Repairing Multiline Syntax Errors Using the Levenshtein-Bar-Hillel Construction

Breandan Considine, Jin Guo, Xujie Si

McGill University, Mila IQIA

bre@ndan.co

November 20, 2024

Overview

1. Examples and Motivation
2. Formal Language Theory
3. Bar-Hillel Refinements
4. Left-to-right Decoding
5. Empirical Evaluation

Can you spot the error?

Original code	**Human repair**

`newlist = []`
`i = set([5, 3, 1)]`
`z = set([5, 0, 4)]`	

Can you spot the error?

Original code	**Human repair**

`newlist = []`
`i = set([5, 3, 1)]`
`z = set([5, 0, 4)]`
	`newlist = []`
	`i = set([5, 3, 1])`
	`z = set([5, 0, 4])`

Can you spot the error?

Original code	Human repair

def average(values):
  if values == (1,2,3):
    return (1+2+3)/3
  else if values == (-3,2):
    return (-3+2+8-1)/4	

Can you spot the error?

Original code	**Human repair**

`def average(values):`
`  if values == (1,2,3):`
`    return (1+2+3)/3`
`  else if values == (-3,2):`
`    return (-3+2+8-1)/4`
	`def average(values):`
	`  if values == (1,2,3):`
	`    return (1+2+3)/3`
	`  elif values == (-3,2):`
	`    return (-3+2+8-1)/4`

Can you spot the error?

Original code	**Human repair**

`import Global from Global`
`globalObj = Global()`
`print(str(globalObj.Test()))`	

Can you spot the error?

Original code	**Human repair**

`import Global from Global`
`globalObj = Global()`
`print(str(globalObj.Test()))`
	`from Global import Global`
	`globalObj = Global()`
	`print(str(globalObj.Test()))`

How many repairs could there possibly be?

Consider the following Python snippet, which contains a small syntax error:

```
def prepend(i, k, L=[])
  n and [prepend(i - 1, k, [b] + L) for b in range(k)]
```

How many repairs could there possibly be?

Consider the following Python snippet, which contains a small syntax error:

```
def prepend(i, k, L=[])
  n and [prepend(i - 1, k, [b] + L) for b in range(k)]
```

It can be fixed by appending a colon after the function signature, yielding:

```
def prepend(i, k, L=[]):
  n and [prepend(i - 1, k, [b] + L) for b in range(k)]
```

How many repairs could there possibly be?

Consider the following Python snippet, which contains a small syntax error:

```
def prepend(i, k, L=[])
  n and [prepend(i - 1, k, [b] + L) for b in range(k)]
```

It can be fixed by appending a colon after the function signature, yielding:

```
def prepend(i, k, L=[]):
  n and [prepend(i - 1, k, [b] + L) for b in range(k)]
```

Let us consider a slightly more ambiguous error: `v = df.iloc(5:, 2:)`. Assuming an alphabet of just a hundred lexical tokens, this statement has millions of two-token edits, yet only six are accepted by the Python parser:

(1) `v = df.iloc(5:, 2,)`

(2) `v = df.iloc(5), 2()`

(3) `v = df.iloc(5[:, 2:])`

(4) `v = df.iloc(5:, 2:)`

(5) `v = df.iloc[5:, 2:]`

(6) `v = df.iloc(5[:, 2])`

High-level architecture overview

This process can be depicted as series of staged transformations lowering the CFL intersection problem onto a finite automaton. Below, we consider a simplified version based on the language of balanced parentheses.

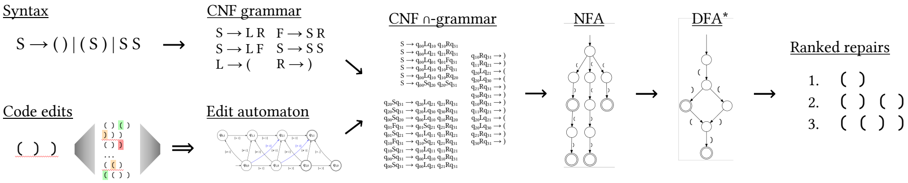
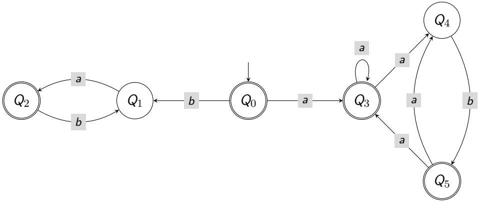

Syntax
$S \rightarrow () \mid (S) \mid S\,S$
CNF grammar
$S \rightarrow L\,R \quad F \rightarrow S\,R$
$S \rightarrow L\,F \quad S \rightarrow S\,S$
$L \rightarrow (\quad R \rightarrow)$
Code edits
( ))
Edit automaton
CNF ∩-grammar
$S \rightarrow q_{00}Lq_{10}\ q_{10}Rq_{31}$
$S \rightarrow q_{00}Lq_{21}\ q_{21}Rq_{31}$
$S \rightarrow q_{00}Lq_{01}\ q_{01}Fq_{31}$
$S \rightarrow q_{00}Lq_{10}\ q_{10}Fq_{31}$
$S \rightarrow q_{00}Lq_{10}\ q_{10}Rq_{20}$
$S \rightarrow q_{00}Sq_{20}\ q_{20}Sq_{31}$
$q_{20}Sq_{31} \rightarrow q_{20}Lq_{21}\ q_{21}Rq_{31}$
$q_{20}Sq_{31} \rightarrow q_{20}Lq_{30}\ q_{30}Rq_{31}$
$q_{00}Sq_{20} \rightarrow q_{00}Lq_{10}\ q_{10}Rq_{20}$
$q_{01}Fq_{31} \rightarrow q_{01}Sq_{21}\ q_{21}Rq_{31}$
$q_{01}Sq_{21} \rightarrow q_{01}Lq_{11}\ q_{11}Rq_{21}$
$q_{10}Fq_{31} \rightarrow q_{10}Sq_{21}\ q_{21}Rq_{31}$
$q_{10}Sq_{21} \rightarrow q_{10}Lq_{11}\ q_{11}Rq_{21}$
$q_{00}Sq_{31} \rightarrow q_{00}Lq_{10}\ q_{10}Rq_{31}$
$q_{00}Sq_{31} \rightarrow q_{00}Lq_{21}\ q_{21}Rq_{31}$
$q_{10}Rq_{31} \rightarrow)$
$q_{11}Rq_{21} \rightarrow)$
$q_{20}Lq_{21} \rightarrow ($
$q_{20}Lq_{30} \rightarrow ($
$q_{21}Rq_{31} \rightarrow)$
$q_{30}Rq_{31} \rightarrow)$
$q_{10}Rq_{31} \rightarrow)$
$q_{11}Rq_{21} \rightarrow)$
$q_{20}Lq_{21} \rightarrow ($
$q_{20}Lq_{30} \rightarrow ($
$q_{21}Rq_{31} \rightarrow)$
$q_{30}Rq_{31} \rightarrow)$
NFA
DFA*
Ranked repairs
1. ( )
2. ( ) ( )
3. (( ))

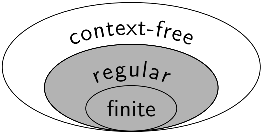
Figure: Simplified dataflow of the language intersection pipeline. Given a grammar and broken code fragment, we (1) create a automaton generating the language of small edits, then (2) construct a grammar representing the intersection of the two languages. This grammar can be (3) converted into a finite automaton, (4) determinized, then (5) decoded to produce a list of repairs.

Background: Regular grammars

A regular grammar (RG) is a quadruple $\mathcal{G} = \langle V, \Sigma, P, S \rangle$ where V are nonterminals, Σ are terminals, $P : V \times (V \cup \Sigma)^{\leq 2}$ are the productions, and $S \in V$ is the start symbol, i.e., all productions are of the form $A \to a$, $A \to aB$ (right-regular), or $A \to Ba$ (left-regular). e.g., the following RG and NFA correspond to the language defined by the *regex* `(a(ab)*)*(ba)*`:

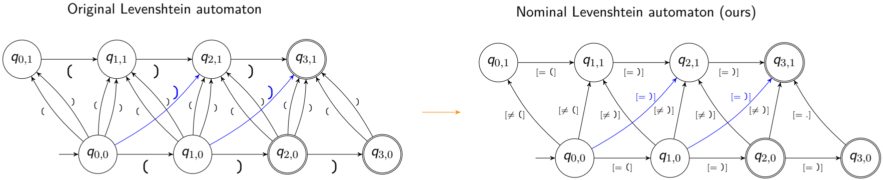
$$\begin{aligned}
S &\to Q_0 \mid Q_2 \mid Q_3 \mid Q_5 \\
Q_0 &\to \varepsilon \\
Q_1 &\to Q_0 b \mid Q_2 b \\
Q_2 &\to Q_1 a \\
Q_3 &\to Q_0 a \mid Q_3 a \mid Q_5 a \\
Q_4 &\to Q_3 a \mid Q_5 a \\
Q_5 &\to Q_4 b
\end{aligned}$$





Levenshtein automaton customization

Consider the string $\underset{\sim}{\sigma} =$ `( ) )` and the alphabet $\Sigma = \{\texttt{)}, \texttt{(}\}$. Every string within one edit of $\underset{\sim}{\sigma}$ is recognized by an NFA with the following structure:



Original Levenshtein automaton

Nominal Levenshtein automaton (ours)

Figure: Automaton recognizing every single patch of the broken string `( ) )` within Levenshtein distance 1. We nominalize the original Levenshtein automaton, ensuring upward arcs denote a mutation, and replace terminals with a symbolic predicate, which deduplicates parallel arcs in large alphabets.

`https://fulmicoton.com/posts/levenshtein/#observations-lets-count-states`

Levenshtein reachability

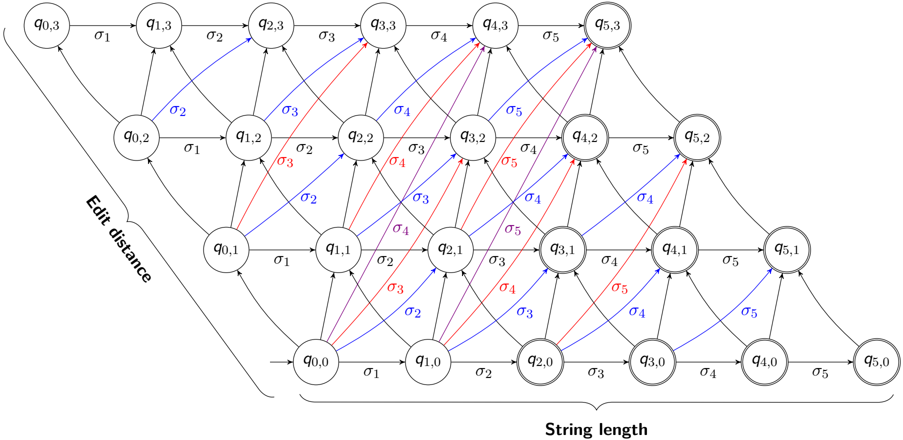


Figure: Bounded Levenshtein reachability from $\sigma : \Sigma^n$ is expressible as an NFA populated by accept states within radius k of $S = q_{n,0}$, which accepts all strings σ' within Levenshtein radius k of σ.

The nominal Levenshtein automaton

The original Levenshtein automaton (Schulz & Mihov, 2002):

$$\frac{s \in \Sigma \quad i \in [0, n] \quad j \in [1, k]}{(q_{i,j-1} \overset{s}{\rightarrow} q_{i,j}) \in \delta} \uparrow \qquad \frac{s \in \Sigma \quad i \in [1, n] \quad j \in [1, k]}{(q_{i-1,j-1} \overset{s}{\rightarrow} q_{i,j}) \in \delta} \nearrow$$

$$\frac{s = \sigma_i \quad i \in [1, n] \quad j \in [0, k]}{(q_{i-1,j} \overset{s}{\rightarrow} q_{i,j}) \in \delta} \rightarrow \qquad \frac{s = \sigma_i \quad i \in [2, n] \quad j \in [1, k]}{(q_{i-2,j-1} \overset{s}{\rightarrow} q_{i,j}) \in \delta} \nearrow$$

$$\frac{}{q_{0,0} \in I} \textsc{Init} \qquad \frac{q_{i,j} \qquad |n - i + j| \leq k}{q_{i,j} \in F} \textsc{Done}$$

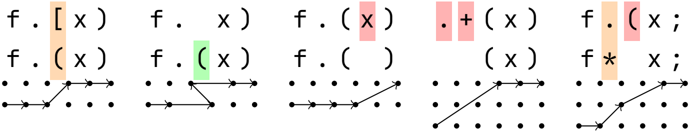
We modify the original automaton with a nominal predicate:

$$\frac{i \in [0, n] \quad j \in [1, k]}{(q_{i,j-1} \overset{[\neq\sigma_{i+1}]}{\rightarrow} q_{i,j}) \in \delta} \uparrow \qquad \frac{i \in [1, n] \quad j \in [1, k]}{(q_{i-1,j-1} \overset{[\neq\sigma_i]}{\rightarrow} q_{i,j}) \in \delta} \nearrow$$

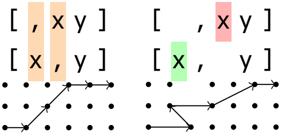
$$\frac{i \in [1, n] \quad j \in [0, k]}{(q_{i-1,j} \overset{[=\sigma_i]}{\rightarrow} q_{i,j}) \in \delta} \rightarrow \qquad \frac{d \in [1, d_{\max}] \quad i \in [d + 1, n] \quad j \in [d, k]}{(q_{i-d-1,j-d} \overset{[=\sigma_i]}{\rightarrow} q_{i,j}) \in \delta} \nearrow$$

Geometrically interpreting the edit calculus

Each arc plays a specific role. $\uparrow$ handles insertions, $\nearrow$ handles substitutions and $\nearrow$ handles deletions of ≥ 1 tokens. Consider some illustrative cases:



Note that the same $\langle \underset{\sim}{\sigma}, \sigma' \rangle$ pair can have multiple Levenshtein alignments:



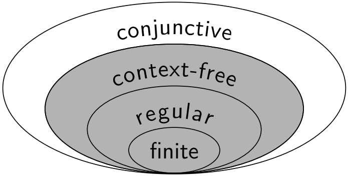
Non-uniqueness of geodesics has implications for CFG ∩ L-NFA ambiguity.

Background: Context-free grammars

In a context-free grammar $\mathcal{G} = \langle V, \Sigma, P, S\rangle$ all productions are of the form $P : V \times (V \cup \Sigma)^+$, i.e., RHS may contain any number of nonterminals, V. Recognition decidable in $\mathcal{O}(n^\omega)$, n.b. CFLs are **not** closed under $\cap$!

For example, consider the grammar $\underline{S \to SS \mid (S) \mid ()}$. This represents the language of balanced parentheses, e.g. $(), ()(), (()), ()(()), (()()), (())() \ldots$

Every CFG has a normal form $P^* : V \times (V^2 \mid \Sigma)$, i.e., every production can be refactored into either $v_0 \to v_1 v_2$ or $v_0 \to \sigma$, where $v_{0\ldots2} : V$ and $\sigma : \Sigma$, e.g., $\{S \to SS \mid (S) \mid ()\} \Leftrightarrow^* \{S \to XR \mid SS \mid LR, L \to (, R \to), X \to LS\}$



Parsing for linear algebraists

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

$$s_1 \otimes s_2 \coloneqq \{C \mid \langle A, B \rangle \in s_1 \times s_2, (C \to AB) \in P\}$$

e.g., $\{\text{A} \to \text{BC, C} \to \text{AD, D} \to \text{BA}\} \subseteq \text{P} \vdash \{\text{A, B, C}\} \otimes \{\text{B, C, D}\} = \{\text{A, C}\}$

Parsing for linear algebraists

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

$$s_1 \otimes s_2 := \{C \mid \langle A, B\rangle \in s_1 \times s_2, (C \to AB) \in P\}$$

e.g., $\{\text{A} \to \text{BC, C} \to \text{AD, D} \to \text{BA}\} \subseteq \text{P} \vdash \{\text{A, B, C}\} \otimes \{\text{B, C, D}\} = \{\text{A, C}\}$

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

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

$S \Rightarrow^* \sigma \iff \sigma \in \mathcal{L}(\mathcal{G})$ iff $S \in \Lambda_\sigma^*$, i.e., $\mathbb{1}_{\Lambda_\sigma^*}(S) \iff \mathbb{1}_{\mathcal{L}(\mathcal{G})}(\sigma)$.

Satisfiability + holes

Let us consider an example with two holes, $\sigma = 1\ \_\ \_$, and the grammar being $G \coloneqq \{S \rightarrow NON, O \rightarrow + \mid \times, N \rightarrow 0 \mid 1\}$. This can be rewritten into CNF as $G' \coloneqq \{S \rightarrow NL, N \rightarrow 0 \mid 1, O \rightarrow \times \mid +, L \rightarrow ON\}$. Using the algebra where $\oplus = \cup$, $X \otimes Z = \left\{ w \mid \langle x, z \rangle \in X \times Z, (w \rightarrow xz) \in P \right\}$, the fixpoint $M' = M + M^2$ can be computed as follows:

Satisfiability + holes

Let us consider an example with two holes, $\sigma = 1\ \_\ \_$, and the grammar being $G \coloneqq \{S \to NON, O \to + \mid \times, N \to 0 \mid 1\}$. This can be rewritten into CNF as $G' \coloneqq \{S \to NL, N \to 0 \mid 1, O \to \times \mid +, L \to ON\}$. Using the algebra where $\oplus = \cup$, $X \otimes Z = \left\{\ w \mid \langle x, z\rangle \in X \times Z, (w \to xz) \in P\ \right\}$, the fixpoint $M' = M + M^2$ can be computed as follows:

	2^V	$\mathbb{Z}_2^{\lvert V\rvert}$	$\mathbb{Z}_2^{\lvert V\rvert} \to \mathbb{Z}_2^{\lvert V\rvert}$
M_0:	$\begin{pmatrix} & \{N\} & & \\ & & \{N,O\} & \\ & & & \{N,O\} \\ & & & \end{pmatrix}$	$\begin{pmatrix} & \square\blacksquare\square\square & & \\ & & \square\blacksquare\blacksquare\square & \\ & & & \square\blacksquare\blacksquare\square \\ & & & \end{pmatrix}$	$\begin{pmatrix} & V_{0,1} & & \\ & & V_{1,2} & \\ & & & V_{2,3} \\ & & & \end{pmatrix}$

Satisfiability + holes

Let us consider an example with two holes, $\sigma = 1\ \_\ \_$, and the grammar being $G \coloneqq \{S \rightarrow NON, O \rightarrow + \mid \times, N \rightarrow 0 \mid 1\}$. This can be rewritten into CNF as $G' \coloneqq \{S \rightarrow NL, N \rightarrow 0 \mid 1, O \rightarrow \times \mid +, L \rightarrow ON\}$. Using the algebra where $\oplus = \cup$, $X \otimes Z = \left\{\ w \mid \langle x, z\rangle \in X \times Z, (w \rightarrow xz) \in P\ \right\}$, the fixpoint $M' = M + M^2$ can be computed as follows:

	2^V	$\mathbb{Z}_2^{\lvert V\rvert}$	$\mathbb{Z}_2^{\lvert V\rvert} \rightarrow \mathbb{Z}_2^{\lvert V\rvert}$
M_0:	$\begin{pmatrix} & \{N\} & & \\ & & \{N,O\} & \\ & & & \{N,O\} \\ & & & \end{pmatrix}$	$\begin{pmatrix} & \square\blacksquare\square\square & & \\ & & \square\blacksquare\blacksquare\square & \\ & & & \square\blacksquare\blacksquare\square \\ & & & \end{pmatrix}$	$\begin{pmatrix} & V_{0,1} & & \\ & & V_{1,2} & \\ & & & V_{2,3} \\ & & & \end{pmatrix}$
M_1:	$\begin{pmatrix} & \{N\} & \varnothing & \\ & & \{N,O\} & \{L\} \\ & & & \{N,O\} \\ & & & \end{pmatrix}$	$\begin{pmatrix} & \square\blacksquare\square\square & \square\square\square\square & \\ & & \square\blacksquare\blacksquare\square & \blacksquare\square\square\square \\ & & & \square\blacksquare\blacksquare\square \\ & & & \end{pmatrix}$	$\begin{pmatrix} & V_{0,1} & V_{0,2} & \\ & & V_{1,2} & V_{1,3} \\ & & & V_{2,3} \\ & & & \end{pmatrix}$

Satisfiability + holes

Let us consider an example with two holes, $\sigma = 1\ \_\ \_$, and the grammar being $G := \{S \to NON, O \to + \mid \times, N \to 0 \mid 1\}$. This can be rewritten into CNF as $G' := \{S \to NL, N \to 0 \mid 1, O \to \times \mid +, L \to ON\}$. Using the algebra where $\oplus = \cup$, $X \otimes Z = \left\{\, w \mid \langle x, z \rangle \in X \times Z, (w \to xz) \in P \,\right\}$, the fixpoint $M' = M + M^2$ can be computed as follows:

	2^V	$\mathbb{Z}_2^{\lvert V \rvert}$	$\mathbb{Z}_2^{\lvert V \rvert} \to \mathbb{Z}_2^{\lvert V \rvert}$
M_0:	$\begin{pmatrix} \{N\} & & \\ & \{N, O\} & \\ & & \{N, O\} \end{pmatrix}$	$\begin{pmatrix} \square\blacksquare\square\square & & \\ & \square\blacksquare\blacksquare\square & \\ & & \square\blacksquare\blacksquare\square \end{pmatrix}$	$\begin{pmatrix} V_{0,1} & & \\ & V_{1,2} & \\ & & V_{2,3} \end{pmatrix}$
M_1:	$\begin{pmatrix} \{N\} & \varnothing & \\ & \{N, O\} & \{L\} \\ & & \{N, O\} \end{pmatrix}$	$\begin{pmatrix} \square\blacksquare\square\square & \square\square\square\square & \\ & \square\blacksquare\blacksquare\square & \blacksquare\square\square\square \\ & & \square\blacksquare\blacksquare\square \end{pmatrix}$	$\begin{pmatrix} V_{0,1} & V_{0,2} & \\ & V_{1,2} & V_{1,3} \\ & & V_{2,3} \end{pmatrix}$
M_∞:	$\begin{pmatrix} \{N\} & \varnothing & \{S\} \\ & \{N, O\} & \{L\} \\ & & \{N, O\} \end{pmatrix}$	$\begin{pmatrix} \square\blacksquare\square\square & \square\square\square\square & \square\square\square\blacksquare \\ & \square\blacksquare\blacksquare\square & \blacksquare\square\square\square \\ & & \square\blacksquare\blacksquare\square \end{pmatrix}$	$\begin{pmatrix} V_{0,1} & V_{0,2} & V_{0,3} \\ & V_{1,2} & V_{1,3} \\ & & V_{2,3} \end{pmatrix}$

Background: Closure properties of formal languages

Formal languages are not always closed under set-theoretic operations, e.g., CFL $\cap$ CFL is not CFL in general. Let $\cdot$ denote concatenation, $*$ be Kleene star, and $\complement$ be complementation:

	$\cup$	$\cap$	$\cdot$	$*$	$\complement$
Finite[1]	✓	✓	✓	✓	✓
Regular[1]	✓	✓	✓	✓	✓
Context-free[1,2]	✓	✗[†]	✓	✓	✗
Conjunctive[1,2]	✓	✓	✓	✓	?
Context-sensitive[2]	✓	✓	✓	$+$	✓
Recursively Enumerable[2]	✓	✓	✓	✓	✗

We would like a language family that is (1) tractable, i.e., has polynomial recognition and search complexity and (2) reasonably expressive, i.e., can represent syntactic properties of real-world programming languages.

[†] However, CFLs are closed under intersection with regular languages.

The Bar-Hillel construction and its specialization

Theorem (Bar-Hillel, 1961)

Given a CFG, G, and an NFA, A, there exists a $G_\cap = \langle V_\cap, \Sigma_\cap, P_\cap, S\rangle$, such that $\mathcal{L}(G_\cap) = \mathcal{L}(G) \cap \mathcal{L}(A)$.

The Bar-Hillel construction and its specialization

Theorem (Bar-Hillel, 1961)

Given a CFG, G, and an NFA, A, there exists a $G_\cap = \langle V_\cap, \Sigma_\cap, P_\cap, S\rangle$, such that $\mathcal{L}(G_\cap) = \mathcal{L}(G) \cap \mathcal{L}(A)$.

Salomaa (1973) constructs the intersection grammar as follows:

$$\frac{q \in I \quad r \in F}{\big(S \to qSr\big) \in P_\cap}\sqrt{} \qquad \frac{(A \to a) \in P \qquad (q \overset{a}{\to} r) \in \delta}{\big(qAr \to a\big) \in P_\cap}\uparrow$$

$$\frac{(w \to xz) \in P \qquad p, q, r \in Q}{\big(pwr \to (pxq)(qzr)\big) \in P_\cap}\bowtie$$

The Bar-Hillel construction and its specialization

Theorem (Bar-Hillel, 1961)

Given a CFG, G, and an NFA, A, there exists a $G_\cap = \langle V_\cap, \Sigma_\cap, P_\cap, S\rangle$, such that $\mathcal{L}(G_\cap) = \mathcal{L}(G) \cap \mathcal{L}(A)$.

Salomaa (1973) constructs the intersection grammar as follows:

$$\frac{q \in I \quad r \in F}{(S \to qSr) \in P_\cap}\sqrt{} \qquad \frac{(A \to a) \in P \qquad (q \overset{a}{\to} r) \in \delta}{(qAr \to a) \in P_\cap}\uparrow$$

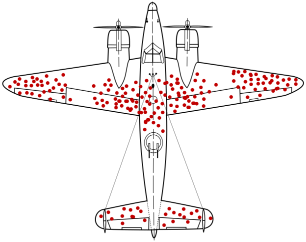
$$\frac{(w \to xz) \in P \qquad p, q, r \in Q}{\big(pwr \to (pxq)(qzr)\big) \in P_\cap}\bowtie$$

Observation: too many (q, v, q') triples! Three low-hanging optimizations:

1. Only consider (q, q') where $\exists \sigma : \Sigma^*$ s.t. $q \overset{\sigma}{\Longrightarrow} q'$.
2. Filter impossible (q, v, q') triples based on path length.
3. Remove unreachable states q', i.e., $\nexists \sigma \in \mathcal{L}(G)$ s.t., $q_0 \overset{\sigma}{\Longrightarrow} q'$.

Edit location refinement: intuition

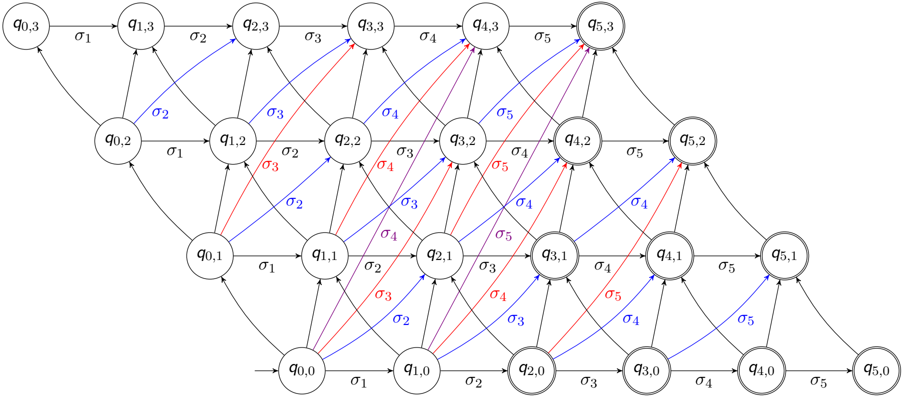
Let's think: do we really need $|\sigma| \times d_{\max}$ states? Can we somehow narrow down the edit range? Where can we remove dead weight?



It is typically easier to determine where *not* to make the edits. Certain regions, no matter their contents, will never yield a viable repair.

Edit location refinement: example

Let's prune states absorbing impossible repair trajectories!

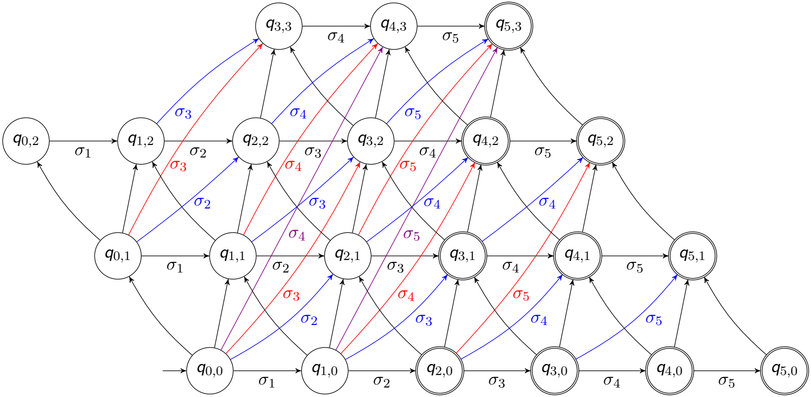

$q_{0,3}$ $q_{1,3}$ $q_{2,3}$ $q_{3,3}$ $q_{4,3}$ $q_{5,3}$
$q_{0,2}$ $q_{1,2}$ $q_{2,2}$ $q_{3,2}$ $q_{4,2}$ $q_{5,2}$
$q_{0,1}$ $q_{1,1}$ $q_{2,1}$ $q_{3,1}$ $q_{4,1}$ $q_{5,1}$
$q_{0,0}$ $q_{1,0}$ $q_{2,0}$ $q_{3,0}$ $q_{4,0}$ $q_{5,0}$
σ_1 σ_2 σ_3 σ_4 σ_5
σ_2 σ_3 σ_4 σ_5
σ_1 σ_2 σ_3 σ_4 σ_5
σ_3 σ_4 σ_5
σ_2 σ_3 σ_4 σ_4
σ_4 σ_5
σ_1 σ_2 σ_3 σ_4 σ_5
σ_3 σ_4 σ_5
σ_2 σ_3 σ_4 σ_5
σ_1 σ_2 σ_3 σ_4 σ_5

G: S $\rightarrow$ (S) |
[S] |
S + S | 1

σ: [(+)] ✗

Edit location refinement: example

Let's prune states absorbing impossible repair trajectories!

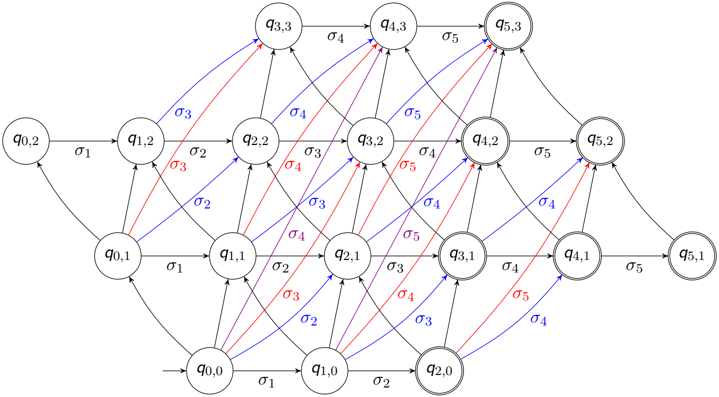

$q_{3,3}$ σ_4 $q_{4,3}$ σ_5 $q_{5,3}$
σ_3 σ_4 σ_5
$q_{0,2}$ σ_1 $q_{1,2}$ σ_2 $q_{2,2}$ σ_3 $q_{3,2}$ σ_4 $q_{4,2}$ σ_5 $q_{5,2}$
σ_3 σ_4 σ_5
σ_2 σ_3 σ_4 σ_4
σ_4 σ_5
$q_{0,1}$ σ_1 $q_{1,1}$ σ_2 $q_{2,1}$ σ_3 $q_{3,1}$ σ_4 $q_{4,1}$ σ_5 $q_{5,1}$
σ_3 σ_4 σ_5
σ_2 σ_3 σ_4 σ_5
$q_{0,0}$ σ_1 $q_{1,0}$ σ_2 $q_{2,0}$ σ_3 $q_{3,0}$ σ_4 $q_{4,0}$ σ_5 $q_{5,0}$

G: S $\rightarrow$ (S) |
[S] |
S + S | 1

σ: [(+)] ❌
_ _ +)] ❌ $\wedge$ _ _ _)] ✅

Edit location refinement: example

Let's prune states absorbing impossible repair trajectories!



G: S → (S) |
[S] |
S + S | 1

σ: [(+)] ✗
_ _ +)] ✗ ∧ _ _ _)] ✅
[(+ _ _ ✗ ∧ [(_ _ _ ✅

Grammar refinement: Parikh interval maps

If we're reusing the CFG, it makes sense to precompute some statistics.

Definition (Parikh map for nonterminals)

The **Parikh image**, $p : \Sigma^* \rightarrow \mathbb{N}^{|\Sigma|}$, counts the number of times each terminal appears in a string. The **Parikh map**, $\pi : V \times \mathbb{N} \rightarrow \Pi$, is the smallest interval containing the Parikh image across all strings generated by a nonterminal R of length n, i.e, $\forall \sigma \in \mathcal{L}(R) \cap \Sigma^n \vdash p(\sigma) \in \pi(R, n)$.

Grammar refinement: Parikh interval maps

If we're reusing the CFG, it makes sense to precompute some statistics.

Definition (Parikh map for nonterminals)

The **Parikh image**, $p : \Sigma^* \rightarrow \mathbb{N}^{|\Sigma|}$, counts the number of times each terminal appears in a string. The **Parikh map**, $\pi : V \times \mathbb{N} \rightarrow \Pi$, is the smallest interval containing the Parikh image across all strings generated by a nonterminal R of length n, i.e, $\forall \sigma \in \mathcal{L}(R) \cap \Sigma^n \vdash p(\sigma) \in \pi(R, n)$.

Likewise, we can define a Parikh interval for subautomata.

Definition (Parikh map for subautomata)

The **Parikh map** for a subautomaton, $\pi : Q \times Q \rightarrow \Pi$, returns the smallest interval s.t. $\forall \sigma : \Sigma^*, q, q' : Q, q \overset{\sigma}{\Longrightarrow} q' \vdash p(\sigma) \in \pi(q, q')$.

Now when we have a new automaton, we can check whether (q, v, q') is compatible by checking whether the Parikh range for q, q' and v overlaps.

Grammar refinement: Parikh compatibility

Criteria to overapproximate compatible nonterminals and states:

Definition (Parikh compatibility)

We call a nonterminal R and state pair q, q' *compatible* when:

$$\frac{\mathcal{L}(R) \cap \bigcup_{n \in [\![q,q']\!]} \Sigma^n \neq \varnothing \qquad \pi(q, q') \cap \bigcup_{n \in [\![q,q']\!]} \pi(R, n) \neq \varnothing}{R \lhd q, q'} \lhd$$

where $[\![q, q']\!] = [\min|\sigma|, \max|\sigma|]$ s.t. $q \overset{\sigma}{\Longrightarrow} q'$.

Now, we are ready to re[de]fine the original $\bowtie$ rule:

$$\frac{(w \rightarrow xz) \in P \qquad p, q, r \in Q}{\big(pwr \rightarrow (pxq)(qzr)\big) \in P_\cap} \bowtie$$

Grammar refinement: Parikh compatibility

Criteria to overapproximate compatible nonterminals and states:

Definition (Parikh compatibility)

We call a nonterminal R and state pair q, q' *compatible* when:

$$\frac{\mathcal{L}(R) \cap \bigcup_{n \in [\![q,q']\!]} \Sigma^n \neq \varnothing \qquad \pi(q,q') \cap \bigcup_{n \in [\![q,q']\!]} \pi(R,n) \neq \varnothing}{R \lhd q, q'} \lhd$$

where $[\![q, q']\!] = [\min|\sigma|, \max|\sigma|]$ s.t. $q \overset{\sigma}{\Longrightarrow} q'$.

Now, we are ready to re[de]fine the original $\bowtie$ rule:

$$\frac{w \lhd pr \quad x \lhd pq \quad z \lhd qr \qquad (w \rightarrow xz) \in P \qquad p, q, r \in Q}{\big(pwr \rightarrow (pxq)(qzr)\big) \in P_\cap} \hat{\bowtie}$$

So, we have a grammar. Now what?

There still many unnecessary productions, since we over-approximated the grammar. To reduce the number of productions, we normalize $G_\cap \implies G_\cap^*$ to CNF, removing unreachable and unproductive productions.

We could sample repairs from $G_\cap^*$ directly, but there are a few issues here:

1. It is not obvious how to steer a CFG sampler with a LLM.
2. It is nontrivial to implement a parallel, replacement-free CFG sampler while also prioritizing high-likelihood repairs.
3. Even if we could, this would not guarantee repair uniqueness, because the CFG can be ambiguous, so we could end up sampling distinct trees representing the same string.

Now, let's step back and think...

So, we have a grammar. Now what?

There still many unnecessary productions, since we over-approximated the grammar. To reduce the number of productions, we normalize $G_\cap \Longrightarrow G_\cap^*$ to CNF, removing unreachable and unproductive productions.

We could sample repairs from $G_\cap^*$ directly, but there are a few issues here:

1. It is not obvious how to steer a CFG sampler with a LLM.
2. It is nontrivial to implement a parallel, replacement-free CFG sampler while also prioritizing high-likelihood repairs.
3. Even if we could, this would not guarantee repair uniqueness, because the CFG can be ambiguous, so we could end up sampling distinct trees representing the same string.

Now, let's step back and think...the intersection language is finite...

So, we have a grammar. Now what?

There still many unnecessary productions, since we over-approximated the grammar. To reduce the number of productions, we normalize $G_\cap \Longrightarrow G_\cap^*$ to CNF, removing unreachable and unproductive productions.

We could sample repairs from $G_\cap^*$ directly, but there are a few issues here:

1. It is not obvious how to steer a CFG sampler with a LLM.
2. It is nontrivial to implement a parallel, replacement-free CFG sampler while also prioritizing high-likelihood repairs.
3. Even if we could, this would not guarantee repair uniqueness, because the CFG can be ambiguous, so we could end up sampling distinct trees representing the same string.

Now, let's step back and think...the intersection language is finite...
So we should be able to represent it as a DFA.

Potential ambiguity of Levenshtein-Bar-Hillel grammars

The previous technique produces a grammar that exactly generates $G_\cap$, but does not guarantee string uniqueness since the CFG may be ambiguous.

Lemma (Possible ambiguity of LBH grammar)

If the NFA, α, is ambiguous, the intersection CFG, $G_\cap$, can be ambiguous.

Proof.

Let ℓ be the language defined by $G = \{S \rightarrow LR, L \rightarrow \texttt{(}, R \rightarrow \texttt{)}\}$, where $\alpha = L(\underset{\sim}{\sigma}, 2)$, the broken string $\underset{\sim}{\sigma}$ is `)(`, and $\mathcal{L}(G_\cap) = \ell \cap \mathcal{L}(\alpha)$. Then, $\mathcal{L}(G_\cap)$ contains the following two identical repairs: `)()` with the parse $S \rightarrow q_{00}Lq_{21}\ q_{21}Rq_{22}$, and `()` with the parse $S \rightarrow q_{00}Lq_{11}\ q_{11}Rq_{22}$. □

We can eliminate ambiguity and thereby improve the rate of convergence for natural syntax repair by first translating $G_\cap$ into an FSA.

Existence of an FSA that generates $\mathcal{L}(G_\cap)$

There is an FSA generating $\mathcal{L}(G_\cap)$. We first show this non-constructively:

Lemma (Acyclicity of LBH grammar)

The intersection grammar, $G_\cap$, is acyclic.

Proof.

Assume $G_\cap$ is cyclic. Then $\mathcal{L}(G_\cap)$ must be infinite. But since $G_\cap$ generates $\ell \cap \mathcal{L}(\alpha)$ by construction and α is acyclic, $\mathcal{L}(G_\cap)$ is necessarily finite. Therefore, $G_\cap$ must not be cyclic. □

Since $G_\cap$ is acyclic and thus finite, it must be representable as an FSA. Using an FSA for decoding has many advantages, notably, it can be efficiently minimized and decoded in order of n-gram likelihood using a Markov chain or standard pretrained autoregressive language model.

Translating from $G_\cap$ to a DFA

Let $+, * : \mathcal{A} \times \mathcal{A} \to \mathcal{A}$ be automata operators satisfying the property $\mathcal{L}(A_1 + A_2) = \mathcal{L}(A_1) \cup \mathcal{L}(A_2)$, and $\mathcal{L}(A_1 * A_2) = \mathcal{L}(A_1) \times \mathcal{L}(A_2)$. We can translate $G_\cap$ to $\mathcal{A}$, as follows, recalling FSAs are closed over $+, *$:

$$\mathcal{Y}(v : V) \mapsto \begin{cases} \alpha \mid \mathcal{L}(\alpha) = \{s \in \Sigma_\cap \mid v \to s\} & \textit{if v is a unit NT,} \\ \sum_{\langle v_1, v_2 \rangle \mid v \to v_1 v_2} \mathcal{Y}(v_1) * \mathcal{Y}(v_2) & \text{otherwise.} \end{cases}$$

In the case of LBH intersections, $\mathcal{Y}(S)$ yields $\alpha : \mathcal{A} \mid \mathcal{L}(\alpha) = \ell \cap L(\underset{\sim}{\sigma}, d)$, which can be minimized via Brzozowski's algorithm then decoded:

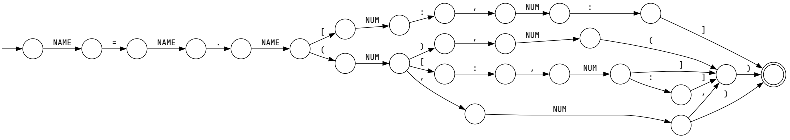


Figure: $L($`NAME = NAME . NAME ( NUM : , NUM : )`$, 2) \cap \ell_{\text{PYTHON}}$

Decoding the DFA

Algorithm Steerable DFA walk

```
Require: $\mathcal{A} = \langle Q, \Sigma, \delta, I, F \rangle$ DFA, $P_\theta : \Sigma^d \to \mathbb{R}$ Markov chain
$\mathcal{T} \leftarrow \varnothing$ total trajectories, $\mathcal{P} \leftarrow \left[\langle \varepsilon, i, 0 \rangle \mid i \in I\right]$ partial trajectories
repeat
    let $\langle \sigma, q, \gamma \rangle = \texttt{head}(\mathcal{P})$ in
        $\mathbf{T} = \left\{\langle s\sigma, q', \gamma - \log P_\theta(s \mid \sigma_{1..d-1}) \rangle \mid (q \overset{s}{\to} q') \in \delta\right\}$
    for $\langle \sigma, q, \gamma \rangle = T \in \mathbf{T}$ do
        $\mathcal{P} \leftarrow \texttt{tail}(\mathcal{P}) \oplus T$                ▷ Add partial trajectory to PQ.
        if $q \in F$ then
            $\mathcal{T} \leftarrow \mathcal{T} \oplus T$          ▷ Accepting state reached, add to queue.
until interrupted or $\mathcal{P} = \varnothing$.
return $\left[\sigma_{|\sigma|..1} \mid \langle \sigma, q, \gamma \rangle = T \in \mathcal{T}\right]$                ▷ Return in sorted order
```

Characteristics of the repair dataset

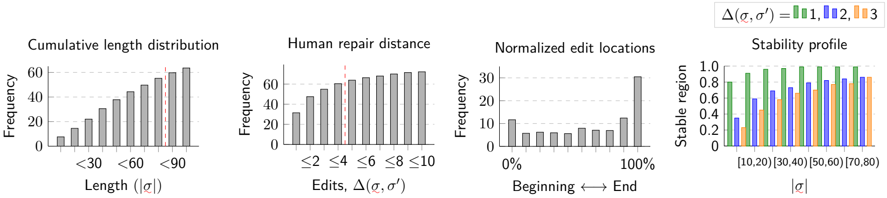


Figure: Repair statistics across the StackOverflow dataset, of which Tidyparse can handle about half in under ∼30s and ∼150 GB. Larger repairs and edit distances are possible, albeit requiring additional time and memory. The stability profile measures the average fraction of all edit locations that were never altered by any repair in the $L\big(\sigma, \Delta(\sigma, \sigma')\big)$-ball across repairs of varying length and distance.

Ranked repair

We train on lexical n-grams using the standard MLE for Markov chains. To score the repairs, we use the conventional length-normalized NLL:

$$\mathsf{NLL}(\sigma) = -\frac{1}{|\sigma|}\sum_{i=1}^{|\sigma|} \log P_\theta(\sigma_i \mid \sigma_{i-1}\dots\sigma_{i-n}) \tag{1}$$

For each retrieved set $\hat{A} \subseteq A$ drawn before a predetermined timeout and each $\sigma \in \hat{A}$, we score the repair and return $\hat{A}$ in ascending order.

To evaluate the quality of our ranking, we use the Precision@k statistic. Specifically, given a repair model, $R : \Sigma^* \to 2^{\Sigma^*}$ and a parallel corpus, $\mathcal{D}_{\text{test}}$, of errors ($\sigma^\dagger$) and repairs ($\sigma'$), we define Precision@k as:

$$\mathsf{Precision@k}(R) = \frac{1}{|\mathcal{D}_{\text{test}}|}\sum_{\langle\sigma^\dagger,\sigma'\rangle\in\mathcal{D}_{\text{test}}} \mathbb{1}\Big[\sigma' \in \operatorname*{argmax}_{\boldsymbol{\sigma}\subset R(\sigma^\dagger),|\boldsymbol{\sigma}|\leq k} \sum_{\sigma\in\boldsymbol{\sigma}} \mathsf{NLL}(\sigma)\Big] \tag{2}$$

Precision and latency comparison

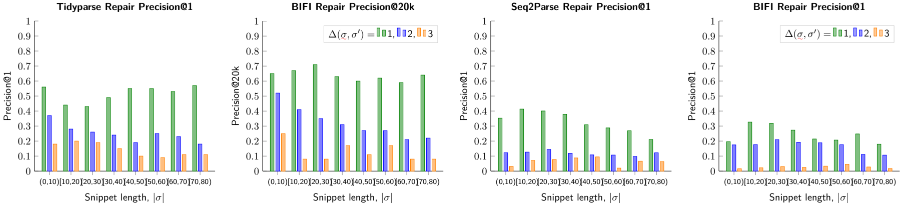


Figure: Tidyparse, Seq2Parse and BIFI repair precision across length and edits.

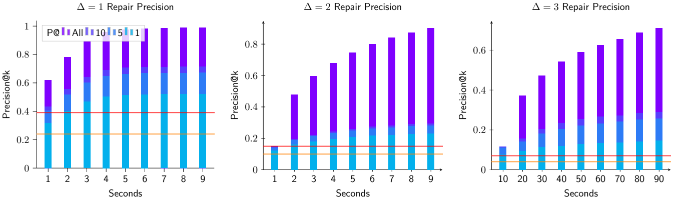


Figure: Latency benchmarks. Note the varying axis ranges. The red line marks Seq2Parse and the orange line marks BIFI's Precision@1 on the same repairs.

Precision and latency comparison

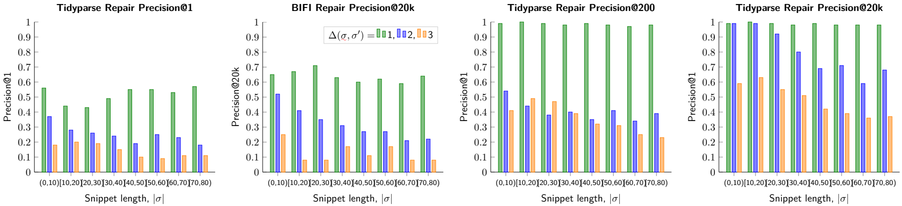


Figure: Tidyparse, Seq2Parse and BIFI repair precision across length and edits.

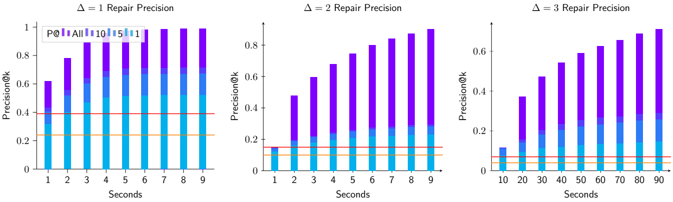
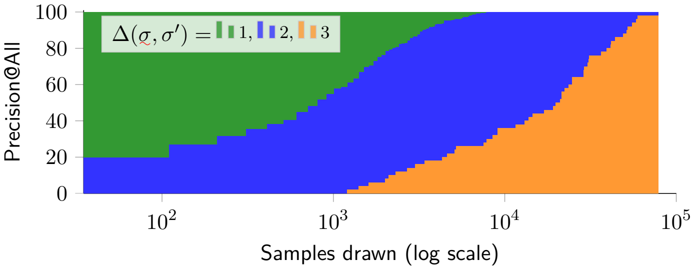
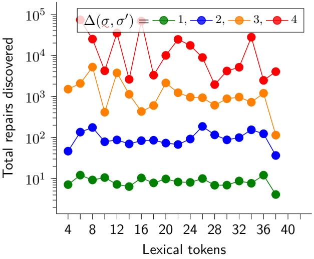


Figure: Latency benchmarks. Note the varying axis ranges. The red line marks Seq2Parse and the orange line marks BIFI's Precision@1 on the same repairs.

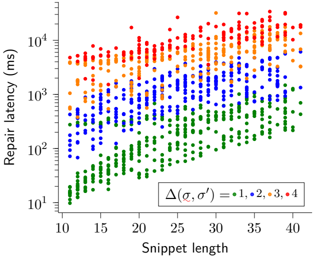
Results from uniform sample efficiency experiments



Uniform Sampling Efficiency



Average throughput



Latency across repair size

Outcomes in the syntax repair pipeline

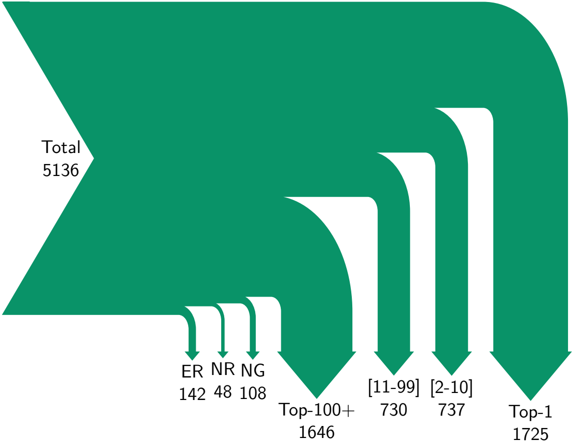


Figure: Sankey diagram of 5136 repair instances sampled uniformly from the StackOverflow Python dataset balanced acrost $(\lfloor|\underset{\sim}{\sigma}|/10\rfloor \in [0,8], \Delta(\underset{\sim}{\sigma}, \sigma') < 4)$. (ER=Error, NR=Not recognized, NG=Not generated). Time: ∼10hr on M1 Max.

Feature comparison matrix

	Sound	**Complete**	**Natural**	**Theory**	$\|\|$	**Tool**
Tidyparse	✓	✓	✓	$\text{CFG}_\cap$	✓	IDE-ready
Seq2Parse	✓[†]	✗	✓	CFG	✗	Python
BIFI	✗	✗	✓	Σ^*	✗	Python
OrdinalFix	✓	✗	✗	CFG+	✗	Rust
Outlines[1]	✓[†]	✓[†]	✓	EBNF	✗	Python
SynCode[1]	✓	✓	✓	EBNF	✗	Python
Aho/Irons	✓	✗	✗	CFG	✗	None

Sound = generated repairs always syntactically valid.
Complete = all valid repairs are eventually generated.
Natural ≈ statistically likely / designed to model human preferability.
$||$ = Trivially parallelizable, i.e., designed to be executed on multiple cores.

[1] Not specifically intended for syntax repair, but can be adapted.

[†] Claimed by the authors, but counterexamples known to exist.

Abbreviated history of algebraic parsing

- Chomsky & Schützenberger (1959) - The algebraic theory of CFLs
- Cocke–Younger–Kasami (1961) - Bottom-up matrix-based parsing
- Brzozowski (1964) - Derivatives of regular expressions
- Valiant (1975) - first realizes the Boolean matrix correspondence
    - Naïvely, has complexity $\mathcal{O}(n^4)$, can be reduced to $\mathcal{O}(n^\omega)$, $\omega < 2.763$
- Lee (1997) - Fast CFG Parsing $\iff$ Fast BMM, formalizes reduction
- Might et al. (2011) - Parsing with derivatives (Brzozowski $\Rightarrow$ CFL)
- Bakinova, Okhotin et al. (2010) - Formal languages over GF(2)
- Bernady & Jansson (2015) - Certifies Valiant (1975) in Agda
- Cohen & Gildea (2016) - Generalizes Valiant (1975) to parse and recognize mildly context sensitive languages, e.g. LCFRS, TAG, CCG
- **Considine, Guo & Si (2022) - SAT + Valiant (1975) + holes**
- **Considine, Guo & Si (2024) - Levenshtein Bar-Hillel repairs**

Future work

Parsing

- Error propagation in discrete dynamical systems and TRS
- Dynamic matrix inverse and incremental transitive closure
- Language Edit Distance with metrizable Boolean semirings
- Unify parser-lexer for scannerless ECP on a real language
- Investigate the feasibility of grammar induction and repair
- Strengthen the connection to Leibnizian differentiability

Probability

- Look into Markov chains (detailed balance, stationarity, reversibility)
- Fuse Valiant parser and probabilistic context-free grammar
- Contextualize belief propagation and graph diffusion processes
- Look into constrained optimization (e.g., L/QP) to rank feasible set

Special thanks

Jin Guo, Xujie Si, David Bieber,
David Chiang, Brigitte Pientka, David Hui,
Ori Roth, Younesse Kaddar, Michael Schröder
Will Chrichton, Kristopher Micinski, Alex Lew
Matthijs Vákár, Michael Coblenz, Maddy Bowers

Learn more at:

https://tidyparse.github.io