T_1, T_2\rangle \in \texttt{children}(T)} |T_1| \cdot |T_2| & \text{otherwise.} \end{cases} \tag{13}$$

To sample all trees in a given $T : \mathbb{T}_2$ uniformly without replacement, we then construct a modular pairing function $\varphi : \mathbb{T}_2 \rightarrow \mathbb{Z}_{|T|} \rightarrow \texttt{BTree}$, that we define as follows:

$$\varphi(T : \mathbb{T}_2, i : \mathbb{Z}_{|T|}) \mapsto \begin{cases} \Big\langle \texttt{BTree}\big(\texttt{root}(T)\big), i \Big\rangle & \text{if } T \text{ is a leaf,} \\ \text{Let } b = |\texttt{children}(T)|, & \\ \quad q_1, r = \big\langle \lfloor \frac{i}{b} \rfloor, i \pmod b \big\rangle, & \\ \quad lb, rb = \texttt{children}[r], & \\ \quad T_1, q_2 = \varphi(lb, q_1), & \\ \quad T_2, q_3 = \varphi(rb, q_2) \text{ in} & \\ \Big\langle \texttt{BTree}\big(\texttt{root}(T), T_1, T_2\big), q_3 \Big\rangle & \text{otherwise.} \end{cases} \tag{14}$$

If the language is suffiently small, instead of sampling trees, we can sample integers uniformly without replacement from $\mathbb{Z}_{|T|}$ then decode them into trees using φ. This procedure is the basis for our sampling algorithm and the method we use to decode repairs from the intersection grammar.

4.6 Ranked repair

Returning to the ranked repair problem (Def. 3.2), the above procedure returns a set of consistent repairs, and we need an ordering over them. We note that any metric is sufficient, such as the log likelihood of the repair under a large language model, Markov chain perplexity, or the probability under a PCFG. We turn to simplest solution: the likelihood of a low-order Markov chain. This solution is computationally fast, and as we will show, already yields competitive results in practice.

Specifically, given a string $\sigma : \Sigma^*$, we factorize the probability $P_\theta(\sigma)$ as a product of conditionals $\prod_{i=1}^{|\sigma|} P_\theta(\sigma_i \mid \sigma_{i-1} \ldots \sigma_{i-n})$, for some small $n \in \mathbb{N}$. To obtain the parameters θ, we use the standard maximum likelihood estimator for Markov chains. We approximate the joint distribution $P(\Sigma^n)$ directly from data, then the conditionals by normalizing n-gram counts with Laplace smoothing.

To score the repairs, we use the conventional length-normalized negative log likelihood:

$$\text{NLL}(\sigma) = -\frac{1}{|\sigma|}\sum_{i=1}^{|\sigma|} \log P_\theta(\sigma_i \mid \sigma_{i-1} \ldots \sigma_{i-n}) \tag{15}$$

Then, for each $\sigma \in A(\underset{\sim}{\sigma})$, we score the repair and return the admissible set in ascending order. That is, we impose an ordering over $A(\underset{\sim}{\sigma})$, where the first repair is most likely under the model, and the last is least likely. To evaluate the accuracy of our ranking, 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, $R : \Sigma^* \rightarrow 2^{\Sigma^*}$ and a test set $\mathcal{D}_{\text{test}}$, we define Precision@k as:

$$\text{Precision@k}(R) = \frac{1}{|\mathcal{D}_{\text{test}}|} \sum_{\langle \sigma^\dagger, \sigma' \rangle \in \mathcal{D}_{\text{test}}} \mathbb{1}\left[\sigma' \in \underset{\boldsymbol{\sigma} \subset R(\sigma^\dagger), |\boldsymbol{\sigma}| \leq k}{\text{argmax}} \sum_{\sigma \in \boldsymbol{\sigma}} \text{NLL}(\sigma)\right] \tag{16}$$

This is a variation on a standard metric used in information retrieval, and a common way to measure the quality of ranked results in machine translation and recommender systems.

5 EXPERIMENTAL SETUP

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. Precision is determined by checking for lexical equivalence with the ground-truth repair, following Sakkas et al. (2022) [33].

We compare Tidyparse against two separate baselines, Seq2Parse (2022) and Break-It-Fix-It (BIFI) [38] and two datasets. The first dataset [37] consists of 10k naturally-occurring pairs of Python errors and their corresponding human fixes from StackOverflow, and the second consists of synthetically generated errors in natural Python snippets [38]. We compare precision at recovering the ground truth repair across varying edit distances, snippet lengths and latency cutoffs.

We preprocess the data by filtering for broken-fixed snippet pairs shorter than 80 tokens and fewer than four Levenshtein edits apart, whose broken and fixed form is accepted and rejected, respectively, by the Python 3 parser. In our synthetic experiments, we apply the pretrained BIFI breaker to synthetically corrupt Python snippets from the BIFI good code test set, using the clean source as the ground truth repair, and filter broken-fixed snippet pairs by the same criteria.

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 model reported by the authors, with a variable-width beam search to control the number of predictions, and let the model predict the top-k repairs, for $k = \{1, 5, 10, 20 \times 10^5\}$.

The language intersection experiments were conducted on 40 Intel Skylake cores running at 2.4 GHz, with 140 GB of RAM, running bytecode compiled for JVM 17.0.2. To train our scoring function, we use a length-5 variable-order Markov (VOM) 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 samples at varying latency cutoffs. We apply a 30-second latency cutoff for our sampler.

6 EVALUATION

We consider the following research questions for our evaluation:

- **RQ 1**: What properties do natural repairs exhibit? (e.g., error frequency, edit distance)
- **RQ 2**: How precise is our technique at fixing syntax errors? (e.g., vs. Seq2Parse and BIFI)
- **RQ 3**: Which design choices are most significant? (e.g., search vs. sampling, n-gram order)

6.1 Dataset

In the following experiments, we use two datasets of Python snippets. The first is a set of 20,500 pairwise-aligned (broken, fixed) Python code snippets from Wong et al.'s StackOverflow dataset [37] shorter than 80 lexical tokens, whose patch sizes are less than four lexical tokens, ($|\Sigma| = 50, |\underset{\sim}{\sigma}| \leq 40, \Delta(\underset{\sim}{\sigma}, \sigma') < 4$). We preprocess the dataset to lexicalize both the broken and fixed code snippets, then filter the dataset by length and edit distance and depict the cutoffs in the figure below.

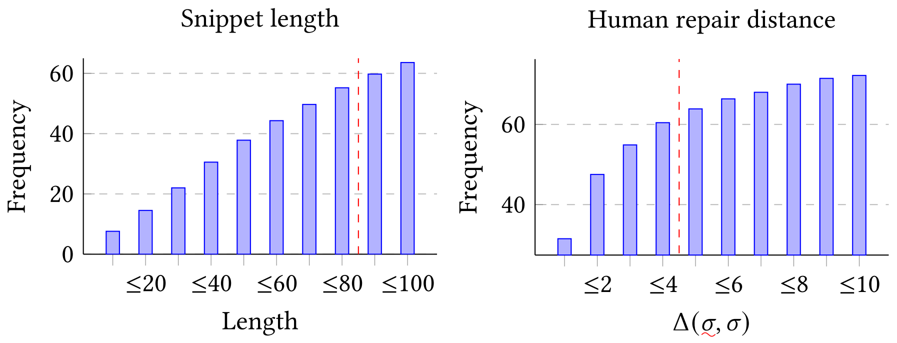


Fig. 4. Repair statistics across the StackOverflow dataset, of which our method can handle about half in ~30s and ~140GB. Larger repairs and Levenshtein radii are possible, albeit requiring additional time and memory.

In the second set of experiments, we use a dataset of valid Python snippets from BIFI [38], then synthetically corrupt them by introducing synthetic syntax errors and measure the Precision@k of our repair procedure at recovering the original, uncorrupted snippet. This set is effectively unlimited as we can generate as many synthetic errors as needed, across a range of edit distances and snippet lengths.

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

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

6.2 StackOverflow evaluation

For our first experiment, we sample the StackOverflow dataset, and run the sampler until the human repair is detected, then measure the number of samples required to draw the exact human repair across varying Levenshtein radii. This gives a sense of how many samples are required on average to saturate the admissible set for repairs of varying Levenshtein distance.

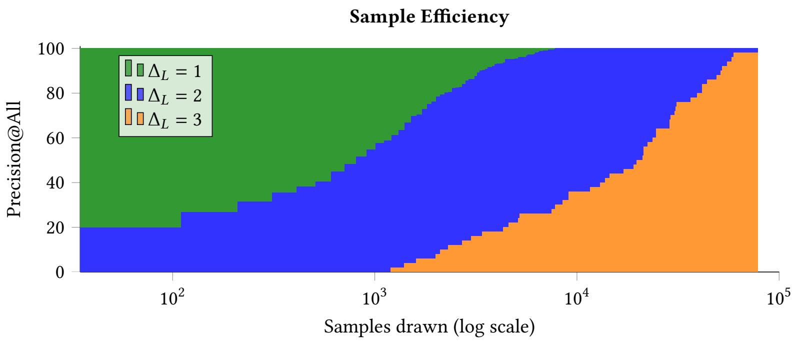


Fig. 5. Sample efficiency of LBH sampler at varying Levenshtein radii. After drawing up to $\sim 10^5$ samples without replacement we can usually retrieve the human repair for almost all repairs fewer than four edits.

The advantage of using an enumerative sampler is that it can terminate early, since it knows, after drawing $|\mathbb{T}_2(\mathcal{G}_\cap)|$ distinct repairs, it has exhausted the admissible set. This also provides a way to calibrate the threshold for how to sample given a fixed budget: if, for example, we can only afford to sample a small fraction of the admissible set, we should clearly sample with replacement. Otherwise, if we can saturate the admissible set, we should sample without replacement.

Next, 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 20k distinct samples.

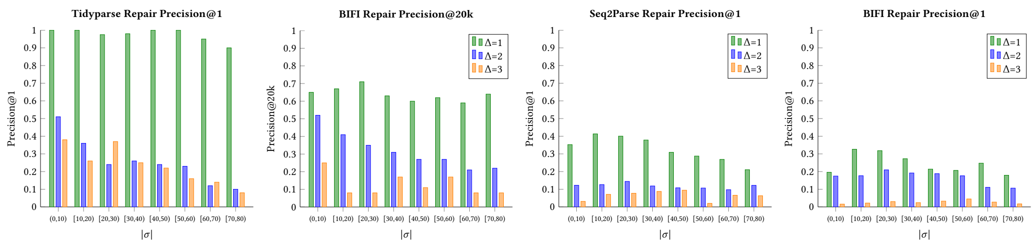


Fig. 6. Tidyparse, Seq2Parse and BIFI repair precision at various lengths and Levenshtein distances.

As we can see, the precision of our method is highly competitive with 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. We report the full data from these experiments in Appendix **??**.

End-to-end throughput varies significantly with the edit distance of the repair. Some errors are trivial to fix, while others require a large number of edits to be sampled before a syntactically valid edit is discovered. We evaluate throughput by sampling edits across invalid strings $|\underset{\sim}{\sigma}| \leq 40$ from the StackOverflow dataset of varying length, and measure the total number of syntactically valid edits discovered, as a function of string length and edit distance $\Delta \in [1, 4]$. Each trial is terminated after 10 seconds, and the experiment is repeated across 7.3k total repairs. Note the y-axis is log-scaled, as the number of admissible repairs increases sharply with edit distance. Our approach discovers a large number of syntactically valid repairs in a relatively short amount of time, and is able to quickly saturate the admissible set for $\Delta(\underset{\sim}{\sigma}, \sigma) \in [1, 4]$ before timeout.

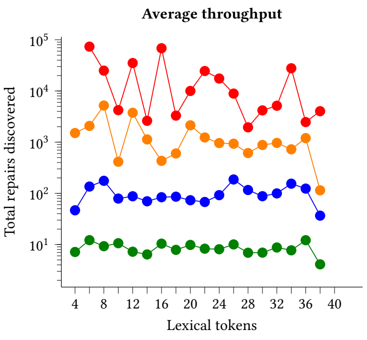


Fig. 7. Distinct repairs found in 30s.

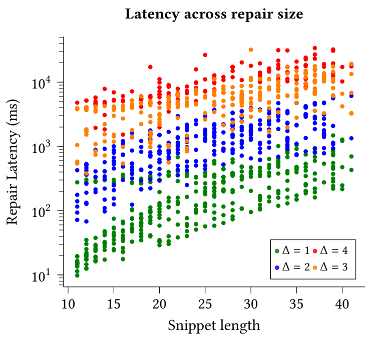


Fig. 8. End-to-end repair timings.

In Fig. 8, we plot the end-to-end repair timings across snippet length. To measure the timings, we collect 1000 repair samples of varying lengths and edit distances, then measure the wallclock time until the sampler retrieves the human repair. For each code snippet, we measure the total time taken for repairs between 1-4 Levenshtein edits. Note the logarithmic scale on the y-axis. Our method is typically able to saturate the admissible set for 1- and 2-edit repairs before the timeout elapses, while 3+-edit throughput is heavily constrained by compute around 16 lexical tokens, when Python's Levenshtein ball approaches a volume of 6×10^8 edits. This bottleneck can be relaxed with a longer timeout or additional CPU cores.

Next, measure the precision at various ranking cutoffs for varying wall-clock timeouts. Here, Precision@{k=1, 5, 10, All} indicates the percentage of syntax errors with a human repair that was located within the top-k results, based on n-gram likelihood.

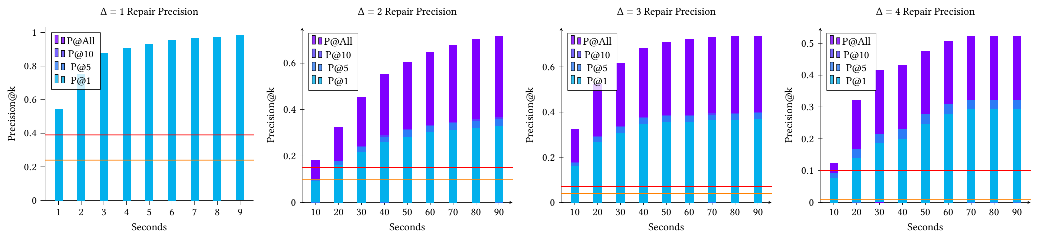


Fig. 9. Human repair benchmarks. Note the y-axis across different edit distance plots has varying ranges. The red line indicates Seq2Parse and the orange line indicates BIFI's Precision@1 on the same repairs.

As can be seen in Fig. 9, our method attains the same precision as Seq2Parse and BIFI for 1-edit repairs at comparable latency, however takes significantly longer for 2- and 3-edit repairs. We note however, that it uses a significantly smaller model and requires no training data. We believe that rewriting the sampler in CUDA or using a more informed prior could significantly improve the latency-performance tradeoff.

6.3 Subcomponent ablation

Previously, we used a rejection-based sampler, which did not sample directly from the admissible set, but the entire Levenshtein ball using a contextual prior, then rejected invalid samples. Although rejection sampling has a much lower minimum latency threshold, i.e., a few seconds, the average time required to attain a fixed precision is much higher. We present the results from the earlier evaluation for comparison below.

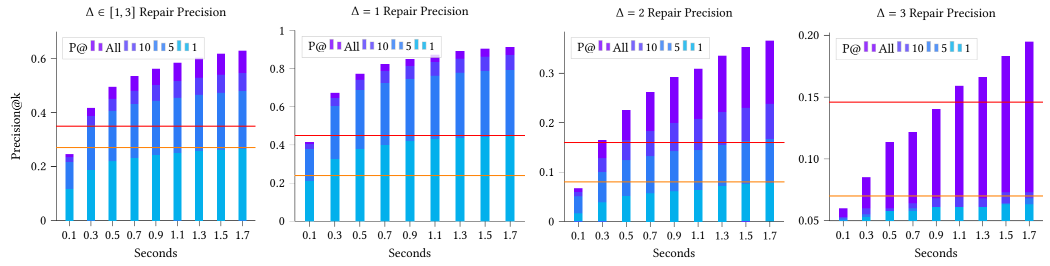
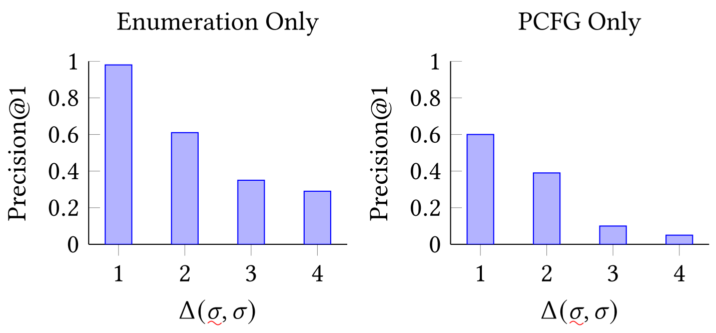


Fig. 10. Adaptive sampling repairs. The red line indicates Seq2Parse Precision@1 on the same dataset. Since it only supports generating one repair, we do not report Precision@k or the intermediate latency cutoffs.

We also evaluate Seq2Parse on the same dataset. Seq2Parse only supports Precision@1 repairs, and so we only report Seq2parse Precision@1 from the StackOverflow benchmark for comparison. Unlike our approach which only produces syntactically correct repairs, Seq2Parse also produces syntactically incorrect repairs and so we report the percentage of repairs matching the human repair for both our method and Seq2Parse. Seq2Parse latency varies depending on the length of the repair, averaging 1.5s for $\Delta = 1$ to 2.7s for $\Delta = 3$, across the entire StackOverflow dataset.

Next, we conduct an ablation study to compare the effectiveness of PCFG sampling vs. the enumerative sampler. In both experiments, we run the corresponding sampler for 30 seconds (either enumerative or PCFG), then rerank all repairs by n-gram perplexity and measure the Precision@1 for each each across varying edit distances.



While the overall precision is notably lower for PCFG sampling than enumeration, the average number of samples drawn is also significantly lower, indicating a relatively higher sample efficiency. This illustrates the tradeoff between sample efficiency and diversity, and suggests that a hybrid approach may be the most effective. When the repair language is very large, a PCFG offers a more informed prior, albeit at the cost of lower coverage.

In general, enumeration has an advantage when the CFL is small. For example, if the CFL contains 2,000 sentences, enumeration will recover all 2,000, whereas PCFG sampling may only recover 100 of the most likely samples. However, if the CFL has 200,000 sentences, enumeration may only be able to recover 10,000 uniform random samples and the PCFG may only recover 5,000, but due to the higher sample efficiency, the PCFG samples are more likely to contain the human repair.

We also measure the relative improvement in throughput (measured by the number of distinct repairs found after 30s) as a function of the number of additional CPU cores, averaged across 1000 trials. We observe from Fig.11 the relative throughput increases logarithmically with the number of additional CPU cores, with at least four CPU cores needed to offset the parallelization overhead. Generally, increasing parallelism only helps when the size of the admissible set is large enough to absorb the additional computation, which is seldom the case for small-radii Levenshtein balls. Further speedups may be possible to realize by rewriting the sampler in CUDA, which we leave for future work.

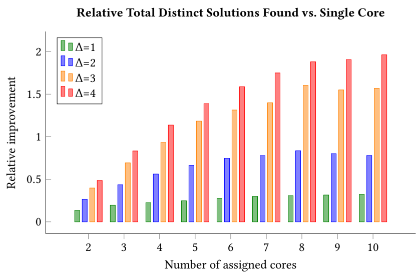


Fig. 11. Observed improvement in throughput relative to total CPU cores assigned.

7 DISCUSSION

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 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.

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 all repairs found we can achieve a significant improvement over one-shot neural repair. We note this approach may not succeed in all languages, and it may be possible to construct pathological cases where the metastability heuristic fails.

Latency can vary depending on many factors including string length and grammar size, and critically the Levenshtein edit distance. This can be an advantage because in the absence of any contextual or statistical information, syntax and Levenshtein edits are often sufficiently constrained to identify a small number of valid repairs. It is also a limitation because as the number of edits grows, the admissible set grows rapidly and the number of valid repairs may become too large to be useful without a good metric, depending on the language and source code snippet under repair.

Tidyparse in its current form has several other technical shortcomings: firstly, it does not incorporate any neural language modeling technology at present, an omission we hope to address. Training a language model to predict likely repair locations and rank admissible results could lead to lower overall latency and more natural repairs. We also hope to explore the use of sequential Monte-Carlo [27] to encourage constrained sampling with diversity.

Lastly and perhaps most significantly, Tidyparse does not incorporate any semantic constraints, so its repairs whilst syntactically admissible, are not guaranteed to be semantically valid. This can be partly alleviated by filtering the results through an incremental compiler or linter, however, the latency introduced may be non-negligible. It is also possible to encode type-based semantic constraints into the solver and we intend to explore this direction more fully in future work.

We envision a few primary use cases for our tool: (1) helping novice programmers become more quickly familiar with a new programming language, (2) autocorrecting common typos among proficient but forgetful programmers, (3) as a prototyping tool for PL designers and educators, and (4) as a pluggable library or service for parser-generators and language servers. Featuring a grammar editor and built-in syntax repair, Tidyparse helps developers navigate the language design space, visualize syntax trees, debug parsing errors and quickly generate simple examples and counterexamples for benchmarking and testing.

8 RELATED WORK

Three important questions arise when repairing syntax errors: (1) is the program broken in the first place? (2) if so, where are the errors located? (3) how should those locations then be altered? In the case of syntax correction, those questions are addressed by three related research areas, (1) parsing, (2) language equations and (3) repair. We survey each of those areas in turn.

8.1 Parsing

Context-free language (CFL) parsing is the well-studied problem of how to turn a string into a unique tree, with many different algorithms and implementations (e.g., shift-reduce, recursive-descent, LR). Many of those algorithms expect grammars to be expressed in a certain form (e.g., left- or right- recursive) or are optimized for a narrow class of grammars (e.g., regular, linear).

General CFL parsing allows ambiguity (non-unique trees) and can be formulated as a dynamic programming problem, as shown by Cocke-Younger-Kasami (CYK) [32], Earley [19] and others. These parsers have roughly cubic complexity with respect to the length of the input string.

As shown by Valiant [36], Lee [25] and others, general CFL recognition is in some sense equivalent to binary matrix multiplication, another well-studied combinatorial problem with broad applications, known to be at worst subcubic. This reduction opens the door to a range of complexity-theoretic speedups to CFL recognition and fast general parsing algorithms.

Okhotin (2001) [30] extends CFGs with language conjunction in *conjunctive grammars*, followed by Zhang & Su (2017) [39] who apply conjunctive language reachability to dataflow analysis.

8.2 Language equations

Language equations are a powerful tool for reasoning about formal languages and their inhabitants. First proposed by Ginsburg et al. [21] for the ALGOL language, language equations are essentially systems of inequalities with variables representing *holes*, i.e., unknown values, in the language or grammar. Solutions to these equations can be obtained using various fixpoint techniques, yielding members of the language. This insight reveals the true algebraic nature of CFLs and their cousins.

Being an algebraic formalism, language equations naturally give rise to a kind of calculus, vaguely reminiscent of Leibniz' and Newton's. First studied by Brzozowski [11, 12] and Antimirov [4], one can take the derivative of a language equation, yielding another equation. This can be interpreted as a kind of continuation or language quotient, revealing the suffixes that complete a given prefix. This technique leads to an elegant family of algorithms for incremental parsing [1, 29] and automata minimization [10]. In our setting, differentiation corresponds to code completion.

In this paper, we restrict our attention to language equations over context-free and weakly context-sensitive languages, whose variables coincide with edit locations in the source code of a computer program, and solutions correspond to syntax repairs. Although prior work has studied the use of language equations for parsing [29], to our knowledge they have never previously been considered for the purpose of code completion or syntax error correction.

8.3 Syntax repair

In finite languages, syntax repair corresponds to spelling correction, a more restrictive and largely solved problem. Schulz and Stoyan [34] construct a finite automaton that returns the nearest dictionary entry by Levenshtein edit distance. Though considerably simpler than syntax correction, their work shares similar challenges and offers insights for handling more general repair scenarios.

When a sentence is grammatically invalid, parsing grows more challenging. Like spelling, the problem is to find the minimum number of edits required to transform an arbitrary string into a syntactically valid one, where validity is defined as containment in a (typically) context-free language. Early work, including Irons [23] and Aho [2] propose a dynamic programming algorithm to compute the minimum number of edits required to fix an invalid string. Prior work on error correcting parsing only considers the shortest edit(s), and does not study multiple edits over the Levenshtein ball. Furthermore, the problem of actually generating the repairs is not well-posed, as there are usually many valid strings that can be obtained within a given number of edits. We instead focus on bounded Levenshtein reachability, which is the problem of finding useful repairs within a fixed Levenshtein distance of the broken string, which requires language intersection.

8.4 Classical program synthesis

There is related work on string constraints in the constraint programing literature, featuring solvers like CFGAnalyzer and HAMPI [24], which consider bounded context free grammars and intersections thereof. Bojańczyk et al. (2014) [9] introduce the theory of nominal automata. Around the same time, D'Antoni et al. (2014) introduce *symbolic automata* [15], a generalization of finite automata which allow infinite alphabets and symbolic expressions over them. Hague et al. (2024) [22] use Parikh's theorem in the context of symbolic automata to speed up string constraint solving, from which we draw partial inspiration for the Levenshtein-Bar-Hillel construction 4.3. In none of the constraint programming literature we surveyed do any of the approaches specifically consider the problem of syntax error correction, which is the main focus of our work.

8.5 Error correcting codes

Our work focuses on errors arising from human factors in computer programming, in particular *syntax error correction*, which is the problem of fixing partially corrupted programs. Modern research on error correction, however, can be traced back to the early days of coding theory when researchers designed *error-correcting codes* (ECCs) to denoise transmission errors induced by external interference, e.g., collision with a high-energy proton, manipulation by an adversary or even typographical mistake. In this context, *code* can be any logical representation for communicating information between two parties (such as a human and a computer), and an ECC is a carefully-designed scheme which ensures that even if some portion of the message should become corrupted, one can still recover the original message by solving a linear system of equations. When designing ECCs, one typically assumes a noise model over a certain sample space, such as the Hamming [16, 35] or Levenshtein [6, 7, 26] balls, from which we draw inspiration for this work.

8.6 Neural program repair

The recent success of deep learning has lead to a variety of work on neural program repair [3, 14, 17]. These approaches typically employ Transformer-based large language models (LLMs) and model the problem as a sequence-to-sequence transformation. Although recent work on circuit lower bounds have cast doubt on the ability of Transformers to truly learn formal languages [13, 28], expressivity aside, these models have been widely adopted for practical program repair tasks. In particular, two papers stand out being closely related to our own: Break-It-Fix-It (BIFI) [38] and Seq2Parse [33].

BIFI adapts techniques from semi-supervised learning to generate synthetic errors in clean code and fixes them. This reduces the amount of pairwise training data, but tends to generalize poorly to length and out-of-distribution repairs. Seq2Parse combines a transformer-based model with an augmented version of the Early parser to suggest error rules, but only suggests a single repair. Our work differs from both in that we suggest multiple repairs at much higher precision, do not require a pairwise repair dataset, and can fix syntax errors in any language with a well-defined grammar. We note our approach is complementary to existing work in neural program repair, and may be used to generate synthetic repairs for training or employ an LLM for ranking.

9 CONCLUSION

Our work, while a case study on syntax repair, is part of a broader line of inquiry in program synthesis that investigates how to combine the strengths of formal language theory and machine learning to build more powerful and flexible programming tools. One approach is to filter the outputs of a generative language model to satisfy a formal specification, typically by constrained sampling. Alternatively, some attempt to use a formal language to guide the search for valid programs via a reinforcement learning or hybrid neurosymbolic approach.

In our work, we take a more pragmatic tack - by incorporating the distance metric into a formal language, we try to exhaustively enumerate repairs by increasing distance, then use the stochastic language model to rank the resulting solutions by naturalness. The more constraints we can encode into the formal language, the more efficient sampling becomes, and the more precise control we have over the output. This reduces the need for training a large, expensive language model to relearn syntax, and allows us to leverage compute for more efficient search.

The great compromise in program synthesis is one of efficiency versus expressiveness. The more expressive a language, the more concise and varied the programs it can represent, but the harder those programs are to synthesize without resorting to domain-specific heuristics. Likewise, the simpler a language is to synthesize, the weaker its concision and expressive power.

Most existing work on program synthesis has focused on general λ-calculi, or narrow languages such as finite sets or regular expressions. The former are too expressive to be efficiently synthesized or verified, whilst the latter are too restrictive to be useful. In our work, we focus on context-free and mildly context-sensitive grammars, which are expressive enough to capture a variety of useful programming language features, but not so expressive as to be unsynthesizable.

The second great compromise in program synthesis is that of reusability versus specialization. In programming, as in human communications, there is a vast constellation of languages, each requiring specialized generators and interpreters. Are these languages truly irreconcilable? Or, as Noam Chomsky argues, are these merely dialects of a universal language? *Synthesis* then, might be a misnomer, and more aptly called *recognition*, in the analytic tradition.

In our work, we argue these two compromises are not mutually exclusive, but complementary and reciprocal. Programs and the languages they inhabit are indeed synthetic, but can be analyzed and reused in the metalanguage of algebraic language theory. Not only does this admit an efficient synthesis algorithm, but allows users to introduce additional constraints without breaking compositionality, one of the most sacred tenets in programming language design.

Over the last few years, there has been a surge of progress in applying language models to write programs. That work is primarily based on methods from differential calculus and continuous optimization, leading to the so-called *naturalness hypothesis*, which suggests programming languages are not so different from natural ones. In contrast, programming language theory takes the view that languages are essentially discrete and finitely-generated sets governed by logical calculi. Programming, thus viewed, is more like a mathematical exercise in constraint satisfaction.

These constraints naturally arise at various stages of syntax validation, type-checking and runtime verification, and help to ensure programs fulfill their intended purpose.

As our work shows, not only is linear algebra over finite fields an expressive language for probabilistic inference, but also an efficient framework for inference on languages themselves. Borrowing analysis techniques from multilinear algebra and tensor completion in the machine learning setting, we develop an equational theory that allows us to translate various decision problems on formal languages into a system of inequalities over finite fields. We demonstrate the effectiveness of our approach for syntax repair in context-free languages, and show that our approach is competitive with state-of-the-art methods in terms of both accuracy and efficiency. In future work, we hope to extend our method to more natural grammars like conjunctive languages, TAG, LCFRS and other mildly context-sensitive languages.

Syntax correction tools should be as user-friendly and widely-accessible as autocorrection tools in word processors. From a practical standpoint, 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 can be used in a live programming environment.

Despite its computational complexity, the design of the tool itself is relatively simple. Tidyparse accepts a context-free language and a string. If the string is valid, it returns the parse forest, otherwise, it returns a set of repairs, ordered by likelihood. By allowing the string to contain holes, repairs may contain either concrete tokens or nonterminals, which can be expanded by the user or a neural-guided search procedure to produce a valid program. This approach to parsing has many advantages, enabling us to repair syntax errors, correct typos and recover from errors in real time, as well as being provably sound and complete with respect to the grammatical specification. It is also compatible with neural program synthesis and repair techniques and shares the same masked language modeling (MLM) target used by Transformer-based LLMs.

We have implemented our approach as an IDE plugin and demonstrated its viability as a practical tool for realtime programming. A considerable amount of effort was devoted to supporting fast error correction functionality. Tidyparse is capable of generating repairs for invalid code in a range of practical languages with very little downstream language integration required. We plan to continue expanding its grammar and autocorrection functionality to cover a broader range of languages and hope to conduct a more thorough user study to validate its effectiveness.

REFERENCES

[1] Michael D Adams, Celeste Hollenbeck, and Matthew Might. 2016. On the complexity and performance of parsing with derivatives. In Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation. 224–236.

[2] Alfred V Aho and Thomas G Peterson. 1972. A minimum distance error-correcting parser for context-free languages. SIAM J. Comput. 1, 4 (1972), 305–312.

[3] Miltiadis Allamanis, Henry Jackson-Flux, and Marc Brockschmidt. 2021. Self-supervised bug detection and repair. Advances in Neural Information Processing Systems 34 (2021), 27865–27876.

[4] Valentin Antimirov. 1996. Partial derivatives of regular expressions and finite automaton constructions. Theoretical Computer Science 155, 2 (1996), 291–319.

[5] Yehoshua Bar-Hillel, Micha Perles, and Eli Shamir. 1961. On formal properties of simple phrase structure grammars. Sprachtypologie und Universalienforschung 14 (1961), 143–172.

[6] Daniella Bar-Lev, Tuvi Etzion, and Eitan Yaakobi. 2021. On Levenshtein Balls with Radius One. In 2021 IEEE International Symposium on Information Theory (ISIT). 1979–1984. https://doi.org/10.1109/ISIT45174.2021.9517922

[7] Leonor Becerra-Bonache, Colin de La Higuera, Jean-Christophe Janodet, and Frédéric Tantini. 2008. Learning Balls of Strings from Edit Corrections. Journal of Machine Learning Research 9, 8 (2008).

[8] Richard Beigel and William Gasarch. [n.d.]. A Proof that if $L = L_1 \cap L_2$ where L_1 is CFL and L_2 is Regular then L is Context Free Which Does Not use PDA's. http://www.cs.umd.edu/~gasarch/BLOGPAPERS/cfg.pdf

[9] Mikołaj Bojańczyk, Bartek Klin, and Sławomir Lasota. 2014. Automata theory in nominal sets. Logical Methods in Computer Science 10 (2014).

[10] Janusz A Brzozowski. 1962. Canonical regular expressions and minimal state graphs for definite events. In Proc. Symposium of Mathematical Theory of Automata. 529–561.

[11] Janusz A Brzozowski. 1964. Derivatives of regular expressions. Journal of the ACM (JACM) 11, 4 (1964), 481–494.

[12] Janusz A. Brzozowski and Ernst Leiss. 1980. On equations for regular languages, finite automata, and sequential networks. Theoretical Computer Science 10, 1 (1980), 19–35.

[13] David Chiang, Peter Cholak, and Anand Pillay. 2023. Tighter Bounds on the Expressivity of Transformer Encoders. arXiv preprint arXiv:2301.10743 (2023).

[14] Nadezhda Chirkova and Sergey Troshin. 2021. Empirical study of transformers for source code. In Proceedings of the 29th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering. 703–715.

[15] Loris D'Antoni and Margus Veanes. 2014. Minimization of symbolic automata. In Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. 541–553.

[16] Dingding Dong, Nitya Mani, and Yufei Zhao. 2023. On the number of error correcting codes. Combinatorics, Probability and Computing (2023), 1–14. https://doi.org/10.1017/S0963548323000111

[17] Dawn Drain, Chen Wu, Alexey Svyatkovskiy, and Neel Sundaresan. 2021. Generating bug-fixes using pretrained transformers. In Proceedings of the 5th ACM SIGPLAN International Symposium on Machine Programming. 1–8.

[18] Philippe Duchon, Philippe Flajolet, et al. 2004. Boltzmann samplers for the random generation of combinatorial structures. Combinatorics, Probability and Computing 13, 4-5 (2004), 577–625.

[19] Jay Earley. 1970. An efficient context-free parsing algorithm. Commun. ACM 13, 2 (1970), 94–102.

[20] Denis Firsov and Tarmo Uustalu. 2015. Certified normalization of context-free grammars. In Proceedings of the 2015 Conference on Certified Programs and Proofs. 167–174.

[21] Seymour Ginsburg and H Gordon Rice. 1962. Two families of languages related to ALGOL. Journal of the ACM (JACM) 9, 3 (1962), 350–371.

[22] Matthew Hague, Artur Jeż, and Anthony W Lin. 2024. Parikh's Theorem Made Symbolic. Proceedings of the ACM on Programming Languages 8, POPL (2024), 1945–1977.

[23] E. T. Irons. 1963. An Error-Correcting Parse Algorithm. Commun. ACM 6, 11 (nov 1963), 669–673. https://doi.org/10.1145/368310.368385

[24] Adam Kiezun, Vijay Ganesh, Philip J Guo, Pieter Hooimeijer, and Michael D Ernst. 2009. HAMPI: a solver for string constraints. In Proceedings of the eighteenth international symposium on Software testing and analysis. 105–116.

[25] Lillian Lee. 2002. Fast context-free grammar parsing requires fast boolean matrix multiplication. Journal of the ACM (JACM) 49, 1 (2002), 1–15. https://arxiv.org/pdf/cs/0112018.pdf

[26] Vladimir I Levenshtein et al. 1966. Binary codes capable of correcting deletions, insertions, and reversals. In Soviet physics doklady, Vol. 10. 707–710. https://nymity.ch/sybilhunting/pdf/Levenshtein1966a.pdf

[27] Alexander K Lew, Tan Zhi-Xuan, Gabriel Grand, and Vikash K Mansinghka. 2023. Sequential monte carlo steering of large language models using probabilistic programs. arXiv preprint arXiv:2306.03081 (2023).

[28] William Merrill, Ashish Sabharwal, and Noah A Smith. 2022. Saturated transformers are constant-depth threshold circuits. Transactions of the Association for Computational Linguistics 10 (2022), 843–856.

[29] Matthew Might, David Darais, and Daniel Spiewak. 2011. Parsing with derivatives: a functional pearl. ACM sigplan notices 46, 9 (2011), 189–195.

[30] Alexander Okhotin. 2001. Conjunctive grammars. Journal of Automata, Languages and Combinatorics 6, 4 (2001), 519–535.

[31] Rohit J. Parikh. 1966. On Context-Free Languages. J. ACM 13, 4 (oct 1966), 570–581. https://doi.org/10.1145/321356.321364

[32] Itiroo Sakai. 1961. Syntax in universal translation. In Proceedings of the International Conference on Machine Translation and Applied Language Analysis.

[33] Georgios Sakkas, Madeline Endres, Philip J Guo, Westley Weimer, and Ranjit Jhala. 2022. Seq2Parse: neurosymbolic parse error repair. Proceedings of the ACM on Programming Languages 6, OOPSLA2 (2022), 1180–1206.

[34] Klaus U Schulz and Stoyan Mihov. 2002. Fast string correction with Levenshtein automata. International Journal on Document Analysis and Recognition 5 (2002), 67–85.

[35] Michalis K Titsias and Christopher Yau. 2017. The Hamming ball sampler. J. Amer. Statist. Assoc. 112, 520 (2017), 1598–1611.

[36] Leslie G Valiant. 1975. General context-free recognition in less than cubic time. Journal of computer and system sciences 10, 2 (1975), 308–315. http://people.csail.mit.edu/virgi/6.s078/papers/valiant.pdf

[37] Alexander William Wong, Amir Salimi, Shaiful Chowdhury, and Abram Hindle. 2019. 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). IEEE, 318–322.

[38] Michihiro Yasunaga and Percy Liang. 2021. Break-it-fix-it: Unsupervised learning for program repair. In International Conference on Machine Learning. PMLR, 11941–11952.

[39] Qirun Zhang and Zhendong Su. 2017. Context-sensitive data-dependence analysis via linear conjunctive language reachability. In Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages. 344–358.

A EXAMPLE REPAIRS

Below, we provide a few examples of broken code snippets and their corresponding human repairs that were successfully discovered and ranked first by our method. On the left is a complete snippet fed to the model and on the right, the corresponding human repair that was correctly predicted.

Original broken code	First predicted repair

`form sympy import *`
`x = Symbol('x', real=True)`
`x, re(x), im(x)`
	`from sympy import *`
	`x = Symbol('x', real=True)`
	`x, re(x), im(x)`

`result = yeald From(item.create())`
`raise Return(result)`
	`result = yield From(item.create())`
	`raise Return(result)`

`return 1/sum_p if sum_p \`
`return 0 else`
	`return 1/sum_p if sum_p \`
	`else 0`

`sum(len(v) for v items.values()))`	`sum(len(v) for v in items.values())`
`df.apply(lambda row: list(set(row['ids'))))`	`df.apply(lambda row: list(set(row['ids'])))`

`import numpy ad np`
`A_concate = np.array([a_0, a_1, a_2,..., a_n])`
	`import numpy as np`
	`A_concate = np.array([a_0, a_1, a_2,..., a_n])`

`class MixIn(object)`
`  def m():`
`    pass`
 	
`class classA(MixIn):`
 	
`class classB(MixIn):`
	`class MixIn(object):`
	`  def m():`
	`    pass`
	
	`class classA(MixIn): pass`
	
	`class classB(MixIn): pass`

B RAW DATA

Raw data from Precision@k experiments across snippet length and Levenshtein distance from § 6.2. $|\underset{\sim}{\sigma}|$ indicates the snippet length and Δ indicates the Levenshtein distance between the broken and code and human fix computed over lexical tokens. For Tidyparse, we sample until exhausting the admissible set or a timeout of 30s is reached, whichever happens first, then rank the results. For the other models Precision@1, we sample one repair and report the percentage of repairs matching the human repair. For Precision@All, we report the percentage of repairs matching the human repair within the top 20000 samples.

	Δ	Precision@1							
$\lvert\underset{\sim}{\sigma}\rvert$		(0, 10)	[10, 20)	[20, 30)	[30, 40)	[40, 50)	[50, 60)	[60, 70)	[70, 80)
Tidyparse	1	1.00	1.00	0.98	0.98	1.00	1.00	0.95	0.90
	2	0.51	0.36	0.24	0.26	0.24	0.23	0.12	0.10
	3	0.38	0.26	0.37	0.25	0.22	0.16	0.14	0.08
Seq2Parse	1	0.35	0.41	0.40	0.37	0.31	0.29	0.27	0.21
	2	0.12	0.13	0.14	0.12	0.11	0.11	0.10	0.12
	3	0.03	0.07	0.08	0.09	0.09	0.02	0.07	0.06
BIFI	1	0.20	0.33	0.32	0.27	0.21	0.21	0.25	0.18
	2	0.18	0.18	0.21	0.19	0.19	0.18	0.11	0.11
	3	0.02	0.02	0.03	0.02	0.03	0.05	0.03	0.02
		Precision@All							
Tidyparse	1	1.00	1.00	0.98	0.98	1.00	1.00	1.00	0.91
	2	0.91	0.89	0.85	0.82	0.68	0.82	0.58	0.50
	3	0.50	0.37	0.53	0.40	0.44	0.27	0.34	0.22
BIFI	1	0.65	0.67	0.70	0.65	0.60	0.62	0.60	0.64
	2	0.52	0.41	0.37	0.32	0.27	0.27	0.21	0.24
	3	0.20	0.13	0.08	0.17	0.15	0.18	0.17	0.07

Synthetic evaluation

	Δ	Precision@1							
$\lvert\underset{\sim}{\sigma}\rvert$		(0, 10)	[10, 20)	[20, 30)	[30, 40)	[40, 50)	[50, 60)	[60, 70)	[70, 80)
Tidyparse	1	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.98
	2	0.80	0.62	0.63	0.72	0.65	0.81	0.64	0.62
	3	0.32	0.16	0.25	0.42	0.29	0.39	0.38	0.32
		Precision@All							
Tidyparse	1	1.00	1.00	1.00	1.00	1.00	1.00	0.98	1.00
	2	1.00	0.99	0.92	0.95	0.98	0.97	0.89	0.90
	3	1.00	0.96	0.93	0.85	0.87	0.90	0.84	0.72