Syntax Error Correction as Idempotent Matrix Completion

ANONYMOUS AUTHOR(S)

In this work, we illustrate how syntax error correction with bounded errors is closely related to context-free language reachability and demonstrate a novel reduction onto a multilinear system of equations over finite fields. In addition to its theoretical value, this connection has demonstrated empirical speedups on real-world syntax repair tasks. To accelerate code completion and repair, we design and implement a novel incremental parser-synthesizer that translates context-free language intersection into Boolean satisfiability, enabling us to suggest syntax repairs in-between keystrokes. Our approach attains state-of-the-art repair precision@{5, 10} on a human repair dataset at a fraction of the time and cost of existing neural network based techniques, and is highly flexible to additional constraints and readily extensible to new programming languages.

1 INTRODUCTION

When writing a program, nearly all of the intermediate editing states are invalid. Manual repair, though fairly routine, adds friction to the development process by sporadically diverting attention, a programmer's most precious and fickle resource. *Syntax correction* is the problem of automatically repairing a syntactically invalid program so that it is no longer invalid. This problem may sound trivial but can be quite challenging, as well-formed programs have many semantic constraints, but also because the problem itself is under-determined: although repairs by definition must be valid, even assuming a tiny number of modifications, the search space of possible repairs can be vast.

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

In this work, we offer a new technique for repairing real-world syntax errors from first-principles, using a purely algebraic approach. We show how syntax error correction can be expressed as the conjunction of two context-free languages, then reduced onto a multilinear system of equations over a finite field. Our primary technical contributions are twofold: (1) is a novel reduction from linear conjunctive language reachability with bounded string variables onto Boolean tensor completion, and (2) is a randomized algorithm for quickly discovering and ranking admissible repairs.

Grounded in formal language theory, our approach borrows insights from Boolean matrix parsing [38] and conjunctive language reachability [42] to unify *parsing*, *code completion*, *error correction* under a simple algebraic framework. We provide exact and approximate algorithms for repairing real-world syntax errors, and implement them in a real-time editor called Tidyparse. This tool features a state-of-the-art fast repair procedure that is provably sound and complete up to a Levenshtein bound. We show how it can be used to complete unfinished code, parse incomplete code, and repair unparseable syntax fragments in arbitrary context-free and conjunctive languages.

This paper presents two high-level approaches to syntax error correction: one that uses equational reasoning to find the satisfying assignments to a multilinear system of equations over finite fields (Theory 1, model-theoretic), and a second which samples random edits and accepts only those which parse (Theory 2, probabilistic correction). Finally, we show how these two approaches can be combined to attain human-level precision and state-of-the-art wall clock latency on a dataset of human syntax errors and repairs – all without using a neural language model.
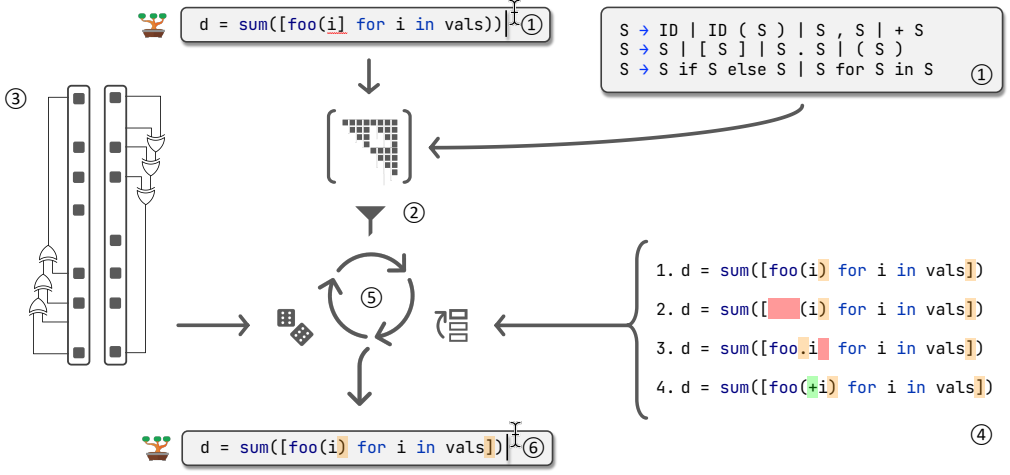

Fig. 1. Our framework consists of three components, namely (2) a solver, (4) ranker and (3) sampler. Given an invalid string and grammar (1), we first compile them into a Boolean tensor (2) representing a multilinear system of equations whose fixed points characterize the admissible set. The system can be solved directly, yielding a set of repairs that are ranked using a suitable scoring function (4). Optionally, we may introduce stochastic edits to the string using the Levenshtein ball sampler (3) and extract the solutions incrementally (5). When the langauge edit distance and solutions are time-sensitive, (5) is typically more efficient.

2 OVERVIEW

Tidyparse accepts an arbitrary CFG and a string, which we first attempt to parse using Valiant's parser (§4). If the string is invalid, we attempt to repair it by sampling a stochastic sequence of holes without replacement from the Levenshtein ball, then extract all valid strings. This sequence can be viewed as an elliptic curve over the set of all edits within a small neighborhood, filtered through a recognizer. Below is an example illustrating this procedure on a single Python snippet.

(7) (a) `d = sum([foo(+i) for i in vals])` $\Longrightarrow$ (b) `d = sum([foo(i) for i in vals])`
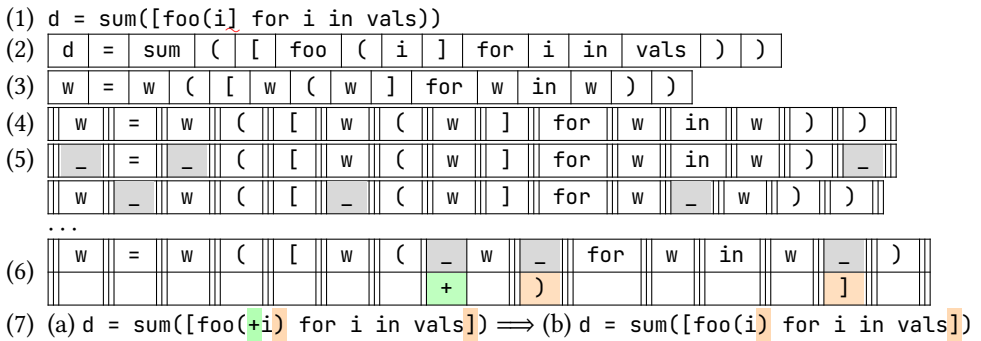
The initial broken string, `d = sum([foo(i] for i in vals))` (1), is first tokenized using a lexer to obtain the sequence in (2). Lexical tokens containing identifiers are abstracted in step (3), and interleaved with the empty token in step (4). We then sample hole configurations without replacement in step (5), many of which will have no admissible solutions. Eventually, the solver will discover an admissible solution, as seen in step (6). This solution is then used to generate a patch, which is applied to the original string in step (7a), then reduced to its minimal form in step (7b), and sampling is repeated until all possibilities are exhausted or a predetermined timeout expires.

Optionally, we may bias the sampler toward previously discovered repairs by adding successful patches to a replay buffer ranked by likelihood. This buffer can then be stochastically resampled using a Dirichlet process with stepwise decaying exploration to encourage diversity. As shown in §8, we found this approach to be significantly more sample-efficient with respect to language membership, yielding much higher precision and repair throughput under low-latency constraints.

Using the Levenshtein ball sampler as a proposal distribution, we then solve for the least fixpoint of an idempotent matrix to obtain the set of syntactically admissible repairs, all of which are guaranteed to be minimal and syntactically valid. These repairs are finally reranked according to their statistical likelihood, and the most likely repairs discovered within a fixed time are presented to the user. We describe our solver in more detail in §4 and this whole process is illustrated in Fig. 1.

During the conception of this work, a number of design choices were made to increase its utility as a real-time programming assistant, which aims to provide precise and continuous feedback while the user is typing across a variety of programming languages. To support this use case, the following criteria were taken into account when designing and evaluating various components:

- First and foremost, the framework must be **sound** and **complete**. That is, the parser-generator must (1) accept arbitrary conjunctive grammars, and (2) generate a parser which accepts all and only syntactically valid strings and (3) when a string is invalid, our synthesizer must eventually generate every syntactically valid string within a fixed distance from it.
- Second, we require that the resulting repairs be **plausible** and **diverse**. In other words, the framework should generate repairs that are likely to be written by a human being, consistent with the surrounding context, and reasonably diverse in their structure.
- Third, the framework must be **efficient** and **responsive**. That is, it must be able to recognize well-formed strings in subcubic time, and generate admissible repairs in subpolynomial time. These conditions are necessary to provide real-time feedback whilst the user is typing.
- Fourth, the framework must also be **robust** and **scalable**. In practice, this means that the framework should be robust to multiple errors, handle grammars with a large number of productions and be able to scale linearly as the number of processor cores are increased.
- Finally, the framework must be **flexible** and **extensible**. Intended as a general-purpose tool for generating syntax repairs in a wide variety of programming languages, end-users should be able to extend the framework with their own custom grammars and side-constraints.

As we demonstrate both formally and experimentally, our framework is able to satisfy most of these criteria simultaneously, while attaining state-of-the-art performance in terms of precision, throughput and wall clock latency. However, there are still some open problems that remain to be solved to realize its full potential as an everyday programming tool. In particular, we will discuss some of those open problems (e.g., 4.9) and design trade-offs (§9) that were taken in its construction.

With the above criteria in mind, the following paper addresses four primary research questions:

(1) **What theoretical guarantees do we offer?** (i.e., complexity, soundness and completeness)
(2) **How do we reduce syntax repair onto finite fields?** (i.e., algorithms and data structures)
(3) **How does Tidyparse compare with existing approaches?** (i.e., neural program repair)
(4) **What are the limitations of this technique?** (i.e., precision, latency and error tolerance)

We will begin by precisely defining the Levenshtein-CFL reachability problem (§3), then construct an algebraic theory of syntax correction (§4), followed by a corollary description of the parsing algorithm (§5). After presenting some practical examples of the tool's features (§6) and taking a short detour into the related literature on syntax repair (§7), we will then empirically evaluate the end-to-end performance on a dataset of human syntax errors and repairs in (§8). Finally, we will discuss the results (§9) and conclude with a summary of our contributions and future work (§10).

3 PROBLEM STATEMENT

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 3.1. 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, $\ell : \mathcal{L}$, i.e., $\Delta^*(\sigma, \ell) := \min_{\sigma \in \ell} \Delta(\sigma, \sigma)$, and Δ is the Levenshtein distance.

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

LEMMA 3.2. *For any nonempty language $\ell : \mathcal{L}$ and invalid string $\sigma : \Sigma^n \cap \bar{\ell}$, there exists an $(\tilde{\sigma}, m)$ such that $\tilde{\sigma} \in \ell \cap \Sigma^m$ and $0 < \Delta(\sigma, \ell) \leq \max(m, n) < \infty$.*

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

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

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

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

LEMMA 3.3. *The complexity of searching A is upper bounded by $\mathcal{O}\left(\sum_{c=1}^{q} \binom{cn+n+1}{c}(|\Sigma|+1)^c\right)$.*

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 σ of length n. This can be overapproximated by interleaving ε^c around every token, i.e., $\sigma_\varepsilon := (\varepsilon^c \sigma_i)_{i=1}^n \varepsilon^c$, where $|\sigma_\varepsilon| = cn + n + 1$, and only considering substitution. We augment $\Sigma_\varepsilon := \Sigma \cup \{\varepsilon\}$ so that deletions and insertions may be treated as special cases of substitution. Thus, we have $cn + n + 1$ positions to substitute ($|\Sigma_\varepsilon|$) tokens, i.e., $\binom{cn+n+1}{c}|\Sigma_\varepsilon|^c$ ways to edit σ_ε for each $c \in [1, q]$. This upper bound is not tight, as overcounts many identical edits w.r.t. σ. Nonetheless, it is sufficient to show $|A| < \sum_{c=1}^{q} \binom{cn+n+1}{c}|\Sigma_\varepsilon|^c$. □

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

In practice, this problem is ill-posed even when $q = \Delta^*(\sigma, \ell) \approx 1$. For example, consider the language of ursine dietary preferences. Although $\sigma :=$ "Bears like to eat plastic" is not a valid sentence, e.g., $\tilde{\sigma} :=$ "Bears like to eat" is ($\Delta^* = 1$), however there are many others with roughly the same edit distance, e.g., "Bears like to eat {berries, honey, fish}", or "{Polar, Panda} bears like to eat

---

[1] This reflects our general approach, which builds a surjection from the interleaved Hamming ball onto the Levenshtein ball.

{seals, bamboo}". In general, there are usually many strings nearby σ, and we seek to find those among them which are both syntactically valid and semantically plausible as quickly as possible.

4 MATRIX THEORY

Recall that a CFG is a quadruple consisting of terminals (Σ), nonterminals (V), productions ($P\colon V \to (V \mid \Sigma)^*$), and a start symbol, ($S$). It is a well-known fact that every CFG is reducible to *Chomsky Normal Form*, $P'\colon V \to (V^2 \mid \Sigma)$, in which every production takes one of two forms, either $w \to xz$, or $w \to t$, where $w, x, z\colon V$ and $t\colon \Sigma$. For example:

$$\mathcal{G} \coloneqq \big\{\, S \to S\,S \mid (\,S\,) \mid (\,) \,\big\} \Longrightarrow \mathcal{G}' = \big\{\, S \to Q\,R \mid S\,S \mid L\,R, \quad R \to), \quad L \to (, \quad Q \to L\,S \,\big\}$$

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

$$X \otimes Z \coloneqq \big\{\, w \mid \langle x, z \rangle \in X \times Z, (w \to xz) \in P \,\big\} \tag{1}$$

If we define $\sigma_r^{\updownarrow} \coloneqq \{w \mid (w \to \sigma_r) \in P\}$, then construct a matrix with nonterminals on the superdiagonal representing each token, $M^0_{r+1=c}(\mathcal{G}', e) \coloneqq \sigma_r^{\updownarrow}$ and solve for the fixpoint $M^* = M + M^2$,

$$M^0 \coloneqq \begin{pmatrix} \varnothing & \sigma_1^{\to} & \varnothing & \cdots & \varnothing \\ & & & & \\ & & & & \varnothing \\ & & & & \sigma_n^{\uparrow} \\ \varnothing & \cdots & & & \varnothing \end{pmatrix} \Rightarrow \begin{pmatrix} \varnothing & \sigma_1^{\to} & \Lambda & \cdots & \varnothing \\ & & & & \\ & & & & \Lambda \\ & & & & \sigma_n^{\uparrow} \\ \varnothing & \cdots & & & \varnothing \end{pmatrix} \Rightarrow \ldots \Rightarrow M^* = \begin{pmatrix} \varnothing & \sigma_1^{\to} & \Lambda & \cdots & \Lambda_\sigma^* \\ & & & & \\ & & & & \Lambda \\ & & & & \sigma_n^{\uparrow} \\ \varnothing & \cdots & & & \varnothing \end{pmatrix}$$

we obtain the recognizer, $R(\mathcal{G}', \sigma) \coloneqq [S \in \Lambda_\sigma^*] \Leftrightarrow [\sigma \in \mathcal{L}(\mathcal{G})]$ [2].

Since $\bigoplus_{c=1}^n M_{r,c} \otimes M_{c,r}$ has cardinality bounded by $|V|$, it can be represented as $\mathbb{Z}_2^{|V|}$ using the characteristic function, $\mathbb{1}$. Note that any encoding which respects linearity $\varphi(\Lambda \circledast \Lambda') \equiv \varphi(\Lambda) \circledast \varphi(\Lambda')$ is suitable – this particular representation shares the same algebraic structure, but is more widely studied in error correction, and readily compiled into circuits and BLAS primitives. Furthermore, it enjoys the benefit of complexity-theoretic speedups to matrix multiplication.

Details of the bisimilarity between parsing and matrix multiplication can be found in Valiant [38], who first realized its time complexity was subcubic $\mathcal{O}(n^\omega)$ where ω is the asymptotic lower bound for Boolean matrix multiplication ($\omega < 2.77$), and Lee [26], who shows that speedups to CFL parsing were realizable by Boolean matrix multiplication algorithms. While more efficient specialized parsers are known to exist for restricted CFGs, this technique is typically linearithmic under sparsity and believed to be the most efficient general procedure for CFL parsing.

4.1 Parsing with holes

Valiant's algorithm can be abstracted by lifting it into the domain of linear equations over finite fields, where each bitvector in the upper triangular matrix instead contains polynomials whose solutions identify nonterminals participating in the parse forest of a *porous string* on the superdiagonal (our work). This interpretation of Valiant's algorithm as an equational theory offers a novel polynomial reduction from language equations onto XORSAT. More formally, we solve the following problem:

Definition 4.1 (Hole subsumption). Let $\underline{\Sigma} \coloneqq \Sigma \cup \{\_\}$, where $\_$ represents a hole. We denote $\sqsubseteq\colon \underline{\Sigma}^n \times \Sigma^n$ as the relation $\{\langle \sigma, \sigma' \rangle \mid \sigma_i \in \Sigma \implies \sigma_i = \sigma_i'\}$ and the set of all $\{\sigma' \mid \sigma' \sqsubseteq \sigma\}$ as $\mathrm{H}(\sigma)$.

---

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

Definition 4.2 (Parsing with holes). Let $\underline{\Sigma} := \Sigma \cup \{\_\}$, where holes denote free variables over Σ. Given a CFG, $\mathcal{G} : \mathbb{G}$, and a *porous string*, $\sigma : \underline{\Sigma}^{n-1}, n : \mathbb{N}^{\geq 3}$, let us define $A(\sigma, \mathcal{G}) := \mathrm{H}(\sigma) \cap \mathcal{L}(\mathcal{G})$. We can extract $A(\sigma, \mathcal{G})$ by encoding $\langle \sigma, \mathcal{G} \rangle$ as an idempotent matrix, $\mathcal{M}_\sigma := (\mathbb{Z}_2^{|V|} \to \mathbb{Z}_2^{|V|})^{n \times n}$, and solving for all fixedpoints $[\mathcal{M}_\sigma = \mathcal{M}_\sigma^2]$, whose models are then decoded into $\sigma' \in A(\sigma, \mathcal{G})$.

The precise encoding is immaterial, as long as $\langle \boxplus, \boxtimes \rangle$ are defined so the diagram below commutes,[3]
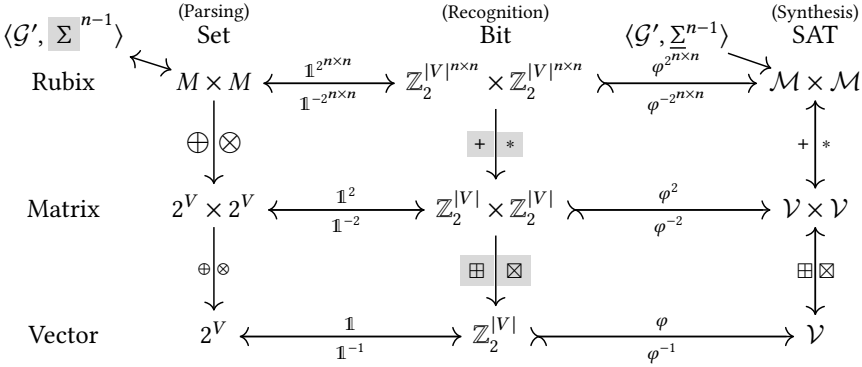
where $\mathcal{V}$ is a function $\mathbb{Z}_2^{|V|} \to \mathbb{Z}_2$. Note that while always possible to encode $\mathbb{Z}_2^{|V|} \to \mathcal{V}$ using the identity function, an arbitrary $\mathcal{V}$ might have zero, one, or in general, multiple solutions in $\mathbb{Z}_2^{|V|}$. In practice, this means that a language equation can be unsatisfiable or underconstrained, however if a solution exists, it can always be decoded into a valid sentence in the language $\mathcal{L}(\mathcal{G})$.

So far, we have only considered the syntactic theory of breadth-bounded CFLs with holes, however, our construction can be easily extended to handle the family of CFLs closed under conjunction. The additional expressivity afforded by the language conjunction will be indispensable when considering more practical program repair scenarios that may not be context-free.

4.2 Linear conjunctive reachability

While generally quite expressive, CFLs are themselves not closed under intersection and have other practical limitations, i.e., are unable to express indentation or variable binding. These limitations motivate us toward more expressive yet still efficiently parsable formalisms. In the case of intersection, let us consider the traditional example, $\mathcal{L}_\cap := \mathcal{L}(\mathcal{G}_1) \cap \mathcal{L}(\mathcal{G}_2)$ defined as follows:

$$P_1 := \{ S \to L\,R, \quad L \to a\,b \mid a\,L\,b, \quad R \to c \mid c\,R \}$$
$$P_2 := \{ S \to L\,R, \quad R \to b\,c \mid b\,R\,c, \quad L \to a \mid a\,L \}$$

Note that $\mathcal{L}_\cap$ generates the language $\{ a^d b^d c^d \mid d > 0 \}$, which according to the pumping lemma is not context-free. However, we can encode the intersection between two or more languages as a single SAT formula by representing each upper-triangular matrix $\bigcap_{i=1}^{c} \mathcal{L}(\mathcal{G}_i)$ as a polygonal prism with upper-triangular matrices adjoined to each rectangular face. More precisely, we intersect all terminals $\Sigma_\cap := \bigcap_{i=1}^{c} \Sigma_i$, then for each $t_\cap \in \Sigma_\cap$ and CFG, construct an equivalence class $E(t_\cap, \mathcal{G}_i) = \{w_i \mid (w_i \to t_\cap) \in P_i\}$ and bind them together using conjunction:

---

[3]Hereinafter, we use gray highlighting to denote types and functions defined over strings and binary constants only.

$$\bigwedge_{t \in \Sigma_{\cap}} \bigwedge_{j=1}^{c-1} \bigwedge_{i=1}^{|\sigma|} E(t_{\cap}, \mathcal{G}_j) \equiv_{\sigma_i} E(t_{\cap}, \mathcal{G}_{j+1}) \tag{2}$$

Fig. 2. Orientations of a $\bigcap_{i=1}^{4} \mathcal{L}(\mathcal{G}_i) \cap \Sigma^6$ configuration. As $c \to \infty$, this shape approximates a tower of Hanoi whose symmetric axis joins σ_i with orthonormal unit productions $w_i \to t_{\cap}$, and $[S_i \in \Lambda_{\sigma}^*]$ inhabiting the outermost bitvectors. Equations of this form are equiexpressive with linear conjunctive grammars, i.e., $\mathbb{G}^+$.

Following Okhotin [29], we then extend our grammar DSL with one additional operator for combining CFGs, $\wedge : \mathbb{G}^+ \times \mathbb{G}^+ \to \mathbb{G}^+$, where $\mathbb{G}^+$ is a conjunctive grammar (CG). In our setting, CGs naturally subsume CFGs, and in so doing, observe the following denotational semantics:

$$\frac{\Gamma \vdash \mathcal{G} : \mathbb{G}}{\Gamma \vdash \mathcal{G} : \mathbb{G}^+} + \quad \frac{\Gamma \vdash \mathcal{G}, \mathcal{G}' : \mathbb{G}^+}{\Gamma \vdash \mathcal{G} \wedge \mathcal{G}' : \mathbb{G}^+} \wedge \quad \frac{\Gamma \vdash \sigma : \mathcal{L}(\mathcal{G}) \quad \Gamma \vdash \sigma : \mathcal{L}(\mathcal{G}')}{\Gamma \vdash \sigma : \mathcal{L}(\mathcal{G} \wedge \mathcal{G}')} \in \quad \frac{\Gamma \vdash \sigma : \mathcal{L}(\mathcal{G} \wedge \mathcal{G}')}{\Gamma \vdash \sigma : \mathcal{L}(\mathcal{G}) \cap \mathcal{L}(\mathcal{G}')} \cap$$

Given two CFGs $\mathcal{G}, \mathcal{G}'$, we can compute the language intersection $\mathcal{L}(\mathcal{G}) \cap \mathcal{L}(\mathcal{G}')$ by encoding $[(\mathcal{M}_{\mathcal{G}}^* \equiv_{\sigma} \mathcal{M}_{\mathcal{G}'}^*)_{i=j-1}]$. With this feature, we can finally express multiway intersections between two or more CFLs, which will be used to encode the bounded Levenshtein-CFL reachability problem.

4.3 Levenshtein reachability

The Levenshtein ball is finite and therefor context-free, however materializing this set is intractable for all but the smallest radii and alphabets. Instead, we dynamically instantiate a CFG that can recognize and generate the members of $\Delta_d(\sigma)$ for any arbitrary $d : \mathbb{N}, \sigma : \Sigma^*$. This approach follows from a straightforward extension of Levenshtein automata [34].

In the case where σ and σ' are both fixed strings, Levenshtein distance can be interpreted as a shortest path problem over an unweighted graph whose vertices are the strings σ, σ', and all possible intermediate editor states, and edges represent the Levenshtein edits. Thus viewed, the Levenshtein distance between σ and σ' is simply the geodesic distance [31]. When σ' is instead a free variable and the distance is fixed, we can define a finite automaton accepting all and only strings within Levenshtein distance d of σ by unrolling the transition dynamics $\mathcal{L}(\sigma, d)$ up to a fixed horizon d.

Fig. 3. Levenshtein reachability from Σ^n can be described as an NFA, or a left-linear CFG.

Levenshtein reachability, then, is recognizable by the nondeterministic infinite automaton (NIA) whose topology $\mathcal{L} = \mathbb{K}$ factorizes into a product of (a) the monotone Chebyshev topology $\mathbb{K}$, equipped with horizontal transitions accepting σ_i and vertical transitions accepting Kleene stars, and (b) the monotone knight's topology, equipped with transitions accepting σ_{i+2}. The structure of this space is approximated by an acyclic NFA, populated by accept states within radius k of $q_{n,0}$, or equivalently, a left-linear CFG whose productions bisimulate the NFA.
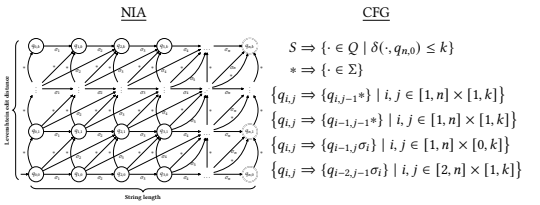
4.4 Bounded Levenshtein-CFL reachability

Let $G(\underset{\sim}{\sigma} : \Sigma^*, d : \mathbb{N}^+) \mapsto \mathbb{G}$ be the specific construction described in §4.3 which accepts a string, $\underset{\sim}{\sigma}$, and an edit distance, d, and returns a grammar representing the NFA that recognizes the language of all strings within Levenshtein radius d of $\underset{\sim}{\sigma}$. To find the language edit distance and corresponding least-distance edits, we must find the smallest d such that $\mathcal{L}_d^\cap$ is nonempty, where $\mathcal{L}_d^\cap$ is defined as $\mathcal{L}(G(\underset{\sim}{\sigma}, d)) \cap \mathcal{L}(\mathcal{G}')$. In other words, we seek $\tilde{\sigma}$ and d^* under which three criteria are all satisfied: (1) $\tilde{\sigma} \in \mathcal{L}(\mathcal{G}')$, and (2) $\Delta(\underset{\sim}{\sigma}, \tilde{\sigma}) \leq d^* \iff \tilde{\sigma} \in \mathcal{L}(G(\underset{\sim}{\sigma}, d^*))$, and (3) $\nexists \sigma' \in \mathcal{L}(\mathcal{G}').\big[\Delta(\underset{\sim}{\sigma}, \sigma') < d^*\big]$. To satisfy these criteria, it suffices to check $d \in (1, d^*]$ by encoding the Levenshtein automata and the original grammar as a single SAT formula, call

Fig. 4. LED is computed gradually by incrementing d until $\mathcal{L}_d^\cap \neq \varnothing$.

it, $\varphi_d(\cdot)$, and gradually admitting new acceptance states at increasing radii. If $\varphi_d(\cdot)$ returns UNSAT, d is increased until either (1) a satisfying assignment is found or (2) d^* is attained. Following 3.2, this procedure is guaranteed to terminate in at most either (1) the number of steps required to overwrite every symbol in $\underset{\sim}{\sigma}$, or (2) the length of the shortest string in $\mathcal{L}(\mathcal{G}')$, whichever is greater. When $\mathcal{L}(\mathcal{G}')$ is context-free, $\mathcal{L}_d^\cap$ provably context-free using the Bar-Hillel construction [7], however in general, the resulting intersection is a conjunctive language.
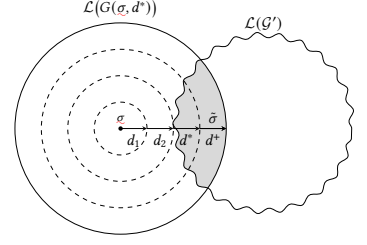
4.5 Tensor sparsification

Although we can encode $[\mathcal{M}_\sigma = \mathcal{M}_\sigma^2]$ explicitly, naïvely encoding the formula as a dense constraint system is highly suboptimal. For starters, we can always ignore all lower-triangular comparisons $\{\mathcal{M}_{ij} \mid i \geq j\}$. A considerably sparser but logically equivalent representation that avoids many elementwise comparisons in the upper-triangular region can be achieved by analyzing the specific string and grammar in question. Since we are typically interested in strings with at least some bound variables, $\{\sigma : \underline{\Sigma}^n \mid \exists \sigma_i \in \Sigma\}$, there is likely to be many upper-triangular nonterminals $\langle v, i, j \rangle \subset V \times \{i, j \mid 0 \leq i \leq j \leq |\sigma|\}$ that are incompatible with σ, i.e., which are unreachable from $\sigma_{i...j}$ regardless of which values they assume. We call these elements *impossible nonterminals*.

Definition 4.3 (Impossible nonterminals). Given $\sigma : \underline{\Sigma}^n$ and $\mathcal{G} : \mathbb{G}$, an impossible nonterminal is a triple $\langle v, i, j \rangle : V \times \{i, j \mid 0 \leq i \leq j \leq |\sigma|\}$ such that $\forall \sigma' : \Sigma^{j-i}, \sigma' \in H(\sigma_{i...j}) \implies v \notin \Lambda_{\sigma'}^*$.

We can immediately rule out nonterminals absent from the parse forest of any static substring.

LEMMA 4.4. *Given $\sigma, \langle v, i, j \rangle$, if $\sigma_{i...j} : \boxed{\Sigma^*}$ and $v \notin \Lambda_{\sigma_{i...j}}^*$ hold, this implies $\langle v, i, j \rangle$ is impossible.*

We will now show how it is possible to discharge many nonterminal triples by considering the CFG reachability of v in relation to its position $\mathcal{M}_{i,j}$ and n, irrespective of the specific string σ.

Definition 4.5 (CFG reachability). Let us define a reachability relation $R(v, \mu, \eta)$ over CFGs as $\{\langle v, \mu, \eta \rangle : (\Sigma \cup V)^2 \times \mathbb{N} \mid \exists \alpha, \beta : (V \cup \Sigma)^*, v \Rightarrow_{\mathcal{G}}^{\leq \eta} \alpha\mu\beta\}$. We write $T^\eta(U)$ to denote the set of all $t : \Sigma \cup V$ where there exists $u \in U$ such that $R(u, t, \eta)$, and likewise $T^{-\eta}(U)$ to denote "..." $R(t, u, \eta)$.

LEMMA 4.6. *Given $n, \langle v, i, j \rangle$, if $v \notin T^{i-j}(\Sigma)$, this implies $v \not\Rightarrow_{\mathcal{G}}^{\leq j-i} \Sigma^{j-i}$ and therefor $\langle v, i, j \rangle$ must be impossible. Furthermore, if $v \notin T^{n-j+i}(\{S\})$, then $S \not\Rightarrow_{\mathcal{G}}^{\leq n-j+i} v$ and therefor $\langle v, i, j \rangle$ must be impossible.*

Regardless of σ, it is always possible to discharge nonterminals inhabiting $\Lambda_{i,j}$ bitvectors outside the $(n - j + i)$-step neighborhood of S, or the $(j - i)$-step neighborhood of any terminal due to unreachability. This constrains several of the least- and greatest- upper diagonals of $\mathcal{M}_\sigma$. Further improvements to sparsity can be achieved by considering the specific structure of σ and $\mathcal{G}$.

Definition 4.7 (Parikh image). Let $\pi : \underline{\Sigma}^* \to \mathbb{N}^{|\Sigma|}$ be the Parikh vector [30], which counts the number of times each terminal appears in a string. We define the Parikh image as the set of all terminals indexed a nonzero element of the Parikh vector, i.e., $\Pi(\sigma : \underline{\Sigma}^*) := \{\sigma' : \Sigma \mid 0 < \pi(\sigma)[\sigma']\}$.

LEMMA 4.8. *Given* $\sigma, \langle v, i, j \rangle$, *then* $\Pi(\sigma_{i\ldots j}) \nsubseteq T^\infty(\{v\})$ *implies* $v \not\Rightarrow_{\mathcal{G}}^* \sigma_{i\ldots j}$ *and* $\langle v, i, j \rangle$ *is impossible.*

We can overapproximate the set of possible nonterminals by comparing the Parikh image of $\sigma_{i\ldots j}$ with the closure of v, then discharging impossible nonterminal triples $\langle v, i, j \rangle$. If $\Pi(\sigma_{i\ldots j}) \nsubseteq T^\infty(\{v\})$, then we know $\sigma_{i\ldots j}$ cannot derive v and $\left[(\mathcal{M}_\sigma = \mathcal{M}_\sigma^2)_{i,j,v} \right]$ may be safely ignored. Note that a tighter approximation can be achieved by considering the minimal Parikh vectors of all $\sigma : \Sigma^\eta$ such that $v \Rightarrow_{\mathcal{G}}^* \sigma \in \mathrm{H}(\sigma_{i\ldots j})$, rather than just the image on V, however this would be computationally more expensive and $\Pi(\sigma)$ already discharges a large fraction of impossible nonterminals in practice.

4.6 Isolation and reduction

Depicted right is a SAT tensor representing $\sigma_1\ \sigma_2\ \sigma_3 \_ \cdots \_$ where shaded regions demarcate known bitvector literals $\Lambda_{r,c}$ (i.e., representing established nonterminal forests) and unshaded regions correspond to bitvector variables $\Lambda_{r,c}$ (i.e., representing unknown nonterminal forests to be solved) for an incomplete string. Since $\Lambda_{r,c}$ are fixed, we can precompute them outside of the SAT solver.

Fig. 5. When is this reduction admissible?

Clearly, formula complexity is heavily dependent on $|\sigma|$ and for the sake of complexity, it would be highly advantageous if σ were shorter. Naturally, this raises the question of when well-formed substrings can be collapsed, i.e., under what circumstances can the reduction in Fig. 5 be applied? This transformation is admissible when a subexpression is "isolated", i.e., its derivation cannot somehow be altered by appending or prepending text. For example, the string (- b) is *isolated* in the sense that adjoining text should not alter the interior derivation, whilst - b is not, as adjacent text (e.g., a - b) may

Fig. 6. We can inspect $\Lambda_\alpha^\to, \Lambda_\alpha^\uparrow$ to find out.

alter the derivation of its contents. This question can be reduced to a quotient: does there exist any string, that when so adjoined will "strip away" any nonterminals, leading to another derivation?

More formally, given an arbitrary (potentially ambiguous) context free grammar $\mathcal{G} : \mathbb{G}$, and string $\alpha : \Sigma^*$, is there a decision procedure that returns whether appending and/or prepending symbols can alter the parse forest of α? In other words, we want a function $F : (\mathbb{G} \times \Sigma^*) \to \mathbb{B}$ that returns whether α's parse forest according to $\mathcal{G}$ is unique over $\beta\alpha\gamma$, for all $\beta, \gamma : \underline{\Sigma}^*$. Specifically,

Definition 4.9 (Isolation). Let T_α denote the set of all parse trees that are generated by the string α using $\mathcal{G}$, and consider $\mathbf{T}_\alpha$, the union of all parse trees and their subtrees that can be generated by $\beta\alpha\gamma$ using $\mathcal{G}$ for arbitrary $\beta, \gamma \in \underline{\Sigma}^*$, and have a leaf in α. We call the parse forest T_α *isolated* iff for all $t \in \mathbf{T}_\alpha$, there exists $t' \in T_\alpha$, such that t is either a subtree of t', or t' is a proper subtree of t.

If we can identify an $\alpha : \Sigma^*$ where $\alpha = \sigma_{i\ldots j}$ for some $i, j : \mathbb{N}^{<|\sigma|}$, such that α has an isolated parse forest, then we can safely replace $\sigma_{i\ldots j} \mapsto \Lambda_\alpha^*$. For bounded-length strings, this can be solved by padding α with adjacent holes $(\_)^n \alpha (\_)^n$ for sufficiently large n, and checking $\Lambda_\alpha^{\scriptsize\updownarrow}$, the union of all right- and left-quotients of α for emptiness, i.e., $[\varnothing = \Lambda_\alpha^\uparrow \cup \Lambda_\alpha^\to]$. A general solution for arbitrary CFGs, $\mathcal{G} : \mathbb{G}$ and $\beta, \gamma \in \Sigma^*$ is more difficult, but could significantly extend the context window and remove the need for ad hoc preprocessing. We leave this as an open problem for future work.
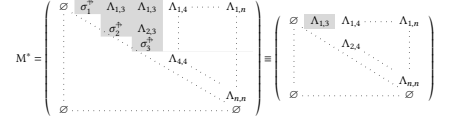
4.7 Sampling the Levenshtein ball without replacement in $\mathcal{O}(1)$

Now that we have a reliable method to synthesize admissible completions for strings containing holes, i.e., fix *localized* errors, $F : (\mathcal{G} \times \underline{\Sigma}^n) \to \{\Sigma^n\} \subseteq \mathcal{L}(\mathcal{G})$, how can we use F to repair some unparseable string, i.e., $\sigma_1 \ldots \sigma_n : \Sigma^n \cap \mathcal{L}(\mathcal{G})^{\complement}$ where the holes' locations are unknown? Three questions stand out in particular: how many holes are needed to repair the string, where should we put those holes, and how ought we fill them to obtain a parseable $\tilde{\sigma} \in \mathcal{L}(\mathcal{G})$?

One plausible approach would be to draw samples with a PCFG, minimizing tree-edit distance, however these are computationally expensive metrics and approximations may converge poorly. A more efficient strategy is to sample string perturbations, $\boldsymbol{\sigma} \sim \Sigma^{n\pm q} \cap \Delta_q(\underline{\sigma})$ uniformly across the Levenshtein q-ball centered on $\underline{\sigma}$, i.e., the space of all admissible edits with Levenshtein distance $\leq q$, loosely analogous to a finite difference approximation over words in a finite language.

To implement this strategy, we first construct a surjection $\varphi^{-1} : \mathbb{Z}_2^m \twoheadrightarrow \Delta_q(\underline{\sigma})$ from bitvectors to Levenshtein edits over $\underline{\sigma}, \Sigma$, sample bitvectors without replacement using a characteristic polynomial, then decode the resulting bitvectors into Levenshtein edits. This ensures the sampler eventually visits every Levenshtein edit at least exactly once and at most approximately once, without needing to store any samples in memory, and discovers a steady stream of admissible edits throughout the solving process, independent of the grammar or string under repair.

More specifically, we employ a pair of [un]tupling functions $\kappa, \rho : \mathbb{N}^k \leftrightarrow \mathbb{N}$ which are (1) bijective (2) maximally compact (3) computationally tractable (i.e., closed form inverses). κ will be used to index $\{^n_k\}^2$-combinations via the Maculay representation [25] and ρ will index Σ^k tuples, but is slightly more tricky to define. To maximize compactness, there is an elegant pairing function courtesy of Szudzik [36], which enumerates concentric square shells over the plane $\mathbb{N}^2$ and can be generalized to hypercubic shells in $\mathbb{N}^k$. For our purposes, this generalization will suffice.

Although $\langle \kappa, \rho \rangle$ could be used directly to exhaustively search the Levenshtein ball, they are temporally biased samplers due to lexicographic ordering. Rather, we would prefer a path that uniformly visits every fertile subspace of the Levenshtein ball over time regardless of the grammar or string in question: subsequences of $\langle \kappa, \rho \rangle$ should discover valid repairs with frequency roughly proportional to the filtration rate, i.e., the density of the admissible set relative to the Levenshtein ball. These additional constraints give rise to two more criteria: (4) ergodicity and (5) periodicity.

To achieve ergodicity, we permute the elements of $\{^n_k\} \times \Sigma^k$ using a finite field with a characteristic polynomial C of degree $m := \lceil \log_p \binom{n}{k} |\Sigma_\varepsilon|^k \rceil$. By choosing C to be some irreducible polynomial, one ensures the path has the mixing properties we desire, e.g., suppose $U : \mathbb{Z}_2^{m \times m}$ is a matrix whose structure is depicted to the right, wherein C represents a primitive polynomial over $\mathbb{Z}_2^m$ with known coefficients $C_{1\ldots m}$ and semiring op-

$$U^\mathsf{T}Y = \begin{pmatrix} C_1 & \cdots\cdots\cdots\cdots & C_m \\ \top & \circ & & \circ \\ \circ & & & \vdots \\ \vdots & & & \vdots \\ \circ & \cdots & \circ & \top & \circ \end{pmatrix}^t \begin{pmatrix} Y_1 \\ \vdots \\ \vdots \\ \vdots \\ Y_m \end{pmatrix}$$

erators $\oplus := + \pmod 2$, $\otimes := \wedge$, $\top := 1$, $\circ := 0$. Since C is primitive, the sequence $\mathbf{R} = (U^{0\ldots 2^m-1}Y)$ must have *full periodicity*, i.e., for all $i, j \in [0, 2^m)$, $\mathbf{R}_i = \mathbf{R}_j \Rightarrow i = j$. To uniformly sample $\boldsymbol{\sigma}$ without replacement, we construct a partial surjective function from $\mathbb{Z}_2^m$ onto the Levenshtein ball, $\mathbb{Z}_2^m \rightharpoonup \{^n_d\} \times \Sigma_\varepsilon^d$, cycle over $\mathbf{R}$, then discard samples which have no witness in $\{^n_d\} \times \Sigma_\varepsilon^d$.

This procedure requires $\mathcal{O}(1)$ per sample and roughly $\binom{n}{d}|\Sigma_\varepsilon|^d$ samples to exhaustively search $\{^n_d\} \times \Sigma_\varepsilon^d$. Its acceptance rate $p^{-m}\binom{n}{d}|\Sigma_\varepsilon|^d$ can be slightly improved with a more suitable base p, however this introduces some additional complexity and so we elected to defer this optimization.

In addition to its statistically desirable properties, our sampler has the practical benefit of being trivially parallelizable using the leapfrog method, i.e., given p independent processors, each one can

---

[2]Following Stirling, we use the notation $\{^n_d\}$ to denote the set of all d-element subsets of $\{1, \ldots, n\}$.

independently check $[\varphi^{-1}(\langle\kappa,\rho\rangle^{-1}(\mathbf{R}_i),\sigma)\in\mathcal{L}(\mathcal{G})]$ where $i\equiv p_j\pmod{p}$. This procedure linearly scales with the number of processors, exhaustively searching $\Delta_q(\sigma)$ in p^{-1} of the time required by a single processor, or alternately drawing p times as many samples in the same amount of time.

To admit variable-length edits and enable deletion, we first define a ε^+-production and introduce it to the right- and left-hand side of each terminal in a unit production in our grammar, $\mathcal{G}$:

$$\frac{\mathcal{G}\vdash\varepsilon\in\Sigma}{\mathcal{G}\vdash(\varepsilon^+\to\varepsilon\mid\varepsilon\,\varepsilon^+)\in P}\;\varepsilon\text{-DUP}\qquad\frac{\mathcal{G}\vdash(A\to B)\in P}{\mathcal{G}\vdash(A\to B\,\varepsilon^+\mid\varepsilon^+\,B\mid B)\in P}\;\varepsilon^+\text{-INT}$$

Finally, to sample $\sigma\sim\Delta_q(\sigma)$, we first interleave σ as σ_ε (see Lemma 3.3), then enumerate hole templates $\mathrm{H}(\sigma_\varepsilon,i)=\sigma_{1\ldots i-1}\_\_\sigma_{i+1\ldots n}$ for each $i\in\cdot\in\binom{n}{d}$ and $d\in 1\ldots q$, then solve for $\tilde{\sigma}\in\mathrm{H}(\sigma_\varepsilon,i)$ satisfying $[S\in\Lambda^*_{\tilde{\sigma},\mathcal{G}}]\Leftrightarrow[\tilde{\sigma}\in\mathcal{L}(\mathcal{G})]$. If $\boldsymbol{\sigma}:=\mathrm{H}(\sigma_\varepsilon,i)$ is nonempty, then each edit from each patch in each $\tilde{\sigma}\in\boldsymbol{\sigma}$ will match one of the following patterns, covering all three Levenshtein edits:

$$\text{Deletion}=\left\{\ldots\sigma_{i-1}\;\gamma_1\;\gamma_2\;\sigma_{i+1}\ldots\quad\gamma_{1,2}=\varepsilon\right.$$

$$\text{Substitution}=\begin{cases}\ldots\sigma_{i-1}\;\gamma_1\;\gamma_2\;\sigma_{i+1}\ldots & \gamma_1\neq\varepsilon\wedge\gamma_2=\varepsilon\\\ldots\sigma_{i-1}\;\gamma_1\;\gamma_2\;\sigma_{i+1}\ldots & \gamma_1=\varepsilon\wedge\gamma_2\neq\varepsilon\\\ldots\sigma_{i-1}\;\gamma_1\;\gamma_2\;\sigma_{i+1}\ldots & \{\gamma_1,\gamma_2\}\cap\{\varepsilon,\sigma_i\}=\varnothing\end{cases}$$

$$\text{Insertion}=\begin{cases}\ldots\sigma_{i-1}\;\gamma_1\;\gamma_2\;\sigma_{i+1}\ldots & \gamma_1=\sigma_i\wedge\gamma_2\notin\{\varepsilon,\sigma_i\}\\\ldots\sigma_{i-1}\;\gamma_1\;\gamma_2\;\sigma_{i+1}\ldots & \gamma_1\notin\{\varepsilon,\sigma_i\}\wedge\gamma_2=\sigma_i\\\ldots\sigma_{i-1}\;\gamma_1\;\gamma_2\;\sigma_{i+1}\ldots & \gamma_{1,2}=\sigma_i\end{cases}$$

Although complete with respect to $\Delta_q(\sigma)$, this approach can produce patches containing more Levenshtein edits than are strictly necessary to repair σ. To ensure patches are both minimal and syntactically valid, we first introduce a simple technique to minimize the repairs in §4.8. By itself, uniformly sampling minimal repairs $\tilde{\sigma}\sim\Delta_q(\sigma)\cap\mathcal{L}(\mathcal{G})$ is sufficient but can be quite time-consuming, as we empirically show in §8.1. To further reduce sample complexity and enable real-time repairs, we will then introduce a more efficient density estimator based on adaptive resampling (§4.9).

4.8 Patch minimization

Suppose we have a string, a (b, and discover the patch, $\tilde{\sigma}=$ (a + b). Although $\tilde{\sigma}$ is syntactically admissible, it is not minimal. To minimize a patch, we consider the set of all of its constituent subpatches, namely, (a + b, (a (b), a + b), (a (b, a + b, and a (b), then retain only the smallest syntactically valid instance(s) by Levenshtein distance. This forms a so-called *patch powerset*, which can be lazily enumerated from the top-down using the Macaulay representation, after which we take all valid elements from the lowest level containing at least one admissible element, i.e., a + b and a (b). When patches are very large, minimization can be used in tandem with the delta debugging technique [41] to first

Fig. 7. Patch powerset of $\tilde{\sigma}=$ (a + b).

simplify contiguous edits, then apply the patch powerset construction. Minimization is often useful for estimating the language edit distance: given a single valid repair of arbitrary size, minimization lets us quickly approximate an upper-bound on $\Delta(\sigma,\ell)$ – much tighter than indicated by Lemma 3.2.
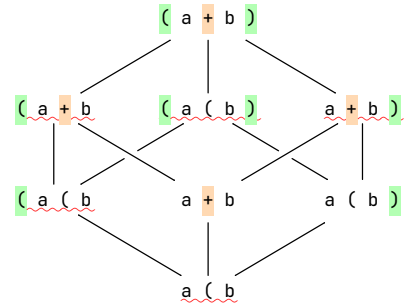
4.9 Probabilistic reachability

Since there are $\Sigma_{d=1}^{q} \binom{n}{d}$ total hole templates, each with $|\Sigma_\varepsilon|^d$ individual edits to check, if n and q are large, this space can be slow to exhaustively search and a uniform prior may be highly sample-inefficient. Furthermore, naïvely sampling $\sigma \sim \Delta_q(\sigma)$ is likely to produce a large number of unnatural edits and converge poorly on $\Delta_q(\sigma) \cap \mathcal{L}(\mathcal{G})$. To rapidly rank and render relevant repair recommendations, we prioritize candidate edits according to the following six-step procedure:

(1) Draw samples $\hat{\sigma} \sim \Delta_q(\sigma)$ without replacement using §4.7 with leapfrog parallelization.

(2) Score by perplexity $PP(\hat{\sigma})$ using a pretrained variable-order Markov chain (VOMC) [35].

(3) Resample using a concurrent variant of the A-Res [21] online weighted reservoir sampler.

(4) Filter Levenshtein edits by admissibility with respect to the grammar, i.e., $[\hat{\sigma} \in \mathcal{L}(\mathcal{G})]$.

(5) Minimize and store admissible repairs to a replay buffer, $\mathcal{Q} \leftarrow \tilde{\sigma}$, ranked by perplexity.

(6) Repeat steps (1)-(5), alternately sampling from the LFSR/VOMC-reweighted online resevoir sampler with probability ϵ or stochastically resampled $\mathcal{Q}$ with probability $(1 - \epsilon)$, where ϵ decreases from 1 to 0 according to a stepwise schedule relative to the time remaining.

Initially, the replay buffer $\mathcal{Q}$ is empty and repairs are sampled uniformly without replacement from the Levenshtein ball, $\Delta_q(\sigma)$. As time progresses, $\mathcal{Q}$ is gradually populated with admissible repairs and resampled with increasing probability, allowing the algorithm to initially explore, then exploit the most promising candidates. This is summarized in Algorithm 1 which is run in parallel across all available CPU cores.

We would prefer hole templates likely to yield repairs that are (1) admissible (i.e., grammatically correct) and (2) plausible (i.e., likely to have been written by a human author). To

---

Algorithm 1 Probabilistic reachability

Require: $\mathcal{G}$ grammar, σ broken string, p process ID, c total CPU cores, t_total timeout.
1: $\mathcal{Q} \leftarrow \varnothing, \mathcal{R} \leftarrow \varnothing, \epsilon \leftarrow 1, i \leftarrow 0, Y \sim \mathbb{Z}_2^m, t_0 \leftarrow t_\text{now}$ ▷ Initialize replay buffer $\mathcal{Q}$ and reservoir $\mathcal{R}$.
2: **repeat**
3:    **if** $\mathcal{Q} = \varnothing$ or **Rand**$(0, 1) < \epsilon$ **then**
4:       $\hat{\sigma} \leftarrow \varphi^{-1}\left(\langle\kappa, \rho\rangle^{-1}(U^{ci+p}Y), \sigma\right), i \leftarrow i + 1$ ▷ Sample WoR using the leapfrog method.
5:    **else**
6:       $\hat{\sigma} \sim \mathcal{Q} + \textbf{Noise}(\mathcal{Q})$ ▷ Sample replay buffer with additive noise.
7:    **end if**
8:    $\mathcal{R} \leftarrow \mathcal{R} \cup \{\hat{\sigma}\}$ ▷ Insert repair candidate $\hat{\sigma}$ into reservoir $\mathcal{R}$.
9:    **if** $\mathcal{R}$ is full **then**
10:       $\hat{\sigma} \leftarrow \text{argmin}_{\hat{\sigma} \in \mathcal{R}} PP(\hat{\sigma})$ ▷ Select lowest perplexity repair candidate.
11:       **if** $\hat{\sigma} \in \mathcal{L}(\mathcal{G})$ **then**
12:          $\mathcal{Q} \leftarrow \mathcal{Q} \cup \{\hat{\sigma}\}$ ▷ Insert successful repair into replay buffer.
13:       **end if**
14:       $\mathcal{R} \leftarrow \mathcal{R} \setminus \{\hat{\sigma}\}$ ▷ Remove checked sample from the reservoir.
15:    **end if**
16:    $\epsilon \leftarrow \textbf{Schedule}\left((t_\text{now} - t_0)/t_\text{total}\right)$ ▷ Update exploration/exploitation rate.
17: **until** t_total elapses.
18: **return** $\tilde{\sigma} \in \mathcal{Q}$ ranked by $PP(\tilde{\sigma})$.

---

do so, we draw holes and rank admissible repairs using a probabilistic distance metric over $\Delta_q(\sigma)$. For example, suppose we are given an invalid string, $\sigma_\varepsilon : \Sigma^{90}$ and $\mathcal{Q} \subseteq [0, |\sigma_\varepsilon|) \times \Sigma_\varepsilon^q$, a distribution over previously successful edits, which we can use to localize admissible repairs. By marginalizing onto σ_ε, the distribution $\mathcal{Q}(\sigma_\varepsilon)$ could take the form depicted in Fig. 8.

More specifically in our setting, we want to sample from a discrete product space that factorizes into (1) the specific edit locations (e.g., informed by caret position, historical edit locations, or a static analyzer), (2) probable completions (e.g., from a Markov chain or neural language model) and (3) an accompanying *cost model*, $C : (\Sigma^* \times \Sigma^*) \to \mathbb{R}$, which may be any number of suitable distance metrics, such as
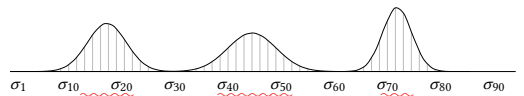

Fig. 8. The distribution $\mathcal{Q}$, projected onto the invalid string, suggests edit locations likely to yield admissible repairs, from which we draw subsets of size d.

language edit distance, finger travel distance on a physical keyboard, weighted Levenshtein distance, or stochastic contextual edit distance [16] in the case of probabilistic edits. Our goal then, is to discover repairs which minimize $C(\sigma, \tilde{\sigma})$, subject to the given grammar and latency constraints.

5 PARSING

Although parsing is not the primary objective of this work, it is an integral component of the repair process, and one of the advantages of using a matrix representation is its excellent error recovery properties. Unlike traditional parsers which fail on an error, matrix-based parsers support parsing invalid strings. In this section, we will describe how to parse strings using matrix powering, and how we use those results aid human debugging and inform the repair process.

5.1 Tree denormalization

Our parser emits a binary forest consisting of parse trees for the candidate string which are constructed bottom-up using a variant of $\otimes$ called $\hat{\otimes}$, which simply records backpointers:

$$X \,\hat{\otimes}\, Z := \Big\{\, w \begin{smallmatrix} \nearrow x \\ \searrow z \end{smallmatrix} \mid \langle x, z \rangle \in X \times Z, (w \to xz) \in P \,\Big\} \tag{3}$$

Due to Chomsky normalization however, the resulting forests are full of trees that are thin and crooked. To restore the natural shape of the tree, we first construct the parse forests bottom-up, then prune away synthetic nonterminals top-down by recursively grafting denormalized grandchildren onto the root. This transformation is purely cosmetic and only used when rendering the parse trees.

Algorithm 2 Tree denormalization

procedure CUT(t: Tree)
    stems ← {CUT(c) | c ∈ t.children}
    if t.root ∈ $(V_{\mathcal{G}'} \setminus V_{\mathcal{G}})$ **then**
        return stems
    else
        return { Tree(t.root, stems) }
    end if
end procedure
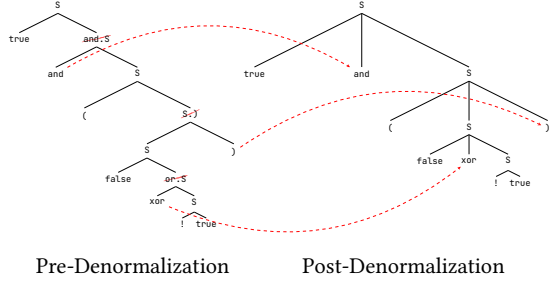

Pre-Denormalization   Post-Denormalization

Fig. 9. Since $\mathcal{G}'$ contains synthetic nodes, to recover a parse tree congruent with the original grammar $\mathcal{G}$, we prune all synthetic nodes and graft their stems onto the grandparent via a simple recursive procedure (Alg. 2).

5.2 Relation between parsing and repair

Parsing and repair are intimately related: parsing tells us roughly where the repairs should occur and repair is aided by the results of parsing: parseable subtrees can be used to guide the repair process. Although reparsing may be avoided after repair, it is not straightforward how to decode the solution to the matrix equivalence relation as a parse forest, so we resort to decoding the generating string, then reparsing it to obtain the concrete syntax trees (CSTs).

Fig. 10. Parsing and repair share the same grammar.

The process of parsing the string $S + S = S$ by matrix powering is depicted below. This occurs by repeatedly joining rooted subtrees using the $\hat{\otimes}$ operator. In this particular case, the parse forest is unique, although in general, M_{ij} may contain multiple subtrees should ambiguity arise:
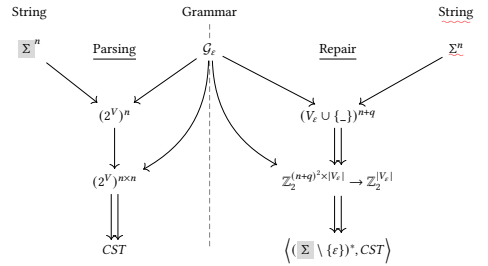
Fig. 11. The parse forest for the string $S + S = S$ is constructed incrementally by computing $M_\sigma^2 + M_\sigma = M_\sigma$.

For valid strings, the fixpoint search is sure to converge in at most $|\sigma| - 1$ steps, as each iteration will solve for a single diagonal of the matrix. Matrix entries may each contain up to $|V_\varepsilon|$ rooted subtrees, although in practice, this is typically much smaller for unambiguous grammars. If the string is invalid, the algorithm produces a set of admissible corrections, alongside their CSTs. An interesting consequence of repairing with a SAT solver is that we can directly encode matrix idempotency $[\mathcal{M}_\sigma = \mathcal{M}_\sigma^2]$, instead of solving Valiant's fixpoint $M_\sigma = M_\sigma + M_\sigma^2$.

5.3 Incrementalization

When a parsed string is altered, we can reuse prior work by only recomputing affected submatrices, yielding a reparser whose complexity is location-dependent, i.e., at worst quadratic in terms of $|\Sigma^*|$ assuming $\mathcal{O}(1)$ cost for each CNF-nonterminal subset join, $V_1' \otimes V_2'$. Letting shaded nodes represent observed or bound variables and unshaded nodes as free variables, we depict the worst-case post-editing state of the parse trellis in Fig. 12.
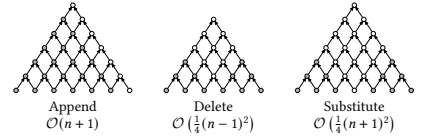

Fig. 12. Incremental reparsing only requires computing submatrices affected by an edit.

The problem of incremental parsing is closely related to *dynamic matrix inversion* in the linear algebra setting, and *incremental transitive closure* with vertex updates in the graph setting. By carefully encoding the matrix relation from §4 and employing an incremental SAT solver, we can gradually update SAT constraints as new keystrokes are received to eliminate redundancy.

5.4 Error recovery

Not only is Tidyparse capable of suggesting repairs to invalid strings, it can also return partial trees for those same strings, which is often helpful for debugging purposes. Unlike LL- and LR-style parsers which require special rules for error recovery, Tidyparse can simply analyze the structure of M^* to recover parse branches. If $S \notin \Lambda_\sigma^*$, the upper triangular entries of M^* will take the form of a jagged-shaped ridge whose peaks signify the roots of maximally-parsable substrings $\hat{\sigma}_{i,j}$.

These branches are located on peaks of the upper triangular (UT) matrix ridge. As depicted in Fig. 14, we traverse the peaks by decreasing elevation to collect partial AST branches and display the highest nonoverlapping branches, in this case T_C and T_A to the user, to help them diagnose the parsing error and manually repair it.
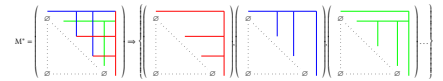

Fig. 13. The matrix M^* contains all admissible binary trees of a fixed breadth.
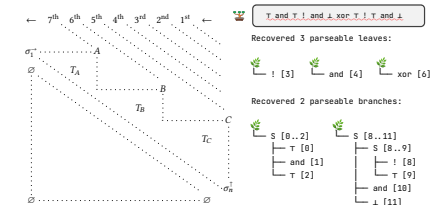

Fig. 14. We can recover the partial subtrees for invalid strings by inspecting M^*.

6 USAGE EXAMPLES

Tidyparse offers a user interface featuring a text editor, a grammar editor and a parse tree viewer for interactive prototyping. For example, suppose we have the following context-free grammar:

```
S -> S and S | S xor S | ( S ) | true | false | ! S
```

As described in § 4, this grammar will automatically be rewritten into Chomsky Normal Form:

F.! → !	S.) → S F.)	and.S → F.and S	S → F.! S	S → false	S → S ε+
F.(→ (	F.xor → xor	xor.S → F.xor S	S → S and.S	S → true	ε+ → ε
F.) →)	F.and → and	S → S xor.S	S → F.(S.)	S → <S>	ε+ → ε+ ε+

Given a string containing holes, our tool will return several completions in a few milliseconds:

```
true _ _ _ ( false _ ( _ _ _ _ ! _ _ ) _ _ _ _
-------------------------------------------------
1. true xor ! ( false xor ( <S> ) or ! <S> ) xor <S>
2. true xor ! ( false and ( <S> ) or ! <S> ) xor <S>
3. true xor ! ( false and ( <S> ) and ! <S> ) xor <S>
4. true xor ! ( false and ( <S> ) and ! <S> ) and <S>
...
```

Similarly, if provided with a string containing various errors, it will return several suggestions how to fix it, where green is insertion, orange is substitution and red is deletion.

```
true and ( false or and true false
-------------------------------------------------
1. true and ( false or ! true )
2. true and ( false or <S> and true )
3. true and ( false or ( true ) )
...
9. true and ( false or ! <S> ) and true false
```

Since CFLs are closed under homomorphisms, it is possible to unify lexing and parsing, however most languages explicitly define a separate lexer, which we avail to substitute named identifiers with their type. Given an invalid string, the tool will first abstract the raw characters, generate edits in the abstract token space, then remap successful repairs back to character space as shown below:

```
d = sum([foo(i] for i in vals))
-------------------------------------
1. d = sum([foo(i) for i in vals])
2. d = sum([(i) for i in vals])
3. d = sum([foo.i for i in vals])
4. d = sum([foo(+i) for i in vals])
```

```
w = w ( [ w ( w ] for w in w ) )
-------------------------------------
1. w = w ( [ w ( i ) for i in w ] )
2. w = w ( [ w ( w ) for w in w ] )
3. w = w ( [ w . w )  for w in w ] )
4. w = w ( [ w ( + w ) for w in w ) )
```

736 This coarsening reduces the number of possible corrections, although is not strictly necessary.
737 For simplicity, it is also possible to define a grammar and string side-by-side, as shown in the
738 untyped λ-calculus example below:

```
sxp -> λ var . sxp | sxp sxp | var | ( sxp )
var -> a | b | c | f | x | y | z
---
( λ f . ( λ x . f ( x x ) ) ( λ x . f ( x x ) █
------------------------------------------------------------
1. ( λ f . ( λ x . f ( x x ) ) ) λ x . f ( x x )
2. ( λ f . ( λ x . f ( x x ) ) x ) λ x . f ( x x )
3. ( λ f . ( λ x . f ( x x ) ) ( λ x . f ( x ) ) )
```

Name resolution and scope checking is also possible but requires a more sophisticated grammar.

6.1 Syntax highlighting

We use the parse forest from §5.4 to segment maximally parseable regions. Subsequences which can
be partly parseable are underlined in blue. Alphabetic tokens which cannot be joined are marked
orange. All other tokens are marked red.

```
( true xor false ) and true xor  and  not false
```

6.2 Grammar assistance

Tidyparse uses a CFG to parse the CFG, so it can provide editing assistance while the user is
designing the CFG. For example, if the CFG does not parse, will suggest a list of possible fixes.

```
START -> CFG
CFG -> PRD | CFG \n CFG
PRD -> TOK '->' RHS
TOK -> [A-Za-z']+
RHS -> TOK | RHS RHS | RHS '|' RHS
```

```
B ::= true | false | █
------------------------------------
1. B -> true | false
2. B -> true | false <RHS>
3. B -> true | false | <RHS>
```

6.3 Interactive nonterminal expansion

Users can interactively build up a complex expression by placing the caret over a nonterminal they
wish to expand, then pressing `ctrl`+`Space` to receive a list of possible substitutions.

```
true and ( false or <S> and true )
------------------------------------------
1. true and ( false or true and true )
2. true and ( false or false and true )
3. true and ( false or ! <S> and true )
```

6.4 Nonterminal stubs

Tidyparse augments CFGs with two additional rules, which are desugared into a vanilla CFG before parsing. The first rule, α-SUB, allows the user to define a nonterminal parameterized by α, a non-recursive nonterminal in the same the CFG representing some finite type and its inhabitants. α-SUB replaces all productions containing $\langle\alpha\rangle$ with the terminals in their transitive closure, $\alpha \rightarrow^* \beta$. The second rule, α-INT, introduces homonymous terminals for each user-defined nonterminal.

$$\frac{\mathcal{G} \vdash (w\langle\alpha\rangle \rightarrow xz) \in P \qquad \alpha^* : \{\beta \mid (\alpha \rightarrow^* \beta) \in P\}}{\mathcal{G} \vdash \forall\beta \in \alpha^*.(w\langle\alpha\rangle \rightarrow xz)[\beta/\alpha] \in P'} \; \alpha\text{-SUB} \qquad \frac{\mathcal{G} \vdash v \in V}{\mathcal{G} \vdash (v \rightarrow \langle v\rangle) \in P} \; \langle\cdot\rangle\text{-INT}$$

Tidyparse can also perform a limited form of type checking. Typed expressions are automatically expanded into ordinary nonterminals using the α-SUB rule, for example when parsing an expression of the form $x + y$, the grammar will recognize `true + false` and `1 + 2`, but not `1 + true`.

```
E<X> -> E<X> + E<X> | E<X> * E<X> | ( E<X> )
X -> Int | Bool
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
E<Int> -> E<Int> + E<Int> | E<Int> * E<Int>
E<Bool> -> E<Bool> + E<Bool> | E<Bool> * E<Bool>
```

6.5 Conjunctive grammars

Many natural and programming languages exhibit context-sensitivity, such as Python indentation. Unlike ordinary parser-generators, Tidyparse can encode CFL intersection, allowing it to correct errors in linear conjunctive languages. For example, consider the same language from § 4.2, with $\mathcal{G}_1$ to the left and $\mathcal{G}_2$ to the right, whose conjunction, $\mathcal{G}_1 \wedge \mathcal{G}_2$, recognizes the language $a^n b^n c^n$.

```
S -> LEFT RIGHT
LEFT -> a b | a LEFT b
RIGHT -> c | c RIGHT
```

```
S -> LEFT RIGHT
RIGHT -> b c | b RIGHT c
LEFT -> a | a LEFT
```

```
1. a b c
2. a a b b c c
3. a a a b b b c c c
...
```

Tidyparse uses a programmatic interface to signify $\mathcal{L}(\mathcal{G}_1) \cap \mathcal{L}(\mathcal{G}_2)$, i.e., the intersection of two more grammars' languages, as described in §4.2. This allows the expression of limited indexicality and opens the door to more semantic program analyses like scope checking and name resolution.

7 RELATED WORK

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

7.1 Parsing

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

General CFL parsing allows ambiguity (non-unique trees) and can be formulated as a dynamic programming problem, as shown by Cocke-Younger-Kasami (CYK) [32], Earley [20] and others. These parsers have roughly cubic complexity with respect to the length of the input string.

As shown by Valiant [38], Lee [26] 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 unlocks the door to a wide range of complexity-theoretic and practical speedups to CFL recognition and fast general parsing algorithms.

Okhotin (2001) [29] extends CFGs with language conjunction in *conjunctive grammars*, followed by Zhang & Su (2017) [42] who apply conjunctive language reachability to dataflow analysis.

7.2 Language equations

Language equations are a powerful tool for reasoning about formal languages and their inhabitants. First proposed by Ginsburg et al. [22] for the ALGOL language, language equations are essentially systems of inequalities with variables representing *holes*, i.e., unknown values, in the language or grammar. Solutions to these equations can be obtained using various fixpoint techniques, yielding members of the language. This insight reveals the true algebraic nature of CFLs and their cousins.

Being an algebraic formalism, language equations naturally give rise to a kind of calculus, vaguely reminiscent of Leibniz' and Newton's. First studied by Brzozowski [11, 12] and Antimirov [4], one can take the derivative of a language equation, yielding another equation. This can be interpreted as a kind of continuation or language quotient, revealing the suffixes that complete a given prefix. This technique leads to an elegant family of algorithms for incremental parsing [1, 28] and automata minimization [10]. In our setting, differentiation corresponds to code completion.

In this paper, we restrict our attention to language equations over context-free and weakly context-sensitive languages, whose variables coincide with edit locations in the source code of a computer program, and solutions correspond to syntax repairs. Although prior work has studied the use of language equations for parsing [28], to our knowledge they have never previously been considered for the purpose of code completion or syntax error correction.

7.3 Syntax repair

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

When a sentence is grammatically invalid, parsing grows more challenging. Like spelling, the problem is to find the minimum number of edits required to transform an arbitrary string into a syntactically valid one, where validity is defined as containment in a (typically) context-free

language. Early work, including Irons [23] and Aho [2] propose a dynamic programming algorithm to compute the minimum number of edits required to fix an invalid string. Prior work on error correcting parsing only considers the shortest edit(s), and does not study multiple edits over the Levenshtein ball. Furthermore, the problem of actually generating the repairs is not well-posed, as there are usually many valid strings that can be obtained within a given number of edits. We instead focus on bounded Levenshtein reachability, which is the problem of finding useful repairs within a fixed Levenshtein distance of the broken string, which requires language intersection.

7.4   Classical program synthesis

There is related work on string constraint solving in the constraint programing literature, featuring solvers like CFGAnalyzer and HAMPI [24], which consider bounded context free grammars and intersections thereof. Axelson et al. (2008) [6] has some work on incremental SAT encoding but does not exploit the linear-algebraic structure of parsing, conjunctive reachability nor provide real-time guarantees. D'Antoni et al. (2014) introduces *symbolic automata* [17], a generalization of finite automata which allow allow infinite alphabets and symbolic expressions over them. In none of the constraint programming literature we surveyed do any of the approaches employ matrix-based parsing, and therefor do not enjoy the optimality guarantees of Valiant's parser. Our solver can handle context-free and conjunctive grammars with finite alphabets and does not require any special grammar encoding. The matrix encoding makes it particularly amenable to parallelization.

7.5   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 event space, such as the Hamming [18, 37] or Levenshtein [8, 9] balls, from which we draw inspiration for our work.

7.6   Neural program repair

The recent success of deep learning has lead to a variety of work on neural program repair [3, 14, 19]. These approaches typically employ Transformer-based neural language models (NLMs) and model the problem as a sequence-to-sequence transformation. Although recent work on circuit lower bounds have cast doubt on the ability of Transformers to truly learn formal languages [13, 27], expressivity aside, these models have been widely adopted for practical program repair tasks. In particular, two papers stand out being most closely related to our own: Break-It-Fix-It (BIFI) [40] and Seq2Parse [33]. BIFI adapts techniques from semi-supervised machine translation to generate synthetic errors in clean code and fixes them. This reduces the amount of pairwise training data, but may generalize poorly to natural errors. 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 with much lower latency, 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 a NLM model to rank its repairs.

8 EXPERIMENTAL SETUP

To evaluate our model, we primarily use 5,600 pairs of (broken, fixed) Python code snippets from Wong et al.'s StackOverflow dataset [39] shorter than 40 lexical tokens, whose minimized patch sizes are shorter than four lexical tokens ($|\Sigma| = 50, |\sigma| \leq 40, \Delta(\sigma, \ell) < 4$), and adapt the Python grammar from SeqParse. Minimization uses the Delta debugging [41] technique to isolate the smallest lexical patch that repairs a broken Python snippet.

In the first set of experiments, we uniformly sample without replacement from the Levenshtein edit ball using a LFSR over $\mathbb{Z}_2^m$, and measure the precision@k of our repair procedure against human repairs of varying edit distances and latency cutoffs. This provides a baseline for the relative density of the admissible set, and an upper bound on the latency of attaining a given precision.

In the second set of experiments, we use an adaptive sampler that stochastically resamples edits using a Dirichlet process. Repairs are scored and placed into a ranked buffer, from which new edits are resampled with frequency relative to their perplexity, and additive noise is introduced. The adaptive sampler is described in further detail in §4.9.

To train the scoring function, we use a length-5 variable-order Markov (VOM) chain implemented using a count-min sketch based on Apache Datasketches [5]. Training on 55 million StackOverflow tokens from the BIFI [40] dataset took roughly 10 minutes, after which calculating perplexity is nearly instantaneous. Sequences are scored using negative log likelihood with Laplace smoothing and our evaluation measures the precision@{1, 5, 10, All} for samples at varying latency cutoffs.

Both sets of experiments were conducted on 40 Intel Skylake cores running at 2.4 GHz, with 16 GB of RAM, running bytecode compiled for JVM 17.0.2. We measure the precision using abstract lexical matching, following the Seq2Parse [33] evaluation, and give a baseline for their approach on the same dataset.

8.1 Uniform sampling benchmark

Below, we plot the precision of the uniform sampling procedure described in §4.7 against human repairs of varying edit distances and latency cutoffs. Repairs discovered before the latency cutoff are reranked based on their tokenwise perplexity and compared for an exact lexical match with the human repair at or below rank k. We note that the uniform sampling procedure is not intended to be used in practice, but rather provides a baseline for the empirical density of the admissible set, and an upper bound on the latency required to attain a given precision.

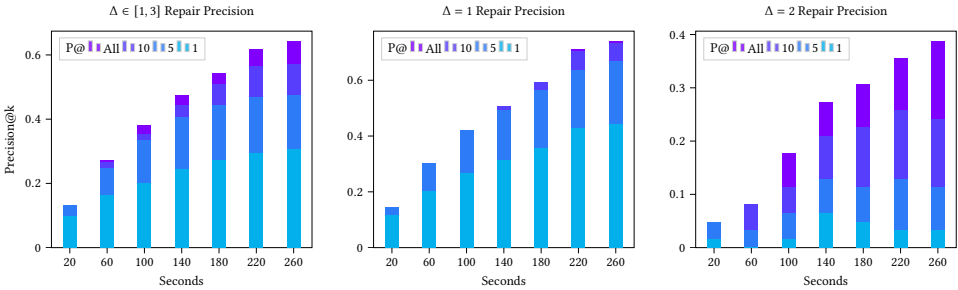Fig. 15. Human repair benchmark. Note the y-axis across different edit distance plots has varying ranges.

Despite the high-latency, this demonstrates a uniform prior with post-timeout reranking is still able to achieve competitive precision@k using a relatively cheap ranking metric. This suggests that we can use the metric to bias the sampler towards more likely repairs, which we will now do.

8.2 Repair with an adaptive sampler

In the following benchmark, we measure the precision@k of our repair procedure against human repairs of varying edit distances and latency cutoffs, using an adaptive resampling procedure described in §4.9. This sampler maintains a buffer of successful repairs ranked by perplexity and uses stochastic local search to resample edits within a neighborhood. Initially, edits are sampled uniformly at random. Over time and as the admissible set grows, it prioritizes edits nearby low-perplexity repairs. This technique offers a significant advantage in the low-latency setting.
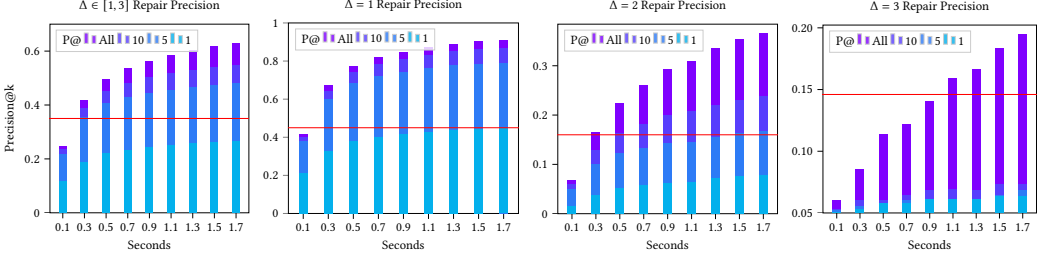

Fig. 16. Adaptive sampling repairs. The red line indicates Seq2Parse precision@1 on the same dataset. Since it only supports generating one repair, we do not report precision@k or the intermediate latency cutoffs.

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

While the sampler is able to saturate the admissible set for 1- and 2-edit repairs before the timeout elapses, 3-edit throughput is heavily constrained by compute around 16 lexical tokens, when Python's Levenshtein ball has a volume of roughly 6×10^8 edits. This bottleneck can be relaxed with a longer timeout or additional CPU cores. Despite the high computational cost of sampling multi-edit repairs, our precision@all remains competitive with the Seq2Parse neurosymbolic baseline at the same latency. We provide some qualitative examples of repairs in Table A.

8.3 Throughput benchmark

End-to-end throughput varies significantly with the edit distance of the repair. Some errors are trivial to fix, while others require a large number of edits to be sampled before a syntactically valid edit is discovered. We evaluate throughput by sampling edits across invalid strings $|\sigma| \leq 40$ from the StackOverflow dataset of varying length, and measure the total number of syntactically valid edits discovered, as a function of string length and language edit distance $\Delta \in [1, 3]$. 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 language edit distance. Our approach discovers a large number

of syntactically valid repairs in a relatively short amount of time, and is able to quickly saturate the admissible set for 1- and 2-edit repairs before timeout. As the Seq2Parse baseline is unable to generate more than one syntactically valid repair per string, we do not report its throughput.
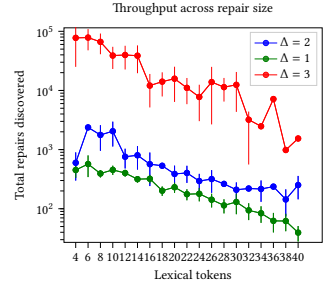
8.4 Synthetic repair benchmark

In addition to the StackOverflow dataset, we also evaluate our approach on two datasets containing synthetic strings generated by a Dyck language, and bracketing errors of synthetic and organic provenance in organic source code. The first dataset contains length-50 strings sampled from various Dyck languages, i.e., the Dyck language containing n different types of balanced parentheses. The second contains abstracted Java and Python source code mined from GitHub repositories. The Dyck languages used in the remaining experiments are defined by the following context-free grammar(s):

```
Dyck-1 -> ( ) | ( Dyck-1 ) | Dyck-1 Dyck-1
Dyck-2 -> Dyck-1 | [ ] | ( Dyck-2 ) | [ Dyck-2 ] | Dyck-2 Dyck-2
Dyck-3 -> Dyck-2 | { } | ( Dyck-3 ) | [ Dyck-3 ] | { Dyck-3 } | Dyck-3 Dyck-3
```

In experiment (1a), we sample a random valid string $\sigma \sim \Sigma^{50} \cap \mathcal{L}_{\text{Dyck-n}}$, then replace a fixed number of indices in $[0, |\sigma|)$ with holes and measure the average time required to decode ten syntactically-admissible repairs across 100 trial runs. In experiment (1b), we sample a random valid string as before, but delete p tokens at random and rather than provide their location(s), ask our model to solve for both the location(s) and repair by sampling uniformly from all n-token HCs, then measure the total time required to decode the first admissible repair. Note the logarithmic scale on the y-axis.
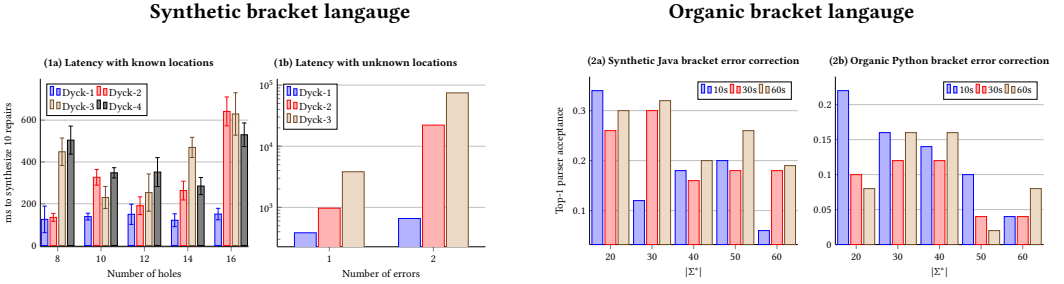

Fig. 17. Benchmarking bracket correction latency and accuracy across two bracketing langauges, one generated from Dyck-n, and the second uses an abstracted source code snippet with imbalanced parentheses.

In the second set of experiments, we analyze bracketing errors in a dataset of Java and Python code snippets mined from open-source repositories on GitHub using the Dyck-nw[4], in which all source code tokens except brackets are replaced with a w token. For Java (2a), we sample valid single-line statements with bracket nesting more than two levels deep, synthetically delete one bracket uniformly at random, and repair using Tidyparse, then take the top-1 repair after t seconds, and validate using ANTLR's Java 8 parser. For Python (2b), we sample invalid code fragments uniformly from the imbalanced bracket category of the Break-It-Fix-It (BIFI) dataset [40], a dataset of organic Python errors, which we repair using Tidyparse, take the top-1 repair after t seconds, and validate repairs using Python's `ast.parse()` method. Since the Java and Python datasets do not have a ground-truth human fix, we report the percentage of repairs that are accepted by the language's official parser for repairs generated under a fixed time cutoff. Although the Java and Python datasets are not directly comparable, we observe that Tidyparse can detect and repair a significant fraction of bracket errors in both languages with a relatively unsophisticated grammar.

---

[4]Using the Dyck-n grammar augmented with a single additional production, Dyck-1 → w | Dyck-1. Contiguous non-bracket characters are substituted with a single placeholder token, w, and restored verbatim after bracket repair.

9 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 extremely sample-efficient, their size comes at the cost of higher latency, and domain adaptation requires fine-tuning or retraining on pairwise repairs. Our approach uses a tiny grammar and a relatively cheap ranking metric to achieve comparable accuracy at the same latency. 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 SoTA precision@{5,10} is that repairs are typically concentrated near a small number of edit locations, and by biasing the search toward previously successful edit locations, we can achieve a significant speedup over a naïve search. We note this heuristic may not be applicable to all grammars, and it may be possible to construct less natural counterexamples where this heuristic fails, although we consider these unlikely in practice.

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

Although possible to further reduce constraint size by preprocessing techniques such as those posed in §4.5, the length of the string under repair is by far the dominating factor in formula size. For this reason, the isolation problem posed in Def. 4.9 is a critical obstacle to overcome. A general solution would allow us to collapse well-formed substrings and handle much larger code fragments than are currently supported. We consider this a promising direction for future work.

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

Secondly, our current method does not specialize language intersection to the grammar family, nor employ Bar-Hillel's [7] construction for REG-CFL intersection, which would lead to a more efficient encoding of the Levenshtein-CFL reachability. Furthermore, considering recent extensions of Boolean matrix-based parsing to linear context-free rewriting systems (LCFRS) [15], it may be feasible to search through richer language families within the SAT solver without employing an external stochastic search to generate and validate candidate repairs.

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

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

10 CONCLUSION

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

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

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

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

Over the last few years, there has been a surge of progress in applying language models to write programs. That work is primarily based on methods from differential calculus and continuous optimization, leading to the so-called *naturalness hypothesis*, which suggests programming languages are not so different from natural ones. In contrast, programming language theory takes the view that languages are essentially discrete and finitely-generated sets, although perhaps uncomputably large ones, governed by logical calculi. Programming, thus viewed, is more like a mathematical exercise in constraint satisfaction, an oft-overlooked but unavoidable aspect of translating abstract ideas into computing machinery. These constraints naturally arise at various stages of syntax validation, type-checking and runtime verification, and help to ensure programs fulfill their intended purpose.

As our work shows, not only is linear algebra over finite fields an expressive language for probabilistic inference, but also an efficient framework for inference on languages themselves. Borrowing analysis techniques from multilinear algebra and tensor completion in the machine learning setting, we develop an equational theory that allows us to translate various decision problems on formal languages into a system of inequalities over finite fields. As a practical consequence, this means we can efficiently encode a number of problems in parsing, code completion and program repair using SAT solvers, which are known to be highly efficient, scalable and flexible to domain-specific constraints. We demonstrate the effectiveness of our approach in a variety of practical scenarios, including code completion and program repair for linear conjunctive languages, and show that our approach is competitive with state-of-the-art methods in terms of both accuracy and efficiency. In contrast with LL and LR-style parsers, our technique can recover partial forests from invalid strings, and handle arbitrary conjunctive languages. In future work, we hope to extend our method to more natural grammars like PCFG, LCFRS and other mildly context-sensitive languages.

Syntax correction tools should be as user-friendly and widely-accessible as autocorrection tools in word processors. From a practical standpoint, we argue it is possible to reduce disruption from manual syntax repair and improve the efficiency of working programmers by driving down the latency needed to synthesize an acceptable repair. In contrast with program synthesizers that require

intermediate editor states to be well-formed, our synthesizer does not impose any constraints on the code itself being written and can be used in a live programming environment.

Despite its computational complexity, the design of the tool itself is relatively simple. Tidyparse accepts a linear conjunctive language and a string. If the string is valid, it returns the parse forest, otherwise, it returns a set of repairs, ordered by their perplexity. Tidyparse compiles the grammar and candidate string onto a discrete dynamical system using an extended version of Valiant's algorithm and solves for its fixedpoints using an incremental SAT solver. By allowing the string to contain holes, repairs may contain either concrete tokens or nonterminals, which can be expanded by the user or a neural-guided search procedure to produce a valid program. This approach to parsing has many advantages, enabling us to repair syntax errors, correct typos and recover from errors in real time, as well as being provably sound and complete with respect to the grammatical specification. It is also compatible with neural program synthesis and repair techniques and shares the same masked language modeling (MLM) target used by Transformer-based LLMs.

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

REFERENCES

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

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

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

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

[5] Apache Software Foundation. 2022. Apache DataSketches. https://datasketches.apache.org/.

[6] Roland Axelsson, Keijo Heljanko, and Martin Lange. 2008. Analyzing context-free grammars using an incremental SAT solver. In Automata, Languages and Programming: 35th International Colloquium, ICALP 2008, Reykjavik, Iceland, July 7-11, 2008, Proceedings, Part II 35. Springer, 410–422.

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

[8] 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

[9] 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).

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

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

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

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

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

[15] Shay B Cohen and Daniel Gildea. 2016. Parsing linear context-free rewriting systems with fast matrix multiplication. Computational Linguistics 42, 3 (2016), 421–455.

[16] Ryan Cotterell, Nanyun Peng, and Jason Eisner. 2014. Stochastic Contextual Edit Distance and Probabilistic FSTs. In Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics, Vol. 2 (Short Papers). Association for Computational Linguistics, Baltimore, Maryland, 625–630.

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

[21] Pavlos S Efraimidis. 2015. Weighted random sampling over data streams. Algorithms, Probability, Networks, and Games: Scientific Papers and Essays Dedicated to Paul G. Spirakis on the Occasion of His 60th Birthday (2015), 183–195.

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

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

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

[25] D. E. Knuth. 2005. Generating All Combinations and Partitions. Addison-Wesley, 5–6.

[26] 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

[27] 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.

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

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

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

[31] Perrin E. Ruth and Manuel E. Lladser. 2023. Levenshtein graphs: Resolvability, automorphisms determining sets. Discrete Mathematics 346, 5 (2023), 113310. https://doi.org/10.1016/j.disc.2022.113310

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

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

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

[35] Marcel H Schulz, David Weese, Tobias Rausch, Andreas Döring, Knut Reinert, and Martin Vingron. 2008. Fast and adaptive variable order Markov chain construction. In Algorithms in Bioinformatics: 8th International Workshop, WABI 2008, Karlsruhe, Germany, September 15-19, 2008. Proceedings 8. Springer, 306–317.

[36] Matthew Szudzik. 2006. An elegant pairing function. In Special NKS 2006 Wolfram Science Conference. 1–12.

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

[38] 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

[39] 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.

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

[41] Andreas Zeller. 2002. Isolating cause-effect chains from computer programs. ACM SIGSOFT Software Engineering Notes 27, 6 (2002), 1–10.

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

A EXAMPLE REPAIRS

We give some example human repairs that were correctly predicted from the StackOverflow dataset.

Original method	Human repair
```python form sympy import * x = Symbol('x', real=True) x, re(x), im(x) ```	```python from sympy import * x = Symbol('x', real=True) x, re(x), im(x) ```
```python result = yeald From(item.create()) raise Return(result) ```	```python result = yield From(item.create()) raise Return(result) ```
```python return 1/sum_p if sum_p \ return 0 else ```	```python return 1/sum_p if sum_p \ else 0 ```
```python from itertools improt permutations, product a='ABC' b=['*', '%3A'] l=[a]*2+[b] def g(list): for p in permutations(list): yield product(*p) result=g(l) ```	```python from itertools import permutations, product a='ABC' b=['*', '%3A'] l=[a]*2+[b] def g(list): for p in permutations(list): yield product(*p) result=g(l) ```
```python sum(len(v) for v items.values())) ```	```python sum(len(v) for v in items.values()) ```
```python df.apply(lambda row: list(set(row['ids')))) ```	```python df.apply(lambda row: list(set(row['ids']))) ```
```python import numpy ad np A_concate = np.array([a_0, a_1, a_2,..., a_n]) ```	```python import numpy as np A_concate = np.array([a_0, a_1, a_2,..., a_n]) ```