Syntax Error Correction as Idempotent Matrix Completion

ANONYMOUS AUTHOR(S)

In this work, we illustrate how to lower context-free language recognition onto a tensor algebra over finite fields. In addition to its theoretical value, this connection has yielded surprisingly useful applications in incremental parsing, code completion and program repair. For example, we use it to repair syntax errors, perform sketch-based program synthesis, and decide various language induction and membership queries. This line of research provides an elegant unification of context-free program repair, code completion and sketch-based program synthesis. To accelerate code completion, we design and implement a novel incremental parser-synthesizer that transforms CFGs onto a dynamical system over finite field arithmetic, enabling us to suggest syntax repairs in-between keystrokes.

1 INTRODUCTION

Modern research on error correction 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, whether due to collision with a high-energy proton, manipulation by an adversary or some 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 code which ensures that even if some portion of the message should be corrupted through accidental or intentional means, one can still recover the original message by solving a linear system of equations. In particular, we frame our work inside the context of errors arising from human factors in computer programming.

In programming, most such errors initially manifest as syntax errors, and though often cosmetic, manual repair can present a significant challenge for novice programmers. The ECC problem may be refined by introducing a language, $\mathcal{L} \subset \Sigma^*$ and considering admissible edits transforming an arbitrary string, $s \in \Sigma^*$ into a string, $s' \in \mathcal{L}$. Known as *error-correcting parsing* (ECP), this problem was well-studied in the early parsing literature by Aho and Peterson [1], but fell out of favor for many years, perhaps due to its perceived complexity. By considering only minimal-length edits, ECP can be reduced to the so-called *language edit distance* (LED) problem, recently shown to be subcubic [3], suggesting its tractability. Previous LED results were primarily of a theoretical nature, but now, thanks to our contributions, we have finally realized a practical prototype.

In our work, we recast the problem of ECP as a special case of tensor completion with an logical semiring, and lay the foundation for a new approach to program repair grounded in formal language theory, unifying *code completion*, *error correction* and *incremental parsing*. We provide exact and approximate algorithms for solving these problems in the real-world setting, and implement them in a real-time editor called Tidyparse, demonstrating their practical utility. Given a well-formed grammar, our tool can be used to complete unfinished code, parse incomplete code and repair broken fragments in arbitrary context-free and linear conjunctive languages.

2 TOY EXAMPLE

Suppose we are given the following context-free grammar:

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

SPLASH'23, October 22-27, 2023, Cascais, Portugal
2023. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

For reasons that will become clear in Sec. 3, this will automatically be rewritten into the grammar:

```
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 _ ( _ _ _ _ _ ! _ _ ) _ _ _ _
------------------------------------------------------------------------
true xor ! ( false xor ( <S> ) or ! <S> ) xor <S>
true xor ! ( false and ( <S> ) or ! <S> ) xor <S>
true xor ! ( false and ( <S> ) and ! <S> ) xor <S>
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
```

In the following paper, we will describe how we built it.

3 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 $w, x, z : V$ and $t : \Sigma$. For example:

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

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

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

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

$$M^0 := \begin{pmatrix} \varnothing & \sigma_1^{\rightarrow} & \varnothing & \cdots & \varnothing \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ & & & & \varnothing \\ & & & \ddots & \sigma_n^{\uparrow} \\ \varnothing & \cdots & & \cdots & \varnothing \end{pmatrix} \Rightarrow \begin{pmatrix} \varnothing & \sigma_1^{\rightarrow} & \Lambda & \cdots & \varnothing \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ & & & & \Lambda \\ & & & \ddots & \sigma_n^{\uparrow} \\ \varnothing & \cdots & & \cdots & \varnothing \end{pmatrix} \Rightarrow \ldots \Rightarrow M^* = \begin{pmatrix} \varnothing & \sigma_1^{\rightarrow} & \Lambda & \cdots & \Lambda_\sigma^* \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ & & & & \Lambda \\ & & & \ddots & \sigma_n^{\uparrow} \\ \varnothing & \cdots & & \cdots & \varnothing \end{pmatrix}$$

we obtain the recognizer, $R(\mathcal{G}', \sigma) := S \in \Lambda^*_\sigma? \Leftrightarrow \sigma \in \mathcal{L}(\mathcal{G})?$

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 [10], 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 [7], 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 lineararithmic under sparsity and believed to be the most efficient general procedure for CFL parsing.

Valiant's decision procedure can be abstracted by lifting into the domain of bitvector variables, i.e., linear equations over finite fields. This produces an algebraic expression for each nonterminal inhabitant of the northeasternmost bitvector $\mathcal{T}$, whose solutions correspond to valid parse forests for an incomplete string on the superdiagonal. This yields a novel interpretation of Valiant's algorithm as an equational theory over finite fields, allowing us to solve for admissible completions and their parse forests. In particular, $\boxplus$ and $\boxtimes$ are defined so the following diagram commutes,[1]

$$\begin{tikzcd}
\langle \mathcal{G}', \Sigma^{n-1} \rangle \arrow[dr, leftrightarrow] & \overset{\text{(Parsing)}}{\text{Set}} & \overset{\text{(Recognition)}}{\text{Bit}} & \langle \mathcal{G}', \Sigma^{n-1} \rangle \arrow[dr] & \overset{\text{(Synthesis)}}{\text{SAT}} \\
\text{Rubix} & M \times M \arrow[r, leftrightarrow, "\mathbb{1}^{2^{n\times n}}", "\mathbb{1}^{-2^{n\times n}}"'] \arrow[d, "\oplus \mid \otimes"] & \mathbb{Z}_2^{|V|^{n\times n}} \times \mathbb{Z}_2^{|V|^{n\times n}} \arrow[rr, leftrightarrow, "\varphi^{2^{n\times n}}", "\varphi^{-2^{n\times n}}"'] \arrow[d, "+ \mid *"] & & \mathcal{M} \times \mathcal{M} \arrow[d, leftrightarrow, "+ \mid *"] \\
\text{Matrix} & 2^V \times 2^V \arrow[r, leftrightarrow, "\mathbb{1}^2", "\mathbb{1}^{-2}"'] \arrow[d, "\oplus \mid \otimes"] & \mathbb{Z}_2^{|V|} \times \mathbb{Z}_2^{|V|} \arrow[rr, leftrightarrow, "\varphi^2", "\varphi^{-2}"'] \arrow[d, "\boxplus \mid \boxtimes"] & & \mathcal{V} \times \mathcal{V} \arrow[d, leftrightarrow, "\boxplus \mid \boxtimes"] \\
\text{Vector} & 2^V \arrow[r, leftrightarrow, "\mathbb{1}", "\mathbb{1}^{-1}"'] & \mathbb{Z}_2^{|V|} \arrow[rr, leftrightarrow, "\varphi", "\varphi^{-1}"'] & & \mathcal{V}
\end{tikzcd}$$

where $\mathcal{V}$ is a function $\mathbb{Z}_2^{|V|} \rightarrow \mathbb{Z}_2$. Note that while always possible to encode $\mathbb{Z}_2^{|V|} \rightarrow \mathcal{V}$ 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 and parse forest in the language.

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 operator will be indispensable when considering practical program repair scenarios, which may require extra-grammatical constraints such as indentation-sensitivity or Levenshtein-bounded reachability. That extension, and the resulting theory of breadth-bounded CJLs with holes, will be explored in Sec. 10.

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

3.1 From CFG parsing to SAT solving

$$\begin{tikzcd}
\text{String} & & \text{Grammar} \arrow[dddd, dash, dashed] & & \text{String} \\
\Sigma^n \arrow[dr] & \underline{\text{Parsing}} & \mathcal{G}_\varepsilon \arrow[dl] \arrow[dr] \arrow[ddl, bend right] \arrow[ddr, bend left] & \underline{\text{Repair}} & \Sigma^n \arrow[dl] \\
& V^n \arrow[d] & & (V_\varepsilon \cup \{\_\})^{n+q} \arrow[d, Rightarrow] & \\
& \mathbb{Z}_2^{n\times n\times |V|} \arrow[d, Rightarrow] & & \mathbb{Z}_2^{(n+q)^2\times|V_\varepsilon|} \arrow[r] \arrow[d, Rightarrow] & \mathbb{Z}_2^{|V_\varepsilon|} \\
& CST & & \left\langle (\Sigma \setminus \{\varepsilon\})^*, CST \right\rangle &
\end{tikzcd}$$

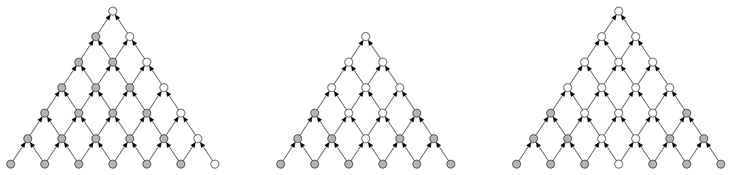
Our algorithm produces set of concrete syntax trees (CSTs) for a given valid string. Otherwise, if the string is invalid, the algorithm generates a set of admissible corrections, alongside their CSTs.

$$M^* = \begin{pmatrix}
\varnothing & \sigma_1^{\uparrow\rightarrow} & \mathcal{L}_{1,3} & \mathcal{L}_{1,3} & \mathcal{V}_{1,4} & \cdots & \mathcal{V}_{1,n} \\
\vdots & \ddots & \sigma_2^{\uparrow\rightarrow} & \mathcal{L}_{2,3} & \vdots & & \vdots \\
\vdots & & \ddots & \sigma_3^{\uparrow\rightarrow} & & & \vdots \\
\vdots & & & \ddots & \mathcal{V}_{4,4} & & \vdots \\
\vdots & & & & & \ddots & \mathcal{V}_{n,n} \\
\varnothing & \cdots & \cdots & \cdots & \cdots & \cdots & \varnothing
\end{pmatrix}$$

Depicted above is a SAT tensor representing $\sigma_1\ \sigma_2\ \sigma_3\ \_\ \ldots\ \_$ where shaded regions demarcate known bitvector literals $\mathcal{L}_{r,c}$ (i.e., representing established nonterminal forests) and unshaded regions correspond to bitvector variables $\mathcal{V}_{r,c}$ (i.e., representing seeded nonterminal forests to be grown). Since $\mathcal{L}_{r,c}$ are fixed, we precompute them outside the SAT solver.

3.2 Incrementalizing the parser

Should only need to recompute submatrices affected by individual edits. In the worst case, each edit requires quadratic complexity in terms of $|\Sigma^*|$, assuming $\mathcal{O}(1)$ cost for each CNF-nonterminal subset join, $\mathbf{V}_1' \otimes \mathbf{V}_2'$.



Append $\mathcal{O}(n+1)$ | Delete $\mathcal{O}\left(\frac{1}{4}(n-1)^2\right)$ | Insert $\mathcal{O}\left(\frac{1}{4}(n+1)^2\right)$

Related to **dynamic matrix inverse** and **incremental transitive closure** with vertex updates. With a careful encoding, we can incrementally update SAT constraints as new keystrokes are received to eliminate redundancy.

4 TREE DENORMALIZATION

Our parser emits a binary forest consisting of parse trees for the candidate string constructed bottom-up using $\hat{\otimes}$, which simply stores backpointers, however due to Chomsky normalization this forest is full of trees that are crooked and unnatural.

$$X \hat{\otimes} Z := \left\{ w \begin{smallmatrix} \nearrow x \\ \searrow z \end{smallmatrix} \mid \langle x, z \rangle \in X \times Z, (w \rightarrow xz) \in P \right\} \tag{2}$$

To restore the natural shape of the tree, we first construct the parse forests, then prune away synthetic nonterminals by recursively grafting denormalized grandchildren onto the root.

Algorithm 1 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(root, stems) }
    end if
end procedure
```

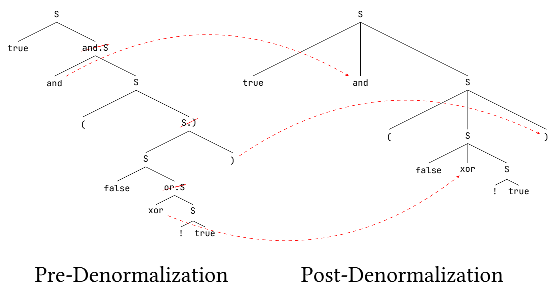


Fig. 1. Since $\mathcal{G}'$ contains synthetic nodes, to recover a parse tree congruent with the original grammar $\mathcal{G}$, we prune all synthetic nodes and graft their stems onto the grandparent via a simple recursive procedure (Alg. 1).

5 COARSENING AND SIMPLIFICATION OF SUBEXPRESSIONS

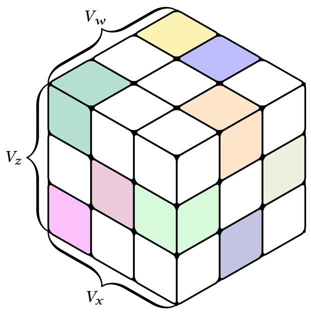
Clearly, the solving process is polynomial in the size of the string. For the sake of complexity, it would be convenient if well-formed subexpressions could be collapsed into a single nonterminal, or multiple nonterminals (in case of ambiguity). Naturally, this raises the question of when can partial derivations be simplified, i.e., under what circumstances is the following reduction admissible?

$$\mathbf{M}^* = \begin{pmatrix} \varnothing & \sigma_1^{\nrightarrow} & \mathcal{L}_{1,3} & \mathcal{L}_{1,3} & \mathcal{V}_{1,4} & \cdots & \mathcal{V}_{1,n} \\ \vdots & \ddots & \sigma_2^{\nrightarrow} & \mathcal{L}_{2,3} & \vdots & & \vdots \\ & & \ddots & \sigma_3^{\nrightarrow} & & & \\ & & & \ddots & \mathcal{V}_{4,4} & & \\ & & & & \ddots & \ddots & \\ \vdots & & & & & \ddots & \mathcal{V}_{n,n} \\ \varnothing & \cdots & & & & & \varnothing \end{pmatrix} \equiv \begin{pmatrix} \varnothing & \mathcal{L}_{1,3} & \mathcal{V}_{1,4} & \cdots & \mathcal{V}_{1,n} \\ \vdots & \ddots & \mathcal{V}_{2,4} & & \vdots \\ & & \ddots & \ddots & \\ \vdots & & & \ddots & \mathcal{V}_{n,n} \\ \varnothing & \cdots & & & \varnothing \end{pmatrix}$$

This transformation is admissible when the subexpression is "complete", i.e., its derivation cannot be altered by appending or prepending text. For example, the string `( - b )` is morally *complete* in the sense that inserting adjacent text should not alter the interior derivation, while `- b` is not, as introducing adjacent text (e.g. `a - b`) may alter the derivation of its contents depending on the structure of the CFG. This question can be reduced to a quotient: Does there exist another nonterminal, when so adjoined that will "strip away" any tokens, leading to another derivation? To resolve this question, we require the concept of a language derivative.

$$\begin{aligned} o &\to so \mid rs \mid rr \mid oo \\ r &\to so \mid ss \mid rr \mid os \\ s &\to so \mid rs \mid or \mid oo \end{aligned}$$

$$\mathcal{H}_{\{o\}} = \begin{pmatrix} \frac{\partial^2 o}{\partial\overleftarrow{o}\partial\overrightarrow{o}} & \frac{\partial^2 o}{\partial\overleftarrow{o}\partial\overrightarrow{r}} & \frac{\partial^2 o}{\partial\overleftarrow{o}\partial\overrightarrow{s}} \\ \frac{\partial^2 o}{\partial\overleftarrow{r}\partial\overrightarrow{o}} & \frac{\partial^2 o}{\partial\overleftarrow{r}\partial\overrightarrow{r}} & \frac{\partial^2 o}{\partial\overleftarrow{r}\partial\overrightarrow{s}} \\ \frac{\partial^2 o}{\partial\overleftarrow{s}\partial\overrightarrow{o}} & \frac{\partial^2 o}{\partial\overleftarrow{s}\partial\overrightarrow{r}} & \frac{\partial^2 o}{\partial\overleftarrow{s}\partial\overrightarrow{s}} \end{pmatrix}$$



$$\mathcal{H}_{\{r\}} = \begin{pmatrix} \frac{\partial^2 r}{\partial\overleftarrow{o}\partial\overrightarrow{o}} & \frac{\partial^2 r}{\partial\overleftarrow{o}\partial\overrightarrow{r}} & \frac{\partial^2 r}{\partial\overleftarrow{o}\partial\overrightarrow{s}} \\ \frac{\partial^2 r}{\partial\overleftarrow{r}\partial\overrightarrow{o}} & \frac{\partial^2 r}{\partial\overleftarrow{r}\partial\overrightarrow{r}} & \frac{\partial^2 r}{\partial\overleftarrow{r}\partial\overrightarrow{s}} \\ \frac{\partial^2 r}{\partial\overleftarrow{s}\partial\overrightarrow{o}} & \frac{\partial^2 r}{\partial\overleftarrow{s}\partial\overrightarrow{r}} & \frac{\partial^2 r}{\partial\overleftarrow{s}\partial\overrightarrow{s}} \end{pmatrix}$$

$$\mathcal{H}_{\{s\}} = \begin{pmatrix} \frac{\partial^2 s}{\partial\overleftarrow{o}\partial\overrightarrow{o}} & \frac{\partial^2 s}{\partial\overleftarrow{o}\partial\overrightarrow{r}} & \frac{\partial^2 s}{\partial\overleftarrow{o}\partial\overrightarrow{s}} \\ \frac{\partial^2 s}{\partial\overleftarrow{r}\partial\overrightarrow{o}} & \frac{\partial^2 s}{\partial\overleftarrow{r}\partial\overrightarrow{r}} & \frac{\partial^2 s}{\partial\overleftarrow{r}\partial\overrightarrow{s}} \\ \frac{\partial^2 s}{\partial\overleftarrow{s}\partial\overrightarrow{o}} & \frac{\partial^2 s}{\partial\overleftarrow{s}\partial\overrightarrow{r}} & \frac{\partial^2 s}{\partial\overleftarrow{s}\partial\overrightarrow{s}} \end{pmatrix}$$

Fig. 2. CFGs are witnessed by a rank-3 tensor, whose inhabitants indicate CNF productions. Gradients in this setting effectively condition the parse tensor M by constraining the superposition of admissible parse forests.

6 BACKPROPAGATION OF ERROR

Valiant's ⊗-operator, which yields the set of productions unifying known factors in a binary CFG, naturally implies the existence of a left- and right-quotient, which yield the set of nonterminals that may appear the right or left side of a known factor and its corresponding root. In other words, a known factor not only implicates subsequent expressions that can be derived from it, but also adjacent factors that may be composed with it to form a given derivation.

Valiant's ⊗-operator	Left Quotient	Right Quotient
$x \otimes z := \{\, w \mid (w \to xz) \in P \,\}$	$\frac{\partial w}{\partial \overleftarrow{x}} := \{\, z \mid (w \to xz) \in P \,\}$	$\frac{\partial w}{\partial \overrightarrow{z}} := \{\, x \mid (w \to xz) \in P \,\}$
$\begin{array}{\|c\|c\|}\hline x & w \\ \hline & z \\ \hline\end{array}$	$\begin{array}{\|c\|c\|}\hline x & w \\ \hline & z \\ \hline\end{array}$	$\begin{array}{\|c\|c\|}\hline x & w \\ \hline & z \\ \hline\end{array}$

The left quotient coincides with the derivative operator first proposed by Brzozowski [4] and Antimirov [2] over regular languages, lifted into the context-free setting (our work). When the root and LHS are fixed, e.g., $\frac{\partial S}{\partial \overleftarrow{x}} : (\overleftarrow{V} \to S) \to \overrightarrow{\mathcal{V}}$ returns the set of admissible nonterminals to the RHS. One may also consider a gradient operator, $\overleftarrow{\nabla} S : (\overleftarrow{\mathcal{V}} \to S) \to \overrightarrow{\mathcal{V}}$, which simultaneously tracks the partials with respect to a set of multiple LHS nonterminals produced by a fixed root.

If the root itself is unknown, we can define an operator, $\mathcal{H}_{\mathcal{W}\subseteq\mathcal{V}} : (\overleftarrow{\mathcal{V}} \times \overrightarrow{\mathcal{V}} \times \mathcal{W}) \to (\overleftarrow{\mathcal{V}} \times \overrightarrow{\mathcal{V}} \to \mathcal{W})$, which tracks second-order partial derivatives for all roots in $\mathcal{W}$. Unlike differential calculus on smooth manifolds, partials in this calculus do not necessarily commute depending on the CFG. By allowing the matrix $\mathcal{M}^*$ to contain bitvector variables representing holes in σ, we obtain a set of multilinear equations whose solutions exactly correspond to the set of admissible repairs and their corresponding parse forests. Specifically, the repairs coincide with holes in the superdiagonal $\mathcal{M}^*_{r+1=c}$, and the parse forests occur along upper-triangular entries $\mathcal{M}^*_{r+1<c}$.

6.1 Gradient estimation

Now that we have a reliable method to fix *localized* errors, $S : \mathcal{G} \times (\Sigma \cup \{\varepsilon, \_\})^n \rightarrow \{\Sigma^n\} \subseteq \mathcal{L}_{\mathcal{G}}$, given some unparseable string, i.e., $\underset{\sim}{\sigma_1 \ldots \sigma_n} : \Sigma^n \cap \mathcal{L}_{\mathcal{G}}^{\complement}$, where should we put holes to obtain a parseable $\sigma' \in \mathcal{L}_{\mathcal{G}}$? One way to do so is by sampling repairs, $\boldsymbol{\sigma} \sim \Sigma^{n \pm q} \cap \Delta_q(\underset{\sim}{\sigma})$ from the Levenshtein q-ball centered on $\underset{\sim}{\sigma}$, i.e., the space of all admissible edits with Levenshtein distance $\leq q$, loosely analogous to a finite difference approximation. To admit variable-length edits, we first add an ε^+-production to each unit production:

$$\frac{\mathcal{G} \vdash \varepsilon \in \Sigma}{\mathcal{G} \vdash (\varepsilon^+ \rightarrow \varepsilon \mid \varepsilon\, \varepsilon^+) \in P}\ \varepsilon\text{-DUP}$$

$$\frac{\mathcal{G} \vdash (A \rightarrow B) \in P}{\mathcal{G} \vdash (A \rightarrow B\, \varepsilon^+ \mid \varepsilon^+\, B \mid B) \in P}\ \varepsilon^+\text{-INT}$$

Next, suppose $U : \mathbb{Z}_2^{m \times m}$ is a matrix whose structure is shown in Eq. 3, wherein C is a primitive polynomial over $\mathbb{Z}_2^m$ with coefficients $C_{1\ldots m}$ and semiring operators $\oplus := \veebar, \otimes := \wedge$:

$$U^t V = \begin{pmatrix} C_1 & \cdots & \cdots & C_m \\ \top & \circ & \cdots & \circ \\ \circ & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \vdots \\ \circ & \cdots \circ & \top & \circ \end{pmatrix}^t \begin{pmatrix} V_1 \\ \vdots \\ \vdots \\ V_m \end{pmatrix} \tag{3}$$

Since C is primitive, the sequence $\mathbf{S} = (U^{0\ldots 2^m-1} V)$ must have *full periodicity*, i.e., for all $i, j \in [0, 2^m)$, $\mathbf{S}_i = \mathbf{S}_j \Rightarrow i = j$. To uniformly sample $\boldsymbol{\sigma}$ without replacement, we form an injection $\mathbb{Z}_2^m \rightharpoonup \left\{{n \atop d}\right\}^2 \times \Sigma_\varepsilon^{2d}$ using a combinatorial number system, cycle over $\mathbf{S}$, then discard samples which have no witness in $\left\{{n \atop d}\right\} \times \Sigma_\varepsilon^{2d}$. This requires $\widetilde{\mathcal{O}}(1)$ per sample and $\widetilde{\mathcal{O}}\left(\binom{n}{d} |\Sigma + 1|^{2d}\right)$ to exhaustively search $\left\{{n \atop d}\right\} \times \Sigma_\varepsilon^{2d}$.

Finally, to sample $\boldsymbol{\sigma} \sim \Delta_q(\underset{\sim}{\sigma})$, we enumerate templates $H(\sigma, i) = \sigma_{1\ldots i-1}\ \_\ \_\ \sigma_{i+1\ldots n}$ for each $i \in \cdot \in \left\{{n \atop d}\right\}$ and $d \in 1 \ldots q$, then solve for $\mathcal{M}_{\boldsymbol{\sigma}}^*$. If $S \in \Lambda_{\boldsymbol{\sigma}}^*$? has a solution, each edit in each $\sigma' \in \boldsymbol{\sigma}$ will match one of the following seven patterns:

$$\begin{aligned}
\text{Deletion} &= \left\{ \ldots \sigma_{i-1}\ \gamma_1\ \gamma_2\ \sigma_{i+1} \ldots \quad \gamma_{1,2} = \varepsilon \right. \\
\text{Substitution} &= \left\{ \begin{array}{ll} \ldots \sigma_{i-1}\ \gamma_1\ \gamma_2\ \sigma_{i+1} \ldots & \gamma_1 \neq \varepsilon \wedge \gamma_2 = \varepsilon \\ \ldots \sigma_{i-1}\ \gamma_1\ \gamma_2\ \sigma_{i+1} \ldots & \gamma_1 = \varepsilon \wedge \gamma_2 \neq \varepsilon \\ \ldots \sigma_{i-1}\ \gamma_1\ \gamma_2\ \sigma_{i+1} \ldots & \{\gamma_1, \gamma_2\} \cap \{\varepsilon, \sigma_i\} = \varnothing \end{array} \right. \\
\text{Insertion} &= \left\{ \begin{array}{ll} \ldots \sigma_{i-1}\ \gamma_1\ \gamma_2\ \sigma_{i+1} \ldots & \gamma_1 = \sigma_i \wedge \gamma_2 \notin \{\varepsilon, \sigma_i\} \\ \ldots \sigma_{i-1}\ \gamma_1\ \gamma_2\ \sigma_{i+1} \ldots & \gamma_1 \notin \{\varepsilon, \sigma_i\} \wedge \gamma_2 = \sigma_i \\ \ldots \sigma_{i-1}\ \gamma_1\ \gamma_2\ \sigma_{i+1} \ldots & \gamma_{1,2} = \sigma_i \end{array} \right.
\end{aligned}$$

This approach is tractable for $n \lesssim 100, q \lesssim 3$, however more complex repairs require a more efficient gradient estimator.

[2]Where $\left\{{n \atop d}\right\}$ is used to denote the set of all d-element subsets of $\{1, \ldots, n\}$.

7 ADAPTIVE SAMPLING

Since there are $\Sigma_{d=1}^{q}\binom{n}{d}$ total sketch templates, each with $(|\Sigma|+1)^{2d}$ individual edits to check, if n and q are large, this space can be intractable to exhaustively search and a uniform prior may be highly sample-inefficient. Furthermore, naïvely sampling $\boldsymbol{\sigma}_i \sim \Sigma^{n\pm q} \cap \Delta_q(\underset{\sim}{\sigma})$ is likely to produce unnatural edits. To provide rapid and relevant suggestions, we prioritize likely repairs according to the following six-step procedure:

(1) Retrieve the most recent $\mathcal{G}$ and $\underset{\sim}{\sigma}$ from the editor.
(2) Enumerate $i_{1\ldots d} \in \left\{\begin{smallmatrix} n \\ d \end{smallmatrix}\right\}$ for increasing values of $d \geq 1$.
(3) Rerank edit locations according to $\int \mathcal{F}_\theta(\cdot \mid i_{1\ldots d})d\theta$.
(4) Draw $\boldsymbol{\sigma}_i \sim \Sigma^{n\pm q} \cap \Delta_q(\underset{\sim}{\sigma})$ sans replacement using 6.1.
(5) Decode and rerank repairs by the cost model, $C(\underset{\sim}{\sigma}, \sigma')$.
(6) Display the top-k repairs in p-seconds to the user.

For example, given an erroneous string, $\underset{\sim}{\sigma} : \Sigma^{90}$, suppose we have $\mathcal{F}_\theta$, a distribution over possible edits provided by a probabilistic or neural language model, which can be used to localize admissible repairs. By marginalizing onto $\underset{\sim}{\sigma}$, the distribution $\mathcal{F}_\theta$ could take the following form:

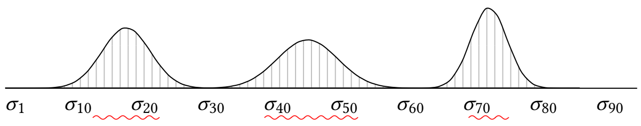


Fig. 3. The distribution $\int \mathcal{F}_\theta(\cdot \mid i_{1\ldots d})d\theta$, projected onto the invalid string, suggests edit locations most likely to yield admissible repairs, from which we draw subsets of size d.

Morally, we would prefer sketch 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 do so, we draw holes and rank admissible repairs using a distance metric over $\Delta_q(\underset{\sim}{\sigma})$. One such metric, the Kantorovich–Rubinstein (KR) metric, δ_{KR}, can be viewed as an optimal transport problem minimizing $\Pi(\mu, \nu)$, the set of all mass-conserving transportation plans between two probability distributions μ and ν over a metric space Ω:

$$\delta_{\mathrm{KR}}(\mu, \nu) := \inf_{\pi \in \Pi(\mu, \nu)} \int_{\Omega \times \Omega} \delta(x, y) d\pi(x, y) \tag{4}$$

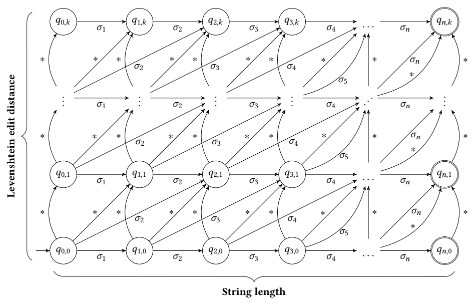
More specifically in our setting, Ω is a discrete product space that factorizes into (1) the specific edit locations (e.g., informed by caret position, historical edit locations, or a static analyzer), (2) probable completions (e.g., from a Markov chain or neural language model) and (3) an accompanying *cost model*, $C : (\Sigma^* \times \Sigma^*) \rightarrow \mathbb{R}$, which may be any number of suitable distance metrics, such as language edit distance, finger travel distance on a physical keyboard in the case of typo correction, weighted Levenshtein distance, or stochastic contextual edit distance [6] in the case of probabilistic edits. Our goal then, is to discover repairs which minimize $C(\underset{\sim}{\sigma}, \sigma')$, subject to the given grammar and latency constraints.

In the following section, we will give a more efficient construction for generating and accepting $\Delta_q(\underset{\sim}{\sigma})$ that does not require enumerating hole configurations.

8 LEVENSHTEIN REACHABILITY

Levenshtein reachability is recognized by the nondeterministic infinite automaton (NIA) whose topology $\mathcal{L}$ = can be factored into a product of (a) the monotone Chebyshev topology , 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 representable as an acyclic NFA [9], populated by accept states within radius k of $q_{n,0}$, or equivalently, a left-linear CFG whose productions finitely instantiate the transition dynamics:

NFA



CFG

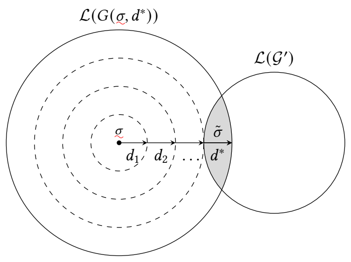
$$S \Rightarrow \{\cdot \in Q \mid \delta(\cdot, q_{n,0}) \leq k\}$$
$$* \Rightarrow \{\cdot \in \Sigma\}$$
$$\big\{q_{i,j} \Rightarrow \{q_{i,j-1}*\} \mid i,j \in [1,n] \times [1,k]\big\}$$
$$\big\{q_{i,j} \Rightarrow \{q_{i-1,j-1}*\} \mid i,j \in [1,n] \times [1,k]\big\}$$
$$\big\{q_{i,j} \Rightarrow \{q_{i-1,j}\sigma_i\} \mid i,j \in [1,n] \times [0,k]\big\}$$
$$\big\{q_{i,j} \Rightarrow \{q_{i-2,j-1}\sigma_i\} \mid i,j \in [2,n] \times [1,k]\big\}$$

Fig. 4. Levenshtein reachability from Σ^n can be described as either an acyclic ϵ-free NFA, or a left-linear CFG.

9 LANGUAGE EDIT REACHABILITY

Assume a hypothetical $\Phi(\mathcal{G}' : \mathbb{G}, \utilde{\sigma} : \Sigma^*) \mapsto \tilde{\sigma} : \mathcal{L}(\mathcal{G}')$ which takes a CFG, $\mathcal{G}'$, generating an arbitrary nonempty CFL, and an unparseable string, $\utilde{\sigma}$, and which returns element(s) of $\mathcal{L}(\mathcal{G}')$ most similar to $\utilde{\sigma}$ according to their Levenshtein distance $\Delta(\utilde{\sigma}, \cdot)$.

Let $G(\utilde{\sigma} : \Sigma^*, d : \mathbb{N}^+) \mapsto \mathbb{G}$ be the specific construction described in Sec. 8 which accepts a string, $\utilde{\sigma}$, and an edit distance, d, and returns a grammar representing the NFA that recognizes the language of all strings within Levenshtein radius d of $\utilde{\sigma}$. To find the language edit distance and corresponding least-distance edit, we must find the smallest d such that $\mathcal{L}_d^\cap$ is nonempty, where $\mathcal{L}_d^\cap$ is defined as $\mathcal{L}(G(\utilde{\sigma}, d)) \cap \mathcal{L}(\mathcal{G}')$. In other words, we seek $\tilde{\sigma}$ and d^* under which three criteria are equisatisfiable: (1) $\tilde{\sigma} \in \mathcal{L}(\mathcal{G}')$, and (2) $\Delta(\utilde{\sigma}, \tilde{\sigma}) \leq d^* \iff \tilde{\sigma} \in \mathcal{L}(G(\utilde{\sigma}, d^*))$, and (3) $\nexists \sigma' \in \mathcal{L}(\mathcal{G}').[\Delta(\utilde{\sigma}, \sigma') < d^*]$. To satisfy these criteria, it suffices to check $d \in (1, d_{\max}]$ by encoding the Levenshtein automata and the original grammar as a single SAT formula, call 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) $d_{\max}$ is attained. This procedure is guaranteed to terminate in at most either (1) the number of steps required to overwrite every symbol in $\utilde{\sigma}$, or (2) the length of the shortest string in $\mathcal{L}(\mathcal{G}')$, whichever is greater. More precisely:



$\mathcal{L}(G(\utilde{\sigma}, d^*))$
$\mathcal{L}(\mathcal{G}')$
$\utilde{\sigma}$
$\tilde{\sigma}$
d_1
d_2
$\ldots$
d^*

Fig. 5. LED is computed gradually by incrementing d until $\mathcal{L}_d^\cap \neq \varnothing$.

$$\varphi_{d+1}(\mathcal{G}', \utilde{\sigma}) := \begin{cases} \varphi[\tilde{\sigma} \in \mathcal{L}(G(\utilde{\sigma}, d)) \land \tilde{\sigma} \in \mathcal{L}(\mathcal{G}')] & \text{if } d = 1 \text{ or SAT.} \\ \varphi_d \oplus \bigoplus_{\{q \in Q \mid \delta(q, q_{n,0}) = d+1\}} \varphi[S \to q] & \text{if } d \leq \max(|\utilde{\sigma}|, \min_{\sigma \in \mathcal{L}(\mathcal{G}')} |\sigma|). \end{cases} \tag{5}$$

The function $\varphi_{d+1}(\mathcal{G}', \utilde{\sigma})$ is a realizer of Φ.

10 LINEAR CONJUNCTIVE REACHABILITY

It is well-known that the family of CFLs is not closed under intersection. For example, consider $\mathcal{L}_\cap := \mathcal{L}(\mathcal{G}_1) \cap \mathcal{L}(\mathcal{G}_2)$:

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

Note that $\mathcal{L}_\cap$ generates the language $\left\{\, a^d b^d c^d \mid d > 0 \,\right\}$, which according to the pumping lemma is not context-free. We can encode $\bigcap_{i=1}^{c} \mathcal{L}(\mathcal{G}_i)$ 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 $E(t_\cap, \mathcal{G}_i) = \{w_i \mid (w_i \rightarrow t_\cap) \in P_i\}$ and bind them together:

$$\bigwedge_{t \in \Sigma_\cap} \bigwedge_{j=1}^{c-1} \bigwedge_{i=1}^{|\sigma|} E(t_\cap, \mathcal{G}_j) \equiv_{\sigma_i} E(t_\cap, \mathcal{G}_{j+1}) \tag{6}$$

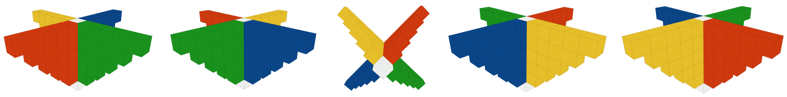


Fig. 6. Orientations of a $\bigcap_{i=1}^{4} \mathcal{L}(\mathcal{G}_i) \cap \Sigma^6$ configuration. As $c \rightarrow \infty$, this shape approximates a circular cone whose symmetric axis joins σ_i with orthonormal unit productions $w_i \rightarrow t_\cap$, and $S_i \in \Lambda_\sigma^*?$ represented by the outermost bitvector inhabitants. Equations of this form are equiexpressive with the family of CSLs realizable by finite CFL intersection.

It is a well-known fact in formal language theory that CFLs are closed under union, composition, substitution and intersection with regular languages, and the intersection between two CFLs is decidable. However CFLs are not closed under intersection, which requires a more general formalism. Following Okhotin [8], we define higher-order grammar combinators $\cup, \cap : \mathcal{G}^* \times \mathcal{G}^* \rightarrow \mathcal{G}^*$ where $\mathcal{G}^*$ is a conjunctive grammar, which are basically finite collections of CFGs. Unlike parser combinators which are susceptible to ambiguity errors, our grammar combinators return parse forests in case of syntactic ambiguity, and do not suffer from the same shortcomings.

Given two CFLs $\mathcal{L}_\mathcal{G}, \mathcal{L}_{\mathcal{G}'}$, we can compute the intersection $\mathcal{L}_\mathcal{G} \cap \mathcal{L}_{\mathcal{G}'} \cap \Sigma^d$ by encoding $(\mathbf{M}_\mathcal{G}^* \sigma) = (\mathbf{M}_{\mathcal{G}'}^* \sigma)$. This allows us to build a DSL of grammar combinators to constrain the solution space.

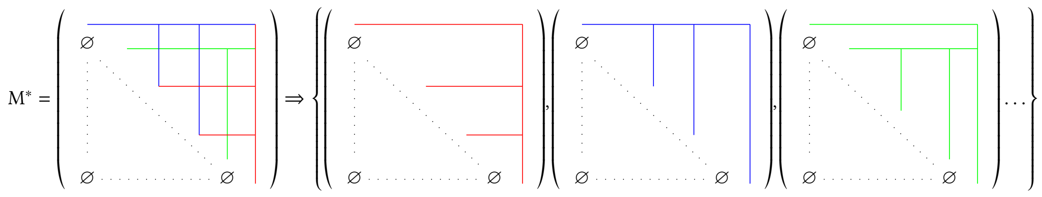
For example, we can solve $\Sigma^d \cap \overline{\mathcal{L}_\mathcal{G}}$ by enumerating $\{\beta\sigma'\gamma \mid \sigma' \in \Sigma^d, \beta = \gamma = \_^k\}$, overapproximating the prefix and suffix (padding left and right), and checking for UNSAT to underapproximate *impossible substrings*, strings which cannot appear in any $\{\sigma \in \mathcal{L}_\mathcal{G}\}$. Precomputing impossible substrings for a given grammar allows us to quickly eliminate inadmissible repairs and localize syntax errors in candidate strings.

Using the technique from Sec. 8, we can also compute language edit distance, the minimum number of Levenshtein edits required to fix a syntactically invalid string. Language intersection is significantly faster than approximating the gradient via sampling.

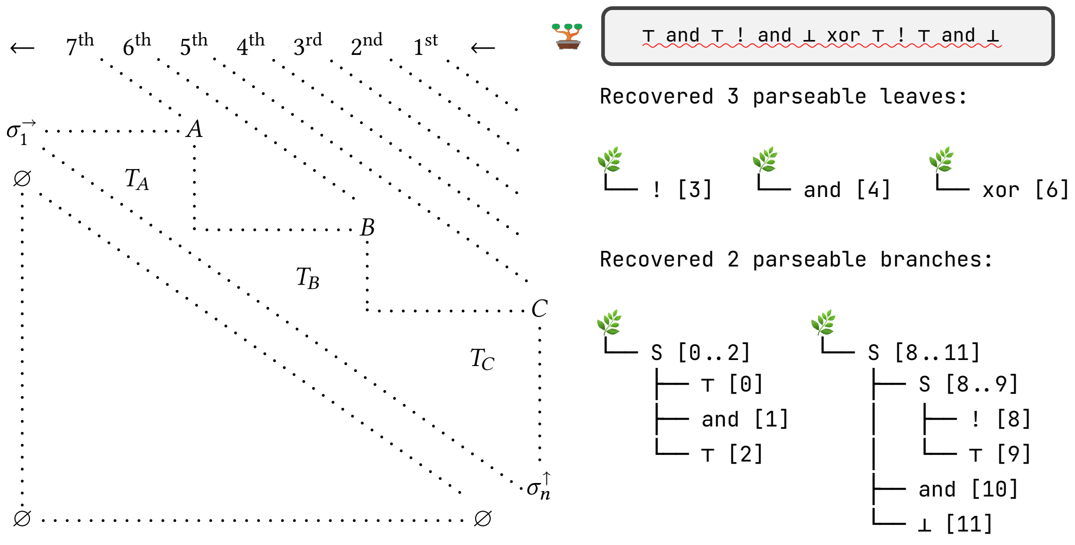
We can also build a set of grammars of increasing granularity, like a lattice structure. Basically, we can build up a lattice (in the order theoretic sense), consisting of grammars of increasing granularity. All programming languages require balanced parentheses, but some have additional constraints. So we can combine grammars, count and do bounded linear integer arithmetic.

11 ERROR RECOVERY

The matrix $\mathcal{M}^*$ encodes a superposition of all admissible binary trees of a fixed breadth. Consider the string _ ... _, which might generate various parse trees:



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 $\hat{\sigma}_{i,j}$.



```
⊤ and ⊤ ! and ⊥ xor ⊤ ! ⊤ and ⊥

Recovered 3 parseable leaves:

└── ! [3]    └── and [4]    └── xor [6]

Recovered 2 parseable branches:

└── S [0..2]        └── S [8..11]
    ├── T [0]           ├── S [8..9]
    ├── and [1]         │   ├── ! [8]
    └── T [2]           │   └── T [9]
                        ├── and [10]
                        └── ⊥ [11]
```

Fig. 7. Peaks along the UT matrix ridge correspond to maximally parseable substrings. By recursing over upper diagonals of decreasing elevation and discarding all subtrees that fall under the shadow of another's canopy, we can recover the partial subtrees. The example depicted above contains three such branches, rooted at nonterminals C, B, A.

These branches correspond to peaks on the upper triangular (UT) matrix ridge. As depicted in Fig. 7, we traverse the peaks by decreasing elevation to collect partial AST branches.

12 NONTERMINAL STUBS

As a notational conveinience, non-recursive nonterminal stubs are permitted inside the CFG definition. To support this feature, we introduce a simple substitution rule that replaces all productions containing parameterized nonterminals, $\langle\alpha\rangle$, with the terminals in their transitive closure, $\alpha \to^* \beta$:

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

Some dependently typed programming languages can do parsing in the type checker. Our parser can do some type checking too. Tidyparse enables adding a placeholder for typed expressions, which will be expanded into ordinary nonterminals using the α-SUB rule:

```
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>
```

When completing a bounded-width string, one finds it is often convenient to admit nonterminal stubs, representing unexpanded subexpressions. To enable this functionality, we introduce a synthetic production for each $v \in V$ using the $\langle\cdot\rangle$-INT rule. Users can interactively build up a complex expression by placing the caret over a placeholder they wish to expand, then invoking Tidyparse by pressing ctrl + Space:

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

This functionality can also be used inside a completion:

```
if <Vexp> _ _ _ _ _
---------------------------------------------------------------------
1.) if map X then <Vexp> else <Vexp>
2.) if uncurry X then <Vexp> else <Vexp>
3.) if foldright X then <Vexp> else <Vexp>
...
```

13 PRACTICAL EXAMPLE

Tidyparse requires a grammar – this can be either provided by the user or ingested from a BNF-like specification. The following is a slightly more complex grammar, designed to resemble a more realistic use case:

```
S -> A | V | ( X , X ) | X X | ( X )
A -> Fun | F | L | L in X
Fun -> fun V `->` X
F -> if X then X else X
L -> let V = X | let rec V = X
V -> Vexp | ( Vexp ) | Vexp Vexp
Vexp -> VarName | FunName | Vexp VO Vexp | ( VarName , VarName ) | Vexp Vexp
VarName -> a | b | c | d | e | ... | z
FunName -> foldright | map | filter
VO ->  + | - | * | / | > | = | < | `||` | &&
---
let curry f = ( fun x y -> f ( _ _ _) )
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
let curry f = ( fun x y -> f ( <X> ) )
let curry f = ( fun x y -> f ( <FunName> ) )
let curry f = ( fun x y -> f ( curry <X> ) )
...
```

We can also handle the untyped λ-calculus, as shown below:

```
sxp -> λ var . sxp | sxp sxp | var | ( sxp ) | const
const -> 1 | 2 | 3 | 4 | 5 | 6
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 ) ) )
...
```

13.1 Grammar Assistance

Tidyparse uses a CFG to parse the CFG, so it can provide assistance while the user is designing the CFG. For example, if the CFG does not parse, it will suggest possible fixes. In the future, we intend to use this functionality to perform example-based codesign and grammar induction.

```
B -> true | false |
-----------------------------------------------------------------
B -> true | false
B -> true | false <RHS>
B -> true | false | <RHS>
...
```

13.2 Interactive Nonterminal Expansion

Users can interactively build up a complex expression by placing the caret over a placeholder they wish to expand, then invoking Tidyparse by pressing ctrl + Space:

```
if <Vexp> X then <Vexp> else <Vexp>
-----------------------------------------------------------------
if map X then <Vexp> else <Vexp>
if uncurry X then <Vexp> else <Vexp>
if foldright X then <Vexp> else <Vexp>
...
```

13.3 Conjunctive Grammars

Many natural and programming languages exhibit context-sensitivity, such as Python indentation. Unlike traditional parser-generators, Tidyparse can encode CFL-intersection, allowing it to detect and correct errors in a broader family of languages than would normally be possible using CFGs alone. For example, consider the grammar from Sec. 10:

```
S -> L R    L -> a b | a L b    R -> c | c R
&&&
S -> L R    R -> b c | b R c    L -> a | a L
---
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
-----------------------------------------------------------------
1.) a b c
2.) a a b b c c
3.) a a a b b b c c c
4.) a a a a b b b b c c c c
...
```

Tidyparse uses the notation `G1 &&& G2` and `G1 ||| G2` to signify $\mathcal{L}_{\mathcal{G}_1} \cap \mathcal{L}_{\mathcal{G}_2}$ and $\mathcal{L}_{\mathcal{G}_1} \cup \mathcal{L}_{\mathcal{G}_2}$ respectively, i.e., the intersection or union of two or more grammars' languages. Composition, complementation and other operations on finite languages are also possible, although undocumented at present.

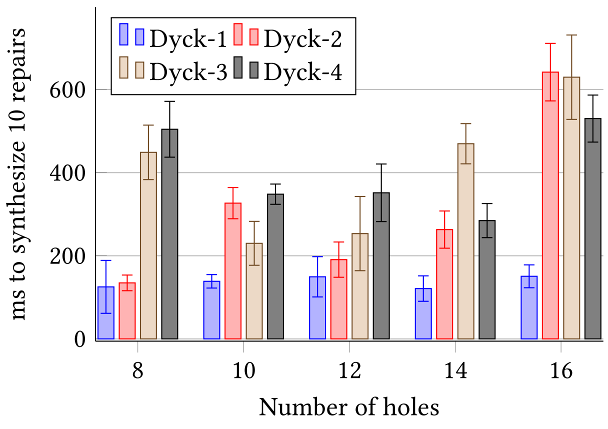
14 LATENCY BENCHMARK

In the following benchmarks, we measure the wall clock time required to synthesize solutions to length-50 strings sampled from various Dyck languages, where Dyck-n is the Dyck language containing n types of balanced parentheses.

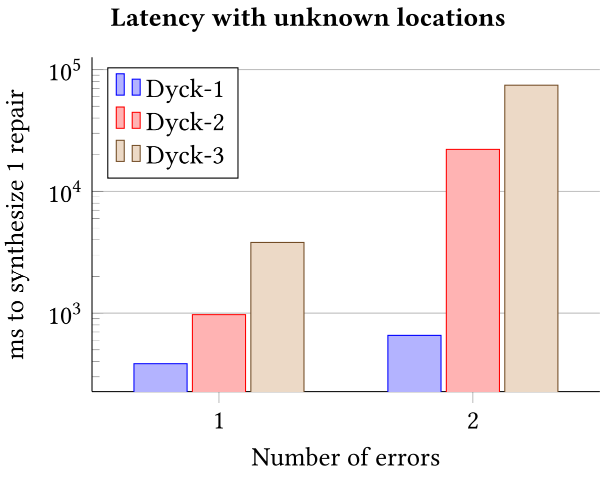
```
Dyck-1 -> ( ) | ( Dyck-1 ) | Dyck-1 Dyck-1
Dyck-2 -> Dyck-1 | [ ] | ( Dyck-2 ) | [ Dyck-2 ] | Dyck-2 Dyck-2
Dyck-3 -> Dyck-2 | { } | ( Dyck-3 ) | [ Dyck-3 ] | { Dyck-3 } | Dyck-3 Dyck-3
```

In the first experiment, we sample a random valid string $\sigma \sim \Sigma^{50} \cap \mathcal{L}_{\text{Dyck-n}}$, then replace a fixed number tokens with holes and measure the average time taken to decode ten syntactically-admissible repairs across 100 trial runs.

Latency with known locations

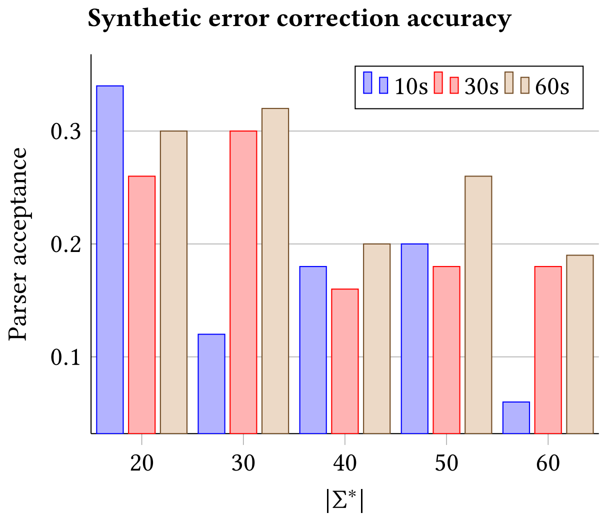


In the second experiment, we sample a random valid string as before, but delete p tokens at random and rather than provide the location(s), ask our model to solve for both the location(s) and repair by sampling uniformly from all n-token HCs, then measure the total time required to decode the first admissible repair. Note the the logarithmic scale on the y-axis.



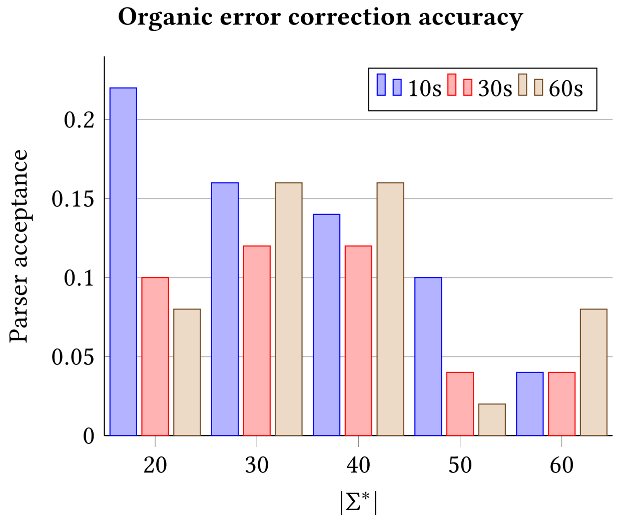
15 ACCURACY BENCHMARK

In the following benchmark, we analyze bracketing errors in a dataset of Java and Python code snippets mined from open-source repositories on GitHub. For Java, we sample valid single-line statements with bracket nesting more than two levels deep, synthetically delete one bracket uniformly at random, repair using Tidyparse[2], then take the top-1 repair after t seconds, and validate using ANTLR's Java 8 parser.



For Python, we sample invalid code fragments uniformly from the imbalanced bracket category of the Break-It-Fix-It (BIFI) dataset [11], a dataset of organic Python errors, repair using Tidyparse??[0], take the top-1 repair after t seconds, and validate repairs using Python's `ast.parse()` method.

[2]Using the Dyck-n grammar augmented with `D1` $\rightarrow$ `w` | `D1`. Contiguous non-bracket tokens are substituted with a single placeholder token, `w`, and restored verbatim after bracket repair.



16 DISCUSSION

While error correction with a few errors is tolerable, latency can vary depending on many factors including string length and grammar size. If errors are localized to the beginning or end of a string, then latency is typically below 500ms. We observe that errors are typically concentrated nearby historical edit locations, which can be retrieved from the IDE or version control.

Tidyparse in its current form has a number of technical shortcomings: firstly it does not incorporate any neural language modeling technology at present, an omission we hope to address in the near future. Training a language model to predict likely repair locations and rank admissible results could lead to lower overall latency and more natural repairs.

Secondly, our current method generates sketch templates using a naïve enumerative search, feeding them individually to the SAT solver, which has the tendency to duplicate prior work and introduces unnecessary thrashing. Considering recent extensions of Boolean matrix-based parsing to linear context-free rewriting systems (LCFRS) [5], it may be feasible to search through these edits within the SAT solver, leading to yet unrealized and possibly significant speedups.

Lastly and perhaps most significantly, Tidyparse does not incorporate any semantic constraints, so its repairs while syntactically admissible, are not guaranteed to be semantically valid. We note however, that it is possible to encode type-based semantic constraints into the solver and intend to explore this direction more fully in future work.

Not only is linear algebra over finite fields an expressive language for inference, but also an efficient framework for inference on languages themselves. We illustrate a few of its applications for parsing incomplete strings and repairing syntax errors in context- free and sensitive languages. In contrast with LL and LR-style parsers, our technique can recover partial forests from invalid strings by examining the structure of M^* and handles arbitrary context-free langauges. In future work, we hope to extend our method to more natural grammars like PCFG and LCFRS.

We envision three primary use cases: (1) helping novice programmers become more quickly familiar with a new programming language (2) autocorrecting common typos among proficient but forgetful programmers and (3) as a prototyping tool for PL designers and educators. 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 testing.

17 CONCLUSION

Tidyparse accepts a CFG and a string to parse. If the string is valid, it returns the parse forest, otherwise, it returns a set of repairs, ordered by their Levenshtein edit distance to the invalid string. Our method compiles each CFG and candidate string onto a matrix dynamical system using an extended version of Valiant's construction and solves for its fixedpoints using an incremental SAT solver. This approach to parsing has many advantages, enabling us to repair syntax errors, correct typos and generate parse trees for incomplete strings. By allowing the string to contain holes, repairs can contain either concrete tokens or nonterminals, which can be manually expanded by the user or a neural-guided search procedure. From a theoretical standpoint, this technique is particularly amenable to neural program synthesis and repair, naturally integrating with the masked-language-modeling task (MLM) used by transformer-based neural language models.

From a practical standpoint, we have implemented our approach as an IDE plugin and demonstrated its viability as a tool for live programming. Tidyparse is capable of generating repairs for invalid code in a range of toy languages. 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 in the near future.

REFERENCES

[1] 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.

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

[3] Karl Bringmann, Fabrizio Grandoni, Barna Saha, and Virginia Vassilevska Williams. 2019. Truly subcubic algorithms for language edit distance and RNA folding via fast bounded-difference min-plus product. SIAM J. Comput. 48, 2 (2019), 481–512.

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

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

[6] 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.

[7] 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

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

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

[10] 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

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

A EXAMPLE REPAIRS

1.a) Original method	1.b) Synonymous variant
`public void flush(int b) { buffer.write((byte) b); buffer.compact(); }`	`public void flush(int b) { cushion.write((byte) b); cushion.compact(); }`
2.a) Multi-masked method	**2.b) Multi-masked variant**
`public void <MASK>(int b) { buffer.<MASK>((byte) b); <MASK>.compact(); }`	`public void <MASK>(int b) { cushion.<MASK>((byte) b); <MASK>.compact(); }`
3.a) Model predictions	**3.b) Model predictions**
`public void output(int b) { buffer.write((byte) b); buffer.compact(); }`	`public void append(int b) { cushion.add((byte) b); cushion.compact(); }`