Passive and Active Automatic Grammar Repair

Moeketsi Raselimo, Bernd Fischer

[a]*Stellenbosch University, Stellenbosch, South Africa*

Abstract

We describe the first approach to automatically repair faults in context-free grammars: given a grammar that fails some tests in a test suite, we iteratively and gradually transform the grammar until it passes all tests. Our core idea is to use 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 implement and evaluate a passive and an active variant of our repair approach. In passive repair, we repair against the fixed input test suite as specification. The key extension of the active repair is to exploit an oracle in the form of a black-box parser for the target language to incorporate a test suite enrichment into the repair. This generates additional tests from each repair candidate and issues membership queries to the oracle to confirm the outcome of each of these tests.

We demonstrate the effectiveness of both repair variants using thirty-three student grammars that contain multiple real faults. We show that both variants are effective in fixing real faults in these grammars but that active repair produces repairs with higher precision than passive repair.

Keywords: Software testing and debugging, Grammars and context-free languages, Fault localization, Repair

1. Introduction

Grammars are software, and can contain bugs like any other software. This is true even for well-curated grammars. Lämmel and Verhoef (2001) found "*more errors than one would expect from a language reference manual*" when analyzing COBOL, and Zaytsev (2010) shows errors and inconsistencies in the language specifications of both Java and C#. Grammar testing (Lämmel, 2001b) can demonstrate the presence of such bugs in grammars and grammar fault localization (Raselimo and Fischer, 2019) can identify rules that are likely to contain bugs, but neither of the two techniques can automatically *repair* bugs and *fix* grammars.

In this paper, we introduce and formalize the grammar repair problem, and present 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.

Motivating example. 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 (Kaplan and Shoup, 2000) grammar specification against a small test suite $TS_{\mathcal{T}oy}$ with positive tests to complement an informal description of the target language $\mathcal{T}oy$. Assume that our grammar $G'_{\mathcal{T}oy}$ is similar to the correct version $G_{\mathcal{T}oy}$ shown in Figure 1, with the exception of the last two rules that have the following form:

name → `id` | `id [` *simple* `]` | `id (` *name namelist* `)`
namelist → *namelist* `,` *name* | ε

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

```
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:

```
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 $G'_{\mathcal{T}oy}$ where this sequence occurs, i.e.,

name → `id` | `id [` *simple* `]` | `id (` • *name namelist* `)`

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.

Email addresses: `moeketsi@acm.org` (Moeketsi Raselimo), `bfischer@sun.ac.za` (Bernd Fischer)

```
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_{Toy} suitable for CUP.

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.,

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

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

Approach. 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. We propose a *generate-and-validate* grammar repair approach that automates the find-and-fix loop illustrated in the example above. This approach constructs a repaired grammar G' from an initial user-provided test suite (used as a repair specification) and a faulty grammar G. At its core, our method involves the following steps:

1. *Localization*: we use spectrum-based fault localization for CFGs (Raselimo and Fischer, 2019, 2023) to identify promising repair sites (i.e., specific positions in rules).
2. *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.
3. *Validation*: in addition to the static patch validation, each generated candidate grammar is tested for fitness over the same initial test suite or even an evolving one to determine whether it improves over the parent grammar. We use a priority queue to keep improving the most promising candidate grammars.
4. *Control*: we alternate between localization, transformation, and validation as they reinforce one another and iterate until we find a fix.

We describe two refinements of our automatic repair approach and their realization in to the highly configurable gfixr tool: (*i*) *passive repair*: we described this variant in our SLE'21 paper (Raselimo and Fischer, 2021); it takes as input the faulty grammar G and repairs it against an input test suite that we keep constant throughout an entire repair process; and (*ii*) *active repair*: this variant has access to an explicit *membership oracle* and introduces a *test suite enrichment* where we judiciously generate new tests from each candidate grammar and use the oracleto confirm the outcome of these tests.

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") (DeMillo et al., 1978) 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.

Evaluation. We have successfully used gfixr to repair 33 student grammars that contain multiple, real faults. The passive repair variant finds full fixes in all but four cases, where it returns partially repaired grammar variants after 150 iterations. We show that even these partially repaired variants have improved in quality over their corresponding faulty input grammars. We also show that passive repair produces grammars that generalize well to new unseen tests that were generated from the target grammars (i.e., the fixed grammars improved the recall score). However, some of these repaired grammars are too permissive; hence they over-generalize beyond the target language. We use the precision computation to measure this over-generalization, where we calculate the proportion of tests generated from the output repair, in which the output grammar and the target grammar produce the same result.

We develop and evaluate active repair to address this over-generalization in passive repair. We show that the test suite enrichment introduced by active repair produces repairs that are less prone to over-generalization.Active repair achieves 100% precision in about half of the input grammars. It also produces high quality patches that capture the original intent of the grammar. It achieves a 100% F1 score in eight grammars, compared to none in the passive case.

Potential applications. An automatic grammar repair can 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 (Barraball et al., 2020) 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 (Di Penta et al., 2008); (*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 (Crepinsek et al., 2005; Di Penta et al., 2008).

Summary of contributions. In summary, this paper makes the following contributions enumerated below:

1. we present present the first approach to automatically repair faults in context-free grammars;
2. we describe two variants of the approach that both use test suites as specification for repair;
3. we realize this approach into a gfixr tool;
4. we demonstrate the effectiveness of our method over grammars written by students - these contain real and multiple faults; and
5. we compare the two repair variants of our method.

Relation to previous publications. This paper reports substantial extensions of the work described in our paper "Automatic Grammar Repair" (Raselimo and Fischer, 2021). That paper was presented at the 2021 ACM SIGPLAN International Conference on Software Language Engineering (SLE'21). The main extensions are the description and implementation of the active repair variant, an extended evaluation over a wider range of grammars with more faults, and a quantitative characterization of both passive and active repair in terms of precision, recall, and F1 scores.

2. Preliminaries

2.1. Context-Free Grammars

Grammar notation. A *context-free grammar* (CFG) or simply *grammar* 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 in N, $a, b, c, \ldots$ for terminals in T, X, Y, Z for *grammar symbols* in V, p, q, r for *productions* or *rules* in 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. In concrete examples, 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. We write $A \to \gamma$ for a rule $(A, \gamma) \in P$ and $P_A = \{A \to \gamma \in P\}$ 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 $A \to \gamma \in P$ and use $\Rightarrow^*$ for its reflexive-transitive closure. We write $\Rightarrow_R$ if $A \to \gamma \in R \subseteq P$. 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 $\varepsilon \in \text{yield}(\alpha)$. 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. $\text{follow}(X) = \{a \mid S \Rightarrow^* \alpha X a \beta\}$). Note that we extend the definitions of first and last so that they map to sets of symbols rather than terminals. We use $\text{first}_U(\alpha) = \text{first}(\alpha) \cap U$ to denote the restriction of first to a set $U \subseteq T$ of terminals, and similar for the other functions.

We call u a *viable k-prefix* of a word $w = uv$ if $|u| \leq k$ 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 (Raselimo et al., 2019). 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 (Lämmel, 2001b) coverage. For the running example, the *cdrc* test suite $TS_{\mathcal{T}oy}$ contains 79 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 (Abreu et al., 2006; Jones and Harrold, 2005; Naish et al., 2011; Renieris and Reiss, 2003; Wong et al., 2014) 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 (Jones and Harrold, 2005), Ochiai Ochiai (1957), or Jaccard (Chen et al., 2002)) 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 (Raselimo et al., 2019; Raselimo and Fischer, 2023). We localize faults at the level of individual symbols in rules (Raselimo and Fischer, 2023), 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.

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 $\sim_{\checkmark}, \sim_{\text{✗}} \subseteq P^\bullet \times TS_{\mathcal{L}}$ between items and tests that encode test execution and test outcome. We then define $\text{pass}(p^\bullet) = \{w \in TS_{\mathcal{L}} \mid p^\bullet \sim_{\checkmark} w\}$ and $\text{fail}(p^\bullet) = \{w \in TS_{\mathcal{L}} \mid p^\bullet \sim_{\text{✗}} w\}$ 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 $\text{score}(p^\bullet) > 0 \Rightarrow \text{fail}(p^\bullet) \neq \emptyset$. The usual formulas can be used based on the definitions of the counts given above.

We finally use the specialized *k-max tie breaking* mechanism (Raselimo and Fischer, 2023) 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.

Localization Example. Table 1 shows the counts aggregated from the grammar spectrum that we collected by running the CUP parser generated from the example grammar $G'_{\mathcal{T}oy}$ (see Figure 1) over the test suite $TS_{\mathcal{T}oy}$, 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 $A \to \alpha \bullet \omega$ from the n-th alternative production for A where $|\alpha| = m$. The Ochiai score of an item $p^\bullet$ is given by

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

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.

Table 1: Spectral counts, Ochiai scores and ranks for $G'_{\mathcal{T}oy}$ over $TS_{\mathcal{T}oy}$.

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

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 Algorithms 1 and 2 ; more implementation details are given in Section 7.

3.1. 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., $TS^+ \not\subseteq L(G)$ 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., $(TS^- \cap L(G)) \cup (TS^+ \setminus L(G)) \neq \emptyset$;

(*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 are not arbitrary token sequences but are wrongly accepted by the (current) grammar candidate and thus contain enough structure that can be exploited for repair attempts. In the remainder of the paper, we assume an initial 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., $TS^+ \subseteq L(G') \not\subseteq \mathcal{L}$. We can therefore evaluate the quality of our repairs only through manual inspection or based on performance over an additional *validation suite*.

3.2. 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 $G \rightsquigarrow_{\mathfrak{p}} G'$. 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*) $\forall w \in TS_{\mathcal{L}} \cdot \text{prefix}_G(w) \preceq \text{prefix}_{G'}(w)$

Hence, the patched grammar accepts more of the positive and fewer of the negative tests, and accepts longer input prefixes of the tests that it rejects. 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 $TS^- \cap L(G') = \emptyset$.

3.3. 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.

3.4. 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 $ua \preceq_k w$ 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 (Raselimo et al., 2019) and our repair attempts to break this poisoned pair property for $(a, b)_w$. 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). Examples of these will be shown in subsequent sections.

Algorithm 1: The passive repair algorithm

```
input  : A faulty grammar G = ⟨N, T, P, S⟩
input  : A test suite TS
output : A fully repaired variant G′ or ⊥
1  Q ← ∅
2  ⟨P, F, Pre⟩ ← run_tests(G, TS)
3  Q.enqueue(G, ⟨P, F, Pre, ∞⟩)
4  Seen ← {G}
5  repeat
6      ⟨G′, ⟨P_G′, F_G′, Pre_G′, _⟩⟩ ← Q.dequeue()
7      Seen.add(G′)
8      Ranks ← localize(G′, TS)
9      Cands ← transform(G′, Ranks)
10     for C ∈ Cands \ Seen do
11         ⟨F_C, Pre_C⟩ ← run_tests(C, TS)
12         if F_C = ∅ then
13             return C
14         if improves(⟨F_G′, Pre_G′⟩, ⟨F_C, Pre_C⟩) then
15             Q.enqueue(C, ⟨F_C, Pre_C, Ranks[C]⟩)
16 until Q.empty()
17 return ⊥
```

3.5. Patch Validation against Sample Bigrams

We can prevent some over generalization by providing negative tests, which can be seen as pre-emptive answers to some membership queries. In the active repair setting we can update this set automatically during the repair process, but in the passive setting we cannot. 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 can also occur as sample bigrams; if not, we reject the patch. Note that is of course a heuristic, it relies on the fact that TS^+ provides enough examples to approximate the follow-relation well enough through the bigrams. Note also that we can over-approximate the bigrams by simply assuming all pairs of tokens are possible (i.e., $\Gamma_2(TS_{\mathcal{L}}) = T \times T$); in this case, all patches are trivially validated, and we have to rely on the fitness function (see below) to rule out "bad" candidates.

3.6. Passive Repair

Algorithm 1 shows the passive repair loop that implements the find-and-fix cycle described earlier. It dequeues the top-ranked faulty grammar variant G' from a central priority queue Q that manages all repair candidates, 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 new candidate C resulting from applying a patch, it uses `run_tests` to generate an executable parser and run it over the test suite TS. If the candidate C fails no tests, it is returned as a full repair. Otherwise, if C `improves` on its parent G', it is enqueued.

Algorithm 2: The active repair algorithm

```
input  : A faulty grammar G = ⟨N, T, P, S⟩
input  : A test suite TS_L
input  : A boolean valued oracle O
output : A fully repaired variant G′ or ⊥
1  Q ← ∅
2  TS ← TS_L ∪ generate_tests(G)
3  Q.enqueue(G, _)
4  Seen ← {G}
5  repeat
6      ⟨G′, _⟩ ← Q.dequeue()
7      Seen.add(G′)
8      Ω ← run_oracle(TS)
9      Ranks ← localize(G′, Ω, TS)
10     Cands ← transform(G′, Ranks)
11     for C ∈ Cands \ Seen do
12         TS ← TS ∪ generate_tests(C)
13     ⟨F_G′, Pre_G′⟩ ← run_tests(G′, TS)
14     for C ∈ Cands \ Seen do
15         ⟨F_C, Pre_C⟩ ← run_tests(C, TS)
16         if F_C = ∅ then
17             return C
18         if improves(⟨F_G′, Pre_G′⟩, ⟨F_C, Pre_C⟩) then
19             Q.enqueue(C, ⟨F_C, Pre_C, Ranks[C]⟩)
20 until Q.empty()
21 return ⊥
```

The priority queue Q contains pending grammar candidates derived from improving patches. It is keyed by a three-tuple (F, Pre, R), where F is the number of failing tests, Pre is the total 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.

3.7. Active Repair

In the passive repair setting, we repair against an initial test suite as target specification, but keep this constant throughout the process; in particular, we also use this to localize (i.e., find repair sites) and validate new candidates as they are constructed.

If we have access to a membership oracle (e.g., a black-box parser) for the target language L, we can improve the localization and validation steps by generating new tests from grammar candidates as they are constructed, relying on the oracle to determine the true status of these unseen tests. Since this is similar

to Angluin's active learning (Angluin, 1987)[1], we call this the active repair.

Algorithm 2 presents a high-level description of this active repair approach. We extend and build on the passive repair algorithm shown in Algorithm 1. There are several differences in the flow of the search that we highlight. First, Algorithm 2 takes as input an extra variable, a boolean valued oracle O.

Second, in active repair we generate tests TS_C from each candidate C that we add to the growing pool of test cases TS, which already includes the user provided target test suite $TS_{\mathcal{L}}$ and the test suite TS_G generated from the input grammar G. We call the union of $TS_{\mathcal{L}}$ and TS_G test suites an initial test suite, and denote it by TS_{init}. Each candidate C is tested for fitness over the growing pool TS and enqueued if it improves over its parent. Note that TS is global and monotonously growing. Candidates therefore get tested for fitness against a test suite that includes tests not derived from themselves, thus improving the fitness selection. The algorithm returns as a fix a candidate that produces test outcomes that are consistent with the language accepted by the oracle O, i.e., passes all tests in TS and thus TS_{init}.

4. Symbol Editing Patches

Our first group of patches is modelled on the basic string editing operations, (i.e., deletion, insertion, substitution, and transposition), 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 $TS'_{\mathcal{T}oy} \supseteq TS_{\mathcal{T}oy}$ (see Appendix A for a complete test suite) 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 $G_{\mathcal{T}oy}$ because neither *paramlist* nor *arglist* are nullable, and their addition to $TS_{\mathcal{T}oy}$ can be seen as a "change request" to $G_{\mathcal{T}oy}$ to allow empty formal parameter and argument lists.

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

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

In these 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 (*paramlist* resp. *arglist*) are $\{\texttt{)}\}$ as well.

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

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

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 $\mathit{name} \Rightarrow_{G'} \texttt{id (}\ \mathit{namelist}\ \texttt{)} \Rightarrow^*_{G'} \texttt{id (, id)}$. This overgeneralization could be prevented by additional explicit counter-examples, but we instead rely on a careful formalization of the *synchronization patch* and corresponding *patch validation*.

Definition 4.1 (synchronization). *Let $p = A \rightarrow \alpha \bullet \beta\omega$ be an item in $P^\bullet$ with $\mathit{left}(p) \subseteq T^+_p$.*

(a) If $\omega = X\gamma$ with X non-nullable and $T^-_p \subseteq \mathit{first}(X)$, 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 $T^-_p \subseteq \mathit{follow}(A)$, let $\mathfrak{d}(p,\beta\omega) = A \rightarrow \alpha\bullet$ 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' around the repair site against the bigrams.

Definition 4.2 (synchronization validation). *Let $p = A \rightarrow \alpha\bullet \beta\omega$ be an item in $P^\bullet$. The synchronization patch $G \rightsquigarrow_{\mathfrak{d}(p,\beta)} G'$ is* validated *over $TS_{\mathcal{L}}$ if $\mathit{left}_{G'}(\mathfrak{d}(p,\beta)) \times \mathit{right}_{G'}(\mathfrak{d}(p,\beta)) \subseteq \Gamma_2(TS_{\mathcal{L}})$.*

In the running example, the deletions of *paramlist* and *arglist* both only expose the single "repair bigram" (`(` , `)`), which occurs in $\Gamma_2(TS'_{\mathcal{T}oy})$.

Example Repair. gfixr patches the baseline grammar $G_{\mathcal{T}oy}$ against $TS'_{\mathcal{T}oy}$ as expected, by adding the three rules

$$\begin{aligned} &\mathit{fdecl} \rightarrow \texttt{define id ( )}\ \mathit{body} \\ &\mathit{fdecl} \rightarrow \texttt{define id ( ) ->}\ \mathit{type}\ \texttt{id}\ \mathit{body} \\ &\mathit{name} \rightarrow \ldots \mid \texttt{id ( )} \end{aligned}$$

The rules are created from the corresponding baseline rules by the deletion of a single symbol at the identified fault locations

[1] Note that Angluin also requires an equivalence oracle to decide termination of the learning process; we still use the initial test suite for this purpose.

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.[2] 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 $\Gamma_2(TS'_{\mathcal{T}oy})$ does not contain the induced bigram (`(` , `,`).

In the variant $G'_{\mathcal{T}oy}$ with a nullable *namelist*-rule, 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 $G_{\mathcal{T}oy}$ (see Figure 1) 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 in $G_{\mathcal{T}oy}$). Note that 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., $T_p^- \cap \text{right}(A \to \alpha \bullet \omega) \neq \emptyset$), i.e., we only require the patch to "eat up" at least one bad token, to allow a patch to (partially) repair a subset of failing tests at a time.

Definition 4.3 (symbol insertion). *Let $p = A \to \alpha \bullet \omega$ be an item in $P^\bullet$ with $left(p) \subseteq T_p^+$, 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 $T_p^- \cap right(\mathfrak{i}(p, X)) \neq \emptyset$, 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 4.4 (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 $left_{G'}(\mathfrak{i}(p, X)) \times right_{G'}(\mathfrak{i}(p, X)) \subseteq \Gamma_2(TS_{\mathcal{L}})$.*

[2] 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. The evaluation in Section 8 uses a different computational setup, and times are not necessarily comparable.

Example Repair. If we remove the rule

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

from $G_{\mathcal{T}oy}$, 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*). A substitution patch replaces the symbol at the designated position by another one that "eats up" at least one of the bad tokens.

Definition 4.5 (symbol substitution). *Let $p = A \to \alpha \bullet X\omega$ be an item in $P^\bullet$ with $left(p) \subseteq T_p^+$, $Y \in V$, and $\mathfrak{s}(p, Y) = A \to \alpha \bullet Y\omega$ be the result of replacing X at the designated position by Y. If $T_p^- \cap right(A \to \alpha \bullet Y\omega) \neq \emptyset$, 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 4.6 (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*

(i) $left_{G'}(\mathfrak{s}(p, Y)) \times right_{G'}(\mathfrak{s}(p, Y)) \subseteq \Gamma_2(TS_{\mathcal{L}})$, and

(ii) $left_{G'}(A \to \alpha Y \bullet \omega) \times right_{G'}(A \to \alpha Y \bullet \omega) \subseteq \Gamma_2(TS_{\mathcal{L}})$.

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 $A \to \alpha \bullet X\omega \rightsquigarrow A \to \alpha YZ \bullet \omega$ is realized via $A \to \alpha Y \bullet X\omega$ and not via $A \to \alpha Y \bullet \omega$, which reduces the search space.

Example Repair. Substitutions, deletions, and insertions can interact to create larger repairs. Consider for example a student implementation of the language of $G_{\mathcal{T}oy}$ where the rule: *factor* → `(` *expr* `)` is missing, so that it rejects bracketed expressions. The following five tests fail from $TS_{\mathcal{T}oy}$

```
program a begin write 0 * (0) end
program a begin write not(0) end
program a begin a(0, (0)) end
program a begin write(0) end
program a begin a((0)) end
```

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. Note that this already constitutes a fix, because it makes all tests pass, even though it does not fit the intent (which would have also replaced the *factor* at the right-hand side of the rule by *expr*).

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 very common bug pattern, it does occur in connection with list rules. For example, consider the following variant of $G_{\mathcal{T}oy}$ that has 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 the `id` and `,` tokens 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 4.7 (symbol transposition). *Let $p = A \to \alpha \bullet XY\omega$ be an item in $P^\bullet$ with $left(p) \subseteq T_p^+$, and $\mathfrak{t}(p) = A \to \alpha \bullet YX\omega$ be the result of swapping the symbols X and Y at the designated position. If $T_p^- \cap right(\mathfrak{t}(p)) \neq \emptyset$, 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 $A \to \alpha \bullet YX\omega$, $A \to \alpha Y \bullet X\omega$, and $A \to \alpha YX \bullet \omega$.

Definition 4.8 (transposition validation). *Let $p = A \to \alpha \bullet XY\omega$ be an item in $P^\bullet$. The transposition patch $p \rightsquigarrow \mathfrak{t}(p)$ is* validated *over $TS_{\mathcal{L}}$ if*

(i) $left_{G'}(\mathfrak{t}(p)) \times right_{G'}(\mathfrak{t}(p)) \subseteq \Gamma_2(TS_{\mathcal{L}})$, *and*

(ii) $left_{G'}(A \to \alpha Y \bullet X\omega) \times right_{G'}(A \to \alpha Y \bullet X\omega) \subseteq \Gamma_2(TS_{\mathcal{L}})$.

(iii) $left_{G'}(A \to \alpha YX \bullet \omega) \times right_{G'}(A \to \alpha YX \bullet \omega) \subseteq \Gamma_2(TS_{\mathcal{L}})$.

5. Listification Patches

Our second group of patches is geared towards more structural changes in the grammar. In this section, we introduce two "listification" patches, *right recursion introduction* and its generalization, *list synthesis*.

5.1. Right Recursion Introduction

Right recursion introduction patches are useful to handle bugs where the grammar fails to properly handle repetitions. Consider for example a variant of $G_{\mathcal{T}oy}$ submitted by a student where the the *body*-, *vdecllist*-, and *vdecl*-rules in $G_{\mathcal{T}oy}$ 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. The obvious fix is to restore the intent behind the *vdecls*-rule by making it right-recursive.

Definition 5.1 (right recursion introduction). *Let $G = (N, T, P, S)$, $G' = (N, T, P', S)$ be CFGs, and $p = A \to \alpha\bullet \in P^\bullet$ a reduction item with $first(A) \subseteq T^-$.*

(a) If A is nullable and $A \to \varepsilon \in P$, let $P' = P \setminus \{p\} \cup \{A \to \alpha A\}$.

(b) If A is nullable and $A \to \varepsilon \notin P$, let $P' = P \setminus \{p\} \cup \{A \to \alpha A, A \to \varepsilon\}$.

(c) If A is not nullable, let $P' = P \cup \{A \to \alpha A\}$.

Then $G \rightsquigarrow_{\mathfrak{L}_1(p)} G'$ is a right recursion introduction patch.

Right recursion introduction 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 this listification patch can be seen as a 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 5.2 (listification validation). *Let $p = A \to \alpha\bullet$ be an item in $P^\bullet$. The listification patch $G \rightsquigarrow_{\mathfrak{L}(p)} G'$ is* validated *over $TS_{\mathcal{L}}$ if $left_{G'}(A \to \alpha \bullet A) \times right_{G'}(A \to \alpha \bullet A) \subseteq \Gamma_2(TS_{\mathcal{L}})$.*

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

5.2. List Synthesis

Right recursion introduction does not generalize to all repetitions in a grammar. We identify two scenarios where it falls short and cannot be applied as a patch. First, when a repetitive structure is used in a local context and occurs in the middle of the definition of some A-production. Assume, for example, that the *body*-rule in $G_{\mathcal{T}oy}$ is replaced by the following rules:

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

This allows at most one variable declaration captured via the sequence *type* `id` `;` in the middle of the second alternative of the *body*-rule. This variant of $G_{\mathcal{T}oy}$ thus fails one of the tests,

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

with the •-symbol also indicating the error location observed in the input. Right recursion introduction cannot fix this because the repetition needs to be spliced into the middle of the second *body*-rule, but there is no non-terminal symbol that "summarizes" the elements to be repeated.

Second, right recursion introduction cannot handle delimiter-separated repetitions. These list structures are omnipresent in most languages: think of a comma-separated list of function parameters in popular programming languages or a semicolon-separated list of SQL queries and many more others. The list synthesis patch handles such cases.

We can further extend the machinery and extract the next token at the right-hand side of the bad token before parsing stops due to a syntax error. We call this token the *right token*. We use T_p^* for a set of right tokens for an item p executed in failing tests. We only use these tokens for patches that synthesize list structures.

Definition 5.3 (list synthesis). *Let $G = (N, T, P, S)$ be a CFG and $p = A \to \alpha\gamma \bullet \omega$ an item in $P^\bullet$. If $G' = (N', T, P', S)$ is a CFG with $N' = N \cup \{B\}$, $B \notin N$, and $P' = P \setminus p \cup \{A \to \alpha\gamma B\omega, B \to \beta\gamma B, B \to \varepsilon\}$ then $G \rightsquigarrow_{\mathfrak{L}_2(p)} G'$ is a* list synthesis patch*, provided:*

(a) β is nullable, $T_p^+ \subseteq left(A \to \alpha\gamma \bullet \omega)$ and $T_p^- \cap right(A \to \alpha\gamma \bullet B\omega) \neq \emptyset$.

(b) β is not nullable, $T_p^+ \subseteq left(A \to \alpha\gamma \bullet \omega)$, $T_p^- \cap right(B \to \bullet\beta\gamma B) \neq \emptyset$, and $T_p^* \cap right(B \to \beta \bullet \gamma B) \neq \emptyset$.

The first scenario that we sketched out in the motivation for the need for list synthesis patch, is handled by case (a) in the above definition, where the separator symbol β is empty. Case (b) synthesizes β-separated list elements. In practice, however, this transformation needed extra control flags that restrict its applicability. For example, in the example repair in Section 4.2 where the fix required multiple iterations, list synthesis gets better fitness in the first iteration because it consumes more of the input than the partial insertion patch that ultimately leads to the full fix. We, therefore, only accept list synthesis patches if their application leads to fewer test failures than the parent faulty variant.

Example repair. If we modify the *name*-rule for function call expressions from $G_{\mathcal{T}_{Toy}}$

$$name \to \ldots \mid \texttt{id}\ \texttt{(}\ expr \bullet\ \texttt{)}\ \mid \ldots$$

the following four tests fail

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

and the sets of good, bad, and right tokens are $T^+ = \{$ `)` , `a` , `0` $\}$, $T^- = \{$ `,` $\}$ and $T^* = \{$ `(` , `a` , `0` $\}$ respectively.

gfixr reconstructs it correctly to the following rules.

$$\begin{aligned} name &\to \ldots \mid \texttt{id}\ \texttt{(}\ expr\ expr\_list\ \texttt{)}\ \mid \ldots \\ expr\_list &\to\ \texttt{,}\ expr\ expr\_list \mid \varepsilon \end{aligned}$$

Note that the actual patch contains a randomly generated identifier for the new non-terminal introduction, but we use *expr_list* here for clarity. gfixr generated 25 candidate grammars and took 1 minute and 40 seconds to find the patch.

6. Language Tightening Transformations

The example grammar in Figure 1 contains three different quirks which allow procedure calls without argument lists, call expressions as *lvalues*, and indexing expressions as statements. It does not distinguish properly between simple identifiers, array indexing expressions, and function calls, and instead subsumes all three under the non-terminal *name*:

$$\begin{aligned} assign &\to name \mid name\ \texttt{::=}\ expr \mid name\ \texttt{::=}\ \textbf{array}\ simple \\ input &\to \textbf{read}\ name \\ factor &\to name \mid \ldots \\ name &\to \texttt{id} \mid \texttt{id}\ \texttt{[}\ simple\ \texttt{]} \mid \texttt{id}\ \texttt{(}\ expr\ exprlist\ \texttt{)} \end{aligned}$$

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
```

These idiosyncrasies should (and can) already be filtered out by syntactic analysis. The common cause of these and similar issues is that the grammar is too permissive, i.e., $\mathcal{L} \subseteq L(G)$. A repair of this permissiveness requires a language restriction or *tightening*, which can be specified by negative tests. We focus here on false positives or *counter-examples* because arbitrary negative tests do not provide enough structure to guide the repair. In the following, we look at specific tightening patches, rule deletion and non-terminal splitting or "downcasting", de-listification patches, and a patch that tightens list structures by pushing down some list elements. Note that we apply the language tightening patches only at reduction items.

6.1. Rule Deletion

Clearly, deleting a rule tightens the language; the only non-trivial 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 6.1 (rule deletion). *Let $G = (N, T, P, S)$ with $TS^+ \subseteq L(G)$, $p = A \to \alpha\bullet \in P^\bullet$ 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 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 $\text{fail}(p) \subseteq TS^-$), 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 patch 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* → **`read`** *name*, which only fails in combination with *name* → `id (` *arglist* `)`.

We therefore need an enabling patch that moves rules into the right contexts (similar in spirit to *cdrc* coverage (Lämmel, 2001b)) and so separates out passing and failing rule applications.

Definition 6.2 (non-terminal splitting). *Let $G = (N, T, P, S)$, $p = A \to \alpha B \omega\bullet \in P^\bullet$ a reduction item with $P_B = \{B \to \beta_i\}_{i>1}$, $ep(p) > 0$, $ef(p) > 0$, and $\text{fail}(p) \subseteq TS^-$. If $G' = (N, T, P', S)$ is a CFG with $P' = P \setminus \{p\} \cup \{A \to \alpha\beta_i\omega\}$ then $G \rightsquigarrow_{\Xi(p,B)} G'$ is a* non-terminal splitting patch *for B.*

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

Example Repair. We repaired the idiosyncrasies in $G_{\mathcal{T}oy}$ with a $step_2$ (van Heerden et al., 2020) 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:

stmt → `id (` *arglist* `)`
| `id ::=` *expr* | `id [` *expr* `] ::=` *expr* | `id ::= array` *simple*
| *cond* | ...
input → **`read`** `id` | **`read`** `id [` *expr* `]`

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 *input* rules) 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*.

6.3. Token Splitting

The need for token splitting occurs when multiple alternative lexemes that belong to different contexts are subsumed under the same structured token, (e.g., a grammar with an `ADDOP`-token that captures the lexemes "`+`" and "`-`", but without a proper (unary) `MINUS`-token). Due to the considerable freedom the CFG formalism allows grammar developers, such faults are prevalent and are difficult to spot, especially, under assumptions that a stable lexer-parser interaction is made available, and the focus is purely on the context-free syntax. For example, the two Pascal grammars that were proved non-equivalent by Madhavan et al. (2015) have different terminal sets, one of the grammars defines specific terminal symbols for the basic types such as **`BOOLEAN`** while the other subsumes them under identifiers. We extend and build on the non-terminal splitting transformation introduced above to implement token splitting.

Definition 6.3 (token splitting). *Let $G = (N, T, P, S)$, $p = A \to \alpha a \omega\bullet \in P^\bullet$ a reduction item and $\text{fail}(p) \subseteq TS^-$. Let $RE = \{T \cup \{S_L\}, \Sigma, P_L \cup \{S_L \to t \mid t \in T\}, S_L\}$ be a lexical grammar that captures structured tokens, $a \to b_i$ with $b_i \in (\Sigma \cup T)^*$. If $G' = (N, T', P', S)$ is a CFG with $T' = T \setminus \{a\}$ and $P' = P \setminus \{p\} \cup \{A \to \alpha b_i \omega\}$ then $G \rightsquigarrow_{\mathfrak{T}(p,a)} G'$ is the* token splitting patch *for the structured token a.*

Note that gfixr's current implementation of the token splitting transformation ignores some of the common lexer policies, such as longest match and rule ordering. Their integration is not straightforward, and we leave its investigation for future work.

Example repair. Consider, for example, a student's implementation showing the rules for *simple*, *termlist*, and a structured token `ADDOP`

simple → `ADDOP` *termlist* • | *termlist*
termlist → *term* | *termlist* `ADDOP` *term*
`ADDOP` → **`-`** | **`+`** | **`or`**

The lexer returns the same token `ADDOP` for lexemes **`+`**, **`-`** and **`or`**. The parser fails six negative tests, including the following,

```
program a begin a := array or 0 end
program a begin write or 0 or 0 end
program a begin write 0 = or 0 end
```

because it wrongly accepts **`or`** as a prefix-operator.

gfixr finds the fix the fault in two iterations in under ten minutes, generating 225 candidate grammars. In the first iteration, gfixr splits all occurrences of the `ADDOP`-token in the *simple*- and *termlist*-rules into the three lexemes. This gives us

simple → **`-`** *termlist* | **`+`** *termlist* | **`or`** *termlist* | *termlist*
termlist → *term* | *termlist* **`-`** *term* | *termlist* **`+`** *term* | *termlist* **`or`** *term*

In the second iteration, a rule deletion patch is applied to rules *simple* → `+` *termlist* and *simple* → `or` *termlist*, which leaves us with a full fix:

simple → `-` *termlist* | *termlist*
termlist → *term* | *termlist* `-` *term* | *termlist* `+` *term* | *termlist* `or` *term*

6.4. Recursion Elimination

Recursion elimination is another language tightening transformation like rule deletion and splitting transformations. Unlike other tightening transformations, it specifically restricts overly permissive repetitions in a grammar. It can be seen as an inverse of the listification transformations introduced in Section 5. To illustrate grammar bugs that this transformation targets, consider the following *expr*-rules in $G_{\mathcal{T}oy}$

$$expr \rightarrow simple\ relop\ simple \mid simple$$

Also consider $G''_{\mathcal{T}oy}$ submitted by a student where the *expr*-rules in $G_{\mathcal{T}oy}$ are replaced by the following rules:

$$expr \rightarrow expr\ relop\ simple \mid simple$$

The target language restricts relational operators (*relop*) to only two simple terms as arguments, but the first alternative *expr*-rule in $G''_{\mathcal{T}oy}$ over-generalizes this and allows an arbitrary number of simple terms and therefore wrongly accepts the following tests (among others):

```
program a begin if 0 == 0 == 0 then relax end end
program a begin while a == a >= a do relax end end
```

Due to the global nature of the notion of poisoned pairs used in the mutation-based negative test suites construction algorithms (Raselimo et al., 2019), above test cases cannot be generated, hence, the fault in the *expr*-rules in $G''_{\mathcal{T}oy}$ remains undetected. These mutation-based algorithms do not find poisoned pairs around the (substitution) mutation location and invalidates the following mutation *expr* → *expr relop simple* from which the fault in $G''_{\mathcal{T}oy}$ can be detected.

Definition 6.4 (immediate recursion elimination). *Let* $G = (N, T, P, S)$*,* $p = A \rightarrow \alpha A \omega \bullet$ *a reduction item with other alternatives for A,* $P_A = \{A \rightarrow \beta_i\}_{i>1}$*,* $ep(p) > 0$*,* $ef(p) > 0$*, and* $fail(p) \subseteq TS^{-}$*. If* $G' = (N', T, P', S)$ *is a CFG with* $N' = N \cup \{B\}$*,* $B \notin N$*, and* $P' = P \setminus \{p\} \cup \{A \rightarrow \alpha B \omega, B \rightarrow \beta_i\}$ *then* $G \rightsquigarrow_{\mathfrak{E}(p,A)} G'$ *is the* immediate recursion elimination patch *for A.*

Example repair. gfixr prevents the over-generalization in the *expr*-rule for $G''_{\mathcal{T}oy}$ by transforming the rules as follows:

expr → *rest relop simple* | *rest*
rest → *simple*

It generates 86 candidate grammars in 5 minutes in a single iteration.

6.5. Push-down List Elements

Another widespread occurrence of over-approximation common to most student's implementations is the permissive definition of list elements. Consider, for example, a faulty implementation of a function call with the following rules.

name → ... | `id (` *expr_list* `)` | ...
expr_list → *expr_list* `,` *expr* | *expr* | ε

The above rules capture all syntactically valid function calls but allows function call arguments to be preceded by a comma, e.g.,

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

It is perhaps worth noting that, although straightforward, the above counter-examples may not be generated due to some restrictions (e.g., deletion and insertion of nullable symbols) by the rule mutation algorithm, depending on how the golden grammar that describes the target language is formulated. This shows that these type of faults can be difficult to spot.

Definition 6.5 and Definition 6.6 formulate the left and right recursive variations, respectively.

Definition 6.5 (push-down list elements). *Let* $G = (N, T, P, S)$*,* $p = A \rightarrow A\gamma\upsilon \bullet \in P^{\bullet}$ *a reduction item with other alternatives for A,* $P_A = \{A \rightarrow \upsilon, A \rightarrow \omega_i\}_{i>1}$*,* $ep(p) > 0$*,* $ef(p) > 0$*, and* $fail(p) \subseteq TS^{-}$*. If* $G' = (N', T, P', S)$ *is a CFG with* $N' = N \cup \{B\}$*,* $B \notin N$*, and* $P' = P \setminus \{p\} \cup \{A \rightarrow B, A \rightarrow \omega_i, B \rightarrow B\gamma\upsilon, B \rightarrow \upsilon\}$ *then* $G \rightsquigarrow_{\mathfrak{P}_l(p,A)} G'$ *is the* push-down list elements patch *for A.*

Definition 6.6 (push-down list elements). *Let* $G = (N, T, P, S)$*,* $p = A \rightarrow \alpha\beta A \bullet \in P^{\bullet}$ *a reduction item with other alternatives for A,* $P_A = \{A \rightarrow \alpha, A \rightarrow \gamma_i\}_{i>1}$*,* $ep(p) > 0$*,* $ef(p) > 0$*, and* $fail(p) \subseteq TS^{-}$*. If* $G' = (N', T, P', S)$ *is a CFG with* $N' = N \cup \{B\}$*,* $B \notin N$*, and* $P' = P \setminus \{p\} \cup \{A \rightarrow B, A \rightarrow \gamma_i, B \rightarrow \alpha\beta B, B \rightarrow \alpha\}$ *then* $G \rightsquigarrow_{\mathfrak{P}_r(p,A)} G'$ *is the* push-down list elements patch *for A.*

The combination of non-terminal splitting and rule deletion patches can also be used here to achieve the required result. However, these patches may require extra iterations if there are multiple occurrences of non-terminal *A*.

Example repair. gfixr finds the fix for the above over-generalization in about 4 minutes and generated 106 candidate patches.

expr_list → *exprs* | ε
exprs → *exprs* `,` *expr* | *expr*

Here, we also use a self-explanatory name for the new non-terminal that is introduced. The actual patch contains a randomized name.

7. Implementation

We have prototyped both passive and active repair variants, described in the previous section, in a gfixr tool.

System Architecture. gfixr implements the repair loop variations shown in Algorithms 1 and 2. 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 5.5kLoC.

gfixr currently only repairs 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 enough in our experiments, but this can be reconfigured easily.

The input oracle O used in the active case can be in the form of a black-parser that can confirm membership. In our experiments, we use parsers from ANTLR for ground-truth grammars describing the respective target languages.

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 Section 4-6 at each repair site to produce candidate patches. The order in which the applicable patches are tried is implementation-dependent and mostly fixed; however, users can control which symbols are used for insertion and substitution patches (see below for details). Patch selection is therefore integrated into the `transform` module.

In the passive repair variant, gfixr evaluates the performance of each candidate patch over the original input test suite; in the active repair case, it uses an ever-growing test suite that is updated after each iteration with the test cases generated from the iteration's new candidates. 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 that 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, an optional oracle which switches it to the active repair mode, and the test suite used to specify the repair. The option `-bigrams_file` specifies the separately created file containing the bigrams used for patch validation. `-oracle` provides the black-box parser that implements the ground-truth grammar describing the unknown target language.

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 4.3). Both settings enable more transformations but may lead to overgeneralization.

We also introduce more control flags. `-strict` is an option that prevents greedy transformations like list synthesis from overmatching and thus over-generalizing beyond the target language. Some languages can be inherently ambiguous by design and not parsable using the LALR algorithms; we therefore introduce the option `-non_lr` to discard positive test cases generated from a grammar variant G', and not in $L(G')$ because of some conflict in the grammar.

Further options control CUP's parsing algorithm. `-rr` sets the number of reduce/reduce conflicts that are allowed in the candidate; the 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. Evaluation

Our experimental evaluation addresses the following three research questions:

RQ1. How effective is our proposed passive repair approach in fixing faults in grammars?

RQ2. How effective is our proposed active repair approach in fixing faults in grammars?

RQ3. Does the active repair approach induce better fixes than the passive repair approach?

8.1. Experimental Setup

Evaluation Subjects. In our experiments, we used CUP grammars written by students to evaluate gfixr's efficacy. These grammars describe different but structurally similar medium-size Pascal-style languages used in different graduate compiler engineering courses. Many of the submissions have lexical issues and could not handle the interactions between parser and lexer properly. We discarded submissions with known lexical issues (e.g., wrong regular expressions for strings). The first ten grammars (#1 to #10, see Table 2) were taken from different small cohorts; they were randomly selected from all submissions that failed at least one test. The remainder of the grammars (#11 to #33) are from the most recent cohort, where the class size was significantly larger, with a total enrolment of 28. In one assignment, the students were tasked with writing CUP parsers

for two languages $\mathcal{G}$ and $\mathcal{H}$. The grammar for the language $\mathcal{G}$ was straightforward, since students were given its description in a different formalism and only had to adapt it for CUP parsing. For the second language, however, they were given a textual description of the language that they had to formalize into a CUP grammar. We discarded four submissions that contain reduce/reduce conflicts, as well as the grammars that produced parsers that pass all tests. This leaves us with a total of 23 grammars for both languages that we repair. Note that these grammars are free of semantic actions; we leave handling of grammars with semantic predicates for future work.

Test Suites. For each target language we generated two test suites from the 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. In the active repair case, we also generate the *cdrc* test suite from each generated candidate patch that we then add to the initial test suite, as described in Section 3.7.

Evaluation Metrics. To determine how well the gfixr-repaired grammars generalize, and to enable a fair comparison between the passive and active repair configurations, we adopt the evaluation approach used to evaluate the accuracy of the learned grammars in the Arvada system (Kulkarni et al., 2021). This relies on validation test suites that are generated from the target (resp. repaired) grammar to measure recall (resp. precision). Unlike in Arvada, however, our validation suite includes negative test cases. We generate larger and more diverse bfs_k, *deriv*, and random test suites. We also use negative test suites generated via the rule mutation algorithm as validation tests. We randomly sample 1000 test cases of which a third are negative tests. The precision, recall, and F1 scores shown in Tables 3 and 4 are average runs over five samples of 1000 tests each.

Recall: We use recall to determine how well each gfixr-repaired grammar variant generalizes to new, unseen tests. Here, we generate the validation suite from the oracle grammar, and we measure in how many of the tests in the validation suite the repaired variant is consistent with the golden grammar (i.e., the generated parser reports the expected result).

Precision: We use precision to determine how closely each gfixr-repaired grammar variant approximates the repair target. Here, we generate the validation suite from the repaired grammar variant. We measure the proportion of tests sampled from this validation suite where the repaired variant and the oracle grammar are consistent.

F1 Score: We use the F1 score as combined measure of how accurate the repaired grammar is. It is the harmonic mean of precision and recall.

Note that in the cases where the validation test suite contains only positive tests, low recall indicates overfitting (i.e., the repair target is overly specialized towards the input test suite specification) and low precision indicates over-generalization, i.e., the repaired grammar is too permissive. However, since our validation tests include negative tests, the terms over-fitting and over-generalizations are not as intuitive as they are when using only positive tests. For example, when a repaired grammar variant accepts a negative test generated from the target grammar, it indicates that the repaired grammar is too permissive and according to our set up above, we get low recall.

8.2. Passive Repair Results (RQ1)

Table 2 summarizes the results of our passive repair approach for the student grammars. $\mathcal{L}$ is the target language, with $\mathcal{T}oy$ the running example (see Figure 1), and $\mathcal{A}$ to $\mathcal{H}$ the languages from the different assignments. *bugs* is the number of faults in the student grammars revealed by the input test suite $TS_{\mathcal{L}}$. This was determined by manually inspecting the rules identified as suspicious by a spectrum-based fault localization metric (i.e., Ochiai) using $TS_{\mathcal{L}}$ for the localization and confirmed in the successfully repaired grammar variant.[3] $|TS_{\mathcal{L}}|$ is the number of positive tests in the repair test suite, with *fails* the number of failing tests. *iter* is the number of iterations of the repair loop. We limit the total number of iterations to 150; when this is reached, the repair algorithm stops the search and returns the best candidate as partial repair. Partial repair entries are shaded grey in Tables 2 and 3. *cand* is the number of candidate grammars generated by the repair algorithm. *time* is the overall runtime of the repair; measured as wall-clock time on an otherwise idle 2.70 GHz server with 36 cores (i.e., 72 hyper-threads) and 378 GB RAM and given as hours:minutes:seconds. The times include the compilation of the candidate grammars for CUP (and their corresponding lexical specifications in JFlex format) to Java and further to executable code, the execution of this code over the test suite, the fault localization, the computation of the grammar predicates for each selected candidate, the application of the actual repair transformations, and the output of the new candidates in CUP format. The timings are dominated by the first of these steps: the compilation of the CUP grammars takes on average about five seconds.

Efficacy. Table 2 shows overall promising results, and we can observe a few trends. First, and foremost, gfixr can indeed fix grammar bugs: our passive repair configuration returns a patch that is consistent with the repair specification given by the test suite $TS_{\mathcal{L}}$, in all but four grammars (#2, #10, #25, and #33) where it failed to find the repairs within 150 iterations. This indicates that the localization directs the repair to the right locations, despite the fact that the technique it uses is based on single fault assumption and some studies have shown that multiple fault interactions may harm their effectiveness (Abreu et al., 2009; Xue and Namin, 2013). Moreover, it also indicates that the combined patches are sufficiently expressive. In the failing cases, however, the localization ranked the faulty location too

[3] Note that we could not manually find and fix all bugs for some grammars; this is indicated by the entry >5. In these cases, gfixr was also unable to find a full fix.

Table 2: Passive repair results for student grammars. Partial repairs are shaded grey.

		grammar				tests		gfixr		
#	$\mathcal{L}$	$\|N\|$	$\|T\|$	$\|P\|$	bugs	$\|TS_{\mathcal{L}}\|$	fails	iter.	cand.	time
1	$\mathcal{Toy}$	36	32	68	2	86	12	2	43	00:01:29
2	$\mathcal{A}$	46	42	102	1	179	3	150	10744	04:23:53
3	$\mathcal{A}$	49	43	107	1	179	2	1	94	00:02:59
4	$\mathcal{B}$	45	42	88	2	79	2	2	55	00:01:59
5	$\mathcal{C}$	35	27	60	1	86	1	1	2	00:00:30
6	$\mathcal{D}$	45	30	78	1	80	14	1	89	00:02:15
7	$\mathcal{E}$	46	24	79	4	199	14	20	332	00:15:59
8	$\mathcal{E}$	47	32	84	4	199	17	11	576	00:15:25
9	$\mathcal{F}$	39	46	96	2	212	18	5	513	00:39:09
10	$\mathcal{F}$	49	72	145	> 5	212	58	150	36924	15:19:03
11	$\mathcal{G}$	32	49	94	2	194	17	2	398	00:10:35
12	$\mathcal{G}$	32	49	80	-	194	-	-	-	-
13	$\mathcal{G}$	43	49	92	2	194	11	3	188	00:05:54
14	$\mathcal{G}$	53	49	98	9	194	181	9	2412	01:01:30
15	$\mathcal{G}$	31	49	75	1	194	5	3	198	00:05:44
16	$\mathcal{G}$	37	49	84	1	194	3	1	201	00:04:40
17	$\mathcal{G}$	37	49	83	1	194	5	1	22	00:05:51
18	$\mathcal{G}$	38	49	99	4	194	17	7	178	00:08:17
19	$\mathcal{G}$	35	48	87	2	194	10	5	309	00:07:39
20	$\mathcal{H}$	52	62	124	-	205	-	-	-	-
21	$\mathcal{H}$	42	62	110	2	205	46	4	215	00:09:07
22	$\mathcal{H}$	46	62	120	4	205	56	8	3775	01:39:11
23	$\mathcal{H}$	44	62	106	2	205	2	2	38	00:01:49
24	$\mathcal{H}$	54	62	121	4	205	13	4	134	00:07:31
25	$\mathcal{H}$	39	62	102	> 5	205	38	150	11838	08:40:34
26	$\mathcal{H}$	48	62	121	-	205	-	-	-	-
27	$\mathcal{H}$	56	62	139	2	205	12	2	249	00:07:42
28	$\mathcal{H}$	47	59	103	1	205	5	1	89	00:02:29
29	$\mathcal{H}$	61	62	116	1	205	44	1	102	00:06:28
30	$\mathcal{H}$	57	62	116	-	205	-	-	-	-
31	$\mathcal{H}$	49	62	119	2	205	25	2	543	00:14:17
32	$\mathcal{H}$	41	62	110	1	205	1	1	238	00:08:54
33	$\mathcal{H}$	35	62	98	> 5	205	205	150	8704	08:02:41

low, and the repair kept trying to fix correct rules (see Section 9 for a more detailed discussion).

Second, the wall-clock repair times are typically below or around 15 minutes using a moderately powerful server, in particular if the grammar contains only a few (up to four) faults. Grammars with multiple faults that require several patches obviously take longer, but gfixr can still find fixes comprising patches and in most cases in less than 60 minutes wall-clock time.[4] The overall runtime is approximately linear with the number of candidate grammars.

Third, in about half of the cases, the number of iterations of the repair loop is the same as the number of bugs, 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, note that our input test suite $TS_{\mathcal{L}}$ (which satisfies *cdrc* coverage) cannot reveal bugs in four grammars (#12, #20, #26, and #30) but the active repair approach demonstrates that these grammars indeed contain faults.

Accuracy Evaluation. Table 3 shows the accuracy evaluation of our passive repair approach. Columns R_o, P_o, and $F1_o$ contain recall, precision, and F1 score values for the faulty original input grammar, respectively. We include these values in order to investigate whether the repair does indeed produce grammars with better quality. From these values, we see that grammars with fewer than five bugs already have moderately high recall scores, which validates our assumption of the competent programmer hypothesis. However, low precision scores mean that most the input grammars over-generalize beyond the target language.

The corresponding recall, precision, and F1 score values for the repaired grammar variants are shown in columns R_p, P_p, and

[4] This is scalable because the candidates can be evaluated in parallel, so this gives a good indication of a real-world scenario.

Table 3: Summary of results showing accuracy of the passive repair approach and the number of applied patches for each repaired grammar.

			accuracy						symbol edit				listification	
#	$\mathcal{L}$	bugs	R_o	P_o	$F1_o$	R_p	P_p	$F1_p$	$\mathfrak{d}$	$\mathfrak{i}$	$\mathfrak{s}$	$\mathfrak{t}$	$\mathfrak{L}_1$	$\mathfrak{L}_2$
1	$\mathcal{Toy}$	2	0.902	0.649	0.755	0.997	0.637	0.777			2			
2	$\mathcal{A}$	1	0.972	0.769	0.859	0.975	0.751	0.848						
3	$\mathcal{A}$	1	0.972	0.910	0.940	0.999	0.985	0.992			1			
4	$\mathcal{B}$	2	0.952	0.902	0.926	0.999	0.910	0.952					1	1
5	$\mathcal{C}$	1	0.969	0.999	0.984	0.977	0.930	0.953						1
6	$\mathcal{D}$	1	0.638	0.465	0.538	0.998	0.759	0.862	1					
7	$\mathcal{E}$	4	0.818	0.733	0.773	0.987	0.704	0.822	2	14	3			1
8	$\mathcal{E}$	4	0.579	0.499	0.536	0.974	0.521	0.679	3	2	4			2
9	$\mathcal{F}$	2	0.939	0.792	0.859	0.996	0.560	0.717	3		2			
10	$\mathcal{F}$	>5	0.697	0.466	0.559	0.909	0.391	0.547						
11	$\mathcal{G}$	2	0.743	0.595	0.661	0.956	0.913	0.934						2
12	$\mathcal{G}$	-	0.999	0.333	0.460	-	-	-						
13	$\mathcal{G}$	2	0.821	0.333	0.474	0.999	0.589	0.741	1		2			
14	$\mathcal{G}$	9	0.366	0.333	0.349	0.991	0.806	0.889			1		6	2
15	$\mathcal{G}$	1	0.931	0.761	0.837	0.999	0.736	0.848			3			
16	$\mathcal{G}$	1	0.971	0.574	0.700	1.000	0.875	0.933			1			
17	$\mathcal{G}$	1	0.932	0.938	0.935	1.000	0.898	0.946			1			
18	$\mathcal{G}$	4	0.886	0.786	0.833	0.915	0.768	0.835	1	4	2			
19	$\mathcal{G}$	2	0.936	0.529	0.675	0.963	0.485	0.645	2	1				2
20	$\mathcal{H}$	-	0.994	0.553	0.711	-	-	-						
21	$\mathcal{H}$	2	0.896	0.782	0.835	0.994	0.705	0.825	1	2	1			
22	$\mathcal{H}$	4	0.860	0.827	0.843	0.952	0.672	0.788	3	1	2			2
23	$\mathcal{H}$	2	0.978	0.894	0.934	0.982	0.899	0.939	1					1
24	$\mathcal{H}$	4	0.912	0.720	0.805	0.953	0.727	0.825	1	1				2
25	$\mathcal{H}$	>5	0.945	0.762	0.843	0.974	0.655	0.783						
26	$\mathcal{H}$	-	0.999	0.549	0.709	-	-	-						
27	$\mathcal{H}$	2	0.984	0.641	0.776	0.984	0.604	0.749	1	1				
28	$\mathcal{H}$	1	0.962	0.718	0.822	0.980	0.717	0.828						1
29	$\mathcal{H}$	1	0.899	0.913	0.906	0.985	0.909	0.945	1					
30	$\mathcal{H}$	-	0.984	0.787	0.875	-	-	-						
31	$\mathcal{H}$	2	0.962	0.805	0.877	0.980	0.810	0.887		1				
32	$\mathcal{H}$	1	0.990	0.828	0.902	0.993	0.832	0.905	1					
33	$\mathcal{H}$	>5	0.332	0.333	0.332	0.918	0.670	0.774						

$F1_p$, respectively. We see from the table a significant increase in recall in most cases; we even achieve 100% recall for two grammars (#16 and #17). The precision results, however, are mixed and sometimes the repaired grammar has lower precision. Overall, this translates to slightly better F1 scores compared to the input grammars. This shows that even though we achieve moderate recall improvements with our passive repair approach, it often produces grammars that over-generalize beyond the target language. This problem is addressed in the active repair approach (see Section 8.3).

Applied Patches. The right-most columns of Table 3 give insight on the interaction of the grammar transformations discussed in Sections 4 and 5 to induce the fixes described above. Specifically, it shows for each repair how often each patch type was applied. Here, $\mathfrak{d}$ is symbol deletion, $\mathfrak{i}$ is symbol insertion, $\mathfrak{s}$ is symbol substitution, while $\mathfrak{t}$ is the symbol transposition patches. $\mathfrak{L}_1$ means right recursion introduction and $\mathfrak{L}_2$ the list synthesis patches. Note that we are repairing the input grammar in our experimental setup against a fixed test suite containing positive tests only; hence, the language-tightening transformations described in Section 6 are never used for repairs. Note also that we do not consider the patch usage count for partial repairs.

Most patch types are used widely, but symbol transposition is not applied at all. More specifically, symobol deletion is applied 22 times, symbol insertion 27 times, symbol substitution 25 times, and two listification patches are applied 7 and 17 times respectively.

RQ1. The passive repair approach is effective in fixing faults in medium-sized grammars with real faults. It fully repaired 25 out of 33 grammars against a *cdrc* test suite for the target grammar as repair specification, and partially repaired four grammars. The repairs universally improve the recall but reduce the precision in about half of the cases, indicating that the repaired grammars over-generalize beyond the target language.

Table 4: Active repair results for student grammars.

			tests		gfixr			accuracy		
#	$\mathcal{L}$	bugs	TS_{init}	fails	iter.	cand.	time	R_a	P_a	$F1_a$
1	$\mathcal{T}oy$	3	(86, 218)	18	4	227	00:06:49	1.000	1.000	1.000
2	$\mathcal{A}$	7	(179, 182)	36	7	1184	00:32:20	1.000	0.976	0.988
3	$\mathcal{A}$	2	(179, 203)	8	2	224	00:06:41	0.999	1.000	0.999
4	$\mathcal{B}$	4	(79, 82)	7	4	324	00:09:50	1.000	1.000	1.000
5	$\mathcal{C}$	1	(86, 104)	1	1	2	00:00:30	0.977	0.935	0.956
6	$\mathcal{D}$	3	(80, 116)	98	4	739	00:20:12	1.000	1.000	1.000
7	$\mathcal{E}$	7	(199, 371	80	15	1605	00:48:49	0.987	1.000	0.993
8	$\mathcal{E}$	8	(199, 137)	78	24	6069	05:57:27	1.000	0.838	0.912
9	$\mathcal{F}$	4	(212, 173)	25	7	1037	01:07:49	0.998	1.000	0.999
10	$\mathcal{F}$	>5	(212, 155)	149	150	12972	20:01:01	0.794	0.457	0.580
11	$\mathcal{G}$	4	(194, 221)	33	3	1035	00:34:26	1.000	1.000	1.000
12	$\mathcal{G}$	3	(194, 121)	15	4	245	00:06:32	1.000	1.000	1.000
13	$\mathcal{G}$	5	(194, 104)	22	5	323	00:11:07	1.000	1.000	1.000
14	$\mathcal{G}$	9	(194, 70)	249	10	4934	02:49:56	0.991	0.829	0.903
15	$\mathcal{G}$	5	(194, 209)	25	10	3573	03:02:05	0.962	0.633	0.764
16	$\mathcal{G}$	2	(194, 104)	9	2	316	00:08:41	1.000	1.000	1.000
17	$\mathcal{G}$	2	(194, 104)	11	2	108	00:08:01	1.000	1.000	1.000
18	$\mathcal{G}$	6	(194, 214)	53	18	4184	02:00:55	0.914	0.802	0.854
19	$\mathcal{G}$	>5	(194, 134)	41	70	3143	02:12:14	0.832	0.373	0.515
20	$\mathcal{H}$	3	(205, 233)	120	4	485	00:23:54	0.985	1.000	0.992
21	$\mathcal{H}$	4	(205, 213)	134	7	1221	01:05:08	0.994	0.943	0.968
22	$\mathcal{H}$	5	(205, 303)	72	8	3682	02:42:17	0.961	0.839	0.961
23	$\mathcal{H}$	3	(205, 272)	3	3	131	00:05:03	0.982	0.987	0.984
24	$\mathcal{H}$	6	(205, 186)	104	8	1444	01:15:06	0.953	0.999	0.975
25	$\mathcal{H}$	>5	(205, 300)	137	150	9196	07:59:45	0.973	0.806	0.881
26	$\mathcal{H}$	1	(205, 265)	119	1	324	00:08:56	0.999	0.952	0.975
27	$\mathcal{H}$	3	(205, 306)	172	4	1033	00:58:53	0.984	0.971	0.977
28	$\mathcal{H}$	4	(205, 114)	71	7	1126	00:53:28	0.980	0.909	0.943
29	$\mathcal{H}$	1	(205, 113)	44	1	102	00:06:54	0.985	0.904	0.943
30	$\mathcal{H}$	3	(205, 160)	3	3	308	00:15:04	0.984	0.999	0.991
31	$\mathcal{H}$	3	(205, 227)	44	3	637	00:33:13	0.980	1.000	0.990
32	$\mathcal{H}$	1	(205, 322)	1	1	238	00:13:20	0.993	0.836	0.908
33	$\mathcal{H}$	>5	(205, 261)	466	150	11586	15:28:59	0.921	0.740	0.820

8.3. Active Repair Results (RQ2)

Table 4 summarizes the repair results using the active repair approach described in Section 3.7. Here, TS_{init} comprises the *cdrc* test suite $TS_{\mathcal{L}}$ that is generated from the target language (that also serves as oracle O), and an initial *cdrc* test suite TS_G generated from the input grammar G. Note that TS_G can contain negative tests if G over-generalizes $\mathcal{L}$. Table 4 shows the sizes of $|TS_{\mathcal{L}}|$ and $|TS_G|$. Columns R_a, P_a, and $F1_a$ show recall, precision, and F1 scores for the repaired grammar, respectively. Candidates where only a partial repair is found in 150 iterations are again shaded in grey.

Note also that the active repair loop can stop with a candidate that is a full repair with respect to TS_{init} but still fails some of the tests generated from some other repair candidates. These "premature" terminations are shown in a lighter shade of grey. We could find better repairs by restarting the repair process with these tests added to TS_{init}, but we leave this for future work.

Efficacy. First, Table 4 shows that the incorporation of the oracle allows us to construct tailor-made repair test suites from each grammar, by adding TS_G to $TS_{\mathcal{L}}$. This leads, for most grammars, to an increase in the number of bugs revealed by the test suite TS_{init} compared to the previous passive case, e.g., our running example has three bugs exposed here, an increase from just two in the passive repair experiments. Grammar #2 in particular exhibits the biggest jump from one bug revealed in the passive case to seven in the active case. We see also that TS_{init} now reveals bugs in grammars #12, #20, #26, and #30 that were marked as non-buggy in the previous experiment.

Second, our approach finds fixes in less than 20 iterations in most cases. This also shows that the fault localizer remains effective and identifies faults sufficiently well. However, the active repair still returns partial repairs for six grammars. Out of these, grammars #10, #25, and #33 require more than 150 iterations and the repair loop terminates prematurely for the grammars #15, #18, and #19.

Table 5: Patches applied by the active repair approach for each faulty grammar.

			symbol edit				list.		tightening			
#	$\mathcal{L}$	bugs	$\mathfrak{d}$	$\mathfrak{i}$	$\mathfrak{s}$	$\mathfrak{t}$	$\mathfrak{L}_1$	$\mathfrak{L}_2$	$\mathfrak{S}$	$\mathfrak{D}$	$\mathfrak{P}$	$\mathfrak{E}$
1	*Toy*	3			2				1	1		
2	$\mathcal{A}$	7						1	1	2		4
3	$\mathcal{A}$	2			1				1	1		
4	$\mathcal{B}$	4					2					2
5	$\mathcal{C}$	1						1				
6	$\mathcal{D}$	3	1						2	3		
7	$\mathcal{E}$	7	2	2				1	4	6	1	2
8	$\mathcal{E}$	8	3					2	17	19		
9	$\mathcal{F}$	4	1		5				1	2		
10	$\mathcal{F}$	>5										
11	$\mathcal{G}$	4			1			1		1		
12	$\mathcal{G}$	3							3	4		
13	$\mathcal{G}$	5			4					1		
14	$\mathcal{G}$	9	1				7	3				
15	$\mathcal{G}$	5										
16	$\mathcal{G}$	2			1				1	1		
17	$\mathcal{G}$	2			1				1	1		
18	$\mathcal{G}$	6										
19	$\mathcal{G}$	>5										
20	$\mathcal{H}$	3								1		2
21	$\mathcal{H}$	4	1	2	1				1	3		
22	$\mathcal{H}$	5	3	2				2	1	1		
23	$\mathcal{H}$	3		1				1		1		
24	$\mathcal{H}$	6						4	1	4		
25	$\mathcal{H}$	>5										
26	$\mathcal{H}$	1								1		
27	$\mathcal{H}$	3	2					1		1		
28	$\mathcal{H}$	4						1	5	5	1	
29	$\mathcal{H}$	1	1									
30	$\mathcal{H}$	3							2	2	1	
31	$\mathcal{H}$	3		1	1					1		
32	$\mathcal{H}$	1	1									
33	$\mathcal{H}$	>5										
		Total	16	8	17	0	9	18	42	62	3	10

Third, repair times are typically below 30 minutes, with about five grammars where the full repair took more than 60 minutes. This is a significant increase in runtimes compared to the passive case, but an increase in the number of revealed bugs trivially means we see an increase in the number of iterations and generated candidate patches.

Finally, we see that the active repair configuration can direct the fault localization. For example, for grammar #2 in the passive case, it took over four hours and 150 iterations to fix one fault that caused three test failures because the fault localizer could not identify the correct repair site because of some unexpected behaviour in CUP's parsing algorithm. Here, however, the faulty rule was correctly identified because the oracle rejected all test cases where it was applied in their derivation.

Accuracy Evaluation. The second part of Table 4 gives the detailed accuracy of the fixes. We see that our active repair approach significantly improves recall; we even achieve 100% recall in ten cases (i.e., in about a third of the cases). The active repair approach also produces "tight" patches with respect to the target language. We achieve perfect precision in thirteen cases, which is about half of the cases where the repair loop returned a full fix. The repaired grammar variants, on average, improve the quality of the input grammars by about 1.5×. These variants approximate the target language sufficiently well, in fact, we even achieve 100% F1 score in eight cases, which demonstrates that the subsets of the languages described by one these repaired grammar variants and their corresponding target grammar are (approximately) equivalent with respect to the validation suite.

In addition to some limitations (see Section 9 below) that in some cases prevent the active repair approach from achieving 100% precision, we also observed some *sampling biases* effects as described by Rossouw and Fischer (2021). In fact, we generate test suites using the same generic cover algorithm (Fischer

et al., 2011) used there to describe and evaluate these biases. In our repair case here, counter-examples are not generated that would make the right patches to have better fitness than patches that over-generalize (i.e., make the grammar too permissive) the language.

Applied Patches. Table 5 gives the detailed number of patches applied for each repaired grammar. The patch types are labeled as before but we now also include language tightening patches here: $\mathfrak{S}$ refers to the non-terminal and token splitting patches, $\mathfrak{D}$ to the rule deletion patch, $\mathfrak{P}$ to the push-down list elements patches and $\mathfrak{E}$ to the immediate recursion elimination patch.

Like in the passive case, most of the patch types are used widely, but symbol transposition remains unused. More specifically, symbol deletion is applied 16 times, symbol insertions 8 times, symbol substitution 17, and two listification patches are applied 9 and 18 times respectively. We also see that non-terminal (and token) splitting and rule deletion (used 42 and 62 times, respectively) are the most widely used language tightening transformations, with rule deletion used in all but 4 grammars. The last two list tightening patches are applied 3 and 10 times respectively.

RQ2. The active repair approach is effective in fixing faults in medium-sized grammars with real faults. It fully repaired 27 out of 33 grammars, and partially repaired six grammars. The repairs universally improve the recall, precision, and the F1 scores in about more than half of the grammars.

8.4. Pasive Repair vs Active Repair (RQ3)

In this section, we compare the passive and active repair approaches "like-for-like" for all 33 grammars. Figure 2 summarizes the results of this comparison through a series of boxplots. We see that active repair produces repairs with better recall, and achieves 100% recall in ten grammars, while passive repair achieves that in only two cases. We also observe the passive repair approach induces patches that over-generalize beyond the target language, as its repaired grammars give low precision values. In the active case, however, incorporating test suite generation and membership queries into the repair loop prevents some over-generalization to some degree even when using weaker test suites like *cdrc* as repair specifications.

RQ3. The active repair approach produces patches that generalize better than the passive repair approach, but the runtimes are generally higher.

9. Limitations

In this section, we discuss some limitations that affect the efficacy of our proposed repair approach. These can, in part, be attributed to our choice of parsing tools, more generally the underlying parsing algorithms, and to shortcomings of our realization of both variants in gfixr.

9.1. Mislocalization due to Unexpected Behaviour

The first limitation concerns CUP, the parser generator we use to log grammar spectra for fault localization. More specifically, we illustrate how, in some cases, CUP returns the wrong spectra and therefore does not identify the faulty rules. Consider in particular grammar #2 from Table 2, where *startVarIDs*- and *variableIDs*-rules are defined as follows:

$$\begin{aligned} \textit{startVarIDs} &\rightarrow \texttt{IDENT}\ \textit{variableIDs} \\ \textit{variableIDS} &\rightarrow \bullet\ \texttt{IDENT COMMA variableIDs} \mid \varepsilon \end{aligned}$$

The original intent was to capture a `COMMA`-separated list of identifiers (`IDENT`). The fault location (identified by a manual inspection) is marked by •. The grammar fails the following three test cases

```
EENHEID a BEGIN VER a •, a : WAARHEID EINDE a .

EENHEID a BEGIN
  FUNKSIE a(a •, a : WAARHEID) : WAARHEID := BEGIN EINDE a
  EINDE a .

EENHEID a BEGIN VER a •, a, a : WAARHEID EINDE a .
```

We also use the •-symbol here to mark the error location observed in the inputs.

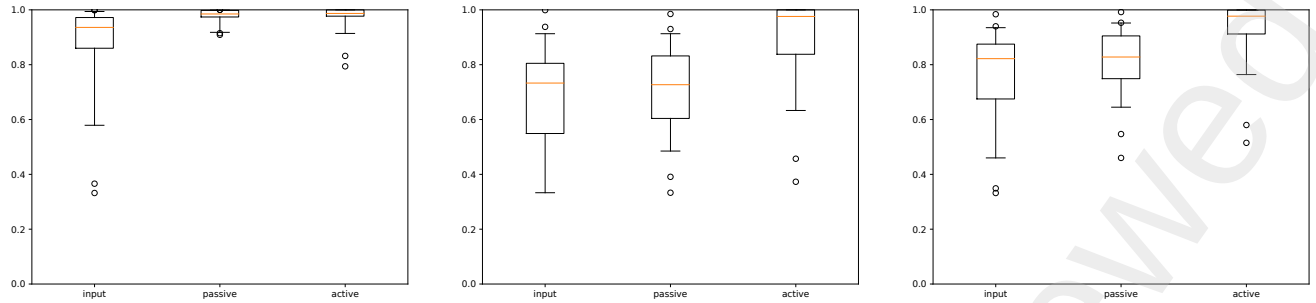
All items from the *startVarIDs*-rule, i.e., *startVarIDs*:1:0, *startVarIDs*:1:1 and *startVarIDs*:1:2 have the same spectral counts, each with a fail count of 3 and pass count of 26. Our tie resolution strategy that prefers items with the right-most designated position over other items from the same rules in a tie, picks the reduction item *startVarIDs*:1:2. The item *variableIDs*:2:0 (i.e., the ε-production) also has the same counts as the items from the *startVarIDs*-rule because the ε-production is applied just before the error location. However, none of the items (which include the faulty *variableIDs*:1:0) from the first alternative of the *variableIDs*-rule are executed in any of the test cases, i.e., the rule is not executed in either failing or passing test cases and therefore has spectral counts of zero. Hence, it is never selected for repair.

The above illustration explains why it took gfixr a little over 4 hours, 150 iterations and generated 10744 candidate patches as shown in Table 2.

9.2. Parsing Restrictions

As we mentioned earlier, applying splitting patches to a rule $A \rightarrow \gamma_i$ can introduce parsing conflicts. In our experimental evaluation, we observed that conflict introduction is indeed prevalent despite some control measures we put in place to mitigate their effects. First, we do not attempt to repair input grammars with reduce/reduce conflicts to prevent parsing instability. Second, we discard patches that introduce reduce/reduce conflicts into the grammars (except for two cases in the passive repair experiments where we set the number of allowed reduce/reduce conflicts to one (i.e., `-rr = 1`)). Finally, for grammars #20 to #33, we set the flag `-non_lr` because the language is inherently ambiguous by design.

While in most cases, enabling `-non_lr` worked well, in some cases LR(1) parsing restrictions did not allow the search to stop with good quality patches. In particular, consider grammar



(a) Recall. (b) Precision. (c) F1 scores.

Figure 2: Accuracy evaluation results for passive and active grammar repairs. Higher is better. Recall results on the left plot, precision results in the middle and F1 scores on the right. `input` shows results for input grammar G, `passive` for passive repair approach and `active` for active repair approach.

#8 where the target language has the same expression structure as our example grammar in Figure 1. The *simple-* and *simple_list-*rules are written as

$$\begin{aligned} \mathit{simple} &\rightarrow \mathit{term}\ \mathit{simple\_list} \mid \texttt{-}\ \mathit{term}\ \mathit{simple\_list} \\ \mathit{simple\_list} &\rightarrow \mathit{simple\_list}\ \mathit{addop} \mid \mathit{simple\_list}\ \mathit{term} \mid \varepsilon \end{aligned}$$

The oracle rejects test cases derived from these rules, including the tests

```
source a begin a ::= 0 - end
source a begin a ::= a a a end
```

The non-terminal splitting patch correctly transforms the *simple_list-*rule to the following seven alternatives

$$\begin{aligned} \mathit{simple\_list} \rightarrow\ & \mathit{simple\_list}\ \mathit{addop}\ \mathit{term} \mid \mathit{simple\_list}\ \mathit{term}\ \mathit{term} \mid \mathit{term} \\ & \mid \mathit{simple\_list}\ \mathit{addop}\ \mathit{addop} \mid \mathit{simple\_list}\ \mathit{term}\ \mathit{addop} \\ & \mid \mathit{addop} \mid \varepsilon \end{aligned}$$

The expectation was that, in the next iteration, gfixr would apply rule deletion patches to the second, third, fourth, fifth and sixth alternatives of *simple_list*, as they are applied in the generation of rejected tests and incorrectly accept the two test cases shown above. Deleting these rules would leave us with the following correct *simple_list* rules

$$\mathit{simple\_list} \rightarrow \mathit{simple\_list}\ \mathit{addop}\ \mathit{term} \mid \varepsilon$$

However, the LALR parsing algorithm implemented by CUP is too restrictive and did not allow for this, and gfixr returned a patch with 0.838 precision.

9.3. Loop Restart

In active repair, the basic idea of a repair loop restart is to stop the search after some iteration when some specified condition is met (e.g., time-out expired), collect all the information learned so far, this includes the candidate C at the front of the priority queue that awaits further processing, and current set of failing tests TS_F from the common test pool TS. Then (automatically or manually) restart the repair process with the candidate C or even the input grammar G, with the user-provided input test suite, and previous failing test cases TS_F added to the new common test pool TS'.

While the development of several conditions that warrant restarting the search are left for future work, the current version of gfixr prompts the user to manually restart the repair process when the candidate variant C at the front of the queue is consistent with the oracle O on the initial test suite (recall from Table 4 that this includes a user provided seeds and test suite generated from the faulty input grammar G), but rejects a test case $w \in L(O)$ that was added to TS in some later iteration. Patches from grammars #15, #18, and #19 were prompted to restart the search.

10. Threats to Validity

Our observations are based on experiments conducted using medium-sized grammars written by students. While the faults made by students are real and tend to be unpredictable, our results may not generalize to other grammars, to other ranking metrics used for localization, or to other parsing environments.

The grammar transformations described in this work are mostly example-driven; we carefully inspected different faults and designed corresponding transformations that target these faults. They may not be sufficient to other target grammars. However, the different patch types are widely used in most grammars, which gives us some confidence in their generality. Our implementation also allows for easier integration of more transformations. However, we focus almost exclusively on syntactic elements; the token creation and splitting transformations only address specific lexical issues and grammars suffering from other lexical problems may not be fixable.

Finally, we mitigated against the usual internal validity threats of human error, human bias and human performance by automating experiments, carefully testing our implementation and scripts, and by using well-established tools for item-level spectra collection and test suite generation.

11. Related Work

11.1. Grammar Transformations

Lämmel (2001a) and Zaytsev (2009, 2010) have defined general grammar transformations and used them for grammar construction, refactoring, and adaptation (Lämmel and Zaytsev,

2009a; Zaytsev, 2014), including the extraction and comparison of several complete grammars from different language specifications (Lämmel and Verhoef, 2001; Lämmel and Zaytsev, 2009b). Jain et al. (2004) 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.

11.2. Grammar Learning

Grammar learning (also known as *grammatical inference*) denotes a process of deriving an adequate grammar for a finitely presented (e.g., via examples) but typically infinite language. While there are different techniques applied to the problem of grammar learning, we discuss search-based methods (in particular, methods that employ genetic algorithms) and inductive grammar learning methods.

Genetic Grammar Learning. Genetic algorithms (GA) have been used to learn CFGs from test suites. The applied genetic operations include point mutations such as replacement, insertion, or deletion of symbols (Di Penta et al., 2008) and modification of EBNF operators (Crepinsek et al., 2005) in a single rule, global mutations such as merging and splitting of nonterminal symbols (Petasis et al., 2004), mutated rule duplication (Di Penta et al., 2008), or different rule generalizations (Petasis et al., 2004), 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 usually evaluated, as in our approach, by running the corresponding candidate over the test suite; in practice, results can improve if positive examples get priority, but negative examples are required to prevent overgeneralization (Crepinsek et al., 2005). Scoring functions are typically based on some version of balanced accuracy, sometimes taking the length of the longest recognized fragment into account (Lankhorst, 1994). Our priority function follows similar ideas.

Unlike in our grammar repair task, where generation of test suites from candidate grammars and use of an oracle O to answer membership queries on the generated sentences, are intrinsically incorporated into the repair loop, the GA-based algorithm by Crepinsek et al. (2005) leaves sentence generation from candidates after the plausible candidate that parses all positive examples is returned; in order to determine the need for further introduction of negative examples. The eg-GRIDS system by Petasis et al. (2004) ignores sentence generation from candidates completely, but the authors use it as a measure of quality on the output grammar variant that captures the positive input training set. The system also does away with leveraging negative evidence to avoid some overgeneralization; the *minimum description length* (MDL) principle is rather employed that ensures "compact" learned grammars with respect to encoded training examples.

Di Penta and Taneja (2005) and Di Penta et al. (2008) used GAs to learn the well-separated extension of a programming language, starting from the full grammar of the base language. Their inference approach, however, involves an initial manual flagging of differences between the source grammar and its dialect, then extracts sub-grammars from the source that reflect those differences because earlier attempts to infer a complete general-purpose grammar did not yield favourable results. We showed in our earlier work that our approach can be used to capture the dialect of a language; we rely on fault localization to automatically identify deviations between the input grammar and its dialect, and we do not derive subsets of the input grammar. However, we are aware that we may be addressing slightly different problems, and it remains an interesting and open question to see how our approach can be used to replace the blind rule selection in genetic grammar learning methods.

Inductive grammar learning. Our work can be seen as grammatical inference, which has a long history (e.g., Solomonoff (1959)) and has been widely addressed, both in theory and in practice (see (Lee, 1996; Sakakibara, 1997; de la Higuera, 2010; Stevenson and Cordy, 2014) for overviews).

Our approach has the full test suite available with access to a membership oracle. It therefore sits between Gold's model of *identification in the limit* (Gold, 1967), where observations are presented in sequence (and approaches are often order-sensitive, e.g., (Knobe and Knobe, 1976)) and Angluin's *query model* (Angluin, 1987), 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 (Sakakibara, 1997; Drewes and Högberg, 2003).

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 (Isberner, 2015). We focus on heuristic approaches here.

Several systems such as Synapse (Nakamura and Ishiwata, 2000; Nakamura, 2006) or Gramin (Saha and Narula, 2011) 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 (Bastani et al., 2017) 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 (Solomonoff, 1959). 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.

Kulkarni et al. (2021) introduce and evaluate a non-deterministic grammar learning tool, Arvada, that takes as input (like Glade) training examples $\mathcal{S}$ and an oracle O that answers membership queries. From each input example $s \in \mathcal{S}$, the tool creates a flat tree (i.e., a tree with a root node with all characters c_i from s as leaf labels). The main learning loop of Arvada can be summarized by two heuristic generalization operations; (i) *bubbling* which introduces new non-terminals by assigning a new parent node (with non-terminals as labels) to a sequence of sibling nodes; and (ii) *merging* which subsequently validates bubbles by checking whether two nodes t_a and t_b can be commutatively substituted, i.e., if replacing t_a by t_b (and vice versa), does not produce words outside the target language. Arvada's experimental evaluation shows that it achieves higher recall (i.e., Arvada-mined grammars generalize better to unseen tests) and better F1 scores than Glade. This result led to one of the few and rare replication studies published in the history of the PLDI conference (Bendrissou et al., 2022). The replication study disputes some of the claims of the original paper by Bastani et al. (2017) such as "overly optimistic" F1 scores and raises scalability concerns about Glade.

AUTOGRAM (Höschele and Zeller, 2016, 2017) 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 (Mathis et al., 2019), the parser itself is used to guide the sentence generation. Mimid (Gopinath et al., 2020) extends this to extract an explicit CFG.

11.3. Grammar-based Test suite Construction

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* (Lämmel, 2001b), *k-path* coverage Havrikov and Zeller (2019), derivable pair coverage (van Heerden et al., 2020), or automata-based methods (Zelenov and Zelenova, 2005; Rossouw and Fischer, 2020).

Grammar-based fuzzers (e.g., LangFuzz (Holler et al., 2012) and IFuzzer (Veggalam et al., 2016)) mostly use random sentence generation techniques, and often exploit a given corpus to extract seed code fragments (Holler et al., 2012; Veggalam et al., 2016; Wang et al., 2017). Nautilus (Aschermann et al., 2019) exploits grey-box access to the SUT to provide feedback to the sentence generation. These systems all assume that a correct grammar is available.

11.4. Automatic Program Repair

Automatic program repair (APR) techniques, also called automatic patch generation or automatic bug fixing, take as input a faulty program and a set of test cases which include at least one fault-revealing test case, exploit fault localization to identify potential repair sites, apply modifications either directly at source code or binary level to these sites and give an output of a repaired variant of the program that is consistent with the input test cases or meets some specifications or output none if the repair cannot be found. APR comes in different flavours, e.g., generate-and-validate, semantics- and data-driven techniques. Studies by Monperrus (2018), (Gazzola et al., 2019), and Ghanbari et al. (2019) give a comprehensive view of the field.

Like our repair task, many approaches are based on generating candidate patches using different search strategies such as genetic programming (Arcuri, 2009; Arcuri and Yao, 2008; Arcuri, 2011; Forrest et al., 2009; Weimer et al., 2009, 2010; Le Goues et al., 2012), random search (Qi et al., 2014, 2013; Ji et al., 2016), or bug templates (Martinez and Monperrus, 2018; Liu et al., 2019b; Saha et al., 2017; Liu and Zhong, 2018; Liu et al., 2019a; Koyuncu et al., 2020; Kim et al., 2013; Hua et al., 2018) and validating each candidate 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 (Le Goues et al., 2012) 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. gfixr uses the Ochiai (Ochiai, 1957) metric (see Section 2), another common technique, but can be easily extended to work with other metrics.

In semantics-driven approaches (Ke et al., 2015; Nguyen et al., 2013; Roychoudhury, 2016; Le et al., 2017) the faulty code fragments (which are identified using spectrum-based fault localization approaches) are executed symbolically while the non-faulty fragments are executed using concrete values using symbolic execution. This family of APR techniques share the use of spectrum-based fault localization with our work, but because grammars do not have an equivalent executable semantic model, their underlying ideas are not obviously transferrable.

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.

12. Conclusions and Future Work

We described the first approach to fix faults in context-free grammars automatically. Our approach alternates over three key steps and gradually improves the grammars until all tests used as repair specification all pass: (*i*) We use a fine-grained spectrum-based fault localization method to identify suspicious items

(i.e., specific positions within 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. (*iii*) We validate each candidate grammar over the same test suite to determine if it improves over the parent grammar. Candidates that do not improve over the parent are discarded. We further use a priority queue to keep improving the most promising candidates.

We described two variations of our repair approach: (*i*) In the passive repair variant, we repair against a fixed test suite specification. (*ii*) In the active repair variant, we exploit a membership oracle and introduce a test suite enrichment where we generate new tests from each candidate grammar, and use the oracle to confirm the expected outcome of these tests.

We implemented these ideas in the gfixr system, and successfully used it to fix CUP grammars that students submitted as homeworks in a compiler engineering course. Both repair variants are effective in fixing real and multiple faults in grammars. A comparison of both variations showed that the active repair approach works better than passive repair. We got better F1 scores and achieved many perfect fixes with the active repair than the passive repair approach.

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) (Lämmel, 2001a; Zaytsev, 2009).

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 (Economopoulos et al., 2009).

Finally, we plan to use the repair in teaching; in particular, we plan to integrate gfixr into or gtutr feedback system (Barraball et al., 2020) to help students to improve their grammars when they are stuck.

Acknowledgements

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

Appendix A. *cdrc* Test Suite $TS_{\mathcal{T}oy}$

The test suite $TS_{\mathcal{T}oy}$ we use in the running example to illustrate different types of grammar transformations.

```
program a begin boolean array a; relax end
program a begin boolean a, a, a; relax end
program a begin boolean a, a; relax end
program a begin boolean a; boolean a; relax end
program a begin boolean a; relax end
program a begin a() end
program a begin a(0) end
program a begin a(0, 0) end
program a begin a(0, 0, 0) end
program a begin a := array 0 end
program a begin a := 0 end
program a begin a[0] := 0 end
program a begin if 0 then leave end end
program a begin if 0 then relax else leave end end
program a begin if 0 then relax else relax end end
program a begin if 0 then relax elsif 0 then leave end end
program a begin if 0 then relax elsif 0 then relax elsif 0 then relax end end
program a begin if 0 then relax elsif 0 then relax end end
program a begin if 0 then relax end end
program a begin integer a; relax end
program a begin leave; a() end
program a begin leave; if 0 then relax end end
program a begin leave; leave; leave end
program a begin leave; leave end
program a begin leave; read a end
program a begin leave; while 0 do relax end end
program a begin leave; write "" end
program a begin leave end
program a begin read a[0] end
program a begin read a end
program a begin relax end
program a begin while 0 do leave end end
program a begin while 0 do relax end end
program a begin write(0) end
program a begin write - 0 end
program a begin write false end
program a begin write a() end
program a begin write a[0] end
program a begin write a end
program a begin write not(0) end
program a begin write not false end
program a begin write not a end
program a begin write not not 0 end
program a begin write not 0 end
program a begin write not true end
program a begin write 0 # 0 end
program a begin write 0 % 0 end
program a begin write 0 * (0) end
program a begin write 0 * false end
program a begin write 0 * a end
program a begin write 0 * not 0 end
program a begin write 0 * 0 * 0 end
program a begin write 0 * 0 end
program a begin write 0 * true end
program a begin write 0 + 0 + 0 end
program a begin write 0 + 0 end
program a begin write 0 - 0 end
program a begin write 0 / 0 end
program a begin write 0 < 0 end
program a begin write 0 <= 0 end
program a begin write 0 = 0 end
program a begin write 0 > 0 end
program a begin write 0 >= 0 end
program a begin write 0 and 0 end
program a begin write 0 end
program a begin write 0 or 0 end
program a begin write "" . 0 end
program a begin write "" . "" . "" end
program a begin write "" . "" end
program a begin write "" end
program a begin write true end
program a define a(boolean array a) begin relax end begin relax end
program a define a(boolean a) -> boolean a begin relax end begin relax end
program a define a(boolean a) -> integer a begin relax end begin relax end
program a define a(boolean a) begin relax end begin relax end
program a define a(boolean a) begin relax end define a(boolean a) begin relax end begin relax end
program a define a(boolean a, boolean a) begin relax end begin relax end
program a define a(boolean a, boolean a, boolean a) begin relax end begin relax end
program a define a(integer a) begin relax end begin relax end
```

References

Abreu, R., Zoeteweij, P., van Gemund, A.J.C., 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. pp. 39–46. URL: https://doi.org/10.1109/PRDC.2006.18, doi:10.1109/PRDC.2006.18.

Abreu, R., Zoeteweij, P., van Gemund, A.J.C., 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. pp. 88–99. URL: https://doi.org/10.1109/ASE.2009.25, doi:10.1109/ASE.2009.25.

Angluin, D., 1987. Queries and concept learning. Mach. Learn. 2, 319–342. URL: https://doi.org/10.1007/BF00116828, doi:10.1007/BF00116828.

Arcuri, A., 2009. Automatic software generation and improvement through search based techniques. Ph.D. thesis. University of Birmingham, UK. URL: http://etheses.bham.ac.uk/400/.

Arcuri, A., 2011. Evolutionary repair of faulty software. Appl. Soft Comput. 11, 3494–3514. URL: https://doi.org/10.1016/j.asoc.2011.01.023, doi:10.1016/j.asoc.2011.01.023.

Arcuri, A., Yao, X., 2008. A novel co-evolutionary approach to automatic software bug fixing, in: Proceedings of the IEEE Congress on Evolutionary Computation, CEC 2008, June 1-6, 2008, Hong Kong, China, IEEE. pp. 162–168. URL: https://doi.org/10.1109/CEC.2008.4630793, doi:10.1109/CEC.2008.4630793.

Aschermann, C., Frassetto, T., Holz, T., Jauernig, P., Sadeghi, A., Teuchert, D., 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. URL: https://www.ndss-symposium.org/ndss-paper/nautilus-fishing-for-deep-bugs-with-grammars/.

Barraball, C., Raselimo, M., Fischer, B., 2020. An interactive feedback system for grammar development (tool paper), in: Lämmel, R., Tratt, L., de Lara, J. (Eds.), Proceedings of the 13th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2020, Virtual Event, USA, November 16-17, 2020, ACM. pp. 101–107. URL: https://doi.org/10.1145/3426425.3426935, doi:10.1145/3426425.3426935.

Bastani, O., Sharma, R., Aiken, A., Liang, P., 2017. Synthesizing program input grammars, in: Cohen, A., Vechev, M.T. (Eds.), Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017, ACM. pp. 95–110. URL: https://doi.org/10.1145/3062341.3062349, doi:10.1145/3062341.3062349.

Bendrissou, B., Gopinath, R., Zeller, A., 2022. "synthesizing input grammars": a replication study, in: Jhala, R., Dillig, I. (Eds.), PLDI '22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA, USA, June 13 - 17, 2022, ACM. pp. 260–268. URL: https://doi.org/10.1145/3519939.3523716, doi:10.1145/3519939.3523716.

Chen, M.Y., Kiciman, E., Fratkin, E., Fox, A., Brewer, E.A., 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, IEEE Computer Society. pp. 595–604. URL: https://doi.org/10.1109/DSN.2002.1029005, doi:10.1109/DSN.2002.1029005.

Crepinsek, M., Mernik, M., Bryant, B.R., Javed, F., Sprague, A.P., 2005. Inferring context-free grammars for domain-specific languages. Electron. Notes Theor. Comput. Sci. 141, 99–116. URL: https://doi.org/10.1016/j.entcs.2005.02.055, doi:10.1016/j.entcs.2005.02.055.

de la Higuera, C., 2010. Grammatical Inference: Learning Automata and Grammars. Cambridge University Press.

DeMillo, R.A., Lipton, R.J., Sayward, F.G., 1978. Hints on test data selection: Help for the practicing programmer. Computer 11, 34–41. URL: https://doi.org/10.1109/C-M.1978.218136, doi:10.1109/C-M.1978.218136.

Di Penta, M., Lombardi, P., Taneja, K., Troiano, L., 2008. Search-based inference of dialect grammars. Soft Comput. 12, 51–66. URL: https://doi.org/10.1007/s00500-007-0216-5, doi:10.1007/s00500-007-0216-5.

Di Penta, M., Taneja, K., 2005. Towards the automatic evolution of reengineering tools, in: 9th European Conference on Software Maintenance and Reengineering (CSMR 2005), 21-23 March 2005, Manchester, UK, Proceedings, IEEE Computer Society. pp. 241–244. URL: https://doi.org/10.1109/CSMR.2005.52, doi:10.1109/CSMR.2005.52.

Drewes, F., Högberg, J., 2003. Learning a regular tree language from a teacher, in: Ésik, Z., Fülöp, Z. (Eds.), Developments in Language Theory, 7th International Conference, DLT 2003, Szeged, Hungary, July 7-11, 2003, Proceedings, Springer. pp. 279–291. URL: https://doi.org/10.1007/3-540-45007-6_22, doi:10.1007/3-540-45007-6_22.

Economopoulos, G., Klint, P., Vinju, J.J., 2009. Faster scannerless GLR parsing, in: de Moor, O., Schwartzbach, M.I. (Eds.), 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, Springer. pp. 126–141. URL: https://doi.org/10.1007/978-3-642-00722-4_10, doi:10.1007/978-3-642-00722-4_10.

Fischer, B., Lämmel, R., Zaytsev, V., 2011. Comparison of context-free grammars based on parsing generated test data, in: Sloane, A.M., Aßmann, U. (Eds.), Software Language Engineering - 4th International Conference, SLE 2011, Braga, Portugal, July 3-4, 2011, Revised Selected Papers, Springer. pp. 324–343. URL: https://doi.org/10.1007/978-3-642-28830-2_18, doi:10.1007/978-3-642-28830-2_18.

Forrest, S., Nguyen, T., Weimer, W., Le Goues, C., 2009. A genetic programming approach to automated software repair, in: Rothlauf, F. (Ed.), Genetic and Evolutionary Computation Conference, GECCO 2009, Proceedings, Montreal, Québec, Canada, July 8-12, 2009, ACM. pp. 947–954. URL: https://doi.org/10.1145/1569901.1570031, doi:10.1145/1569901.1570031.

Gazzola, L., Micucci, D., Mariani, L., 2019. Automatic software repair: A survey. IEEE Transactions on Software Engineering 45, 34–67. doi:10.1109/TSE.2017.2755013.

Ghanbari, A., Benton, S., Zhang, L., 2019. Practical program repair via bytecode mutation, in: Zhang, D., Møller, A. (Eds.), Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019, Beijing, China, July 15-19, 2019, ACM. pp. 19–30. URL: https://doi.org/10.1145/3293882.3330559, doi:10.1145/3293882.3330559.

Gold, E.M., 1967. Language identification in the limit. Inf. Control. 10, 447–474. URL: https://doi.org/10.1016/S0019-9958(67)91165-5, doi:10.1016/S0019-9958(67)91165-5.

Gopinath, R., Mathis, B., Zeller, A., 2020. Mining input grammars from dynamic control flow, in: Devanbu, P., Cohen, M.B., Zimmermann, T. (Eds.), 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, ACM. pp. 172–183. URL: https://doi.org/10.1145/3368089.3409679, doi:10.1145/3368089.3409679.

Havrikov, N., Zeller, A., 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. pp. 189–199. URL: https://doi.org/10.1109/ASE.2019.00027, doi:10.1109/ASE.2019.00027.

van Heerden, P., Raselimo, M., Sagonas, K., Fischer, B., 2020. Grammar-based testing for little languages: an experience report with student compilers, in: Lämmel, R., Tratt, L., de Lara, J. (Eds.), Proceedings of the 13th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2020, Virtual Event, USA, November 16-17, 2020, ACM. pp. 253–269. URL: https://doi.org/10.1145/3426425.3426946, doi:10.1145/3426425.3426946.

Holler, C., Herzig, K., Zeller, A., 2012. Fuzzing with code fragments, in: Kohno, T. (Ed.), Proceedings of the 21th USENIX Security Symposium, Bellevue, WA, USA, August 8-10, 2012, USENIX Association. pp. 445–458. URL: https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/holler.

Höschele, M., Zeller, A., 2016. Mining input grammars from dynamic taints, in: Lo, D., Apel, S., Khurshid, S. (Eds.), Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ACM, New York, NY, USA. pp. 720–725. URL: https://doi.org/10.1145/2970276.2970321, doi:10.1145/2970276.2970321.

Höschele, M., Zeller, A., 2017. Mining input grammars with AUTOGRAM, in: Uchitel, S., Orso, A., Robillard, M.P. (Eds.), Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017 - Companion Volume, IEEE Computer Society. pp. 31–34. URL: https://doi.org/10.1109/ICSE-C.2017.14, doi:10.1109/ICSE-C.2017.14.

Hua, J., Zhang, M., Wang, K., Khurshid, S., 2018. Towards practical program repair with on-demand candidate generation, in: Chaudron, M., Crnkovic, I., Chechik, M., Harman, M. (Eds.), Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018, ACM. pp. 12–23. URL: https://doi.org/10.1145/3180155.3180245, doi:10.1145/3180155.3180245.

Isberner, M., 2015. Foundations of active automata learning: an algorithmic perspective. Ph.D. thesis. Technical University Dortmund, Germany. URL: http://hdl.handle.net/2003/34282.

Jain, R., Aggarwal, S.K., Jalote, P., Biswas, S., 2004. An interactive method for extracting grammar from programs. Softw. Pract. Exp. 34, 433–447. URL: https://doi.org/10.1002/spe.568, doi:10.1002/spe.568.

Ji, T., Chen, L., Mao, X., Yi, X., 2016. Automated program repair by using similar code containing fix ingredients, in: 40th IEEE Annual Computer Software and Applications Conference, COMPSAC 2016, Atlanta, GA, USA, June 10-14, 2016, IEEE Computer Society. pp. 197–202. URL: https://doi.org/10.1109/COMPSAC.2016.69, doi:10.1109/COMPSAC.2016.69.

Jones, J.A., Harrold, M.J., 2005. Empirical evaluation of the tarantula automatic fault-localization technique, in: Redmiles, D.F., Ellman, T., Zisman, A. (Eds.), 20th IEEE/ACM International Conference on Automated Software Engineering (ASE 2005), November 7-11, 2005, Long Beach, CA, USA, ACM. pp. 273–282. URL: https://doi.org/10.1145/1101908.1101949, doi:10.1145/1101908.1101949.

Kaplan, A., Shoup, D., 2000. CUPV - a visualization tool for generated parsers, in: Cassel, L.B., Dale, N.B., Walker, H.M., Haller, S.M. (Eds.), Proceedings of the 31st SIGCSE Technical Symposium on Computer Science Education, SIGCSE 2000, Austin, Texas, USA, March 7-12, 2000, ACM. pp. 11–15. URL: https://doi.org/10.1145/330908.331801, doi:10.1145/330908.331801.

Ke, Y., Stolee, K.T., Le Goues, C., Brun, Y., 2015. Repairing programs with semantic code search (T), in: Cohen, M.B., Grunske, L., Whalen, M. (Eds.), 30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015, IEEE Computer Society. pp. 295–306. URL: https://doi.org/10.1109/ASE.2015.60, doi:10.1109/ASE.2015.60.

Kim, D., Nam, J., Song, J., Kim, S., 2013. Automatic patch generation learned from human-written patches, in: Notkin, D., Cheng, B.H.C., Pohl, K. (Eds.), 35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013, IEEE Computer Society. pp. 802–811. URL: https://doi.org/10.1109/ICSE.2013.6606626, doi:10.1109/ICSE.2013.6606626.

Knobe, B., Knobe, K., 1976. A method for inferring context-free grammars. Inf. Control. 31, 129–146. URL: https://doi.org/10.1016/S0019-9958(76)80003-4, doi:10.1016/S0019-9958(76)80003-4.

Koyuncu, A., Liu, K., Bissyandé, T.F., Kim, D., Klein, J., Monperrus, M., Traon, Y.L., 2020. Fixminer: Mining relevant fix patterns for automated program repair. Empir. Softw. Eng. 25, 1980–2024. URL: https://doi.org/10.1007/s10664-019-09780-z, doi:10.1007/s10664-019-09780-z.

Kulkarni, N., Lemieux, C., Sen, K., 2021. Learning highly recursive input grammars, in: 36th IEEE/ACM International Conference on Automated Software Engineering, ASE 2021, Melbourne, Australia, November 15-19, 2021, IEEE. pp. 456–467. URL: https://doi.org/10.1109/ASE51524.2021.9678879, doi:10.1109/ASE51524.2021.9678879.

Lämmel, R., 2001a. Grammar adaptation, in: Oliveira, J.N., Zave, P. (Eds.), FME 2001: Formal Methods for Increasing Software Productivity, International Symposium of Formal Methods Europe, Berlin, Germany, March 12-16, 2001, Proceedings, Springer. pp. 550–570. URL: https://doi.org/10.1007/3-540-45251-6_32, doi:10.1007/3-540-45251-6_32.

Lämmel, R., 2001b. Grammar testing, in: Hußmann, H. (Ed.), 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, Springer. pp. 201–216. URL: https://doi.org/10.1007/3-540-45314-8_15, doi:10.1007/3-540-45314-8_15.

Lämmel, R., Verhoef, C., 2001. Semi-automatic grammar recovery. Softw. Pract. Exp. 31, 1395–1438. URL: https://doi.org/10.1002/spe.423, doi:10.1002/spe.423.

Lämmel, R., Zaytsev, V., 2009a. An introduction to grammar convergence, in: Leuschel, M., Wehrheim, H. (Eds.), Integrated Formal Methods, 7th International Conference, IFM 2009, Düsseldorf, Germany, February 16-19, 2009. Proceedings, Springer. pp. 246–260. URL: https://doi.org/10.1007/978-3-642-00255-7_17, doi:10.1007/978-3-642-00255-7_17.

Lämmel, R., Zaytsev, V., 2009b. 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. pp. 178–186. URL: https://doi.org/10.1109/SCAM.2009.29, doi:10.1109/SCAM.2009.29.

Lankhorst, M.M., 1994. Grammatical inference with a genetic algorithm, in: Dekker, L., Smit, W., Zuidervaart, J.C. (Eds.), Massively Parallel Processing Applications and Developpent, Proceedings of the 1994 EUROSIM Conference on Massively Parallel Processing Applications and Developpent, 21-23 June 1994, Delft, The Netherlands, Elsevier. pp. 423–430.

Le, X.D., Chu, D., Lo, D., Le Goues, C., Visser, W., 2017. JFIX: semantics-based repair of java programs via symbolic pathfinder, in: Bultan, T., Sen, K. (Eds.), Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, Santa Barbara, CA, USA, July 10 - 14, 2017, ACM. pp. 376–379. URL: https://doi.org/10.1145/3092703.3098225, doi:10.1145/3092703.3098225.

Le Goues, C., Nguyen, T., Forrest, S., Weimer, W., 2012. Genprog: A generic method for automatic software repair. IEEE Trans. Software Eng. 38, 54–72. URL: https://doi.org/10.1109/TSE.2011.104, doi:10.1109/TSE.2011.104.

Lee, L., 1996. Learning of Context-Free Languages: A Survey of the Literature. Technical Report Computer Science Group Technical Report TR-12-96. Harvard University.

Liu, K., Koyuncu, A., Bissyandé, T.F., Kim, D., Klein, J., Traon, Y.L., 2019a. 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. pp. 102–113. URL: https://doi.org/10.1109/ICST.2019.00020, doi:10.1109/ICST.2019.00020.

Liu, K., Koyuncu, A., Kim, D., Bissyandé, T.F., 2019b. Tbar: revisiting template-based automated program repair, in: Zhang, D., Møller, A. (Eds.), Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019, Beijing, China, July 15-19, 2019, ACM. pp. 31–42. URL: https://doi.org/10.1145/3293882.3330577, doi:10.1145/3293882.3330577.

Liu, X., Zhong, H., 2018. Mining stackoverflow for program repair, in: Oliveto, R., Di Penta, M., Shepherd, D.C. (Eds.), 25th International Conference on Software Analysis, Evolution and Reengineering, SANER 2018, Campobasso, Italy, March 20-23, 2018, IEEE Computer Society. pp. 118–129. URL: https://doi.org/10.1109/SANER.2018.8330202, doi:10.1109/SANER.2018.8330202.

Madhavan, R., Mayer, M., Gulwani, S., Kuncak, V., 2015. Automating grammar comparison, in: Aldrich, J., Eugster, P. (Eds.), 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, ACM. pp. 183–200. URL: https://doi.org/10.1145/2814270.2814304, doi:10.1145/2814270.2814304.

Martinez, M., Monperrus, M., 2018. Ultra-large repair search space with automatically mined templates: The cardumen mode of astor, in: Colanzi, T.E., McMinn, P. (Eds.), Search-Based Software Engineering - 10th International Symposium, SSBSE 2018, Montpellier, France, September 8-9, 2018, Proceedings, Springer. pp. 65–86. URL: https://doi.org/10.1007/978-3-319-99241-9_3, doi:10.1007/978-3-319-99241-9_3.

Mathis, B., Gopinath, R., Mera, M., Kampmann, A., Höschele, M., Zeller, A., 2019. Parser-directed fuzzing, in: McKinley, K.S., Fisher, K. (Eds.), Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019, ACM. pp. 548–560. URL: https://doi.org/10.1145/3314221.3314651, doi:10.1145/3314221.3314651.

Monperrus, M., 2018. Automatic software repair: A bibliography. ACM Comput. Surv. 51, 17:1–17:24. URL: https://doi.org/10.1145/3105906, doi:10.1145/3105906.

Naish, L., Lee, H.J., Ramamohanarao, K., 2011. A model for spectra-based software diagnosis. ACM Trans. Softw. Eng. Methodol. 20, 11:1–11:32. URL: https://doi.org/10.1145/2000791.2000795, doi:10.1145/2000791.2000795.

Nakamura, K., 2006. Incremental learning of context free grammars by bridging rule generation and search for semi-optimum rule sets, in: Sakakibara, Y., Kobayashi, S., Sato, K., Nishino, T., Tomita, E. (Eds.), Grammatical Inference: Algorithms and Applications, 8th International Colloquium, ICGI 2006, Tokyo, Japan, September 20-22, 2006, Proceedings, Springer. pp. 72–83. URL: https://doi.org/10.1007/11872436_7, doi:10.1007/11872436_7.

Nakamura, K., Ishiwata, T., 2000. Synthesizing context free grammars from sample strings based on inductive CYK algorithm, in: Oliveira, A.L. (Ed.), Grammatical Inference: Algorithms and Applications, 5th International Colloquium, ICGI 2000, Lisbon, Portugal, September 11-13, 2000, Proceedings, Springer. pp. 186–195. URL: https://doi.org/10.1007/978-3-540-45257-7_15, doi:10.1007/978-3-540-45257-7_15.

Nguyen, H.D.T., Qi, D., Roychoudhury, A., Chandra, S., 2013. Semfix: program repair via semantic analysis, in: Notkin, D., Cheng, B.H.C., Pohl, K. (Eds.), 35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013, IEEE Computer Society. pp. 772–781. URL: https://doi.org/10.1109/ICSE.2013.6606623, doi:10.1109/ICSE.2013.6606623.

Ochiai, A., 1957. Zoogeographical studies on the soleoid fishes found in japan and its neighhouring regions-ii. Bulletin of the Japanese Society of Scientific Fisheries 22, 526–530. doi:10.2331/suisan.22.526.

Petasis, G., Paliouras, G., Spyropoulos, C.D., Halatsis, C., 2004. eg-grids: Context-free grammatical inference from positive examples using genetic search, in: Paliouras, G., Sakakibara, Y. (Eds.), Grammatical Inference: Algorithms and Applications, 7th International Colloquium, ICGI 2004, Athens, Greece, October 11-13, 2004, Proceedings, Springer. pp. 223–234. URL: https://doi.org/10.1007/978-3-540-30195-0_20, doi:10.1007/978-3-540-30195-0_20.

Qi, Y., Mao, X., Lei, Y., 2013. Efficient automated program repair through fault-recorded testing prioritization, in: 2013 IEEE International Conference on Software Maintenance, Eindhoven, The Netherlands, September 22-28, 2013, IEEE Computer Society. pp. 180–189. URL: https://doi.org/10.1109/ICSM.2013.29, doi:10.1109/ICSM.2013.29.

Qi, Y., Mao, X., Lei, Y., Dai, Z., Wang, C., 2014. The strength of random search on automated program repair, in: Jalote, P., Briand, L.C., van der Hoek, A. (Eds.), 36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014, ACM. pp. 254–265. URL: https://doi.org/10.1145/2568225.2568254, doi:10.1145/2568225.2568254.

Raselimo, M., Fischer, B., 2019. Spectrum-based fault localization for context-free grammars, in: Nierstrasz, O., Gray, J., d. S. Oliveira, B.C. (Eds.), Proceedings of the 12th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2019, Athens, Greece, October 20-22, 2019, ACM. pp. 15–28. URL: https://doi.org/10.1145/3357766.3359538, doi:10.1145/3357766.3359538.

Raselimo, M., Fischer, B., 2021. Automatic grammar repair, in: Visser, E., Kolovos, D.S., Söderberg, E. (Eds.), SLE '21: 14th ACM SIGPLAN International Conference on Software Language Engineering, Chicago, IL, USA, October 17 - 18, 2021, ACM. pp. 126–142. URL: https://doi.org/10.1145/3486608.3486910, doi:10.1145/3486608.3486910.

Raselimo, M., Fischer, B., 2023. Spectrum-based rule- and item-level localization of faults in context-free grammars. This volume.

Raselimo, M., Taljaard, J., Fischer, B., 2019. Breaking parsers: mutation-based generation of programs with guaranteed syntax errors, in: Nierstrasz, O., Gray, J., d. S. Oliveira, B.C. (Eds.), Proceedings of the 12th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2019, Athens, Greece, October 20-22, 2019, ACM. pp. 83–87. URL: https://doi.org/10.1145/3357766.3359542, doi:10.1145/3357766.3359542.

Renieris, M., Reiss, S.P., 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. pp. 30–39. URL: https://doi.org/10.1109/ASE.2003.1240292, doi:10.1109/ASE.2003.1240292.

Rossouw, C., Fischer, B., 2020. Test case generation from context-free grammars using generalized traversal of lr-automata, in: Lämmel, R., Tratt, L., de Lara, J. (Eds.), Proceedings of the 13th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2020, Virtual Event, USA, November 16-17, 2020, ACM. pp. 133–139. URL: https://doi.org/10.1145/3426425.3426938, doi:10.1145/3426425.3426938.

Rossouw, C., Fischer, B., 2021. Vision: bias in systematic grammar-based test suite construction algorithms, in: Visser, E., Kolovos, D.S., Söderberg, E. (Eds.), SLE '21: 14th ACM SIGPLAN International Conference on Software Language Engineering, Chicago, IL, USA, October 17 - 18, 2021, ACM. pp. 143–149. URL: https://doi.org/10.1145/3486608.3486902, doi:10.1145/3486608.3486902.

Roychoudhury, A., 2016. Semfix and beyond: semantic techniques for program repair, in: Naik, R., Medicherla, R.K., Banerjee, A. (Eds.), Proceedings of the International Workshop on Formal Methods for Analysis of Business Systems, ForMABS@ASE 2016, Singapore, Singapore, September 4, 2016, ACM. p. 2. URL: https://doi.org/10.1145/2975941.2990288, doi:10.1145/2975941.2990288.

Saha, D., Narula, V., 2011. Gramin: a system for incremental learning of programming language grammars, in: Bahulkar, A., Kesavasamy, K., Prabhakar, T.V., Shroff, G. (Eds.), Proceeding of the 4th Annual India Software Engineering Conference, ISEC 2011, Thiruvananthapuram, Kerala, India, February 24-27, 2011, ACM. pp. 185–194. URL: https://doi.org/10.1145/1953355.1953380, doi:10.1145/1953355.1953380.

Saha, R.K., Lyu, Y., Yoshida, H., Prasad, M.R., 2017. ELIXIR: effective object oriented program repair, in: Rosu, G., Di Penta, M., Nguyen, T.N. (Eds.), Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017, IEEE Computer Society. pp. 648–659. URL: https://doi.org/10.1109/ASE.2017.8115675, doi:10.1109/ASE.2017.8115675.

Sakakibara, Y., 1997. Recent advances of grammatical inference. Theor. Comput. Sci. 185, 15–45. URL: https://doi.org/10.1016/S0304-3975(97)00014-5, doi:10.1016/S0304-3975(97)00014-5.

Solomonoff, R.J., 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). pp. 285–289.

Stevenson, A., Cordy, J.R., 2014. A survey of grammatical inference in software engineering. Sci. Comput. Program. 96, 444–459. URL: https://doi.org/10.1016/j.scico.2014.05.008, doi:10.1016/j.scico.2014.05.008.

Veggalam, S., Rawat, S., Haller, I., Bos, H., 2016. Ifuzzer: An evolutionary interpreter fuzzer using genetic programming, in: Askoxylakis, I.G., Ioannidis, S., Katsikas, S.K., Meadows, C.A. (Eds.), Computer Security - ESORICS 2016 - 21st European Symposium on Research in Computer Security, Heraklion, Greece, September 26-30, 2016, Proceedings, Part I, Springer. pp. 581–601. URL: https://doi.org/10.1007/978-3-319-45744-4_29, doi:10.1007/978-3-319-45744-4_29.

Wang, J., Chen, B., Wei, L., Liu, Y., 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. pp. 579–594. URL: https://doi.org/10.1109/SP.2017.23, doi:10.1109/SP.2017.23.

Weimer, W., Forrest, S., Le Goues, C., Nguyen, T., 2010. Automatic program repair with evolutionary computation. Commun. ACM 53, 109–116. URL: https://doi.org/10.1145/1735223.1735249, doi:10.1145/1735223.1735249.

Weimer, W., Nguyen, T., Le Goues, C., Forrest, S., 2009. Automatically finding patches using genetic programming, in: 31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings, IEEE. pp. 364–374. URL: https://doi.org/10.1109/ICSE.2009.5070536, doi:10.1109/ICSE.2009.5070536.

Wong, W.E., Debroy, V., Gao, R., Li, Y., 2014. The dstar method for effective software fault localization. IEEE Trans. Reliab. 63, 290–308. URL: https://doi.org/10.1109/TR.2013.2285319, doi:10.1109/TR.2013.2285319.

Xue, X., Namin, A.S., 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. pp. 113–122. URL: https://doi.org/10.1109/ESEM.2013.22, doi:10.1109/ESEM.2013.22.

Zaytsev, V., 2009. Language convergence infrastructure, in: Fernandes, J.M., Lämmel, R., Visser, J., Saraiva, J. (Eds.), Generative and Transformational Techniques in Software Engineering III - International Summer School, GTTSE 2009, Braga, Portugal, July 6-11, 2009. Revised Papers, Springer. pp. 481–497. URL: https://doi.org/10.1007/978-3-642-18023-1_16, doi:10.1007/978-3-642-18023-1_16.

Zaytsev, V., 2010. Recovery, Convergence and Documentation of Languages.

Zaytsev, V., 2014. Negotiated grammar evolution. J. Object Technol. 13, 1: 1–22. URL: https://doi.org/10.5381/jot.2014.13.3.a1, doi:10.5381/jot.2014.13.3.a1.

Zelenov, S.V., Zelenova, S.A., 2005. Generation of positive and negative tests for parsers. Program. Comput. Softw. 31, 310–320. URL: https://doi.org/10.1007/s11086-005-0040-6, doi:10.1007/s11086-005-0040-6.