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]

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.

[1] https://plugins.jetbrains.com/plugin/19570-tidyparse

2 Prior work

Prior work in this area follows two main streams. Kats [6] and deJong [4] and Diekmann [5] investigate error repair in LR grammars and more recently, Raselimo and Fischer use spectrum-based fault localization techniques [9]. Our approach can handle a more general class of context-free and bounded context-sensitive grammars and has a more theoretically rigorous grounding in language reachability [8]. Consequently, it is much simpler and can scaled up using a GPU. As always, tradeoffs exist, which we discuss in Sec. 11.

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

```
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.

4 Matrix Theory

Recall that a CFG is a quadruple consisting of terminals (Σ), nonterminals (V), productions ($P\colon V \rightarrow (V \mid \Sigma)^*$), and a start symbol, ($S$). It is a well-known fact that every CFG is reducible to *Chomsky Normal Form*, $P'\colon V \rightarrow (V^2 \mid \Sigma)$, 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:

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

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:

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

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

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

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 lifting into the domain of bitvector variables, producing an algebraic expression for each scalar inhabitant of the northeasternmost 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,[2]

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

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, 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 [10], who shows its time complexity to be $\mathcal{O}(n^\omega)$ where ω is the matrix multiplication bound ($\omega < 2.77$), and Lee [7], 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.

4.1 Context-sensitive reachability

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

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

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

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

Figure 1. 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.

4.2 Sampling k-combinations without replacement

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 $\oplus := \veebar, \otimes := \land$:

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

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 $\boldsymbol{\sigma} \sim \Sigma^d$ without replacement, we form an injection $\mathbb{Z}_2^m \rightharpoonup \Sigma^d$, cycle over $\mathbf{S}$, then discard samples which index any nonexistent element(s) of Σ. This method

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

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 $x^4 + x^3 + 1$:

$$\begin{array}{cccccccc} i & 0 & 1 & 2 & 3 & 4 & 5 & 6 \\ \mathbf{S}_i & 1000 & 0100 & 0010 & 1001 & 1100 & 0110 & 1011 \\ \sigma' & \text{C A} & \text{B A} & \text{A C} & \text{C B} & & \text{B C} & \end{array} \tag{4}$$

We will use this technique to lazily sample from the space of hole configurations without replacement as described in §7.

4.3 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 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 $M^*_{r+1=c}$, and the parse forests occur along the upper-triangular entries $M^*_{r+1<c}$.

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

Depicted above is a SAT tensor representing $\sigma_1\ \sigma_2\ \sigma_3$ _ . . . _ 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.

4.4 Deletion, substitution, and insertion

Deletion, substitution and insertion can be simulated by first adding a left- and right- ε-production to each unit production:

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

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

To generate the sketch templates, we substitute two holes at an index-to-be-repaired, and solve for admissible repairs. If a given sketch template has at least one repair, for each edit location in that template, exactly one of seven possible scenarios will occur, one corresponding to deletion, three for substitution and three for insertion:

$$\text{Deletion} = \begin{cases} \sigma_1\ \ldots\ \gamma_1\ \gamma_2\ \ldots\ \sigma_n & \gamma_{1,2} = \varepsilon \end{cases}$$

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

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

5 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. Occasionally, it is not possible to decode a full tree. In a typical parser, error recovery requires special tricks. Here, we simply analyze the structure of the matrix M to decode the parseable subtrees:

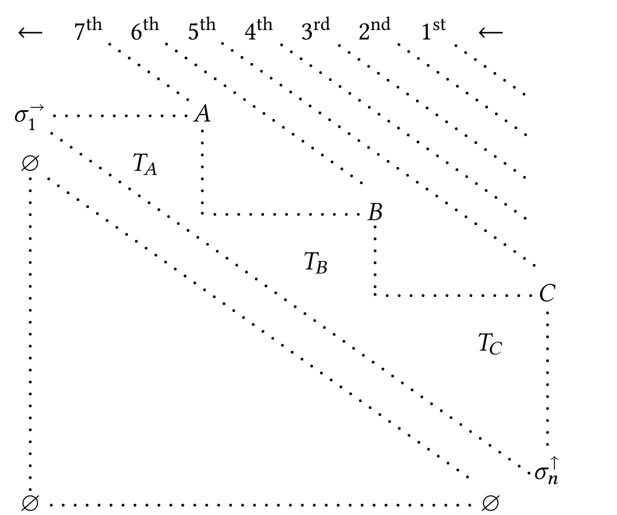


Figure 2. Peaks along the UT matrix ridge correspond to maximally parseable substrings. By recursing over upper diagonals of decreasing elevation and discarding all subtrees that fall under the shadow of another's canopy, we can recover the partial subtrees. The example depicted above contains three such branches, rooted at nonterminals C, B, A.

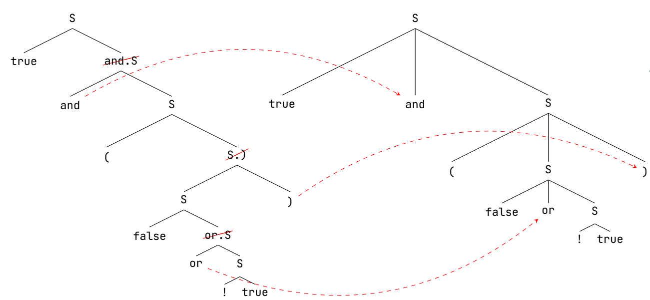
These branches correspond to peaks on the upper triangular (UT) matrix ridge. As depicted in Fig. 2, we traverse the peaks by decreasing elevation to collect partial AST branches.

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

```
Recovered 3 parseable leaves:
 🌿            🌿           🌿
└── ! [3]   └── and [4]   └── or [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]
```

6 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` $\in V_{\mathcal{G}'} \setminus V_{\mathcal{G}}$ **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).

7 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 §4.4.
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.

8 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 V0 Vexp
Vexp -> ( VarName , VarName ) | Vexp Vexp
VarName -> a | b | c | d | e | ... | z
FunName -> foldright | map | filter
V0 ->  + | - | * | / | > | = | < | `||` | &&
---
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> ) )
...
```

8.1 Context-sensitive languages

Many programming languages exhibit either lexical or syntactic context-sensitivity, e.g., Python indentation. Tidyparse can analyze such languages using finite CFL-intersection, allowing it to generalize to a broader family of languages:

```
S -> L R                S -> L R
L -> a b | a L b        R -> b c | b R c
R -> c | c R            L -> a | a L
```

8.2 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 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>
...
```

8.3 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 consistient with the grammar:

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

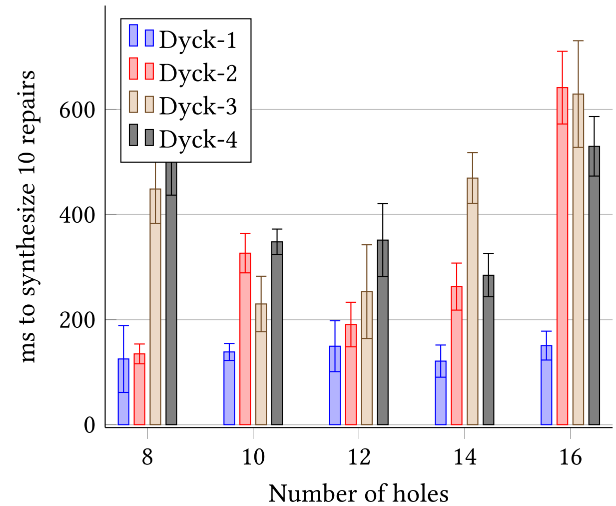
9 Latency Benchmark

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
```

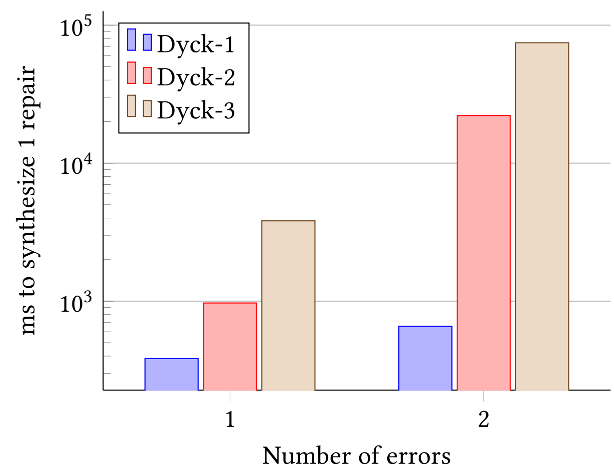
In the first experiment, we sample a random valid string $\sigma \sim \Sigma^{50} \cap \mathcal{L}_{\text{Dyck-n}}$, then replace a fixed number tokens with holes and measure the average time taken to decode ten syntactically-admissible repairs across 100 trial runs.

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



10 Accuracy Benchmark

We analyze synthetic errors in Java and Python.

We also compare with the Break-it-fix-it (BIFI) dataset [11].

11 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.

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.

Although not intended to be a dedicated parser and we make no attempt to rigorously compare parsing latency, parsing valid strings with Tidyparse is typically competitive with classical parsing methods. Our primary motivation is to facilitate the usability and explainability of parsing with errors. 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 educators and designers.

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. Although the algorithm may seem esoteric at first glance, in our experience it is much more interpretable than classical parsers, which exhibit poor error-recovery and diagnostics.

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

13 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] 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. 322–325.

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

[6] 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.

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

[9] 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

[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.