Syntax Repair as Language Intersection

BREANDAN CONSIDINE, JIN GUO, and XUJIE SI

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 construct a grammar representing the language intersection between a Levenshtein automaton and a context-free grammar, then decode it in order of n-gram likelihood.

1 INTRODUCTION

Syntax errors are a familiar nuisance for programmers, arising due to a variety of factors, from inexperience, typographic error, 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.

Early work on syntax repair by Irons [30] and Aho [2] use techniques from dynamic programming to find the nearest parse trees for an erroneous input. These methods guarantee correctness, but do not attempt to completely recover all nearby corrections. Instead they find just one or a small number of corrections, which are not necessarily the most likely or natural repairs. Nevertheless, these methods are appealing for their interpretability and well-understood algorithmic properties.

More recently, probabilistic repair techniques have been introduced using neural language models to predict the most likely correction [3, 41, 47]. While these techniques generate far more natural edits, they are prone to misgeneralization, costly to train, and challenging to incorporate new constraints thereafter. Furthermore, the generated repairs are not necessarily sound without additional filtering, and we observe the released models often hallucinate false positive repairs.

Recent work by Merrill et al. [35] and Chiang et al. [15] suggest that the issue may be more foundational: transformer-based language models, a popular class of neural language models used in probabilistic program repair, are fundamentally less expressive than context-free grammars, which formally describe the syntax of most programming languages. This suggests such models, despite their useful approximation properties, are ill-suited for the task of end-to-end syntax repair. Yet, they may still be useful for resolving ambiguity between valid repairs of differing likelihood.

In this work, we consider the problem of ranked syntax repair under finite Levenshtein bounds. We demonstrate it is possible to attain a significant advantage over state-of-the-art neural repair techniques by exhaustively retrieving every valid Levenshtein edit in a certain distance and scoring it. Not only does this approach guarantee both soundness and completeness, we find it also improves precision when ranking by naturalness. Our proposed solution 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) We decode the data structure, returning a stream of concrete syntax repairs which is guaranteed to halt, but can be terminated at any time beforehand. The repairs are all sound, reachable within a few edits, and the most likely repairs are returned first. (§ ??)

Authors' address: Breandan Considine, bre@ndan.co; Jin Guo, jguo@cs.mcgill.ca; Xujie Si, six@utoronto.ca.

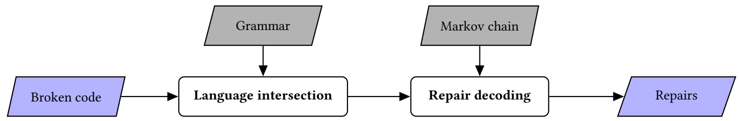


Fig. 1. Simplified architecture. Given a grammar and broken code fragment, we return a set of likely repairs.

Our primary technical contributions are (1) the adaptation of the Levenshtein automaton and Bar-Hillel construction to syntax repair and (2) a method for enumerating or sampling valid sentences in finite context-free languages in order of naturalness, as seen in Fig. 1. The efficacy of our technique owes to the fact it does not synthesize likely edits, but unique, fully-formed repairs within a given 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.

2 EXAMPLE

Syntax errors are usually fixable with a small number of edits. If we assume the intended repair contains just a few edits, this imposes strong locality constraints on the space of possible edits. For example, let us 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 samples – 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 will need a more efficient procedure for sampling all and only small valid repairs.

We will now give an informal intuition behind of our method, then proceed to formalize it in the following sections. By way of illustration, suppose we have a string `( ) )`, and wish to find nearby repairs. There is a nondeterministic finite automaton, called the Levenshtein automaton, recognizing every single string that can be formed by inserting, substituting or deleting a parenthesis. We will use a variant that removes some unnecessary edges, but does not affect the generated language.

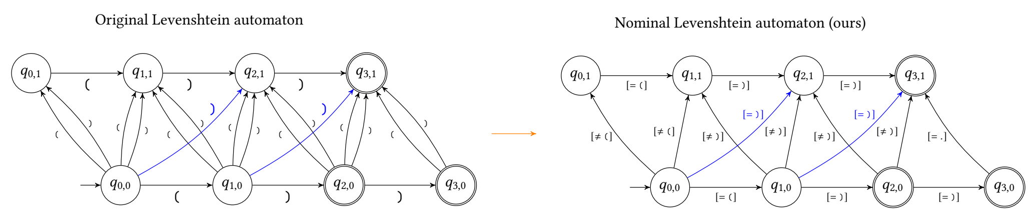


Fig. 2. Automaton recognizing every 1-edit patch. We nominalize the original automaton, ensuring upwards arcs denote a mutation, and use a symbolic predicate, which deduplicates parallel arcs in large alphabets.

Let us consider a very simple grammar: $G = \{S \to () \mid (S) \mid SS\}$. For convenience, G can be reduced into an equivalent normal form, G', by refactoring the right hand side of each production to be in either binary or unary form: $G' = \{S \to LR, S \to LI, S \to SS, I \to SR, L \to (, R \to)\}$. Now, we proceed to stitch together the automaton and grammar into new grammar, $G_\cap$, that will recognize every string in the intersection of their respective languages, and no other string.

This stitching process is known in the literature as the Bar-Hillel (BH) construction, and applies to any context-free grammar and nondeterministic finite automaton. It has three rules: first, for every initial state in the automaton (here there is just one, q_{00}) and every final state, (q_{20}, q_{30}, q_{31}), there will be a production $S \to q_{00}Sq_{xy}$. We call this rule $\sqrt{}$, and leave it unchanged.

Next, the BH construction states: for every production $A \to a$ and arc $q \overset{a}{\to} q'$ in the automaton, there will be a production, $qAq' \to a$ in the intersection grammar. Here is where our variation, the Levenshtein Bar-Hillel (LBH) construction, now diverges: we only consider unit productions matching the nominal predicate depicted in Fig. 2, and will call this modified rule $\hat{\uparrow}$.

So far, the intersection grammar now contains the following productions:

$\sqrt{}$	$\hat{\uparrow}$			
$S \to q_{00}Sq_{20}$	$q_{00}Rq_{01} \to)$	$q_{10}Lq_{11} \to ($	$q_{20}Rq_{21} \to)$	$q_{01}Lq_{11} \to ($
$S \to q_{00}Sq_{30}$	$q_{00}Rq_{11} \to)$	$q_{10}Lq_{21} \to ($	$q_{20}Rq_{31} \to)$	$q_{11}Rq_{21} \to)$
$S \to q_{00}Sq_{31}$	$q_{00}Lq_{10} \to ($	$q_{10}Rq_{20} \to)$	$q_{20}Lq_{30} \to ($	$q_{21}Rq_{31} \to)$
	$q_{30}Lq_{31} \to ($	$q_{10}Rq_{31} \to)$	$q_{30}Rq_{31} \to)$	$q_{00}Rq_{21} \to)$

The final and most expensive rule of the BH construction, $\bowtie$, stipulates: for every $w \to xz$ in G' and every state triplet $\langle p, q, r\rangle$, we will have $pwr \to (pxq)(qzr)$ in $G_\cap$. This rule creates a synthetic production recognizing every combination of NFA states consistent with every nonterminal in every production. Fortunately, most of these combinations are simply impossible. For example, we note that $q_{31}Sq_{00}$ is an impossible nonterminal, as there is no path from q_{31} to q_{00} in the Levenshtein automaton (Fig. 2). Likewise, the nonterminal $q_{i,j}Iq_{i+1,j}$ is clearly impossible, as the nonterminal I requires at least three tokens, and the only path from $q_{i,j}$ to $q_{i+1,j}$ has length one. Criteria such as these are essential for filtering out synthetic nonterminals generated by the original BH $\bowtie$ rule.

In Sec. 4.3, we will consider a refinement to the $\bowtie$ rule which eliminates many impossible productions from BH intersection grammars by a sound overapproximation to the Parikh image. Aggressively pruning synthetic productions from $G_\cap$ by (1) nominalizing the Levenshtein automaton and (2) refining the $\bowtie$ rule, are key insights to unlocking the full potential of the Bar-Hillel construction and scaling up this technique to handle real-world program repair scenarios.

3 PROBLEM STATEMENT

Source code in a programming language can be treated as a string over a finite alphabet, Σ. 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 3.1 (Bounded Levenshtein-CFL reachability). Given a CFL, ℓ, and an invalid string, $\underset{\sim}{\sigma} : \bar{\ell}$, 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 $\mathrm{P}_\theta : \Sigma^* \to [0, 1] \subset \mathbb{R}$, the ranked repair problem is to find the top-k maximum likelihood repairs under the language model. That is,

$$R(A, P_\theta) = \underset{\boldsymbol{\sigma} \subseteq A, |\boldsymbol{\sigma}| \leq k}{\operatorname{argmax}} \sum_{\sigma \in \boldsymbol{\sigma}} \mathrm{P}_\theta(\sigma) \tag{1}$$

A popular approach to ranked repair involves learning a distribution over strings, however this is highly sample-inefficient and generalizes poorly to new languages. Approximating a distribution over Σ^* forces the model to jointly learn syntax and stylometry. Furthermore, 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 $L(\underset{\sim}{\sigma}, d)$, reject duplicates, 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 neural language models.

As we will demonstrate, the ranked repair problem can be factorized into a bilevel objective: first maximal retrieval, then ranking. Instead of working with strings, we will explicitly construct a grammar which soundly and completely generates the set $\ell \cap L(\underset{\sim}{\sigma}, d)$, then retrieve repairs from its language. By ensuring retrieval is sufficiently precise and exhaustive, maximizing likelihood over the retrieved set can be achieved with a much simpler, syntax-oblivious language model.

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 [24]. 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 3.3 (Linear convergence). Given a finite CFL, ℓ, we want a randomized generating function, $\boldsymbol{\varphi} : \mathbb{N}_{\leq|\ell|} \to 2^\ell$, whose rate of convergence is linear in expectation, i.e., $\mathbb{E}_{i \in [1,n]} |\boldsymbol{\varphi}(i)| \propto n$.

To satisfy Def. 3.3, we construct a bijection from syntax trees to integers (§ 4.5), 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. As long as $|\ell_\cap|$ is sufficiently small and enough samples are drawn, $\boldsymbol{\varphi}$ is sure to include the most natural repairs, and additionally, will terminate after exhausting all sentences.

Finally, once we have a set of small and valid repairs, the problem of ranked repair reduces to sorting retrieved samples by likelihood, which can be approximated using an autoregressive language model or any suitable scoring function of the implementer's choice. In our case, we use a low-order Markov model for its inference speed, data efficiency, and simplicity.

4 METHOD

The method we describe in this paper takes as input the invalid code fragment, and returns a set of plausible repairs. We assume to know the target syntax and a low-rank distribution of lexical n-grams to estimate the likelihood of candidate repairs. At a high level, our method can be decomposed into two main steps: (1) language intersection, (2) repair decoding.

First, we generate a synthetic grammar representing the intersection between the syntax and the Levenshtein ball around the source code, then during extraction, we retrieve as many repairs as possible from the intersection grammar via sampling or enumeration. This can be depicted in more detail as a flowchart (Fig. 3).

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 programming language syntax (G) and an automaton recognizing the Levenshtein edit ball of a given radius, $L(\sigma, d)$. As the CFL family is closed under intersection with regular languages, the intersection language $\mathcal{L}(G_\cap)$ should contain every repair within a given Levenshtein distance and no invalid repairs. Either the grammar will be empty, in which case there are no repairs within the given radius, or it will be nonempty, in which case we can directly proceed to decode repairs.

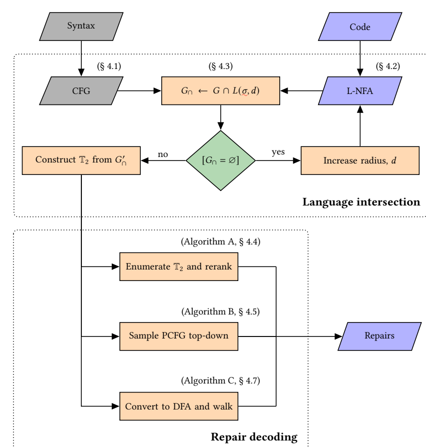


Fig. 3. Dataflow of our proposed method.

To decode the repairs, we present three basic methods: (A) enumerate the CFG, $G_\cap$, and rerank each sentence, (B) sample $G_\cap$ with learned PCFG transitions, and then rerank, or (C) translate $G_\cap$ to an equivalent DFA, $\mathcal{A}_\cap$, minimize it using Brzozowski's algorithm to produce $\mathcal{A}^*_\cap$, then sample trajectories without replacement through the DFA according to a Markov chain until a fixed timeout is reached. We describe (Alg. A) and (Alg. B) in § 4.6, and (Alg. C) in § 4.7.

In all cases, if the language is sufficiently small, this will generate every possible repair and halt early. Otherwise, if the language is too large to exhaustively search, it will draw a representative subset containing the most likely repairs with high probability, then halt. The decoders (A-C) differ in the order which they retrieve repairs, and the likelihood model they use to rank them.

We will first describe how to generate the intersection grammar (§ 4.2, 4.3), then, describe a data structure compactly representing its language, allowing us to efficiently decode all repairs contained within (§ 4.5). Optionally, we can choose to rerank the repairs by a more sophisticated language model, such as a neural network, to improve the naturalness of the top-k repairs (§ **??**).

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 ($P\colon V \rightarrow (V \mid \Sigma)^*$), and a start symbol, ($S$). Every CFG is reducible to so-called *Chomsky Normal Form*, $P'\colon V \rightarrow (V^2 \mid \Sigma)$, where every production is either (1) a binary production $w \rightarrow xz$, or (2) a unit production $w \rightarrow t$, 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 lexical edits with the nominal Levenshtein automaton

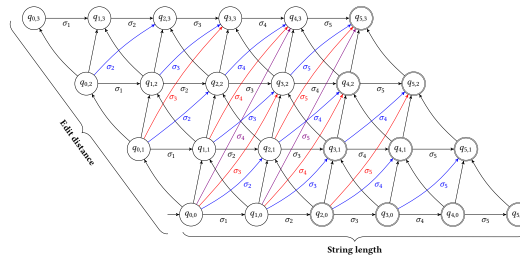


Fig. 4. NFA recognizing Levenshtein $L(\sigma : \Sigma^5, 3)$.

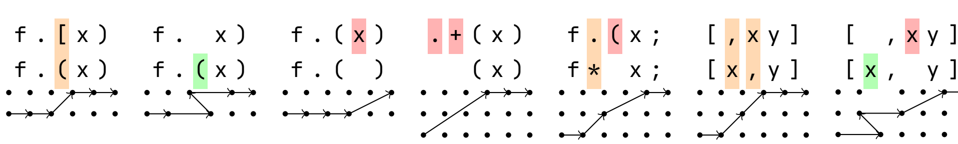
Levenshtein edits are recognized by an automaton known as the Levenshtein automaton. As the original construction defined by Schultz and Mihov [42] contains cycles and ε-transitions, we propose a variant which is ε-free and acyclic. Furthermore, we adopt a nominal form which supports infinite alphabets and considerably simplifies the language intersection to follow. Illustrated in Fig. 4 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 within 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.

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

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

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

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.



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.

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

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

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 edits 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:

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

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 $\langle q, v, q'\rangle$ matches a substring simultaneously recognized by (1) a pair of states q, q' in the original NFA and (2) a nonterminal, v, 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 $p : \Sigma^* \rightarrow \mathbb{N}^{|\Sigma|}$ be the Parikh operator [38], which counts the frequency of terminals in a string. We define the Parikh map, $\pi : V \rightarrow \Pi$, 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 $\pi : Q \times Q \rightarrow \Pi$ 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 is nonzero, 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 (Parikh compatibility). Let q, q' be NFA states and v be a CFG nonterminal. We call $\langle q, v, q'\rangle : Q \times V \times Q$ *compatible* iff their divergence is zero, i.e., $v \lhd qq' \iff (\pi(v) \parallel \pi(q, q')) = 0$.

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

$$\frac{(A \to a) \in P \quad (q \xrightarrow{[\cdot]} r) \in \delta \quad a[\cdot]}{(qAr \to a) \in P_\cap}\hat{\uparrow} \qquad \frac{w \lhd pr \quad x \lhd pq \quad z \lhd qr \quad (w \to xz) \in P \quad p, q, r \in Q}{\big(pwr \to (pxq)(qzr)\big) \in P_\cap}\hat{\bowtie}$$

Once constructed, we normalize $G_\cap$ by removing unreachable and non-generating productions [23] to obtain $G'_\cap$, which is a recognizer for the admissible set, i.e., $\mathcal{L}(G'_\cap) = A$, satisfying Def. 3.1.

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 eventually generate all repairs in the language, preferably in a natural order. 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 length-bounded Parikh map for the Levenshtein Bar-Hillel (LBH) construction, and a method for sampling the language with or without replacement.

4.4 Code completion 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, porous synthesizer, parser, generator, Parikh map and other convenient structures for CFL intersection and membership.

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:

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

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. The fixedpoint iteration proceeds as follows:

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

Once obtained, the proposition $[S \in \Lambda^*_\sigma]$ decides language membership, i.e., $[\sigma \in \mathcal{L}(\mathcal{G})]$ [1]. So far, this procedure is essentially the textbook CYK algorithm in a linear algebraic notation [26].

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 $\mathrm{H}(\sigma)$. Given a *porous string*, $\sigma : \underline{\Sigma}^*$ we seek all syntactically valid inhabitants, i.e., $A(\sigma) = \mathrm{H}(\sigma) \cap \ell$.

Let us consider an example with two holes, $\sigma = 1$ _ _, and the context-free grammar being $G = \{S \to NON, O \to + \mid \times, N \to 0 \mid 1\}$. This grammar will first be rewritten into CNF as $G' = \{S \to NL, N \to 0 \mid 1, O \to \times \mid +, L \to ON\}$. Using the powerset algebra we just defined, the matrix fixpoint $M' = M + M^2$ can be computed as follows, shown in the leftmost column below:

[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 solution $\sigma' \in A$, but M_∞ does not explicitly reveal its identity.

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 $X \boxplus Z = [X_i \lor Z_i]_{i \in [0,|V|)}$, mirroring $\oplus, \otimes$ from the powerset domain, now over bitvectors. 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 $V_{0,2}, V_{1,3}$:

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

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

Now we solve for the corner entry $V_{0,3}$ by dotting the first row and last column, which yields:

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

Since we only care about $V_{0,3,3} \Leftrightarrow [S \in V_{0,3}]$, 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 $\sigma = 1\ \underline{\text{O}}\ \underline{\text{N}}$ 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 inhabitants, 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 arborescent structure.

We will now describe a datatype for compactly representing CFL parse forests, then redefine the matrix algebra over this domain. This datatype is particularly convenient for tracking provenance under ambiguity, constructing the Parikh map for a CFG, counting the size of a finite CFL, and sampling parse trees with or without replacement.

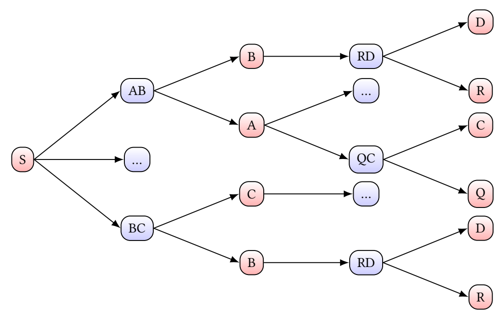


Fig. 5. A partial $\mathbb{T}_2$ corresponding to the grammar $\{S \to BC \mid \ldots \mid AB, B \to RD \mid \ldots, A \to QC \mid \ldots\}$.

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:

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

Depicted in Fig. 5 is a partial $\mathbb{T}_2$, where red nodes are `roots` and blue nodes are `children`. The shape of type $\mathbb{T}_2$ is congruent with an acyclic CFG in Chomsky Normal Form, i.e., $\mathbb{T}_2 \cong \mathcal{G}'$, so assuming the CFG recognizes a finite language, as is the case for $G'_\cap$, then it can be translated directly. Since the RHS of CNF productions must each be nonterminals, we define $P(a)$ as $\Sigma + V + VL\big(V^2P(a)^2\big)$, otherwise, we could write $\Sigma + V + VL\big(P(a)^2\big)$ to allow productions containing mixed terminals and nonterminals.

If the language is infinite, we first slice the CFL, $\mathcal{L}(G) \cap \Sigma^n$, and solve the fixpoint for each slice $n \in [2, n]$. Given a porous string $\sigma : \underline{\Sigma}^n$ representing the slice, we can construct $\mathbb{T}_2$ from the bottom-up, and read off structures from the top-down. We construct the first upper diagonal $\hat{\sigma}_r = \Lambda(\sigma_r)$ as follows:

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

This initializes the superdiagonal entries of M_0, 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 of a CFG (Def. 4.2), we may use the following recurrence,

[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.

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

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

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

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

To obtain the parameterized Parikh map (PPM) of a length-bounded CFG, we abstractly parse the porous string and take the union of all intervals, which subsumes the Parikh image of every repair in the Levenshtein ball. Given a specific programming language syntax, G, the following function can be precomputed and cached for all $v : V$, and reasonable values of $m, n : \mathbb{N}$ for the sake of efficiency, then used to retrieve the Levenshtein-Parikh-$\langle v, m, n\rangle$ map in constant time:

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

By constructing the PPM for a grammar, G, we are effectively precomputing conditional upper and lower bounds on the Parikh image of any string generated by v whose length falls within a fixed interval – conditioned on that interval. Given a pair of FSA states $q, q' : Q$, let m and n be the greatest and least values, respectively, such that for all $\sigma \in \mathcal{L}(q \Longrightarrow q'), |\sigma| \in [m, n]$. To obtain the corresponding Parikh map for each $\langle q, v, q'\rangle$-triplet in $\hat{\bowtie}$, we can then directly lookup $\pi(G, v, m, n)$.

4.6 Sampling parse trees from $\mathbb{T}_2$ with and without replacement

Once we have a $T : \mathbb{T}_2$, one option would be to sample as a top-down generative model, using some form of PCFG sampling. Constructing a sampler for $\mathbb{T}_2$ is straightforward. 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,

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

from analytic combinatorics, however unlike Duchon et al. [20], our work does not depend on rejection to guarantee exact-size sampling, as all trees from $\mathbb{T}_2 \cong \mathcal{G}'_\cap$ are destined to all be within a small Levenshtein distance of each other.

Algorithm 1 Ordinary PCFG sampling

Require: $T : \mathbb{T}_2$ intersection grammar, $\Gamma : \mathbb{T}_2 \rightsquigarrow \mathbb{T}$ sampler, $P_\theta : \mathbb{T} \to \mathbb{R}$ PCFG

```
Â ← ∅                                  ▷ Initialize set of parse trees.
repeat
    Â ← Â ∪ {Γ(T)}                     ▷ Sample a parse tree and add to retrieved set.
until interrupted.
return Â ranked by P_θ(·)              ▷ Rank by PCFG likelihood.
```

This sampler is simple and fast, but does not guarantee uniformity over the set of all generable parse trees, and does not converge linearly, failing to satisfy Def. 3.3. To get around this issue, we will define an alternate sampler based on an integer bijection, sample integers without replacement, decode them into trees, and finally rerank by a Markov chain.

The number of labeled binary trees inhabiting a single instance of $\mathbb{T}_2$ is sensitive 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 $T : \mathbb{T}_2$ uniformly without replacement, we precompute a histogram for each production counting the size of its language relative to the size of the root nonterminal's language, assign a commensurate integer range, and then construct a modular pairing function $\varphi : \mathbb{T}_2 \to \mathbb{Z}_{|T|} \to \texttt{BTree}$ that recursively selects values within that range:

$$\varphi(T : \mathbb{T}_2, i : \mathbb{Z}_{|T|}) \mapsto \begin{cases} \texttt{BTree}\big(\texttt{root}(T)\big) & \text{if } T \text{ is a leaf,} \\ \textbf{let } r = |\texttt{children}(T)|, & \\ \quad F(n) = \sum_{\langle l, r \rangle \in \texttt{children}[0\ldots n]} |l| \cdot |r|, & \\ \quad F^{-1}(u) = \inf\big\{x \mid u \le F(x)\big\}, & \\ \quad q = i - F(F^{-1}(i)), & \\ \quad l, r = \texttt{children}[t], & \\ \quad q_1, q_2 = \Big\langle \lfloor \frac{q}{|r|} \rfloor, q \pmod{|r|} \Big\rangle, & \\ \quad T_1, T_2 = \Big\langle \varphi(l, q_1), \varphi(r, q_2) \Big\rangle \textbf{ in} & \\ \texttt{BTree}\big(\texttt{root}(T), T_1, T_2\big) & \text{otherwise.} \end{cases} \tag{14}$$

Then, instead of top-down incremental sampling, we can create a randomized φ' from φ by sampling integers uniformly without replacement from $\mathbb{Z}_{|T|}$, then decode them into whole parse trees. Assuming $\varphi(T, \cdot)$ is in fact bijective, then letting $\boldsymbol{\varphi}(i) = \bigcup_{j \in [1, i]} \{\varphi'(T, j)\}$ will satisfy Def. 3.3 by construction. This procedure is trivially parallelizable across an arbitrary number of processors, enabling communication-free sampling without replacement. If allotted sufficient resources and $|T|$ is small, it will retrieve every parse tree, otherwise sample them uniformly until interrupted.

Returning to the ranked repair problem (Def. 3.2), the above procedure returns a set of syntactically 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 or the probability under a PCFG. We implement the simplest solution: the likelihood of a low-order Markov chain. This solution is computationally fast, does not require a trained PCFG, and as we will show, 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:

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

Then, for each retrieved set $\hat{A} \subseteq A$ drawn by the sampler before a predetermined timeout elapses and each $\sigma \in \hat{A}$, we score the repair and return $\hat{A}$ in ascending order by score. If $\hat{A} = A$ and the Markov chain is itself the language model being maximized, then this procedure satisfies Def. 3.2. Otherwise it is a heuristic, and the quality of the ranking will depend on the quality of $\hat{A}$.

Algorithm 2 Enumerative tree sampling with n-gram reranking

Require: $T : \mathbb{T}_2$ intersection grammar, $P_\theta : \Sigma^d \to \mathbb{R}$ Markov chain

$\hat{A} \leftarrow \varnothing$, `seed` $\leftarrow 0$ ▷ Initialize set of parse trees.
for `seed++` $< |T|$ and uninterrupted **do**
    $t \leftarrow \varphi(T, \texttt{RandomIntWoR}(\texttt{seed}, |T|))$ ▷ Draw unique $\mathbb{Z}_{|T|}$ and decode into fresh parse tree.
    $\hat{A} \leftarrow \hat{A} \cup \{t\}$
return $\hat{A}$ **ranked by** $NLL(\text{🍂}(t))$ ▷ Defoliate and rank by n-gram likelihood.

4.7 Decoding repairs in order of maximal likelihood using an FSA

The previous technique will enumerate parse trees in a given $\mathbb{T}_2$, but does not guarantee string uniqueness, as the same string may have more than one parse, i.e., the CFG may be ambiguous. While potentially insignificant, this becomes problematic for large finite CFLs and language intersections involving highly ambiguous CFGs. First, we make the following observation:

Lemma 4.7. *If the FSA, α, is ambiguous, then the intersection grammar, $G_\cap$, can be ambiguous.*

Proof. Let ℓ be the language defined by $G = \{S \to LR, L \to \texttt{(}, R \to \texttt{)}\}$, where $\alpha = L(\underset{\sim}{\sigma}, 2)$, the broken string $\underset{\sim}{\sigma}$ is `)(`, and $\mathcal{L}(G_\cap) = \ell \cap \mathcal{L}(\alpha)$. Then, $\mathcal{L}(G_\cap)$ contains the following two identical repairs: `)()` with the parse $S \to q_{00}Lq_{21}\ q_{21}Rq_{22}$, and `()` with the parse $S \to q_{00}Lq_{11}\ q_{11}Rq_{22}$. ☐

In practice, this means the tree sampler can produce multiple parse trees which represent the same string, impeding convergence. We can eliminate ambiguity and thereby improve the rate of convergence for natural syntax repair by translating $\mathbb{T}_2$ into a DFA, then sampling repair trajectories in order of decreasing string likelihood. First, let us note the following:

Lemma 4.8. *The intersection grammar, $G_\cap$, is acyclic.*

Proof. Assume $G_\cap$ is cyclic. Then $\mathcal{L}(G_\cap)$ must be infinite. But since $G_\cap$ generates $\ell \cap \mathcal{L}(\alpha)$ by construction and α is acyclic, $\mathcal{L}(G_\cap)$ is necessarily finite. Therefor, $G_\cap$ must not be cyclic. ☐

Since $G_\cap$ is acyclic, it can be directly translated into a $\mathbb{T}_2$ and thus an FSA. Using an FSA for decoding has many advantages, notably, it does not require gathering a dataset of parse trees to calibrate the sampler and unlike CFGs, can be efficiently minimized and converges linearly regardless of syntactic ambiguity. It is also more easily steerable than a PCFG sampler, and can be decoded in order of n-gram likelihood using a standard pretrained autoregressive language model.

Constructively, let $+, * : \mathcal{A} \times \mathcal{A} \to \mathcal{A}$ be the automata operators corresponding to language union and concatenation satisfying $\mathcal{L}(A_1 + A_2) = \mathcal{L}(A_1) \cup \mathcal{L}(A_2)$, and $\mathcal{L}(A_1 * A_2) = \mathcal{L}(A_1) \times \mathcal{L}(A_2)$. This can be implemented using the standard textbook construction, recalling that FSAs are closed under these operations. We can translate the $\mathbb{T}_2$ ADT to a finite state automaton $\mathcal{A}$ as follows:

$$\mathcal{Y}(T : \mathbb{T}_2) \mapsto \begin{cases} \alpha \mid \mathcal{L}(\alpha) = \{T\} & T : \Sigma, \\ \sum_{\langle T_1, T_2 \rangle \in \texttt{children}(T)} \mathcal{Y}(T_1) * \mathcal{Y}(T_2) & T : VL\big(V^2 P(a)^2\big) \end{cases}$$

In the case of LBH intersection grammars, $\mathcal{Y}(G'_\cap)$ would then yield an FSA recognizing $\ell \cap L(\sigma, d)$, which can be determinized, minimized using Brzozowski's algorithm [12] and decoded using a k-best paths algorithm to obtain the top-k maximum likelihood repairs.

For example, let us return to the second example given in § 2, where we have the syntactically invalid Python string, `v = df.iloc(5:, 2:)`. The CFG recognizing the language intersection can be translated into an equivalent DFA. After minimization, this will take the following form:

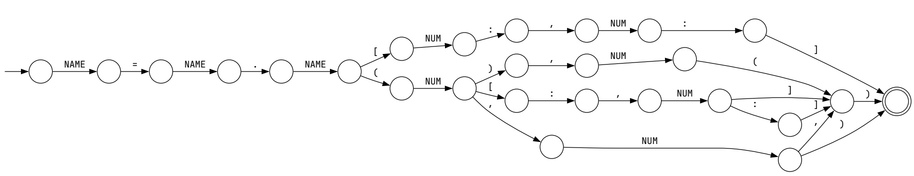


Fig. 6. Minimal DFA recognizing the langauge $L(\texttt{NAME = NAME . NAME (NUM : , NUM :)}, 2) \cap \ell_{\text{PYTHON}}$.

At the first bifurcation, we have a choice. Assuming a 4-gram Markov chain, we estimate the probability of $P(\sigma_i \mid \sigma_{i-1..3} = \texttt{NAME . NAME})$ for $\sigma_i = \texttt{[}$ versus $\sigma_i = \texttt{(}$ by comparing the respective transition probabilities. At each subsequent junction, we clone the trajectory, estimate the likelihood of each branch, then greedily expand the most likely path according to path length-normalized log likelihood. A beam search decoder is also possible, which we leave as an exercise for the reader.

The steerable DFA walk takes a DFA and a Markov chain, then samples trajectories through the DFA from the initial state to a final state, in order of the most likely partial trajectories. It maintains a priority queue of of partial trajectories, ranked by length-normalized log likelihood and which are speculatively extended by the most likely available transition.

Algorithm 3 Steerable DFA walk

Require: $\mathcal{A} = \langle Q, \Sigma, \delta, I, F\rangle$ DFA, $P_\theta : \Sigma^d \to \mathbb{R}$ Markov chain

$\mathcal{T} \leftarrow \varnothing, \mathcal{P} \leftarrow \big[\langle \varepsilon, i, 0\rangle \mid i \in I\big]$ ▷ Initialize priority queue of total and partial trajectories.

repeat

  let $\langle \sigma, q, \gamma\rangle = \mathtt{head}(\mathcal{P})$ **in**

   $\mathrm{T} = \big\{\langle s\sigma, q', \gamma - \log P_\theta(s \mid \sigma_{1..d-1})\rangle \mid (q \xrightarrow{s} q') \in \delta\big\}$ ▷ Extend partial trajectories.

  for $\langle \sigma, q, \gamma\rangle = T \in \mathrm{T}$ **do**

   if $\exists s : \Sigma, q' : Q \mid (q \xrightarrow{s} q') \in \delta$ **then**

    $\mathcal{P} \leftarrow \mathtt{tail}(\mathcal{P}) \oplus T$ ▷ Add partial trajectory to priority queue.

   if $q \in F$ **then**

    $\mathcal{T} \leftarrow \mathcal{T} \oplus T$ ▷ Accepting state reached, add trajectory into total queue.

until interrupted or $\mathcal{P} = \varnothing$.

return $\big[\sigma_{|\sigma|..1} \mid \langle \sigma, q, \gamma\rangle = T \in \mathcal{T}\big]$ ▷ Reverse string and return in order of likelihood.

Regardless of grammatical ambiguity, this procedure satisfies Def. 3.3 and Def. 3.2 simultaneously, as each repair will be unique and in order of decreasing likelihood.

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. Accuracy is evaluated across a test set by checking for lexical equivalence with the ground-truth repair, following Sakkas et al. (2022) [41].

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 $\mathcal{D}_{\text{test}}$, we define Precision@k as:

$$\text{Precision@k}(R) = \frac{1}{|\mathcal{D}_{\text{test}}|} \sum_{\langle \sigma^\dagger, \sigma'\rangle \in \mathcal{D}_{\text{test}}} \mathbb{1}\Big[\sigma' \in \operatorname*{argmax}_{\boldsymbol{\sigma} \subseteq R(\sigma^\dagger), |\boldsymbol{\sigma}| \leq k} \sum_{\sigma \in \boldsymbol{\sigma}} \text{NLL}(\sigma)\Big] \tag{17}$$

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. Precision@All or completeness may be seen as a special case, where $k = \infty$.

We compare our method against two separate baselines, Seq2Parse and Break-It-Fix-It (BIFI) [47] on a single test set. This dataset [46] 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 model

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

reported by the authors, with a variable-width beam search to control the number of predictions, and let the fixer 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 150 GB of RAM, running 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 samples at varying latency cutoffs. We apply a 30-second latency cutoff for our sampler.

6 EVALUATION

We call our method Tidyparse and consider the following research questions:

- **RQ 1**: What statistical properties do natural repairs exhibit? (e.g., length, edit distance)
- **RQ 2**: How performant is Tidyparse at fixing syntax errors? (i.e., vs. Seq2Parse and BIFI)
- **RQ 3**: Which design choices are most significant? (e.g., search vs. sampling, parallelism)

We address **RQ 1** in § 6.1 by analyzing the distribution of natural Python repair lengths and distances, **RQ 2** in § 6.2 by comparing Tidyparse against two existing syntax repair baselines, and **RQ 3** in § 6.3 by ablating various design choices and evaluating the impact on repair precision.

6.1 Dataset

In the following experiments, we use a dataset of Python snippets consisting of 20,500 pairwise-aligned human errors and fixes from StackOverflow [46]. We preprocess the dataset to lexicalize all code snippets, then filter by length and distance shorter than 80 lexical tokens and under five edits, i.e., where pairwise Levenshtein distance is under five lexical edits ($|\Sigma| = 50, |\sigma| < 80, \Delta(\sigma, \sigma') < 5$). We depict the length, edit distance, normalized edit locations and stability profile in Fig. 7.

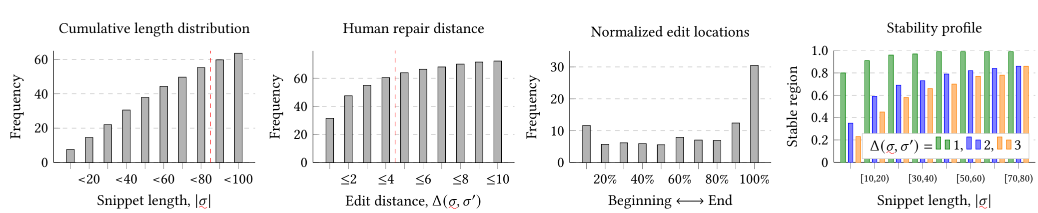


Fig. 7. Repair statistics across the StackOverflow dataset, of which Tidyparse can handle about half in under ~30s and ~150 GB. Larger repairs and edit distances are possible, albeit requiring additional time and memory.

For the stability profile, we enumerate repairs for each syntax error and estimate the average fraction of all edit locations that were never altered by any repair in the $L(\sigma, \Delta(\sigma, \sigma'))$-ball. For example, on average roughly half of the string is stable for 3-edit syntax repairs in the $[10 - 20)$ token range, whereas 1-edit repairs of the same length could modify only $\sim 10\%$ of all locations. For a fixed edit distance, we observe an overall decrease in the number of degrees of caret freedom with increasing length, which intuitively makes sense, as the repairs are more heavily constrained by the surrounding context and their locations grow more concentrated relative to the entire string.

6.2 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 20k distinct samples.

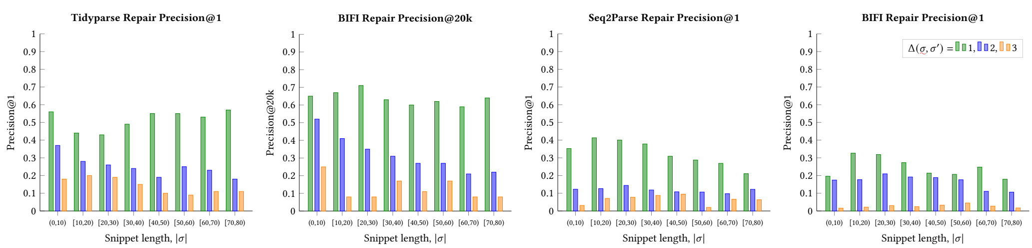


Fig. 8. Tidyparse, Seq2Parse and BIFI repair precision at various lengths and Levenshtein distances.

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. We report the raw data from these experiments in Appendix B.

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.

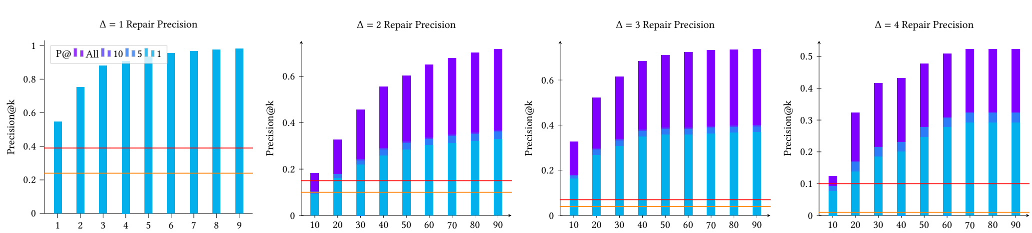


Fig. 9. Human repair benchmarks. Note the y-axis across different edit distance plots has varying ranges. The red line indicates Seq2Parse and the orange line indicates BIFI's Precision@1 on the same repairs.

We present a Sankey diagram of our repair pipeline in Fig. 10. We drew 967 total repairs from the StackOverflow dataset balanced evenly across lengths and edit distances ($\lfloor |\sigma|/10 \rfloor \in [0, 8], \Delta(\sigma, \sigma') < 4$) with a timeout of 30s and tracked individual outcomes. In 87 cases, the intersection grammar was too large to construct and threw an out-of-memory (OOM) error, in 4 cases the human repair was not recognized, in 153 cases the sampler timed out before drawing the human repair, in 238 cases the human repair was drawn but not ranked first, and in the remaining 485 cases the first prediction matched the human repair.

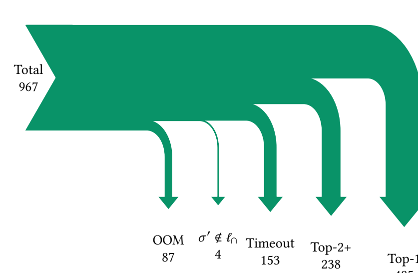


Fig. 10. Outcomes in the repair pipeline.

The remaining experiments in this section were run on a 10-core ARM64 M1 with 16 GB of memory. We balance the StackOverflow dataset across Levenshtein distances, then measure the number of samples required to draw the exact human repair across varying Levenshtein radii. This tells us of how many samples are required on average to saturate the admissible set.

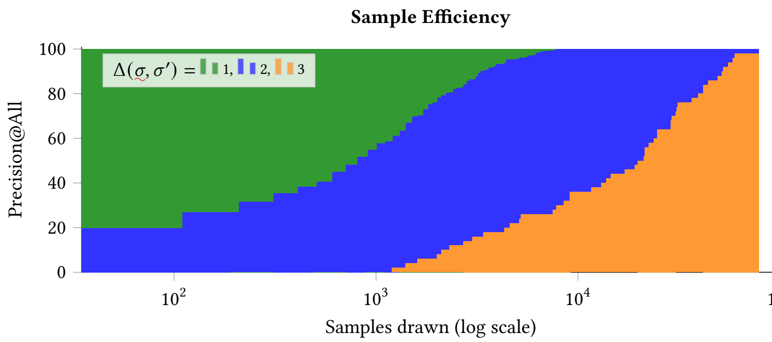


Fig. 11. Sample efficiency of Tidyparse at varying Levenshtein radii. After drawing up to $\sim 10^5$ samples without replacement we can usually retrieve the human repair for almost all repairs fewer than four edits.

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 the ground truth is discovered. We evaluate throughput by sampling patches across invalid strings $|\sigma| \leq 40$ from the StackOverflow dataset balanced across length and distance, and measure the total number of unique valid patches 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 syntactic repairs in a relatively short amount of time, and is able to quickly saturate the admissible set for $\Delta(\sigma, \sigma') \in [1, 4]$ before timeout.

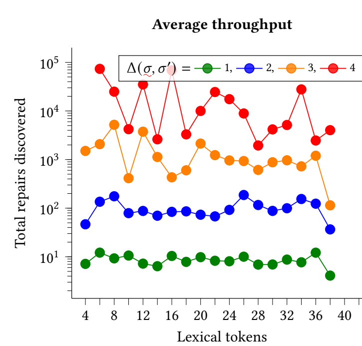


Fig. 12. Distinct repairs found in 30s.

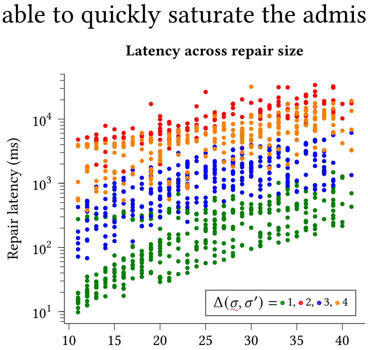


Fig. 13. End-to-end repair timings.

In Fig. 13, we plot the end-to-end repair timings by collecting 1000 samples balanced across length and edit distance, then measure the wallclock time until the sampler retrieves the human repair and report the log latency. While short repairs finish quickly, latency is positively correlated with length and edit distance. Our method is typically able to saturate the admissible set for 1- and 2-edit repairs before timeout, while 4+-edit throughput starts becoming constrained by compute around 30s, when Python's admissible set approaches a volume of 10^5 valid edits. This bottleneck can be relaxed with a longer timeout or additional CPU cores. We anticipate that a much longer delay will begin to tax the patience of most users, and so we consider 30s a reasonable upper bound for repair latency. As we will now show, end-to-end latency can be improved by doing rejection sampling, albeit at the cost of naturalness and sample efficiency.

6.3 Subcomponent ablation

Originally, we used an adaptive rejection-based sampler, which did not sample directly from the admissible set, but the entire Levenshtein ball, and then rejected invalid samples. Although rejection sampling has a much lower minimum latency threshold to return admissible repairs, i.e., a few seconds at most, the average time required to attain a desired precision on human repairs is much higher. We present the results from the rejection-based evaluation for comparison below.

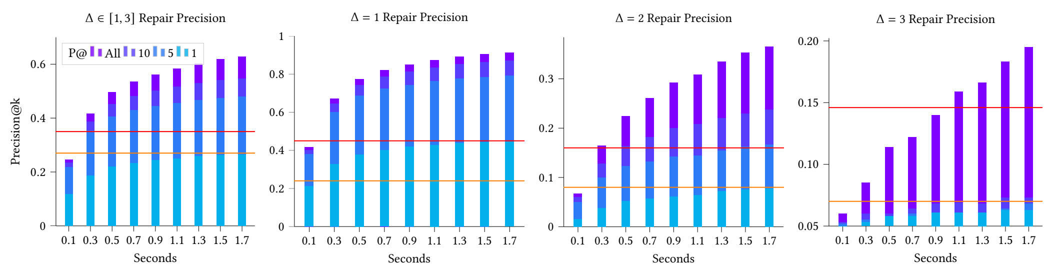
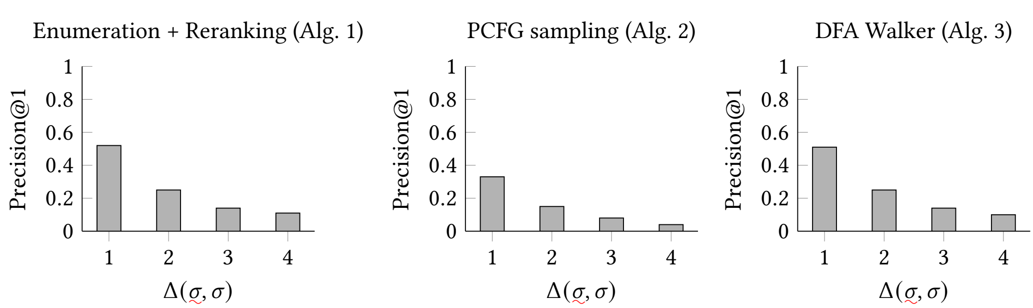


Fig. 14. Adaptive sampling repairs. The red line indicates Seq2Parse Precision@1, and the orange indicates BIFI's precision at single-shot repair, all three of which were evaluated on the exact same repairs.

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 and BIFI also produce syntactically incorrect repairs in practice. The overall latency of Seq2Parse 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, while BIFI consistently achieves subsecond latency across all repairs and distances.

Next, we conduct an ablation study to compare the effectiveness of PCFG sampling versus enumeration. In both experiments, we balance the StackOverflow dataset across edit distances and run the repair sampler for up to 30 seconds (in either enumerative or PCFG mode), then rerank all repairs by n-gram perplexity and measure the Precision@1 for sampling, enumeration, and the default hybrid approach, which uses enumeration if the intersection grammar size $|G'_\cap|$ contains fewer than 10,000 total productions, and PCFG sampling otherwise.



While the overall precision is notably lower for PCFG sampling than enumeration, the average number of samples drawn is also significantly lower, indicating a higher sample efficiency. This illustrates the tradeoff between sample efficiency and diversity, and suggests a hybrid approach is most effective at retrieval across density regimes. When the CFL is very large, uniform sampling is unlikely to retrieve the most natural samples, and top-down PCFG sampling offers a more informed prior, albeit at the cost of lower coverage on small languages.

In general, enumeration has an advantage when the CFL is very small. For example, if the CFL contains 2,000 sentences, enumeration is likely to recover all 2,000 before timeout, whereas PCFG sampling may only recover 100 of the most probable samples depending on the transition probabilities. However, if the CFL has 200,000 sentences, enumeration may only recover 10,000 uniform random samples in the time allotted and the PCFG sampler only 5,000, but due to its higher sample efficiency, is likely to contain a much higher proportion of natural repairs.

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.15 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.

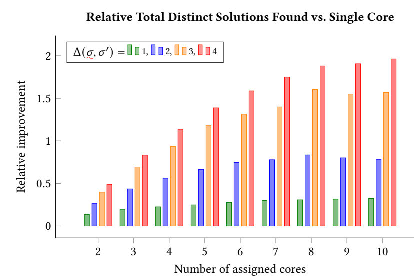


Fig. 15. Observed improvement in throughput relative to total CPU cores assigned.

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. This is especially true for small-radii Levenshtein balls, where the admissible set is small enough to be enumerated completely and ranked. For larger radii, we can still achieve competitive precision by using a PCFG to sample from the admissible set and reranking by perplexity.

Unexpectedly, we find that Precision@1 of our method is competitive with BIFI's Precision@20k, while requiring only a fraction of the data and compute. This is likely because BIFI's training set does not cover the full space of syntactically valid repairs. As Tidyparse uses its own grammar, it can sample from the language directly, and does not require training distribution to suggest valid repairs, only to rank them by naturalness. The emphasis on completeness is especially useful for discovering small repairs, which may be overlooked by neural models.

Although latency and precision are ultimately the deciding usability factors, repair throughput is a crucial intermediate factor to consider when evaluating the performance of a repair system. Even with a perfectly accurate scoring function, if the correct repair is never retrieved, it will be for naught. By maximizing the total number of unique valid repairs, we increase the likelihood of retrieving natural repairs to give the scoring function the best chance of ranking them successfully. For this reason, we prioritize throughput heavily in our design (Def. 3.3) and evaluation (Fig. 12).

Rejection sampling can be a useful technique for quickly retrieving a subset of valid repairs, but has the disadvantage of converging very slowly, requiring far too many samples to achieve competitive precision on natural repairs. One avenue may be to use rejection sampling to find probable edit locations, then switch to an exhaustive method to retrieve all repairs in that region. This approach however, would not offer the same completeness guarantees as language intersection.

7.1 Limitations and future work

7.1.1 Naturalness. Firstly, Tidyparse does not currently support intersections between weighted CFGs and weighted finite automata, a la Pasti et al. [39]. This feature would allow us to put transition probabilities on the Levenshtein automaton corresponding to more likely edits and construct a weighted intersection grammar. It would be interesting to train a weighted intersection grammar and compare the precision versus vanilla PCFG sampling. We also hope to explore more incremental sampling strategies such as sequential Monte-Carlo [34].

The scoring function is currently computed over lexical tokens. We expect that a more precise scoring function could be constructed by splicing candidate repairs back into the original source code and then scoring plaintext, however this would require special handling for insertions and substitutions of names, numbers and identifiers that were absent from the original source code. For this reason, we currently perform the scoring in lexical space, which discards a useful signal, but even this coarse approximation is sufficient to achieve state-of-the-art precision.

Furthermore, the scoring function only considers each candidate repair $P_\theta(\sigma')$ in isolation, returning the most plausible candidate independent of the original error. One way to improve this would be to incorporate the broken sequence ($\underset{\sim}{\sigma}$), parser error message (m), original source (s), and possibly other contextual priors to inform the scoring function. This would require a more expressive probabilistic language model to faithfully model the joint distribution $P_\theta(\sigma' \mid \underset{\sim}{\sigma}, m, s, \ldots)$, but would significantly improve the precision of the generated repairs.

7.1.2 Complexity. Latency can vary depending on several factors including string length, 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 expands rapidly and the number of valid repairs quickly becomes too large to be useful without a very precise naturalness metric to distinguish equidistant repairs.

Space complexity increases sharply with edit distance and to a lesser extent with length. This can be partly alleviated with more precise criteria to avoid creating superfluous productions, but the memory overhead is still considerable. Memory pressure can be attributed to engineering factors such as the grammar encoding, but is also an inherent challenge of grammar intersection. Managing the size of the intersection grammar during construction is a critical factor in scaling up this technique. Weighted intersections would also help to limit the size of the intersection grammar, as it would enable us to prune improbable productions without diminishing precision.

7.1.3 Toolchain integration. 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 partly be alleviated by filtering the results through an incremental compiler or linter, however, the latency necessary to check every repair may be non-negligible. It may also be possible to add a type-based semantic refinement to our language intersection, however this would require a more expressive grammatical formalism than CFGs can naturally provide.

Slicing is an important preprocessing consideration which has so far gone unmentioned. The current implementation expects pre-sliced code fragments, however in a more practical scenario, it would be necessary to leverage information from the programming environment to identify the boundaries of the code fragment to be repaired. This could be done with careful integration with the development environment or via ad hoc slicing. We leave this aspect for future work.

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? Those questions are addressed by three theoretical areas, (1) parsing, (2) language equations and (3) syntax repair. We survey each of those areas, then turn our attention to more engineering-oriented research, including (4) string solving, (5) error-correction, (6) decoding and finally (7) neural program repair.

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) [40], Earley [21] and others. These parsers have roughly cubic complexity with respect to the length of the input string.

As shown by Valiant [45], Lee [32] 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, however large constants tend to limit their practical utility.

8.2 Language equations

Language equations are a powerful tool for reasoning about formal languages and their inhabitants. First proposed by Ginsburg et al. [25] 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 [12, 13] 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, 36] and automata minimization [11]. In our setting, differentiation corresponds to code completion.

Bar-Hillel [5] establishes the closure of CFLs under intersection with regular languages, but does not elaborate on how to construct the corresponding grammar in order to recognize it. Beigel [8] and Pasti et al. [39] provide helpful insights into the construction of the intersection grammar, and Nederhof and Satta [37] specifically consider finite CFL intersections, but neither considers Levenshtein intersections. Our work specializes Bar-Hillel intersections to Levenshtein automata.

More concretely, we restrict our attention to language equations over CFLs, whose variables coincide with edit locations in the source code of a computer program, and solutions correspond to syntax repairs. While prior work has studied the use of language equations for parsing [36], to our knowledge they were never specifically applied to 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 [42] 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 [30] 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 String solving

There is related work on string constraints in the constraint programing literature, featuring solvers like CFGAnalyzer and HAMPI [31], which consider bounded context free grammars and intersections thereof. Bojańczyk et al. (2014) [10] introduce the theory of nominal automata. Around the same time, D'Antoni et al. (2014) introduce *symbolic automata* [17], a generalization of finite automata which allow infinite alphabets and symbolic expressions over them. Hague et al. (2024) [27] 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 in § 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 [18, 44] or Levenshtein [6, 7, 33] balls, from which we draw inspiration for this work.

8.6 Decoding

Decoding is a key problem in machine translation, speech recognition, and other sequence-to-sequence tasks. Given a compressed encoding of some finite distribution, its goal is find the maximum likelihood samples. A classic example is Viterbi decoding, which is used to find the most likely sequence of hidden states in a hidden Markov model and is closely related to the CYK algorithm for parsing. For PCFGs, the problem is more challenging, as the total support can be exponentially larger relative to the number of transitions.

In particular, we care about the problem of *top-k decoding*, which attempts to find the exact or approximate k-most likely samples in order of decreasing likelihood. This is closely related to the k-best enumeration [22] problem, a carefully studied problem in graph theory and combinatorial optimization. An exact solution to this problem for large acyclic PCFGs is often intractable, but we can approximate it using a beam search or cube-pruning technique.

A popular solution to k-best decoding in the NLP literature is a technique called cube-pruning [14, 28, 29], which samples maximum likelihood paths through a hypergraph. We take inspiration from this technique, and adapt it with a probabilistic higher-order grammar (PHOG) [9]. Our approach is similar to work by Zhang and McDonald [48], but specialized to language intersections.

An alternate approach would be to use MCMC or sequential Monte Carlo method to steer a transformer-based large langauge model (LLM), as proposed by Lew et al. [34]. This technique is particularly useful for constrained sampling from LLMs, and could be adapted to our setting to improve sample efficiency. The downside is that it introduces a dependency on an LLM, which requires a very large dataset to train and is more computationally expensive than cube-pruning. Furthermore, distinct sampling is unclear how to do properly, as LLMs are not trained to generate unique samples, and sampling without replacement is a fundamentally non-Markovian process. One potential solution proposed by Shi and Bieber [43] assumes trace injectivity and constructs a trie, however their solution is not stateless and can require a significant latency overhead.

Our approach is complementary to existing work in constrained decoding. The bijection proposed in Eq. 14 guarantees that all repairs are well-formed and converge linearly to the exact top-k maximum likelihood samples. This method is completely stateless and can be used to enumerate a bounded Levenshtein ball with linear parallelization speedup. Alternately, in the case of approximate ranked repair over a very large sample space, this technique can be adapted to sample with high probability a representative subset of the most likely sentences in a finite but large PCFG.

8.7 Neural program repair

The recent success of deep learning has lead to a variety of work on neural program repair [3, 16, 19]. These approaches typically employ Transformer-based large language models (LLMs) and model the problem as a sequence-to-sequence transformation. In particular, two papers stand out being closely related to our own: Break-It-Fix-It (BIFI) [47] and Seq2Parse [41]. 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 weave formal language theory and machine learning into helpful programming tools for everyday developers. In some ways, syntax repair serves as a test bench for integrating learning and language theory, as it lacks the intricacies of type-checking and semantic analysis, but is still rich enough to be an interesting challenge. By starting with syntax repair, we hope to lay the foundation for more organic hybrid approaches to program synthesis.

Two high level codesign patterns have emerged to combine the naturalness of neural language models with the precision of formal methods. One seeks to filter the outputs of a generative language model to satisfy a formal specification, typically by some form of rejection sampling. Alternatively, some attempt to use language models to steer an incremental search for valid programs via a reinforcement learning or hybrid neurosymbolic strategy. However, implementing these strategies is often painstaking and their generalization behavior can be difficult to analyze.

In our work, we take a more pragmatic tack - by incorporating the distance metric into a formal language, we attempt to exhaustively enumerate repairs by increasing distance, then use the stochastic language model to sort the resulting solutions by naturalness. The more constraints we can incorporate into 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 and ranking.

There is a delicate balance in formal methods between soundness and completeness. Often these two seem at odds because the target language is too expressive to achieve them both simultaneously. In syntax repair, we also care about *naturalness*. Fortunately, syntax repair is tractable enough to achieve all three by modeling the problem using language intersection. Completeness helps us to avoid missing simple repairs that might be easily overlooked, soundness guarantees all repairs will be valid, and naturalness ensures the most likely repairs receive the highest priority.

From a usability standpoint, 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.

DATA-AVAILABILITY STATEMENT

An artifact for Tidyparse is currently available as a browser application. [5] While the browser demo is single-threaded and does not support ranking synthetic repairs by naturalness, it is capable of automatically repairing syntax errors in arbitrary context-free languages. The data and source code for the experiments contained in this paper will be made available upon publication.

[5] https://tidyparse.github.io

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. https://arxiv.org/pdf/2105.12787.pdf

[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] Pavol Bielik, Veselin Raychev, and Martin Vechev. 2016. PHOG: probabilistic model for code. In International conference on machine learning. PMLR, 2933–2942.

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

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

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

[13] 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.

[14] David Chiang. 2007. Hierarchical phrase-based translation. computational linguistics 33, 2 (2007), 201–228.

[15] David Chiang, Peter Cholak, and Anand Pillay. 2023. Tighter bounds on the expressivity of transformer encoders. In International Conference on Machine Learning. PMLR, 5544–5562. https://proceedings.mlr.press/v202/chiang23a/chiang23a.pdf

[16] 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.

[17] 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.

[18] 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

[19] 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.

[20] 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.

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

[22] David Eppstein. 2014. k-best enumeration. arXiv preprint arXiv:1412.5075 (2014).

[23] 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.

[24] Philippe Flajolet, Daniele Gardy, and Loÿs Thimonier. 1992. Birthday paradox, coupon collectors, caching algorithms and self-organizing search. Discrete Applied Mathematics 39, 3 (1992), 207–229.

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

[26] Joshua Goodman. 1999. Semiring parsing. Computational Linguistics 25, 4 (1999), 573–606. https://aclanthology.org/J99-4004.pdf

[27] 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.

[28] Liang Huang and David Chiang. 2005. Better k-best parsing. In Proceedings of the Ninth International Workshop on Parsing Technology. 53–64.

[29] Liang Huang and David Chiang. 2007. Forest rescoring: Faster decoding with integrated language models. In Proceedings of the 45th annual meeting of the association of computational linguistics. 144–151.
[30] E. T. Irons. 1963. An Error-Correcting Parse Algorithm. Commun. ACM 6, 11 (nov 1963), 669–673. https://doi.org/10.1145/368310.368385
[31] 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.
[32] 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
[33] 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
[34] 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).
[35] 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.
[36] Matthew Might, David Darais, and Daniel Spiewak. 2011. Parsing with derivatives: a functional pearl. ACM sigplan notices 46, 9 (2011), 189–195.
[37] Mark-Jan Nederhof and Giorgio Satta. 2004. The language intersection problem for non-recursive context-free grammars. Information and Computation 192, 2 (2004), 172–184.
[38] Rohit J. Parikh. 1966. On Context-Free Languages. J. ACM 13, 4 (oct 1966), 570–581. https://doi.org/10.1145/321356.321364
[39] Clemente Pasti, Andreas Opedal, Tiago Pimentel, Tim Vieira, Jason Eisner, and Ryan Cotterell. 2023. On the Intersection of Context-Free and Regular Languages. In Proceedings of the 17th Conference of the European Chapter of the Association for Computational Linguistics, Andreas Vlachos and Isabelle Augenstein (Eds.). Association for Computational Linguistics, Dubrovnik, Croatia, 737–749. https://doi.org/10.18653/v1/2023.eacl-main.52
[40] Itiroo Sakai. 1961. Syntax in universal translation. In Proceedings of the International Conference on Machine Translation and Applied Language Analysis.
[41] 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.
[42] Klaus U Schulz and Stoyan Mihov. 2002. Fast string correction with Levenshtein automata. International Journal on Document Analysis and Recognition 5 (2002), 67–85.
[43] Kensen Shi, David Bieber, and Charles Sutton. 2020. Incremental sampling without replacement for sequence models. In International Conference on Machine Learning. PMLR, 8785–8795.
[44] Michalis K Titsias and Christopher Yau. 2017. The Hamming ball sampler. J. Amer. Statist. Assoc. 112, 520 (2017), 1598–1611.
[45] 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
[46] 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.
[47] Michihiro Yasunaga and Percy Liang. 2021. Break-it-fix-it: Unsupervised learning for program repair. In International Conference on Machine Learning. PMLR, 11941–11952.
[48] Hao Zhang and Ryan McDonald. 2012. Generalized higher-order dependency parsing with cube pruning. In Proceedings of the 2012 joint conference on empirical methods in natural language processing and computational natural language learning. 320–331.

A EXAMPLE REPAIRS

Below, we provide a few representative examples of broken code snippets and the corresponding human repairs that were successfully 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)`
`df.apply(lambda row: list(set(row['ids'))))`	`df.apply(lambda row: list(set(row['ids'])))`
`sum(len(v) for v items.values()))`	`sum(len(v) for v in items.values())`

`def average(values):`
`  if values == (1,2,3):`
`    return (1+2+3)/3`
`  else if values == (-3,2,8,-1):`
`    return (-3+2+8-1)/4`
	`def average(values):`
	`  if values == (1,2,3):`
	`    return (1+2+3)/3`
	`  elif values == (-3,2,8,-1):`
	`    return (-3+2+8-1)/4`

`dict = {`
`   "Jan": 1`
`   "January": 1`
`   "Feb": 2 # and so on`
`}`
	`dict = {`
	`   "Jan": 1,`
	`   "January": 1,`
	`   "Feb": 2 # and so on`
	`}`

`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. $|\sigma|$ 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							
$\vert\sigma\vert$		(0, 10)	[10, 20)	[20, 30)	[30, 40)	[40, 50)	[50, 60)	[60, 70)	[70, 80)
Tidyparse	1	0.56	0.44	0.43	0.49	0.55	0.55	0.53	0.57
	2	0.37	0.28	0.26	0.24	0.19	0.25	0.23	0.18
	3	0.18	0.20	0.19	0.15	0.10	0.09	0.11	0.11
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	1.00	0.99	0.99	1.00	0.97	0.97
	2	1.00	0.99	0.98	1.00	1.00	1.00	0.94	0.90
	3	1.00	0.98	0.80	0.70	0.55	0.42	0.42	0.31
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

C SUPPLEMENTAL PROOFS

The problem of syntax error correction under a finite number of typographic errors is reducible to the bounded Levenshtein-CFL reachability problem, which can be formally stated as follows:

Definition C.1. The language edit distance (LED) is the minimum number of edits required to transform an invalid string into a valid one, where validity is defined as containment in a context-free language, ℓ, i.e., $\Delta^*(\utilde{\sigma}, \ell) \coloneqq \min_{\sigma \in \ell} \Delta(\utilde{\sigma}, \sigma)$, and Δ is the Levenshtein distance.

We seek to find the set of strings S such that $\forall \sigma' \in S, \Delta(\utilde{\sigma}, \sigma') \leq q$, where q is greater than or equal to the language edit distance. We call this set the *Levenshtein ball* of $\utilde{\sigma}$ and denote it $\Delta_q(\utilde{\sigma})$. Since $1 \leq \Delta^*(\utilde{\sigma}, \ell) \leq q$, we have $1 \leq q$. We now consider an upper bound on $\Delta^*(\utilde{\sigma}, \ell)$, i.e., the greatest lower bound on q such that $\Delta_q(\utilde{\sigma}) \cap \ell \neq \varnothing$.

Lemma C.2. *For any nonempty language ℓ and invalid string $\utilde{\sigma} : \Sigma^n \cap \bar{\ell}$, there exists an (σ', m) such that $\sigma' \in \ell \cap \Sigma^m$ and $0 < \Delta(\utilde{\sigma}, \ell) \leq \max(m, n) < \infty$.*

Proof. Since ℓ is nonempty, it must have at least one inhabitant $\sigma \in \ell$. Let σ' be the smallest such member. Since σ' is a valid sentence in ℓ, by definition it must be that $|\sigma'| < \infty$. Let $m \coloneqq |\sigma'|$. Since we know $\utilde{\sigma} \notin \ell$, it follows that $0 < \Delta(\utilde{\sigma}, \ell)$. Let us consider two cases, either $\sigma' = \varepsilon$, or $0 < |\sigma'|$:

- If $\sigma' = \varepsilon$, then $\Delta(\utilde{\sigma}, \sigma') = n$ by full erasure of $\utilde{\sigma}$, or
- If $0 < m$, then $\Delta(\utilde{\sigma}, \sigma') \leq \max(m, n)$ by overwriting.

In either case, it follows $\Delta(\utilde{\sigma}, \ell) \leq \max(m, n)$ and ℓ is always reachable via a finite nonempty set of Levenshtein edits, i.e., $0 < \Delta(\utilde{\sigma}, \ell) < \infty$. □

Let us now consider the maximum growth rate of the *admissible set*, $A \coloneqq \Delta_q(\utilde{\sigma}) \cap \ell$, as a function of q and n. Let $\bar{\ell} \coloneqq \{\utilde{\sigma}\}$. Since $\bar{\ell}$ is finite and thus regular, $\ell = \Sigma^* \setminus \{\utilde{\sigma}\}$ is regular by the closure of regular languages under complementation, and thus context-free a fortiori. Since ℓ accepts every string except $\utilde{\sigma}$, it represents the worst CFL in terms of asymptotic growth of A.

Lemma C.3. *The complexity of enumerating A is upper bounded by $\mathcal{O}\left(\sum_{c=1}^{q} \binom{cn+n+c}{c}(|\Sigma|+1)^c\right)$.*

Proof. We can overestimate the size of A by considering the number of unique ways to insert, delete, or substitute c terminals into a string $\utilde{\sigma}$ of length n. This can be overapproximated by interleaving ε^c around every token, i.e., $\utilde{\sigma}_\varepsilon \coloneqq (\varepsilon^c \utilde{\sigma}_i)_{i=1}^{n} \varepsilon^c$, where $|\utilde{\sigma}_\varepsilon| = cn + n + c$, and only considering substitution. We augment $\Sigma_\varepsilon \coloneqq \Sigma \cup \{\varepsilon\}$ so that deletions and insertions may be treated as special cases of substitution. Thus, we have $cn + n + c$ positions to substitute $(|\Sigma_\varepsilon|)$ tokens, i.e., $\binom{cn+n+c}{c}|\Sigma_\varepsilon|^c$ ways to edit $\utilde{\sigma}_\varepsilon$ for each $c \in [1, q]$. This upper bound is not tight, as overcounts many identical edits w.r.t. $\utilde{\sigma}$. Nonetheless, it is sufficient to show $|A| < \sum_{c=1}^{q} \binom{cn+n+c}{c}|\Sigma_\varepsilon|^c$. □

We note that the above bound applies to all strings and languages, and relates to the Hamming bound on $H_q(\utilde{\sigma}_\varepsilon)$, which only considers substitutions. In practice, much tighter bounds may be obtained by considering the structure of ℓ and $\utilde{\sigma}$. For example, based on an empirical evaluation from a dataset of human errors and repairs in Python code snippets ($|\Sigma| = 50, |\utilde{\sigma}| < 40, \Delta(\utilde{\sigma}, \ell) \in [1, 3]$), we estimate the *filtration rate*, i.e., the density of the admissible set relative to the Levenshtein ball, $D = |A|/|\Delta_q(\utilde{\sigma})|$ to have empirical mean $E_\sigma[D] \approx 2.6 \times 10^{-4}$, and variance $\mathrm{Var}_\sigma[D] \approx 3.8 \times 10^{-7}$.