Tidyparse: Real-Time Context Free Error Correction

Breandan Mark Considine
McGill University
bre@ndan.co

Jin Guo
McGill University
jguo@cs.mcgill.ca

Xujie Si
McGill University
xsi@cs.mcgill.ca

Abstract

Tidyparse is a program synthesizer that performs real-time error correction for context free languages. Given both an arbitrary context free grammar (CFG) and an invalid string, the tool lazily generates admissible repairs while the author is typing, ranked by Levenshtein edit distance. Repairs are guaranteed to be complete, grammatically consistent and minimal. Tidyparse is the first system of its kind offering these guarantees in a real-time editor. 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. We have released an IDE plugin demonstrating the system described.*

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 [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 [2], 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.

*https://plugins.jetbrains.com/plugin/19570-tidyparse

2 Related Work

Prior work in this area follows two main streams. Kats [7], De Jong [5] and Diekmann [6] investigate error repair with LR grammars and more recently, Raselimo and Fischer perform repairs within the grammar itself [10]. Our approach handles a more general class of context-free and bounded context-sensitive grammars and offers a more theoretically rigorous grounding in language reachability [9]. Consequently, it is much simpler and can scaled up using a GPU or TPU. As usual, plenty of tradeoffs exist, which are discussed in Sec. 12.

3 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 _ ( _ _ _ _ ! _ _ ) _ _ _ _
```

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

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 $w, x, z : V$ and $t : \Sigma$. For example, the CFG, $P := \{S \to SS \mid (S) \mid ()\}$, corresponds to the CNF:

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

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

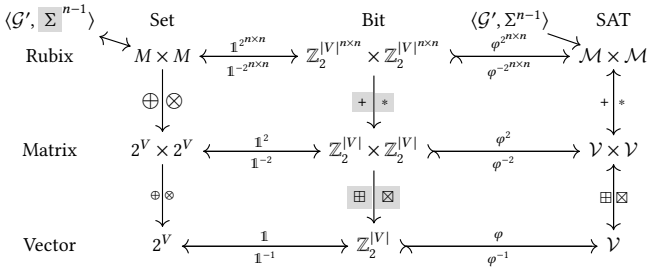
$$X \otimes Z := \left\{ w \mid \langle x, z \rangle \in X \times Z, (w \to xz) \in P \right\} \tag{1}$$

If we define $\sigma_r^{\pitchfork} := \{w \mid (w \to \sigma_r) \in P\}$, then initialize $M^0_{r+1=c}(\mathcal{G}', e) := \sigma_r^{\pitchfork}$ and solve for the fixpoint $M^* = M + M^2$,

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

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 lifting into the domain of bitvector variables, producing an algebraic expression for each scalar inhabitant of the northeastern-most bitvector Λ_σ^*, 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,[†]



where $\mathcal{V}$ is a function $\mathbb{Z}_2^{|V|} \to \mathbb{Z}_2$. Note that while it is always possible to encode $\mathbb{Z}_2^{|V|} \to \mathcal{V}$ using the identity function, the corresponding decoder (φ^{-1}) is not guaranteed to exist, since an arbitrary bitvector expression ($\mathcal{V}$) might take on 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 [11], who shows its time complexity to be $\mathcal{O}(n^\omega)$ where ω is the matrix multiplication bound ($\omega < 2.77$), and Lee [8], who shows that speedups to Boolean matrix multiplication are realizable by CFL parsers. Assuming sparsity, this technique is typically linearithmic, and we show it to be highly efficient in practice.

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

4.1 Context-sensitive reachability

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

$$P_1 := \left\{ S \to LR, \quad L \to ab \mid aLb, \quad R \to c \mid cR \right\}$$
$$P_2 := \left\{ S \to LR, \quad R \to bc \mid bRc, \quad L \to a \mid aL \right\}$$

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 $E(t_\cap, \mathcal{G}_i) = \{w_i \mid (w_i \to t_\cap) \in P_i\}$ and bind them together:

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

Figure 1. Orientations of a $\bigcap_{i=1}^{4} \mathcal{L}(\mathcal{G}_i) \cap \Sigma^6$ configuration. As $c \to \infty$, this shape approximates a circular cone whose symmetric axis joins σ_i with orthonormal unit productions $w_i \to t_\cap$, 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.

4.2 Encoding CFG parsing as SAT solving

By allowing the matrix M^* to contain bitvector variables representing holes in the string and nonterminal sets, we obtain a set of multilinear equations over $\mathbb{Z}_2$, whose solutions exactly correspond to the set of admissible repairs and their corresponding parse forests. Specifically, the repairs coincide with holes in the superdiagonal $M^*_{r+1=c}$, and the parse forests occur along the upper-triangular entries $M^*_{r+1<c}$.

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

Depicted in (3) 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.

4.3 Gradient estimation

Now that we have a reliable method to fix *localized* errors, $S : \mathcal{G} \times (\Sigma \cup \{\varepsilon, \_\})^n \to \{\Sigma^n\} \subseteq \mathcal{L}_{\mathcal{G}}$, given some unparseable string, i.e., $\underset{\sim}{\sigma_1 \ldots \sigma_n} : \Sigma^n \cap \mathcal{L}(\mathcal{G})^c$, 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:

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

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

Next, suppose $U : \mathbb{Z}_2^{m \times m}$ is a matrix whose structure is shown in Eq. 4, wherein C is a primitive polynomial over $\mathbb{Z}_2^m$ with coefficients $C_{1\ldots m}$ and semiring operators $\oplus := \veebar, \otimes := \wedge$:

$$U^t V = \begin{pmatrix} C_1 & \cdots & & C_m \\ \top & \circ & \cdots & \circ \\ \circ & \ddots & & \vdots \\ \vdots & & \ddots & \vdots \\ \circ & \cdots & \circ & \top \end{pmatrix}^t \begin{pmatrix} V_1 \\ \vdots \\ \\ V_m \end{pmatrix} \tag{4}$$

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

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 $i \in \cdot \in \binom{n}{d}$ 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:

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

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

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

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

5 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 $i_{1\ldots d} \in \binom{n}{d}$ 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 4.3.
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:

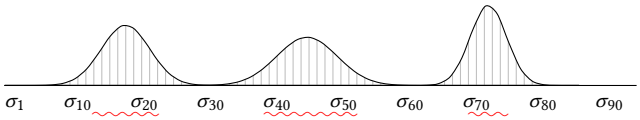


Figure 2. The distribution $\int \mathcal{F}_\theta(\cdot \mid i_{1\ldots d})d\theta$, projected onto the invalid string, suggests edit locations most likely to yield admissible repairs, from which we draw subsets of size d.

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:

$$\delta_{\text{KR}}(\mu, \nu) := \inf_{\pi \in \Pi(\mu,\nu)} \int_{\Omega \times \Omega} \delta(x, y) d\pi(x, y) \tag{5}$$

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*, $C : (\Sigma^* \times \Sigma^*) \to \mathbb{R}$, 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 [**?**] 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.

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

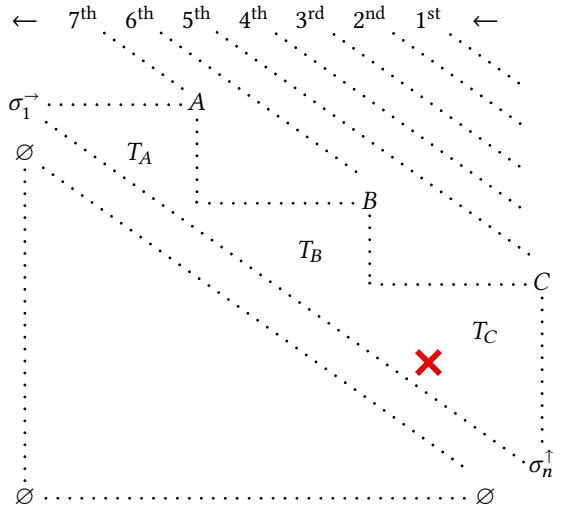


Figure 3. By recursing over upper diagonals of decreasing elevation and discarding all subtrees that fall under the shadow of another's canopy, we can recover parseable subtrees.

$$X \hat{\otimes} Z := \left\{ w \overset{x}{\underset{z}{\nearrow\!\searrow}} \mid \langle x, z\rangle \in X \times Z, (w \to xz) \in P \right\} \quad (6)$$

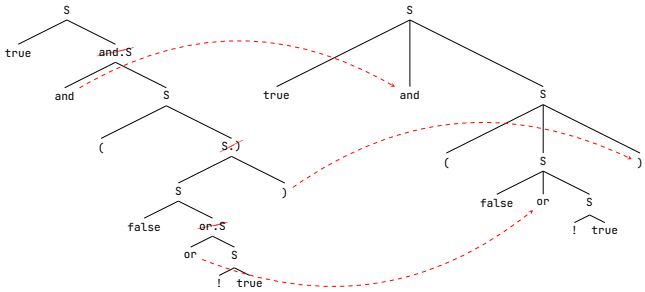
We first construct the parse forests bottom-up using $\hat{\otimes}$, which simply stores backpointers, then if $S \notin \Lambda^*_\sigma$, we traverse the peaks by decreasing elevation to recover parse branches.

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

```
Parseable subtrees (5 leaves / 2 branches):
└─ ! [3]
└─ and [4]
└─ false [5]
└─ or [6]
└─ true [7]
└─ S [0..2]
   ├─ true [0]
   ├─ and [1]
   └─ true [2]
└─ S [8..11]
   ├─ S [8..9]
   │  ├─ ! [8]
   │  └─ true [9]
   ├─ and [10]
   └─ false [11]
```

7 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.



Algorithm 1 Rewrite procedure for tree denormalization

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

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).

8 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, $\alpha \to^* \beta$:

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

This rule enables adding a placeholder for typed expressions:

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

After preprocessing, the above grammar will take the form:

```
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 $v \in V$:

$$\frac{\mathcal{G} \vdash v \in V}{\mathcal{G} \vdash (v \longrightarrow \langle v \rangle) \in P} \langle\cdot\rangle\text{-INT}$$

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

```
false or ! true or <S> and <S> or <S>
```

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

```
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>
...
```

9 Usage Examples

Tidyparse accepts any context-free 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 approximate the OCaml grammar. We use the `---` delimiter to separate the grammar from the example:

```
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 ( _ _ _ ) )
```

```
1.) let curry f = ( fun x y -> f ( <X> ) )
2.) let curry f = ( fun x y -> f ( <FunName> ) )
3.) 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 ) ) )
...
```

9.1 Context-sensitive languages

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. 4.1:

```
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.

9.2 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 is invalid, Tidyparse will suggest possible fixes. In the future, we intend to enhance this functionality to perform example-based codesign and grammar induction.

```
B -> true | false |
```

```
1.) B -> true | false
2.) B -> true | false <RHS>
3.) B -> true | false | <RHS>
...
```

CFG navigation and syntax highlighting are also provided.

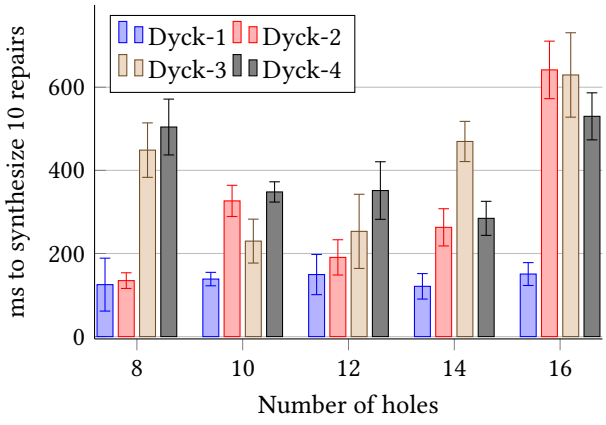
10 Latency Benchmark

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.

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

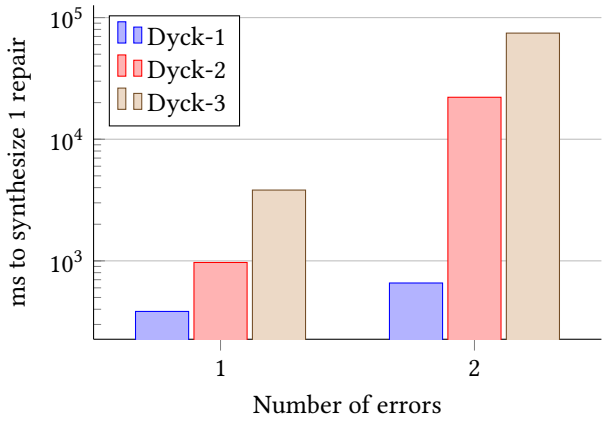
In the first experiment, we sample a random valid string $\sigma \sim \Sigma^{50} \cap \mathcal{L}_{\text{Dyck-n}}$, then replace a fixed number tokens with holes and measure the average time taken to decode ten syntactically-admissible repairs across 100 trial runs.

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.

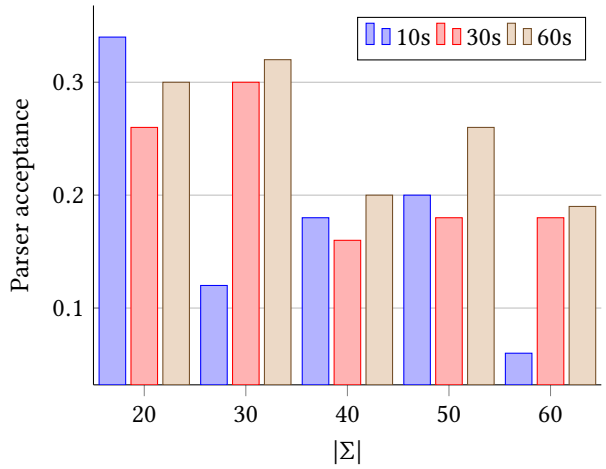
Latency with unknown locations



11 Accuracy Benchmark

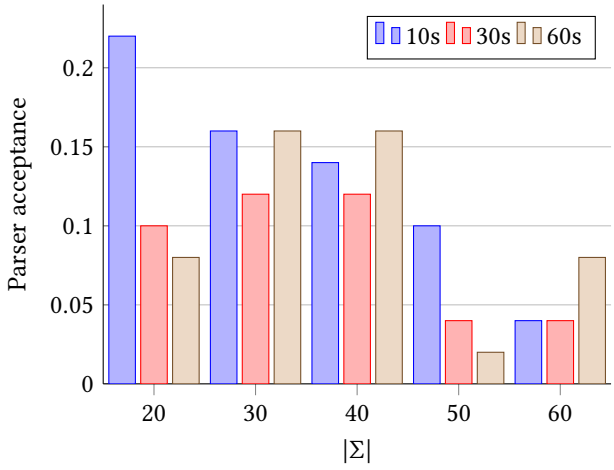
In the following benchmark, we analyze bracketing errors in a dataset of Java and Python code snippets mined from open-source repositories on GitHub. For Java, we sample valid single-line statements with bracket nesting more than two levels deep, synthetically delete one bracket uniformly at random,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

Synthetic error correction accuracy



For Python, we sample invalid code fragments uniformly from the imbalanced bracket category of the Break-It-Fix-It (BIFI) dataset [12], a dataset of organic Python errors, repair using Tidyparse[‡], take the top-1 repair after t seconds, and validate repairs using Python's `ast.parse()` method.

Organic error correction accuracy



As expected, accuracy on organic errors in Python is lower.

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

12 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. 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) [3], 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.

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.

13 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 has many advantages, enabling us to repair syntax errors, correct typos and generate parse trees for incomplete strings.

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. Further examples can be found at our GitHub repository: https://github.com/breandan/tidyparse

14 Acknowledgements

The first author would like to thank his co-advisor Xujie Si for providing many helpful suggestions during the development of this project, including the optimized fixpoint, test cases, and tree denomalization procedure. In addition, the authors extend their thanks to Nghi Bui at FPT Software for early feedback on the IDE plugin, Zhixin Xiong for contributing the OCaml grammar, Brigitte Pientka for asking the crucial question, "Where do you put the holes?", and Ori Roth for providing helpful comments on an early draft of this paper.

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] 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.

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

[4] 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 (Volume 2: Short Papers). Association for Computational Linguistics, Baltimore, Maryland, 625–630. https://doi.org/10.3115/v1/P14-2102

[5] Maartje De Jonge and Eelco Visser. 2012. Automated evaluation of syntax error recovery. In Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering.

[6] Lukas Diekmann and Laurence Tratt. 2018. Don't Panic! Better, Fewer, Syntax Errors for LR Parsers. arXiv preprint arXiv:1804.07133 (2018).

[7] Lennart CL Kats, Maartje de Jonge, Emma Nilsson-Nyman, and Eelco Visser. 2009. Providing rapid feedback in generated modular language environments: adding error recovery to scannerless generalized-LR parsing. ACM SIGPLAN Notices 44, 10 (2009), 445–464.

[8] 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

[9] David Melski and Thomas Reps. 1997. Interconvertbility of set constraints and context-free language reachability. ACM SIGPLAN Notices 32, 12 (1997), 74–89.

[10] Moeketsi Raselimo and Bernd Fischer. 2021. Automatic Grammar Repair. In Proceedings of the 14th ACM SIGPLAN International Conference on Software Language Engineering (Chicago, IL, USA) (SLE 2021). Association for Computing Machinery, New York, NY, USA, 126–142. https://doi.org/10.1145/3486608.3486910

[11] 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

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