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. Our approach attains state-of-the-art repair precision@$\{1, 5, 10\}$ on a dataset of human Python repairs at a fraction of the cost of existing methods, and is easily extensible to other programming languages.

1 INTRODUCTION

Syntax correction is one of the first problems that novice programmers encounter when learning to write code and a perennial issue in computer programming. Put simply, it is the problem of repairing a syntactically incorrect program so that it parses. This problem is challenging because well-formed programs have many extra-syntactic constraints, increasing the complexity of repair, but also because the problem is highly under-determined: repairs must be consistient with the surrounding context, and the correct repair is seldom unique, even assuming minimality.

The majority of prior work on syntax correction can be separated into one of two high-level categories: (1) *pattern matching*, which uses hand-crafted rules to identify and correct common syntax errors, and (2) *machine learning*, i.e., which uses a statistical language model to repair code via a set of learned heuristics. The former approach is effective but requires language-specific rules, while the latter generates more natural edits, but is costly to train, generalizes poorly, has high sample complexity, and is difficult to incorporate new constraints without retraining.

In the following paper, we present a broadly different approach to robust parsing and syntax correction. Rather than manipulating strings or trees directly, our basic idea is to reframe the problem in terms of multilinear algebra, thereby gaining access to a multitude of powerful analysis techniques. This approach shares connections to language reachability, array programming and tensor completion: we show how to recast the syntax correction problem as a special case of tensor completion, integrating smoothly into both machine learning and logic programming.

Traditionally, parsers have ignored developer tools like code completion, to their detriment. The approach we propose offers a compelling alternative to its textbook presentation in the parsing literature, not only because it unifies parsing, code completion and error correction under a simple algebraic framework, but also because it is composable using ordinary logical primitives, which are highly flexible to additional constraints and well-suited for SAT-based implementation.

More specifically, our paper is structured as follows: we present two high-level approaches to syntax correction: one that samples random edits and accepts only those which parse (Theory 1, probabilistic correction), and another that uses equational reasoning to find the satisfying assignments to a system of multilinear equations over finite fields (Theory 2, model-theoretic). Finally, we show how these two approaches can be combined to attain state-of-the-art performance (Theory 1.5), and validate it on a variety of real-world program repair scenarios.

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

2 BACKGROUND

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

2.1 CFL Parsing

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

General CFL parsing allows ambiguity (non-unique trees) and be formulated as a dynamic programming problem, as shown by Cocke-Younger-Kasami (CYK) [22], Earley [15] and others. These parsers have roughly cubic time complexity with respect to the length of the input string.

As shown by Valiant [25], Lee [18] and others, general CFL recognition is in some sense equivalent to binary matrix multiplication, another well-studied combinatorial problem with broad applications, known to be at worst subcubic. This realization unlocks the door to a wide range of complexity-theoretic and practical speedups to CFL recognition and fast general parsing algorithms.

2.2 Language Equations

Language equations are a powerful tool for reasoning about formal languages and their inhabitants. First proposed by Ginsburg et al. [16] for the ALGOL language, language equations are essentially systems of inequalities with variables representing *holes*, i.e., unknown values, in the language or grammar. Solutions to these equations can be obtained using various fixpoint techniques, yielding members of the language. This insight reveals the true algebraic nature of CFLs and their cousins.

Being an algebraic formalism, language equations naturally give rise to a kind of calculus, vaguely reminiscent of Leibniz' and Newton's. First studied by Brzozowski [9, 10] and Antimirov [3], one can take the derivative of a language equation, yielding another equation, which can be interpreted as a kind of continuation or language quotient, returning the suffixes that complete a given prefix. This technique leads to an elegant family of algorithms for incremental parsing [1, 19] and automata minimization [8]. In our setting, differentiation corresponds to code completion.

In this paper, we restrict our attention to language equations over context-free and weakly context-sensitive languages, whose variables coincide with edit locations in the source code of a computer program, and solutions correspond to syntax repairs. Although prior work has studied the use of language equations for parsing [19], to our knowledge they have never previously been considered for the purpose of code completion or syntax error correction.

2.3 Syntax Repair

In finite languages, syntax repair corresponds to spelling correction, a more restrictive and largely solved problem. Schulz and Stoyan [23] construct a finite automaton that returns the nearest dictionary entry by Levenshtein edit distance. Though considerably simpler than syntax correction, their work shares similar challenges and offers insights for handling more general repair scenarios.

When a sentence is grammatically invalid, parsing grows more challenging. Like spelling, the problem is to find the minimum number of edits required to transform an arbitrary string into a syntactically valid one, where validity is defined as containment in a (typically) context-free language. Early work, including Aho [2**?**] propose a dynamic programming algorithm that computes the minimum number of edits required to transform a string into a valid one. The problem of actually

generating the repairs however is not well-posed, as there are often many valid strings that can be obtained within a given number of edits. We instead focus on bounded Levenshtein reachability, which is the problem of finding useful repairs within a fixed Levenshtein distance of the broken string, which requires language intersection.

Bar-Hillel [6] establishes the closure of CFLs under intersection with regular languages, and can be used to construct the corresponding minimum-cost CFG in a straightforward manner. However while generally quite expressive, CFLs are themselves not closed under intersection and have other practical limitations, i.e., are unable to express indentation or variable binding. These limitations motivate us towards more expressive yet still efficiently parsable formalisms.

Conveniently, Okhotin [21] offers exactly the formalism needed: language equations augmented with logical operators like conjunction or disjunction. These operators afford us the flexibility to encode both language union and intersection, which are difficult or impossible to express using a single grammar, as well as incorporate extra-syntactic side-constraints during solving.

3 PROBLEM STATEMENT

The context-free Levenshtein reachability problem can be formally stated as follows:

Definition: The language edit distance (LED) is the minimum number of edits required to transform a string into a valid one, where validity is defined as containment in a context-free language, $\ell : \mathcal{L}$, i.e., $\Delta^*(\underset{\sim}{\sigma}, \ell) := \min_{\sigma \in \ell} \Delta(\underset{\sim}{\sigma}, \sigma)$, and Δ is the Levenshtein distance.

We seek to find the set of strings S such that $\forall \tilde{\sigma} \in S, \Delta(\underset{\sim}{\sigma}, \tilde{\sigma}) \leq k$, where k is the maximum number of edits greater than or equal to the language edit distance. We call this set the *Levenshtein hypersphere* of $\underset{\sim}{\sigma}$ and denote it $\Delta_k(\underset{\sim}{\sigma})$. Since k is lower-bounded by $\Delta^*(\underset{\sim}{\sigma}, \ell)$, we have $1 \leq \Delta_k(\underset{\sim}{\sigma})$.

We now consider an upper bound on $\Delta^*(\underset{\sim}{\sigma}, \ell)$.

Lemma 3.1. *For any nonempty language $\ell : \mathcal{L}(\mathcal{G})$ and invalid string $\underset{\sim}{\sigma} : \Sigma^n$, there exists an $(\tilde{\sigma}, m)$ such that $\tilde{\sigma} \in \ell \cap \Sigma^m$ and $0 < \Delta(\underset{\sim}{\sigma}, \ell) \leq \max(m, n) < \infty$.*

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

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

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

In practice, this problem is ill-posed even when $k = \Delta^*(\underset{\sim}{\sigma}, \ell)$. For example, consider the language of ursine dietary preferences. Although $\underset{\sim}{\sigma} :=$ "Bears like to eat tomatoes" is not in the language, Δ^* is 1, e.g., "Bears like to eat", however there are many other variations with the same edit distance, e.g., or "Bears like to eat {berries, honey, fish}". We may also wish to consider two or more edits, e.g., "{Polar, Panda} bears like to eat {seals, bamboo}". In general, there are many valid strings within a fixed Levenshtein distance of $\underset{\sim}{\sigma}$, and we seek to find among them, the set of strings that are both syntactically valid and semantically plausible within a short amount of time.

4 APPROACH

We now present a number of optimality criteria for the parsing framework. These criteria were used to guide the design of various components and evaluate its performance.

- First and foremost, the framework must be **sound** and **complete**. That is, the parser-generator must (1) accept arbitrary conjunctive grammars, and (2) generate a parser which

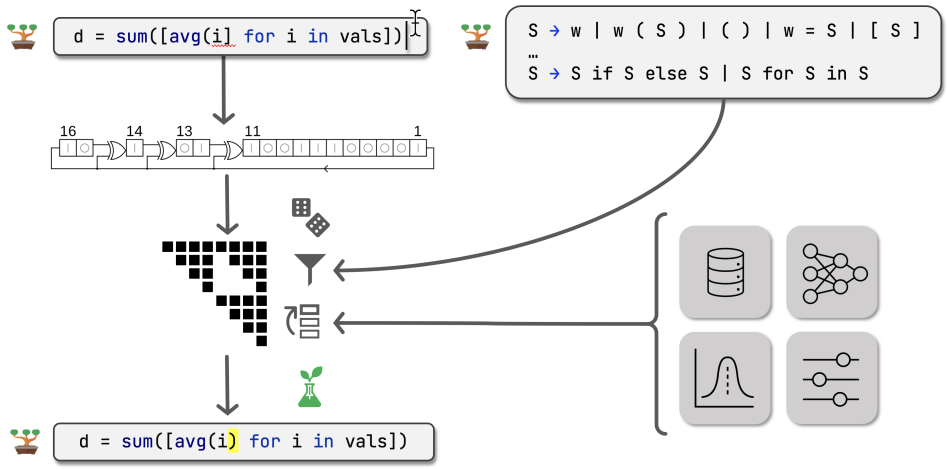


Fig. 1. Overview of the framework.

accepts all and only syntactically valid strings and (3) when a string is invalid, our synthesizer must eventually generate all syntactically valid strings within a fixed distance from it.

- Second, we require that the resulting repairs be **plausible** and **diverse**. In other words, the framework should generate repairs that are likely to be written by a human being, consistent with the surrounding context, and reasonably diverse in their structure.
- Third, the framework must be **efficient** and **responsive**. That is, it must be able to recognize well-formed strings in subcubic time, and generate admissible repairs in subpolynomial time. These conditions are necessary to provide real-time feedback whilst the user is typing.
- Fourth, the framework must also be **robust** and **scalable**. In practice, this means that the framework should be robust to multiple errors, handle grammars with a large number of productions and be able to scale linearly as the number of processor cores are increased.
- Finally, the framework must be **flexible** and **extensible**. Intended as a general-purpose tool for generating syntax repairs in a wide variety of programming languages, end-users should be able to extend the framework with their own custom grammars and side-constraints.

With these criteria in mind, we now present the parsing framework in Section **??**.

5 OVERVIEW

Our architecture accepts a grammar and a string, which can contain any arbitrary characters. Given an input string, we first attempt to parse using Valiant's algorithm [26]. If the string is syntactically correct, we return the parse forest, otherwise, we return the partial parse forest and set of edits that would make the string syntactically correct. This is always guaranteed to exist, as the trivial solution is to delete the entire string and replace it with the shorest string in the langauge.

If the string is invalid, we first form a bijection between edits and the Levenshtein hypersphere using a combinatorial number system, sample without replacement using an LFSR with leapfrog partitioning to distribute the work across cores, and decode the resulting bitvectors to enumerate edit templates. This sequence can be viewed as a permutation of a space-filling curve over the space of strings within a fixed edit distance of the original, invalid, string.

We then use each edit template, in conjunction with the grammar to search for admissible repairs. This is done by encoding the template and grammar as a matrix equivalence relation and solving

for the least fixpoint using a linear system of equations over a finite field. We describe this process in more detail in Section 8. The resulting samples will all be syntactically valid according to the grammar. These are reranked according to a statistical distance metric, and we output the most likely samples to the user. This whole process is illustrated in Figure 1.

6 TOY EXAMPLE

Suppose we are given the following context-free grammar:

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

For reasons that will become clear in Sec. 8, this will automatically be rewritten into the grammar:

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

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

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

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

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

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

7 PYTHON CORRECTION

Given an invalid string, the tool will first map the string to a coarsened version and generate edits:

```
v = [float(n for n in l.split(':')]
v = [float(n for n in l.split(':')]
v = [float()n for n in l.split(':')]
v = [float() for n in l.split(':')]
v = [float(n) for n in l.split(':')]
```

```
w = [ w ( w w w w w . w ( w ) ]
w = [ w ( w w w w w . w ( w ) ]
w = [ w ( ) w w w w w . w ( w ) ]
w = [ w ( ) w w w w . w ( w ) ]
w = [ w ( w ) w w w w . w ( w ) ]
```

This coarsening is done to reduce the number of possible corrections, and is admissible because CFLs are closed under homomorphisms. (If we wanted to provide a lexical expansion, this would be possible.) These candidates then reranked using a probability metric.

```
v = [float(n for n in l.split(':')]
v = [float(n) for n in l.split(':')]
v = [float(n for n in l.split(':')]
v = [float()n for n in l.split(':')]
v = [float() for n in l.split(':')]
```

```
w = [ w ( w w w w w . w ( w ) ]
w = [ w ( w ) w w w w . w ( w ) ]
w = [ w ( w w w w w . w ( w ) ]
w = [ w ( ) w w w w w . w ( w ) ]
w = [ w ( ) w w w w . w ( w ) ]
```

Finally, we eliminate everything which is not parsed by the official Python parser.

```
v = [float(n for n in l.split(':')]
v = [float(n) for n in l.split(':')]
v = [float() for n in l.split(':')]
```

```
w = [ w ( w w w w w . w ( w ) ]
w = [ w ( w ) w w w w . w ( w ) ]
w = [ w ( ) w w w w . w ( w ) ]
```

8 MATRIX THEORY

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

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

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

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

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

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

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

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

Details of the bisimilarity between parsing and matrix multiplication can be found in Valiant [25], who first realized its time complexity was subcubic $\mathcal{O}(n^\omega)$ where ω is the asymptotic lower bound for Boolean matrix multiplication ($\omega < 2.77$), and Lee [18], who shows that speedups to CFL parsing were realizable by Boolean matrix multiplication algorithms. While more efficient specialized parsers are known to exist for restricted CFGs, this technique is typically linearithmic under sparsity and believed to be the most efficient general procedure for CFL parsing.

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

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

where $\mathcal{V}$ is a function $\mathbb{Z}_2^{|V|} \to \mathbb{Z}_2$. Note that while always possible to encode $\mathbb{Z}_2^{|V|} \to \mathcal{V}$ using the identity function, an arbitrary $\mathcal{V}$ might have zero, one, or in general, multiple solutions in $\mathbb{Z}_2^{|V|}$. In practice, this means that a language equation can be unsatisfiable or underconstrained, however if a solution exists, it can always be decoded into a valid sentence and parse forest in the language.

So far, we have only considered the syntactic theory of breadth-bounded CFLs with holes, however, our construction can be easily extended to handle the family of CFLs closed under conjunction. The additional expressivity afforded by the language conjunction operator will be

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

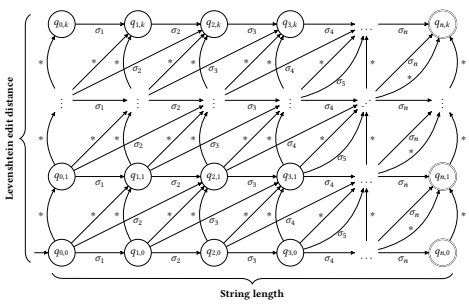
indispensable when considering practical program repair scenarios, which may require extra-grammatical constraints such as indentation-sensitivity or Levenshtein-bounded reachability. That extension, and the resulting theory of breadth-bounded CJLs with holes, will be explored in Sec. 11.

9 LEVENSHTEIN REACHABILITY

In the case where A and B are both fixed strings, Levenshtein distance can be interpreted as a shortest path problem over an unweighted graph whose vertices are the strings A, B and edges represent Levenshtein edits. The distance between A and B then, is simply the length of the shortest path(s). When B is a free variable, we can define a finite automaton accepting only strings within a given Levenshtein distance d of A by unrolling transition dynamics $\mathcal{L}(A, d)$ up to a fixed depth d.

Levenshtein reachability is recognized by the nondeterministic infinite automaton (NIA) whose topology $\mathcal{L}$ = can be factored into a product of (a) the monotone Chebyshev topology , equipped with horizontal transitions accepting σ_i and vertical transitions accepting Kleene stars, and (b) the monotone knight's topology , equipped with transitions accepting σ_{i+2}. The structure of this space is representable as an acyclic NFA [23], populated by accept states within radius k of $q_{n,0}$, or equivalently, a left-linear CFG whose productions finitely instantiate the transition dynamics:

NFA



CFG

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

Fig. 2. Levenshtein reachability from Σ^n can be described as either an acyclic ϵ-free NFA, or a left-linear CFG.

10 LANGUAGE EDIT REACHABILITY

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

Let $G(\underset{\sim}{\sigma} : \Sigma^*, d : \mathbb{N}^+) \mapsto \mathbb{G}$ be the specific construction described in Sec. 9 which accepts a string, $\underset{\sim}{\sigma}$, and an edit distance, d, and returns a grammar representing the NFA that recognizes the language of all strings within Levenshtein radius d of $\underset{\sim}{\sigma}$. To find the language edit distance and corresponding least-distance edit, we must find the smallest d such that $\mathcal{L}_d^{\cap}$ is nonempty, where $\mathcal{L}_d^{\cap}$ is defined as $\mathcal{L}(G(\underset{\sim}{\sigma}, d)) \cap \mathcal{L}(\mathcal{G}')$. In other words, we seek $\tilde{\sigma}$ and d^* under which three criteria are equisatisfiable: (1) $\tilde{\sigma} \in \mathcal{L}(\mathcal{G}')$, and (2) $\Delta(\underset{\sim}{\sigma}, \tilde{\sigma}) \leq d^* \iff \tilde{\sigma} \in \mathcal{L}(G(\underset{\sim}{\sigma}, d^*))$, and (3) $\nexists \sigma' \in \mathcal{L}(\mathcal{G}').[\Delta(\underset{\sim}{\sigma}, \sigma') < d^*]$. To satisfy these criteria, it suffices to check $d \in (1, d_{\max}]$ by

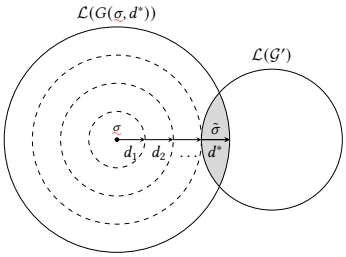


Fig. 3. LED is computed gradually by incrementing d until $\mathcal{L}_d^{\cap} \neq \varnothing$.

encoding the Levenshtein automata and the original grammar as a single SAT formula, call it, $\varphi_d(\cdot)$ and gradually admitting new acceptance states at increasing radii. If $\varphi_d(\cdot)$ returns UNSAT, d is increased until either (1) a satisfying assignment is found or (2) $d_{\max}$ is attained. This procedure is guaranteed to terminate in at most either (1) the number of steps required to overwrite every symbol in $\underset{\sim}{\sigma}$, or (2) the length of the shortest string in $\mathcal{L}(\mathcal{G}')$, whichever is greater. More precisely:

$$\varphi_{d+1}(\mathcal{G}', \underset{\sim}{\sigma}) := \begin{cases} \varphi[\tilde{\sigma} \in \mathcal{L}(G(\underset{\sim}{\sigma}, d)) \wedge \tilde{\sigma} \in \mathcal{L}(\mathcal{G}')] & \text{if } d = 1 \text{ or SAT.} \\ \varphi_d \oplus \bigoplus_{\{q \in Q \mid \delta(q, q_{n,0}) = d+1\}} \varphi[S \to q] & \text{if } d \leq \max(|\underset{\sim}{\sigma}|, \min_{\sigma \in \mathcal{L}(\mathcal{G}')} |\sigma|). \end{cases} \tag{2}$$

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

11 LINEAR CONJUNCTIVE REACHABILITY

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

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

Note that $\mathcal{L}_\cap$ generates the language $\{\, a^d b^d c^d \mid d > 0 \,\}$, which according to the pumping lemma is not context-free. 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 using conjunction:

$$\bigwedge_{t \in \Sigma_\cap} \bigwedge_{j=1}^{c-1} \bigwedge_{i=1}^{|\sigma|} E(t_\cap, \mathcal{G}_j) \equiv_{\sigma_i} E(t_\cap, \mathcal{G}_{j+1}) \tag{3}$$

Fig. 4. 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.

It is a well-known fact in formal language theory that CFLs are closed under union, composition, substitution and intersection with regular languages, and the intersection between two CFLs is decidable. However CFLs are not closed under intersection, which requires a more general formalism. Following Okhotin [20], we define higher-order grammar combinators $\cup, \cap : \mathcal{G}^* \times \mathcal{G}^* \to \mathcal{G}^*$ where $\mathcal{G}^*$ is a conjunctive grammar, which are basically finite collections of CFGs. Unlike parser combinators which are susceptible to ambiguity errors, our grammar combinators return parse forests in case of syntactic ambiguity, and do not suffer from the same shortcomings.

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

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

Using the technique from Sec. 9, we can also compute language edit distance, the minimum number of Levenshtein edits required to fix a syntactically invalid string. Language intersection is significantly faster than approximating the gradient via sampling.

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

12 TREE DENORMALIZATION

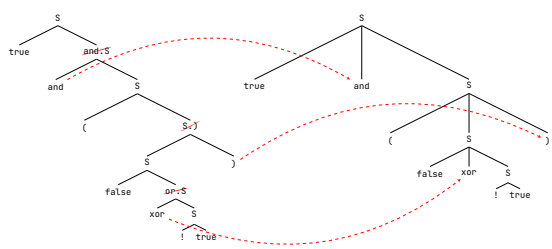
Our parser emits a binary forest consisting of parse trees for the candidate string which are constructed bottom-up using a variant of $\otimes$ called $\hat{\otimes}$, which simply records backpointers:

$$X \hat{\otimes} Z := \left\{ w \begin{smallmatrix} \nearrow x \\ \searrow z \end{smallmatrix} \mid \langle x, z \rangle \in X \times Z, (w \to xz) \in P \right\} \tag{4}$$

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

Algorithm 1 Tree denormalization

```
procedure CUT(t: Tree)
    stems ← {CUT(c) | c ∈ t.children}
    if t.root ∈ (V_G' \ V_G) then
        return stems
    else
        return { Tree(t.root, stems) }
    end if
end procedure
```

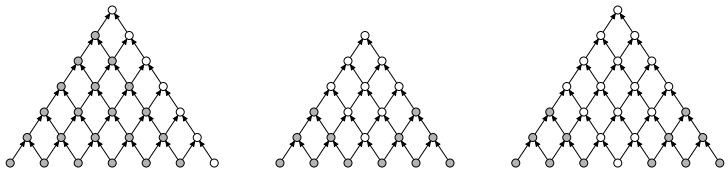


Pre-Denormalization Post-Denormalization

Fig. 5. Since $\mathcal{G}'$ contains synthetic nodes, to recover a parse tree congruent with the original grammar $\mathcal{G}$, we prune all synthetic nodes and graft their stems onto the grandparent via a simple recursive procedure (Alg. 1).

13 PARSER INCREMENTALIZATION

When the string is altered, we can reuse prior work by only recomputing affected submatrices, yielding a parser whose complexity is location-dependent, i.e., at worst quadratic in terms of $|\Sigma^*|$ assuming $\mathcal{O}(1)$ cost for each CNF-nonterminal subset join, $V_1' \otimes V_2'$. Borrowing the notation from probabilistic graphical models, where shaded nodes denote bound variables and unshaded nodes are unobserved:

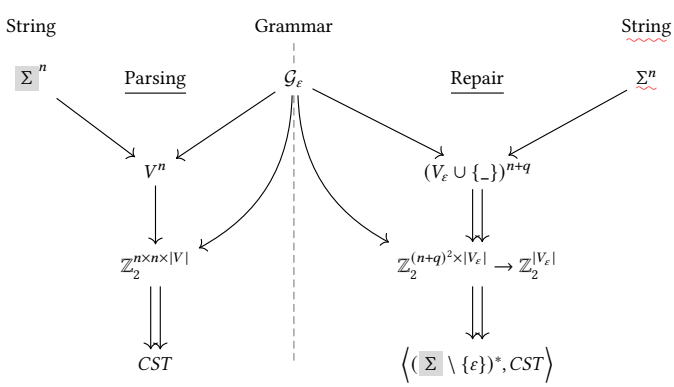


Append $\mathcal{O}(n+1)$ | Delete $\mathcal{O}\left(\frac{1}{4}(n-1)^2\right)$ | Insert $\mathcal{O}\left(\frac{1}{4}(n+1)^2\right)$

The problem of incremental parsing is closely related to *dynamic matrix inverse* in the linear algebra setting, and *incremental transitive closure* with vertex updates in the graph setting. By carefully encoding the matrix relation from Sec. 8 and employing an incremental SAT solver, we can gradually update SAT constraints as new keystrokes are received to eliminate redundancy.

14 FROM CFG PARSING TO SAT SOLVING

Parsing can be viewed as a special case of repair, however for the sake of efficiency, we first attempt to parse, then resort to repair. Repair can be solved analytically as constraint satisfaction problem using SAT, which allows us to easily combine constraints and solve them incrementally.



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

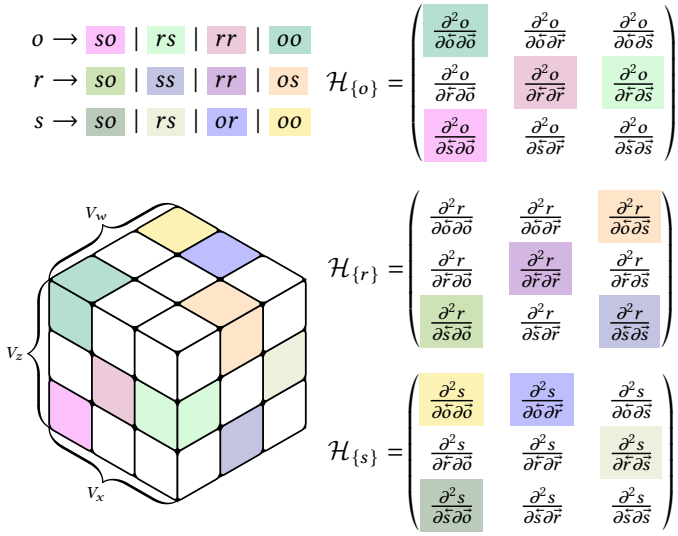


Fig. 6. CFGs are witnessed by a rank-3 tensor, whose inhabitants indicate CNF productions. Gradients in this setting effectively condition the parse tensor M by constraining the superposition of admissible parse forests.

15 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
$x \otimes z := \{ w \mid (w \rightarrow xz) \in P \}$	$\frac{\partial w}{\partial \overleftarrow{x}} := \{ z \mid (w \rightarrow xz) \in P \}$	$\frac{\partial w}{\partial \overleftarrow{z}} := \{ x \mid (w \rightarrow xz) \in P \}$
$\begin{array}{c\|c} x & w \\ \hline & z \end{array}$	$\begin{array}{c\|c} x & w \\ \hline & z \end{array}$	$\begin{array}{c\|c} x & w \\ \hline & z \end{array}$

The left quotient coincides with the derivative operator first proposed by Brzozowski [9] and Antimirov [3] over regular languages, lifted into the context-free setting (our work). When the root and LHS are fixed, e.g., $\frac{\partial S}{\partial \overleftarrow{x}} : (\overleftarrow{V} \rightarrow S) \rightarrow \vec{\mathcal{V}}$ returns the set of admissible nonterminals to the RHS. One may also consider a gradient operator, $\overleftarrow{\nabla} S : (\overleftarrow{\mathcal{V}} \rightarrow S) \rightarrow \vec{\mathcal{V}}$, which simultaneously tracks the partials with respect to a set of multiple LHS nonterminals produced by a fixed root.

If the root itself is unknown, we can define an operator, $\mathcal{H}_{\mathcal{W} \subseteq \mathcal{V}} : (\overleftarrow{\mathcal{V}} \times \vec{\mathcal{V}} \times \mathcal{W}) \rightarrow (\vec{\mathcal{V}} \times \vec{\mathcal{V}} \rightarrow \mathcal{W})$, 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. By allowing the matrix $\mathcal{M}^*$ to contain bitvector variables representing holes in σ, we obtain a set of multilinear equations whose solutions exactly correspond to the set of admissible repairs and their corresponding parse forests. Specifically, the repairs coincide with holes in the superdiagonal $\mathcal{M}^*_{r+1=c}$, and the parse forests occur along upper-triangular entries $\mathcal{M}^*_{r+1<c}$.

15.1 Gradient estimation

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

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

To implement this strategy, we carefully construct a bijection between Levenshtein edits to a known string and the integers, sample integer vectors without replacement using a characteristic polynomial over a finite field, then decode vectors into repairs using a combinatorial indexing scheme that puts integer vectors into one-to-one correspondence with Levenshtein edits. We find this approach computationally more tractable while yielding a steady stream of admissible edits throughout the solving process, regardless of the grammar or string under repair.

More specifically, we employ a pair of [un]tupling functions $\kappa, \rho : \mathbb{N}^k \leftrightarrow \mathbb{N}$ which are (1) bijective (2) maximally compact (3) computationally tractable (i.e., closed form inverses). κ will be used to index $\left\{{n \atop k}\right\}$[2]-combinations via the Maculay representation. ρ will index Σ^k tuples, but is slightly more tricky to define. To maximize compactness, there is an elegant pairing function courtesy of Szudzik [24], which enumerates concentric square shells over the plane $\mathbb{N}^2$ and can be generalized to hypercubic shells in $\mathbb{N}^k$. For our purposes, this generalization will suffice.

Although $\langle \kappa, \rho \rangle$ could be used directly to exhaustively search the Levenshtein hypersphere, they are temporally biased samplers. Rather, we would prefer a path that uniformly visits every fertile subspace of the Levenshtein hypersphere over time regardless of the grammar or string in question: subsequences of $\langle \kappa, \rho \rangle$ should discover valid repairs with frequency roughly proportional to the ratio of admissible edits over all possible edits within a fixed distance of the original string. These additional constraints give rise to two more criteria: (1) ergodicity and (2) periodicity.

To achieve ergodicity, we permute the elements of $\left\{{n \atop k}\right\} \times \Sigma^k$ using a finite field with a characteristic polynomial C of degree $m := \lceil \log_p \binom{n}{k} |\Sigma|^k \rceil$. By choosing C to be some irreducible polynomial, one ensures the path is fully periodic while guaranteeing the mixing properties we desire, i.e., suppose $U : \mathbb{Z}_2^{m \times m}$ is a matrix whose structure is shown in Eq. 5, wherein C is a primitive polynomial over $\mathbb{Z}_2^m$ with coefficients $C_{1 \ldots m}$ and semiring operators $\oplus := + \pmod 2, \otimes := \wedge, \top := 1, \circ := 0$.

$$U^{\intercal} V = \begin{pmatrix} C_1 & \cdots & \cdots & C_m \\ \top & \circ & \cdots & \circ \\ \circ & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \vdots \\ \circ & \cdots & \circ \quad \top & \circ \end{pmatrix}^t \begin{pmatrix} V_1 \\ \vdots \\ \vdots \\ V_m \end{pmatrix} \tag{5}$$

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

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

In addition to its statistically desirable properties, our sampler has the practical benefit of being trivially parallelizable using the leapfrog method, i.e., given p independent processors, each one can independently check $\langle \kappa, \rho \rangle^{-1}(\mathrm{S}_i) \in \mathcal{L}(\mathcal{G})$? where $i \equiv p_j \pmod{p}$. This procedure linearly scales with the number of processors, exhaustively searching $\Delta_q(\underset{\sim}{\sigma})$ in p^{-1} of the time required by a single processor, or alternately drawing p times as many samples in the same amount of time.

To admit variable-length edits, we first define a ε^+-production and introduce it to the right- and left-hand side of each terminal in a unit production:

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

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

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

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

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

This approach is tractable for $n \lesssim 100$, $q \lesssim 3$, however more complex repairs require a more efficient gradient estimator. We will discuss two alternate approaches, one using an adaptive sampler (Sec. 16) and another based on Levenshtein reachability (Sec. 9).

16 PROBABILISTIC REACHABILITY

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 a large number of unnatural edits. To provide rapid and relevant suggestions, we prioritize candidate repairs according to the following seven-step procedure:

(1) Retrieve the most recent grammar, $\mathcal{G}$, and string, $\underset{\sim}{\sigma}$, from the editor.
(2) Sample completions for each template from $\boldsymbol{\sigma}_i \sim \left\{{n \atop d}\right\} \times \Sigma^{n\pm q} \cap \Delta_q(\underset{\sim}{\sigma})$ WoR using Eq. 5.
(3) Filter completions by admissibility with respect to the grammar, $\boldsymbol{\sigma}_i \cap \mathcal{L}_{\mathcal{G}}$.
(4) Rerank admissible repairs by the edit cost model, $C(\underset{\sim}{\sigma}, \tilde{\sigma})$.
(5) Display the top-k repairs by edit cost found within p-seconds to the user.

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

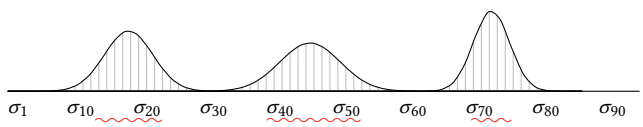


Fig. 7. The distribution $\int \mathcal{F}_\theta(\cdot \mid i_{1\ldots d})d\theta$, projected onto the invalid string, suggests edit locations most likely to yield admissible repairs, from which we draw subsets of size d.

Morally, we would prefer sketch templates likely to yield repairs that are (1) admissible (i.e., grammatically correct) and (2) plausible (i.e., likely to have been written by a human author). To do so, we draw holes and rank admissible repairs using a distance metric over $\Delta_q(\underset{\sim}{\sigma})$. One such metric, the Kantorovich–Rubinstein (KR) metric, δ_{KR}, can be viewed as an optimal transport problem minimizing $\Pi(\mu, \nu)$, the set of all mass-conserving transportation plans between two probability distributions μ and ν over a metric space Ω:

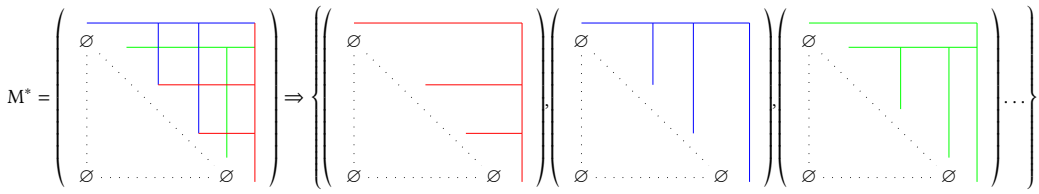
$$\delta_{\mathrm{KR}}(\mu, \nu) := \inf_{\pi \in \Pi(\mu,\nu)} \int_{\Omega\times\Omega} \delta(x, y) d\pi(x, y) \tag{6}$$

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, weighted Levenshtein distance, or stochastic contextual edit distance [13] in the case of probabilistic edits. Our goal then, is to discover repairs which minimize $C(\underset{\sim}{\sigma}, \tilde{\sigma})$, subject to the given grammar and latency constraints.

In the following section, we will give a more efficient construction for generating and accepting $\Delta_q(\underset{\sim}{\sigma})$ that incorporates (1) and (3), but does not explicitly require enumerating hole configurations.

17 ERROR RECOVERY

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



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

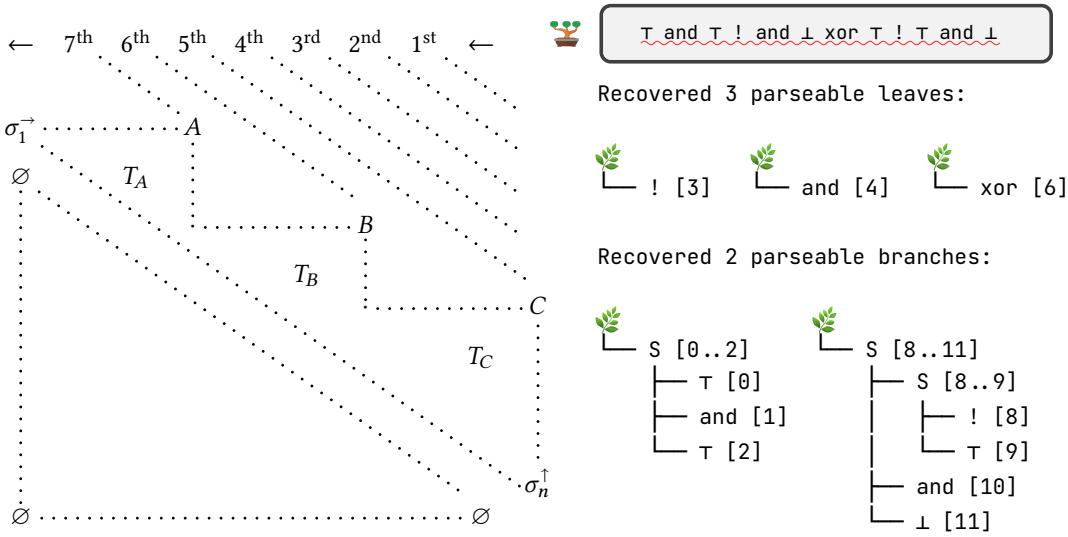


Fig. 8. Peaks along the UT matrix ridge correspond to maximally parseable substrings. By recursing over upper diagonals of decreasing elevation and discarding all subtrees that fall under the shadow of another's canopy, we can recover the partial subtrees. The example depicted above contains three such branches, rooted at nonterminals C, B, A.

These branches are located on peaks of the upper triangular (UT) matrix ridge. As depicted in Fig. 8, we traverse the peaks by decreasing elevation to collect partial AST branches and display the highest nonoverlapping branches, in this case T_C and T_A to the user, to help them diagnose the parsing error and manually repair it.

18 NONTERMINAL STUBS

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

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

Some dependently typed programming languages can do parsing in the type checker – our parser can also perform some form of type checking. Tidyparse automatically expands typed expressions into ordinary nonterminals using the α-SUB rule, for example when parsing an expression of the form $x + y$, the grammar will recognize `true + false` and `1 + 2`, but not `1 + true`.

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

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

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

This functionality can also be useful inside a completion, which might be expanded as follows:

```
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>
...
```

19 PRACTICAL EXAMPLE

Tidyparse requires a grammar, which can either be provided by the user or ingested from a BNF specification. The following grammar represents a slightly more realistic programming language:

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

We can also handle error correction and completion in 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 ) ) )
...
```

19.1 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>
...
```

19.2 Interactive Nonterminal Expansion

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

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

19.3 Conjunctive Grammars

Many natural and programming languages exhibit context-sensitivity, such as Python indentation. Unlike traditional parser-generators, Tidyparse can encode CFL intersection, allowing it to detect and correct errors in a more expressive family of languages than would ordinarily be possible using CFGs alone. For example, consider the grammar from Sec. 11:

```
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.

20 RELATED WORK

The literature on parsers is vast and deep, covering far more ground than we could possibly hope to survey. We take inspiration from their findings, but restrict ourselves to a dozen most closely related papers, in four major research areas: (1) formal language theory, (2) constraint satisfaction (3) program synthesis, and (4) error correction. We survey each of these areas in turn.

It was Noam Chomsky himself who first developed the algebraic theory of *context-free grammars* (CFGs) in 1959 [11], and since then, CFGs have been the subject of extensive research in formal language theory and program analysis. In particular, we take inspiration from Leslie Valiant [25] who first discovered the connection to matrix multiplication in 1975 and Alexander Okhotin [20] who later introduced the idea of *conjunctive grammars* in 2001. More recently, Azimov & Grigorev [5] and Qirun Zhang shed light into the practical applicability of these ideas and made the theory of CFG reachability more accessible to the general public. In our work, we show how to compile their ideas onto a SAT solver to fix syntax errors, which seems like a perfectly natural extension, but was heretofore previously never considered to the best of our knowledge.

There is related work on string constraint solving in the constraint programing literature, featuring solvers like CFGAnalyzer and HAMPI [17], which consider bounded context free grammars and intersections thereof. Axelson et al. (2008) [4] has some work on incremental SAT encoding but does not exploit the linear-algebraic structure of parsing nor provide real-time guarantees. Finally, Loris D'Antoni did some great work on *symbolic automata* [14], a generalization of finite automata which allow allow infinite alphabets and symbolic expressions over them. In none of the constraint programming literature we surveyed do any of the approaches employ matrix-based parsing, and therefor do not enjoy the optimality guarantees of Valiant's parser. Our solver can handle CFGs and conjunctive grammars with finite alphabets and does not require any special grammar encoding. The matrix encoding makes it particularly amenable to parallelization.

In program synthesis, we direct our attention to the problem of *incremental synthesis* or *error correction*, which is the problem of synthesizing programs incrementally while the user is typing. Modern research on error correction can be traced back to the early days of coding theory, when researchers designed *error-correcting codes* (ECCs) to denoise transmission errors induced by external interference, whether due to collision with a high-energy proton, manipulation by an adversary or some typographical mistake. In this context, *code* can be any logical representation for communicating information between two parties (such as a human and a computer), and an ECC is a carefully-designed code which ensures that even if some portion of the message should be corrupted through accidental or intentional means, one can still recover the original message by solving a linear system of equations. In particular, we frame our work inside the context of errors arising from human factors in computer programming.

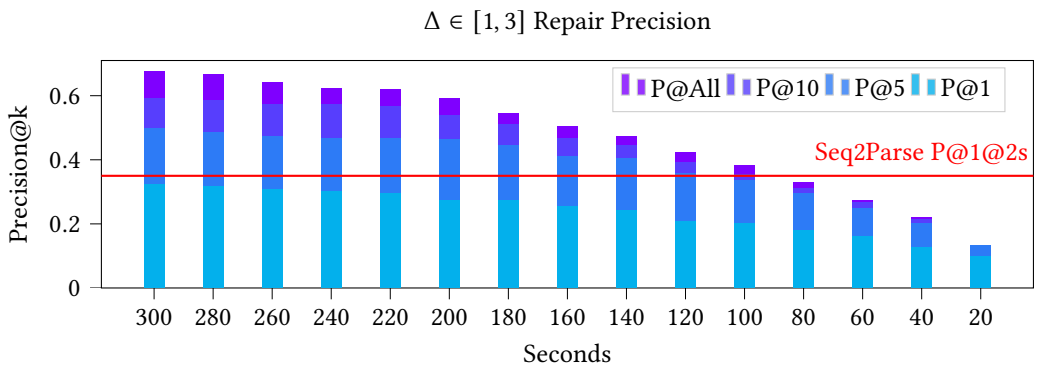
In programming, most such errors initially manifest as syntax errors, and though often cosmetic, manual repair can present a significant challenge for novice programmers. The ECC problem may be refined by introducing a language, $\mathcal{L} \subset \Sigma^*$ and considering admissible edits transforming an arbitrary string, $s \in \Sigma^*$ into a string, $s' \in \mathcal{L}$. Known as *error-correcting parsing* (ECP), this problem was well-studied in the early parsing literature by Aho and Peterson [2], 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 [7], suggesting its tractability. Previous LED results were primarily of a theoretical nature, but now, thanks to our contributions, we have finally realized a practical prototype.

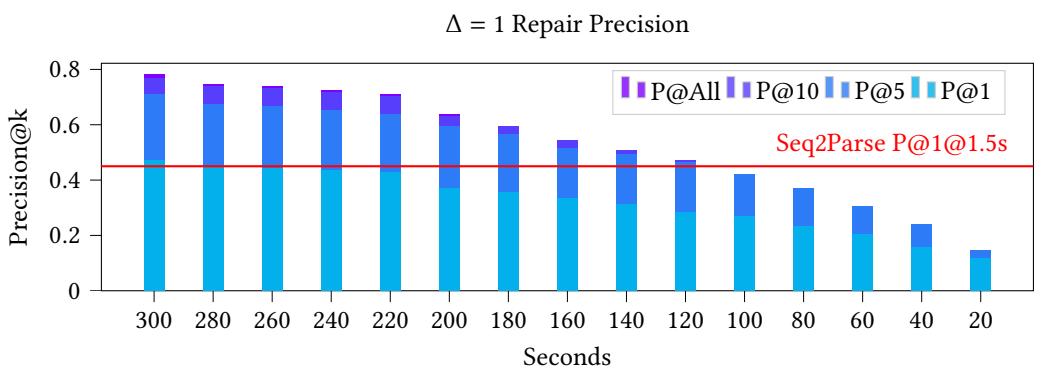
In our work, we recast the problem of ECP as a special case of tensor completion with an logical semiring, and lay the foundation for a new approach to program repair grounded in formal language theory, unifying *code completion*, *error correction* and *incremental parsing*. We provide exact and

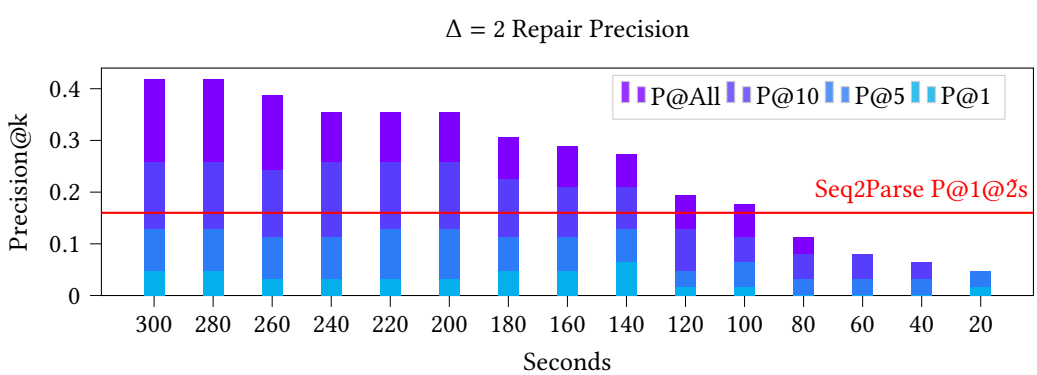
approximate algorithms for solving these problems in the real-world setting, and implement them in a real-time editor called Tidyparse, demonstrating their practical utility. Given a well-formed grammar, our tool can be used to complete unfinished code, parse incomplete code and repair broken fragments in arbitrary context-free and linear conjunctive languages.

21 HUMAN REPAIR BENCHMARK

Below, we plot the results of a human repair benchmark measuring the Precision@k of our repair procedure against human repairs of varying edit distances and latency cutoffs.







For comparison below are the results from Seq2Parse on the same dataset. Seq2Parse only supports Precision@1 repairs, and so we only report precision@1 from the StackOverflow benchmark for comparison. Unlike our approach which only produces syntactically correct repairs, Seq2Parse also produces syntactically incorrect repairs and so we report the percentage of syntactically correct repairs (Syntactic), as well as the precision of the abstract tokens repairs (AbstractEval), and the exact character match precision (CharMatch).

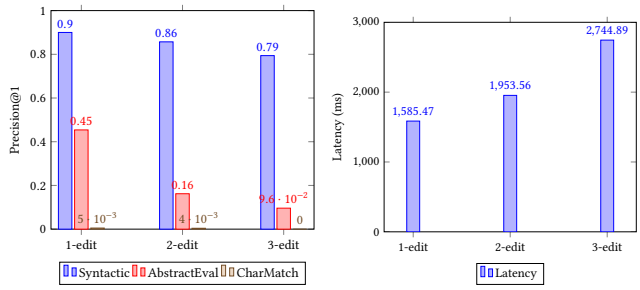
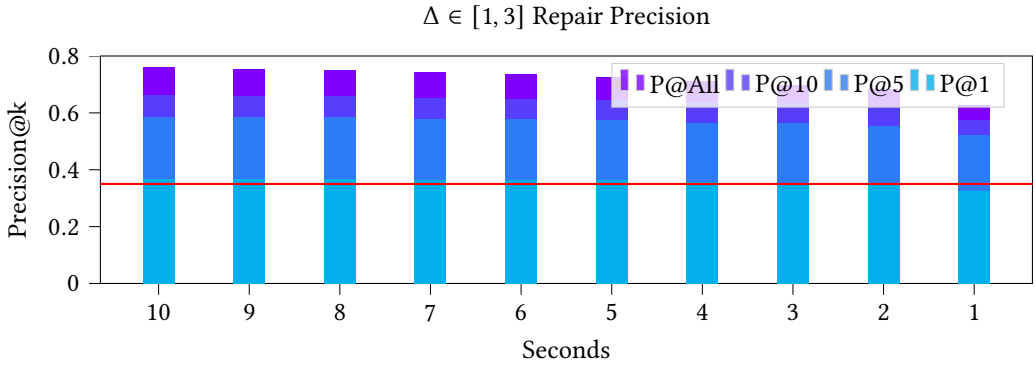
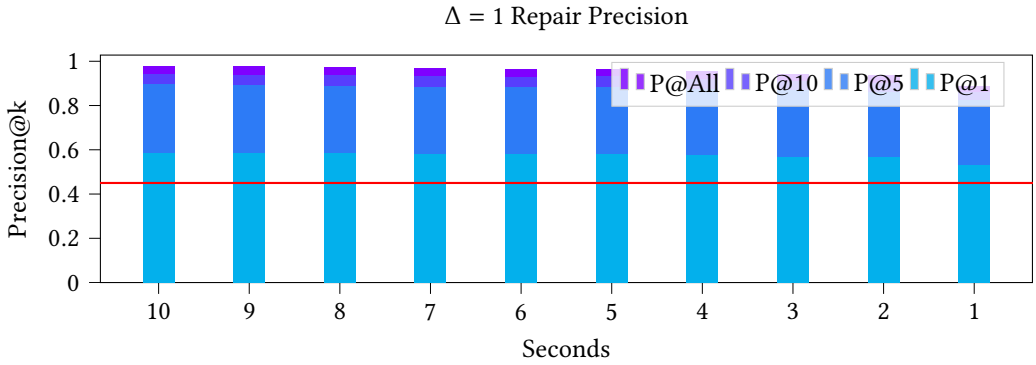
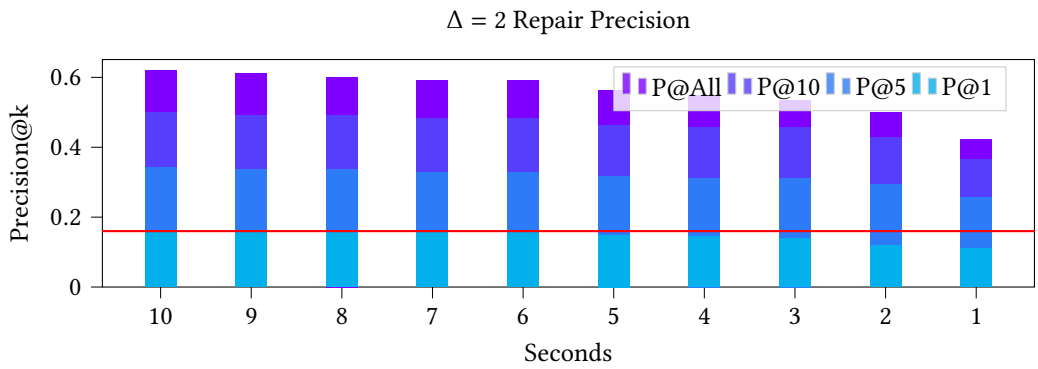


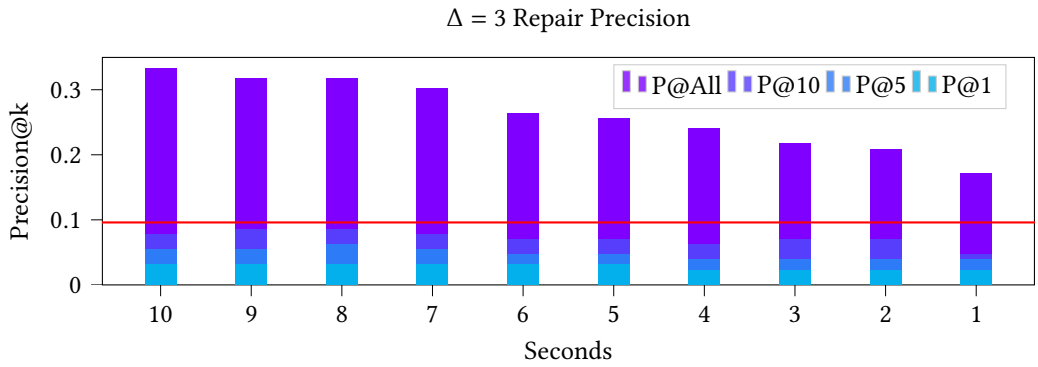
Fig. 9. Seq2Parse precision@1 and latency on the StackOverflow dataset.

22 REPAIR WITH ADAPTIVE SAMPLER







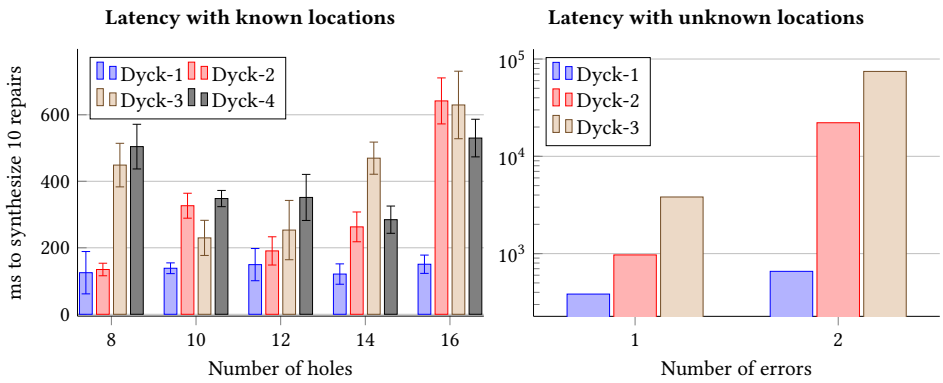


23 LATENCY BENCHMARK

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 different types of balanced parentheses.

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

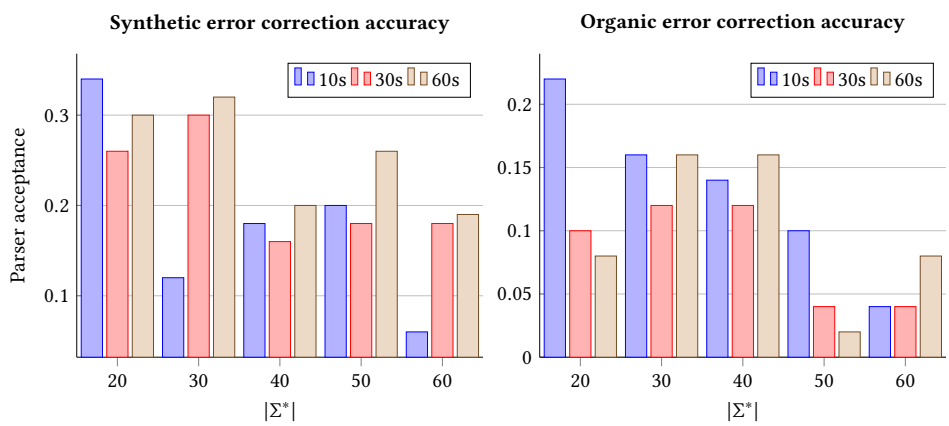
In the first experiment, we sample a random valid string $\sigma \sim \Sigma^{50} \cap \mathcal{L}_{\text{Dyck-n}}$, then replace a fixed number of tokens with holes and measure the average time required to decode ten syntactically-admissible repairs across 100 trial runs.



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 logarithmic scale on the y-axis.

24 ACCURACY BENCHMARK

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



For Python, we sample invalid code fragments uniformly from the imbalanced bracket category of the Break-It-Fix-It (BIFI) dataset [27], 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.

[2]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.

25 DISCUSSION

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

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

Secondly, our current method generates sketch templates using a naïve enumerative search, feeding them individually to the SAT solver, which has the tendency to duplicate prior work and introduces unnecessary thrashing. Considering recent extensions of Boolean matrix-based parsing to linear context-free rewriting systems (LCFRS) [12], it may be feasible to search through these edits within the SAT solver, leading to yet unrealized and possibly significant speedups.

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

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

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

26 CONCLUSION

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

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

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

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

Furthermore, we argue it is possible to improve the efficiency of human programmers without sacrificing expressiveness by considering latency to synthesize an acceptable completion. In contrast with program synthesizers that require intermediate programs to be well-formed, our synthesizer is provably sound and complete up to a Levenshtein distance bound, and attempts to minimize total edits, but does not impose any constraints on the code itself being written.

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

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

REFERENCES

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

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

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

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

[5] Rustam Azimov and Semyon Grigorev. 2018. Context-free path querying by matrix multiplication. In Proceedings of the 1st ACM SIGMOD Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA). 1–10.

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

[7] 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.

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

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

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

[11] Noam Chomsky and Marcel P Schützenberger. 1959. The algebraic theory of context-free languages. In Studies in Logic and the Foundations of Mathematics. Vol. 26. Elsevier, 118–161.

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

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

[14] Loris D'Antoni and Margus Veanes. 2014. Minimization of symbolic automata. In Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. 541–553.

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

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

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

[18] 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

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

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

[21] Alexander Okhotin. 2010. Decision problems for language equations. J. Comput. System Sci. 76, 3-4 (2010), 251–266.

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

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

[24] Matthew Szudzik. 2006. An elegant pairing function. In Wolfram Research (ed.) Special NKS 2006 Wolfram Science Conference. 1–12.

[25] 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

[26] Leslie G Valiant. 1979. Completeness classes in algebra. In Proceedings of the eleventh annual ACM symposium on Theory of computing. 249–261. https://dl.acm.org/doi/pdf/10.1145/800135.804419

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

A EXAMPLE REPAIRS

1.a) Original method	1.b) Synonymous variant

`public void flush(int b) {`
`  buffer.write((byte) b);`
`  buffer.compact();`
`}`
	`public void flush(int b) {`
	`  cushion.write((byte) b);`
	`  cushion.compact();`
	`}`

2.a) Multi-masked method	**2.b) Multi-masked variant**

`public void <MASK>(int b) {`
`  buffer.<MASK>((byte) b);`
`  <MASK>.compact();`
`}`
	`public void <MASK>(int b) {`
	`  cushion.<MASK>((byte) b);`
	`  <MASK>.compact();`
	`}`

3.a) Model predictions	**3.b) Model predictions**

`public void output(int b) {`
`  buffer.write((byte) b);`
`  buffer.compact();`
`}`
	`public void append(int b) {`
	`  cushion.add((byte) b);`
	`  cushion.compact();`
	`}`