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, cf. Aho and Peterson [?], 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 [?], suggesting its possible tractability. Previous results on ECP and LED were primarily of a theoretical nature, but now, thanks to our contributions, we have finally realized a practical prototype.

2 TOY EXAMPLE

Suppose we are given the following context-free grammar:

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

For reasons that will become clear in the following section, this is automatically rewritten into the equivalent grammar:

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

```
F.or → or   S → S ε+    S → F.( S.)
```

Given a string containing holes such as the one below, Tidyparse will return several completions in a few milliseconds:

```
true _ _ _ ( false _ ( _ _ _ _ ! _ _ ) _ _ _ _█
```

```
true or ! ( false or ( <S> ) or ! <S> ) or <S>
true or ! ( false and ( <S> ) or ! <S> ) or <S>
true or ! ( false and ( <S> ) and ! <S> ) or <S>
true or ! ( false and ( <S> ) and ! <S> ) and <S>
...
```

Similarly, if provided with a string containing various errors, Tidyparse 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 ($P: V \rightarrow (V \mid \Sigma)^*$), and a start symbol, ($S$). It is a well-known fact that every CFG is reducible to *Chomsky Normal Form*, $P': V \rightarrow (V^2 \mid \Sigma)$, in which every production takes one of two forms, either $w \rightarrow xz$, or $w \rightarrow t$, where $w, x, z : V$ and $t : \Sigma$. For example, the CFG, $P := \{S \rightarrow SS \mid (S) \mid ()\}$, corresponds to the CNF:

$$P' = \big\{\, S \rightarrow QR \mid SS \mid LR, \quad L \rightarrow (, \quad R \rightarrow), \quad Q \rightarrow LS \,\big\}$$

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

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

If we define $\sigma_r^{\not\rightarrow} := \{w \mid (w \rightarrow \sigma_r) \in P\}$, then initialize $M_{r+1=c}^0(\mathcal{G}', e) := \sigma_r^{\not\rightarrow}$ and solve for the fixpoint $M^* = M + M^2$,

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

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

This decision procedure can be abstracted by lifted into the domain of bitvector variables, producing an algebraic expression for each scalar inhabitant of the northeasternmost bitvector $\mathcal{T}$, whose solutions correspond to valid parse forests for an incomplete string on the superdiagonal. Note that $\bigoplus_{c=1}^{n} M_{r,c} \otimes M_{c,r}$ has cardinality bounded by $|V|$ and is thus representable as a fixed-length vector using the characteristic function, $\mathbb{1}$. In particular, $\oplus, \otimes$ are redefined as $\boxplus, \boxtimes$ over bitvectors so the following diagram commutes,[1]
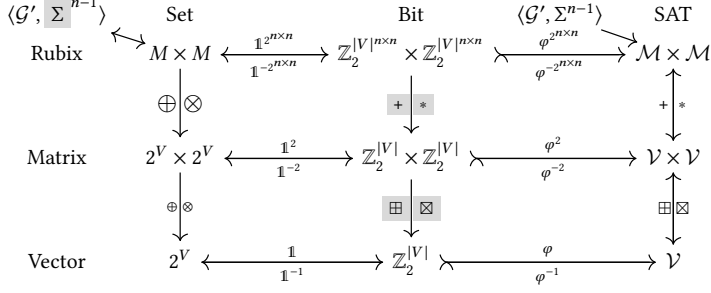
where $\mathcal{V}$ is a function $\mathbb{Z}_2^{|V|} \to \mathbb{Z}_2$. Note that while always possible to encode $\mathbb{Z}_2^{|V|} \to \mathcal{V}$ using the identity function, φ^{-1} may not exist, as an arbitrary $\mathcal{V}$ might have zero, one, or in general, multiple solutions in $\mathbb{Z}_2^{|V|}$.

Full details of the bisimilarity between parsing and matrix multiplication can be found in Valiant [?], who shows its time complexity to be $\mathcal{O}(n^\omega)$ where ω is the matrix multiplication bound ($\omega < 2.77$), and Lee [?], who shows that speedups to Boolean matrix multiplication are realizable by CFL parsers. Assuming sparsity, this technique is typically linearithmic, and is believed to be the most efficient general procedure for CFL recognition.

3.1 Encoding CFG parsing as SAT solving

By allowing the matrix $\mathbf{M}^*$ in Eq. ?? to contain bitvector variables representing holes in the string and nonterminal sets, we obtain a set of multilinear SAT 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 $\mathbf{M}^*_{r+1=c}$, and the parse forests occur along the upper-triangular entries $\mathbf{M}^*_{r+1<c}$.

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

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

---

[1]Hereinafter, we use gray highlighting to distinguish between expressions containing only constants from those which may contain free variables.

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.
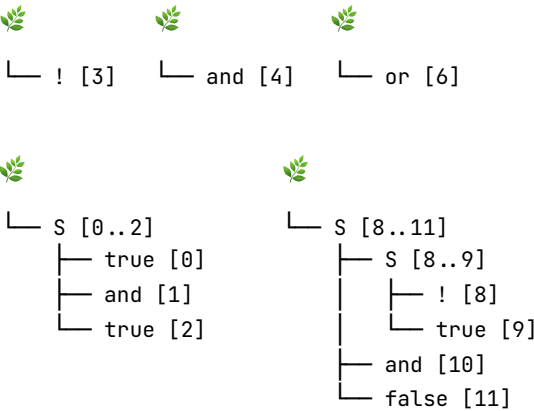
4 ERROR RECOVERY

Unlike classical parsers which need special care to recover from errors, if the input string does not parse, Tidyparse can return partial subtrees. If no solution exists, the upper triangular entries will appear as a jagged-shaped ridge whose peaks represent the roots of parsable ASTs. These provide a natural debugging environment to aid the repair process.

The matrix $\mathcal{M}^*$ encodes a superposition of all labeled binary trees of a fixed breadth. Consider the string _ ... _, which might generate various parse trees, with labels omitted to emphasize their structure:

true and true ! and false or true ! true and false

Parseable subtrees (3 leaves / 2 branches):


```
🌿                🌿                🌿
└── ! [3]        └── and [4]      └── or [6]
```

```
🌿                            🌿
└── S [0..2]                 └── S [8..11]
    ├── true [0]                 ├── S [8..9]
    ├── and [1]                  │   ├── ! [8]
    └── true [2]                 │   └── true [9]
                                 ├── and [10]
                                 └── false [11]
```

These branches correspond to peaks on the upper triangular (UT) matrix ridge. As depicted in Fig. 1, we traverse the peaks by decreasing elevation to collect partial AST branches.
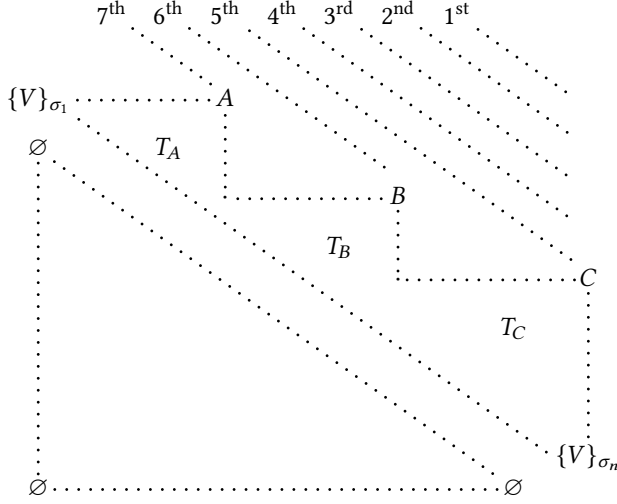
Fig. 1. Peaks along the UT matrix ridge correspond to maximally parseable substrings. By recursing over upper diagonals of decreasing elevation and discarding all subtrees that fall under the shadow of another's canopy, we can recover the partial subtrees. The example depicted above contains three such branches, rooted at nonterminals C, B, A.

5 TREE DENORMALIZATION

Our parser emits a binary forest consisting of parse trees for the candidate string according to the CNF grammar, however this forest contains many so-called *Krummholz*, or *flag trees*, often found clinging to windy ridges and mountainsides.

Pre-Denormalization   Post-Denormalization
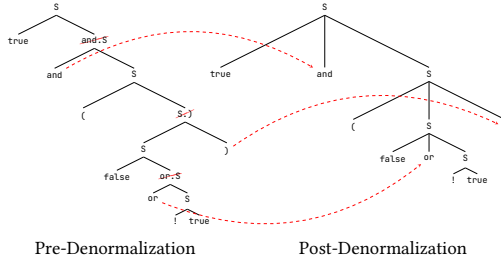
Algorithm 1 Rewrite procedure for tree denormalization

---

procedure DENORMALIZE(t: Tree)
    stems ← { DENORMALIZE(c) | c ∈ t.children }
    if t.root ∈ $V_{G'} \setminus V_G$ **then**
        return stems   ▷ Drop synthetic nonterminals.
    else   ▷ Graft the denormalized children on root.
        return { Tree(root, stems) }
    end if
end procedure

$$o \rightarrow so \mid rs \mid rr \mid oo$$
$$r \rightarrow so \mid ss \mid rr \mid os$$
$$s \rightarrow so \mid rs \mid or \mid oo$$

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

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

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

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

To recover a parse tree congruent with the user-specified grammar, we prune all synthetic nodes and graft their stems onto the grandparent via a simple recursive procedure (Alg. 1).

6 BACKPROPAGATION OF ERROR

Valiant's $\otimes$-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 $\otimes$-operator	Left Quotient	Right Quotient
$x \otimes y := \{ w \mid (w \rightarrow xz) \in P \}$	$\frac{\partial}{\partial \bar{x}} := \{ z \mid (w \rightarrow xz) \in P \}$	$\frac{\partial}{\partial \bar{z}} := \{ x \mid (w \rightarrow xz) \in P \}$

The left quotient coincides with the derivative operator first proposed by Brzozowski [?] and Antimirov [?] over regular languages, lifted into the context-free setting (our work). When the root and LHS are fixed, e.g., $\frac{\partial S}{\partial \bar{x}} : (\bar{V} \rightarrow S) \rightarrow \vec{V}$ returns the set of admissible nonterminals to the RHS. One may also consider a gradient operator, $\bar{\nabla} S : (\bar{V} \rightarrow S) \rightarrow \vec{V}$, 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, $\mathcal{H}_{\mathcal{W} \subseteq \mathcal{V}} : (\vec{V} \times \vec{V} \times \mathcal{W}) \rightarrow (\vec{V} \times \vec{V} \rightarrow \mathcal{W})$, 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.

7 STRING REPAIR

Given some unparseable string, i.e., $\underline{\sigma_1 \dots \sigma_n} : \overline{\Sigma}^n \cap \mathcal{L}(\mathcal{G})^c$, where should we put holes to obtain a parseable $\sigma' \in \mathcal{L}(\mathcal{G})$? To estimate the effect of perturbing σ on Λ^*_σ, one can either (1) backpropagate ∇S across upper-triangular entries of $\mathcal{M}^*$, or (2) stochastically sample *minibatches* $\boldsymbol{\sigma} : \Sigma^{n \pm q} \sim \Delta_q(\sigma)$ from the Levenshtein q-ball centered on σ, i.e., the space of all edits with Levenshtein distance $\leq q$, loosely analogous to a finite difference approximation. In other words, we seek:

$$\bigcup_{\boldsymbol{\sigma} \sim \Delta_k(\sigma)} \{\sigma' \in \boldsymbol{\sigma} \mid S \in \Lambda^*_{\sigma'}\} \tag{2}$$

Let us consider (2). 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 $C_{1 \dots m}$ and semiring operators $\oplus := \vee, \otimes := \wedge$:

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

Since C is primitive, the sequence $\mathbf{S} = (U^{0 \dots 2^m - 1} V)$ 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 $\boldsymbol{\sigma} \sim \Sigma^d$ without replacement, we form an injection $\mathbb{Z}_2^m \rightarrow \Sigma^d$, cycle over $\mathbf{S}$, then discard samples which index any nonexistent element(s) of Σ. This method will reject $(1 - |\Sigma| 2^{-\lceil \log_2 |\Sigma| \rceil})^d$ samples overall, and requires $\mathcal{O}(1)$ per sample and $\mathcal{O}(|\Sigma|^d)$ to cover Σ^d. For example, in order to sample $\sigma \sim \Sigma^2 = \{A, B, C\}^2$, we could use the polynomial $x^4 + x^3 + 1$:

i	0	1	2	3	4	5	6	7
$\mathbf{S}_i$	1000	0100	0010	1001	1100	0110	1011	0101
σ	C A	B A	A C	C B		B C		B B

Next, to admit deletion, we augment P with $(\varepsilon^+ \rightarrow \varepsilon \mid \varepsilon^+ \varepsilon^+)$ and replace each production $(w \rightarrow t)$ with $(w \rightarrow t \varepsilon^+ \mid \varepsilon^+ t \mid t)$.

$$\frac{\Gamma \vdash \varepsilon \in \Sigma}{\Gamma \vdash (\varepsilon^+ \rightarrow \varepsilon \mid \varepsilon^+ \varepsilon^+) \in P} \; \varepsilon\text{-DUP} \qquad \frac{\Gamma \vdash (A \rightarrow B) \in P}{\Gamma \vdash (A \rightarrow B \varepsilon^+ \mid \varepsilon^+ B \mid B) \in P} \; \varepsilon^+\text{-INT}$$

Finally, to generate $\boldsymbol{\sigma} \sim \Delta_q(\sigma)$, we enumerate hole configurations $H(\sigma, i) = \sigma_{1 \dots i-1} \_ \_ \sigma_{i+1 \dots n}$ for each $i \in \binom{n}{d}$ and $d \in 1 \dots q$, then solve for $\mathcal{M}^*$. If $S \in \Lambda^*_\sigma$? has at least one solution, each edit in each $\sigma' \in \boldsymbol{\sigma}$ will match exactly one of seven patterns:

$$\text{Deletion} = \left\{ \sigma_1 \dots \gamma_1 \gamma_2 \dots \sigma_n \;\; \gamma_{1,2} = \varepsilon \right.$$

$$\text{Substitution} = \begin{cases} \sigma_1 \dots \gamma_1 \gamma_2 \dots \sigma_n & \gamma_1 \neq \varepsilon \wedge \gamma_2 = \varepsilon \\ \sigma_1 \dots \gamma_1 \gamma_2 \dots \sigma_n & \gamma_1 = \varepsilon \wedge \gamma_2 \neq \varepsilon \\ \sigma_1 \dots \gamma_1 \gamma_2 \dots \sigma_n & \{\gamma_1, \gamma_2\} \cap \{\varepsilon, \sigma_i\} = \varnothing \end{cases}$$

$$\text{Insertion} = \begin{cases} \sigma_1 \dots \gamma_1 \gamma_2 \dots \sigma_n & \gamma_1 = \sigma_i \wedge \gamma_2 \notin \{\varepsilon, \sigma_i\} \\ \sigma_1 \dots \gamma_1 \gamma_2 \dots \sigma_n & \gamma_1 \notin \{\varepsilon, \sigma_i\} \wedge \gamma_2 = \sigma_i \\ \sigma_1 \dots \gamma_1 \gamma_2 \dots \sigma_n & \gamma_{1,2} = \sigma_i \end{cases}$$

This approach is tractable for $n \lesssim 100, q \lesssim 3$, however more complex repairs require a more efficient gradient estimator. As SAT encoding incurs a fixed penalty, for small repairs, we have

observed it is more efficient to enumerate all sketch templates and solve for admissible repairs outside the SAT solver.

8 CONTEXT-SENSITIVE REACHABILITY

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

$$P_1 := \big\{ \; S \rightarrow LR, \quad L \rightarrow ab \mid aLb, \quad R \rightarrow c \mid cR \; \big\}$$
$$P_2 := \big\{ \; S \rightarrow LR, \quad R \rightarrow bc \mid bRc, \quad L \rightarrow a \mid aL \; \big\}$$

Note that $\mathcal{L}_\cap$ generates the language $\big\{ \; a^d b^d c^d \mid d > 0 \; \big\}$, 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:

$$\bigwedge_{t \in \Sigma_\cap} \bigwedge_{j=1}^{c-1} \bigwedge_{i=1}^{|\sigma|} E(t_\cap, \mathcal{G}_j) \equiv_{\sigma_i} E(t_\cap, \mathcal{G}_{j+1}) \tag{4}$$

Fig. 3. 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.

9 BAR-HILLEL, IMPOSSIBLE SUBSTRINGS, AND CFL INTERSECTION

It is a well-known fact in formal language theory that CFLs are closed under intersection with regular languages, and the intersection between two CFLs is decidable. Anyhow, since we are working with bounded-width CFLs, everything collapses down to finite languages, which are closed under union, intersection and complementation. Thus, Bar-Hillel and other elegant constructions which took CS theorists many years to prove in full-generality become trivial when so restricted.

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.

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.

10   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? This transformation is admissible and indeed desirable when the subexpression is "complete", i.e., its derivation cannot be altered by appending or prepending text. Returning to Fig. 4, under what circumstances is the following reduction admissible?

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

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. How can we formalize this argument to work on arbitrary (e.g., potentially ambiguous) grammars? This question can be reduced to a quotient: Does there exist another nonterminal, when joined with the substring in question, that will "strip away" any of its tokens, leading to another derivation?

11   REALTIME ERROR CORRECTION

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 an input string, Σ^n, where should we put the holes? Assuming k holes, there are $\binom{n}{k}$ possible hole configurations (HCs), each with $(|\Sigma| + 1)^{2k}$ possible repairs (before parsing, cf. Eqs. ??-??). In practice, depending on n and k, this space can be intractable to search through exhaustively, so to facilitate real-time assistance we prioritize likely repairs according to an eight-step procedure:

(1) Fetch the most recent CFG and string from the editor.
(2) Exclude parsable substrings from hollowing.
(3) Lazily enumerate all HCs of increasing length.
(4) Sample HCs without replacement using Eq. ??.
(5) Prioritize HCs first by distance to caret index, then by Earthmover's distance to a set of suspicious indices.[*]
(6) Translate HCs to sketch templates using §??.
(7) Feed sketch templates to an incremental SAT solver.
(8) Decode and rerank models by Levenshtein distance.

[*] These locations can be supplied by local edit history or using tokenwise perplexity from a neural language model. Once a new repair is discovered, it is immediately displayed. Incoming keystrokes interrupt and reset the solving process.

12   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
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> ) )
...
```

12.1   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>
...
```

12.2   Interactive Nonterminal Expansion

Users can interactively build up a complex expression by placing the caret over a placeholder they wish to expand,

```
if <Vexp> X then <Vexp> else <Vexp>
```

then invoking Tidyparse by pressing ctrl + Space , to receive a list of expressions consistent with the grammar:

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

13 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(); }```

14 EVALUATION

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.
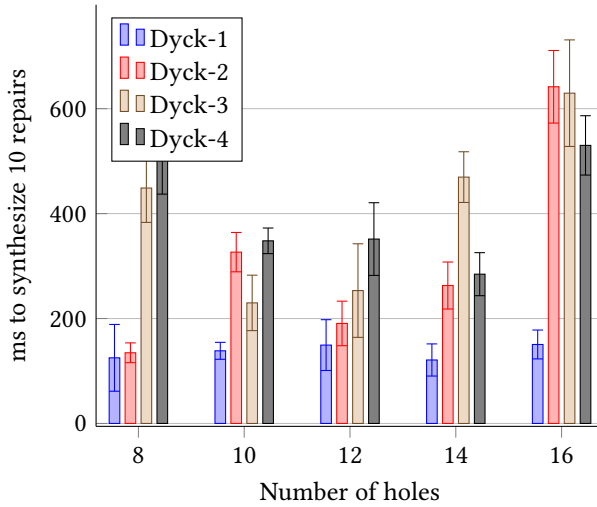
```
D1 -> () | ( D1 ) | D1 D1
D2 -> D1 | [ ] | ( D2 ) | [ D2 ] | D2 D2
D3 -> D2 | { } | ( D3 ) | [ D3 ] | { D3 } | D3 D3
```

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.

Error correction time 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.

Error correction time with unknown locations
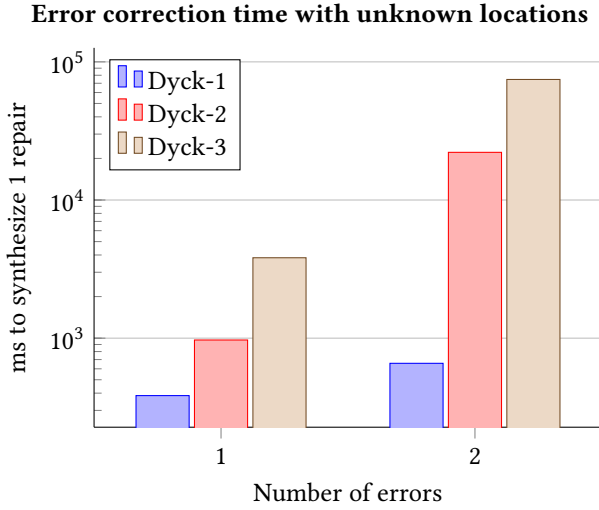
14.1 Task Difficulty

Level ½ - Incomplete code (all missing tokens at end)
Level 1 - Known number of errors at known locations
Level 2 - Known number of errors at unknown locations
Level 3 - Unknown number of errors at unknown locations

14.2 Repair Datasets

Synthetic - Synthetic errors in natural code snippets Natural - Natural errors and fixes mined from Git history

14.3 Evaluation Metrics

Syntactic validity - Only considers whether the repair parses Repair alignment - Measures string distance to true fix

14.4 Model configurations

Model configurations - LLM, ECP, ECP + LLM GraphCodeBERT RobertA Incoder CarperAI?

15 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 known to be concentrated in specific locations, such as the beginning or end of a string, then latency is typically below 500ms. Should errors occur uniformly at random, admissible repairs can take longer to discover, however these scenarios are unusual in our experience. We observe that errors are typically concentrated nearby historical edit locations, which can be retrieved from the IDE or version control. Further optimizations that reduce the total number of repairs checked are possible by eliminating improbable sketch templates.

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.

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

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

647   Although not intended to be a dedicated parser and we make no attempt to rigorously compare
648 parsing latency, parsing valid strings with Tidyparse is typically competitive with classical parsing
649 methods. Our primary motivation is to facilitate the usability and explainability of parsing with
650 errors. We envision three primary use cases: (1) helping novice programmers become more quickly
651 familiar with a new programming language (2) autocorrecting common typos among proficient
652 but forgetful programmers and (3) as a prototyping tool for PL educators and designers.

653   Featuring a grammar editor and built-in SAT solver, Tidyparse helps developers navigate the
654 language design space, visualize syntax trees, debug parsing errors and quickly generate simple
655 examples and counterexamples for testing. Although the algorithm may seem esoteric at first
656 glance, in our experience it is much more interpretable than classical parsers, which exhibit poor
657 error-recovery and diagnostics.

16 CONCLUSION

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

670   From a practical standpoint, we have implemented our approach as an IDE plugin and demon-
671 strated its viability as a tool for live programming. Tidyparse is capable of generating repairs
672 for invalid code in a range of toy languages. We plan to continue expanding its grammar and
673 autocorrection functionality to cover a broader range of languages and hope to conduct a more
674 thorough user study to validate its effectiveness in the near future.