Automatic Grammar Repair

Moeketsi Raselimo
Stellenbosch University
Stellenbosch, South Africa
22374604@sun.ac.za

Bernd Fischer
Stellenbosch University
Stellenbosch, South Africa
bfischer@sun.ac.za

Abstract

We describe the first approach to automatically repair bugs in context-free grammars: given a grammar that fails some tests in a given test suite, we iteratively and gradually transform the grammar until it passes all tests. Our core idea is to build on spectrum-based fault localization to identify promising repair sites (i.e., specific positions in rules), and to apply grammar patches at these sites whenever they satisfy explicitly formulated pre-conditions necessary to potentially improve the grammar.

We have implemented this approach in the gfixr system, and successfully used it to fix grammars students submitted as homeworks in a compiler engineering course, and to map one Pascal dialect grammar against another dialect. gfixr can be configured to explore the repair space in different ways, and can also take advantage of counterexamples to enable restriction patches that make the grammar less permissive.

CCS Concepts: • **Software and its engineering** → **Parsers**; *Syntax*; *Software testing and debugging*; • **Theory of computation** → *Grammars and context-free languages.*

Keywords: Program repair, Fault localization.

ACM Reference Format:
Moeketsi Raselimo and Bernd Fischer. 2021. Automatic Grammar Repair. In *Proceedings of the 14th ACM SIGPLAN International Conference on Software Language Engineering (SLE '21), October 17–18, 2021, Chicago, IL, USA.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3486608.3486910

1 Introduction

Grammars are software, and can contain bugs like any other software. This is true even for well-curated grammars. Lämmel and Verhoef [32] found "*more errors than one would expect from a language reference manual*" when analyzing COBOL, and Zaytsev [68] shows errors and inconsistencies in the language specifications of both Java and C#. Grammar testing [31] can demonstrate the presence of such bugs in grammars and grammar fault localization [50] can identify rules that are likely to contain bugs, but neither of the two techniques can automatically *repair* bugs and *fix* grammars.

Manual grammar repair is tedious because developers need to track information about syntax errors back to the grammar, without much feedback from the parser: since the parser assumes that the grammar is correct and the input wrong, its error messages are not necessarily useful for the repair process.

An automatic grammar repair can thus be useful whenever a given grammar needs to be patched to fit a given test suite for the intended target language, as it eliminates the manual repair efforts. However, automation also enables more interesting application scenarios in various areas, for example (*i*) *teaching*: patches can be integrated into an automated interactive feedback system [5] to help students developing a grammar; (*ii*) *grammar maintenance*: patches can be used to automate the adaptation of a base grammar to capture a dialect from examples [48]; (*iii*) *grammar migration*: patches can fix errors introduced by migration of a grammar from one formalism (e.g., LR with precedences) into another one with different capabilities (e.g., pure LL); (*iv*) *grammar inference*: patches can replace the blind search in the inner loop of genetic grammar learning algorithm [8, 48].

In this paper, we describe the first approach to *automatically repair* bugs in context-free grammars: given a grammar that fails some tests in a given test suite, we iteratively and gradually transform the grammar until it passes all tests.

Our approach is based on the "find-and-fix" cycle typically used in manual repair. As an example, consider a situation where we are trying to develop a CUP [25] grammar specification against a small test suite $TS_{\mathcal{Toy}}$ with passing tests to complement an informal description of the target language $\mathcal{Toy}$. Assume that our grammar $G'_{\mathcal{Toy}}$ is similar to the correct version $G_{\mathcal{Toy}}$ shown in Fig. 1 (See Appendix A), with the exception of the last two rules that have the following form:

$$\begin{aligned} &\textit{name} &&\to \texttt{id} \mid \texttt{id}\,\textbf{[}\ \textit{simple}\ \textbf{]} \mid \texttt{id}\,\textbf{(}\ \textit{name namelist}\ \textbf{)} \\ &\textit{namelist} &&\to \textit{namelist}\ \textbf{,}\ \textit{name} \mid \epsilon \end{aligned}$$

Assume further that we are faced with the following three failing tests in $TS_{\mathcal{Toy}}$:

```
program a begin a(0) end
program a begin a(0, 0) end
program a begin a(0, 0, 0) end
```

In all three cases, CUP's syntax error messages are indeed not useful—in particular, they only confirm the error location and token, but give no further information:

SLE '21, October 17–18, 2021, Chicago, IL, USA
© 2021 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9111-5/21/10.
https://doi.org/10.1145/3486608.3486910

```
Error in line 1, column 19: Syntax error.
Found NUM(0), expected token classes are [].
```

We therefore need to trace the failing tests back to our grammar, to identify the faulty rules and then the precise fault positions within these; in this case, this is relatively straightforward because all three tests fail right after the token sequence `id (`, and there is only one rule in $G'_{\mathcal{Toy}}$ where this sequence occurs, i.e.,

$$name \rightarrow \texttt{id} \mid \texttt{id [} simple \texttt{]} \mid \texttt{id (} \bullet\, name\ namelist \texttt{)}$$

Here, we use the •-symbol to indicate the suspected error position, i.e., the error is at *name* on the right-hand side of the third rule for *name*.

Based on this (manual) *fault localization*, we can now try to *repair* the fault and *fix* the grammar. We first try to *patch* the faulty rule, by applying a small, localized transformation rather than to refactor the entire grammar. Common patches include deleting, inserting, or substituting symbols, and we decide to substitute *name* by `num`, to ensure that the *bad token* `num` is accepted. Note that there are other patches that also ensure this (e.g., inserting `num` or substituting *name* by *expr*) but this is the least intrusive patch.

We then *validate* this patch, i.e., generate a CUP parser from the patched grammar and run it over the test suite. Here, the patch turns out to be a *partial repair* only: it does not introduce any new test failures but does not resolve all previous failures, and we are left with two failing test cases:

```
program a begin a(0, 0) end
program a begin a(0, 0, 0) end
```

In both cases, we get the same syntax error messages as before, with the new error locations showing that we indeed made some progress on these two test as well:

```
Error in line 1, column 22: Syntax error.
Found NUM(0), expected token classes are [].
```

This indicates that the patched grammar still contains another occurrence of *name* that needs to substituted, i.e.,

$$namelist \rightarrow namelist \texttt{ , } \bullet\, name \mid \epsilon$$

Patching the first *namelist*-rule accordingly resolves the two test failures. Both patches together thus constitute a *full repair* that *fixes* the grammar.

Our first contribution in this paper is the automation of this manual find-and-fix cycle, which involves the following:

- *localization*: we use an improved variant [51] of spectrum-based fault localization [50] for CFGs to identify promising repair sites (i.e., specific positions in rules);
- *transformation*: we apply small-scale grammar transformations or *patches* at these sites whenever they satisfy explicitly formulated pre-conditions (see Section 4 to Section 6 for details) that are necessary to potentially improve the grammar;
- *control*: we alternate between localization and transformation, as they reinforce each other and iterate until we find a fix. We use a priority queue to keep improving the most promising candidate grammars.

Our approach is informed by two basic principles, the *competent programmer hypothesis* ("most programmers are competent enough to create correct or almost correct source code") [10] and *Occam's razor* ("entities should not be multiplied without necessity"). In our context, the former means that we can reasonably hope to construct G' from G through a sequence of patches, while the latter means that the repair uses the vocabulary and the structure of the original grammar, and minimizes the number of applied patches.

We have implemented, as a second contribution, this approach in the gfixr system, which takes as input a CFG G in CUP-format and a suite of positive and negative tests for the intended language $\mathcal{L}$, and automatically constructs a "similar" CFG G' that accepts all positive tests and rejects all negative tests. We have successfully used gfixr to fix grammars students submitted as homeworks in a compiler engineering course, and to map one Pascal dialect grammar against another dialect. Given a variant of the example grammar shown in Fig. 1 that uses the two faulty rules above, and a test suite $TS_{\mathcal{Toy}}$ satisfying the CDRC coverage [31] for the target language $\mathcal{Toy}$, gfixr finds the same full repair in less than a minute. gfixr can be configured to explore the repair space differently, preferring the most *general* repairs instead of the most specific ones as above; it then produces the alternative repair.

$$\begin{aligned} name &\rightarrow \ldots \mid \texttt{id (} expr\ namelist \texttt{)} \\ namelist &\rightarrow namelist \texttt{ , } expr \mid \epsilon \end{aligned}$$

Moreover, gfixr can also take advantage of *negative* tests: with only seven counterexamples added to the test suite, gfixr can fix three different quirks in the grammar which allows procedure calls without argument lists, call expressions as *lvalues*, and indexing expressions as statements.

2 Preliminaries

2.1 Context-Free Grammars

Grammar notation. A *context-free grammar* (CFG) is a four-tuple $G = (N, T, P, S)$ with $N \cap T = \emptyset$, $V = N \cup T$, $P \subset N \times V^*$, and $S \in N$. We call S the *start symbol* and use $A, B, C, \ldots$ for non-terminals N, $a, b, c, \ldots$ for terminals T, X, Y, Z for *grammar symbols* V, p, q, r for *productions* or *rules* P, w, x, y, z for *words* over T^*, and $\alpha, \beta, \gamma, \ldots$ for *phrases* over V^*, with ϵ for the empty string and $|\alpha|$ for the length of α. We write $A \longrightarrow \gamma$ for a rule $(A, \gamma) \in P$ and $P_A = \{A \longrightarrow \gamma \in P\}$ for the rules for A.

Derivations. We use $\alpha A \beta \Rightarrow \alpha\gamma\beta$ to denote that $\alpha A \beta$ *produces* $\alpha\gamma\beta$ by application of the rule $A \longrightarrow \gamma \in P$ and use $\Rightarrow^*$ for its reflexive-transitive closure. We write $\Rightarrow_R$ if $A \longrightarrow \gamma \in R \subseteq P$. We call a phrase α a *sentential form* if $S \Rightarrow^* \alpha$. The *yield* of α is the set of all words that can be derived from it, i.e., $\text{yield}(\alpha) = \{w \in T^* \mid \alpha \Rightarrow^* w\}$. The *language* $L(G)$ *generated by a grammar* G is the yield of its start symbol, i.e., $L(G) = \{w \in T^* \mid S \Rightarrow^* w\}$.

α is *nullable* if $\epsilon \in \text{yield}(\alpha)$. We define the *first* (resp. *last*) *set* of a phrase α as $\text{first}(\alpha) = \{a \mid \alpha \Rightarrow^* a\beta\}$ (resp. $\text{last}(\alpha) = \{a \mid \alpha \Rightarrow^* \beta a\}$), and the *precede* (resp. *follow*) *set* of a symbol X $\text{precede}(X) = \{a \mid S \Rightarrow^* \alpha a X \beta\}$ (resp. $\text{follow}(X) = \{a \mid S \Rightarrow^* \alpha X a \beta\}$).

We call u a *viable* k*-prefix* of a word $w = uv$ if $|u| \le k$ and $S \Rightarrow^* uv'$ for a $v' \in T^*$, and denote this by $u \preceq_k w$. We call a viable k-prefix $u \preceq_k w$ *maximal* if there is no $a \in T$ such that $ua \preceq_{k+1} w$. Hence, $w \preceq_{|w|} w$ iff $w \in L(G)$ and, conversely, if the maximal viable prefix u has length $k < |w|$ then w has a syntax error at position $k + 1$. We denote the the maximal viable prefix of w by $\text{prefix}(w)$.

Items. An *item* is a rule $A \to \alpha \bullet \beta$ with a designated position (denoted by $\bullet$) on its right-hand side. We use $P^\bullet$ to denote the set of all items, i.e., all rules with all designated positions. We often use items and rules interchangeably, but where necessary we use $p^\bullet$ to distinguish an item from the underlying rule p.

We define as the *left* (resp. *right*) *set* of an item the sets of symbols that can occur immediately to the left (resp. right) of the designated position [52]. Hence, the left set of an item $A \to \alpha \bullet \beta$ contains all tokens that can occur at the end of α and, if α is nullable, all tokens that in other contexts can occur left of A.

$$\text{left}(A \to \alpha \bullet \beta) = \begin{cases} \text{last}(\alpha) \cup \text{precede}(A) & \text{if } \alpha \text{ nullable} \\ \text{last}(\alpha) & \text{otherwise} \end{cases}$$

$$\text{right}(A \to \alpha \bullet \beta) = \begin{cases} \text{first}(\beta) \cup \text{follow}(A) & \text{if } \beta \text{ nullable} \\ \text{first}(\beta) & \text{otherwise} \end{cases}$$

2.2 Test Suites for CFGs

A *test suite* consists of a list of inputs for a system under test (SUT) and corresponding expected outputs; the SUT *passes* a test if it produces the expected output for the given input. In our case, test inputs are words $w \in T^*$, and the expected outputs are either "accept" (if the test is *positive*, i.e., $w \in \mathcal{L}$) or "reject" (if the test is *negative*, i.e., $w \notin \mathcal{L}$).

More specifically, a *test suite* for a *target language* $\mathcal{L}$ is a pair $TS_{\mathcal{L}} = (TS^+, TS^-)$ of positive tests $TS^+ \subseteq \mathcal{L}$ and negative tests TS^- with $TS^- \cap \mathcal{L} = \emptyset$. By abuse of notation, we also use $TS_{\mathcal{L}}$ for the union $TS^+ \cup TS^-$ of both sets. We require $TS_{\mathcal{L}}$ to be *finite* and *consistent*, i.e., $TS^+ \cap TS^- = \emptyset$. A test w is called a *true positive* if $w \in TS^+ \cap \mathcal{L}$, *false positive* if $w \in TS^- \cap \mathcal{L}$, *true negative* if $w \in TS^- \setminus \mathcal{L}$, and *false negative* if $w \in TS^+ \setminus \mathcal{L}$.

Since we are using test suites as specification data for the repair, it is important to ensure that they adequately reflect the syntactic structure of the target language. More specifically, for most examples and experiments, we therefore use test suites that are automatically generated from the EBNF version of the respective target grammar to satisfy CDRC [31] coverage. For the running example, the CDRC test suite $TS_{\mathcal{Toy}}$ contains 69 positive tests. For patch validation, we also created a test suite that contains all valid bigrams.

2.3 Spectrum-Based Fault Localization in CFGs

Software fault localization [1, 23, 43, 53, 65] techniques attempt to identify likely bug locations in software. *Spectrum-based fault localization* techniques record execution information called a *program spectrum* for a program when running over a given test suite. From the spectrum, they then compute *suspiciousness scores* for each program element (e.g., method or statement), which can be interpreted as the likelihood that that element contains a fault. Different formulas (e.g., Tarantula [23], Ochiai [46], or Jaccard [7]) have been proposed for the score computation but they all combine in different ways the numbers of passed resp. failed tests in which each program element is executed resp. not executed.

Spectrum-based fault localization has also been used to identify the rules that cause a parser to accept words outside (resp. not accept words within) the expected language [50]. We extended this approach to localize faults at the level of individual symbols in rules [51], which reduces the number of possible fault locations compared to a rule-level localization, where we would need to iterate over all positions in the identified rules. The computation of the *item spectra* relies on an instrumented parser to record information as the grammar rules are exercised for a given test case, but the remaining steps of the approach remain unchanged.

We have experimented with different variants of item spectra, but for the repair we can consider localization as a black box, and model a spectrum as the union of two different relations $\sim_{✓}, \sim_{✗} \subseteq P^\bullet \times TS_{\mathcal{L}}$ between items and tests that encode test execution and test outcome. We then define $\text{pass}(p^\bullet) = \{w \in TS_{\mathcal{L}} \mid p^\bullet \sim_{✓} w\}$ and $\text{fail}(p^\bullet) = \{w \in TS_{\mathcal{L}} \mid p^\bullet \sim_{✗} w\}$ as the sets of passing and failing tests executing p up to the designated position, respectively. We can then define the usual counts $N_{\text{pass}} = |\bigcup_{p^\bullet} \text{pass}(p^\bullet)|$, $ep(p^\bullet) = |\text{pass}(p^\bullet)|$, and $np(p^\bullet) = N_{\text{pass}} - ep(p^\bullet)$, and correspondingly, $N_{\text{fail}} = |\bigcup_p \text{fail}(p^\bullet)|$, $ef(p^\bullet) = |\text{fail}(p^\bullet)|$, and $nf(p^\bullet) = N_{\text{fail}} - ef(p^\bullet)$.

We model the suspiciousness scores with an abstract *scoring function* $\text{score} : P^\bullet \to \mathbb{R}^+ \cup \{0\}$, which must satisfy $\text{score}(p^\bullet) > 0 \Rightarrow \text{fail}(p^\bullet) \ne \emptyset$. The usual formulas can be used based on the definitions of the counts given above.

We finally implemented a specialized *tie breaking* mechanism in favour of "longer" items from the same rule, i.e., whenever $\text{score}(A \to \alpha \bullet X\omega) = \text{score}(A \to \alpha X \bullet \omega)$, we set the score of the "shorter" item to zero and so remove it from the pool of possible fault locations. This is based on the left-to-right reading order of the parser: since all executions that got to X also got over X, the error cannot be before X. Appendix B shows details for the localization of faults in $G'_{\mathcal{Toy}}$ over $TS_{\mathcal{Toy}}$.

3 Repair Framework

In this section, we formalize the individual elements of our repair approach. The overall structure of the repair algorithm that follows the find-and-fix cycle mentioned in the introduction is shown in Algorithm 1; more implementation details are given in Section 7.

The Repair Problem. We assume that we have a *test suite* $TS_{\mathcal{L}} = (TS^+, TS^-)$ for an unknown *target language* $\mathcal{L}$ that is comprised of positive tests $TS^+ \subseteq \mathcal{L}$ and negative tests TS^- with $TS^- \cap \mathcal{L} = \emptyset$, and an initial CFG G that fails at least one test in $TS_{\mathcal{L}}$ (i.e., $TS^+ \nsubseteq L(G)$ or $TS^- \cap L(G) \neq \emptyset$). The *repair problem* is then to construct from $TS_{\mathcal{L}}$ and G a "similar" CFG G' that accepts all positive tests (i.e., $TS^+ \subseteq L(G')$) and rejects all negative tests (i.e., $TS^- \cap L(G') = \emptyset$) and so approximates $\mathcal{L}$ better than G. We require in the following that the test suite $TS_{\mathcal{L}}$ is *viable for* G, i.e.,

(i) it detects at least one fault in G, i.e., $(TS^- \cap L(G)) \cup (TS^+ \backslash L(G)) \neq \emptyset$;
(ii) it is *constructive*, i.e., $TS^- \subseteq L(G)$.

The first condition ensures that the test suite is strong enough so we can localize and fix, while the second ensures that negative tests contain enough structure that can be exploited for repair attempts. In the remainder of the paper, we assume a fixed test suite $TS_{\mathcal{L}} = (TS^+, TS^-)$ that is viable for the initial CFG G.

However, the problem is underspecified and a repair can "overgeneralize", i.e., $TS^+ \subseteq L(G') \nsubseteq \mathcal{L}$. In an active learning setting like Angluin's query model [3], the learner can ask the teacher whether a given w in $L(G')$ is also in $\mathcal{L}$, but in our passive learning setting there is no teacher and $\mathcal{L}$ is specified only through $TS_{\mathcal{L}}$. We can therefore evaluate the quality of our repairs only through manual inspection or based on performance over an additional *validation suite*.

Patches, Repairs, and Fixes. A *grammar patch* $\mathfrak{p}$ is simply a transformation from one CFG $G = (N, T, P, S)$ into another CFG $G' = (N', T', P', S')$; we denote this by $G \rightsquigarrow_{\mathfrak{p}} G'$. In principle a patch can also introduce new terminal and non-terminal symbols, but we restrict ourselves to *rule patches*, i.e., we assume $N = N'$, $T = T'$, and $S = S'$. The introduction and deletion of non-terminals can be supported by some simple equivalence transformations, but the introduction of new terminals is technically more complex, due to the required integration of lexer and parser.We thus leave this for future work.

A patch $G \rightsquigarrow_{\mathfrak{p}} G'$ is *viable* with respect to a viable test suite $TS_{\mathcal{L}}$ if G' performs no worse over $TS_{\mathcal{L}}$ than G, i.e.,

(i) $L(G) \cap TS^+ \subseteq L(G') \cap TS^+$;
(ii) $L(G) \cap TS^- \supseteq L(G') \cap TS^-$;
(iii) $\forall w \in TS_{\mathcal{L}} \cdot \text{prefix}_G(w) \leq \text{prefix}_{G'}(w)$

A viable patch is an *improvement* if one of the set inclusions or prefix relations is strict, and a *partial repair* if one of the set inclusions is strict, i.e., G' fails fewer tests than G. It is a *full repair* or a *fix for* G if G' passes all tests, i.e., $TS^+ \subseteq L(G')$ and $TS^- \cap L(G') = \emptyset$.

Induced Patches. In the following sections, we define a series of transformations that compute a patch item $q^\bullet$ from a suspicious item $p^\bullet$. However, we cannot simply patch the grammar by replacing p with q in P: if p was used in at least one passing positive test case (i.e., $p^\bullet \sim_{\checkmark} w$ for a $w \in TS^+$) then an in-place update can make G' fail a test case that G was passing, and so render the patch unviable. We therefore need to control update by spectral counts.

Hence, given $G = (N, T, P, S)$ the patch $G \rightsquigarrow_{(p,q)} G'$ is *induced* by the pair $(p^\bullet, q^\bullet)$ if $G' = (N, T, P', S)$, and

$$P' = \begin{cases} P \cup \{q\} \setminus \{p\} & \text{if } ep^+(p^\bullet) = 0 \\ P \cup \{q\} & \text{if } ep^+(p^\bullet) > 0 \end{cases}$$

By abuse of notation, we also write $p \rightsquigarrow q$ (resp. $G \rightsquigarrow_q G'$) to mean $G \rightsquigarrow_{(p,q)} G'$ if G and G' (resp. p) are clear from the context or are immaterial.

Good Tokens, Bad Tokens. The second essential ingredient to make our approach scalable is that we limit the repairs that are attempted at each repair site through explicit conditions that capture when a patch is likely to yield a repair. These conditions are formulated over the grammar structure (using predicates such as first and follow), pass and fail counts, and lexical context around the failure locations, aggregated over the individual false negatives.

Recall that $w = uabv \notin L(G)$ and $ua \leq_k w$ maximal mean that the (first) syntax error occurs between a and b. We call a, which is the last token successfully consumed just before the parser reports the syntax error, the *good token* for w and b the *bad token.* A pair $(a, b)_w$ of good and bad tokens for w can be seen as a poisoned pair in G [52] and our repair attempts to break this property. We define the sets of good tokens T_p^+ and bad tokens T_p^- for an item p as the sets of good and bad tokens from the failing tests in which p is executed, i.e., $(T_p^+, T_p^-) = \bigcup\{(a, b)_w \mid p \sim_{\times} w, w \in TS^-\}$ (where the union is taken componentwise).

Patch Validation against Sample Bigrams. We can prevent some overgeneralization by providing negative tests (see Section 6), which can be seen as pre-emptive answers to some membership queries, but this static set cannot be updated automatically during the repair process without changing to an active learning setting. We can, however, extract more information from the positive tests and use this to check whether a patch can be valid or not. More specifically, given a test suite $TS_{\mathcal{L}} = (TS^+, TS^-)$, we collect all *sample bigrams* $\Gamma_2(TS_{\mathcal{L}}) = \{(a, b) \mid w = xaby \in TS^+\}$ that occur in the positive tests, and check that the terminals that can after the repair occur directly to the left and right of the repair site also occur as sample bigrams; if not, we reject the patch.

Repair Algorithm. Algorithm 1 shows the main repair loop that implements the find-and-fix cycle described in the introduction. It dequeues the top-ranked faulty grammar

Algorithm 1: The main repair loop

```
input : A faulty grammar $G = \langle N, T, P, S \rangle$
input : A test suite $TS$
output: A fully repaired variant $G'$ or $\bot$
1  $Q \leftarrow \emptyset$
2  $\langle P, F, Pre \rangle \leftarrow$ run_tests($G, TS$)
3  $Q$.enqueue($G, \langle P, F, Pre, \infty \rangle$)
4  $Seen \leftarrow \{G\}$
5  repeat
6      $\langle G', \langle P_{G'}, F_{G'}, Pre_{G'}, \_ \rangle \rangle \leftarrow Q$.dequeue()
7      $Ranks \leftarrow$ localize($G', TS$)
8      $Cands \leftarrow$ transform($G', Ranks$)
9      for $C \in Cands \setminus Seen$ do
10         $Seen$.add($G'$)
11         $\langle P_C, F_C, Pre_C \rangle \leftarrow$ run_tests($C, TS$)
12         if $F_C = \emptyset$ then
13             return $C$
14         if improves($\langle P_{G'}, F_{G'}, Pre_{G'} \rangle, \langle P_C, F_C, Pre_C \rangle$)
            then
15             $Q$.enqueue($C, \langle P_C, F_C, Pre_C, Ranks[C] \rangle$)
16 until $Q$.empty()
17 return $\bot$
```

variant G' from a central priority queue Q, runs `localize` to determine possible repair sites (i.e., suspicious items), and then calls `transform` to try and apply the patches described in more detail in the following sections. For each unseen candidate C, it uses `execute_tests` to generate an executable parser and run it over the test suite TS. If the candidate C `improves` on its parent G', it is enqueued.

The priority queue Q contains pending grammar candidates derived from improving patches. It is keyed by a four-tuple (P, F, Pre, R), where P and F are the number of passing and failing tests, respectively, Pre is the length of the successfully parsed prefixes, and R is the localization rank of the patched item from which the candidate was derived. We use lexical order to determine the priority. The algorithm also maintains a set of *Seen* candidate grammars to prevent non-termination.

The `localize` module determines potential repair sites in the faulty grammar variant, and provides further spectral information such as basic counts (ef, ep, nf, np) and the aggregated sets of good T_p^+ and bad tokens T_p^- for each item p to the `transform` module.

4 Symbol Editing Patches

Our first group of patches is modelled on the basic string editing operations (i.e., deletion, insertion, and substitution), applied to the symbols on the right-hand sides of the rules.

4.1 Symbol Deletions

We first consider symbol deletion patches. These are useful to fix bugs where the grammar fails to properly handle optional elements. Consider for example a test suite $TS'_{\mathcal{Toy}} \supseteq TS_{\mathcal{Toy}}$ that also includes the three (positive) tests:

```
program a define a() begin relax end begin relax end
program a
  define a() -> int begin relax end
  begin relax end
program a begin a() end
```

These tests fail under $G_{\mathcal{Toy}}$ because neither *paramlist* nor *arglist* are nullable, and their addition to $TS_{\mathcal{Toy}}$ can be seen as a "repair request" to change $G_{\mathcal{Toy}}$ to allow empty formal parameter and argument lists.

The localization identifies amongst others the following three items as suspicious:

$$fdecl \rightarrow \texttt{define}\ \texttt{id}\ \texttt{(}\ \bullet\ paramlist\ \texttt{)}\ body$$
$$fdecl \rightarrow \texttt{define}\ \texttt{id}\ \texttt{(}\ \bullet\ paramlist\ \texttt{)}\ \texttt{->}\ type\ \texttt{id}\ body$$
$$name \rightarrow \ldots \mid \texttt{id}\ \texttt{(}\ \bullet\ arglist\ \texttt{)}$$

In all cases, the sets of good and bad tokens are $T^+ = \{\texttt{(}\}$ and $T^- = \{\texttt{)}\}$, respectively. We use the former to check that the designated position in the item is actually correlated to the lexical error contexts and, specifically, that the item's left set contains only good tokens. This is trivially the case here, since the left sets of all three items are $\{\texttt{(}\}$ as well. We use the latter similar to the way a parser's panic mode error recovery uses *synchronization tokens*: starting at the designated position, we delete symbols from the rule until this synchronizes the rule with the bad tokens, i.e., until the right set of the item after the deletion contains all bad tokens. This is again trivially the case here, since in all three cases the corresponding right sets after the deletion of the first symbol are $\{\texttt{)}\}$ as well.

However, we need to be careful that we are not adding rules with exposed nullable symbols, which can use the ϵ-derivation to accept the new tests but which allow unintended derivations and thus overgeneralize. Consider the repaired variant $G'_{\mathcal{Toy}}$ from Section 1 again:

$$\begin{aligned} name &\rightarrow \ldots \mid \texttt{id}\ \texttt{(}\ \bullet\ expr\ namelist\ \texttt{)} \\ namelist &\rightarrow namelist\ \texttt{,}\ expr \mid \epsilon \end{aligned}$$

Deleting the *expr*-symbol at the localized position in the *name*-rule allows us to synchronize on `)` because *namelist* is nullable but this also allows for example a derivation $name \Rightarrow_{G'} \texttt{id}\,\texttt{(}\, namelist\,\texttt{)} \Rightarrow^*_{G'} \texttt{id}\,\texttt{(}\,\texttt{,}\,\texttt{id}\,\texttt{)}$. This overgeneralization could be prevented by explicit counter-examples, but we instead rely on a careful formalization of the *synchronization patch* and corresponding *patch validation.*

Definition 1 (synchronization). *Let $p = A \rightarrow \alpha \bullet \beta\omega$ be an item in $P^\bullet$ with* $\text{left}(p) \subseteq T_p^+$.

(a) If $\omega = X\gamma$ with X non-nullable and $T_p^- \subseteq \text{first}(X)$, let $\mathfrak{d}(p, \beta) = A \rightarrow \alpha \bullet \omega$ be the result of deleting β at the

designated position. Then $p \rightsquigarrow \mathfrak{d}(p, \beta)$ *is a* synchronization patch.

(b) If ω *is nullable and* $T_p^- \subseteq \text{follow}(A)$*, let* $\mathfrak{d}(p, \beta\omega) = A \to \alpha\bullet$ *be the result of deleting* $\beta\omega$ *at the designated position. Then* $p \rightsquigarrow \mathfrak{d}(p, \beta\omega)$ *is a* panic mode synchronization patch.

We validate synchronization patches by checking that the test suite contains all bigrams that are newly possible by the deletion of β. More specifically, we compare the left- and right-sets in G' against the bigrams around the repair site.

Definition 2 (synchronization validation)**.** *Let* $p = A \to \alpha \bullet \beta\omega$ *be an item in* $P^\bullet$*. The synchronization patch* $G \rightsquigarrow_{\mathfrak{d}(p,\beta)} G'$ *is* validated *over* $TS_{\mathcal{L}}$ *if* $\text{left}_{G'}(\mathfrak{d}(p, \beta)) \times \text{right}_{G'}(\mathfrak{d}(p, \beta)) \subseteq \Gamma_2(TS_{\mathcal{L}})$.

In the running example, the deletions of *paramlist* and *arglist* both only expose the single "repair bigram" (**(** , **)**), which occurs in $\Gamma_2(TS'_{\mathcal{Toy}})$.

Example Repairs. gfixr patches the baseline grammar $G_{\mathcal{Toy}}$ against $TS'_{\mathcal{Toy}}$ as expected, by adding the three rules

fdecl → **define** id **( )** *body*
fdecl → **define** id **( ) ->** *type* id *body*
name → . . . | id **( )**

The rules are created from the corresponding baseline rules by deletion of a single symbol at the identified fault locations shown above, and are added to the grammar, rather than replacing the baseline rules, because the latter are used in other passing tests. gfixr finds this fix with three patches in roughly two minutes, generating 26 candidate grammars.[1] Note that the initially top-ranked item *param* → •*type* **array** id induces a rule *param* → ϵ through a panic mode synchronization, but this fails the patch validation and gets ruled out because $\Gamma_2(TS'_{\mathcal{Toy}})$ does not contain the bigram (**(** , **,**).

In the variant $G'_{\mathcal{Toy}}$ from Section 1, the synchronization deletes both the *expr* and the subsequent nullable *namelist* symbols in the *name*-rule (and similarly for the *fdecl*-rule). gfixr finds the corresponding fix with three patches in less than 90 seconds, generating 18 candidate grammars.

As an example for the deletion of longer sequence of symbols consider a faulty version of $G_{\mathcal{Toy}}$ where the first *fdecl*-rule is missing (e.g., due to a missing ?-operator around the sequence **->** *type* id at the EBNF level). Here, gfixr introduces a copy of *fdecl*-rule without the segment **->** *type* id. It finds this single patch fix in roughly 30 seconds, generating only five candidate grammars.

4.2 Symbol Insertions

Symbol insertion patches are useful to fix bugs where grammar developers have missed one or more symbols in a rule, or even an entire rule (e.g., the second *fdecl*-rule). However, we only insert a single symbol and rely on repeated repairs to grow larger patches symbol by symbol, in order to limit the number of different repairs that we need to consider at each suspicious location. In contrast to symbol deletion patches, where we effectively check that the bad tokens are a subset of the right-set (i.e., $T_p^- \subset \text{right}(A \to \alpha \bullet \omega)$) and the patch thus covers *all* failing tests associated with the item, we check here only for a non-empty intersection (i.e., $T_p^- \cap \text{right}(A \to \alpha \bullet \omega) \neq \emptyset$), in order to allow patch to (partially) repair a subset of failing tests at a time.

Definition 3 (symbol insertion)**.** *Let* $p = A \to \alpha \bullet \omega$ *be an item in* $P^\bullet$ *with* $\text{left}(p) \subseteq T_p^+$*, and* $\mathfrak{i}(p, X) = A \to \alpha \bullet X\omega$ *be the result of inserting* $X \in V$ *at the designated position of* p*. If* $T_p^- \cap \text{right}(\mathfrak{i}(p, X)) \neq \emptyset$*, then* $p \rightsquigarrow \mathfrak{i}(p, X)$ *is an* insertion patch.

We validate insertion patches by checking the same condition as for synchronization patches, with the designated position *before* the inserted symbol; we do not check the symmetric condition for the designated position *after* the inserted symbol, because the insertion could be part of a larger patch that is found through repeated insertions.

Definition 4 (insertion validation)**.** *Let* $p = A \to \alpha \bullet \omega$ *be an item in* $P^\bullet$ *and* $X \in V$*. The insertion patch* $G \rightsquigarrow_{\mathfrak{i}(p,X)} G'$ *is* validated *over* $TS_{\mathcal{L}}$ *if* $\text{left}_{G'}(\mathfrak{i}(p, X)) \times \text{right}_{G'}(\mathfrak{i}(p, X)) \subseteq \Gamma_2(TS_{\mathcal{L}})$.

Example Repair. If we remove the rule

fdecl → **define** id **( ) ->** *type* id *body*

from $G_{\mathcal{Toy}}$, gfixr re-introduces it with three patches, each inserting an individual symbol to form the segment **->***type*id. It takes 53 seconds, generating 13 candidate grammars.

4.3 Symbol Substitutions

Substitution patches fix bugs where grammar developers have used a wrong symbol, as shown in the example from the introduction. Such bugs are particularly difficult to detect when the grammar is either too permissive (e.g., *name* → id **[** *expr* **]**) or too restrictive, in a way that is only uncovered by structurally complex tests (e.g., *paramlist* → *param* | *param* **,** *param*).

Definition 5 (symbol substitution)**.** *Let* $p = A \to \alpha \bullet X\omega$ *be an item in* $P^\bullet$ *with* $\text{left}(p) \subseteq T_p^+$*,* $Y \in V$*, and* $\mathfrak{s}(p, Y) = A \to \alpha \bullet Y\omega$ *be the result of replacing* X *at the designated position by* Y*. If* $T_p^- \cap \text{right}(A \to \alpha \bullet Y\omega) \neq \emptyset$*, then* $p \rightsquigarrow \mathfrak{s}(p, Y)$ *is a* substitution patch.

In contrast to insertion patches, substitution patch validation checks both sides of the repair site, to ensure the substituted symbol fits tightly.

Definition 6 (substitution validation)**.** *Let* $p = A \to \alpha \bullet X\omega$ *be an item in* $P^\bullet$ *and* $Y \in V$*. The substitution patch* $G \rightsquigarrow_{\mathfrak{s}(p,Y)} G'$ *is* validated *over* $TS_{\mathcal{L}}$ *if*

[1] All runtimes given in Section 4 to Section 6 were measured as wall-clock time on an otherwise idle standard 3.20 GHz desktop with 6 cores and 16 GB RAM.

(i) $\text{left}_{G'}(\mathfrak{s}(p, Y)) \times \text{right}_{G'}(\mathfrak{s}(p, Y)) \subseteq \Gamma_2(TS_{\mathcal{L}})$, *and*
(ii) $\text{left}_{G'}(A \longrightarrow \alpha Y \bullet \omega) \times \text{right}_{G'}(A \longrightarrow \alpha Y \bullet \omega) \subseteq \Gamma_2(TS_{\mathcal{L}})$.

Substitution patch validation has two specific effects. First, it leads to a preference for deletions over substitutions with nullable symbols, which in turn leads to better grammars. Second, it leads to a preference of insertions over substitutions; in particular, a "compound" patch $A \longrightarrow \alpha \bullet X\omega \rightsquigarrow A \longrightarrow \alpha YZ \bullet \omega$ is realized via $A \longrightarrow \alpha Y \bullet X\omega$ and not via $A \longrightarrow \alpha Y \bullet \omega$, which reduces the search space.

***Example Repair*.** Substitutions, deletions, and insertions can interact to create larger repairs. Consider for example a student implementation of $G_{\mathcal{Toy}}$ where the rule *factor* → **(** *expr* **)** is missing, so that it rejects bracketed expressions. The top-ranked item *name* → •`id` **[** *simple* **]** fails the precondition on the good tokens for each potential patch, and gfixr tries to patch the *factor*-rules which are ranked next. There are seven possible insertions and substitutions, which all pass the validations, but the substitution patch *factor* → • **not** *factor* ⇝ *factor* → • **(** *factor* improves most, as it accepts longer prefixes. The resulting grammar is therefore picked in the next iteration, where an insertion patch inserts the missing **)**-token, completing the fix. gfixr generated 61 candidates in roughly 3 minutes and 30 seconds.

4.4 Symbol Transpositions

The final symbol edit patch we consider is symbol transposition, which swaps the two symbols following the designated position. While this is not a common bug pattern, it does occur in connection with list rules. For example, one student submission for $G_{\mathcal{Toy}}$ had the following bug in the *idlist*-rule

idlist → `id` *idlisttail*
idlisttail → • `id` **,** *idlisttail* | ϵ

that leads to a pair of adjacent `id`-tokens in the beginning and a trailing comma at the end of an *idlist*. gfixr generates a patch that swaps `id` and **,** in *idlisttail*, which in turn fixes the rule. It found this in a single iteration, in about 1 minute 20 seconds, generating 23 candidates.

Definition 7 (symbol transposition). *Let* $p = A \longrightarrow \alpha \bullet XY\omega$ *be an item in* $P^\bullet$ *with* $\text{left}(p) \subseteq T_p^+$, *and* $\mathfrak{t}(p) = A \longrightarrow \alpha \bullet YX\omega$ *be the result of swapping the symbols X and Y at the designated position. If* $T_p^- \cap \text{right}(p \rightsquigarrow \mathfrak{t}(p)) \neq \emptyset$, *then* $p \rightsquigarrow \mathfrak{t}(p)$ *is a* transposition patch.

Transposition patch validation follows the same lines as substitution patch validation, and checks the corresponding conditions on the three items $A \longrightarrow \alpha \bullet XY\omega$, $A \longrightarrow \alpha X \bullet Y\omega$, and $A \longrightarrow \alpha XY \bullet \omega$.

5 Listification Patches

Right recursion introduction or "listification" patches are useful to handle bugs where the grammar fails to properly handle repetitions. Consider for example a variant of $G_{\mathcal{Toy}}$ submitted by a student where the the *body*-, *vdecllist*-, and *vdecl*-rules in $G_{\mathcal{Toy}}$ are replaced by the following rules:

body → **begin** *vdecls stmts* **end**
vdecls → *type* `id` *idlist* **;** • | ϵ

This allows only at most one variable declaration (despite the intent of the name *vdecls*) and thus fails the test

```
program a begin bool a; • bool a relax end
```

with the •-symbol also indicating the error location observed in the input.

Definition 8 (listification). *Let* $G = (N, T, P, S)$, $G' = (N, T, P', S)$ *be CFGs, and* $p = A \longrightarrow \alpha \bullet \in P^\bullet$ *a reduction item with* $\text{first}(A) \subseteq T^-$.
(a) If A is nullable and $A \longrightarrow \epsilon \in P$, *let* $P' = P \setminus \{p\} \cup \{A \longrightarrow \alpha A\}$.
(b) If A is nullable and $A \longrightarrow \epsilon \notin P$, *let* $P' = P \setminus \{p\} \cup \{A \longrightarrow \alpha A, A \longrightarrow \epsilon\}$.
(c) If A is not nullable, let $P' = P \cup \{A \longrightarrow \alpha A\}$.
Then $G \rightsquigarrow_{\mathfrak{L}(p)} G'$ *is a* right recursion introduction *or* listification patch.

Listification checks if A is nullable to decide whether to allow empty lists or not; this is a heuristic, but further patches can refine the repair, if required. It also checks for an existing ϵ-rule before adding it, to prevent introducing conflicts.

Note that listification can be seen as special case of symbol insertion that always uses an in-place grammar update. This can lead to an overgeneralization, because all occurrences of A are listified at the same time. We can prevent this by checking that the bigrams introduced by the recursion actually occur in the test suite.

Definition 9 (listification validation). *Let* $p = A \longrightarrow \alpha \bullet$ *be an item in* $P^\bullet$. *The listification patch* $G \rightsquigarrow_{\mathfrak{L}(p)} G'$ *is* validated *over* $TS_{\mathcal{L}}$ *if* $\text{left}_{G'}(A \longrightarrow \alpha \bullet A) \times \text{right}(_{G'}A \longrightarrow \alpha \bullet A) \subseteq \Gamma_2(TS_{\mathcal{L}})$.

In the example, gfixr finds the single patch fix *vdecls* → *type* `id` *idlist* **;** *vdecls* in roughly 30 seconds, generating only five candidate grammars.

6 Patches from Counterexamples

Recall that the grammar in Fig. 1 does not distinguish properly between simple identifiers, array indexing expressions, and function calls, and instead subsumes all three under the non-terminal *name*:

assign → *name* | *name* **::=** *expr* | *name* **::=** **array** *simple*
input → **read** *name*
factor → *name* | ...
name → `id` | `id` **[** *simple* **]** | `id` **(** *expr exprlist* **)**

This means that the compiler's semantic analysis must filter out idiosyncratic constructions, such as

- simple identifiers as statements (i.e., function calls without argument lists), e.g.,
  `program a begin a end`
- array indexing expressions as statements, e.g.,

```
program a begin a[0] end
```

- function calls as *lval* in assignments, array initializations, and input statements, e.g.,
```
program a begin a(0) ::= 0 end
program a begin a(0) ::= array 0 end
program a begin read a(0) end
```
- array indexing expressions as *lval* in array initializations (which would require nested arrays), e.g.,
```
program a begin a[0] ::= array 0 end
```

The common cause of such issues is that the grammar is too permissive, i.e., $\mathcal{L} \subsetneq L(G)$. A repair requires a language restriction or *tightening*, which can be specified by negative tests. We focus here on false positives or *counterexamples* because arbitrary negative tests do not provide enough structure to guide the repair. In the following, we look at two specific tightening patches, rule deletion and non-terminal splitting or "downcasting".

6.1 Rule Deletion

Clearly, deleting a rule tightens the language; the only nontrivial aspect is to ensure that this actually is a viable patch, i.e., that the deletion does not inadvertently block valid derivations in G of positive tests.

We can ensure this if the rule is only ever used in reductions in false positives (i.e., can be seen as an error production), and if the patch is applied as an *approximation from above* (i.e., all positive tests are already passing without it):

Definition 10 (rule deletion). *Let* $G = (N, T, P, S)$ *with* $TS^+ \subseteq L(G)$, $p = A \longrightarrow \alpha\bullet \in P^\bullet$ *a reduction item, and* $ef(p) > 0$. *If* $G = (N, T, P', S)$ *is a CFG with* $P' = P \setminus \{p\}$ *then* $G \rightsquigarrow_{\mathfrak{D}(p)} G'$ *is a* rule deletion patch.

The gfixr implementation actually uses a relaxed condition that simply requires that the rule has not been used in parsing any true positive (i.e., $ep(p) = 0$ and $\mathrm{fail}(p) \subseteq TS^-$), although this could in principle delete it when it would still be used for a true positive after another patch.

6.2 Non-terminal Splitting

In practice, the conditions of the rule deletion are rarely met, because the rule is used both in failing and passing tests, and the error only manifests in certain rule combinations. Consider for example the rule *input* $\longrightarrow$ **read** *name*, which only fails in combination with *name* $\longrightarrow$ `id` **(** *arglist* **)** .

We therefore need an enabling patch that moves rules into the right contexts (similar in spirit to CDRC testing [31]) and so separates out passing and failing rule applications.

Definition 11 (non-terminal splitting). *Let* $G = (N, T, P, S)$, $p = A \longrightarrow \alpha B \omega \bullet \in P^\bullet$ *a reduction item with* $P_B = \{B \longrightarrow \beta_i\}_{i>1}$, $ep(p) > 0$, $ef(p) > 0$, *and* $\mathrm{fail}(p) \subseteq TS^-$. *If* $G = (N, T, P', S)$ *is a CFG with* $P' = P \setminus \{p\} \cup \{A \longrightarrow \alpha\beta_i\omega\}$ *then* $G \rightsquigarrow_{\mathfrak{S}(p,B)} G'$ *is the* non-terminal splitting patch *for* B.

Note that splitting a non-terminal only in one of the rules $A \longrightarrow \gamma_i$ can introduce parsing conflicts. In the gfixr-implementation, we split across all rules $A \longrightarrow \alpha_i B \omega_i$ where the split non-terminal occurs.

Example Repair. We repaired the idiosyncrasies in $G_{\mathcal{Toy}}$ with a $step_2$ [60] test suite with 159 positive tests and seven negative tests, including the test

```
program a begin a ::= 0; a end
```

in addition to the six tests shown above. gfixr finds the following fix in 7 minutes and 36 seconds in 9 generations, after testing 125 candidates:

$$
\begin{aligned}
stmt \;&\longrightarrow \texttt{id}\,\textbf{(}\,arglist\,\textbf{)} \\
&\mid \texttt{id}\,\textbf{::=}\,expr \mid \texttt{id}\,\textbf{[}\,expr\,\textbf{]}\,\textbf{::=}\,expr \mid \texttt{id}\,\textbf{::=}\,\textbf{array}\,simple \\
&\mid cond \mid \ldots \\
input &\longrightarrow \textbf{read}\,\texttt{id} \mid \textbf{read}\,\texttt{id}\,\textbf{[}\,expr\,\textbf{]}
\end{aligned}
$$

The key patches are several splits of *name* in different contexts, followed by the deletion of the split rule variants that are only used in parsing negative tests. Note that splits at irrelevant contexts (e.g., in *factor*) are ruled out because they do not improve the grammar.

This result is arguably not too far away from a manual repair (that may introduce a proper *lvalues* non-terminal to factor out the commonalities in *assign* and *read*) but the quality of gfixr's repairs obviously depends only on the completeness of the test suite and not on the intent. In this case, the first six tests only indicate errors in the first *stmt* of a *stmtlist*, and the seventh test case was crucial to confine the splits to *assign* and *input*, and to prevent them from recursively "bubbling up" through *stmt* to *stmtlist*.

7 The gfixr System

We have prototyped our repair approach as described in the previous sections in the gfixr tool.

System Architecture. gfixr implements the repair loop shown in Algorithm 1. It uses Python and Maven to orchestrate the repair (e.g., parameter handling or parser generation) and Java to implement the grammar analyses (such as computing the left- and right-functions) and transformations for the patches. The overall system size is about 4.3kLoC.

gfixr can currently only repair CUP grammars, but the system can be adapted to work with other parser generators. This requires modifications in the `localize` (where a modified parser is required to extract spectral information), `transform` (where the grammar meta-model needs to be adapted), and `run_tests` (where the build system needs to be adapted) modules.

The `localize` module currently uses the Ochiai-metric that worked well in our experiments, but this can be reconfigured easily.

Patch Selection. Currently, gfixr uses a simplistic strategy to select the subset and order of the suspicious items identified by `localize` where repairs are attempted: it simply selects all items with a non-zero score and processes them in descending score order. It tries all transformations described in the Section 4 to Section 5 at each repair site to produce candidate patches. Patch selection is therefore integrated into the `transform` module.

gfixr evaluates the performance of each candidate patch over the same input test suite $TS_{\mathcal{L}}$. Better performing patches are pushed towards the front of the priority queue and stand better chances of further transformations until a fix is found.

Patch Validation. In addition to the specific patch validation via bigrams, each candidate patch goes through a generic patch validation to determine whether they improve over their parent, following the definition of improvements in Section 3: (*i*) the candidate reduces the number of failing test cases, or (*ii*) when the number of failing test cases remains unchanged, the candidate must consume at least one longer (and no shorter) prefix than the parent. gfixr discards candidates which do not improve over their parent.

The bigram-based validation requires sample bigrams that can be extracted from the test suite or a different set of sample tests, using a separate small script.

Configuration. gfixr takes as input the initial grammar and the test suite used for the repair. The mandatory option `-bigrams_file` specifies the separately created file containing the bigrams used for patch validation.

The repair algorithm can be configured through a number of command line arguments. `-tight` restricts the symbol substitutions and insertions patches and allows only the most specific possible symbol in a maximal chain $A \Rightarrow^* B$ to be inserted and substituted. `-weak_left` and `-strong_right` change the relation between good resp. bad tokens and left- resp. right-sets required to enable a transformation to non-empty intersection resp. containment (see for example Definition 3). Both settings enable more transformations but may lead to overgeneralization.

Further options control CUP's parsing algorithm. `-rr` sets the number of reduce/reduce conflicts that are allowed in the candidate; default is 0. gfixr discards grammars with more conflicts. `-compact_red` enables CUP's action table compaction, which often allows it to execute reductions pending on the stack when a syntax error is encountered. Both options can have an impact on the localization and should be used only if gfixr cannot repair the grammar.

8 Experiments

8.1 Repairing Student Grammars

In a first set of experiments, we used CUP grammars written by students to evaluate gfixr's efficacy. These grammars describe different medium-size Pascal-style languages used in a graduate compiler engineering course. Many of the approximately 40 submissions have lexical issues and could not handle the interactions between parser and lexer properly. Since the current version of gfixr does not support new token creation we discarded submissions with known lexical issues. We then randomly picked from all submissions that fail on at least one test the 10 submissions shown in Table 1.

For each target language we generated two test suites from instructor's golden grammars, following the approach outlined in Section 2.2, and use the CDRC test suite as repair specification, and the more diverse one to compute the bigrams for patch validation.

Results. Table 1 shows overall promising results, and we can observe a few trends already. First, and foremost, gfixr can indeed fix grammar bugs: it produces high-quality equivalent fixes that capture the original (human) intent of the grammar for seven of the ten cases, technically correct but low-quality fixes that pass all tests for two cases, and fails to find a repair in one case only. This indicates that the localization directs the repair to the right locations, despite the fact that the localization technique it uses is based on single fault assumption and some studies have shown that multiple fault interactions may drop their effectiveness [2, 66], and that the combined patches are sufficiently expressive; in the failing case, however, the localization ranked the faulty location too low, and the repair kept trying to fix correct rules.

Second, repair times are often below a minute, in particular if the grammar contains only a single or two related bugs, as in the running example. Grammars with multiple bugs that require several patches obviously take longer, but gfixr can still find fixes comprising ten patches in about 20 minutes wall-clock time. The overall runtime is approximately linear with the number of candidate grammars.

Third, the number of iterations of the repair loop is typically the same as the number of applied patches, and the number of candidate grammars remains small. This again indicates that the fault localization can identify the faults sufficiently well, and that the priority queue keeps the most promising candidates on top.

Finally, most patch types are used widely, but symbol transposition is applied only once.

8.2 Repairing Pascal Types

In the second experiment we mapped one Pascal dialect to another. We used the same Pascal grammars as in previous work [40, 50]. These grammars are derived from different sources and have been shown to contain multiple deviations [50]. The YACC grammar, which we converted into CUP format, was extracted from ftp://ftp.iecc.com/pub/file/pascal-grammar, and the ANTLR(v4) [47] grammar is available at https://github.com/antlr/grammars-v4/blob/master/pascal/pascal.g4.

In this experiment we used the ANTLR grammar to generate a CDRC test suite with a total of 221 tests. The parser generated from the CUP grammar fails on 47 of these tests, with more than half (25) of the failing tests related to the deviation in the type structure. A manual analysis of these

Table 1. Results of student grammar repairs. $\mathcal{L}$ is the target language, with $\mathcal{T}oy$ the running example, and $\mathcal{A}$ to $\mathcal{E}$ similarly complex languages from assignments. bugs is the number of known faults in the student grammars. $|TS_{\mathcal{L}}|$ is the number of positive tests in the suite, with fails the number of failing tests. iter. is the number of iterations of the repair loop and cand. the number of candidate grammars generated. time is measured as wall-clock time on an otherwise idle standard 3.20 GHz desktop with 6 cores and 16 GB RAM and given as hours:minutes:seconds. applied patches gives the number of different patches in the fix. The last column gives the perceived quality of the fix (inspired by [63]), with ⋆ denoting a fix that corresponds to the (human) intent of the repair, ✓ denoting a technically correct fix that passes all tests but is of low quality, and ✗ denoting a failure of gfixr to find a fix. The patches are listed in the appendix.

		grammar				tests		gfixr			applied patches					
#	$\mathcal{L}$	$\|N\|$	$\|T\|$	$\|P\|$	bugs	$\|TS_{\mathcal{L}}\|$	fails	iter.	cand.	time	$\mathfrak{d}$	$\mathfrak{i}$	$\mathfrak{s}$	$\mathfrak{t}$	$\mathfrak{L}$	
1	$\mathcal{T}oy$	36	32	68	2	69	3	2	32	00:01:57			2			⋆
2	$\mathcal{A}$	46	42	102	1	179	3	1	23	00:01:20				1		⋆
3	$\mathcal{A}$	49	43	107	1	179	2	1	94	00:05:14			1			⋆
4	$\mathcal{B}$	45	42	88	2	78	2	2	16	00:00:52					2	⋆
5	$\mathcal{C}$	35	27	60	1	86	1	2	6	00:00:30		2				✓
6	$\mathcal{D}$	45	30	78	1	79	3	1	6	00:00:23	1					⋆
7	$\mathcal{E}$	46	24	79	4	80	7	22	250	00:20:00	3	5				⋆
8	$\mathcal{E}$	47	32	84	4	80	17	10	316	00:21:36	4		4		2	✓
9	$\mathcal{F}$	39	46	96	2	212	26	5	177	00:11:15	2	2			1	⋆
10	$\mathcal{F}$	49	72	145	> 5	212	58	260	36924	∞						✗

tests and the grammars revealed that, the ANTLR grammar has explicit function and procedure types

```
typeDefinitionPart: TYPE (typeDefinition SEMI)+;
typeDefinition:
  id EQUAL (type | functionType | procedureType);
```

while the CUP grammar allows them in formal parameter lists:

```
type_def    ::= ID EQ type;
type        ::= simple_type | PACKED struct
              | struct | CAP ID;
simple_type ::= LPAR idlist RPAR | ... ;
struct      ::= ... | FILE OF type;
```

We manually reduced the test suite and removed failing tests which did not expose this type deviation, in order to focus the repair on the types. This leaves us with a repair specification in form of 199 positive test cases.

This experiment was performed inside a Docker container on a 2.70 GHz server with 72 cores and 376GB RAM.

Results. The localizer identified the item `type_def` $\longrightarrow$ `id EQ` $\bullet$ `type` as most suspicious, from *ef* and *ep* counts of 25 and 2, respectively. Starting from this, gfixr learned in 11 iterations a patch that captures the previously failed tests. The fragment of the candidate patch is shown below.

```
type_def    ::= ID EQ PROCEDURE LPAR formal_p_sects RPAR
              | ID EQ PROCEDURE | ID EQ FUNCTION type
              | ID EQ FUNCTION COLON type | ID EQ type;
type        ::= simple_type | PACKED struct
              | struct | CAP ID;
simple_type ::= LPAR idlist colons RPAR colons
              | LPAR idlist RPAR | ... ;
struct      ::= ... | FILE OF type | FILE;
```

It took gfixr about 21 minutes to find the fix, generating a total of 317 grammar variants.

8.3 Learning a Language Extension

In a final experiment we used gfixr in a structurally more complex scenario. We removed from $G_{\mathcal{T}oy}$ the alternatives for the *cond*-symbol that involve **`else`**- and **`elsif`**-tokens, leaving conditionals only with the basic **`if`**-**`then`**-**`end`** structure, and tried to find a repair for this variant that "re-learns" the dropped structure. However, the highly recursive nature of the *elsif*-rule posed a challenge for the transformations that gfixr currently implements, because they cannot create a new rule from the middle of another rule to handle recursion. We therefore manually added an unreachable seed rule *elsifs* $\longrightarrow$ **`elsif`**. The resulting modified variant $G'_{\mathcal{T}oy}$ has thus the following rules:

$$\begin{aligned} cond &\longrightarrow \texttt{if}\ expr\ \texttt{then}\ stmts\ \texttt{end} \\ elsifs &\longrightarrow \texttt{elsif} \end{aligned}$$

For the repair we used the same test suite with 442 tests we use to construct bigrams for the on-the-fly patch validation. This satisfies *adjacent pair coverage* (i.e., it contains a word w with $S \Rightarrow^* \alpha XY\omega \Rightarrow^* w$ for all X, Y with $Y \in \text{follow}(X)$) and thus provides more syntactic variance than $TS_{\mathcal{T}oy}$. We conducted this experiment on the same machine used to obtain results shown in Table 1.

Results. The initial grammar failed 56 test cases. The faulty item $cond \longrightarrow$ **`if`** *expr* **`then`** *stmts* $\bullet$ **`end`** is ranked within the top 1% of all items, with its bad token set composed of {**`elsif`**, **`else`**}. The *elsifs* seed rule is not executed and has a suspiciousness score of zero in the first iteration.

gfixr finds a fix in about 56 minutes that passes all tests after 59 generations. In the process, it generated a total of 937 candidate grammars. Below is the fragment of the generated grammar that is consistent with the test suite.

```
cond → if expr then stmts elsif simple then relax elsifs end
     | if expr then stmts elsif simple then relax end
     | if expr then stmts else stmts end
     | if expr then stmts end
elsifs → else relax
     | elsif num then stmts
```

Note that gfixr has re-learned the core structure of the deleted rules, but not entirely re-created them. There are two shortcomings. First, some symbols are not general enough (e.g., *simple* instead of *expr* and **relax** instead of *stmts* in the first two *cond*-rules). Second, gfixr is missing the list structure of **elsif**-clauses. Both can be seen as a shortcoming of the test suite, since the observed good resp. bad tokens remain too restricted to allow further generalizations. A larger test suite or weaker conditions could allow gfixr to find the expected generalizations, although can lead to overgeneralizations elsewhere. However, since the grammar is fixed using the original names, a manual clean-up is feasible.

9 Related Work

Grammar Transformations. Lämmel and Zaytsev [30, 67, 68] have defined general grammar transformations and used them for grammar construction, refactoring, and adaptation [33, 69], including the extraction and comparison of several complete grammars from different language specifications [32, 34]. Jain et. al [22] propose a semi-automatic approach for building new rules starting from an approximate grammar and a knowledge base of common grammar constructs. However, this work relies on a human expert to select from a large number of expressive grammar transformations. Our approach, in contrast, is fully automatic.

Genetic Grammar Learning. Genetic algorithms (GA) have also been used to learn CFGs from test suites. The applied genetic operations include point mutations such as replacement, insertion, or deletion of symbols [48] and modification of EBNF operators [8] in a single rule, global mutations such as merging and splitting of non-terminal symbols [49], mutated rule duplication [48], or different rule generalizations [49], and different crossovers where rules from one grammar are spliced into the other. Our transformations are similar to those mutations, but we give explicit, static conditions for their viability, and immediately validate them against the sample bigrams, which reduces the number of possible applications; note that sample bigram validation is only useful in repair, where the parent grammar is already a good approximation of the target language. We do not use crossovers, because we repair a single initial grammar and all candidate grammars have been derived from this, so that crossovers do not add diversity.

The fitness of a grammar is evaluated, as in our approach, by running the corresponding over the test suite; in practice, results can improve if positive examples get priority, but negative examples are required to prevent over-generalization [8]. Scoring functions are typically based on some version of balanced accuracy, sometimes taking the length of the longest recognized fragment into account [35]. Our priority function follows similar ideas.

Di Penta et al. [48] used GAs to learn the well-separated extension of a programming language, starting from the full grammar of the base language. Section 8.3 shows that this can be achieved with our approach as well.

Inductive Grammar Learning. Our work can be seen as grammatical inference, which has a long history (e.g., [58]) and has been widely addressed, both in theory and in practice (see [9, 36, 57, 59] for overviews).

Our approach has the full test suite available, but no teacher. It therefore sits between Gold's model of *identification in the limit* [13], where observations are presented in sequence (and approaches are often order-sensitive, e.g., [28]) and Angluin's *query model* [3], where the learner can ask the teacher membership and equivalence queries and use the teacher's response in guiding the learning process. However, since we are given an initial grammar, we are solving a simpler problem than learning the full grammar from scratch. We focus on learning from unstructured text (*textual presentation*) because we cannot use the grammar under repair to construct parse tree skeletons (*structural presentation*), from which only the labels need to be learned [11, 57].

Most complete learning algorithms work for regular languages only, where all necessary properties (e.g., language equivalence) are decidable, but some work carries over to restricted subclasses of context-free languages [21]. We focus on heuristic approaches here.

Several systems such as Synapse [44, 45] or Gramin [55] iteratively parse the positive tests using the current grammar; when an attempt fails, they introduce a new rule to match this input. Synapse uses the negative presentation after each generalization to prevent overgeneralizations. Gramin adds some heuristics to reduce the search space.

Glade [6] implements a two phase generate-and-test approach comprising a regular expression generalization (which introduces alternatives and repetitions), followed by a CFG generalization (which introduces recursions); repetition and recursion introduction are somewhat similar to Solomonoff's approach [58]. Glade also generates specific check words from the generalized locations to reject candidates (similar to our bigram-based validation), but this relies on a teacher. Glade has been used to successfully learn useful approximations of some production grammars and represents the current state-of-the-art in CFG inference.

Grammar-based Test Suite Generation. Since we repair a CFG against a finite test suite, we need to ensure that this covers the syntactic structure of the target language $\mathcal{L}$ well. In some application scenarios (e.g., education, grammar migration, or language modification) we can take advantage of a grammar for $\mathcal{L}$ that may be available but not accessible to the developers (i.e., students) or sufficient (e.g., in the wrong formalism), and automatically generate a test suite.

Several algorithms yield sufficiently detailed test suites that strike the right balance between syntactic regularity and variation, e.g., CDRC [31], k-path coverage [16], derivable pair coverage [60], or automata-based methods [54, 70].

Grammar-based fuzzers (e.g., LangFuzz [17] and IFuzzer [61]) mostly use random sentence generation techniques, and often exploit a given corpus to extract seed code fragments [17, 61, 62]. Nautilus [4] exploits grey-box access to the SUT to provide feedback to the sentence generation. These systems all assume that a correct grammar is available; AUTOGRAM [18, 19] uses dynamic tainting to produce a CFG for the input language but this again requires grey-box access to the SUT. In parser-directed fuzzing [42], the parser itself is used to guide the sentence generation. Mimid [14] extends this to extract an explicit CFG.

Automatic Program Repair. Automated program repair comes in different flavours, e.g., using symbolic execution or data-driven techniques. Like gfixr, many approaches are based on generating candidate patches using different strategies such as genetic programming [15, 64], semantic code search [26], or bug templates [7, 20, 27, 29, 37–39, 41, 56], and validating each generated patch over a test suite.

Fault localization plays a key role in such generate-and-validate approaches, because identifying potentially faulty code fragments reduces the amount of possible repair sites that need to be validated. GenProg [15] uses a simple fault localization technique where statements that are executed by failing (resp. passing) tests only are assigned a score of one (resp. zero), and statement executed by both failing and passing tests a fractional value. SearchRepair [26] uses Tarantula [24], a common and widely used spectrum-based fault localization metric. gfixr uses the Ochiai [46] metric (see Section 2), another common technique, but can be easily extended to work with other metrics.

In principle, we could use program repair tools directly on the parser's implementation of the grammar. However, our approach presents several advantages. Fixing the parser code directly is impossible for table-driven implementations, and induces much larger fix spaces for recursive descent parsers, due to the lower level of abstraction. Moreover, it does not help in applications where the grammar itself must be fixed, e.g., grammar-based fuzzing.

10 Conclusion

We have described the first approach to automatically repair bugs in context-free grammars. Our approach alternates over two key steps and gradually improves the grammar until it passes all tests in a given test suite: (*i*) We use fine-grained spectrum-based fault localization to identify suspicious items (i.e., specific positions in rules) as potential repair sites. (*ii*) We use small-scale transformations to patch the grammar and formulate with each transformation explicit pre- and post-conditions that are necessary for it to improve the grammar. Both steps significantly reduce the number of potential repairs, compared to prior approaches to apply GAs for grammar learning. We further use a priority queue to keep improving the most promising candidate grammars.

We implemented this approach in the gfixr system, and successfully used it to fix CUP grammars that students submitted as homeworks in a compiler engineering course, and to map one Pascal dialect grammar against another dialect.

In our experiments, gfixr worked well when the grammar developers were indeed competent, i.e., when the initial grammars were good approximations of the target grammars. However, for the simultaneous repair of multiple faults, or for large repairs that require many patches, the search spaces become large, and the process becomes slow. Moreover, it can become sensitive to the test suite and the different heuristics, and in the worst cases, the repair failed.

Future Work. We plan to extend gfixr to repair grammars for LL-parsers such as JavaCC or ANTLR, and possibly even for generalized GLR or GLL parsers, and to run more experiments to evaluate the effect of different test suites, but we see several directions beyond that to improve our work.

Partial repairs using insertion or substitution patches can introduce multiple mutated copies of the same base rule. We plan to clean up the fixed grammar using grammar refactorings (e.g., introducing new non-terminals for alternatives or common sub-sequences) [30, 67].

Many bugs (especially by students) emerge at the interface between lexer and parser, due to interactions between the lexer's *first* and *longest match* policies. Fixing such bugs is easy in principle (e.g., a new keyword can be introduced through a substitution patch), but the automation is more complex because lexer and parser need to be updated synchronously. We plan to extend gfixr accordingly, or alternatively, use a scannerless parsing approach [12].

We will also investigate an active repair approach, where we assume a parser for the unknown target language that can answer membership queries and serve as teacher in the sense of Angluin's query model [3]. The key extension is to replace bigram-based patch validation by a test suite enrichment, where we judiciously generate (positive and negative) tests [52] from the patch candidates, and use the parser to obtain the expected outcome.

Acknowledgements

The financial assistance of the National Research Foundation (NRF) under Grant 113364 towards this research is hereby acknowledged.

Table 2. Spectral counts, Ochiai scores and ranks for $G_{\mathcal{Toy}}$ over $TS_{\mathcal{Toy}}$.

item	*ef*	*ep*	*nf*	*np*	score	rank	
program:1:0	3	65	0	8	0.21	=12	-
program:1:1	3	65	0	8	0.21	=12	-
program:1:2	3	65	0	8	0.21	=12	6
program:2:0	3	8	0	65	0.52	=7	-
program:2:1	3	8	0	65	0.52	=7	-
program:2:2	3	8	0	65	0.52	=7	4
body:1:1	3	67	0	6	0.20	15	7
body:2:1	3	6	0	67	0.58	6	3
name:1:0	3	9	0	64	0.50	=10	-
name:1:1	3	9	0	64	0.50	=10	5
name:2:1	3	1	0	72	0.87	=4	-
name:2:2	3	1	0	72	0.87	=4	2
name:3:0	3	0	0	73	1.00	=1	-
name:3:1	3	0	0	73	1.00	=1	-
name:3:2	3	0	0	73	1.00	=1	1

A Running Example Grammar

We mostly draw on examples from teaching in this paper, since students tend to make many mistakes that can be fixed. In particular, we use grammars from a compiler engineering course where students were given a grammar for the $\mathcal{Toy}$ language in EBNF form, and had to develop corresponding CUP parsers. We use the instructor's grammar shown in Fig. 1 as baseline and retrofit errors made by students in their own submitted grammars to this baseline to illustrate our approach.

Note that the language exhibits some quirks, most notably that formal parameter and argument lists cannot be empty, that call expressions are allowed in *lvalue* positions, and, conversely, that *lvalues* (i.e., simple identifiers and indexing expressions) are allowed as statements.

prog	→ **`program`** `id` *body*
	| **`program`** `id` *fdecllist body*
fdecllist	→ *fdecl* | *fdecl fdecllist*
fdecl	→ **`define`** `id` **`(`** *paramlist* **`)`** *body*
	| **`define`** `id` **`(`** *paramlist* **`)`** **`->`** *type* `id` *body*
paramlist	→ *param* | *param* **`,`** *paramlist*
param	→ *type* `id` | *type* **`array`** `id`
type	→ **`boolean`** | **`int`**
body	→ **`begin`** *stmts* **`end`**
	| **`begin`** *vdecllist stmts* **`end`**
vdecllist	→ *vdecl* | *vdecl vdecllist*
vdecl	→ *type idlist* **`;`** | *type* **`array`** *idlist* **`;`**
idlist	→ `id` | `id` **`,`** *idlist*
stmts	→ **`relax`** | *stmtlist*
stmtlist	→ *stmt* | *stmt* **`;`** *stmtlist*
stmt	→ *assign* | *cond* | *input* | **`leave`** | *output* | *loop*
assign	→ *name* | *name* **`::=`** *expr* | *name* **`::=`** **`array`** *simple*
cond	→ **`if`** *expr* **`then`** *stmts* **`end`**
	| **`if`** *expr* **`then`** *stmts elsiflist* **`end`**
	| **`if`** *expr* **`then`** *stmts* **`else`** *stmts* **`end`**
	| **`if`** *expr* **`then`** *stmts elsiflist* **`else`** *stmts* **`end`**
elsiflist	→ **`elsif`** *expr* **`then`** *stmts*
	| **`elsif`** *expr* **`then`** *stmts elsiflist*
input	→ **`read`** *name*
output	→ **`write`** *elemlist*
elemlist	→ *elem* | *elem* **`.`** *elemlist*
elem	→ `string` | *expr*
loop	→ **`while`** *expr* **`do`** *stmts* **`end`**
expr	→ *simple* | *simple relop simple*
relop	→ **`=`** | **`>=`** | **`>`** | **`<=`** | **`<`** | **`/=`**
simple	→ **`-`** *termlist* | *termlist*
termlist	→ *term* | *term addop termlist*
addop	→ **`-`** | **`or`** | **`+`**
term	→ *factorlist*
factorlist	→ *factor* | *factor mulop factorlist*
mulop	→ **`and`** | **`/`** | **`*`** | **`rem`**
factor	→ *name* | `num` | **`(`** *expr* **`)`** | **`not`** *factor* | **`true`** | **`false`**
name	→ `id` | `id` **`[`** *simple* **`]`** | `id` **`(`** *arglist* **`)`**
arglist	→ *expr* | *expr* **`,`** *arglist*

Figure 1. BNF baseline grammar $G_{\mathcal{Toy}}$ suitable for CUP. We also use *italics* and **`bold typewriter`** font for non-terminal and terminal symbols, respectively; we use `normal typewriter` font for structured tokens with different instances such as identifiers.

B Localization Example

Table 2 shows the counts aggregated from the grammar spectrum that we collected by running the CUP parser generated from the example grammar $G'_{\mathcal{Toy}}$ (see Section 1) over the test suite $TS_{\mathcal{Toy}}$, as well as the Ochiai scores and corresponding ranks for the items $p^\bullet$ with $ef(p^\bullet) > 0$. Here, $A{:}n{:}m$ denotes the item $A \longrightarrow \alpha \bullet \omega$ from the n-th alternative production for A where $|\alpha| = m$. The Ochiai score of an item $p^\bullet$ is given by

$$\mathrm{score}(p^\bullet) = \frac{ef(p^\bullet)}{\sqrt{(ef(p^\bullet) + nf(p^\bullet)) \times (ef(p^\bullet) + ep(p^\bullet))}}$$

The last two columns show the items ranked by score. On the left, all elements are ranked, with ties indicated by a preceding "="; on the right, ties between items from the same rule are resolved as described in Section 2.3.

Note that the localization phase identifies only 7 out of 172 items as suspicious; this substantially reduces the number of patches attempted, and is a main reason for the good performance of our approach. Moreover, it ranks the actual fault location as the most suspicous amongst those seven locations and tries to patch there first, thus prioritizing the eventual fix, but not shutting out other options.

Note further that the first fault blocks the second fault in *namelist*, and the corresponding item is scored zero in the first iteration; however, after the first partial repair, this one is ranked highest, leading to a fix of both faults in just two patches.

C Patches for Student Grammars

In this appendix, we show for each grammar used in Section 8.1 the faulty fragments from the students' grammars and the corresponding fragments as patched by gfixr.

Submission #1 (𝒯oy).

Faulty grammar fragments:

```
name     ::= ID RPAR name namelist RPAR;
namelist ::= namelist COMMA name | ϵ;
```

gfixr patched fragments:

```
name     ::= ID RPAR expr namelist RPAR;
namelist ::= namelist COMMA expr | ϵ;
```

Submission #2 (𝒜).

Faulty grammar fragments:

```
startVarIDs ::= IDENT variableIDs;
variableIDs ::= IDENT COMMA variableIDs | ϵ;
```

gfixr patched fragments:

```
startVarIDs ::= IDENT variableIDs;
variableIDs ::= COMMA IDENT variableIDs | ϵ;
```

Submission #3 (𝒜).

Faulty grammar fragments:

```
vdecl_more ::= vdecl_more COMMA vdecl | ϵ;
```

gfixr patched fragments:

```
vdecl_more ::= vdecl_more SEMI vdecl | ϵ;
```

Submission #4 (ℬ).

Faulty grammar fragments:

```
vardef   ::= type IDENT identList SEMI  | ϵ;
stmtList ::= SEMI stmt | ϵ;
```

gfixr patched fragments

```
vardef   ::= type IDENT identList SEMI vardef  | ϵ;
stmtList ::= SEMI stmt stmtList | ϵ;
```

Submission #5 (𝒞).

Faulty grammar fragments:

```
zeroormanysimple ::= ADDOP term zeroormanysimple | ϵ;
factor ::= NUM | ...;
```

gfixr patched fragments:

```
zeroormanysimple ::= ADDOP term zeroormanysimple  | ϵ;
factor ::= NUM NEGATE factor | NUM | ...;
```

Submission #6 (𝒟).

Faulty grammar fragments:

```
factor ::= ID LBRAC simple RBRAC
         | ID LPAR expr exprlist RPAR
         | LPAR expr RPAR
         | NOT factor
         | INT | TRUE | FALSE;
```

gfixr patched fragments:

```
factor ::= ID LBRAC simple RBRAC
         | ID LPAR expr exprlist RPAR
         | ID
         | LPAR expr RPAR
         | NOT factor
         | INT | TRUE | FALSE;
```

Submission #7 (ℰ).

Faulty grammar fragments:

```
call   ::= CALL ID LPAR comma_expr RPAR;
output ::= PUT STRING DOT STRING | ...
factor ::= TRUE | FALSE | NOT factor | NUM | ID
         | ID LBRAC simple RBRAC
         | ID LPAR comma_expr RPAR;
```

gfixr patched fragments:

```
call   ::= CALL ID LPAR RPAR
         | CALL ID LPAR comma_expr RPAR;
output ::= PUT STRING DOT STRING DOT STRING
         | PUT STRING DOT STRING | ...
factor ::= TRUE | FALSE | NOT factor | NUM | ID
         | LPAR NUM RPAR
         | ID LBRAC simple RBRAC
         | ID LPAR RPAR
         | ID LPAR comma_expr RPAR;
```

Submission #8 (ℰ).

Faulty grammar fragments:

```
vardef ::= ϵ
         | type ID COMMA
         | vardef_list SEMICOLON ;
vardef_list ::= vardef_list COMMA type ID;
assign ::= ID SQR_LEFT simple SQR_RIGHT ASSIGN expr
         | ID SQR_LEFT simple SQR_RIGHT ASSIGN ARRAY
             simple
         | ID ASSIGN expr
         | ID ASSIGN ARRAY simple;
factor ::= ID BRACKET_LEFT expr_list BRACKET_RIGHT
         | ID SQR_LEFT simple SQR_RIGHT
         | NUM
         | BRACKET_LEFT expr BRACKET_RIGHT
         | NOT factor
         | TRUE | FALSE;
string ::= QUOTE STRING QUOTE;
```

gfixr patched fragments:

```
vardef ::= ϵ
         | type ID COMMA vardef
         | type ID SEMICOLON vardef
         | vardef_list SEMICOLON ;
vardef_list ::= vardef_list COMMA type ID;
assign ::= ID SQR_LEFT simple SQR_RIGHT ASSIGN expr
         | ID SQR_LEFT simple SQR_RIGHT ASSIGN ARRAY
             simple
         | ID COMMA expr
         | ID ASSIGN expr
         | ID
         | ID ASSIGN ARRAY simple;
factor ::= ID BRACKET_LEFT BRACKET_RIGHT
         | ID BRACKET_LEFT expr_list BRACKET_RIGHT
         | ID SQR_LEFT simple SQR_RIGHT
         | ID
         | NUM
         | BRACKET_LEFT expr BRACKET_RIGHT
         | NOT factor
         | TRUE | FALSE;
string ::= STRING
         | QUOTE STRING QUOTE;
```

Submission #9 ($\mathcal{F}$).

Faulty grammar fragments:

```
varlist    ::= var SEMI varlist | ϵ;
seg        ::= MINUS term termlist
             | PLUS term termlist;
```

gfixr patched fragments:

```
varlist    ::= var SEMI varlist | var | ϵ;
seg        ::= MINUS MINUS term termlist
             | MINUS term termlist
             | PLUS term termlist;
```

References

[1] Rui Abreu, Peter Zoeteweij, and Arjan J. C. van Gemund. 2006. An Evaluation of Similarity Coefficients for Software Fault Localization. In *12th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC 2006), 18-20 December, 2006, University of California, Riverside, USA*. IEEE Computer Society, 39–46. https://doi.org/10.1109/PRDC.2006.18

[2] Rui Abreu, Peter Zoeteweij, and Arjan J. C. van Gemund. 2009. Spectrum-Based Multiple Fault Localization. In *ASE 2009, 24th IEEE/ACM International Conference on Automated Software Engineering, Auckland, New Zealand, November 16-20, 2009*. IEEE Computer Society, 88–99. https://doi.org/10.1109/ASE.2009.25

[3] Dana Angluin. 1987. Queries and Concept Learning. *Mach. Learn.* 2, 4 (1987), 319–342. https://doi.org/10.1007/BF00116828

[4] Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. 2019. NAUTILUS: Fishing for Deep Bugs with Grammars. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. The Internet Society. https://www.ndss-symposium.org/ndss-paper/nautilus-fishing-for-deep-bugs-with-grammars/

[5] Chelsea Barraball, Moeketsi Raselimo, and Bernd Fischer. 2020. An interactive feedback system for grammar development (tool paper). In *Proceedings of the 13th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2020, Virtual Event, USA, November 16-17, 2020*, Ralf Lämmel, Laurence Tratt, and Juan de Lara (Eds.). ACM, 101–107. https://doi.org/10.1145/3426425.3426935

[6] Osbert Bastani, Rahul Sharma, Alex Aiken, and Percy Liang. 2017. Synthesizing program input grammars. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, Albert Cohen and Martin T. Vechev (Eds.). ACM, 95–110. https://doi.org/10.1145/3062341.3062349

[7] Mike Y. Chen, Emre Kiciman, Eugene Fratkin, Armando Fox, and Eric A. Brewer. 2002. Pinpoint: Problem Determination in Large, Dynamic Internet Services. In *2002 International Conference on Dependable Systems and Networks (DSN 2002), 23-26 June 2002, Bethesda, MD, USA, Proceedings*. 595–604. https://doi.org/10.1109/DSN.2002.1029005

[8] Matej Crepinsek, Marjan Mernik, Barrett R. Bryant, Faizan Javed, and Alan P. Sprague. 2005. Inferring Context-Free Grammars for Domain-Specific Languages. *Electron. Notes Theor. Comput. Sci.* 141, 4 (2005), 99–116. https://doi.org/10.1016/j.entcs.2005.02.055

[9] Colin de la Higuera. 2010. *Grammatical Inference: Learning Automata and Grammars*. Cambridge University Press.

[10] Richard A. DeMillo, Richard J. Lipton, and Frederick G. Sayward. 1978. Hints on Test Data Selection: Help for the Practicing Programmer. *Computer* 11, 4 (1978), 34–41. https://doi.org/10.1109/C-M.1978.218136

[11] Frank Drewes and Johanna Högberg. 2003. Learning a Regular Tree Language from a Teacher. In *Developments in Language Theory, 7th International Conference, DLT 2003, Szeged, Hungary, July 7-11, 2003, Proceedings (Lecture Notes in Computer Science, Vol. 2710)*, Zoltán Ésik and Zoltán Fülöp (Eds.). Springer, 279–291. https://doi.org/10.1007/3-540-45007-6_22

[12] Giorgios Economopoulos, Paul Klint, and Jurgen J. Vinju. 2009. Faster Scannerless GLR Parsing. In *Compiler Construction, 18th International Conference, CC 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings (Lecture Notes in Computer Science, Vol. 5501)*, Oege de Moor and Michael I. Schwartzbach (Eds.). Springer, 126–141. https://doi.org/10.1007/978-3-642-00722-4_10

[13] E. Mark Gold. 1967. Language Identification in the Limit. *Inf. Control.* 10, 5 (1967), 447–474. https://doi.org/10.1016/S0019-9958(67)91165-5

[14] Rahul Gopinath, Björn Mathis, and Andreas Zeller. 2020. Mining input grammars from dynamic control flow. In *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, Prem Devanbu, Myra B. Cohen, and Thomas Zimmermann (Eds.). ACM, 172–183. https://doi.org/10.1145/3368089.3409679

[15] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2012. GenProg: A Generic Method for Automatic Software Repair. *IEEE Trans. Software Eng.* 38, 1 (2012), 54–72. https://doi.org/10.1109/TSE.2011.104

[16] Nikolas Havrikov and Andreas Zeller. 2019. Systematically Covering Input Structure. In *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019*. IEEE, 189–199. https://doi.org/10.1109/ASE.2019.00027

[17] Christian Holler, Kim Herzig, and Andreas Zeller. 2012. Fuzzing with Code Fragments. In *Proceedings of the 21th USENIX Security Symposium, Bellevue, WA, USA, August 8-10, 2012*, Tadayoshi Kohno (Ed.). USENIX Association, 445–458. https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/holler

[18] Matthias Höschele and Andreas Zeller. 2016. Mining input grammars from dynamic taints. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE 2016)*, David Lo, Sven Apel, and Sarfraz Khurshid (Eds.). ACM, New York, NY, USA, 720–725. https://doi.org/10.1145/2970276.2970321

[19] Matthias Höschele and Andreas Zeller. 2017. Mining input grammars with AUTOGRAM. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017 - Companion Volume*, Sebastián Uchitel, Alessandro Orso, and Martin P. Robillard (Eds.). IEEE Computer Society, 31–34. https://doi.org/10.1109/ICSE-C.2017.14

[20] Jinru Hua, Mengshi Zhang, Kaiyuan Wang, and Sarfraz Khurshid. 2018. Towards practical program repair with on-demand candidate generation. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman (Eds.). ACM, 12–23. https://doi.org/10.1145/3180155.3180245

[21] Malte Isberner. 2015. *Foundations of active automata learning: an algorithmic perspective*. Ph.D. Dissertation. Technical University Dortmund, Germany. http://hdl.handle.net/2003/34282

[22] Rahul Jain, Sanjeev Kumar Aggarwal, Pankaj Jalote, and Shiladitya Biswas. 2004. An interactive method for extracting grammar from programs. *Softw. Pract. Exp.* 34, 5 (2004), 433–447. https://doi.org/10.1002/spe.568

[23] James A. Jones and Mary Jean Harrold. 2005. Empirical evaluation of the tarantula automatic fault-localization technique. In *20th IEEE/ACM International Conference on Automated Software Engineering (ASE 2005), November 7-11, 2005, Long Beach, CA, USA*, David F. Redmiles, Thomas Ellman, and Andrea Zisman (Eds.). ACM, 273–282. https://doi.org/10.1145/1101908.1101949

[24] James A. Jones, Mary Jean Harrold, and John T. Stasko. 2002. Visualization of test information to assist fault localization. In *Proceedings of the 24th International Conference on Software Engineering, ICSE 2002, 19-25 May 2002, Orlando, Florida, USA*, Will Tracz, Michal Young, and Jeff Magee (Eds.). ACM, 467–477. https://doi.org/10.1145/581339.581397

[25] Alan Kaplan and Denise Shoup. 2000. CUPV - a visualization tool for generated parsers. In *Proceedings of the 31st SIGCSE Technical Symposium on Computer Science Education, SIGCSE 2000, Austin, Texas, USA, March 7-12, 2000*, Lillian (Boots) Cassel, Nell B. Dale, Henry MacKay Walker, and Susan M. Haller (Eds.). ACM, 11–15. https://doi.org/10.1145/330908.331801

[26] Yalin Ke, Kathryn T. Stolee, Claire Le Goues, and Yuriy Brun. 2015. Repairing Programs with Semantic Code Search (T). In *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015*, Myra B. Cohen, Lars Grunske, and Michael Whalen (Eds.). IEEE Computer Society, 295–306. https://doi.org/10.1109/ASE.2015.60

[27] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. 2013. Automatic patch generation learned from human-written patches. In *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, David Notkin, Betty H. C. Cheng, and Klaus Pohl (Eds.). IEEE Computer Society, 802–811. https://doi.org/10.1109/ICSE.2013.6606626

[28] Bruce Knobe and Kathleen Knobe. 1976. A Method for Inferring Context-free Grammars. *Inf. Control.* 31, 2 (1976), 129–146. https://doi.org/10.1016/S0019-9958(76)80003-4

[29] Anil Koyuncu, Kui Liu, Tegawendé F. Bissyandé, Dongsun Kim, Jacques Klein, Martin Monperrus, and Yves Le Traon. 2020. FixMiner: Mining relevant fix patterns for automated program repair. *Empir. Softw. Eng.* 25, 3 (2020), 1980–2024. https://doi.org/10.1007/s10664-019-09780-z

[30] Ralf Lämmel. 2001. Grammar Adaptation. In *FME 2001: Formal Methods for Increasing Software Productivity, International Symposium of Formal Methods Europe, Berlin, Germany, March 12-16, 2001, Proceedings (Lecture Notes in Computer Science, Vol. 2021)*, José Nuno Oliveira and Pamela Zave (Eds.). Springer, 550–570. https://doi.org/10.1007/3-540-45251-6_32

[31] Ralf Lämmel. 2001. Grammar Testing. In *Fundamental Approaches to Software Engineering, 4th International Conference, FASE 2001 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001 Genova, Italy, April 2-6, 2001, Proceedings (Lecture Notes in Computer Science, Vol. 2029)*, Heinrich Hußmann (Ed.). Springer, 201–216. https://doi.org/10.1007/3-540-45314-8_15

[32] Ralf Lämmel and Chris Verhoef. 2001. Semi-automatic grammar recovery. *Softw. Pract. Exp.* 31, 15 (2001), 1395–1438. https://doi.org/10.1002/spe.423

[33] Ralf Lämmel and Vadim Zaytsev. 2009. An Introduction to Grammar Convergence. In *Integrated Formal Methods, 7th International Conference, IFM 2009, Düsseldorf, Germany, February 16-19, 2009. Proceedings (Lecture Notes in Computer Science, Vol. 5423)*, Michael Leuschel and Heike Wehrheim (Eds.). Springer, 246–260. https://doi.org/10.1007/978-3-642-00255-7_17

[34] Ralf Lämmel and Vadim Zaytsev. 2009. Recovering Grammar Relationships for the Java Language Specification. In *Ninth IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2009, Edmonton, Alberta, Canada, September 20-21, 2009*. IEEE Computer Society, 178–186. https://doi.org/10.1109/SCAM.2009.29

[35] Marc M. Lankhorst. 1994. Grammatical Inference with a Genetic Algorithm. In *Massively Parallel Processing Applications and Develompent, Proceedings of the 1994 EUROSIM Conference on Massively Parallel Processing Applications and Develompent, 21-23 June 1994, Delft, The Netherlands*, Len Dekker, Wim Smit, and Jan C. Zuidervaart (Eds.). Elsevier, 423–430.

[36] Lillian Lee. 1996. *Learning of Context-Free Languages: A Survey of the Literature.* Technical Report Computer Science Group Technical Report TR-12-96. Harvard University.

[37] Kui Liu, Anil Koyuncu, Tegawendé F. Bissyandé, Dongsun Kim, Jacques Klein, and Yves Le Traon. 2019. You Cannot Fix What You Cannot Find! An Investigation of Fault Localization Bias in Benchmarking Automated Program Repair Systems. In *12th IEEE Conference on Software Testing, Validation and Verification, ICST 2019, Xi'an, China, April 22-27, 2019*. IEEE, 102–113. https://doi.org/10.1109/ICST.2019.00020

[38] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F. Bissyandé. 2019. TBar: revisiting template-based automated program repair. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019, Beijing, China, July 15-19, 2019*, Dongmei Zhang and Anders Møller (Eds.). ACM, 31–42. https://doi.org/10.1145/3293882.3330577

[39] Xuliang Liu and Hao Zhong. 2018. Mining stackoverflow for program repair. In *25th International Conference on Software Analysis, Evolution and Reengineering, SANER 2018, Campobasso, Italy, March 20-23, 2018*, Rocco Oliveto, Massimiliano Di Penta, and David C. Shepherd (Eds.). IEEE Computer Society, 118–129. https://doi.org/10.1109/SANER.2018.8330202

[40] Ravichandhran Madhavan, Mikaël Mayer, Sumit Gulwani, and Viktor Kuncak. 2015. Automating grammar comparison. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*, Jonathan Aldrich and Patrick Eugster (Eds.). ACM, 183–200. https://doi.org/10.1145/2814270.2814304

[41] Matias Martinez and Martin Monperrus. 2018. Ultra-Large Repair Search Space with Automatically Mined Templates: The Cardumen Mode of Astor. In *Search-Based Software Engineering - 10th International Symposium, SSBSE 2018, Montpellier, France, September 8-9, 2018, Proceedings (Lecture Notes in Computer Science, Vol. 11036)*, Thelma Elita Colanzi and Phil McMinn (Eds.). Springer, 65–86. https://doi.org/10.1007/978-3-319-99241-9_3

[42] Björn Mathis, Rahul Gopinath, Michaël Mera, Alexander Kampmann, Matthias Höschele, and Andreas Zeller. 2019. Parser-directed fuzzing. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, Kathryn S. McKinley and Kathleen Fisher (Eds.). ACM, 548–560. https://doi.org/10.1145/3314221.3314651

[43] Lee Naish, Hua Jie Lee, and Kotagiri Ramamohanarao. 2011. A model for spectra-based software diagnosis. *ACM Trans. Softw. Eng. Methodol.* 20, 3 (2011), 11:1–11:32. https://doi.org/10.1145/2000791.2000795

[44] Katsuhiko Nakamura. 2006. Incremental Learning of Context Free Grammars by Bridging Rule Generation and Search for Semi-optimum Rule Sets. In *Grammatical Inference: Algorithms and Applications, 8th International Colloquium, ICGI 2006, Tokyo, Japan, September 20-22, 2006, Proceedings (Lecture Notes in Computer Science, Vol. 4201)*, Yasubumi Sakakibara, Satoshi Kobayashi, Kengo Sato, Tetsuro Nishino, and Etsuji Tomita (Eds.). Springer, 72–83. https://doi.org/10.1007/11872436_7

[45] Katsuhiko Nakamura and Takashi Ishiwata. 2000. Synthesizing Context Free Grammars from Sample Strings Based on Inductive CYK Algorithm. In *Grammatical Inference: Algorithms and Applications, 5th International Colloquium, ICGI 2000, Lisbon, Portugal, September 11-13, 2000, Proceedings (Lecture Notes in Computer Science, Vol. 1891)*, Arlindo L. Oliveira (Ed.). Springer, 186–195. https://doi.org/10.1007/978-3-540-45257-7_15

[46] Akira Ochiai. 1957. Zoogeographical studies on the soleoid fishes found in Japan and its neighhouring regions-II. *Bulletin of the Japanese Society of Scientific Fisheries* 22, 9 (1957), 526–530. https://doi.org/10.2331/suisan.22.526

[47] Terence Parr, Sam Harwell, and Kathleen Fisher. 2014. Adaptive LL(*) parsing: the power of dynamic analysis. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014*, Andrew P. Black and Todd D. Millstein (Eds.). ACM, 579–598. https://doi.org/10.1145/2660193.2660202

[48] Massimiliano Di Penta, Pierpaolo Lombardi, Kunal Taneja, and Luigi Troiano. 2008. Search-based inference of dialect grammars. *Soft Comput.* 12, 1 (2008), 51–66. https://doi.org/10.1007/s00500-007-0216-5

[49] Georgios Petasis, Georgios Paliouras, Constantine D. Spyropoulos, and Constantine Halatsis. 2004. eg-GRIDS: Context-Free Grammatical Inference from Positive Examples Using Genetic Search. In *Grammatical Inference: Algorithms and Applications, 7th International Colloquium, ICGI 2004, Athens, Greece, October 11-13, 2004, Proceedings (Lecture Notes in Computer Science, Vol. 3264)*, Georgios Paliouras and Yasubumi Sakakibara (Eds.). Springer, 223–234. https://doi.org/10.1007/978-3-540-30195-0_20

[50] Moeketsi Raselimo and Bernd Fischer. 2019. Spectrum-based fault localization for context-free grammars. In *Proceedings of the 12th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2019, Athens, Greece, October 20-22, 2019*, Oscar Nierstrasz, Jeff Gray, and Bruno C. d. S. Oliveira (Eds.). ACM, 15–28. https://doi.org/10.1145/3357766.3359538

[51] Moeketsi Raselimo and Bernd Fischer. 2021. Precise Spectrum-Based Fault Localization for Grammars. In preparation.

[52] Moeketsi Raselimo, Jan Taljaard, and Bernd Fischer. 2019. Breaking parsers: mutation-based generation of programs with guaranteed syntax errors. In *Proceedings of the 12th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2019, Athens, Greece, October 20-22, 2019*, Oscar Nierstrasz, Jeff Gray, and Bruno C. d. S. Oliveira (Eds.). ACM, 83–87. https://doi.org/10.1145/3357766.3359542

[53] Manos Renieris and Steven P. Reiss. 2003. Fault Localization With Nearest Neighbor Queries. In *18th IEEE International Conference on Automated Software Engineering (ASE 2003), 6-10 October 2003, Montreal, Canada*. IEEE Computer Society, 30–39. https://doi.org/10.1109/ASE.2003.1240292

[54] Christoff Rossouw and Bernd Fischer. 2020. Test case generation from context-free grammars using generalized traversal of LR-automata. In *Proceedings of the 13th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2020, Virtual Event, USA, November 16-17, 2020*, Ralf Lämmel, Laurence Tratt, and Juan de Lara (Eds.). ACM, 133–139. https://doi.org/10.1145/3426425.3426938

[55] Diptikalyan Saha and Vishal Narula. 2011. Gramin: a system for incremental learning of programming language grammars. In *Proceeding of the 4th Annual India Software Engineering Conference, ISEC 2011, Thiruvananthapuram, Kerala, India, February 24-27, 2011*, Arun Bahulkar, K. Kesavasamy, T. V. Prabhakar, and Gautam Shroff (Eds.). ACM, 185–194. https://doi.org/10.1145/1953355.1953380

[56] Ripon K. Saha, Yingjun Lyu, Hiroaki Yoshida, and Mukul R. Prasad. 2017. ELIXIR: effective object oriented program repair. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*, Grigore Rosu, Massimiliano Di Penta, and Tien N. Nguyen (Eds.). IEEE Computer Society, 648–659. https://doi.org/10.1109/ASE.2017.8115675

[57] Yasubumi Sakakibara. 1997. Recent Advances of Grammatical Inference. *Theor. Comput. Sci.* 185, 1 (1997), 15–45. https://doi.org/10.1016/S0304-3975(97)00014-5

[58] Ray J. Solomonoff. 1959. A new method for discovering the grammars of phrase structure languages. In *Information Processing, Proceedings of the 1st International Conference on Information Processing, UNESCO, Paris 15-20 June 1959*. UNESCO (Paris), 285–289.

[59] Andrew Stevenson and James R. Cordy. 2014. A survey of grammatical inference in software engineering. *Sci. Comput. Program.* 96 (2014), 444–459. https://doi.org/10.1016/j.scico.2014.05.008

[60] Phillip van Heerden, Moeketsi Raselimo, Konstantinos Sagonas, and Bernd Fischer. 2020. Grammar-based testing for little languages: an experience report with student compilers. In *Proceedings of the 13th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2020, Virtual Event, USA, November 16-17, 2020*, Ralf Lämmel, Laurence Tratt, and Juan de Lara (Eds.). ACM, 253–269. https://doi.org/10.1145/3426425.3426946

[61] Spandan Veggalam, Sanjay Rawat, István Haller, and Herbert Bos. 2016. IFuzzer: An Evolutionary Interpreter Fuzzer Using Genetic Programming. In *Computer Security - ESORICS 2016 - 21st European Symposium on Research in Computer Security, Heraklion, Greece, September 26-30, 2016, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 9878)*, Ioannis G. Askoxylakis, Sotiris Ioannidis, Sokratis K. Katsikas, and Catherine A. Meadows (Eds.). Springer, 581–601. https://doi.org/10.1007/978-3-319-45744-4_29

[62] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2017. Skyfire: Data-Driven Seed Generation for Fuzzing. In *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*. IEEE Computer Society, 579–594. https://doi.org/10.1109/SP.2017.23

[63] Kaiyuan Wang, Allison Sullivan, and Sarfraz Khurshid. 2018. Automated model repair for Alloy. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*, Marianne Huchard, Christian Kästner, and Gordon Fraser (Eds.). ACM, 577–588. https://doi.org/10.1145/3238147.3238162

[64] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. 2009. Automatically finding patches using genetic programming. In *31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings*. IEEE, 364–374. https://doi.org/10.1109/ICSE.2009.5070536

[65] W. Eric Wong, Vidroha Debroy, Ruizhi Gao, and Yihao Li. 2014. The DStar Method for Effective Software Fault Localization. *IEEE Trans. Reliab.* 63, 1 (2014), 290–308. https://doi.org/10.1109/TR.2013.2285319

[66] Xiaozhen Xue and Akbar Siami Namin. 2013. How Significant is the Effect of Fault Interactions on Coverage-Based Fault Localizations?. In *2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement, Baltimore, Maryland, USA, October 10-11, 2013*. IEEE Computer Society, 113–122. https://doi.org/10.1109/ESEM.2013.22

[67] Vadim Zaytsev. 2009. Language Convergence Infrastructure. In *Generative and Transformational Techniques in Software Engineering III - International Summer School, GTTSE 2009, Braga, Portugal, July 6-11, 2009. Revised Papers (Lecture Notes in Computer Science, Vol. 6491)*, João M. Fernandes, Ralf Lämmel, Joost Visser, and João Saraiva (Eds.). Springer, 481–497. https://doi.org/10.1007/978-3-642-18023-1_16

[68] Vadim Zaytsev. 2010. Recovery, Convergence and Documentation of Languages.

[69] Vadim Zaytsev. 2014. Negotiated Grammar Evolution. *J. Object Technol.* 13, 3 (2014), 1: 1–22. https://doi.org/10.5381/jot.2014.13.3.a1

[70] Sergey V. Zelenov and Sophia A. Zelenova. 2005. Generation of Positive and Negative Tests for Parsers. *Program. Comput. Softw.* 31, 6 (2005), 310–320. https://doi.org/10.1007/s11086-005-0040-6