Fault Localization and Repair for Grammarware

by

Moeketsi Raselimo

Dissertation presented for the degree of Doctor of Philosophy (Computer Science) in the Faculty of Science at Stellenbosch University

Supervisor: Prof. Bernd Fischer

March 2023

The financial assistance of the National Research Foundation (NRF) under Grant 113364 towards this research is hereby acknowledged. Opinions expressed and conclusions arrived at, are those of the author and are not necessarily to be attributed to the NRF.

Declaration

By submitting this dissertation electronically, I declare that the entirety of the work contained therein is my own, original work, that I am the sole author thereof (save to the extent explicitly otherwise stated), that reproduction and publication thereof by Stellenbosch University will not infringe any third party rights and that I have not previously in its entirety or in part submitted it for obtaining any qualification.

Date: March 2023

Abstract

Fault Localization and Repair for Grammarware

M. Raselimo
Computer Science Division,
Department of Mathematical Sciences
Stellenbosch University,
Private Bag X1, Matieland 7602, South Africa.

Dissertation: PhD (CS)
March 2023

Context-free grammars (CFGs) or simply grammars, are widely used to describe and encode the structure of complex objects. Systems that process such structured objects are known as grammarware. CFGs are also used for software engineering tasks such as testing grammarware, specifically in test suite generation and fuzzing. However, correct and complete CFGs are rarely available, which limits the benefits of these automated techniques.

While current grammar engineering approaches can demonstrate the presence of faults in CFGs, none of them can be used to confirm the locations of these faults, much less repair them. In this thesis, we address this problem and develop, implement and evaluate novel automated methods that find locations of faults in CFGs and repair them against a test suite specification.

We describe and evaluate the first spectrum-based method aimed at finding faults CFGs. In its basic form, it takes as input a test suite and a modified parser for the grammar that can collect grammar spectra, i.e., the sets of grammar elements used in attempts to parse the individual test cases, and returns as output a ranked list of suspicious elements. We define grammar spectra suitable for localizing faults on the level of the grammar rules and at the rules' individual symbols, respectively. We show how these types of spectra can be collected in widely used parsing tools. We also show how the spectra can be constructed directly from test cases derived from a grammar, and how these synthetic spectra can be used for localization in cases where standalone parsers implementing the grammars are not available.

We qualitatively and quantitatively demonstrate the effectiveness of our fault localization approach. We first evaluate our method over a large number of medium-sized single fault CFGs, which we constructed by fault seeding from a common origin grammar. At the rule level, it ranks the rules containing the seeded faults within the top five rules in about 40%–70% of the cases, and pinpoints them (i.e., correctly identifies them as unique most suspicious rule) in about 10%–30% of the cases, with significantly better results for the synthetic spectra. On average, it ranks the faulty rules within about 25% of all rules. At the item level, our method remains remarkably effective despite the larger number of possible locations: it typically ranks the seeded faults within the top five positions in about 30%–60% of the cases, and pinpoints them in about 15%–40% of the cases. On average, it ranks the seeded faults within about 10%–20% of all positions.

We further evaluate our method over CFGs that contain real faults. We show that an iterative approach can be used to localize and manually remove one by one multiple faults in grammars submitted by students enrolled in various compiler engineering courses; in most iterations, the top-ranked rule already contains an error, and no error is ranked outside the top five ranked rules. We finally apply our method to a large open-source SQLite grammar and show where the original version deviates from the language accepted by the actual SQLite system.

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

We implement and evaluate passive and active repair variants of our approach. In passive repair, we repair against the fixed input test suite as specification. The active repair variant takes a black-box parser for the unknown target language or oracle that can answer membership queries. The key extension of active repair is to incorporate some test suite enrichment, by generating additional tests from each repair candidate and using 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 faults. We show that both variants are effective in fixing real faults in these grammars. A like-for-like comparison of both variants shows that active repair produces more high quality repairs than passive repair.

Uittreksel

Foutlokalisering en Herstel vir Grammatika

("Foutlokalisering en Herstel vir Grammatikar")

M. Raselimo

Afdeling van Rekenaarwetenskap,
Universiteit van Stellenbosch,
Privaatsak X1, Matieland 7602, Suid Afrika.

Proefskrif: PhD (RW)

Maart 2023

Konteksvrye grammatikas (CFG's) of bloot grammatikas, is wyd gebruik om die struktuur van komplekse voorwerpe te beskryf en te kodeer. Stelsels dat sulke gestruktureerde voorwerpe verwerk word, staan bekend as grammatika. CFG's word ook gebruik vir sagteware-ingenieurstake soos toetsing grammatika, spesifiek In sagteware-ingenieurswese word CFG's spesifiek gebruik in die toets van grammatika in toetssuite generering en fuzzing. Korrekte en volledige CFG's is egter selde beskikbaar, wat die voordele van hierdie outomatiese tegnieke beperk. Dit is grootliks omdat ingenieurspraktyke soos toetsing

Terwyl huidige grammatika-ingenieursbenaderings die teenwoordigheid van kan demonstreer foute in CFG's, óf deur statiese analise tegnieke óf dinamies nie een van hulle kan gebruik word om die liggings van te bevestig nie hierdie foute, nog minder herstel hulle. Die lokalisering en In hierdie tesis spreek ons hierdie probleem aan en ontwikkel, implementeer en evalueer nuwe geoutomatiseerde metodes wat vind liggings van foute in CFG's en herstel dit teen 'n toetssuite-spesifikasie.

Ons beskryf en evalueer die eerste spektrum-gebaseerde metode wat daarop gemik is om te vind foute CFG's. In sy basiese vorm neem dit as inset 'n toetsreeks en 'n gewysigde ontleder vir die grammatika wat grammatikaspektra kan versamel, dit wil sê die stelle grammatika-elemente wat in poog om die individuele toetsgevalle te ontleed, en gee as uitvoer 'n gerangorde lys van verdagte elemente. Ons definieer grammatikaspektra wat geskik is vir die lokalisering van foute op die vlak van die grammatika reëls en by die reëls se individuele simbole, onderskeidelik. Ons wys hoe hierdie tipe spektra kan versamel word in wyd gebruikte ontledingsinstrumente. Ons wys ook hoe die spektra gekonstrueer kan word direk van toetsgevalle afgelei van 'n grammatika, en hoe hierdie sintetiese spektra kan gebruik word vir lokalisering in gevalle waar selfstandige ontleders wat die grammatika is nie beskikbaar nie.

Ons demonstreer kwalitatief en kwantitatief die effektiwiteit van ons foutlokaliseringsbenadering. Ons evalueer eers ons metode oor 'n groot aantal mediumgrootte enkelfout-CFG's, wat ons gekonstrueer het deur foutsaaiing uit 'n algemene oorsprong-grammatika. Op reëlvlak rangskik dit die reëls wat die gesaaide foute bevat binne die top vyf reëls in ongeveer 40%–70% van die gevalle, en identifiseer hulle (d.w.s. identifiseer hulle korrek as die unieke mees verdagte reël) in ongeveer 10%–30% van die gevalle, met aansienlik beter resultate vir die sintetiese spektra. Gemiddeld rangskik dit die foutiewe reëls binne ongeveer 25% van alle reëls. Op itemvlak bly ons metode merkwaardig effektief ten spyte van die groter aantal moontlike liggings: dit rangskik tipies die gesaaide foute binne die top vyf posisies in ongeveer 30%–60% van die gevalle, en identifiseer hulle in ongeveer 15%–40% van die gevalle. Dit rangskik die gesaaide foute gemiddeld binne ongeveer 10%–20% van alle posisies.

Ons evalueer verder ons metode oor CFG's wat werklike foute bevat. Ons wys dat 'n iteratiewe benadering gebruik kan word om een vir een veelvuldige foute in grammatikas wat ingedien is deur studente wat in verskeie samestelleringenieurswese-kursusse ingeskryf is te lokaliseer en handmatig te verwyder; in die meeste iterasies bevat die reël wat die hoogste gerangskik reeds 'n fout, en geen fout word buite die top vyf reëls gerangskik nie. Ons pas uiteindelik ons metode toe op 'n groot oopbron SQLite-grammatika en wys waar die oorspronklike weergawe afwyk van die taal wat deur die werklike SQLite-stelsel aanvaar word.

Ons beskryf dan die eerste benadering om outomaties te herstel foute in CFG's: gegewe 'n grammatika wat sommige toetse in 'n gegewe toetsreeks druip, ons transformeer die grammatika iteratief en geleidelik totdat dit alle toetse slaag. Ons kern idee is om voort te bou op spektrum-gebaseerde fout lokalisering te identifiseer belowende herstelpersele (d.w.s. spesifieke posisies in reëls), en om aansoek te doen grammatika kolle op hierdie webwerwe wanneer hulle bevredig eksplisiet geformuleer voorwaardes wat nodig is om die grammatika moontlik te verbeter.

Ons implementeer en evalueer passiewe en aktiewe herstel variante van ons benadering. In passiewe herstel herstel ons teen die vaste invoertoetssuite as spesifikasie. Die aktiewe herstelvariant neem 'n swartboks-ontleder vir die onbekende teiken taal of orakel wat lidmaatskapnavrae kan beantwoord. Die sleutel uitbreiding van aktiewe herstel is om te inkorporeer 'n mate van toetsreeksverryking, deur bykomende toetse uit elkeen te genereer herstelkandidaat en die gebruik van die orakel om die uitslag van elk van hierdie toetse te bevestig.

Ons demonstreer die doeltreffendheid van beide herstel variante met behulp van drie-en-dertig studente grammatika wat veelvuldige foute bevat. Ons wys dat beide variante effektief is om vas te maak werklike foute in hierdie grammatika. ’n Soortgelyke vergelyking van beide variante toon dat aktiewe herstel meer herstelwerk van hoë gehalte lewer as passiewe herstel.

Acknowledgements

First, I would like to thank my supervisor, Prof. Bernd Fischer, for his guidance that made this research possible.

Second, I thank my family for all the support: my mother, Mafelleng Raselimo and sisters, Felleng Raselimo, Lineo Raselimo, Liako Tŝenoli, and Mookho Raselimo.

Third, a special note of gratitude to Prof. Eric Van Wyk from the University of Minnesota, Dr. Vadim Zaytsev from the University of Twente, and Dr. Cornelia Inggs, for examining this thesis.

Moreover, to my labmates, "professor" Dylan Callaghan, "prince" Lucas Roos, Kevin Brand, and Proscovia Nakiranda: thanks for all the fruitful discussions, and the air-conditioning fights.

Lastly, a shout-out to all my friends who were a huge part of my PhD journey. The following deserve a special mention: Lintle Semoli, 'Nyane Makara, Phuthehang Maphatŝoe, Phillip van Heerden, Imraan Badrodien, Bohlajana Qacha, Mabatho Fooko, Ramafothole Mothobi, Relebohile Mathaba, Moabi Mokhoro, Tŝolo Lesofe, Langa Horoto, Teboho Mochai, and Thato Mokhothu.

Dedications

For my mother, ′Mafelleng Raselimo

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Grammarware [76] denotes software (including the grammars themselves) that is intrinsically linked to an underlying formal grammar. Archetypical examples are compilers, but grammarware is prevalent: think of JSON or XML preprocessors, email systems, audio and video streaming software, theorem provers, network protocol analysers, and many more. More recently, we see the usage of grammars in the field of software testing; fuzzing tools leverage grammars to encode complex inputs of their target *systems under test* (SUT) [15, 141, 151, 152, 153, 163, 164, 164]. This allows these grammar-based fuzzers to expose bugs and security vulnerabilities that lie in deeper stages beyond the syntax analysis stage. However, correct and complete grammars are rarely available. This can in part be attributed to the little attention their development gets compared to "traditional" software engineering. Lämmel [82] and Klint *et al.* [76] explicitly address this issue by bridging the theoretical and engineering gaps and develop techniques that treat grammars as proper software artefacts.

In this work, we take this view of regarding grammars as proper software artefacts further. Like any other software, grammars can contain bugs, and testing remains the mostly used method to demonstrate the presence of bugs [40] but testing does not directly give any further information about their *location*, much less about possible *repairs* and can thus not automatically *fix* grammars. Grammar bugs are prevalent, even well-curated grammars are no exceptions to bugs. For example, Lämmel and Verhoef [84] found "*more errors than one would expect from a language reference manual*" when analysing COBOL and Zaytsev [169] shows errors and inconsistencies in language specifications of both Java [86, 87] and C#.

Traditionally, grammar testing (or grammar-based testing) has focused almost exclusively on automatic sentence generation for testing parsers and for debugging grammars themselves [124], i.e., finding faults in a grammar which cause it to define some language other than the one intended to. Starting with Purdom [124]'s seminal work on sentence generation from grammars, many approaches and algorithms have been proposed that satisfy different grammar coverage criteria and generate test suites with certain characteristics [82, 83, 130, 131, 170]. Sentence generation methods typically work under the assumption that the generative grammars are correct. However, when under- or over-approximations of the intended true language are detected, the grammar needs to be debugged. This debugging process, which involves fault-finding, fault understanding, and fault fixing, typically is a primarily manual, labour-intensive and often knowledge-intensive process.

There is some work that can under certain restrictions support manual debugging effects. Lämmel and Zaytsev [85] propose grammar convergence; a procedure that takes any two input grammars and systematically modifies them to become structurally equivalent. Other approaches exploit generated test suites to compare pairs of input grammars. Fischer *et al.* [43] propose an approach that leverages generated test data and returns as output matching non-terminals from both input grammars, while Madhavan *et al.* [99] find counter-examples that prove non-equivalence. However, none of these approaches can be used to debug grammars fully automatically and all require human intervention.

In this thesis, we propose two novel approaches, *fault localization* and *automatic repair* for grammars, that aid grammar developers in their debugging tasks with little to no manual intervention. Our fault localization approach for grammars borrows ideas from software fault localization techniques to automatically identify faulty rules in the grammar. Software fault localization techniques [32, 158] build on testing and try to identify likely bug locations. *Spectrum-based fault localization* (SFL) methods [6, 26, 69, 70, 109, 115, 157] execute the SUT over a given test suite and record coverage information for the SUT's individual program elements. Most SFL methods use *binary statement coverage*, i.e., record whether a statement has been executed or not. A representation of this recorded coverage information is called a *program spectrum*. From the spectrum, SFL methods leverage some ranking formulas and compute a *suspiciousness score* for each program element. These ranking metrics differ in the number of input parameters and thus in their score computation, but higher scores typically indicate higher bug likelihood. We apply SFL methods to grammars with minimal changes. The automatic grammar repair builds on fault localization to identify promising *repair sites* and applies grammar transformations at these sites whenever they satisfy explicitly formulated preconditions that are necessary to potentially fix the grammar.

Automated grammar debugging approaches proposed in this work do not only directly extend the limits of grammar testing, but also enable more interesting application scenarios in various areas, for example, (i) *teaching*: our fault localization approach has already been integrated into an automated interactive feedback system [16] for grammar development at Stellenbosch University; (ii) *grammar learning*: our solution can substitute the blind search in the inner loop of genetic learning approaches [30, 149] and potentially speed up the search; (iii) *grammar migration*: our repair approach can be used to automatically fix faults that are introduced when a grammar is migrated from one formalism into another one with different capabilities, e.g., substitution of LL(1) based parser by a more capable PEG parser [46] in Python 3.9.0; (iv) *grammar recovery and maintenance*: to ensure backward compatibility, extension of a base grammar to capture a dialect from examples [149] can benefit from our automation.

1.1 Problem Statement

Correct and complete grammars are rarely available because they are often overlooked as proper software artefacts [76, 82]. This is still in fact true despite the growing use of grammars in popular software testing fields such as fuzzing. Lämmel [82] refers to the state of practice of grammar development as "grammar hacking" where proper engineering activities such as testing play a minor role. Therefore, grammar validation tasks such as fault localization and repair must be done manually and remain time-consuming and knowledge-intensive.

To illustrate some issues faced in grammar hacking, consider a situation where we are trying to develop a CUP [2] [1] grammar specification against a small positive test suite TS_{Toy} to complement an informal description of the target language *Toy*. Assume that we currently have the grammar G_{Toy} shown in Figure 1.1.

Assume further we are faced with the following five failing tests in TS_{Toy}

```
program ab = { ab = ab(21); }
program ab = { ab = ab(21, 21); }
program ab = { ab = ab(21, 21, 21); }
program ab = { ab = ab((21)); }
program ab = { ab = ab((21), (21)); }
```

In all five cases, CUP's syntax error messages are not that useful. They only confirm the error location and token, but give no further information. For example, the following is a syntax error message for the first three cases.

```
Error in line 1, column 24:  Syntax error.
Found NUM(21), expected token classes are [].
```

We now need to sift through these syntax error messages and trace the failing tests back to our grammar, to identify the faulty rules and then precise fault positions within these rules. In this case case, it is relatively straightforward because all failing test cases fail right after the token sequence

[1]CUP is a popular parser generator for Java.

$$\begin{array}{lll}
\mathit{prog} & \rightarrow & \texttt{\textbf{program}}\ \texttt{id}\ \texttt{=}\ \mathit{block}\ \texttt{.} \\
\mathit{block} & \rightarrow & \texttt{\{}\ \mathit{decls}\ \mathit{stmts}\ \texttt{\}} \mid \texttt{\{}\ \mathit{decls}\ \texttt{\}} \mid \texttt{\{}\ \mathit{stmts}\ \texttt{\}} \mid \texttt{\{}\ \texttt{\}} \\
\mathit{decls} & \rightarrow & \mathit{decl}\ \texttt{;}\ \mathit{decls} \mid \mathit{decl}\ \texttt{;} \\
\mathit{decl} & \rightarrow & \texttt{\textbf{var}}\ \texttt{id}\ \texttt{:}\ \mathit{type} \\
\mathit{type} & \rightarrow & \texttt{\textbf{bool}} \mid \texttt{\textbf{int}} \\
\mathit{stmts} & \rightarrow & \mathit{stmt}\ \texttt{;}\ \mathit{stmts} \mid \mathit{stmt}\ \texttt{;} \\
\mathit{stmt} & \rightarrow & \texttt{\textbf{sleep}} \\
 & \mid & \texttt{\textbf{if}}\ \mathit{expr}\ \texttt{\textbf{then}}\ \mathit{stmt} \\
 & \mid & \texttt{\textbf{if}}\ \mathit{expr}\ \texttt{\textbf{then}}\ \mathit{stmt}\ \texttt{\textbf{else}}\ \mathit{stmt} \\
 & \mid & \texttt{\textbf{while}}\ \mathit{expr}\ \texttt{\textbf{do}}\ \mathit{stmt} \\
 & \mid & \texttt{id}\ \texttt{=}\ \mathit{expr} \\
 & \mid & \mathit{block} \\
\mathit{expr} & \rightarrow & \mathit{expr}\ \texttt{=}\ \mathit{expr} \mid \mathit{expr}\ \texttt{+}\ \mathit{expr} \mid \texttt{id}\ \texttt{(}\ \texttt{id}\ \mathit{namelist}\ \texttt{)} \mid \texttt{(}\ \mathit{expr}\ \texttt{)} \mid \texttt{id} \mid \texttt{num} \\
\mathit{namelist} & \rightarrow & \texttt{,}\ \texttt{id}\ \mathit{namelist} \mid \varepsilon
\end{array}$$

Figure 1.1: An example grammar G_{Toy} in BNF format.

"`ab (`", and there is only one rule in which this sequence can occur, i.e.,

$$\mathit{expr} \rightarrow \texttt{id}\ \texttt{(}\ \bullet\ \texttt{id}\ \mathit{namelist}\ \texttt{)}$$

Note that we use the •-symbol to indicate the suspected fault position, i.e., the error is at the second `id` on the right-hand side of the *expr* rule.

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. The basic idea is to try to modify the grammar, so it consumes the *bad tokens* {**`0`**, **`(`** } (on column 24) all failing tests. We are faced with another challenge here; we have to identify a symbol whose first set includes both bad tokens, and we decide to substitute `id` with *expr* because derivations through *expr* can start with both `num` and **`(`** (see Section 2.1 for formal definitions).

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 three failing test cases:

```
program ab = { ab = ab(21, 21); }
program ab = { ab = ab(21, 21, 21); }
program ab = { ab = ab((21), (21)); }
```

In the first two 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 tests as well:

```
Error in line 1, column 28:  Syntax error.
Found NUM(21), expected token classes are [].
```

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

$$\textit{namelist} \to \texttt{,} \bullet \texttt{id}\ \textit{namelist} \mid \varepsilon$$

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

The example above illustrates that manual debugging efforts are tedious. The parser does not help much, since it assumes the grammar is correct and the input is wrong. Its syntax error messages are thus not necessarily useful and can sometimes be very complicated, e.g., convoluted due to the cascading error problem as a result of error recovery attempts by the parser. In order to tackle these issues, we need approaches that automate fault localization and subsequent repair attempts. Such automated debugging approaches have some benefits:

- They increase developer productivity, since they reduce the need for the manual debugging of grammars.
- They increase the usability of grammar-based testing methods, since they increase the availability of appropriate grammars.
- They leverage human insight, since they take advantage of approximate grammars written by developers, and automatically improve them.

1.2 Research Objectives

The overarching goal of this research is to automatically find and repair bugs in grammarware. More specifically, we aim to develop and evaluate an approach to lead a grammar developer to the location of the bug in a context–free grammar using spectrum-based fault localization (SFL) techniques and to develop and evaluate an approach to automatically repair the localized grammar bugs. While there are different fault localization methods, we have identified SFL techniques as the most suitable candidate because they can be easily adapted to our domain. Our research has the following specific objectives.

1. To extend the framework of spectrum-based fault localization to context-free grammars.
2. To extend common parser generator tools to automate grammar spectra extraction.

3. To evaluate the efficacy of the fault localizer against different kinds of faults, in particular
    - artificial faults introduced by mutation of the rules of the grammar, i.e, grammars with single and well-defined faults;
    - faults on grammars written by students; and
    - faults in real-world production grammars, i.e., grammars with unknown and multiple faults.
4. To investigate, develop and evaluate different grammar repair operations, specifically, to develop grammar refactoring operations such as splitting and joining non-terminal symbols, deletion of rule alternatives, pulling up and pushing down of non-terminals.

1.3 The Proposed Solution

In this section, we provide context for our realization of both spectrum-based fault localization and automatic repair for grammars approaches. We describe how both approaches were developed, and we empirically evaluate them under different fault models and different test suites.

1.3.1 Spectrum-based Fault Localization for Grammars

In Chapters 3 and 4, we describe our adaptation of the spectrum-based fault localization techniques in the context of grammars. Our key insight is that the SFL framework applies with minimal changes. We only need to replace the notion of executed program elements by grammar elements (i.e., rules and items) involved in the derivation of a word w. We define grammar spectrum at two levels of granularity, *rule spectra* and the fined-grained *item spectra*.

Rule-level Localization

In Chapter 3, we first describe and evaluate localization at the level of grammar rules. We view a rule to possibly contain a fault if it is used in a derivation of a word w that is successfully consumed by the parser that implements the grammar but is outside the true language (which may be described by a different grammar), or conversely, if the parser rejects w but it is within the true language. We therefore formally fix the notion of rule spectra as the representation and summary of the grammar rules which have been (partially) used in an attempt to parse inputs. We describe rule spectra extraction for both LL and LR parsers. Extraction attempts in LR parsers are, however, not straightforward, especially in the identification of partially applied rules in cases where an LR parser detects syntax violations and thus fails to perform full reductions that mark a successful rule application. In such cases, we analyze the parse stack and recover partially applied rules from corresponding states left on the parse stack at the time of syntax error.

In practice, the extraction of grammar spectra requires runtime support from the implementing parser, which maintains a full record of applied rules. Parsers generated by ANTLR [118] already provide this support through some extensions, but parsers generated by CUP [2] or JavaCC [3] do not. We therefore extend CUP and JavaCC sources to generate parsers with the spectral logging support.

We also introduce a "flipped" version of rule level localization; we construct *synthetic* grammar spectra directly from the test cases generated from the grammar. This is useful in the case of black-box parsers, which cannot be easily instrumented to log information for SFL. We use these synthetic grammar spectra to localize deviations between the grammar and the language accepted by the SUT.

Item-level Localization

In Chapter 4, we introduce a refinement of rule localization that allows us to localize faults more precisely, at the level of individual grammar symbols in the corresponding rule. The main difference here is that we define and evaluate spectra over *items*, i.e., rules with designated positions. We show that this method of spectrum collection improves over rule-level localization. We exploit the fact that the designated position marks the boundary between the part of a rule that has been successfully processed and the part that awaits further processing; hence, we can assume that the fault is at the symbol at the right of the designated position.

We collect item spectra for both LL and LR parsers; in both cases, the logging is carried out on shift operations implicitly (in LL) or explicitly (in LR). In the LL case, the logger captures the corresponding item when the parser successfully consumes a token or returns from a function call that implements the non-terminal in the rule. In LR cases, we introduce two slightly different approaches. The first approach can be thought of as expanding of the rule spectra: all items of successfully reduced rules are collected; on encountering a syntax error, we extract and add all items from the states left on the parse stack. In the second approach, we only target the shift operation and add all items associated with a state whenever that state is pushed onto the parse stack. We call these *shift item spectra*. This approach yields larger spectra than the first approach, but this apparent loss of precision in the spectrum collection does not necessarily translate into a worse localization performance, because the metrics are based on the spectral differences and compute a quotient between passing and failing counts.

1.3.2 Automatic Grammar Repair

In Chapter 5, we introduce our second main contribution, a fully automated grammar debugging procedure. Our main goal here is to automate the manual find-and-fix cycle that we illustrated earlier. We describe an iterative generate-localize-transform approach that induces patches that sufficiently approximate the intended target language. This approach constructs a repaired grammar G' from an initial user-provided test suite TS and faulty grammar G, optionally generates test suites TS_i from each candidate variant of G_i, evaluates the generated tests via a teacher (or *oracle*), uses the results to identify repair sites, and finally applies transformations to these sites. We evaluate two configurations of our repair approach: the *passive repair* approach that we described in our paper [129], fixes the faulty input grammar G over a fixed test suite; and an *active repair* approach which uses a parser of the unknown target language that can answer membership queries and serves as the oracle in the sense of Angluin's [11] query model. The key extension is that we introduce a test suite enrichment, where we judiciously generate (positive and negative) tests from repair candidates and use the oracle to obtain the expected outcome.

Our repair approach is informed by two basic principles: the *competent programmer hypothesis* [36] ("most programmers are competent enough to create correct or almost correct source code") 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 is reflected by the fact that the repair uses the vocabulary and the structure of the original grammar, and minimizes the number of applied patches.

1.4 Research Questions

We evaluate our approaches under a number of different scenarios, including the use of different parsing techniques, test suites, and ranking metrics. Overall, we are trying to answer three main research questions, which we further break down into more specific sub-questions that we answer by a series of different experiments.

RQ1: How effective is our spectrum-based fault localization approach in finding faults in grammars?

In the first set of experiments, we evaluate our method for rule-level localization over a large number of medium-sized single fault grammars, which are constructed from a common grammar by fault seeding. Note that fault seeding is widely used in SFL evaluation (e.g., [5, 156]) because it produces a large number of faulty subjects with known error locations. This is the largest set of experiments in our evaluation.

First, as a baseline, we investigate the effectiveness of rule-level fault localization. We analyze in detail whether the different applied parsing techniques, test suites, and ranking metrics have an effect on the effectiveness of the technique.

RQ1a How effective are fault localization techniques based on rule spectra in identifying seeded single faults in grammars?

Then, we investigate the effectiveness of synthetic spectra.

RQ1b How effective are fault localization techniques based on synthetic spectra in identifying seeded single faults?

In the second set of experiments, we look at grammars submitted by students enrolled in compiler engineering courses. Such grammars contain *real* and often *multiple* faults; however, SFL is based on a single fault assumption, and SFL methods can be misled by interactions between multiple faults [7, 162]. We investigate whether this is the case here.

RQ1c How effective are fault localization techniques in identifying *real* faults in grammars that possibly contain multiple faults?

In the final experiment of rule-level evaluation, we address the *scalability* of our approach by applying it to a large, production–quality grammar, specifically the ANTLR4 SQlite grammar.

RQ1d Can our approach remain effective for large grammars?

RQ2: Does the use of item spectra improve the localization accuracy?

Next, we evaluate whether our method remains effective when we switch to item spectra for a more precise localization of faults at the level of individual symbols, and whether the switch from rule spectra does indeed improve its accuracy.

RQ2a How effective are fault localization techniques based on item spectra in identifying seeded single faults in grammars at the level of individual symbols?

RQ2b Does the use of item spectra improve the localization accuracy?

RQ3: Can we use fault localization to drive automatic repair of faults in grammars?

We build on fault localization to identify repair sites and evaluate our approach with grammars that contain real and multiple faults. We first evaluate the efficacy and effectiveness of our passive repair approach.

RQ3a How effective is our proposed passive repair approach in fixing faults in grammars?

We then evaluate the efficacy and effectiveness of our active repair approach, which takes a boolean valued oracle $\mathcal{O}$ which answers *membership queries* to test cases generated from each candidate patch.

RQ3b How effective is our proposed active repair approach in fixing faults in grammars?

Next, we do a like-for-like comparison of the passive repair approach that we described in our SLE paper [129] and the active repair approach.

RQ3c Does the active repair approach induce better fixes than the passive repair approach?

1.5 Summary of Results

Spectrum-based fault localization. We qualitatively and quantitatively evaluate our method under a number of different fault models and scenarios, including the use of different parsing techniques, test suites, and ranking metrics. Our method typically ranks the rules containing the seeded faults within the top five grammar rules for about 40%–70% of the grammars, depending on the applied parsing technique, test suite, and ranking metric. It pinpoints them (i.e., correctly identifies them as unique most suspicious rule) in about 10%–30% of the cases. On average, it ranks the faulty rules within about 25% of all rules, and in less than 15% for a very large test suite containing both positive and negative test cases. Our method pinpoints far fewer of the seeded faults down to the exact symbol position, or even ranks them within the top five positions, due to the larger number of possible locations and corresponding larger ties (i.e., groups of equally suspicious locations). However, a simple tie-breaking strategy that prefers the right-most position amongst the rules in a tie proves remarkably effective: it typically ranks the seeded faults within the top five positions in about 30%–60% of the cases, and pinpoints them in about 15%–40% of the cases. On average, it ranks the seeded faults within about 10%–20% of all positions. The specialized symbol-level localization also significantly outperforms a simplistic extension of the rule-level localization, where all positions within a rule are given the same score.

Second, we look at grammars submitted by students enrolled in compiler engineering courses. Such grammars contain *real* and often *multiple* faults; however, SFL is based on a single fault assumption, and SFL methods can be misled by interactions between multiple faults [7, 162]. Our method remains effective in this more difficult situation. We use an iterative "one-bug-at-a-time" approach originally proposed by Jones et al. [70] in the context of fault localization in programs to localize and manually remove multiple faults one by one; in most iterations, the top-ranked rule already contains a fault, and no fault is ranked outside the top five ranked rules.

Finally, we address the *scalabilty* of our method, and use it to identify four locations in an open-source SQLite grammar [74] where it deviates from the language accepted by the actual SQLite system [4]. Here, we construct synthetic grammar spectra directly from the test cases derived from the grammar, and use the SQLite system as a black box to collect the required pass/fail information.

Automatic Grammar Repair. We have successfully used gfixr to repair 33 grammars that contain multiple, real faults. Passive repair 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 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 less prone to the over-generalization problem. The active algorithm 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 100% F1 score (i.e., combined measure of quality calculated as the harmonic mean of the corresponding recall and precision) in eight grammars and none in the passive case.

1.6 Benefits

Our approaches work at a higher level of abstraction than generic SFL and automatic program repair (APR) approaches. For fault localization, we get results in domain-specific terms (i.e., rules or symbols within these rules), instead of parser statements. This offers several advantages. First, it simplifies any subsequent repair attempts – grammar writers can directly use our results and do not need to manually trace back from the parser's implementation to the grammar's rules. Second, it increases the localization precision because it discards all aspects of the parser's internal bookkeeping and error handling code that could impact the localization process if generic program spectra were used. Third, it can also be meaningfully applied when the parser uses a table-driven implementation and there is no direct representation of the individual rules as executable code; this is typically the case for LR parsers.

Finally, for grammar repair, we can, in principle, use APR tools on the parser code that implement 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.

1.7 Contributions

The main contribution of this PhD thesis is the development of two automated software language validation tasks that were previously to be performed manually.

Fault Localization. We present the first method to localize faults in a context-free grammar, both at the level of rules and at the level of individual symbols within the rules. We then describe how common parsing generator tools can be extended to provide useful information necessary for fault localization. In applications where standalone parsers cannot be extended or instrumented to extract such information, we show how we can exploit test cases generated from a grammar to derive the required logging for fault localization. We demonstrate the effectiveness of this method over grammars with seeded faults, as well as grammars from student submissions from compiler engineering courses that contain real and multiple faults. We also show that our method remains effective even for larger production-quality grammars.

Grammar Repair. The second main contribution is the proposal of the first method aimed at fixing faults in context-free grammars. At its core, this method involves the following steps:

- *localization*: we build our fault localization method to identify promising repair sites (i.e., specific positions in rules);

- *transformation*: we apply small-scale grammar transformations or *patches* at these sites whenever they satisfy explicitly formulated pre-conditions (see Section 5.2 to Section 5.4 for details) that are necessary to potentially improve the grammar;
- *control*: we alternate between localization and transformation, as they reinforce each other and iterate until we find a fix. We use a priority queue to keep improving the most promising candidate grammars.

We detail ingredients necessary to realize this automatic repair method. We describe two repair variants, the passive repair variant in which we repair against a static test suite specification and the active repair variant, which exploits a membership oracle and test suite enrichment where we generate test suites from each generated repair candidate and use the oracle to provide the expected outcome. We demonstrate the effectiveness of both settings over grammars with real faults. We finally compare the two repair configurations.

1.8 Structure of the Thesis

In Chapter 2, we fix the basic grammar notations that we use in the thesis and give necessary background for different parsing mechanisms and spectrum-based fault localization. We describe and evaluate our fault localization method that works at the granularity of grammar rules in Chapter 3. The item-level fault localization method's description and evaluation follow in Chapter 4. Chapter 5 addresses the grammar repair problem. We describe two repair algorithms and our realization of the repair framework in the gfixr tool. We then evaluate the two algorithms and discuss current challenges and limitations of the approach. We discuss related work in Chapter 6. We conclude and give plans for future work in Chapter 7.

Declaration of Joint Work

The main contributions of this thesis were partially reported in the following publications:

1. **Moeketsi Raselimo** and Bernd Fischer. 2019. Spectrum-Based Fault Localization for Context-Free Grammars. In Proceedings of the 12th ACM SIGPLAN International Conference on Software Language Engineering (SLE '19), October 20 – 22, 2019, Athens, Greece. ACM, New York, NY, USA, 14 pages. `https://doi.org/10.1145/3357766.3359538`.

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

The investigation, development, and evaluation of the concepts in these papers were done under close supervision of Bernd Fischer. However, this thesis describes further substantial extensions of the work reported in these papers, namely the item-level localization and the active repair approach, that are as yet unpublished.

We adapt concepts and other preliminary materials described in Chapter 2 from papers co-authored with other collaborators. The definition of negative test suite construction algorithms in Section 2.3.3 was done in collaboration with Jan Taljaard and Bernd Fischer in our paper

- **Moeketsi Raselimo**, Jan Taljaard, and Bernd Fischer. 2019. Breaking Parsers: Mutation-Based Generation of Programs with Guaranteed Syntax Errors. In Proceedings of the 12th ACM SIGPLAN International Conference on Software Language Engineering (SLE '19), October 20 – 22, 2019, Athens, Greece. ACM, New York, NY, USA, 5 pages. `https://doi.org/10.1145/3357766.3359542`. ACM Distinguished Paper Award.

We also use test suite construction methods that were introduced in our paper

- Phillip van Heerden, **Moeketsi Raselimo**, Konstantinos Sagonas, and Bernd Fischer. 2020. Grammar-Based Testing for Little Languages: An Experience Report with Student Compilers. In Proceedings of the 13th ACM SIGPLAN International Conference on Software Language Engineering (SLE '20), November 16 – 17, 2020, Virtual, USA. ACM, New York, NY, USA, 17 pages. `https://doi.org/10.1145/3426425.3426946`.

I worked closely with Phillip van Heerden to design the experimental framework used in the evaluation there, and contributed draft versions of the corresponding sections of that paper.

Chapter 2

Background

This chapter gives the background for the concepts described in subsequent chapters. In Section 2.1, we present preliminary material related to context-free grammars (CFGs), we describe the parsing technologies, over which our approaches were evaluated, we then describe test suite construction from CFGs (see Section 2.3) and finally introduce the spectrum-based fault localization framework in Section 2.4.

2.1 Context-Free Grammars

The previous chapter introduced context-free grammars (or simply grammars) and highlighted their importance in enabling several applications; CFGs define the syntax of most programming languages and can be exploited (by fuzzers) to capture the complex input characteristics of some software systems. In this section, we take a closer look at CFGs.

CFGs can be seen as limited rule-based replacement systems in which some symbols can be replaced by other symbols to form strings. Symbols which cannot be replaced are called *terminal* symbols. *Non-terminal* symbols are syntactic variables which can be replaced by other symbols. CFGs also contain a special non-terminal called the *start* symbol and *productions* (or *rules*) which define the replacement relation, i.e., define the interaction between terminals and non-terminals in forming strings. Before we fix the notational conventions that we adapt in our work, it is important that we look into the structure of productions. Each production consists of:

(a) a left-hand side non-terminal called the *head* or rule name;

(b) the symbol $\rightarrow$ which separates the left-hand side and the right side; and

(c) the *body* on the right-hand side, which contains zero or more terminals and non-terminals. This is also known as the definition of the production. The character "|" is often used to denote several alternative definitions of the same head.

Conventionally, productions from the start symbols are listed first. We use the grammar G for arithmetic expressions shown in Figure 2.1 below as a running example to illustrate concepts we have already introduced and some coming later in the chapter.

$$\begin{aligned}&S \to E\\&E \to E\,\texttt{+}\,E \mid E\,\texttt{*}\,E \mid \texttt{(}\,E\,\texttt{)} \mid \texttt{id}\end{aligned}$$

Figure 2.1: An example grammar G for arithmetic expressions

Adapted Notations. Formally, a CFG is a four-tuple $G = (N, T, P, S)$ with $N \cap T = \emptyset$, $V = N \cup T$, $P \subset N \times V^*$, and $S \in N$. We follow the notation in [10] and call S the *start symbol* and use the meta-variables $A, B, C, \ldots$ for non-terminals N, $a, b, c, \ldots$ for terminals T, X, Y, Z for *grammar symbols* V, p, q, r for *productions* or *rules* P, w, x, y, z for *words* over T^*, and $\alpha, \beta, \gamma, \ldots$ for *phrases* over V^*, with ε for the empty string and $|\alpha|$ for the length of α. We write $A \to \gamma$ for a rule $(A, \gamma) \in P$ and $P_A = \{A \to \gamma \in P\}$ for the rules for A. As usual, we assume that G is *augmented*, i.e., $P_S = \{S \to \dashv S' \vdash\}$, and S, $\dashv$, and $\vdash$ occur in no other production. $\dashv$ and $\vdash$ denote the start and end of input, respectively.

Using this notational convention, for the grammar shown in Figure 2.1, we get:

- $N = \{S, E\}$
- $T = \{\texttt{+}, \texttt{*}, \texttt{(}, \texttt{)}, \texttt{id}\}$
- $P = \{S \to E, E \to E\,\texttt{+}\,E, E \to E\,\texttt{*}\,E, E \to \texttt{(}\,E\,\texttt{)}, E \to \texttt{id}\}$
- S as the start symbol.

We call $\text{left}(R) = R \cup \{B \to \beta \in P \mid A \to B\alpha \in R\}$ the *left expansion* of $R \subseteq P$ and use $\text{closure}(R)$ to denote the closure of R under left expansion. Note that this mirrors the item set closure operation used in the construction of LR parsers (see Section 2.2.2).

The notation introduced above is sometimes not used when grammars are large and readability and reusability play an important role, as is the case for programming language grammars. The example grammar in Figure 1.1 and more examples in later chapters follow a much more expressive typographical convention and depart from the single letter meta-symbols. We can re-write G as shown in Fig. 2.2

Here, terminal symbols are typeset in **`bold typewriter`** font and non-terminals in *italics*. Also, where appropriate, we adopt extended Backus-Naur Form (EBNF) operators $*$, $+$ and ?. We write $A \to \alpha^*$ for $A \to \alpha A \mid \varepsilon$, $A \to \alpha^+$ for $A \to \alpha A \mid \alpha$ and $A \to \alpha?$ for $A \to \alpha \mid \varepsilon$.

$$\begin{aligned}&\mathit{start} \rightarrow \mathit{expr}\\&\mathit{expr} \rightarrow \mathit{expr}\,\texttt{+}\,\mathit{expr} \mid \mathit{expr} * \mathit{expr} \mid \texttt{(}\,\mathit{expr}\,\texttt{)} \mid \texttt{id}\end{aligned}$$

Figure 2.2: An equivalent re-written expression grammar.

2.1.1 Derivations and Parse Trees

Derivations. The process of forming a string w from the start rule S where at each step a non-terminal is replaced by the body of its definitions is called a *derivation* for w from S. We use the symbol $\Rightarrow$ to denote a replacement of a non-terminal by its definition. For example, for $w = id + id$ from the example grammar G in Figure 2.1, we have the following sequence of derivation steps.

$$S \Rightarrow E\,\texttt{+}\,E \Rightarrow \texttt{id + }E \Rightarrow \texttt{id + id}$$

More generally, we use $\alpha A \beta \Rightarrow \alpha\gamma\beta$ to denote that $\alpha A \beta$ *produces* $\alpha\gamma\beta$ by application of the rule $A \rightarrow \gamma \in P$ and use $\Rightarrow^*$ for its reflexive-transitive closure, which simply reads as "derives in zero or more steps". This enables us to shorten the derivation of $\texttt{id + id}$ to just $S \Rightarrow^* \texttt{id + id}$. We write $\Rightarrow_R$ if $A \rightarrow \gamma \in R \subseteq P$. We call a phrase α a *sentential form* if $S \Rightarrow^* \alpha$. Note that sentential forms may contain both terminals and non-terminals. At each step in the derivation, we have many choices of the order of the non-terminals to expand in a sentential form α. We consider in particular two types of derivations:

1. In a *left-most derivation*, we expand at each derivation step the left-most non-terminal in a sentential form. Hence, $\varphi \Rightarrow_{lm} \psi$ is a left-most derivation step, if there are u, σ, τ with $\varphi = uA\tau$, $A \rightarrow \alpha \in P$, and $\psi = u\alpha\tau$. The derivation for the string $\texttt{id + id}$ shown above is a left-most derivation, and we denote this by $S \Rightarrow^*_{lm} \texttt{id + id}$.

2. In a *right-most derivation* we expand, at each derivation step, the right-most non-terminal in a sentential form. Hence, $\varphi \Rightarrow_{rm} \psi$ is a right-most derivation step if there are σ, u, τ with $\varphi = \sigma A u$, $A \rightarrow \alpha \in P$, and $\psi = \sigma\alpha u$. We can re-order the derivation steps of the derivation above to give the right-most derivation as follows:

   $$S \Rightarrow_{rm} E + E \Rightarrow_{rm} E + \texttt{id} \Rightarrow_{rm} id + id$$

   We denote this by $S \Rightarrow^*_{rm} \texttt{id + id}$.

We use Δ for derivations. For a derivation $\Delta = \alpha_0 \Rightarrow_{p_1} \alpha_1 \Rightarrow_{p_2} \ldots \Rightarrow_{p_n} \alpha_n$, we use $\text{rules}(\Delta) = \bigcup_i \{p_i\}$ to denote the set of applied rules.

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., $L(G) = \{w \in T^* \mid S \Rightarrow^* w\}$. The

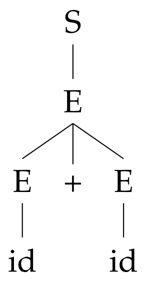


Figure 2.3: Parse tree of **`id + id`**.

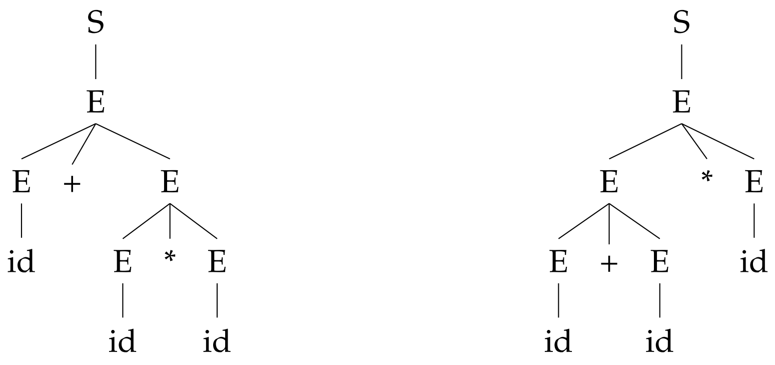


Figure 2.4: Parse trees of $w =$ **`id + id * id`**.

derivation of $w =$ **`id + id`** illustrated above shows that w is in the yield of the start symbol S and therefore, **`id + id`** $\in L(G)$.

Parse Trees. *A parse tree* (also called a derivation tree or concrete syntax tree) is a tree data structure that represents a derivation (more precisely, a set of derivations). It typically has the start symbol as root. It has terminal symbols at the leaf nodes, and at each non-leaf node represents an application of a production whose subtree is rooted by the head of that production and each symbol in the body of the production as child of the subtree. Figure 2.3 shows the derivation tree for the expression **`id + id`**.

We call a grammar *ambiguous* if it allows more than one parse tree for some input string. Our example grammar G allows two parse trees, shown in Figure 2.4 for a string $w =$ **`id + id * id`**.

2.1.2 Prefixes and Bounded Derivations

Sometimes we cannot construct a derivation for a complete word, only for an initial fragment. More specifically, 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 $k + 1$. For example, given the expression grammar G in Fig. 2.1, $u = \texttt{id +}$ is the maximal viable prefix of $uv' = \texttt{id + *}$, because for $v' = \texttt{id}$, $S \Rightarrow^* uv'$ while $\texttt{id}$ is a non-maximal viable prefix.

A derivation $\Delta = S \Rightarrow^* \omega$ is *k-prefix bounded for w* if (*i*) $\omega \Rightarrow^* w$ and (*ii*) for any derivation step $\alpha A \beta \Rightarrow \alpha \gamma \beta$ in Δ we have $\alpha \Rightarrow^* u$ and $\alpha A \beta \Rightarrow^* uv = w$ imply *a*) $|u| < k$ or *b*) $|u| = k$ and $u\beta \Rightarrow^* w$. We denote this by $S \overset{k}{\Rightarrow}{}^*_w \omega$. Intuitively, this means that a k-prefix bounded derivation for w never expands a non-terminal symbol whose yield in w will ultimately start only beyond a prefix of length k. $S \overset{k}{\Rightarrow}{}^*_w \omega$ is *maximal* if $\omega = u\alpha$ for $|u| = k$, i.e., if we have applied all rules in the k-prefix. Consider for example the derivation

$$\Delta = S \Rightarrow E\,\texttt{+}\,E \Rightarrow \texttt{(}\,E\,\texttt{)}\,\texttt{+}\,E \Rightarrow \texttt{(}\,E * E\,\texttt{)} + E \Rightarrow \texttt{(}\,\texttt{id} * E\,\texttt{)}\,\texttt{+}\,E = \omega$$

and $w = \texttt{(id * id) + id}$. Δ is k-prefix for w for k=2 because the yield of all non-terminal symbols expanded in Δ starts on or before w's second position; note that the end position does not matter. Note also that Δ is maximal for k=2 because the first two elements of w contain no non-terminal symbols (In fact Δ is also maximal for k=3 but not for k=4). Note that for any $w = uv \notin L(G)$ with maximal viable k-prefix u, there exist a not necessarily unique $w' = uv' \in L(G)$ and corresponding maximal k-prefix bounded derivation $S \overset{k}{\Rightarrow}{}^*_{w'} uX\alpha$ for w'. We call any such Δ a maximally viable k-prefix bounded derivation for w with *frontier* X and define its *frontier rules* as closure(P_X). We call the implied v' its right completion. For example, for $w = \texttt{(id * +)} \notin L(G)$, with the maximally viable 3-prefix $u = \texttt{(id *}$, we can select $v' = \texttt{id)}$ and the corresponding maximal 3-prefix bounded derivation $\Delta = S \Rightarrow \texttt{(id} * E$, which has the frontier symbol E, with the frontier rules $\{E \rightarrow E\,\texttt{+}\,E, E \rightarrow E * E, E \rightarrow \texttt{(}\,E\,\texttt{)}, E \rightarrow \texttt{id}\}$.

2.1.3 Left Recursion Elimination and Left Refactoring

In this section, we introduce certain grammar properties that allow (or prevent) the application of some practical parser implementations. For example, most top-down (see Section 2.2.1) parsers cannot handle left–recursive or ambiguous grammars. To circumvent this, there exist techniques to eliminate left recursion in a grammar, which guarantee that the transformed grammar (without left recursion) still describes the same language. However, transformations to eliminate ambiguity do not generalize and thus do not offer such guarantees.

Left Recursion Elimination

Generally, a grammar is *left-recursive* if it has a non-terminal A such that there is a derivation of a sentential form that uses itself as the leftmost symbol, i.e., we have $A \Rightarrow^+ A\alpha$ for some phrase α.

Direct Left Recursion. A grammar contains a direct (or an immediate) left recursion if there exists a production of the form $A \rightarrow A\alpha$ for some phrase α. Direct left recursion can be easily eliminated using the following transformations. We assume we have the productions.

$$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \cdots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \cdots \mid \beta_m$$

where no β_i begins with A and $\alpha_i \neq \varepsilon$. We replace the A-productions by

$$\begin{aligned} A &\rightarrow \beta_1 A' \mid \beta_2 A' \mid \cdots \mid \beta_m A' \\ A' &\rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \cdots \mid \alpha_m A' \mid \varepsilon \end{aligned}$$

where A' is a new non-terminal symbol.

This technique is sufficient for all grammars with immediate left recursion only. For example, the expression grammar shown in Figure 2.1 can be transformed using this same technique to produce the grammar in Figure 2.5.

$$\begin{aligned} S &\rightarrow E \\ E &\rightarrow TE' \\ E' &\rightarrow \texttt{+}\,TE' \mid \varepsilon \\ T &\rightarrow FT' \\ T' &\rightarrow *\,FT' \mid \varepsilon \\ F &\rightarrow \texttt{(}\,E\,\texttt{)} \mid \textbf{id} \end{aligned}$$

Figure 2.5: Left recursion eliminated grammar G'.

However, indirect left recursion presents different challenges that we address below.

Indirect Left Recursion. A grammar contains indirect left recursion if it contains rules of the form

$$\begin{aligned} A_0 &\rightarrow \beta_0 A_1 \alpha_0 \\ A_1 &\rightarrow \beta_1 A_2 \alpha_1 \\ &\ldots \\ A_m &\rightarrow \beta_m A_0 \alpha_m \end{aligned}$$

where each β_i can derive the empty string and the α_i are arbitrary sequences of terminals and non-terminals. Then the following derivation exposes the left recursion

$$A_0 \Rightarrow \beta_0 A_1 \alpha_0 \Rightarrow^* A_1 \alpha_0 \Rightarrow \beta_1 A_2 \alpha_1 \alpha_0 \Rightarrow^* \cdots \Rightarrow^* A_0 \alpha_m \ldots \alpha_1 \alpha_0.$$

We can systematically remove all recursion using the algorithm described by Aho *et al.* [10, page 212, Section 4.3.3]. The algorithm is guaranteed to remove all left recursion if the grammar does not allow cycles (i.e., derivations of the form $A \Rightarrow^* A$).

Left Factoring

Another transformation that is useful is *left factoring*. The need for left factoring arises when the choice among A-productions is not clear because they have a common prefix. A-productions have a common prefix if they are of the form, $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \cdots \mid \alpha\beta_m$. Hence, it is not immediately clear which alternative of A to pick in derivation. In the left-factored version of the grammar, the A-productions are replaced by $A \rightarrow \alpha A', A' \rightarrow \beta_1 \mid \beta_2 \mid \cdots \mid \beta_m$, where A' is a new non-terminal symbol as above. This version describes the same language.

2.1.4 Grammar Predicates

In this section, we introduce functions over grammars that simplify the formulation of parsing algorithms; they ensure correct parsing decisions and are invoked by most error recovery algorithms when parsers try to recover from syntax errors.

We call α *nullable* if $\varepsilon \in \text{yield}(\alpha)$. We define the *first set* of a phrase α as the set of all terminal symbols $a \in T$ that can begin strings derived from α. We write $\text{first}(\alpha) = \{a \mid \alpha \Rightarrow^* a\beta\}$. Its reverse, the *last set* of α comprises of terminal symbols that can end strings derived from α, i.e., $\text{last}(\alpha) = \{a \mid \alpha \Rightarrow^* \beta a\}$), We also define the *follow set* of symbol X as the set of all terminal symbols that can appear immediately on the right hand side of X in some phrase at some during the derivation. We formally write $\text{follow}(X) = \{a \mid S \Rightarrow^* \alpha X a \beta\}$. The *precede set* of a symbol X, inversely is the set of all terminals that can appear immediately before X in some sentential form, i.e, $\text{precede}(X) = \{a \mid S \Rightarrow^* \alpha a X \beta\}$.

For the augmented version of our example grammar G in Figure 2.1 we get:

- $\text{first}(S) = \text{first}(E) = \{(, id\}$;
- $\text{last}(S) = \text{last}(E) = \{), id\}$;
- $\text{precede}(S) = \{\dashv\}$;
- $\text{follow}(S) = \{\vdash\}$;
- $\text{follow}(E) = \{\vdash, \textbf{+}, *, \textbf{)}\}$;
- $\text{precede}(E) = \{\dashv, \textbf{+}, *, \textbf{(}\}$.

These algorithms are usually implemented by fix-point computations.

2.2 Parsing Methods

Parsing, in its broadest sense, is the process of searching for derivations. Parsing algorithms can be summarized in three classes that search (i) exhaustively; (ii) for left-most derivations; and (iii) for right-most derivations. In this thesis, we only discuss parsing methods that employ left- and right-most derivation search. Since left-most derivations derive the input from the start symbol, the parsing strategies that find these left-most derivations are also called *top-down* parsing methods. Analogously, we have *bottom-up* parsing strategies that reduce the input to the start symbol.

2.2.1 Top-Down Parsing

Top-down parsers construct parse trees for input strings, starting from the start symbol (root) and creating nodes in a depth-first fashion, working their way down until the entire string is processed. The key challenge is to determine, at each parsing step, the A-production to be applied to expand a non-terminal A. The grammar predicates we introduced in Section 2.1.4 are typically used by a top-down parser in these circumstances, as we will see later in this section.

One class of top-down parsers that are of special interest are *predictive parsers* which are typically used for parsing grammars that belong to the $LL(k)$ family grammars. These parsers select the correct A-production by looking ahead at the input string for k symbols. The first "L" in $LL(k)$ means that these parsers read the input from left to right while searching for left-most derivations (the second "L"). The "k" is the number of lookahead symbols at each parsing step to make parsing decisions. The most common value of k is 1.

Ambiguous grammars and grammars that are left-recursive are not $LL(k)$ for any value of k. Modern parsing tools, in particular, ANTLR [118], generate parsers that use the more powerful $LL(*)$ [119] and adaptive-$LL(*)$ [120] algorithms with unbounded lookahead. These algorithms enable ANTLR to handle grammars with direct left recursion and ambiguous grammars, although grammars with indirect left recursion are rejected by ANTLR.

However, we restrict our discussion to just $LL(1)$ parser implementations, since this is sufficient in understanding more advanced and modern top-down approaches. We look into the most popular approach to implementing predictive parsers; the recursive descent parsing strategy that is typically employed in hand-rolled parsers. More top-down parsing concepts are discussed in detail by Aho *et al.* [10].

Recursive Descent Parsing. A recursive descent parser typically implements the productions directly as functions. For each non-terminal, we have one function that implements it. The body of that function is typically implemented with a switch statement on the next token, with each

$$
\begin{aligned}
\mathit{stmt} &\rightarrow \textbf{if}\ \mathit{expr}\ \textbf{then}\ \mathit{stmt}\ \textbf{else}\ \mathit{stmt} \\
&\mid \textbf{begin}\ \mathit{stmt}\ \textbf{end} \\
&\mid \textbf{print}\ \mathit{expr} \\
\mathit{expr} &\rightarrow \mathtt{id}
\end{aligned}
$$

(a) Example grammar

```
void stmt() {
    switch(tok) {
        case IF: eat(IF); expr();
                 eat(THEN); stmt();
                 eat(ELSE); stmt(); break;
        case BEGIN: eat(BEGIN); stmt();
                    eat(END); break;
        case PRINT: eat(PRINT); expr(); break;
        default: error();
    }
}
```

(b) Function definition that implements *stmt*-productions.

Figure 2.6: A recursive descent parser implementation.

expansion choice in the case clause. The expected next tokens and the corresponding expansion choices are computed via the first and follow functions. Recursive-descent parsing comes in two flavours, recursive-descent parsers with and without backtracking. We, however, do not discuss the latter because they are rarely used in practice due to the high computational overheads that affect their applicability.

Figure 2.6 (a) shows a stripped down example grammar that defines three types of statements, Figure 2.6 (b) is a function that implements these statement productions. We see here that the LL(1) grammar structure allows the parser to decide which *stmt*-alternative to use in a search for derivation.

2.2.2 Bottom-Up Parsing

Bottom-up parsers search for right-most derivations by (implicitly or explicitly) constructing the parse tree from the leaf nodes all the way up to the root. The generalized form of bottom-up parsing is called *shift-reduce* parsing, which typically uses an explicit parse stack to represent derivations. The key operations that the shift-reduce parsers perform are pushing symbols onto the parse stack (shift) and popping symbols off the stack. The pop action is an integral part of the reduce operation, which searches for a specific substring at the top of the stack that matches the body of a A-production and replaces it with a non-terminal A at the head of that production. Informally, we call the substring at the top of the stack the *handle*. Therefore, each reduction step pops the handle off the stack and replaces it with the head of a matching production. A formal definition of a handle

$$
\begin{aligned}
&S \rightarrow E \\
&E \rightarrow E \,\texttt{+}\, T \mid T \\
&T \rightarrow T * F \mid F \\
&F \rightarrow \texttt{(}\, E \,\texttt{)} \mid \texttt{id}
\end{aligned}
$$

Figure 2.7: An expression grammar G_{expr} that reflects the precedence and associativity of operators

Table 2.1: Table showing parse stack configurations and actions taken by a shift-reduce parser that implements the example grammar on an input expression $\texttt{id} * \texttt{id}$.

step	*stack*	*input*	*action*
1	$\dashv$	$\texttt{id}_1 * \texttt{id}_2 \vdash$	shift $\texttt{id}_1$
2	$\dashv \texttt{id}_1$	$* \texttt{id}_2 \vdash$	reduce by $F \rightarrow \texttt{id}$
3	$\dashv F$	$* \texttt{id}_2 \vdash$	reduce by $T \rightarrow F$
4	$\dashv T$	$* \texttt{id}_2 \vdash$	shift $*$
5	$\dashv T *$	$\texttt{id}_2 \vdash$	shift $\texttt{id}_2$
6	$\dashv T * \texttt{id}_2$	$\vdash$	reduce by $F \rightarrow \texttt{id}$
7	$\dashv T * F$	$\vdash$	reduce by $T \rightarrow T * F$
8	$\dashv T$	$\vdash$	reduce by $E \rightarrow T$
9	$\dashv E$	$\vdash$	reduce by $S \rightarrow E$
10	$\dashv S$	$\vdash$	accept

follows below.

Definition 2.2.1 (Handle)**.** Let $G = (N, T, P, S)$ and $S \Rightarrow^*_{rm} \beta A y \Rightarrow_{rm} \beta \gamma y$ then γ is called a *handle* or *redex* of the right-sentential form $\beta\gamma y$. Each prefix of $\beta\gamma$ is called a *viable prefix* of G.

Table 2.1 illustrates the actions taken by a shift-reduce parser that recognizes the language of the expression grammar G_{expr} shown in Figure 2.7 on an input string $w = \texttt{id} * \texttt{id}$. We also use the $\dashv$ symbol to mark the bottom of the stack and $\vdash$ to mark the right end of the input string. The top of the stack is at its right-most end. For clarity, we also denote the left-most (resp. right-most) $\texttt{id}$ in w by $\texttt{id}_1$ (resp. $\texttt{id}_2$) on the input buffer and stack. We can see that the parser, at each step, reads symbols left to right from the input, and shifts them onto the stack, until it finds a handle it can replace by the head of the matching symbol. For example, the leftmost $\texttt{id}_1$ is shifted onto the stack in the first step, and reduced by the production $F \rightarrow \texttt{id}$ in the second step. This process is repeated until a syntax error is detected or until the input buffer is empty, and the stack contains the start symbol, the configuration that marks successful parsing as shown in step 10.

In general, we can summarize the actions that all practical implementations of shift-reduce parsers make beyond shift and reduce as follows:

1. *Shift*. The parser pushes the next symbol onto the parse stack.
2. *Reduce*. This operation marks the actual application of the grammar production. This occurs when the top of the stack (contains the handle) matches one of the A-productions, this handle is popped off and replaced by the non-terminal A.
3. *Accept*. This operation marks the successful end of parsing.
4. *Error*. When syntax violations are detected, the parser often tries to recover from the syntax error by calling an appropriate error recovery and reporting routine. Parsing continues if recovery is successful, otherwise the process terminates fatally.

Items

An item is a rule $A \to \alpha \bullet \beta$ with a designated position (denoted by $\bullet$) on its right-hand side. An item is called *kernel item* if $\alpha \neq \varepsilon$ or $\alpha = \varepsilon$ and $A = S$ and *non-kernel item* otherwise. We use $P^\bullet$ to denote the set of all items, i.e., all rules with all designated positions, and define a function $\text{items}(A \to \gamma) = \{A \to \alpha \bullet \beta \mid \gamma = \alpha\beta\}$ that maps a rule to all its items. 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 terminal symbols that can occur in a derivation immediately to the left (resp. right) of the designated position [130]. 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.

Definition 2.2.2 (left set, right set)**.** The functions left, right : $P^\bullet \to \mathcal{P}(T)$ are defined by

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

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

Basics of LR parsing

Shift-reduce parsers are mostly implemented for grammars that belong to the $LR(k)$ family of grammars. The "L" here also means that the input string is scanned from left to right; the "R" means searching for right-most derivation in reverse, while the k also means the number of lookahead symbols that are used to make parsing decisions. The most common values of k are 0 and 1.

LR parsers differ slightly from generic shift-reduce parsing algorithms in two ways; LR parsers are mostly table-driven and maintain a main stack of states (instead of symbols) to keep track of the parsing progress. An LR parser also uses a finite state machine (or an *LR automaton*) to make parsing decisions, i.e, when to push and pop states off the stack. This automaton comes in different flavours (e.g., LR(0), SLR, LALR, etc); however, here we look at the simplest LR(0) automaton only as it provides a baseline for practical ones such as LALR automata. Each state of the automaton represents a set of $LR(0)$ *items* (or simply items). What an item $A \rightarrow \alpha \bullet \beta$ represents, intuitively, is that the parser has just successfully processed a substring derivable from α and is expecting the substring derivable from β in the input.

Note that we use an augmented grammar G', with the start symbol S' and $S' \rightarrow \dashv S \vdash$ as the only production for S'. Note also that we use $\dashv$ and $\vdash$ as conceptual markers and never use them in any transition. Therefore, an initial item is $S' \rightarrow \dashv \bullet S \vdash$, and we consider $S' \rightarrow \dashv S \bullet \vdash$ as the reduce item. We then define the two functions *closure* and *goto* below.

1. *closure*: Let I be an item set, then the $closure(I)$ comprises I and all items that can be added to $closure(I)$ following the following steps:

   (i) whenever $A \rightarrow \alpha \bullet B\beta$ is in $closure(I)$ and $B \rightarrow \gamma$ is a production, then we add to $closure(I)$ all items of the form $B \rightarrow \bullet\gamma$;

   (i) repeat until no more items need to be added.

2. $goto(I, X)$ is defined to be the closure of all items $\{A \rightarrow \alpha X \bullet \beta\}$ such that $\{A \rightarrow \alpha \bullet X\beta\}$ is in I, where I is a set of items and X is a grammar symbol. Intuitively, $goto(I, X)$ defines a transition from a state representing the item set I on a given input symbol X.

Figure 2.8 shows the LR(0) automaton for the original grammar $G = (\{S, A, B\}, \{a, b\}, \{S \rightarrow Aa, S \rightarrow Ba, S \rightarrow ac, A \rightarrow a, B \rightarrow a\}, S)$. In order to capture the item set representing the initial state 1, we start with the initial item $S' \rightarrow \dashv \bullet S \vdash$ to $closure(1)$. We then add the items from the S-productions, $S \rightarrow \bullet Aa$, $S \rightarrow \bullet Bb$ and $S \rightarrow \bullet ac$, then $A \rightarrow \bullet a$ and $B \rightarrow \bullet a$ as $S \rightarrow \bullet Aa$ and $S \rightarrow \bullet Bb$ are already contained in $closure(1)$ $goto(1, a)$ specifies the transition from state 1 on the symbol a; the target state 5 contains the items $S \rightarrow a \bullet c$, $A \rightarrow a\bullet$ and $B \rightarrow a\bullet$.

Parsing Conflicts

Ambiguous grammars force an LR parser into a configuration in which it, although presented with the knowledge of the stack contents and a valid next input symbol, cannot decide which action to take. We identify three types of conflicts below:

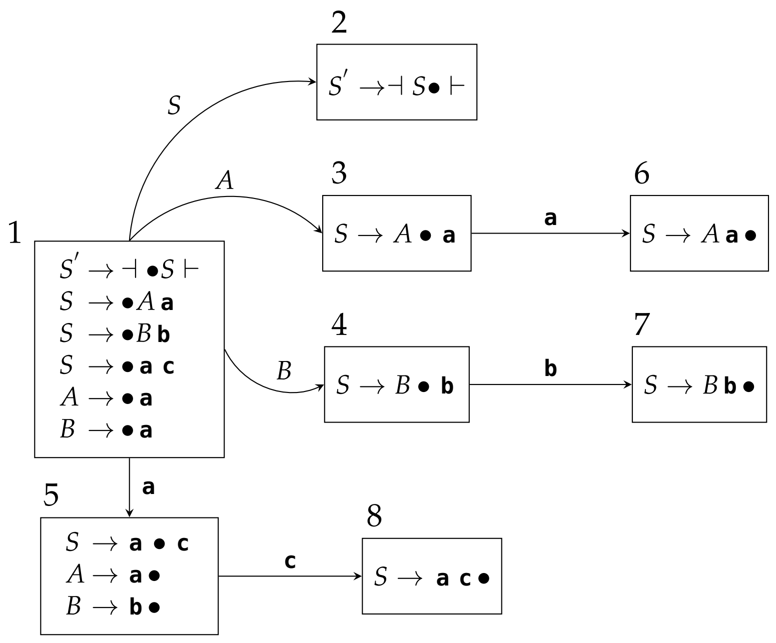


Figure 2.8: $LR(0)$ automaton.

(i) *Shift/reduce conflicts*: This conflict occurs when the LR parser is in a state where it either shifts the next symbol on to the stack and advance to the next state or perform a reduce operation that marks a successful completion of a production. In Figure 2.8, we have this type of conflict in state 5 where the parser can either shift c and advance to state 8 or reduce by either $A \to a$ or $B \to b$. Practical implementations typically resolve this type of conflict in favour of shifting.

(ii) *Reduce/reduce conflicts*: This conflict occurs when the parser cannot deterministically decide among a number of productions which one to pick to perform a reduction. State 5 in the LR(0) automaton in Figure 2.8 has a reduce/reduce conflict between productions $A \to a$ and $B \to b$. Although this type of conflict is typically resolved in favour of an earlier production, resulting parsers often run into stability issues and mostly build the wrong parse trees. In our experimental evaluation, we discard all subject grammars with reduce/reduce conflicts.

(iii) *Accept/reduce conflicts*: These are the least common and usually forgotten (rightly so) type of conflicts. The parser runs into an accept/reduce conflict when it cannot decide whether to mark parsing successfully completed in its entirety or just to mark the application of a single grammar production as successful.

Practical LALR Parsing

The most widely used LR parser implementation in practice is based on the *lookahead-LR (LALR)* method. It can handle more grammars (i.e., construct conflict-free LR-automata for more grammars) than the simple LR(0) method. For example, in the LR(0) automaton in Figure 2.8, state 5 has both shift/reduce and reduce/reduce conflicts and the parser can build wrong trees on any derivations that pass through that state. The LALR algorithm is certainly not the most powerful LR parsing mechanism known, but what makes it attractive is the low memory overhead incurred due to the fact that its automaton has the same number of states as the equivalent LR(0) but items on the LALR automaton contains more information to help in parsing decisions. Each item of the LALR automaton is of the form, $A \rightarrow \alpha \bullet \beta/\{a\}$, where $A \rightarrow \alpha \bullet \beta$ is known as the *core* and $\{a\}$ is the set of terminals a or the end–marker symbol $\vdash$ used as lookahead symbols. We use "/" to separate the core of an item from its set of lookahead symbols.

The two functions *closure* and *goto* that we saw earlier are also slightly different here as they take the lookahead symbol into account. To compute $closure(I)$ for an item set I, we add the item $A \rightarrow \alpha \bullet B\beta/\{a\}$ to $closure(I)$, where $\{a\}$ is the set of lookahead symbols. If $B \rightarrow \gamma$ is a production, we then add to $closure(I)$ all the items of the form $B \rightarrow \bullet\gamma, \{b\}$, where $\{b\}$ represents all terminal symbols in first(βa). The *goto* transition relation remains largely unchanged but the lookahead mechanism heavily influences the next state configuration, i.e., the transition to the next state becomes viable when the next token in the input buffer matches the lookahead symbol in the current state. For example, say we are in some state I with one of the following items $A \rightarrow a \bullet \beta/\{a\}$ and $B \rightarrow a \bullet /\{b\}$. The LR(0) parser would encode a shift/reduce conflict, but the LALR method would reduce by $B \rightarrow a$ if the next input symbol is a b, and shifts otherwise.

Running example. We use the grammar G shown in Figure 2.10 to illustrate the computation of *closure* and *goto* of LR item sets. The LALR automaton in Figure 2.9 represents the collection of LR items for grammar G.

We initialize $closure(0)$ with the item $S' \rightarrow \dashv \bullet S \vdash /\{\vdash\}$, with the lookahead from the S-productions being first$(\vdash) = \{\vdash\}$, we then add $S \rightarrow \bullet AA/\{\vdash\}$. We then add items $A \rightarrow \bullet aB/\{a\}$ and $A \rightarrow \bullet a/\{a\}$ from the A-productions to $closure(0)$, with their lookahead symbol computed via first$(A \vdash) = \{a\}$. $goto(0, A)$ brings us to state 2 with the initial item $S \rightarrow A \bullet A/\{\vdash\}$ in $closure(2)$ and the entire process is repeated until no further items can be added.

The LALR Parse Table

Another useful ingredient that plays a crucial role in the LALR machinery is the parse table, Which, encodes the LALR automaton. We discuss its

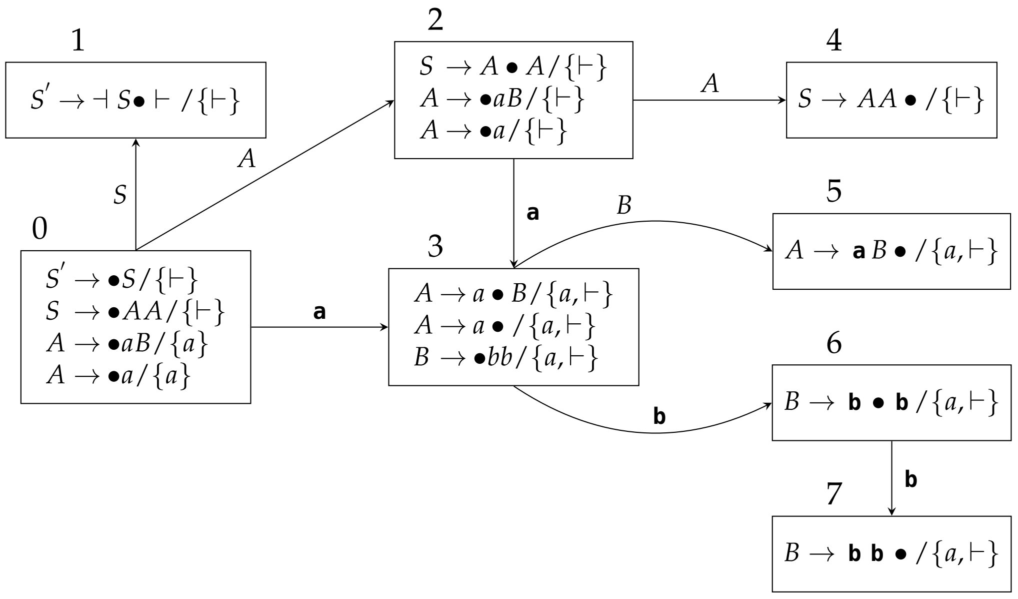

1
$S' \to \dashv S \bullet \vdash / \{\vdash\}$
2
$S \to A \bullet A / \{\vdash\}$
$A \to \bullet aB / \{\vdash\}$
$A \to \bullet a / \{\vdash\}$
A
4
$S \to AA \bullet / \{\vdash\}$
A
S
0
$S' \to \bullet S / \{\vdash\}$
$S \to \bullet AA / \{\vdash\}$
$A \to \bullet aB / \{a\}$
$A \to \bullet a / \{a\}$
a
a
B
5
$A \to \mathbf{a} B \bullet / \{a, \vdash\}$
3
$A \to a \bullet B / \{a, \vdash\}$
$A \to a \bullet / \{a, \vdash\}$
$B \to \bullet bb / \{a, \vdash\}$
6
$B \to \mathbf{b} \bullet \mathbf{b} / \{a, \vdash\}$
b
b
7
$B \to \mathbf{b}\,\mathbf{b} \bullet / \{a, \vdash\}$

Figure 2.9: *LALR* automaton.

$$\begin{aligned} S &\to AA \\ A &\to aB \mid a \\ B &\to bb \end{aligned}$$

Figure 2.10: An example grammar G we use to illustrate LALR parsing concepts.

construction in this section. The systematic construction of the parse table can be summarized by two output functions, $ACTION$ and $GOTO$. From the sets of states $I_0 \ldots I_n$ from the corresponding LALR automaton, we index the rows of the table by the same number of states. We detail the steps to fill entries of each part of the LALR parse table below using the corresponding LALR automaton.

ACTION. Conventionally, columns of the $ACTION$ part of the table are indexed by all terminals (including the endmarker $\vdash$) and entries for state I_i are constructed for each terminal symbol of the grammar.

- If $A \to \alpha \bullet a\beta/\_$ is in I_i and $goto(I_i, a) = I_j$, then we set $ACTION[I_i, a]$ to `shift` I_j.
- If $A \to \alpha \bullet / \{a\}$ is in I_i, then we set $ACTION[I_i, a]$ to `reduce` $A \to \alpha$ for all lookahead symbols a.
- If $S' \dashv\!\to S \bullet \vdash / \{\vdash\}$ is in I_i, then we set $ACTION[I_i, \vdash]$ to `accept`.

GOTO. The $GOTO$ entries for each state I_i are constructed for all non-terminals. More specifically, if $goto(I_i, A) = I_j$, then we set $GOTO[I_i, A]$ to I_j.

Table 2.2: The LALR parse table configuration for example grammar G shown in Figure 2.10 that encodes the automaton in Figure 2.9.

	ACTION			*GOTO*		
states	***a***	***b***	$\vdash$	***S***	***A***	***B***
0	`shift` 3			1	2	
1			`accept`			
2	`shift` 3		`reduce` $A \to a$		4	
3	`reduce` $A \to a$	`shift` 6	`reduce` $A \to a$			5
4			`reduce` $S \to AA$			
5	`reduce` $A \to aB$		`reduce` $A \to aB$			
6		`shift` 7				
7	`reduce`$B \to bb$		`reduce`$B \to bb$			

Algorithm 1: The generic LR parsing algorithm (adapted from [10])

```
input  : An input string $w$
input  : An LR parse table with $ACTION$ and $GOTO$ operations for
         grammar $G$
output: A successful parse if $w \in L(G)$ otherwise syntax error report
parsing_done ← false
repeat
    $s \leftarrow stack.top()$
    $v \leftarrow next\_input(w)$
    if $ACTION[s, v] ==$ shift $t$ then
        $stack.push(t)$
    else if $ACTION[s, v] ==$ reduce $A \to \alpha$ then
        $i \leftarrow 0$
        while $i < |\alpha|$ do
            $stack.pop()$
            $i \to i + 1$
        $stack.push(GOTO[s, A])$
    else if $ACTION[s, v] ==$ accept then
        $parsing\_done \leftarrow true$
    else
        $error\_recovery()$
        $parsing\_done \leftarrow true$
until $parsing\_done$
```

Error. We set entries not defined by the previous steps as `error`, either explicitly or implicitly.

The LR Parsing Algorithm

Algorithm 1 shows the main parsing routine that puts together all the LALR parsing concepts we have discussed so far. It takes as input an input string w and a parse table, maintains an internal parse stack of states and returns as output, at the very least, success if w is accepted or reports syntax violations. It is worth noting that all LR parsing methods use the same parsing algorithm, but differ in the construction of their parse tables.

Here, we assume a stable lexical analyser interface that interacts with the parser via a call to `next_token()` in line 4. The function returns tokens and feeds them to the parser. More lexical analysis concepts can be found in Aho *et al.* [10, Chapter 3]. The four basic actions performed by an LR parser are as follows: (i) *shift*: lines 13 and 14; (ii) *reduce*: from line 7 to 12; (iii) *accept*: lines 13 and 14; and (iv) *error* in the else-branch at line 16 via a call to `error_recovery()`. There exist various algorithms for syntax error recovery, but we leave their discussion for related work in Section 6.6.

An illustrative example. Table 2.3 shows an illustration of the main parsing loop in Algorithm 1 based on the example grammar G (see Figure 2.10) and its corresponding parse table (see Table 2.2). Let $w = aabb$ be an input string that we want to parse. Initially, the stack contains an initial state (state 0), and the input is read from left to right. The parsing process starts from line 3 with the top of the stack $s = 0$ and the next input symbol $v = a$, hence $ACTION[0, a] = \texttt{shift}\ 3$ results in state 3 being pushed onto the stack. The next iteration has values $s = 3$ and another $v = a$, and we get $ACTION[3, a] = \texttt{reduce}\ A \rightarrow a$. The production $A \rightarrow a$ has a length of 1, and we pop off one state from the parse stack (lines 8-11). After the pop we now have $s = 0$. We consult the transition function to give us the next state to push on the stack, since $GOTO[0, A] = 2$, we then push state 2 which then becomes our new top of the stack. The parser continues in this manner, mostly deciding when to shift and and reduce by a production in G until it ends up in a configuration, $ACTION[1, \vdash]$ in which case the input string w is accepted. A different input string $w' = aab$ leads the parser to an undefined configuration from the parse table shown in Table 2.2 and in which case the parser calls an appropriate error recovery routine, reports the error to the user and parsing terminates.

2.3 Grammar-Based Testing

In this section, we introduce one of the established fields that our proposed approaches build on. *Grammar-based testing* is a type of software testing in which a grammar is used to generate test inputs. This form of testing was originally applied to test parsers and even grammars themselves [124]. Similar techniques have been applied to other software, such as compilers,

Table 2.3: Stack configuration and actions performed by an LALR parser that implements an example grammar G in Figure 2.10 on an input string $w = aabb$.

stack	*rest of input*	*action*
0	$aabb \vdash$	`shift 3`
0 3	$abb \vdash$	`reduce` $A \rightarrow a$
0 2	$abb \vdash$	`shift 3`
0 2 3	$bb \vdash$	`shift 6`
0 2 3 6	$b \vdash$	`shift 7`
0 2 3 6 7	$\vdash$	`reduce` $B \rightarrow bb$
0 2 3 5	$\vdash$	`reduce` $A \rightarrow aB$
0 2 4	$\vdash$	`reduce` $S \rightarrow AA$
0 1	$\vdash$	`accept`

and have enabled more recent software testing approaches such as fuzzing to find bug and security vulnerabilities that lie deeper in the system beyond the syntax analysis stage.

We can identify different application scenarios of grammar-based testing, depending on the nature of the system under test, the availability of grammars and the specific goal to be achieved by the generated tests inputs:

System testing. In system testing, the overall goal is to demonstrate the reliability and robustness of the SUT. More specifically, the idea is to generate from a known *reference grammar* G_{ref} test inputs that exercise the SUT thoroughly. The degree to which the SUT is exercised can be measured using common control- and data-flow based software coverage metrics (e.g., the number of branches covered) or the number of system failures induced by the tests (e.g., system crashes).

Grammar testing. In grammar testing, which is the main focus of our discussion in this section, the goal is to demonstrate that a different *test grammar* G_{test} is (not) equivalent to G_{ref}. However, while context–free grammar equivalence is undecidable in general, the idea here is to generate test inputs that show that G_{test} is incorrect with respect to G_{ref} (i.e., $L(G_{test}) \neq L(G_{ref})$) or that G_{test} is incomplete with respect to G_{ref} (i.e., $L(G_{test}) \not\supseteq L(G_{ref})$).

Before we introduce different grammar-based test suite construction methods, we formally define test suites in Section 2.3.1 and introduce and highlight differences among *failure*, *error*, and *fault*, concepts which are commonly used interchangeably in various settings.

2.3.1 Test suites

A *test suite* consists of a list of SUT inputs and corresponding expected outputs (which can also be specific system errors, e.g., for illegal inputs). 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 expected outputs are either "accept" or "reject". More detailed expected outputs (e.g., error locations) could prevent the misclassification of applied rules, and so increase the precision of the fault localization, but are difficult to implement because they may depend on internal aspects of the parser (e.g., an error correction strategy).

2.3.2 Failure, error, fault

The informal notion of a "bug" can be deconstructed into three different concepts [1, 144]. A *failure* is a situation where the system's observed output deviates from the correct output, an *error* is an internal system state that may lead to a failure, and a *fault* is a code fragment which causes an error in the system when it is executed. Note that errors do not necessarily manifest themselves as observable failures.[1] *Fault localization* is an attempt to identify the unknown position of the fault from an observed failure.

2.3.3 Grammar-Based Test suite Construction

For our experimental evaluation, we use test suites that satisfy different grammar coverage criteria and have certain characteristics. We use both positive (i.e., syntactically valid) and negative test suites that contain a single, well-defined error. The material presented in this section is an adoption of our work presented in [130] and [150], and some formal definitions are taken from there.

Positive test suites

For the construction of positive tests TS^+, we use a *generic cover* algorithm proposed by Fischer *et al.* [43]. Its basic idea is to

(i) iterate over all symbols $X \in V$,

(ii) embed X, i.e., compute a minimal derivation $S \Rightarrow^* \alpha X \omega$,

(iii) cover X, i.e., compute a set of minimal derivations $X \Rightarrow^* \gamma$ that conform to the criterion, and

(iv) convert the sentential form into a word, i.e., compute a minimal derivation $\alpha\gamma\omega \Rightarrow^* w$ where each non-terminal A in $\alpha\gamma\omega$ is replaced by its minimal yield w_A.

Note that this algorithm is by construction biased towards very short tests because it uses minimal derivations in all steps.

We use different grammar coverage criteria. The most basic grammar coverage criterion is *symbol coverage* that simply ensures that every grammar

[1]The notion of "defect", "infection", and "failure" by Zeller [171] subtly differs from our separation of "failure - error - fault" and is not widely used.

symbol $X \in V$ is used in the generation of a test suite TS^+. Definition 2.3.1 gives a formal description. Even in its simplest form, test suites that satisfy this criterion can still be very large. However, these test suites are typically too weak to provide confidence that the grammar is correct or that the system under test is well tested. In our experimental comparison of positive syntactic tests [150], we show that symbol coverage test suites fall far behind in achieved line coverage and the number of system crashes.

Definition 2.3.1 (symbol coverage)**.** Let $G = (N, T, P, S)$ be a grammar, $V = N \cup T$ and TS^+ be a positive test suite for G. A word $w \in TS^+$ covers a symbol $X \in V$ iff $S \Rightarrow^* \alpha X \beta \Rightarrow^* w$. TS^+ satisfies symbol coverage (w.r.t. G) iff each symbol $X \in V$ is covered by a word $w \in TS^+$.

Rule coverage (Definition 2.3.2) takes a step further and tries to strengthen symbol coverage. Test suites that satisfy rule coverage guarantee that every production $A \rightarrow \alpha \in P$ is exercised in a derivation of at least one word in the test suite.

Definition 2.3.2 (rule coverage)**.** Let $G = (N, T, P, S)$ be a grammar and TS^+ be a positive test suite for G. A word $w \in TS^+$ covers a rule $A \rightarrow \gamma \in P$ iff $S \Rightarrow^* \alpha A \beta \Rightarrow \alpha \gamma \beta \Rightarrow^* w$. TS^+ satisfies rule coverage (w.r.t. G) iff each rule $A \rightarrow \gamma \in P$ is covered by a word $w \in TS^+$.

Rule coverage offers some confidence of grammar correctness and triggers most of the surface (i.e., easy-to-find) bugs. Lämmel [82] defines *context dependent rule coverage* (CDRC Definition 2.3.3) which generalizes rule coverage and has become a standard criterion. The basic idea behind CDRC is to apply each A-production in all occurrences of the non-terminal A in all rules. CDRC subsumes rule coverage and induces richer test suites.

Definition 2.3.3 (CDRC coverage)**.** Let $G = (N, T, P, S)$ be a grammar and TS^+ be a positive test suite for G, and $p = A \rightarrow x_1 \ldots x_n \in P$, be a rule. A word $w \in TS^+$ *covers the occurrence* $x_i \in N$ *in* p *with the rule* $q_i = x_i \rightarrow \delta \in P$ iff $S \Rightarrow^* \alpha A \beta \Rightarrow \alpha x_1 \ldots x_n \beta \Rightarrow \alpha x_1 \ldots x_{i-1} \delta x_{i+1} \ldots x_n \beta \Rightarrow^* w$. TS^+ *covers* x_i *in* p if it contains a word w_j for each rule $q_j = x_i \rightarrow \delta_j \in P$ such that w_j covers x_i in p with q_j; it covers p if it covers all x_i in p. TS^+ satisfies *context-dependent rule coverage* (w.r.t. G) iff each rule $p \in P$ is covered by a word $w \in TS^+$.

We also use k-step coverage (called k-path in work by Havrikov and Zeller [56]) in our experimental evaluation. This criterion induces longer words by considering k-depth bounded derivations that cover every pair $(X, Y) \in V$.

Definition 2.3.4 (k-step coverage)**.** Let $G = (N, T, P, S)$ be a grammar, $V = N \cup T$ and TS^+ be a positive test suite for G. A word $w \in TS^+$ covers a pair

$(X, Y) \in V \times V$ in at most k steps iff $S \Rightarrow^* \alpha X \beta \Rightarrow^l \alpha\gamma Y \delta\beta \Rightarrow^* w$ with $l \leq k$. TS^+ satisfies k-step coverage (w.r.t. G) iff each pair (X, Y) with $X \Rightarrow^l \alpha Y \beta$ and $l \leq k$ is covered by a word $w \in TS^+$.

We can interpret *symbol*, *rule* and *CDRC* criteria using k-step: *symbol* can be seen as 0-step, *rule* as 1-step while *CDRC* is a 2-step algorithm.

We introduced a fixpoint version of k-step in our previous work [150] shown in Definition 2.3.5. This criterion ensures that all shortest paths between every pair of symbols $(X, Y) \in V$ are covered. We showed that it produces compact test suites that achieve code coverage that is comparable to criteria that induce longer and deeper derivations such as k-step and bfs_k.

Definition 2.3.5 (derivable pair coverage)**.** Let $G = (N, T, P, S)$ be a grammar, $V = N \cup T$ and TS^+ be a positive test suite. A word $w \in TS^+$ covers a pair $(X, Y) \in V \times V$ if $S \Rightarrow^* \alpha X \beta \Rightarrow^* \alpha\gamma Y \delta\beta \Rightarrow^* w$. $TS^+ \subseteq L(G)$ satisfies derivable pair coverage (w.r.t. G) if each pair (X, Y) with $X \Rightarrow^* \mu Y \nu$ for some μ, ν is covered by a word $w \in TS$.

We can further generalize CDRC to explore deeper derivations by simultaneously covering all possible combinations of A-productions in all occurrences of non-terminal A. This expansion is realized by a *breadth-first coverage* that generates tests according to a simultaneous derivation relation $\Rrightarrow$ where $X_1 \ldots X_n \Rrightarrow \gamma_1 \ldots \gamma_n$ if there exists a rule $X_i \to \gamma_i \in P$ for all $X_i \in N$ and $\gamma_i = X_i$ for all $X_i \in T$. We denote its k-fold repetition $\Rrightarrow^k$ by bfs_k because it amounts to k "breadth-first rounds" of rule applications.

Definition 2.3.6 (breadth-first coverage)**.** Let $G = (N, T, P, S)$ be a grammar and TS^+ be a positive test suite for G. TS^+ satisfies bfs_k coverage (w.r.t. G) if for each $X \in V$ and $\gamma \in V^*$ with $X \Rrightarrow^k \gamma$, there is a word $w \in TS^+$ such that $S \Rightarrow^* \alpha X \beta \Rightarrow^* \alpha\gamma\beta \Rightarrow^* w$.

For example, consider the example grammar $G = (\{S, A, B\}, \{a, b\}, \{S \to AB, A \to a \mid b, B \to a \mid b\}, S)$, a test suite $TS^+_{cdrc} = \{aa, bb\}$ and $k = 2$. TS^+_{cdrc} satisfies CDRC, but does not cover all possible combinations of expanding A and B. A and B get fully multiplied out in bfs_2 giving yield to the test suite $TS^+_{bfs} = \{aa, bb, ab, ba\}$.

For smaller values of k, bfs_k test suites are comparable both in size and achieved system code coverage to those generated using k-step, but with increasing values of k, we are much more likely to run into *combinatorial explosion* faster with bfs_k than k-step.

The next coverage criterion also considers a pair $(X, Y) \in V$, but only in a local context. It ensures that every adjacent pair of (X, Y) is covered in a test suite TS^+.

Definition 2.3.7 (adjacent pair coverage)**.** Let $G = (N, T, P, S)$ be a grammar, $V = N \cup T$ and TS^+ be a positive test suite for G. A word $w \in TS^+$

covers an adjacent pair $(X, Y) \in V \times V$ iff $S \Rightarrow^* \alpha XY\beta \Rightarrow^* w$. TS^+ satisfies adjacent pair coverage (w.r.t G) iff each (X, Y) with $Y \in \text{follow}(X)$ is covered by a word $w \in TS^+$.

Random test suites

As is common in grammar-based fuzzing, we also use random derivations. More specifically, we construct a random subset of the $\Rrightarrow^k$ derivations, which allows us to explore longer derivations than full bfs_k. After k iterations, we replace the unexpanded non-terminals with their minimal yield, as in the generic cover algorithm. Note that this setup may introduce some bias, in particular towards the rules used in the yield construction.

Negative test suites

For syntactically invalid test suites, we construct negative test suites using two mutation-based algorithms (word- and rule-mutation) described in [130]. The basic idea is to induce test suites with *poisoned pairs* using string edit-distance operators such as symbol insertion, deletion, substitution and swapping as mutation operators. A poisoned pair is any pair of symbols (X, Y) that can never occur next to each other in any derivation from the start symbol, i.e., iff $X \notin \text{precede}(Y)$ or iff $Y \notin \text{follow}(X)$. We use $PP(G)$ to denote the set of all poisoned pairs in a grammar G. Both algorithms guarantee test suites with a single and well-defined error.

Word Mutation. Word mutation algorithms take as input a positive reference test suite TS^+ and systematically introduce poisoned pairs to each test case $T \in TS^+$. This idea can be implemented by applying a family M of *mutation operators* to each test in a given positive test suite TS^+, i.e., $TS^- = \{m(w) \mid m \in M, w \in TS^+\}$.

Proposition 2.3.8 (Damerau-Levenshtein mutations)**.** Let G be a grammar. Then:

1. *(token deletion)* If $uabcv \in \mathcal{L}(G)$ and $(a, c) \in PP(G)$, then $uacv \notin \mathcal{L}(G)$.

2. *(token insertion)* If $uacv \in \mathcal{L}(G)$, $b \in T$, and either $(a, b) \in PP(G)$ or $(b, c) \in PP(G)$, then $uabcv \notin \mathcal{L}(G)$.

3. *(token substitution)* If $uabcv \in \mathcal{L}(G)$, $d \in T$, and either $(a, d) \in PP(G)$ or $(d, c) \in PP(G)$, then $uadcv \notin \mathcal{L}(G)$.

4. *(token transposition)* If $uabcdv \in \mathcal{L}(G)$, and either $(a, c) \in PP(G)$ or $(c, b) \in PP(G)$ or $(b, d) \in PP(G)$, then $uacbdv \notin \mathcal{L}(G)$.

For any positive test suite TS^+ we denote by $DL(TS^+)$ the negative test suite that results from applying to all words from TS^+ all token mutations at all positions that satisfy the conditions of Proposition 2.3.8; we also call this set the *Damerau-Levenshtein mutants* of TS^+. We can construct $DL(TS^+)$ with a simple *token-stream fuzzing* algorithm: we iterate token-by-token over each word in TS^+ and check whether the conditions of Proposition 2.3.8 are satisfied at the current position; if so, we output the corresponding mutant.

Consider for example the following grammar G_{arith} for arithmetic expressions:

$$E \rightarrow E \mathbin{\texttt{*}} E \mid E \,\texttt{/}\, E \mid E \,\texttt{+}\, E \mid E \,\texttt{-}\, E \mid \texttt{(}\, E \,\texttt{)} \mid \texttt{-}\,?\,\texttt{num} \mid \texttt{id}$$

Note that the minus sign is not part of the `num` token but a unary operator that is only applicable to `num`'s; consequently, **`x*-1`** is a valid word, but **`x*+1`** or **`x*-y`** are not.

We have follow(**(**) $=$ {**(**, **-**, num, id} (with the same for *****, **/**, **+**, and **-**), and follow(**)**) $=$ { *****, **/**, **+**, **-**, **)** } (with the same for `num` and `id`). Now consider the valid word **`x*1`** $\in TS^+$. Since **+** $\notin$ follow(*****) (and hence (**+**, *****) $\in PP(G_{arith})$), we know that inserting a **+** after the ***** produces a syntax error. Similarly, since $(\texttt{id}, \texttt{num}) \in PP(G_{arith})$, we know that deleting the ***** produces a syntax error as well.

The conditions stated in Proposition 2.3.8 are sufficient for a syntax error, but not necessary. The main limitation comes from the fact that the conditions only check the local context, and do not take the derivation into account. For example, since an `id` can follow a **-**, the positive test **`x*-1`** is not mutated into **`x*-y`** $\notin \mathcal{L}(G_{arith})$. Similarly, **`(x)`** is mutated into **`(x)(`** (because **(** $\notin$ follow(**)**)), but not into **`(x))`**.

Rule Mutation. Rule mutation systematically modifies the rules of a grammar G so that every derivation that uses such a modified rule produces a poisoned pair and thus induces a negative test case.

We can then formulate conditions under which we allow a symbol mutation. The idea here is to check for "boundary incursions" over the designated position of the modified item, i.e., to check whether any token that can follow (precede) the left (right) set of the modified item is also in its right (left) set. If that is not the case, then any yield must contain a poisoned pair straddling the designated position, and hence cannot be part of a word in the language.

Definition 2.3.9 (symbol deletion mutation)**.** Let $p = A \rightarrow \alpha \bullet X\beta$ be an item in $P^\bullet$. If either

$$\text{follow}(\text{left}(A \rightarrow \alpha \bullet \beta)) \cap \text{right}(A \rightarrow \alpha \bullet \beta) = \emptyset$$

or

$$\text{left}(A \rightarrow \alpha \bullet \beta) \cap \text{precede}(\text{right}(A \rightarrow \alpha \bullet \beta)) = \emptyset$$

then the deletion of X from p at the designated position yields the *mutated production* $p' = A \rightarrow \alpha\beta$.

Definition 2.3.10 (symbol insertion mutation)**.** Let $p = A \to \alpha \bullet \beta$ be an item in $P^\bullet$, and $X \in V$ be a symbol. If either

$$\text{follow}(\text{left}(A \to \alpha \bullet X\beta)) \cap \text{right}(A \to \alpha \bullet X\beta) = \emptyset$$

or

$$\text{left}(A \to \alpha \bullet X\beta)) \cap \text{precede}(\text{right}(A \to \alpha \bullet X\beta) = \emptyset$$

then the insertion of X into p at the designated position yields the *mutated production* $p' = A \to \alpha X\beta$.

If X is nullable, then both conditions of Definitions 2.3.9 and 2.3.10 are false by construction. Hence, we never delete or insert a nullable symbol.

Definition 2.3.11 (symbol substitution mutation)**.** Let $p = A \to \alpha \bullet X\beta$ be an item in $P^\bullet$, and $Y \in V$. If either

$$\text{follow}(\text{left}(A \to \alpha \bullet Y\beta)) \cap \text{right}(A \to \alpha \bullet Y\beta) = \emptyset$$

or

$$\text{left}(A \to \alpha \bullet Y\beta) \cap \text{precede}(\text{right}(A \to \alpha \bullet Y\beta)) = \emptyset$$

then the substitution of X by Y in p at the designated position yields the *mutated production* $p' = A \to \alpha Y\beta$.

We use the notation $p \rightsquigarrow p'$ to denote that a production p' has been constructed from p by mutation with any of the mutation operations above.

2.4 Spectrum-Based Fault Localization

Spectrum-based fault localization (SFL) is a heuristic, coverage-based, dynamic method used to identify likely faulty program elements in programs. Program elements are typically statements but can also be methods, control-flow graph edges, classes, etc. We can summarize the process in three steps:

1. Execute the SUT over a test suite, collect coverage for each individual test case.
2. Correlate the coverage with the test outcomes and aggregate it into the spectrum.
3. Compute suspiciousness scores for elements and rank. Higher scores and thus ranks indicate higher bug likelihood.

Formally, a program spectrum is a representation of the execution information for the SUT's individual program elements; most SFL methods use (binary) statement coverage, i.e., record whether a statement has been executed for a given test or not.

Table 2.4: Example program spectrum.

		t1	*t2*	*t3*	*t4*	*t5*	*t6*		
	input *a*	2	-2	0	5	2	3		
	input *b*	3	5	0	8	2	7		
	input *op*	sum	sum	sum	mean	mean	mean		
	expected output	5	3	0	6	2	5		
	observed output	-1	-7	0	6	2	5		
line	*program*							$\mathcal{S}$	$\mathcal{R}$
1	`read(a);`	✗	✗	✓	✓	✓	✓	0.33	2
2	`read(b);`	✗	✗	✓	✓	✓	✓	0.33	2
3	`read(op);`	✗	✗	✓	✓	✓	✓	0.33	2
4	`if(op == "sum")`	✗	✗	✓	✓	✓	✓	0.33	2
5	`res = a - b; //fault`	✗	✗	✓				0.67	1
6	`else if(op == "mean")`				✓	✓	✓	0	7
7	`res = (a + b)/2;`				✓	✓	✓	0	7
8	`print(res);`	✗	✗	✓	✓	✓	✓	0.33	2

Table 2.4 illustrates an application of SFL methods to pinpoint a fault in the 8-line program listed in column *program*. The program takes as input two integer values *a* and *b*, and a third argument, which is the operation to perform on *a* and *b*. There are two operations to perform, where `sum` computes the sum of *a* and *b* and *mean* computes the average of *a* and *b*. The fault (or bug) is in line 5. We execute the program over six tests, t_1-t_6 with their inputs, observed and expected outputs shown in the first block of Table 2.4. Tests *t*1 and *t*2 are failing tests and execute the same statements as the passing test *t*3. The second block shows statement-level program spectra, suspiciousness scores and ranks for the faulty example program. ✓ (resp. ✗) indicates that a statement has been executed in a passing (resp. failing) test case. $\mathcal{S}$ shows the actual suspiciousness score assigned to each statement while $\mathcal{R}$ is the ranking. In this example, the suspiciousness score is simply computed by dividing the number of failing tests that executed a statement by the total number of tests. We ignore the number of times a statement was executed in passing tests. More advanced formulas are presented in Section 2.4.2.

In this example, the faulty line (i.e., *line 5*) gets assigned the highest score and thus highest rank. When debugging, the programmer uses the SFL report and starts examining statements in the descending order of their ranks. In this particular case, the first statement to examine contains a fault and can be fixed accordingly. Another phenomenon that we can observe from the example, is that multiple statements are assigned the same scores, and thus have the same ranks. These ties are prevalent in SFL. We use the mid-rank tie breaking mechanism to assign ranks under column $\mathcal{R}$.

The next sections introduce *basic counts* and *ranking metrics* which are the

Table 2.5: SFL ranking metrics

Ranking metric	$score(e)$
Tarantula [69]	$\frac{\frac{ef(e)}{ef(e)+nf(e)}}{\frac{ef(e)}{ef(e)+nf(e)}+\frac{ep(e)}{ep(e)+np(e)}}$
Ochiai [115]	$\frac{ef(e)}{\sqrt{(ef(e)+nf(e))(ef(e)+ep(e))}}$
Jaccard [26]	$\frac{ef(e)}{ef(e)+nf(e)+ep(e)}$
DStar [157]	$\frac{ef(e)^n}{nf(e)+ep(e)}$

essential SFL ingredients used for the computation of suspiciousness scores.

2.4.1 Basic Counts

The spectra for the individual tests are correlated with the test outcomes and aggregated into four basic counts for each individual program element e: $ep(e)$ and $ef(e)$ are the number of passed and failed tests, respectively, in which e is executed, while $np(e)$ and $nf(e)$ are the number of passed and failed tests, respectively, in which e is *not* executed. In the example program in Table 2.4, if e is the faulty statement in line 5, then its basic count values are: $ef(e) = 2$, $ep(e) = 1$, $nf(e) = 0$, and $np(e) = 3$.

2.4.2 Ranking Metrics

SFL methods use the basic counts to compute for each program element a *suspiciousness score*; elements that have a higher score are ranked higher and are seen as more likely to contain a bug. The methods which are traditionally called *ranking metrics*, even though they are not proper metrics, differ in the formulas used for the score computation. In this thesis, we use four ranking metrics that are widely used in SFL. Table 2.5 shows their definitions. Note that Tarantula [69] is the only metric that uses the number of passed tests $np(e)$ in which an element e is *not* executed. Note also that DStar [157] is parameterized over the exponent n; here, we use the most common value $n = 2$. DStar becomes undefined for an element e if it is executed *only* in failing test cases. We assign a maximal score in this case, since we consider e to be the most suspicious element.

The metrics become undefined or degenerate and rank all elements equally if the test suite does not contain at least one *failing* test; similarly, the metrics become undefined or simply rank the elements by occurrence count if the test suite does not contain at least one *passing* test. We therefore assume in our work that test suites indeed contain at least one failing and one passing test.

Ranking metrics can assign the same score to different elements. For ranking purposes, we need to resolve such *ties* and assign a well-defined rank to all tied elements. Here we use the mid-point of the range of elements with the same score; the assigned rank then indicates how many elements a user is expected to inspect before they find the fault if elements with the same score are inspected in random order. A more pessimistic variant uses the lowest possible rank that is consistent with the scores; this would result in a worst-case estimate of the number of elements to be inspected.

Chapter 3

Rule-Level Fault Localization

In this chapter, we illustrate and formalize the notion of grammar spectra which underlies our approach, at the level of grammar rules; we consider a more fine-grained localization at the level of rule items in Chapter 4. In the following we assume that the SUT is a CFG $G = (N, T, P, S)$; we assume that this is implemented faithfully in a parser since we are trying to localize errors in the grammar, not in the parser's implementation.

We first illustrate our method with a worked example based on the toy grammar $G_{\mathcal{T}oy}$ in Section 3.1. We fix formal definitions of rule spectra in Section 3.2.

In order to collect the grammar spectra, we typically modify the parser to log which rules it has applied. The nature of the modifications depends on the general parsing technology and the specifics of the parser; we describe the modifications we made to the JavaCC, ANTLR, and CUP parser generators in order to generate parsers with the required logging extensions in Sections 3.3 and 3.4. In Section 3.5, we describe how rule spectra can be extracted directly from individual test cases derived from the grammar in applications where we cannot obtain a standalone parser derived from the grammar.

We then demonstrate the efficacy of our baseline fault localization method through a series of experiments. We first evaluate our approach over grammars with seeded faults (see Section 3.6.1 and 3.6.2) because the ground-truth is required for evaluation and because it is much easier to produce a large experimental basis. We then evaluate it using grammars which contain real and multiple faults in Section 3.6.3. The last experiment (see Section 3.6.4) addresses scalability questions of our approach by using a large, production-level SQLite grammar to find out how it deviates from the grammar implemented by a fully-fledged SQLite system.

```
prog  → program id = block .
block → { decls stmts } | { decls } | { stmts } | { }
decls → decl ; decls | decl ;
decl  → var id : type
type  → bool | int
stmts → stmt ; stmts | stmt ;
stmt  → sleep
        | if expr then stmt
        | if expr then stmt else stmt
        | while expr do stmt
        | id = expr
        | block
expr  → expr = expr | expr + expr | ( expr ) | id | num
```

Figure 3.1: An example grammar G_{Toy}.

```
1  program x = {x = (x);}.
2  program x = {x = x + x;}.
3  program x = {x = x;}.
4  program x = {x = x = x;}.
5  program x = {x = 0;}.
6  program x = {if x then sleep;}.
7  program x = {if x then sleep else sleep;}.
8  program x = {sleep; sleep;}.
9  program x = {sleep;}.
10 program x = {var x:bool; sleep;}.
11 program x = {var x:bool; var x bool;}.
12 program x = {var x:bool;}.
13 program x = {var x:int;}.
14 program x = {while x do sleep;}.
15 program x = {{};}.
16 program x = {}.
```

Figure 3.2: Test suite satisfying *rule*-coverage for G_{Toy}.

3.1 Worked Example

We illustrate our method with a worked example based on the toy grammar G_{Toy} shown in Figure 3.1 and a corresponding test suite satisfying *rule*-coverage, shown in Figure 3.2. Note that G_{Toy} is not an $LL(k)$ grammar but in a form that is suitable for LR parsers. We assume that the grammar developer has made mistakes in formulating the statement rules in G_{Toy}, requiring the **else**-branch to be present and restricting the body of **while**-loops to be

Table 3.1: Rule spectra, suspiciousness scores, and ranks for the faulty grammar version G'_{Toy} and *rule* test suite.

rule	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	*ep*	*np*	*ef*	*nf*	$\mathcal{T}$		$\mathcal{O}$		$\mathcal{J}$		$\mathcal{D}$	
prog:1	✓	✓	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	11	0	2	0	0.5	8	0.4	6	0.1	7	0.3	6
block:1										✓							1	13	0	2	0	-	0	-	0	-	0	-
block:2											✓	✓	✓				3	11	0	2	0	-	0	-	0	-	0	-
block:3	✓	✓	✓	✓	✓	✗	✓		✓					✗	✓	✓	9	5	2	0	0.6	6	0.4	4	0.2	4	0.4	4
block:4																✓	1	13	0	2	0	-	0	-	0	-	0	-
decls:1											✓						1	13	0	2	0.0	-	0.0	-	0.0	-	0.0	-
decls:2										✓		✓	✓				3	11	0	2	0.0	-	0.0	-	0.0	-	0.0	-
decl:1										✓	✓	✓	✓				4	10	0	2	0.0	-	0.0	-	0.0	-	0.0	-
type:1										✓	✓	✓					1	10	0	2	0.0	-	0.0	-	0.0	-	0.0	-
type:2													✓				1	10	0	2	0.0	-	0.0	-	0.0	-	0.0	-
stmts:1								✓									1	13	0	2	0	-	0	-	0	-	0	-
stmts:2	✓	✓	✓	✓	✓	✗	✓		✓	✓				✗	✓		9	5	2	0	0.6	6	0.4	4	0.2	4	0.4	4
stmt:1						✗	✓	✓	✓	✓							4	10	1	1	0.6	5	0.3	7	0.2	6	0.2	7
stmt:2						✗	✓										**1**	**13**	**1**	**1**	**0.9**	**2**	**0.5**	**3**	**0.3**	**2**	**0.5**	**3**
stmt:3														✗			**0**	**14**	**1**	**1**	**1.0**	**1**	**0.7**	**1**	**0.5**	**1**	**1.0**	**1**
stmt:4	✓	✓	✓	✓	✓												5	9	0	2	0.0	-	0.0	-	0.0	-	0.0	-
stmt:5															✓		1	13	0	2	0.0	-	0.0	-	0.0	-	0.0	-
expr:1				✓													1	13	0	2	0.0	-	0.0	-	0.0	-	0.0	-
expr:2		✓															1	13	0	2	0.0	-	0.0	-	0.0	-	0.0	-
expr:3	✓																1	13	0	2	0.0	-	0.0	-	0.0	-	0.0	-
expr:4	✓	✓	✓	✓		✗	✓							✗			5	9	2	0	0.7	3	0.5	2	0.3	3	0.8	2
expr:5					✓												1	13	0	2	0.0	-	0.0	-	0.0	-	0.0	-

blocks:

$$
\begin{aligned}
\mathit{stmt} \rightarrow\ & \mathtt{\textbf{sleep}} \\
|\ & \mathtt{\textbf{if}}\ \mathit{expr}\ \mathtt{\textbf{then}}\ \mathit{stmt}\ \mathtt{\textbf{else}}\ \mathit{stmt} \\
|\ & \mathtt{\textbf{while}}\ \mathit{expr}\ \mathtt{\textbf{do}}\ \mathit{block} \\
|\ & \mathtt{id} \mathtt{=} \mathit{expr} \\
|\ & \mathit{block}
\end{aligned}
$$

We call this faulty version G'_{Toy}.

We create a parser for the faulty version G'_{Toy} and run it over the test suite to collect the grammar spectra shown in Table 3.1. Rules are denoted by the non-terminal name and the index of the corresponding alternative. ✓(resp. ✗) indicates execution in a passing (resp. failing) test cases. Faulty rules are shown in bold. We finally compute the scores according to the four ranking metrics shown in Table 2.5 and rank the rules; note that ties could in principle also result from different execution counts, although this is not the case here. $\mathcal{T}$ denotes scores and ranks computed using Tarantula, $\mathcal{O}$ using Ochiai, $\mathcal{J}$ using Jaccard and $\mathcal{D}$ using Dstar. Ranks are only shown for rules with non-zero scores.

All four metrics pinpoint the faulty **`while`**-rule (i.e., *stmt*:3) as the unique most suspicious rule. This is hardly surprising because the rule is only applied in one test case and that one is failing. The second fault is more difficult to localize because the faulty rule is executed in both failing and passing test cases. Tarantula and Jaccard rank it second while Ochiai and DStar rank it third, in both cases behind the rule $\mathit{expr} \rightarrow$ `id` that is applied in most derivations.

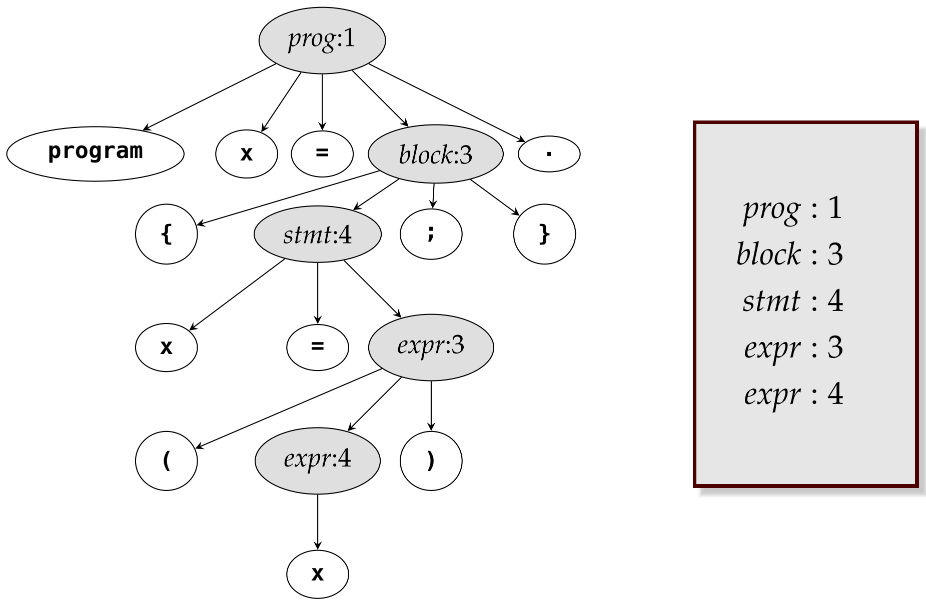


Figure 3.3: An illustration of spectrum collection using a parse tree (left) of w = `program x = {x = (x);} .`. Traversal of the non-leaf nodes gives the set $R \subseteq P$ shown on the right.

If we inspect the rules in rank order and resolve ties by picking rules arbitrarily, we have on average to look at two rules (i.e., 9.1% of all rules) before we find both faults using Tarantula or Jaccard, and three rules (or 13.6%) using Ochiai or DStar.

3.2 Rule Spectra

We can informally define a *rule spectrum* as the set of all rules $R \subseteq P$ that are applied when a word w in the test suite is parsed. For example, consider the parse tree (see Section 2.1.1) in Figure 3.3 resulting from parsing the input w = `program x = {x = (x);} .` (i.e., testcase #4) using G_{Toy} as shown in Figure 3.1. We add to R all non-leaf nodes of the parse tree and get R = {*prog*:1, *block*:3, *stmt*:5, *expr*:3, *expr*:4}. However, which rules are actually applied depends on the nature of the parser, in particular when it rejects w. We therefore first formalize our intuition in terms of generic derivations, and then concretise it in Sections 3.3 and 3.4 for LL and LR parsers.

For accepted words the formal definition of rule spectra directly follows our intuition; note that a word $w \in L(G)$ can induce multiple rule spectra iff G is ambiguous.

Definition 3.2.1 (positive rule spectrum). If $\Delta = S \Rightarrow_{p_1} \alpha_1 \Rightarrow_{p_2} \alpha_2 \Rightarrow_{p_3} \cdots \Rightarrow_{p_n} \alpha_n = w$, then $R = \bigcup_i \{p_i\} = \text{rules}(\Delta)$ is called a *positive rule spectrum* for w.

Note that $\Delta = S \Rightarrow^* w$ is a maximally viable $|w|$-prefix bounded derivation (see Section 2.1.2) for w, since $w \in L(G)$. We can use this observation as a starting point for the definition of spectra in cases where $w \notin L(G)$: we consider all rules of a maximally viable k-prefix bounded derivation Δ for w, i.e., all rules that have been applied to the left of the error position. However, in contrast to the positive case, where there is no frontier, in the negative case we must also consider the frontier rules because the error could be either in the rule in Δ that introduced the frontier, or in any of Δ's frontier rules themselves. Consider for example the situation where we are trying to parse $w =$ `program x = {while x do sleep;}.` (i.e., test case #14) with the faulty variant G'_{Toy}. This fails at **`sleep`** because G'_{Toy} expects *body* to be a *block* and **`sleep`** is a *stmt*. The maximal viable prefix u of w is thus $u =$ `program x = {while x do`, with $v =$ `{};}.`, a possible right completion so that $uv \in L(G'_{Toy})$. One (and in this case the only possible) maximal k-prefix bounded derivation with $k = 7$ for w is

$$\begin{aligned}
\Delta = prog &\Rightarrow_{prog:1} \texttt{\textbf{program} id=}\ block\ \texttt{.}\\
&\Rightarrow_{block:3} \texttt{\textbf{program} id= \{}\ stmts\ \texttt{\} .}\\
&\Rightarrow_{stmts:2} \texttt{\textbf{program} id= \{}\ stmt\ \texttt{\} .}\\
&\Rightarrow_{stmt:3} \texttt{\textbf{program} id= \{ \textbf{while}}\ expr\ \texttt{\textbf{do}}\ block\ \texttt{\} .}\\
&\Rightarrow_{expr:4} \texttt{\textbf{program} id= \{ \textbf{while} id \textbf{do}}\ block\ \texttt{\} .}
\end{aligned}$$

with the frontier symbol *block* and $\text{rules}(\Delta) = \{prog{:}1, block{:}3, stmts{:}2, stmt{:}3, expr{:}4\}$. We can presume that the fault is in *stmt*:3 (where it was indeed introduced) but with the information at hand we cannot rule out that any of the *block*-rules is at fault–for example, *block*:4 could have been meant to be of the form $block \rightarrow$ **`sleep;`** .

SFL resolves this uncertainty by computing the suspiciousness scores over many spectra, but this requires us to include all possible fault locations in the (negative) spectra. Hence, we get the following formal definition:

Definition 3.2.2 (negative rule spectrum)**.** Let $w = uv \notin L(G)$ with maximal viable k-prefix u, and $S \Rightarrow_{p_1} \alpha_1 \Rightarrow_{p_2} \alpha_2 \Rightarrow_{p_3} \cdots \Rightarrow_{p_n} uX\alpha$ be a maximally viable k-prefix bounded derivation for w with frontier X. Then $R = \bigcup_i \{p_i\} \cup \text{closure}(P_X)$ is called a *negative rule spectrum* for w.

Note that this is a "loose" definition in the sense that a $w \notin L(G)$ may induce several different negative spectra, since there can be different maximally viable k-prefix bounded derivations for w, with different right completions and frontiers. Note also that Definition 3.2.2 can be modified to subsume the "positive" and "negative" definitions, but we keep the cases apart for simplicity.

3.3 Spectra for Recursive-descent LL Parsers

Since LL parsers build the maximally viable derivation Δ top-down, left-to-right, every expansion step $\alpha_i \Rightarrow_{p_i} \alpha_{i+1} \in \Delta$ adds the corresponding rule p_i to the spectrum, whether the derivation ends in success or not. In a recursive-descent parser, each rule is implemented by its own parsing function, and each derivation step corresponds to a call to one of these functions. Hence, a rule spectrum includes the set of parse functions entered at least once in a derivation. This holds for both valid and invalid words, but for an invalid word (i.e., $w' \notin L(G)$), there is at least one parse function that was entered and never exited successfully. However, for an invalid word, this set of parse functions does not necessarily include the frontier rules. Consider for example an LL(1)-grammar with the rules $A \rightarrow \alpha B \gamma$ and $B \rightarrow \beta_1 \mid \ldots \mid \beta_n$, and a maximally viable derivation $S \overset{k}{\Rightarrow}{}^{*}_{w} uA\gamma$ for $w = uav$ with $a \notin \text{first}(B)$. The parse function for A checks a against all $b \in \text{first}(\beta_i)$ before calling the parse function for the respective alternative of B, but since $a \notin \text{first}(B)$, none of them will actually be called and added to the spectrum. We must therefore modify the parser's error handling routines to add the corresponding frontier rules explicitly. Note also that an LL(1)-parser only explores a single maximally viable derivation Δ for each $w \notin L(G)$ and we therefore only get a single negative spectrum. For LL(k)-parsers with $k > 1$, the derivation may not even be maximal because the parser may detect the syntax error before actually reaching the error location. The collected rules may thus under-approximate the negative spectra, especially if the definition of frontier rules is not adapted properly to sequences of grammar symbols.

JavaCC. In principle, spectrum collection is a simple logging task that can be implemented easily. For JavaCC, which generates straightforward recursive-descent LL(k) parsers from a given grammar specification, we found it indeed relatively easy to modify the generator source code itself, and to add code for the grammar spectra extraction task as a side effect to parser generation. This allows us full control and management of individual rule alternative resolution. JavaCC offers advanced top-down parsing features like localized lookahead configurations, i.e., users can explicitly set a value of $k > 1$ for portions of a grammar that are not within the LL(1) parsing capabilities, thereby making the generated parser LL(k) for only those portions. Our evaluation, however, focuses on the default LL(1) configuration.

ANTLR. ANTLR, in contrast, generates adaptive LL(*) parsers that use unbounded look-ahead, which complicates the structure of the parse functions, and thus in turn the spectra extraction. ANTLR provides runtime support to automate collection of grammar spectra through tree walkers (via generated listener and visitor interfaces). Figure 3.4 shows an overridden listener method that enables spectrum collection using ANTLR's de-

```
1 @Override
2 public void enterEveryRule(ParserRuleContext ctx) {
3   /* index of matched rule (i.e., non-terminal) */
4   int index = ctx.getRuleIndex();
5   /* alternative of matched rule */
6   int alternative = ctx.getAltNumber();
7   String[] rules = parser.getRuleNames();
8   spectrum.add(rules[index]+":"+alternative);
9 }
```

Figure 3.4: ANTLR tree walker for spectrum collection

fault error recovery strategy. However, this only works when ANTLR actually completes the parse and builds a tree. ANTLR's error recovery strategy allows it to do so in most cases, but this means that rules used after any error recovery will be mis-classified in the spectrum.[1]

As an alternative, we therefore turned off error recovery, forcing the parser to bail out without returning a parse tree when it encounters the first syntax error. We then used aspect-oriented programming [73] to track all calls to ANTLR's internal `enterOuterAltNum` method (see Figure 3.5) that sets the rule and alternative fields in the tree. In this way, we can (in principle) extract spectra conforming to Definitions 3.2.1 and 3.2.2. In practice, however, we encountered two problems that can cause the extracted spectra to be wrong. First, ANTLR's adaptive LL(*) parsing mechanism can cause it to raise a syntax error with unbounded lookahead (typically a `no viable alternative` error) without actually entering the parse function for the corresponding rule, so that frontier rules may be missing. Second, ANTLR's tracking of rule applications is wrong[2] for grammars that contain left-recursive rules.

The adaptive LL(*) parsing mechanism also makes it difficult to track which tokens have been seen; our attempts at an extension to collect item spectra were brittle and unreliable, so that we do not use ANTLR in our evaluation of item-level localization (see Chapter 4).

3.4 Spectra for Table-driven LR Parsers

In table-driven LR parsing, there are no parse functions that could be tracked. Instead, a small parser core interprets the LR tables and maintains an explicit state stack, where each state represents a set of items $\{A \to \alpha \bullet \beta\}$. The application of a rule is then carried out by the two main operations

[1]It should be noted this requires the compilation option `-DcontextSuperClass=RuleContextWithAltNum` in order to get the right alternative for a matched rule. The call to `getAltNumber()` (in line #5) returns the default value 0 otherwise.

[2]i.e., the call to `enterOuterAltNum` is missing, see the open issue #2222

```
pointcut enterRuleAlt(ParserRuleContext ctx, int altNum, Parser
    parser) :
    call(void Parser.enterOuterAlt(ParserRuleContext, int)) &&
        args(ctx, altNum) && target(parser);

before(ParserRuleContext ctx, int altNum, Parser parser) :
    enterRuleAlt(ctx,altNum, parser) {
        String[] rules = parser.getRuleNames();
        spectrum.add(rules[ctx.getRuleIndex()]+":"+altNum);
    }
```

Figure 3.5: Parser aspect that tracks internal calls to `enterOuterAltNum`.

on the stack: shift and reduce. For a valid word, we can rely simply on the reduce operation to extract the rule spectrum, since all rule applications end successfully with a reduction. For invalid words, we use the reduce operation to capture the rules applied fully to the left of the error position, i.e., at the viable prefix, but we also need to capture the partially applied rules and the frontier rules. These are both reflected in the states that remain on the stack when the parser encounters an error. The frontier rules are by construction given by the *non-kernel* items of the state at the top of the stack, while each kernel item $A \to \alpha \bullet \beta$ at the top of the stack represents a partially applied rule, with the yield of each β_i describing the prefixes of possible right continuations v' (see Definition 3.2.2).

Table 3.2 shows CUP's parse stack when it uses the faulty grammar G'_{Toy} to parse the test case `program x = { while x do sleep; }.` and encounters the syntax error at `sleep`. We get *expr*:4 as result of a complete rule application at the reduce operation in state 25. The non-kernel items at the top of the stack give us the frontier rules {*block*:1, *block*:2, *block*:3, *block*:4} while the single kernel item gives us *stmt*:3.

Furthermore, partially applied rules are associated with the kernel items from states further down on the stack; we therefore traverse the stack and extract these. Note that we do not extract any rules that are associated with non-kernel items only because these rules have not been applied even partially, as the designated position is at the beginning of the rule. In the example, we get the rules {*prog*:1, *block*:1, *block*:2, *block*:3, *block*:4, *stmt*:3, *expr*:1, *expr*:2} from this stack traversal; note that the four *block*-rules are logged again.

Note also that sometimes, some rules are extracted from kernel items $A \to \alpha \bullet \beta$ even though they cannot be applied in a *maximally* viable k-prefix bounded derivation, such as in this example, the two *expr* rules in state 50. Here, our implementation is an over-approximation of Definition 3.2.2 but we show in our evaluation that this does not necessarily lead to poor localization performance. The extraction of rule spectra that matches the definition precisely would require some extra bookkeeping, as it requires further

Table 3.2: CUP parse stack when encountering the syntax error while parsing the test case `program x = {while x do sleep;}.` with G'_{Toy}. State 25 (shown in gray) is popped off the stack after reduction.

state	corresponding kernel items	corresponding non-kernel items
2	$prog \rightarrow \mathtt{\textbf{program}} \bullet \mathtt{id} = block\ \mathbf{.}$	
3	$prog \rightarrow \mathtt{\textbf{program}}\ \mathtt{id} \bullet = block\ \mathbf{.}$	
4	$prog \rightarrow \mathtt{\textbf{program}}\ \mathtt{id} = \bullet\ block\ \mathbf{.}$	$block \rightarrow \bullet \{\ decls\ stmts\ \}$
		$block \rightarrow \bullet \{\ decls\ \}$
		$block \rightarrow \bullet \{\ stmts\ \}$
		$block \rightarrow \bullet \{\ \}$
5	$block \rightarrow \{\ \bullet\ decls\ stmts\ \}$	$decls \rightarrow \bullet\ decls\ decl\ ;$
	$block \rightarrow \{\ \bullet\ decls\ \}$	$decls \rightarrow \bullet\ decl\ ;$
	$block \rightarrow \{\ \bullet\ stmts\ \}$	$decl \rightarrow \bullet\ \mathtt{\textbf{bool}}$
	$block \rightarrow \{\ \bullet\ \}$	$decl \rightarrow \bullet\ \mathtt{\textbf{int}}$
		$stmts \rightarrow \bullet\ stmts\ stmt\ ;$
		$stmts \rightarrow \bullet\ stmt\ ;$
		$stmt \rightarrow \bullet\ \mathtt{\textbf{sleep}}$
		$stmt \rightarrow \bullet\ \mathtt{\textbf{if}}\ expr\ \mathtt{\textbf{then}}\ stmt\ \mathtt{\textbf{else}}\ stmt$
		$stmt \rightarrow \bullet\ \mathtt{\textbf{while}}\ expr\ \mathtt{\textbf{do}}\ block$
		$stmt \rightarrow \bullet\ \mathtt{id} = expr$
		$stmt \rightarrow \bullet\ block$
		$block \rightarrow \bullet \{\ decls\ stmts\ \}$
		$block \rightarrow \bullet \{\ decls\ \}$
		$block \rightarrow \bullet \{\ stmts\ \}$
		$block \rightarrow \bullet \{\ \}$
8	$stmt \rightarrow \mathtt{\textbf{while}} \bullet expr\ \mathtt{\textbf{do}}\ block$	$expr \rightarrow \bullet\ expr = expr$
		$expr \rightarrow \bullet\ expr + expr$
		$expr \rightarrow \bullet\ (\ expr\ )$
		$expr \rightarrow \bullet\ \mathtt{id}$
		$expr \rightarrow \bullet\ \mathtt{num}$
25	$expr \rightarrow \mathtt{id} \bullet$	
50	$stmt \rightarrow \mathtt{\textbf{while}}\ expr \bullet \mathtt{\textbf{do}}\ block$	
	$expr \rightarrow expr \bullet = expr$	
	$expr \rightarrow expr \bullet + expr$	
51	$stmt \rightarrow \mathtt{\textbf{while}}\ expr\ \mathtt{\textbf{do}} \bullet block$	$block \rightarrow \bullet \{\ decls\ stmts\ \}$
		$block \rightarrow \bullet \{\ decls\ \}$
		$block \rightarrow \bullet \{\ stmts\ \}$
		$block \rightarrow \bullet \{\ \}$

stack unwinding to filter out items that cannot be applied in a viable k-prefix bounded derivation. We leave this for future work.

CUP. Since CUP does not provide the required logging capabilities, we modified this to the table interpreter accordingly. For the rule spectra and the plain item spectra (see Chapter 4), we added a simple stack traversal to the table interpreter that replaces the normal error handling routine which may modify the stack. We collect the rules in the items in each state by analyzing CUP's output when it builds the parse tables.

3.5 Synthetic Spectra

Sections 3.3 and 3.4 focused on the traditional, and perhaps the most common use case of grammar development, that of developing grammars to use as an input to compiler-compiler tools. We described how these tools can be extended to extract grammar spectra for fault localization purposes. However, applications such as grammar-based fuzzing take a more general view. Here, the grammar is not implemented by the parser, but seen as an abstract model for the input domain of a system under test. In such applications, it is often much harder to obtain a standalone parser that can be extended to produce spectra for inputs. However, we can take advantage of automatic test suite generation and construct *synthetic spectra* directly from individual test cases derived from the grammar G, i.e, simply the rules used to derive the inputs (see Definition 3.2.1). We also need an oracle $\mathcal{O}$ that is capable of answering *membership queries*, i.e., to confirm whether test cases generated from the grammar are (not) in the same language ($L(\mathcal{O})$) described by the oracle. Here a test case w fails if $w \in L(G)$ but $w \notin L(\mathcal{O})$ or if $w \notin L(G)$ but $w \in L(\mathcal{O})$. This setup is also used in popular grammar learning (see Section 6.5 for more details) algorithms [11, 27].

3.6 Evaluation

We evaluate our method over grammars with seeded faults, as well as real world grammars and student grammars submitted in compiler engineering courses that contain real faults. In this section, we aim to answer the following research questions that we introduced in Section 1.4.

RQ1a: How effective are fault localization techniques based on extracted rule spectra in identifying seeded single faults in grammars?

RQ1b: How effective are fault localization techniques based on synthetic spectra in identifying seeded single faults?

RQ1c: How effective are fault localization techniques in identifying *real* faults in student grammars that possibly contain multiple faults?

RQ1d: How effective does rule-level fault localization remain for large grammars?

3.6.1 Effectiveness in Identifying Seeded Single Faults (RQ1a)

Experimental Setup

Base grammars. We used the grammar of a small artificial programming language called SIMPL as the basis for these experiments. SIMPL was originally designed for use in a second-year computer architecture course at Stellenbosch University, where students were given an LL(1) grammar for SIMPL in EBNF format, and had to manually implement a recursive-descent parser. We manually eliminated the EBNF operators for this grammar by adding new BNF rules, in order to simplify the mutation process.

For ANTLR (v4.7.2), we left-factorized the BNF version and eliminated left-recursive rules to minimize the effect of its adaptive LL(*) parsing mechanism, which can lead to imprecise spectra (see the discussion in Section 3.3). The resulting grammar contains 84 rules, 42 non-terminal symbols, and 47 terminal symbols.

JavaCC (v7.0.5) also requires left-factorization and left-recursion elimination; we used the ANTLR version as a starting point, but used a slightly different representation of inner alternatives. This version contains 93 rules, 49 non-terminal symbols, and 47 terminal symbols.

CUP (v0.11b) requires the elimination of the EBNF extensions; this version was developed independently by a student assistant directly from the EBNF version. It contains 80 rules, 32 non-terminal symbols, and 47 terminal symbols. The three baseline grammars pass all tests in the different test suites (see below).

Mutation operators. We mutated the grammars by blindly applying individual symbol edit operations (deletion, insertion, substitution, and transposition) at every position on the right-hand side of every rule of the grammars. We only applied a single mutation to derive each mutant, to ensure that each mutant contains at most one fault. We discarded all grammar mutants that do not allow the parser generator to produce a parser (e.g., by introducing indirect left-recursion in an ANTLR grammar). This leaves us with 30821 mutants for ANTLR, 36443 mutants for JavaCC, and 26490 mutants for CUP.

Test suites. We then executed each mutant on four different test suites derived from the original EBNF form of the SIMPL grammar. The first two test suites, *rule* and *cdrc*, contain only positive test cases. They are constructed according to the rule and cdrc coverage criteria [82], respectively, and contain 43 and 86 test cases, respectively. Note that *rule* is a proper subset of *cdrc*. *large* is a very large, varied test suite that contains 2964 positive tests and 32157 negative tests. The positive tests are constructed according to four different coverage criteria, (bfs_2, $step_6$, and derivable [150] and adjacent pair coverage, respectively), developed to produce diverse test suites. The negative tests are constructed using token mutation over the *rule* test suite, and using mutation of the rules themselves [130]. *instructor* refers to the test suite the instructor used to grade the student submissions. It comprises 20 (syntactically) positive and 61 negative tests.

Test execution. We consider a grammar mutant killed by a test suite if the generated parser fails on at least one test case; however, we consider a mutant *not* killed if the parser fails on all test cases, because the metrics then become undefined or degenerate, as discussed in Section 2.4.2. For ANTLR, we also considered a mutant as not killed if it requires the application of a left-recursive rule, because the computed grammar spectra are known to be imprecise (see the discussion in Section 3.3).

Spectrum extraction and ranking. For each grammar mutant killed by the test suite we ordered the rules by the scores produced by each of the ranking metrics and computed the mutated rule's predicted rank. We resolved ties using the middle rank, as discussed in Section 2.4.2.

Note that we used the location at which we applied the mutation operation as "true" fault location. However, as described in Section 3.6, the proper "blame assignment" is not always as clear, in particular when the mutations impact the first-sets of rules. This can impact the quality of the predictions.

Synthetic Spectra. We used the same SIMPL grammar mutants for ANTLR to generate *rule*, *cdrc*, and *large* test suites from each mutant in order to answer RQ1b. The ANTLR grammar from which these mutants were derived acts as a "teacher". Our grammar-based test case generator tool gtestr [150] ships with a converter that translates ANTLR grammars to their equivalent gtestr grammars. We discarded mutants with unreachable non-terminal symbols and those that contain symbols for which gtestr cannot compute the yield. This leaves us with 27894 mutants.

Experimental Results

Figures 3.6 to 3.9 show the results of these fault seeding experiments as a series of boxplots. Each boxplot summarizes the ranks predicted by the corresponding metric for the mutated (i.e., faulty) rules, given a specific parsing method and test suite. A perfect prediction would be to rank the faulty rule in the grammar as one, i.e., the most suspicious rule, for all test cases. The boxes show the Q3/Q1 interquartile range of the ranks, i.e., the upper end of the box corresponds to the 75th percentile (i.e., in 75% of the cases the faulty rule is ranked better than the corresponding value on the y-axis) while its lower end corresponds to the 25% percentile. The median is indicated by a dotted line across the box. The "whiskers" extend from the 5th to the 95th percentile. Table 3.3 contains more details.

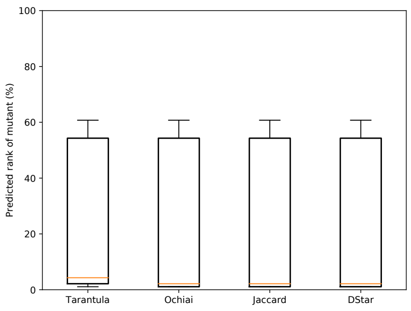
While the details change with the applied parsing technology and ranking metric, and the underlying test suite, the boxplots and Table 3.3 show overall positive results. On average, the metrics rank the faulty rules at $\sim$22% of all rules, with better results for the *large* test suite ($\sim$15%) and worse results for the *instructor* test suite ($\sim$31%). The median is typically at 2.5–5% (except Tarantula under *instructor* at $\sim$17%), and so much smaller than the mean. Hence, in more than half of the cases, the metrics rank the faulty rule as one of the top five most suspicious rules. Furthermore, in 10–40% of the cases they correctly pinpoint the faulty rule, in 15–58% of the cases, the faulty rule is ranked within the top three most suspicious rules and in up to 65% of the cases the faulty rule is ranked within the top five most suspicious rules.

A few high-level observations can be made. First, fault localization works better for JavaCC than for both ANTLR and CUP: for JavaCC we universally achieve lower mean and median values, independent of the test suite and the ranking metric, and typically pinpoint a higher fraction of the observed faults (with Tarantula the only metric with mixed results that are are sometimes better for CUP than the other tools). The #1/#3/#5 values also seem to be in favour of CUP, with ANTLR giving us slightly lower values across the board.

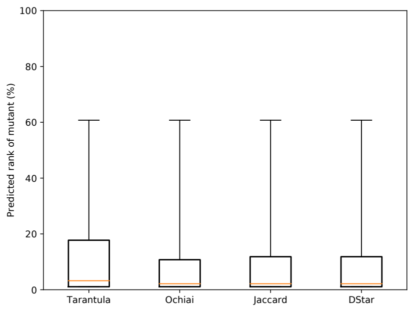
Second, ANTLR's error correction introduces noise into the spectra that compromises the quality of the fault localization. ANTLR with bail-out on error uniformly produces better results than ANTLR* with error correction, although the differences are smaller than between ANTLR and CUP.

Third, the difference between Ochiai, Jaccard, and DStar is negligible, but all three outperform Tarantula. The only exception is for the *large* test suite, where Tarantula produces the tightest interquartile range and the lowest mean (although not the lowest median nor the highest fraction of top-ranked faults). This follows the observation that Tarantula does not particularly get overwhelmed by a high number of failing tests compared to the other three metrics which under such scenarios typically assign the non-faulty rules (mostly dominating ones) executed in most failing tests the highest rank.

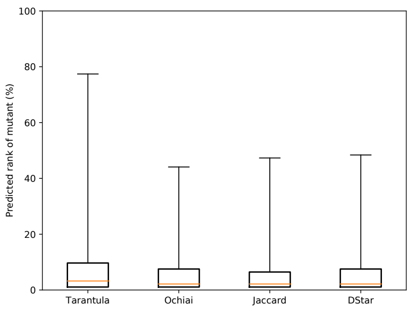
Fourth, the localization performance depends strongly on the size and variance of the test suite. The difference of the results between the *rule* and *cdrc* test suites that contain very similar positive test cases is marginal, despite the fact that *cdrc* includes *rule*. In contrast, both of them induce substantially better results than the manually constructed *instructor* test suite whose size is between both of them. This also indicates that it is hard to manually construct test suites that are well suited for fault localization. *large* has the highest fraction of localized faults compared to the other test suites, and the smallest mean values.



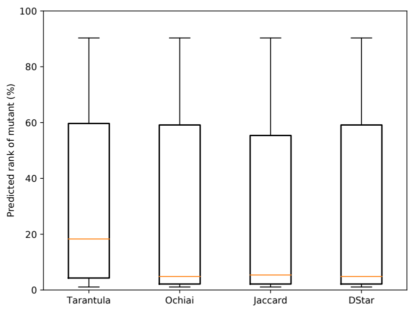
(a) Results for *rule* test suite.



(b) Results for *cdrc* test suite.

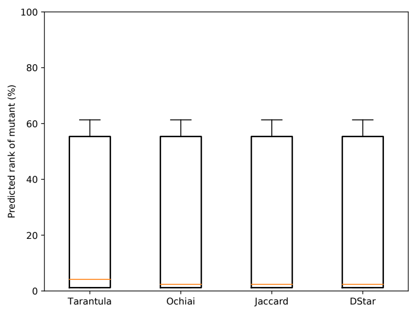


(c) Results for *large* test suite.

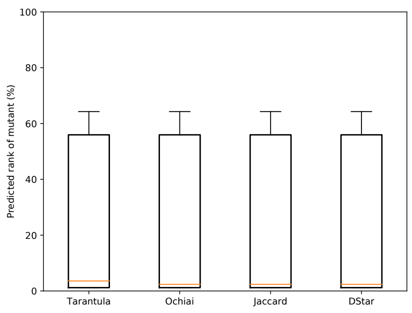


(d) Results for *instructor* test suite.

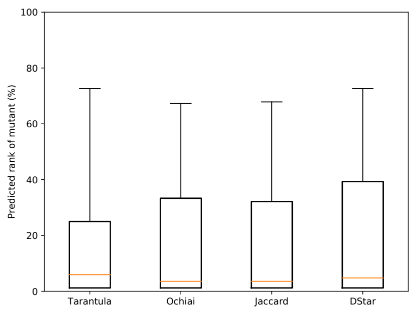
Figure 3.6: Fault seeding experiments over SIMPL grammar using JavaCC. Rows show results for different test suites, top: *rule* (43 positive tests), *cdrc* (86 positive). Bottom: *large* (2964 positive, 32157 negative), *instructor* (20 positive, 41 negative).



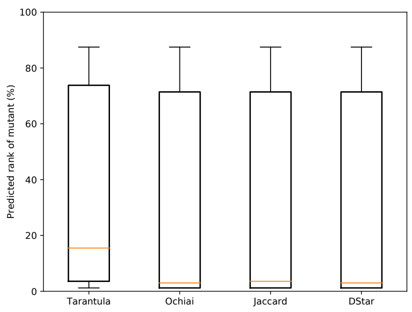
(a) Results for *rule* test suite.



(b) Results for *cdrc* test suite.

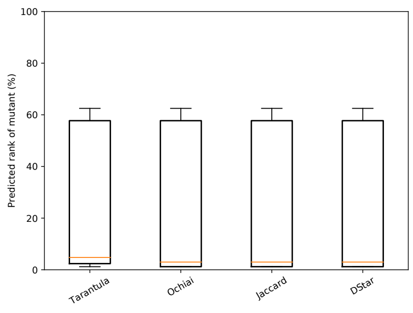


(c) Results for *large* test suite.

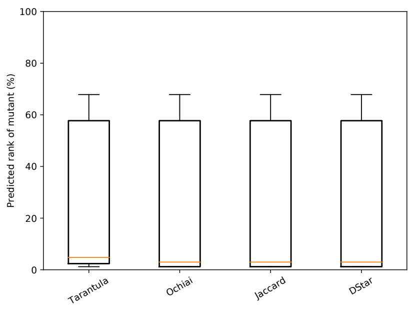


(a)Results for *instructor* test suite.

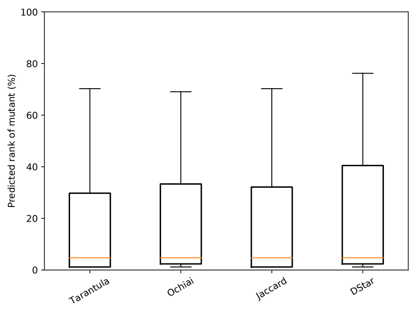
Figure 3.7: Results of fault seeding experiments over SIMPL grammar using ANTLR (without error correction). Boxplot layout as in Fig. 3.6.



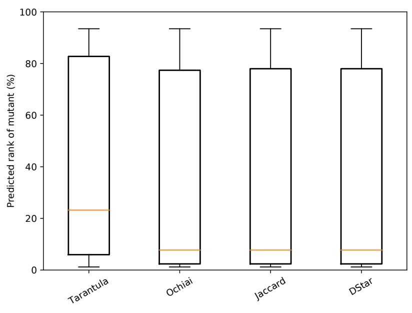
(a) Results for *rule* test suite.



(b) Results for *cdrc* test suite.

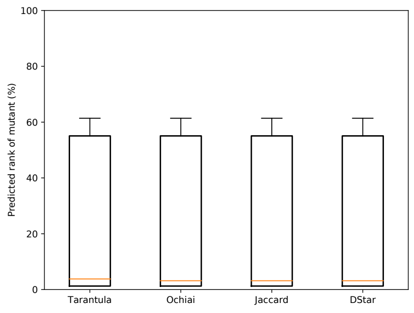


(c) Results for *large* test suite.

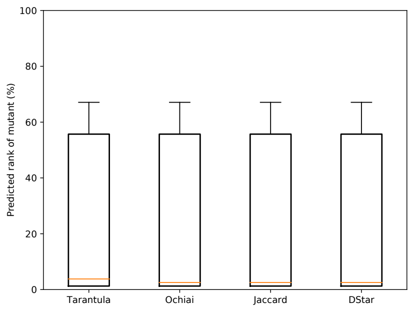


(d) Results for *instructor* test suite.

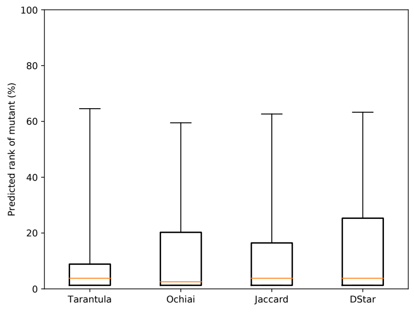
Figure 3.8: Results of fault seeding experiments over SIMPL grammar using ANTLR (with default error correction). Boxplot layout as in Fig. 3.6.



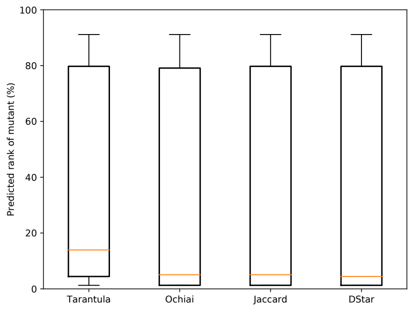
(a) Results for *rule* test suite.



(b) Results for *cdrc* test suite.



(c) Results for *large* test suite.



(d)Results for *instructor* test suite.

Figure 3.9: Results of fault seeding experiments over SIMPL grammar using CUP. Boxplot layout as in Fig. 3.6.

Table 3.3: Detailed rule-level results of fault seeding experiments over SIMPL grammars. $\tilde{x}$ and $\bar{x}$ denote the median and mean rank, respectively, of the seeded fault. #1 denotes the number of cases where the metric ranked the seeded fault as most suspicious, #3 and #5 denote the number of cases where the seeded fault was ranked in the Top 3 and Top 5, respectively. The first block contains the main results to answer RQ1a, the second block demonstrates the results for position-one mutants. The third block contains the results for synthetic spectra.

			Tarantula					Ochiai					Jaccard					DStar				
		killed	$\tilde{x}$	$\bar{x}$	#1	#3	#5	$\tilde{x}$	$\bar{x}$	#1	#3	#5	$\tilde{x}$	$\bar{x}$	#1	#3	#5	$\tilde{x}$	$\bar{x}$	#1	#3	#5
ANTLR (30821)	*rule*	24326	3.6	22.4	6235	12638	13633	1.8	21.5	7166	14874	15063	1.8	21.5	7169	14875	15064	1.8	21.5	7160	14864	15059
	cdrc	24485	3.0	22.5	6559	12714	13804	1.8	21.8	7273	15023	15262	1.8	21.9	7274	15023	15260	1.8	21.9	7267	15010	15155
	large	30539	5.4	18.9	8327	13152	16283	3.6	18.9	8749	15323	16988	3.6	18.7	8642	15717	17656	4.8	21.3	8505	14812	16425
	instr	26162	14.9	32.6	4063	7560	10342	2.4	28.3	7032	14101	14885	3.0	28.5	6849	13895	14681	2.4	28.3	7029	14434	14889
ANTLR*	*rule*	24290	4.2	22.4	5709	11258	13588	2.4	21.1	6770	15171	15413	2.4	21.2	6771	15172	15365	2.4	21.2	6750	15120	15349
	cdrc	24449	4.2	22.5	5891	11502	13899	2.4	21.4	6896	15304	15648	2.4	21.4	6896	15291	15595	2.4	21.5	6887	15286	15519
	large	30501	4.8	19.0	8554	13803	17937	4.2	19.5	7301	14331	17605	4.2	19.3	7815	14063	17946	4.8	21.8	7086	13791	16861
	instr	26126	22.6	39.2	2142	4800	6853	7.1	30.7	4594	10315	11922	7.1	31.0	4507	10003	11524	7.1	30.6	4919	10851	11930
JavaCC (36443)	*rule*	28445	3.8	17.9	6979	14001	17749	1.6	16.5	8650	19338	19762	1.6	16.5	8648	19336	19758	1.6	16.6	8637	19205	19683
	cdrc	29119	2.7	16.5	8664	15602	19008	1.6	15.2	10391	20122	20742	1.6	15.2	10391	20121	20741	1.6	15.5	9807	19481	20187
	large	32348	3.2	12.5	10505	16829	19443	2.2	9.0	12105	20811	23306	2.2	8.6	12319	20967	23798	2.2	9.7	12025	21010	23315
	instr	30837	17.7	32.4	2441	7026	9696	4.3	27.8	6366	14224	16146	4.8	26.9	6262	13999	16067	4.3	27.7	6338	14182	16100
CUP (26490)	*rule*	23526	3.2	24.3	7786	12370	13385	2.5	23.7	9905	13865	13909	2.5	23.7	9905	13866	13909	2.5	23.7	9905	13854	13897
	cdrc	25297	3.2	24.6	8372	13487	14573	1.9	24.0	10495	14903	14955	1.9	24.0	10495	14905	14955	1.9	24.0	10493	14892	14937
	large	26429	3.8	11.9	8625	15467	18666	2.5	14.1	10097	14990	16575	3.8	14.2	9326	14905	17624	3.8	15.9	10245	14564	15938
	instr	25404	13.9	35.6	2679	7205	9792	4.4	32.9	6859	11460	13179	5.1	33.3	6475	10610	12993	3.8	32.8	7399	12965	13821
ANTLR	*rule*	-	55.4	48.8	145	1095	1297	55.4	49.3	170	875	1042	55.4	49.3	173	876	1043	55.4	49.3	164	868	1038
	cdrc	-	56.5	49.9	206	1028	1241	56.5	50.3	207	906	1088	56.5	50.4	208	906	1086	56.5	50.5	201	893	980
	large	-	8.3	23.5	3003	4529	5679	31.0	32.7	1547	2460	2907	22.0	31.4	1691	2946	3656	38.1	38.7	1095	1894	2327
	instr	-	79.8	62.0	307	1042	1414	79.8	63.3	387	1176	1236	79.8	63.3	380	1175	1240	79.8	63.8	369	1139	1182
JAVACC	*rule*	-	53.8	37.8	940	2511	3143	53.8	37.2	1706	3041	3361	53.8	37.2	1704	3039	3357	53.8	37.4	1693	2908	3282
	cdrc	-	54.8	34.5	2231	3437	4037	54.8	34.1	2793	3677	3987	54.8	34.1	2793	3677	3987	54.8	34.7	2209	3036	3432
	large	-	4.3	15.5	4172	6163	7393	6.5	17.5	4321	5879	6492	3.2	14.7	4900	6921	7693	8.1	19.5	4122	5661	6141
	instr	-	59.1	53.6	179	1235	1794	61.8	52.4	947	2311	2737	61.8	52.2	920	2290	2595	62.4	52.5	914	2244	2651
CUP	*rule*	-	54.4	53.6	305	394	435	54.4	53.7	296	387	420	54.4	53.7	296	388	420	54.4	53.7	296	379	418
	cdrc	-	55.1	55.2	342	429	458	55.1	55.3	327	404	445	55.1	55.3	327	406	445	55.1	55.3	327	401	433
	large	-	6.3	19.9	2210	3885	4656	27.2	29.8	1067	1587	1955	19.0	28.0	1242	2567	3248	31.6	34.3	813	1022	1251
	instr	-	81.0	72.9	263	436	501	81.0	73.5	324	435	461	81.0	73.6	327	436	460	81.0	73.7	322	431	456
synthetic	*rule*	23563	1.8	5.1	9894	14805	20197	1.2	2.0	13012	22772	23085	1.2	2.0	13012	22772	23085	1.2	1.9	13014	22780	23105
	cdrc	25756	1.8	4.7	10761	17284	22141	1.2	2.1	13582	24189	24978	1.2	2.1	13567	24170	24964	1.2	2.1	13583	24212	24995
	large	27538	2.4	9.0	12866	19083	22058	1.2	2.8	16482	24187	25296	1.2	3.3	15766	22972	24727	1.2	2.7	16625	24223	25320

RQ1a: Our fault localization based on rule spectra is indeed effective in identifying faults in fault-seeded grammars. In more than half of the cases, the fault is localized within the top three rules. In about 10–40% of the cases, the fault is uniquely identified as the most suspicious rule. We observe also that Ochiai, Jaccard and DStar, by and large, produce identical rankings. They also outperform Tarantula.

Results for Position-one Mutants. The application of a rule, no matter the parsing technology used, heavily relies on lookahead symbols. These lookahead tokens are derived from the first and follow sets, at least in the case of our target LL(1) and LALR(1) parsers. Therefore, it becomes a challenge to correctly localize position-one mutants as the mutated rules may never be executed. The results from the second block of Table 3.3 demonstrate how hard it indeed is to correctly localize these mutants. Specifically, in subject grammars in which the mutation operators have been applied to the first position on the right hand-side of a rule, the median rank of the faults range between 50–82%, i.e., the localization performance can differ by 20% points between mutants at the first position and other symbols. Moreover, the average ranks are lower than median ranks, which means there are only a few cases where the fault is localized in fewer than half of the rules. This is also confirmed by the #1/#3/#5 values; where in less than 20% of the cases, the fault is localized within the top five rules across the board. The *large* test suite is an exception, with lower median values ($\leq$ 27%) because generation of negative test cases from *large* in part follow similar mutation strategies used to seed faults in these subject grammars.

Threats to validity. In addition to the usual concerns about construct (i.e., implementation and data collection errors) and statistical conclusion validity, we see several threats to the validity of generalizing our observations beyond the experimental setup, e.g., to other ranking metrics, parsing methods, grammars, or test suites. Our fault seeding experiments are based on a single grammar; since test suite construction, mutant construction, and spectrum collection all depend on the structure of the grammar, different grammars may yield different results. We encountered this when we used an ANTLR version that was not left-factorized, which triggered the rule tracking issues described in Section 3.3 and led to incomplete spectra that distorted the results. Our fault seeding also includes mutations at the first symbol of a rule, which may produce non-LL(1) mutants that also trigger the rule tracking issues and so distort results; preliminary analysis has shown that the localization performance can differ by 15 rules (i.e., close to 20 percentage points) between mutations at the first and at other symbols. However, we used a variety of other grammars on other (non-seeded faults) experiments, without any substantially different results, which partially mitigates this threat.

Gopinath et al. [53] have shown that mutants are not syntactically close to real faults, but there is evidence that they are nevertheless a valid substitute in many software engineering applications, including fault localization [71]. However, grammar mutations as we have used here have not been investigated, and other mutation operations (e.g., adding epsilon-productions or deleting entire rules) may yield different results. Hence, even though our localization experiments with student grammars (see Section 3.6.3) show similar results, care should be taken in generalizing the results above.

The experiments have shown that the localization performance depends on the test suites and may thus not generalize, despite the differences in the test suites we have used. The *large* test suite contains tests that are constructed based on the same principle as the mutants (i.e., rule mutation) and may thus overestimate performance.

3.6.2 Effectiveness on Synthetic Spectra (RQ1b)

The third block in Table 3.3 shows the results of our approach when using synthetic spectra constructed directly from test cases derived from a grammar under test. We see that our approach remains effective, and in fact, produces better results. First, on average, the faulty rule is ranked in ∼5% of the rules, with a slightly worse figure (9.0%) for Tarantula under *large* test suite. The median range of 1.2–2.8% means that, in half of the cases, we only need to look within the top three rules to find the faulty rule. Second, interestingly, the fault is uniquely localized in at least 40% of the cases and in ∼85% of the cases the prediction is within the top five most suspicious rules. Third, as before, Tarantula performs worse than the three other metrics, with higher median and mean ranks and lower #1/#3/#5 values.

RQ1b: Our rule-level localization based on synthetic spectra is more effective in identifying single faults in mutants than using grammar spectra extracted from parsers. The fault is found within the top five rules in almost all the cases.

3.6.3 Localization of Real Faults (RQ1c)

In the next set of experiments, we used student submissions (which unsurprisingly contain many errors) to compiler engineering courses to see how well our method performs over grammars with multiple real faults.

Experimental Setup. We used two languages, SIMPL (which we also used for the fault seeding experiments in Section 3.6.1), and Blaise, another artificial teaching language of similar syntactic complexity: the instructor's version of the grammar has 38 non-terminals, 40 terminals and 75 rules.

For SIMPL, we used the same positive test cases as in the *large* test suite in Section 3.6.1. For Blaise, we generated tests using the same mechanism; this comprises 7280 positive and 9119 negative test cases.

Both languages were used in compiler engineering courses. In one course, the students were given the same EBNF as in the computer architecture course (in fact, most students were from a cohort that already used SIMPL in that course), and were asked in two different assignments to use ANTLR and CUP (or a similar LALR(1) parser generator of their choice) to develop parsers for SIMPL. We randomly picked ten ANTLR submissions, from which we discarded two that pass all tests and one that did not produce a compilable parser. We picked all ten CUP submissions, from which we discarded three that passed all tests and one that passed none. For Blaise, the students were given a textual language description and a small set of short example programs. We randomly picked nine Blaise grammars from 110 submissions, from which we also discarded two that pass all tests. This leaves us with 20 subject grammars.

We then followed an iterative one-bug-at-a-time (OBA) debugging technique [165] where we focus our attention on a single first discovered fault, fix the fault and then re-localize. In each step, we used the Ochiai metric to compute the suspiciousness scores of the rules. We manually examined the rules in rank order and used our understanding of the "true" grammars to identify and repair faulty rules. In each step, we only repaired the top-ranked faulty rule; note that we made repairs in the lexer as well. After each repair, we continued with the next iteration, until the grammar under test passed all test cases.

Experimental Results. Table 3.4 summarizes the results of our evaluation over student grammars. For each iteration, we show the number of test cases failed by that grammar version, and the rank of the rule that we identified as faulty and repaired for the next iteration. Empty cells indicate that a previous repair allowed the parser to pass all tests.

While we have no guarantee that we always pick the "right" rule for repair, we can observe for all but one of the grammars the number of failed test cases decreases with each repair; the exception is #8, where the repair in iteration 2 triggers more failing test cases. Here, the repair can be seen as the first step in a multi-step refactoring that temporarily increases the number of failures, which then drops significantly in the subsequent iterations. Note also that the final repair of #6 has no associated rank because the error was actually in the lexer which returned an identifier token instead of the AND-operator. In other cases, we could identify similar lexical errors via the rules.

The most common issues are around the formal and actual parameter lists of functions. These cannot be empty, but the textual language specification was vague about this, and many students interpreted this differently. The other fault classes include:

Table 3.4: Results of iterative fault localization in student grammars and manual repair. #fail shows the number of failing test cases in an iteration and rank shows the rank of the manually repaired rule.

			iteration 1		iteration 2		iteration 3		iteration 4		iteration 5		iteration 6		iteration 7	
#	language	type	#fail	rank	#fail	rank	#fail	rank	#fail	rank	#fail	rank	#fail	rank	#fail	rank
1	SIMPL	CUP	557	1.5	254	1	131	1	98	1						
2	SIMPL	CUP	206	2	95	2										
3	SIMPL	CUP	498	1	40	1										
4	SIMPL	CUP	305	5	48	1										
5	SIMPL	CUP	854	3	854	1	295	1	139	1	48	2	19	1	5	1
6	SIMPL	CUP	48	1												
7	Blaise	ANTLR	567	2	4	1	2	1								
8	Blaise	ANTLR	1082	1	535	3	7213	1	358	1	43	1	2	1		
9	Blaise	ANTLR	4	3	2	2										
10	Blaise	ANTLR	1068	1	4	2	2	1								
11	Blaise	ANTLR	38	4	3	1										
12	Blaise	ANTLR	654	1	1	1										
13	Blaise	ANTLR	4	2	2	1										
14	SIMPL	ANTLR	555	1	170	1	47	2	1	1						
15	SIMPL	ANTLR	37	4.5	1	1.5										
16	SIMPL	ANTLR	361	3	46	1										
17	SIMPL	ANTLR	396	1.5	117	2	81	2	47	1	1	1.5				
18	SIMPL	ANTLR	46	2												
19	SIMPL	ANTLR	356	1	233	2	1	1								
20	SIMPL	ANTLR	1	1												

(i) the interaction between the parser and lexer, which some students did not handle well especially in cases involving unary and binary MINUS operators with the latter subsumed in ADDOP operators;

(ii) token issues such as typographical errors and wrong regular expressions (strings in most cases); and

(iii) a few cases of tool specific issues, e.g., wrongful use of EBNF operators in ANTLR.

RQ1c: We can conclude that, with the help of OBA, our fault localization approach remains effective under multiple faults: in all cases the repaired rule is within the top five rules, and even was the top ranked rule in more than half of the cases. This translates to a lower average wasted effort (AWE), the number of rules one has to look before the fault is found, from across the board.

Threats to validity. This experiment is subject to similar threats to validity as the one described in the previous section; in particular, the results may not generalize to other grammars or to other ranking metrics. However, as mitigation we used a broadly similar setup in the experiments described in the following section, where we achieved similar results.

Since the setup involves human judgements by the authors, the results are also subject to possible experimenter bias, human error, and human performance variation. We tried to mitigate against this threat by following an experimental protocol over unseen grammars, but this was not fully defined (e.g., rule selection and choice of the cut-off points).

3.6.4 Localization for Large Black-box Systems (RQ1d)

To address questions related to scalability of our approach, we design a case study where we try to identify parts of a public SQLite grammar that is known to deviate from the language accepted by the actual SQLite system. We have retrieved the ANTLR4 SQLite grammar from `https://github.com/antlr/grammars-v4/blob/master/sql/sqlite/` (commit 37a9df3). The BNF version of the grammar that has been used to generate test queries has 440 rules, 181 non-terminals and 170 terminals. This shows that it is a fairly large grammar, at almost 5× the size of the SIMPL grammars we used in Sections 3.6 and 4.4. The black-box system that we used is the SQLITE3 Python module (v2.6.0) that is essentially an API wrapper for a runtime SQLite library (v3.22.0) written in the C programming language. We then wrote a simple adaptor that creates a database connection, executes generated queries and logs each execution outcome.

We assume that the adaptor provides, on executing each query, the syntactic pass/fail information. Here, we consider a test case to fail if the system detects any syntax errors in the input and to pass if the query executes successfully or if it throws exceptions that lie deeper in the system, beyond the syntax analysis stage. With the above framework established, it appears straight-forward to directly invoke our "flipped" version of our method that uses synthetic grammar spectra derived directly from test cases to identify deviations. However, it proved impractical to blindly generate and run queries on the system, despite our sole interest in exercising the parser. In particular, the system complains of early stage errors such as `incomplete input`; perhaps more importantly, since the runtime system is basically a single-pass compiler, execution stops due to a semantically ill-formed query before it could complete parsing. Lack of a standalone parser also means that we cannot directly exercise these generated queries.

To tackle the aforementioned limitations and handle some of the preconditions, the idea is to provide our test generator with predefined and fixed table- and column-names during query generation. Fortunately, the language accepted by the SQLite system is composed of different types of statements, which can be seen as sub-languages, that define, query and manipulate tables and data in different ways. For example, the symbol *sql_stmt* (which is reached directly from the start symbol *program*) below is a union of entry points to these sub-languages.

$$\begin{aligned} &\mathit{program} \rightarrow \mathit{sql\_stmt}(\ \textbf{;}\ \mathit{sql\_stmt})* \\ &\mathit{sql\_stmt} \rightarrow \ldots \mid \mathit{create\_table\_stmt} \mid \ldots \end{aligned}$$

Our generator grammar defines over 20 alternatives for the *sql_stmt*-rule. This allows us to model the system effectively by overriding the start rule and start derivations from each sub-language entry-point. We also fix values for rules *table_name*, *column_name* and *database_name* by setting allowed names. For example, Figure 3.10 shows a simplistic code snippet written in the Prolog programming language which our test suite generation tool uses as input that sets the start rule to **`CREATE TABLE`** related statements (line #1). The values to set table, column, and database names are given in lines #3, #5, and #7 respectively. This separation also allows us to handle some dependencies between statement types, e.g., the **`DROP TRIGGER`** statement requires a successful execution of the **`CREATE TRIGGER`** statement.

We therefore wrote a series of models (13 in total) like the one shown in Figure 3.10. The one in the figure targets the **`CREATE TABLE`** related statements and uses the *create_table_stmt* and *create_virtual_stmt* rules exclusively to generate tests. The other twelve models generate tests from the remaining statements. Unlike the model in Figure 3.10, they assume the prior successful creation of tables, and like the first model, they are equipped with valid table names and column names. Dependent statement types are handled by chaining up their corresponding rules with a semicolon

```
model :- create_table_stmt | create_virtual_table_stmt .
%%% set the allowed table names
table_name :- 'STAFF' | 'DEPARTMENTS' | 'ORDERS'.
%%% set the allowed column names
column_name :- 'Name' | 'Department' | 'Number'.
%% set database name
databasename :- 'databasename' .
```

Figure 3.10: Example model that generates tests from **`CREATE TABLE`** statements, with hard-coded allowed values for table, column and database names.

separator. For example, *create_trigger_stmt* is always followed by optional (**;** *drop_trigger_stmt*), i.e., *model* $\rightarrow$ *create_trigger_stmt*(**;** *drop_trigger_stmt*)?. Note that *drop_trigger_stmt* can also be used independently with an optional **`IF EXISTS`** clause to avoid an error. However, this was less intuitive as it required meddling with the grammar in order to enforce the clause to always be present in the **`DROP TRGGER`** statements and other highly dependent statements.

We are aware that this exploitation of the structure of the SQLite grammar targets certain parts of the system and does not exercise all grammar rules. However, we still managed to cover a large portion of the grammar. For example, spectra from the model that tests create table related statements are composed of 274 rules that are applied in the generation of the *deriv* and bfs_2 test suites. The highest number of applied rules in generation of the test suite by any model is 371.

We then followed a multi-stage fault localization approach where we, in each stage, use each model to orchestrate the generation of the test suite TS^+, which we then use to localize deviations for the language accepted by the SQLite system. In each stage, we then used the same OBA technique [165] in Section 3.6.3, where we again focus our attention on a single first discovered deviation, manually *fix* this deviation, and then re-generate the tests to re-test the SQLite system. In each iteration, we used the Tarantula metric to calculate suspiciousness scores for all grammar rules. This choice of ranking metric is based on the observation that Tarantula seemed to produce more stable rankings under a high number of test failures. We examined these rules in their descending order of suspiciousness, starting with the most suspicious rule, and identified their corresponding syntactic description as per the SQLite official specification available at `https://sqlite.org/syntaxdiagrams.html`. We then manually inspected the grammar rule and its corresponding description to identify the cause of deviation. We finally repaired the deviation in the grammar rule and repeated this process until no further tests failed.

Results

Deviation #1. The first model (see Figure 3.10) gave us an initial set of 462 failing tests out of a total of 57656 generated tests. In the first iteration, we made the following observations. First, Tarantula ranked the rule

$$expr \rightarrow expr\ (\texttt{=} \mid \texttt{==} \mid \ldots \mid \texttt{IS} \mid \texttt{IN} \mid \ldots)\ expr$$

as the most suspicious rule. The rule defines the structure of binary operators in SQLite. We consulted the official documentation and the corresponding description for expressions, which revealed that the deviation is in the use of the **`IN`** operator. This operator has to be followed by parenthesized expressions and not by an arbitrary expression, as allowed by the grammar rule. However, the grammar contains multiple faults (or more precisely, "deviations"). This is made evident by the fact that the above faulty *expr*-rule has not been executed in all failing tests, but in only 456 of those failing tests.

Since we follow the OBA principle, we first fixed the deviation in the **`IN`** operator. We then found another occurrence of the **`IN`** that was correctly implemented as:

$$expr \rightarrow expr\ \ \texttt{NOT}\,?\,\texttt{IN}\ \texttt{(}\ select\_stmt \mid expr\ \ (\ \texttt{,}\ expr)?\ \texttt{)}$$

This means that the top-ranked rule $expr \rightarrow expr\ (\ldots \mid \texttt{IN} \mid \ldots)\ expr$ is an over-approximation fault on the correct use of the **`IN`** operator which we fixed by simply deleting the symbol **`IN`** from the rule.

Deviation #2. After the modification, we regenerated the test suites using the same (first) model and repeated the process. In this iteration, the following six tests failed:

```
create table STAFF as values(0) limit 0
create table STAFF as with STAFF as(select *) values(0) order by ?
create table STAFF as with STAFF as(select *) values(0) limit 0
create table STAFF as values(0) order by ?  limit 0
create table STAFF as values(0) order by ?
create table STAFF as with STAFF as(select *) values(0) order by ?limit 0
```

The SQLite system reports the following syntax error messages each for each of the test cases above.

```
near "limit":  syntax error
near "order":  syntax error
near "limit":  syntax error
near "order":  syntax error
near "order":  syntax error
near "order":  syntax error
```

From the error messages, it is not straightforward where the cause of the deviation might be; the token **`)`** occurs on the correctly consumed prefix before the offending tokens **`limit`** and **`order`**. The *select_stmt*-rule[3] that causes the deviation is ranked sixth (out of a total 440 rules).

$$
\begin{aligned}
\mathit{select\_stmt} &\rightarrow \ldots \mathit{select\_or\_vals} \ldots \\
&\quad (\texttt{\textbf{ORDER BY}}\, \mathit{ordering\_term}(\texttt{\textbf{,}}\, \mathit{ordering\_term})*)? \\
&\quad (\texttt{\textbf{LIMIT}}\, \mathit{expr}\, ((\texttt{\textbf{OFFSET}} \mid \texttt{\textbf{,}})\, \mathit{expr})?)? \\
\mathit{select\_or\_vals} &\rightarrow \texttt{\textbf{SELECT}} \ldots \mathit{result\_col} \ldots \texttt{\textbf{FROM}}\, \mathit{table\_or\_subquery} \ldots \\
&\mid \texttt{\textbf{VALUES (}}\, \mathit{expr}\, (\texttt{\textbf{,}}\mathit{expr}) * \texttt{\textbf{)}}
\end{aligned}
$$

This deviation is confirmed by the official documentation for the VALUES clause in a select statement. "*There are some restrictions on the use of a VALUES clause that are not shown on the syntax diagrams*:"

- A VALUES clause cannot be followed by ORDER BY.
- A VALUES clause cannot be followed by LIMIT.

We fixed this deviation by transforming the rules to the following,

$$
\begin{aligned}
\mathit{select\_stmt} &\rightarrow \ldots \mathit{select\_or\_vals} \ldots \\
\mathit{select\_or\_vals} &\rightarrow \texttt{\textbf{SELECT}} \ldots \mathit{result\_col} \ldots \texttt{\textbf{FROM}}\, \mathit{table\_or\_subquery} \ldots \\
&\quad (\texttt{\textbf{ORDER BY}}\, \mathit{ordering\_term}(\texttt{\textbf{,}}\, \mathit{ordering\_term})*)? \\
&\quad (\texttt{\textbf{LIMIT}}\, \mathit{expr}\, ((\texttt{\textbf{OFFSET}} \mid \texttt{\textbf{,}})\, \mathit{expr})?)? \\
&\mid \texttt{\textbf{VALUES (}}\, \mathit{expr}\, (\texttt{\textbf{,}}\, \mathit{expr}) * \texttt{\textbf{)}}
\end{aligned}
$$

The transformation pushes down the sequences that capture the **`ORDER BY`** and **`LIMIT`** clauses to the end of the first alternative of the *select_or_vals*-rule.

***Deviation* #3**. In this iteration, we used a model that start derivations from rules that define the structure of triggers (in particular, their creation and removal), more specifically we have the following as the start production,

$$\mathit{model} \rightarrow \mathit{create\_trigger\_stmt}(\, \texttt{\textbf{;}}\, \mathit{drop\_trigger\_stmt})?$$

We execute a generated test suite with 60781 test cases from which 1960 fail. Our fault localization results are not particularly discriminatory in this case, but a tie between two top ranked rules below already give a good idea of the location of the deviation.

$$
\begin{aligned}
\mathit{with\_clause} &\rightarrow \texttt{\textbf{WITH RECURSIVE}}\,?\ \mathit{cte\_table\_name}\ \texttt{\textbf{AS}}\ \texttt{\textbf{(}}\, \mathit{select\_stmt}\, \texttt{\textbf{)}} \ldots \\
\mathit{cte\_table\_name} &\rightarrow \mathit{table\_name}(\, \texttt{\textbf{(}}\, \mathit{column\_name}\, (\, \texttt{\textbf{,}}\, \mathit{column\_name}) * \texttt{\textbf{)}}\,)?
\end{aligned}
$$

[3]The full *select_stmt*-rule is as follows

$$
\begin{aligned}
\mathit{select\_stmt} \rightarrow\ & (\texttt{\textbf{WITH RECURSIVE}}\, \mathit{common\_table\_expression}\, (\texttt{\textbf{,}}\, \mathit{common\_table\_expression})*)? \\
& \mathit{select\_or\_values}\, (\mathit{compound\_operator}\, \mathit{select\_values})* \\
& (\texttt{\textbf{ORDER BY}}\, \mathit{ordering\_term}\, (\texttt{\textbf{,}}\, \mathit{ordering\_term})*)? \\
& (\texttt{\textbf{LIMIT}}\, \mathit{expr}\, ((\texttt{\textbf{OFFSET}} \mid \texttt{\textbf{,}})\, \mathit{expr})?)?
\end{aligned}
$$

The official documentation outlines syntax restrictions on **INSERT**, **UPDATE**, and **DELETE** statements within triggers: "*Common table expression are not supported for statements inside of triggers.*" The trigger-related rules in the grammar completely ignore this restriction and allow generic **INSERT**, **UPDATE**, and **DELETE** statements inside triggers. The faulty rules *with_clause* and *cte_table_name* are directly derivable from these statements; the latter only ever occurs in the former (i.e., *with_clause*-rule).

In the fix for this deviation, we simply duplicate rules from the three statements and remove the call to *with_clause*.

Deviation #4. The above fix did not cater for all failures as we are left with another 1152 failing tests after the modification. Here, all the test failures have the same structure and the system throws similar syntax error messages. Below, we show one of the failing tests

```
create trigger tr1 delete on STAFF begin select 0 between 0 or 0 and 0;end
```

and its corresponding error message:

```
near ";":  syntax error
```

From these failures we can, to some extent, conclude that the interaction among operators **BETWEEN**, **OR** and **AND** (in that order) is problematic. Fault localization also confirms this with the two *expr*-rules (shown below) flagged as the most suspicious and both rules are applied in the derivation of all failing tests (i.e., both have *ef* and *nf* counts of 1152 and 0 respectively).

$$\begin{aligned} expr \to\; & \ldots \\ & \mid expr\ \mathbf{OR}\ expr \\ & \mid expr\ \mathbf{NOT}\ ?\ \mathbf{BETWEEN}\ expr\ \mathbf{AND}\ expr \\ & \mid \ldots \end{aligned}$$

Taking a closer look, it seems the parser detects precedence issues between the operators **OR** and **AND** which has a higher precedence than **OR**, due to a parsing conflict between

$$expr \to expr\ \mathbf{AND}\ expr$$

and

$$expr \to expr\ \mathbf{NOT}\ ?\ \mathbf{BETWEEN}\ expr\ \mathbf{AND}\ expr$$

To circumvent this behaviour, wrapping parentheses around the *expr*- symbol before **AND** in the second *expr* rule seemed to be the most plausible fix. The modified rule is as follows:

$$expr \to expr\ \mathbf{NOT}\ ?\ \mathbf{BETWEEN}\ \mathbf{(}\ expr\ \mathbf{)}\ \mathbf{AND}\ expr$$

Note that this is not a "proper" fix, but rather a "grammar hack" that does not modify the language, but it resolves all remaining test failures and so demonstrates that there are no further deviations

In this experiment, we see that our approach enables us to identify four deviations in larger SQLite grammar. The OBA principle also allows us to find the first deviation in each stage with low average wasted effort, we only needed to inspect one rule to find the first deviation, five rules before the second deviation was found, two rules for the third deviation and finally the cause of the fourth deviation was the top-ranked rule. The other models did not result in any test failures.

RQ1d: The rule-level localization remains effective and scales to large production quality grammars.

3.7 Conclusion

In this chapter, we have introduced our baseline fault localization method that produces results at the level of individual grammar rules. We described how popular parser generator tools such as JavaCC, ANTLR and CUP can be extended to extract the grammar spectra that are necessary for fault localization. We have also described how we take advantage of test cases generated from a faulty input grammar to extract synthetic spectra in cases where the system under test cannot be extended or instrumented.

We have demonstrated the efficacy of our approach using a series of experiments. In a larger experiment using fault seeded grammars, we showed that our approach finds the faults with high precision. We then applied our approach to a much more difficult task, that of finding real faults in student grammars. We showed by using an iterative one-bug-at-time debugging approach that our approach remains effective in this kind of setting. We also used the same technique to identify where a larger, production quality SQLite grammar deviates from the black-box parser implemented by an actual SQLite system.

Chapter 4

Item-Level Fault Localization

Localization with rule spectra allows us to identify faulty rules in a grammar; however, we still have to inspect the individual symbols on the right-hand side of the rules to identify the actual fault location. This can involve substantial effort since the rules can be long; for example, the BNF version of the SQLite grammar [74] has more than twenty rules that each contain six or more symbols, and the size of the longest rule is sixteen.

We therefore refine our method to localize errors more precisely, at the level of the individual symbols in a rule. Our basic idea here is to use spectra over *items* rather than over rules for the localization. This exploits the fact that the designated position marks the boundary between the part of a rule that has already been processed successfully and the part that still needs to be processed; hence, we assume that the error is at the symbol following the designated position.

We build on the extensions for rule spectra collection (see Sections 3.3 and 3.4) for parser generator tools JavaCC and CUP (which we call SymJavaCC and SymCUP, respectively, to distinguish them from their rule-level counterparts).

Furthermore, the development of item-level localization is a necessary step for automatic repair (see Chapter 5). Item-level localization naturally makes the search space easier to navigate than rule-level localization. This follows a similar trend to automated program repair algorithms [48, 49, 108], which use fine-grained statement-level fault localization results, despite several spectrum-based fault localization [32, 158] studies demonstrating that SFL works better at method-level spectra than statement-level spectra.

Outline. We first illustrate the item-level localization method with a worked example based on the same grammar $G_{\mathcal{Toy}}$ used in Chapter 3. We give formal definitions of *plain* item spectra in Section 4.2.1. Section 4.2.2 defines *shift* item spectra, the second approach to item spectra extraction for LR parsers. We also describe the domain-specific tie breaking strategy that prefers the right-most item over other items from the same rule in Section 4.2.3. We then describe how item spectra can be extracted from JavaCC and CUP parser generators in Section 4.3. We demonstrate the efficacy of the item-level localization and compare it to rule-level localization in Section 4.4.

4.1 Worked Example

We illustrate the item-level localization with the same example as in Chapter 3; specifically, we assume the same faulty grammar under test G'_{Toy}, with the faults in the **while**- and **if**-rules, but since we are now trying to locate the position of one or more offending symbols on the right-hand side of a rule, the two items

$$\begin{array}{l} stmt \to \textbf{if}\ expr\ \textbf{then}\ stmt \bullet \textbf{else}\ stmt \\ stmt \to \textbf{while}\ expr\ \textbf{do} \bullet block \end{array}$$

now represent the faults.

Table 4.1 shows the detailed results for the item spectra collected according to Definitions 4.2.1 and 4.2.2. Items are denoted by the non-terminal name, the index of the corresponding alternative, as shown in Figure 3.1, and the index of the designated position. Note that, due to the large spectrum size (81 elements, compared to 15 elements for rule spectra), entries are shown only for the 25 items which have been executed in at least one of the two failing test cases 6 and 11, and thus have non-zero suspiciousness scores. For each metric, Table 4.1 shows two different rankings, the standard ranking and the resolved ranking, where ties are resolved using the *k-max* tie breaking mechanism described in Section 4.2.3. In both cases, ties are indicated by a preceding "=". Items corresponding to the fault locations are shown in bold.[1]

Results. In Table 4.1, we see Tarantula assigns the highest suspiciousness score to three items from the **while**-rule (including the item *stmt*:3:3 corresponding to the fault position), while the other three metrics rank these behind items from the *block*- and *expr*-rules. As in the case of the rule-level localization, the fault in the **if**-rule is harder to localize, and all four metrics rank it behind several (further) items from the *block*- and *expr*-rules, at tied ranks 7 (Tarantula), 9 (Jaccard), and 11 (Ochiai and DStar), respectively.

If we inspect the items in rank order and resolve ties by picking rules arbitrarily, we have on average to look at 8 positions (i.e., 9.8% of all positions) in 6 rules before we find both faults using Tarantula, 10.5 positions (13.0%) in 7 rules using Jaccard, and 12.5 positions (15.4%) in 7 rules using Ochiai

[1]Note also that we cannot directly compare fault localization results in this section with those in Section 3.1 because the worked example in the previous chapter was manually constructed, while in this section, we use the two slightly different LR item spectra extraction approaches introduced in Section 2.4 and implemented in SymCUP.

Table 4.1: Item spectra, suspiciousness scores, ranks, and resolved ranks for the faulty grammar version G'_{Toy} and *rule* test suite. Items are denoted by the non-terminal name, the index of the corresponding alternative, as shown in Figure 3.1, and the index of the designated position. Entries are only shown for items with non-zero scores. The standard ranking is shown on the left side of the rank column, the resolved ranking using the strategy in Section 4.2.3 on the right side; ties are indicated by a preceding "=". Items corresponding to the fault locations are shown in bold.

item	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	*ep*	*np*	*ef*	*nf*	$\mathcal{T}$			$\mathcal{O}$			$\mathcal{J}$			$\mathcal{D}$		
prog:1:0	✓	✓	✓	✓	✓	✗	✓		✓	✓	✓	✓	✓	✗	✓	✓	14	0	2	0	0.5	=21		0.35	=18		0.13	=21		0.29	=17	
prog:1:1	✓	✓	✓	✓	✓	✗	✓		✓	✓	✓	✓	✓	✗	✓	✓	14	0	2	0	0.5	=21		0.35	=18		0.13	=21		0.29	=17	
prog:1:2	✓	✓	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	14	0	2	0	0.5	=21		0.35	=18		0.13	=21		0.29	=17	
prog:1:3	✓	✓	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	14	0	2	0	0.5	=21	14	0.35	=18	=12	0.13	=21	14	0.29	=17	12
block:1:0										✓				✗			1	13	1	1	0.86	12	7	0.50	=11	=8	0.33	=9	=7	0.50	=11	=8
block:1:1						✗				✓				✗			1	13	2	0	0.93	=4	=2	0.82	=1	=1	0.67	=1	=1	4	=1	=1
block:2:0											✓	✓	✓	✗			3	11	1	1	0.70	17	11	0.35	=18	=12	0.20	17	11	0.25	21	13
block:2:1						✗					✓	✓	✓	✗			3	11	2	0	0.82	13	8	0.63	8	6	0.40	8	6	1.33	5	5
block:3:0	✓	✓	✓	✓	✓		✓		✓					✗	✓	✓	9	5	1	1	0.44	25	15	0.22	25	15	0.09	25		0.10	24	15
block:3:1	✓	✓	✓	✓	✓	✗	✓		✓					✗	✓		9	5	2	0	0.61	20	13	0.43	16	10	0.18	18	12	0.44	15	10
block:4:0														✗	✓	✓	2	12	1	1	0.78	14	9	0.41	17	11	0.25	16	10	0.33	16	11
block:4:1						✗								✗	✓	✓	2	12	2	0	0.88	=7	=5	0.71	=4	=4	0.50	=4	=4	2.00	4	4
stmt:1:0						✗	✓	✓	✓	✓							4	10	1	1	0.64	=18		0.32	=23		0.17	=19		0.20	=22	
stmt:1:1						✗	✓	✓	✓	✓							4	10	1	1	0.64	=18	12	0.32	=23	14	0.17	=19	13	0.20	=22	14
stmt:2:1						✗	✓										1	13	1	1	0.88	=7		0.50	=11		0.33	=9		0.50	=11	
stmt:2:2						✗	✓										1	13	1	1	0.88	=7		0.50	=11		0.33	=9		0.50	=11	
stmt:2:3						✗	✓										1	13	1	1	0.88	=7		0.50	=11		0.33	=9		0.50	=11	
stmt:2:4						✗	✓										**1**	**13**	**1**	**1**	**0.88**	**=7**	**=5**	**0.50**	**=11**	**=8**	**0.33**	**=9**	**=7**	**0.50**	**=11**	**=8**
stmt:3:1														✗			0	14	1	1	1.00	=1		0.71	=4		0.50	=4		1.00	=6	
stmt:3:2														✗			0	14	1	1	1.00	=1		0.71	=4		0.50	=4		1.00	=6	
stmt:3:3														✗			**0**	**14**	**1**	**1**	**1.00**	**=1**	**1**	**0.71**	**=4**	**=4**	**0.50**	**=4**	**=4**	**1.00**	**=6**	**6**
expr:1:1				✓		✗								✗			1	13	2	0	0.93	=4	=2	0.82	=1	=1	0.67	=1	=1	4.00	=1	=1
expr:2:1		✓				✗								✗			1	13	2	0	0.93	=4	=2	0.82	=1	=1	0.67	=1	=1	4.00	=1	=1
expr:4:0	✓	✓	✓	✓		✗		✓						✗			5	9	2	0	0.74	=15		0.53	=9		0.29	=14		0.80	=9	
expr:4:1	✓	✓	✓	✓		✗		✓						✗			5	9	2	0	0.74	=15	10	0.53	=9	7	0.29	=14	9	0.80	=9	7

or DStar. Note that the number of rules involved does not change over the different orders in which the tied positions are inspected here, although that is not guaranteed in general.

Note that not all items from a rule are scored identically. In particular, the item *stmt*:2:5 (i.e., $stmt \rightarrow$ **`if`** *expr* **`then`** *stmt* **`else`** $\bullet$ *stmt*) is scored zero by all metrics; this is a strong indication that the fault is located to the left of its designated position. However, note also that many items from the same rule are indeed scored identically, and that the ties are therefore much longer than for the rule-level localization.

4.2 Item Spectra

In this section, we give formal definitions of *plain* item spectra (already in in Table 4.1) and *shift* item spectra (see the example in Table 4.2). The plain item spectra definitions can be seen as extensions to Definitions 3.2.1 and 3.2.2. However, plain item spectra offer no formal guarantees that they can be condensed into an equivalent rule spectra because items from the same rule can have different spectral counts. For example, items from a rule with optional elements may have different failing test executions. Our *k-max* tie resolution strategy (see Section 4.2.3) takes this into account as well and returns more than one item from the same rule if they have the same suspiciousness score but different spectral counts.

4.2.1 Plain Item Spectra

In the first approach, we still define the spectra over maximal derivations; it is based on a more or less straightforward adaptation of the corresponding definitions of rule spectra. We can therefore informally (as we did for rule spectra in Section 3.2) define an *item spectrum* for the word w in the test suite as the set of positions within rules $R^\bullet \subseteq P^\bullet$ that are processed successfully when the word w is parsed. For accepted words, the item spectrum simply includes *all items* from each applied rule.

Definition 4.2.1 (positive item spectrum). If $S \Rightarrow_{p_1} \alpha_1 \Rightarrow_{p_2} \alpha_2 \Rightarrow_{p_3} \cdots \Rightarrow_{p_n} \alpha_n = w$, then $R^\bullet = \bigcup_i \text{items}(p_i)$ is called a *positive item spectrum* for w.

For rejected words, the adaptation is slightly more complex than in the positive case. More specifically, we include in the spectrum only items from rules in derivation steps whose yield up to the designated position occurs before the syntax error. As in the case of the rule spectra (see Definition 3.2.2), we must take the frontier rules into account; however, here we only add the corresponding non-kernel items.

derivation	extracted items
$\Delta = prog$	
$\Rightarrow_{prog:1}$ **program** id = *block* .	$\{prog{:}1{:}0,\ prog{:}1{:}1,\ prog{:}1{:}2,\ prog{:}1{:}3\}$
$\Rightarrow_{block:3}$ **program** id = { *stmts* } .	$\{block{:}3{:}0,\ block{:}3{:}1\}$
$\Rightarrow_{stmts:2}$ **program** id = { *stmt* } .	$\{stmts{:}2{:}0\}$
$\Rightarrow_{stmt:3}$ **program** id = { **while** *expr* **do** *block* } .	$\{stmt{:}3{:}0,\ stmt{:}3{:}1\}$
$\Rightarrow_{expr:4}$ **program** id = { **while** id **do** *block* } .	$\{expr{:}4{:}0,\ expr{:}4{:}1,\ stmt{:}3{:}2,\ stmt{:}3{:}3,$ $block{:}1{:}0,\ block{:}2{:}0,\ block{:}3{:}0,\ block{:}4{:}0\}$

Figure 4.1: Example construction of negative item spectrum.

Definition 4.2.2 (negative item spectrum)**.** Let $w = uv \notin L(G)$ with maximal viable k-prefix u, and $S \Rightarrow_{p_1} \alpha_1 \Rightarrow_{p_2} \alpha_2 \Rightarrow_{p_3} \cdots \Rightarrow_{p_n} uX\alpha$ be a maximally viable k-prefix bounded derivation for w with frontier X. Then

$$R^\bullet = \{p^\bullet \mid \alpha A\beta \Rightarrow_p \alpha\gamma\beta \in \Delta,\ p^\bullet = A \to \mu \bullet \nu \in \text{items}(p),$$
$$\exists x \in T^* \cdot \alpha\mu x \Rightarrow^* u\} \cup \text{closure}(\{X \to \bullet\gamma \mid X \to \gamma \in P\})$$

is called a *negative item spectrum* for w.

Figure 4.1 illustrates the negative item spectrum construction for the same maximal prefix-bounded derivation Δ as in Section 3.2. It shows on the left the individual derivation steps and on the right the corresponding extracted items; the spectrum $R^\bullet$ is the union of all these sets.

4.2.2 Shift Item Spectra

The plain item spectra we introduced in the previous section derive the spectra from the individual derivation for each test case. In the second approach, we extract a single spectrum from *all possible derivations* that consume prefixes of the maximal viable prefix.

Definition 4.2.3 (shift item spectrum)**.** Let u be the maximal viable k-prefix of w. Then

$$R^\bullet = \bigcup_{w'=uv\in L(G)} \bigcup_{u'\preceq_i u,\, i\leq k}$$
$$\{p^\bullet \mid \Delta = S \overset{i}{\Rightarrow}{}^*_{w'} u'\omega, \alpha A\beta \Rightarrow_p \alpha\gamma\beta \in \Delta,\ p^\bullet = A \to \mu \bullet \nu \in \text{items}(p),$$
$$\exists x \in T^* \cdot \alpha\mu x \Rightarrow^* u'\}$$

is called the *shift item spectrum* for w.

Definition 4.2.3 formulates this intuition. It considers all right completions v of the maximal viable prefix u, and then all maximal prefix-bounded derivations of the prefixes of u; called u' from these derivations, it extracts the items of all applied rules. Note that this definition does not explicitly consider the frontier rules (cf. Definition 4.2.2) because they are implied by the different right completions. Note also that Definition 4.2.3 gives larger and denser spectrum than Definition 4.2.1 if $w \in L(G)$ because it considers all possible completions of valid prefixes, while the plain positive item spectra only considers items for successful rule applications.

Worked Example. Table 4.2 shows the corresponding results for shift item spectra (see Definition 4.2.3). It is easy to see that these are indeed both larger (i.e., have more non-zero suspiciousness scores, at 38 compared to 25) and denser (i.e., items are associated with more test cases) than the plain item spectra. However, this seeming loss of precision does not necessarily translate into a worse localization performance, because the metrics are based on the spectral difference between passing and failing tests.

In fact, in this case, the results *improve* noticeably. All four metrics now assign the highest suspiciousness scores to three items from the **`while`**-rule (i.e., *stmt*:3:1, *stmt*:3:2 and *stmt*:3:3). Tarantula and Jaccard both rank the second fault (i.e., *stmt*:2:4) tied fourth, Ochiai ranks it tied fifth, while DStar still struggles and ranks it tied twelfth. Overall, we have on average to look at three positions 3 (i.e., 3.7% of all positions) in 2 rules before we find both faults using Tarantula or Jaccard, 9.5 positions (11.7%) in 9 rules using Ochiai, and 13 positions (16.0%) in 7 rules using DStar.

4.2.3 Specialized Tie Breaking Strategy

Since item-level localization induces larger and denser spectra than rule-level localization, ties are more common and longer than in rule-level localization. Tables 4.1 and 4.2 clearly illustrate this. Our challenge is to reduce the sizes of the ties, and ideally to rank the faulty items uniquely at the top of the tied group. Here, we can take advantage of contextual information such as the type of test suites used or even the structure of the grammar under test to break ties on the fly. In the worked example, we used a test suite with positive tests only, so we can resolve ties between items from the same rule in favour of the item with the largest designated position (and in fact even drop items with subsumed designated positions entirely from consideration). This improves the ranking and reduces the average wasted effort, as the resolved ranks in Tables 4.1 and 4.2 show.

Formally, we propose the *k-max* tie breaking mechanism. The basic idea of tie resolution using *k-max* is to resolve ties in favour of the item from a grammar rule *r* with the larger designated position (i.e., further to the right of the rule) over other items from the same rule *r* with smaller designated positions. If there exists a tie from items of different grammar rules, *k-max* picks from each rule the item with the highest position. The other items are

Table 4.2: Shift item spectra, suspiciousness scores, ranks, and resolved ranks for the faulty grammar version G'_{toy} and *rule* test suite. We use the same layout as in Table 4.1.

item	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	*ep*	*np*	*ef*	*nf*	$\mathcal{T}$			$\mathcal{O}$			$\mathcal{J}$			$\mathcal{D}$		
prog:1:0	✓	✓	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	14	0	2	0	0.50	=18		0.35	=16		0.13	=17		0.29	=16	
prog:1:1	✓	✓	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	14	0	2	0	0.50	=18		0.35	=16		0.13	=17		0.29	=16	
prog:1:2	✓	✓	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	14	0	2	0	0.50	=18		0.35	=16		0.13	=17		0.29	=16	
prog:1:3	✓	✓	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	14	0	2	0	0.50	=18	=9	0.35	=16	=8	0.13	=17	=9	0.29	=16	=8
block:1:0	✓	✓	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	14	0	2	0	0.50	=18		0.35	=16		0.13	=17		0.29	=16	
block:1:1	✓	✓	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	14	0	2	0	0.50	=18	=9	0.35	=16	=8	0.13	=17	=9	0.29	=16	=8
block:2:0	✓	✓	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	14	0	2	0	0.50	=18		0.35	=16		0.13	=17		0.29	=16	
block:2:1	✓	✓	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	14	0	2	0	0.50	=18	=9	0.35	=16	=8	0.13	=17	=9	0.29	=16	=8
block:3:0	✓	✓	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	14	0	2	0	0.50	=18		0.35	=16		0.13	=17		0.29	=16	
block:3:1	✓	✓	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	14	0	2	0	0.50	=18	=9	0.35	=16	=8	0.13	=17	=9	0.29	=16	=8
block:4:0	✓	✓	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	14	0	2	0	0.50	=18		0.35	=16		0.13	=17		0.29	=16	
block:4:1	✓	✓	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	14	0	2	0	0.50	=18	=9	0.35	=16	=8	0.13	=17	=9	0.29	=16	=8
decls:1:0	✓	✓	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	14	0	2	0	0.50	=18	=9	0.35	=16	=8	0.13	=17	=9	0.29	=16	=8
decls:2:0	✓	✓	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	14	0	2	0	0.50	=18	=9	0.35	=16	=8	0.13	=17	=9	0.29	=16	=8
decl:1:0	✓	✓	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	14	0	2	0	0.50	=18	=9	0.35	=16	=8	0.13	=17	=9	0.29	=16	=8
stmts:1:0	✓	✓	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	14	0	2	0	0.50	=18	=9	0.35	=16	=8	0.13	=17	=9	0.29	=16	=8
stmts:2:0	✓	✓	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	14	0	2	0	0.50	=18	=9	0.35	=16	=8	0.13	=17	=9	0.29	=16	=8
stmt:1:0	✓	✓	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	14	0	2	0	0.50	=18	=9	0.35	=16	=8	0.13	=17	=9	0.29	=16	=8
stmt:1:1						✗	✓	✓	✓	✓							4	10	1	1	0.64	17	8	0.32	37	21	0.17	16	8	0.20	37	21
stmt:2:0	✓	✓	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	14	0	2	0	0.50	=18	=9	0.35	=16	=8	0.13	=17	=9	0.29	=16	=8
stmt:2:1						✗	✓										1	13	1	1	0.88	=4		0.50	=5		0.33	=4		0.50	=12	
stmt:2:2						✗	✓										1	13	1	1	0.88	=4		0.50	=5		0.33	=4		0.50	=12	
stmt:2:3						✗	✓										1	13	1	1	0.88	=4		0.50	=5		0.33	=4		0.50	=12	
stmt:2:4						✗	✓										**1**	**13**	**1**	**1**	**0.88**	**=4**	**2**	**0.50**	**=5**	**=3**	**0.33**	**=4**	**2**	**0.50**	**=12**	**7**
stmt:3:0	✓	✓	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	14	0	2	0	0.50	=18	=9	0.35	=16	=8	0.13	=17	=9	0.29	=16	=8
stmt:3:1														✗			0	14	1	1	1.00	=1		0.71	=1		0.50	=1		1.00	=1	
stmt:3:2														✗			0	14	1	1	1.00	=1		0.71	=1		0.50	=1		1.00	=1	
stmt:3:3														✗			**0**	**14**	**1**	**1**	**1.00**	**=1**	**1**	**0.71**	**=1**	**1**	**0.50**	**=1**	**1**	**1.00**	**=1**	**1**
stmt:4:0	✓	✓	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	14	0	2	0	0.50	=18	=9	0.35	=16	=8	0.13	=17	=9	0.29	=16	=8
stmt:5:0	✓	✓	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	14	0	2	0	0.50	=18	=9	0.35	=16	=8	0.13	=17	=9	0.29	=16	=8
expr:1:0	✓	✓	✓	✓	✓	✗	✓							✗			6	8	2	0	0.70	=9		0.50	=5		0.25	=9		0.67	=5	
expr:1:1	✓	✓	✓	✓	✓	✗	✓							✗			6	8	2	0	0.70	=9	=4	0.50	=5	=3	0.25	=9	=4	0.67	=5	=3
expr:2:0	✓	✓	✓	✓	✓	✗	✓							✗			6	8	2	0	0.70	=9		0.50	=5		0.25	=9		0.67	=5	
expr:2:1	✓	✓	✓	✓	✓	✗	✓							✗			6	8	2	0	0.70	=9	=4	0.50	=5	=3	0.25	=9	=4	0.67	=5	=3
expr:3:0	✓	✓	✓	✓	✓	✗	✓							✗			6	8	2	0	0.70	=9	=4	0.50	=5	=3	0.25	=9	=4	0.67	=5	=3
expr:4:0	✓	✓	✓	✓	✓	✗	✓							✗			6	8	2	0	0.70	=9		0.50	=5		0.25	=9		0.67	=5	
expr:4:1	✓	✓	✓	✓		✗	✓							✗			5	9	2	0	0.74	8	3	0.53	4	2	0.29	8	3	0.80	4	2
expr:5:0	✓	✓	✓	✓	✓	✗	✓							✗			6	8	2	0	0.70	=9	=4	0.50	=5	=3	0.25	=9	=4	0.67	=5	=3

dropped altogether.[2] Generally, the *k-max* strategy creates two ranks from a set of tied items from different rules, with items with larger designated positions ranked higher.

Table 4.1 illustrates the benefits of the *k-max* tie breaking strategy for plain item spectra. It resolves the respective ties over both faulty rules' items in favour of the actual fault locations; this allows Tarantula to pinpoint one of the faults (i.e., identify it as the single top-ranked item). However, ties over items from different rules remain unresolved (e.g., using Tarantula, *block*:1:1, *expr*:1:1, and *expr*:2:1 remain tied). If we again inspect the items in rank order and resolve the remaining ties arbitrarily, we now have to look only at 5 positions (i.e., 6.1% of all positions) in 4 rules before we find both faults using Tarantula, 7 positions (8.6%) in 5 rules using Jaccard, and 8 positions (9.8%) in 7 rules using Ochiai or DStar.

In the case of shift item spectra (see Table 4.2), the tie resolution strategy is even more effective, and on average we have to look at only 2 positions (2.5%) in 2 rules using Tarantula or Jaccard, 4.5 positions (5.6%) in 5 rules using Ochiai, and 7 positions (8.6%) in 7 rules using DStar to identify the faults.

4.3 Implementation

This section sketches extensions to JavaCC and CUP (which we call SymJavCC and SymCUP respectively) in order to extract item spectra. As we mentioned before, we do not use ANTLR in our evaluation here because our attempts at an extension to item spectra were brittle and produced unreliable results; we leave the integration of precise spectra extraction alongside ANTLR's unbounded lookahead parsing algorithm for future work.

SymJavaCC. An extension to record item spectra is straightforward: we only need to map the actions (i.e., matching tokens and recursive calls to other parse functions) taken in the body of a parse function that implements the rule back to their positional occurrence on the right hand-side of the corresponding rule. We only extract plain item spectra for SymJavaCC and do not support shift item spectra, as they are geared towards LR-parsers.

SymCUP. Plain item spectra logging as per Definitions 4.2.1 and 4.2.2 in SymCUP closely follows the description in Section 3.4; we do not need to map items left on the stack to their corresponding rules, but simply extract them "as they are." For every successful reduction, we log all items of the corresponding reduced rule. The definition of shift item spectra is easy to operationalize in LR-parsers since the parse stack represents a viable prefix,

[2]Note that dropping items may in principle also drop the actual faults. We can therefore also use a *k-max* variant where the subsumed items are simply ranked directly below the items identified by *k-max*.

the spectra is composed from the (kernel and non-kernel) items in the states that are pushed on the stack.

4.4 Evaluation

In this section, we present our experimental evaluation of the item-level localization. We use this to answer our second research question, i.e., how item-level localization compares to rule-level localization. We break this into two sub-questions:

RQ2a: How effective are fault localization techniques based on item-level spectra in identifying seeded single faults in grammars?

RQ2b: Does the use of item spectra improve the localization accuracy?

4.4.1 Experimental Setup

We first evaluate the efficacy of a fine-grained fault localization method that uses items instead of rules as spectral elements. We adapt the same experimental setup as in Section 3.6 using the mutated versions of SIMPL grammars and the test suites and the respective number of killed mutants are the same. The derived golden versions of JavaCC and CUP grammars have 242 and 258 items, respectively. Table 4.3 presents our experimental results in three blocks. The VANILLA configuration uses the default ranking assigned to items based on the suspiciousness scores computed by different metrics. Tied items, i.e., items with the same suspiciousness scores, are assigned a rank using the mid-rank tie breaking mechanism. The second block summarizes results using a *k-max* tie breaking strategy introduced earlier. In both cases, SymCUP$_{\texttt{shift}}$ shows results based on shift item spectra (see Definition 4.2.3) while SymCUP$_{\texttt{plain}}$ shows results based on plain item spectra as per Definitions 4.2.1 and 4.2.2.

The third block serves bench-marking purposes, to enable a fair comparison between item- and rule-level localization. This comparison is based on the insight that when given a correctly predicted fault using rule-level fault localization, we still need to look at all the symbols at the right-hand side of a faulty rule (on the worse case) to find the offending symbol(s). On average, we need to inspect half of the symbols in the rule to identify the extract fault location. We can therefore extend the rule-level spectra to item-level spectra. More specifically, given a rule r with assigned suspiciousness score s, we replace r by a set of all possible items $r^\bullet$ that can be derived from r and assign s to each item of r. For example, consider the worked example in Section 4.1: the rule $stmt \rightarrow$ **`sleep`** from the grammar G'_{Toy} has a Tarantula score of 0.69 is replaced by its two items $stmt \rightarrow \bullet$ **`sleep`** and *stmt* $\rightarrow$ **`sleep`** $\bullet$. Both items get the Tarantula score of 0.69. We then re-rank these reconstructed items using the mid-rank tie breaking strategy.

4.4.2 Effectiveness of Item-Level Localization (RQ2a)

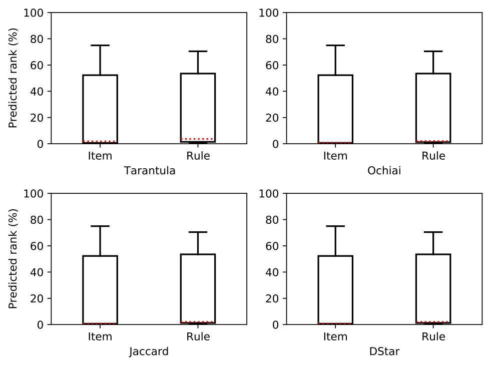
We first focus on the "vanilla" strategy without specialized tie breaking, as shown in the first block of Table 4.3. We can observe the following results. First, as in the rule-level localization, Ochiai, Jaccard, and DStar outperform Tarantula, here even for all test suites and parsing technologies: they give lower mean and median values, and identify more faults. For the smaller test suites (*rule*, *cdrc*, and *instr*), the differences between Ochiai, Jaccard, and DStar are marginal, as for the *large* test suite, Jaccard slightly outperforms Ochiai and DStar for JavaCC but underperforms for both versions of CUP. Second, the mixed test suite *large* yields better results than the other three test suites. Using *large*, we are able to uniquely localize the seeded fault in 6-32% of the cases, and in 24-65% and 32-70% of the cases the fault is localized in the top three and top five of the ranked items respectively. Third, the choice of the parsing technology does have an effect on fault localization. With the exception of low #1 values, item-level localization appears to be more effective in SymJavaCC than in both $\text{SymCUP}_{\texttt{shift}}$ and $\text{SymCUP}_{\texttt{plain}}$ configurations. With SymJavaCC, the fault is typically located at a median rank of 0.6-8.5%, hence, in more than half of the cases the fault is correctly predicted within the top five items. Another interesting insight, and perhaps hardly surprising, is that we cannot tell apart the effectiveness of fault localization based on plain and shift item spectra in the LR case. In particular, $\text{SymCUP}_{\texttt{plain}}$ finds more faults within the top five items, but has slightly worse mean ranks across the board than $\text{SymCUP}_{\texttt{shift}}$.

Tie Breaking. The second block of Table 4.3 summarizes the fault localization results, where we break ties using the *k-max* strategy that picks the item with the highest position among tied items from the same rule. In general, in most cases we see an increase in effectiveness – the median and mean ranks are improved and #1, #3, and #5 numbers increase substantially; in particular, we see an up to 30× increase in the number of seeded faults that are pinpointed exactly (i.e., #1) when we are using small *rule* and *cdrc* test suites, and a 2× increase for *instr*. The relative performance of the different metrics, however, remains largely unaffected.

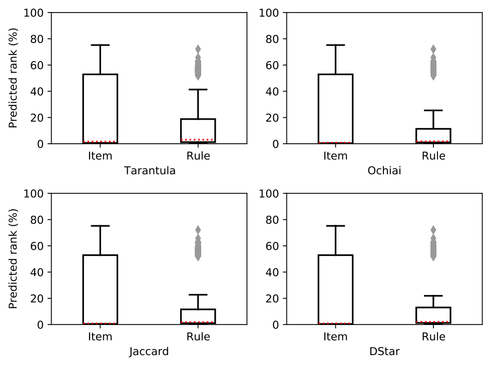
RQ2a: Our fault localization method based on item spectra remains effective in identifying single faults in grammars with seeded faults. The tie breaking mechanism that prefers the item with the highest position over other items from the same rule drastically improves the results.

Table 4.3: Detailed item-level localization results of fault seeding experiments over SIMPL grammars. $\tilde{x}$ and $\bar{x}$ denote the median and mean rank, respectively, of the seeded fault. #1 denotes the number of cases where the metric ranked the seeded fault as most suspicious, #3 and #5 denote the number of cases where the seeded fault was ranked in the top 3 and top 5 most suspicious items respectively.

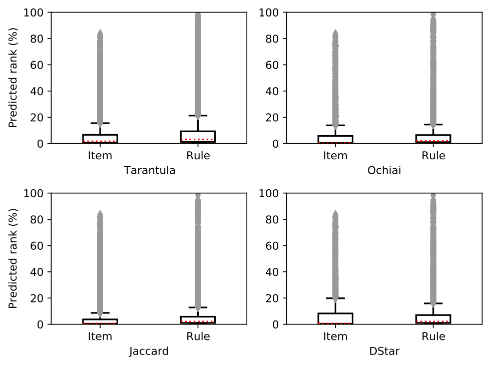
			Tarantula					Ochiai					Jaccard					DStar				
			$\tilde{x}$	$\bar{x}$	#1	#3	#5	$\tilde{x}$	$\bar{x}$	#1	#3	#5	$\tilde{x}$	$\bar{x}$	#1	#3	#5	$\tilde{x}$	$\bar{x}$	#1	#3	#5
VANILLA	SymJavaCC	*rule*	2.9	20.2	120	9931	13548	1.4	19.5	202	13634	16444	1.4	19.5	203	13634	16444	1.4	19.6	202	13566	16370
		cdrc	2.7	20.5	177	10375	13829	1.4	19.9	258	13988	17045	1.4	19.9	259	13989	17045	1.4	19.9	249	13870	16920
		large	1.7	8.3	5211	14731	17988	0.8	7.5	10140	20758	22186	0.8	7.0	9897	20896	22729	0.6	8.5	10183	21115	22142
		instr	8.5	26.7	800	5678	7971	1.2	23.6	4601	16562	18491	1.2	23.7	4579	16291	18226	1.2	23.7	4588	16543	18553
	SymCUP$_{\texttt{shift}}$	*rule*	3.9	14.3	3409	7352	8775	3.3	13.2	4079	8580	10019	3.3	13.2	4079	8579	10017	3.3	13.2	4079	8580	10019
		cdrc	3.9	14.0	3793	7896	9796	2.9	12.8	4491	9175	11381	3.1	12.9	4491	9173	10954	2.9	12.8	4491	9175	11381
		large	5.3	13.5	1716	6368	8696	2.5	10.1	6039	9601	12527	3.9	12.2	5952	8998	11363	1.9	10.6	6661	10501	13464
		instr	11.5	24.0	1442	3266	5248	4.7	16.7	4340	8200	10238	4.7	18.5	4338	7971	10169	3.9	16.2	4402	8398	10411
	SymCUP$_{\texttt{plain}}$	*rule*	2.1	20.0	3126	8869	11563	1.6	19.3	3469	10835	13127	1.6	19.3	3469	10835	13126	1.6	19.3	3469	10835	13127
		cdrc	2.1	20.1	3954	9971	12615	1.4	19.5	4511	12071	14487	1.4	19.5	4511	12071	14486	1.4	19.5	4511	12069	14486
		large	4.3	13.8	2082	7667	10209	1.9	12.5	8842	12298	13603	2.3	12.8	8086	10674	12817	1.6	12.9	10151	12627	13571
		instr	12.2	29.8	1173	4021	5352	3.3	26.4	4029	9246	11318	3.5	26.4	4032	9162	11008	3.1	26.4	4056	9530	11552
k-MAX	SymJavaCC	*rule*	1.9	19.6	5911	12071	15541	0.8	19.1	6926	16836	17449	0.8	19.1	6927	16836	17448	0.8	19.2	6926	16768	17387
		cdrc	1.7	19.9	6238	12675	15967	0.8	19.5	7565	17252	17873	0.8	19.5	7566	17253	17873	0.8	19.5	7556	17154	17757
		large	1.7	8.2	7676	15116	18204	0.6	7.5	13863	20888	22148	0.6	7.0	13566	20964	22662	0.6	8.5	14026	21117	22106
		instr	7.6	26.3	2851	7008	8949	0.8	23.3	9704	17636	18492	0.8	23.4	9673	17267	18129	0.8	23.5	9702	17571	18395
	SymCUP$_{\texttt{shift}}$	*rule*	2.1	13.4	5054	8346	11490	1.4	12.7	6003	9378	13329	1.4	12.7	6003	9377	13327	1.4	12.6	6003	9378	13329
		cdrc	1.8	13.1	5506	9220	12796	1.4	12.2	6492	10321	14907	1.4	12.3	6492	10319	14480	1.4	12.3	6492	10321	14907
		large	5.3	13.5	1800	6424	8700	2.5	10.1	6296	9730	12548	3.9	12.1	6194	9046	11387	1.9	10.6	6932	10508	13476
		instr	10.9	23.7	2005	4990	6841	4.1	16.5	5633	9662	10843	4.1	18.3	5629	9548	10776	3.7	16.0	5699	9858	11015
	SymCUP$_{\texttt{plain}}$	*rule*	1.0	19.1	5447	13250	14022	0.8	18.8	5930	14422	14982	0.8	18.9	5930	14422	14981	0.8	18.9	5930	14422	14982
		cdrc	0.8	19.2	7999	14308	15282	0.8	19.0	8799	15479	16025	0.8	19.0	8799	15477	16022	0.8	19.1	8799	15476	16021
		large	4.1	13.5	2168	7924	10355	1.6	12.2	9119	12443	13772	2.3	12.7	8341	10887	12875	1.6	12.8	10601	12678	13643
		instr	12.0	29.4	1935	5438	7518	3.1	26.1	5734	10841	12184	3.1	26.1	5732	10760	11873	2.7	26.2	5763	11128	12417
RULE	JavaCC	*rule*	3.7	17.9	6	6465	10172	2.1	16.5	14	8597	14449	2.1	16.5	14	8597	14449	2.1	16.5	14	8582	14437
		cdrc	3.1	16.4	16	8260	11776	1.9	15.2	23	10286	15525	1.9	15.2	23	10286	15524	2.1	15.5	19	9700	14929
		large	3.1	12.1	19	9850	13217	2.1	8.9	23	11412	16521	2.1	8.4	24	11527	16470	2.1	9.7	22	11279	16455
		instr	17.4	32.0	0	1788	4250	4.5	27.3	4	5481	11554	4.5	26.6	4	5368	11066	4.5	27.2	4	5530	11597
	CUP	*rule*	4.5	24.9	0	4882	6777	3.5	24.3	0	6194	8982	3.5	24.3	0	6194	8982	3.5	24.3	0	6194	8982
		cdrc	4.3	25.2	0	5458	7462	3.3	24.6	0	6777	9514	3.3	24.6	0	6777	9514	3.3	24.6	0	6775	9511
		large	3.1	11.5	6	6289	10291	3.7	15.0	3	6932	11059	3.5	14.9	4	6409	10615	3.7	17.2	3	6954	11052
		instr	15.2	36.1	2	1997	3186	7.6	33.4	3	5403	7884	7.8	33.9	3	5017	7502	6.4	33.3	3	5648	8373



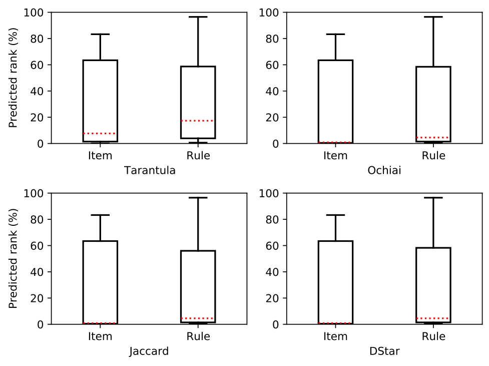
(a) Results for *rule* test suite.



(b) Results for *cdrc* test suite.

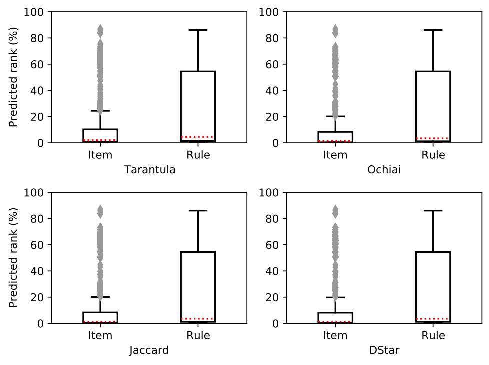


(c) Results for *large* test suite.

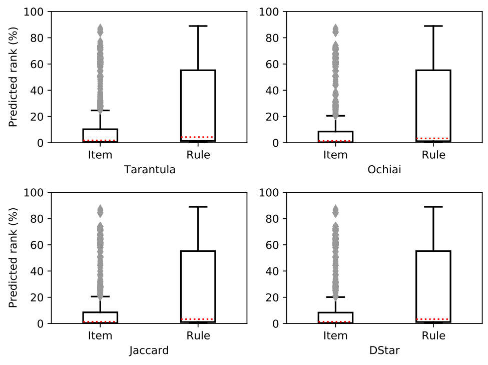


(d) Results for *instr* test suite.

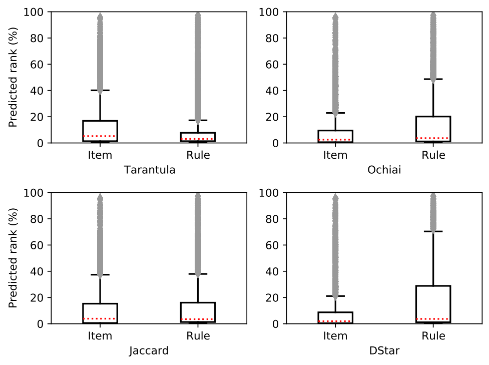
Figure 4.2: Comparison results of item- and rule-level localization over fault-seeded SIMPL grammars using SymJavaCC. Top left sub-plots show the comparison on *rule* test suite, top right *cdrc*, bottom left *large* and bottom right *instr*. For each test suite, plots are shown for each of the four metrics Tarantula, Ochiai, Jaccard, and DStar. Each diagram shows two boxplots for the item-level and the corresponding rule-level results, where all items in the rule are ranked equally. The boxplots have the same structure as described in Section 3.6.1; outliers outside the 95th percentile are denoted by individual grey diamonds (which can overlap in the plot).



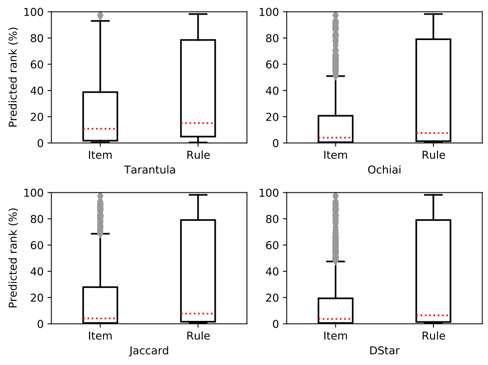
(a) Results for *rule* test suite.



(b) Results for *cdrc* test suite.

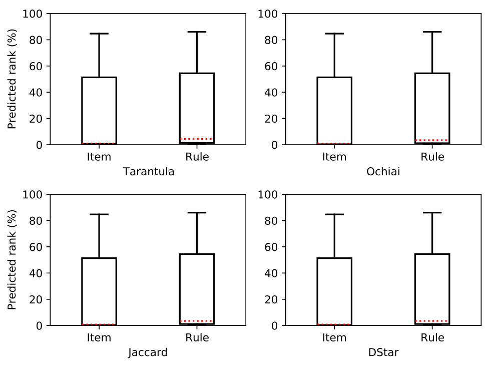


(c) Results for *large* test suite.

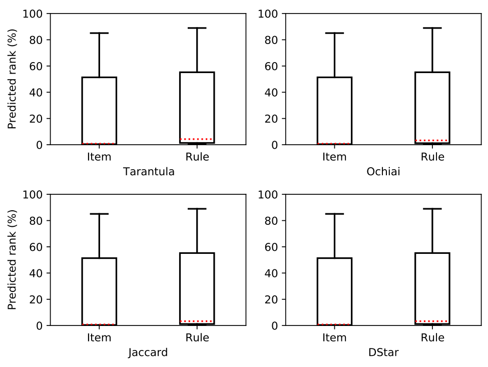


(d) Results for *instr* test suite.

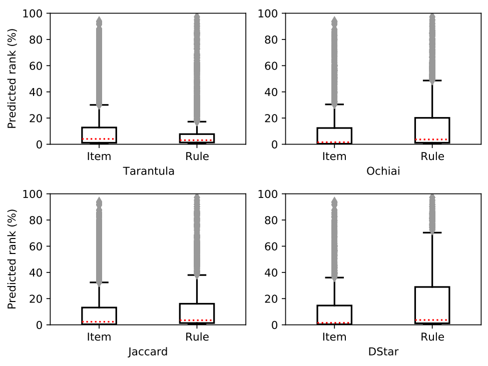
Figure 4.3: Comparison results of item- and rule-level localization over fault seeded SIMPL grammars under SymCUP$_{\texttt{shift}}$. Boxplots layout as in Figure 4.2.



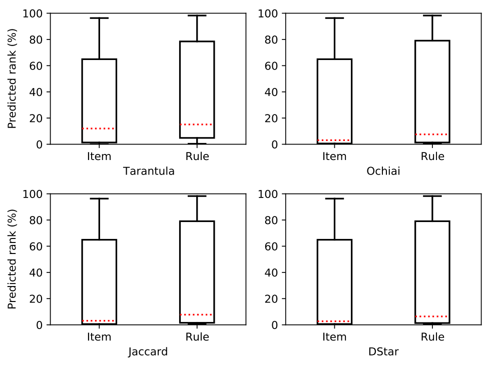
(a) Results for *rule* test suite.



(b) Results for *cdrc* test suite.



(c) Results for *large* test suite.



(d) Results for *instr* test suite.

Figure 4.4: Comparison results of item- and rule-level localization over fault seeded SIMPL grammars under SymCUP$_{\texttt{plain}}$. Boxplots layout as in Figure 4.2.

4.4.3 Item- vs Rule-level Localization (RQ2b)

Here we address RQ2b, Figures 4.2 to 4.4 summarize the comparison of item- and rule-level fault localization as a series of paired box plots. Each pair contains ranks from item spectra and ranks from rule spectra computed using each ranking metric over each test suite on each of the two parsing mechanisms. Similarly, here, the upper and lower ends of the box in each box plot represent the 75th and 25th percentile, respectively. The median is also shown as a line that cuts across the box. We derive the comparison from the second block (*k*-MAX) and the third block (RULE) from Table 4.3.

While results differ with the applied parsing mechanism and the underlying test suite, it is easily observable that, the *k-max* strategy performs better than rule-level localization. This is made evident by better median (0.6-10.9% vs 2.1-17.2%) and mean ranks. Note that according to our intuition of replacing each rule in the rule spectra by its set of items, we cannot make comparisons based on the #1/#3/#5 values.

RQ2b: Item-level localization with the specialized tie breaking mechanism outperforms the simplistic extension of rule-level localization where all positions within a rule are assigned the same score.

4.5 Threats to Validity

Our experiments in this chapter are subject to similar threats to validity as those in Section 3.6.1, because we use the same experimental setup, and in particular, the same base grammars. However, in these experiments we introduce more heuristic elements, especially in our handling of ties, the *k-max* tie breaking strategy is modelled on positive tests and while we got better results for the larger mixed test suite which contain negative tests, *k-max* may not generalize to other grammars, parsing technologies and other ranking metrics.

Another threat to validity is that all our insights are based on fault seeding experiments and we may make different observations on grammars with real and multiple faults. However, in the next chapter, we use item-level localization to identify faults in a fully automatic repair framework and see good results there.

4.6 Conclusion

This chapter described our item-level localization approach. We formally defined three types of item spectra and showed how the parser generator tools JavaCC and CUP can further be extended to capture these spectra during parsing. We also proposed a simple yet effective tie breaking mechanism that prefers items with the highest positions within ties of items from the same rules.

We showed that fault localization based on item spectra remains effective and that our tie breaking mechanism drastically improves the results. We further demonstrated that we can effectively achieve better precision using specialized item spectra than simply extending rule spectra.

Chapter 5

Automatic Grammar Repair

In this chapter, we describe the automation of the manual find-and-fix cycle illustrated in Section 1.1 and used in the experimental evaluation of fault localization described in Section 3.6.3 and Section 3.6.4. We first formalize our repair framework in Section 5.1; in particular, we then describe two algorithms, the passive repair algorithm, in Section 5.1.7 and the active repair algorithm in Section 5.1.8.

Sections 5.2, 5.3 and 5.4 describe grammar transformations that we use to patch faulty input grammars. We use an example G_{test} in Figure 5.1 to illustrate each grammar transformation by injecting some faults, some of which are modelled on real faults introduced by students.

Section 5.5 describes our realization of the approach by the highly configurable gfixr prototype tool that takes as input a test suite specification that we repair against, an optional oracle that that answers additional membership queries, and the input grammar G.

We then demonstrate the efficacy of our approach through a series of experiments in Section 5.6. Sections 5.6.2, 5.6.3 and 5.6.4 summarize our results of repairing 33 grammars written by students. We consider both repair configurations (passive and active repair) of our approach in the evaluation and compare both methods like-for-like in Section 5.6.4. We conclude the chapter with the discussion of the limitations and challenges that our approach faces in Section 5.7.

5.1 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 2 and 3; more implementation details are given in Section 5.5.

5.1.1 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., $TS^+ \nsubseteq L(G)$ or $TS^- \cap L(G) \neq \emptyset$). The *repair problem* is then to construct from $TS_{\mathcal{L}}$ and G a "similar" CFG G' that accepts all positive tests (i.e., $TS^+ \subseteq L(G')$) and rejects all negative tests (i.e., $TS^- \cap L(G') = \emptyset$) and so approximates $\mathcal{L}$ better than G. We require in the following that the test suite $TS_{\mathcal{L}}$ is *viable for* G, i.e.,

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

(ii) it is *constructive*, i.e., $TS^- \subseteq L(G)$.

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

However, the problem is underspecified and a repair can "overgeneralize", i.e., $TS^+ \subseteq L(G') \nsubseteq \mathcal{L}$. We can therefore evaluate the quality of our repairs only through manual inspection or based on performance over an additional *validation suite*.

5.1.2 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 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 $TS^- \cap L(G') = \emptyset$.

5.1.3 Spectrum-Based Fault Localization for Repair

We have experimented with different variants of item spectra (see the previous chapter), 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 (see Section 2.4) can be used based on the definitions of the counts given above.

5.1.4 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.

5.1.5 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 [130] and our repair attempts to break this property. We define the sets of good tokens T_p^+ and bad tokens T_p^- for an item p as the sets of good and bad tokens from the failing tests in which p is executed, i.e., $(T_p^+, T_p^-) = \bigcup\{(a, b)_w \mid p \sim_{\text{✗}} w, w \in TS^-\}$ (where the union is taken componentwise). Examples of these will be shown in subsequent sections.

5.1.6 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 that the terminals that can occur directly to the left and right of the repair site 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.

5.1.7 Passive Repair Algorithm

Algorithm 2 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.

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

Algorithm 2: The passive repair algorithm

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

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.

5.1.8 Active Repair Algorithm

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

Algorithm 3: The active repair algorithm

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

this is similar to Angluin's active learning [11][1], we call this the active repair setting.

Algorithm 3 presents a high-level description of this active repair approach. We extend and build on the passive repair algorithm shown in Algorithm 2. There are several differences in the flow of the search that we highlight. First, Algorithm 3 takes as input an extra variable, a boolean valued oracle $\mathcal{O}$.

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

[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.

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 $\mathcal{O}$, i.e., passes all tests in TS and thus TS_{init}.

5.2 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.

5.2.1 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 $TS'_{test} \supseteq TS_{test}$ (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_{test} because neither *paramlist* nor *arglist* are nullable, and their addition to TS_{test} can be seen as a "repair request" to change G_{test} to allow empty formal parameter and argument lists.

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

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

In these all cases, the sets of good and bad tokens are $T^+ = \{\ \textbf{(}\ \}$ and $T^- = \{\ \textbf{)}\ \}$, 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 $\{\ \textbf{(}\ \}$ 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

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 5.1: BNF baseline grammar G_{test} suitable for CUP.

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'_{test}:

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

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 $\mathit{name} \Rightarrow_{G'} \texttt{id}\,\texttt{(}\,\mathit{namelist}\,\texttt{)} \Rightarrow^*_{G'} \texttt{id}\,\texttt{(}\ \texttt{,}\,\texttt{id}\,\texttt{)}$. 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 5.2.1 (synchronization)**.** Let $p = A \to \alpha \bullet \beta\omega$ be an item in $P^\bullet$ with $\mathrm{left}(p) \subseteq T_p^+$.

(a) If $\omega = X\gamma$ with X non-nullable and $T_p^- \subseteq \mathrm{first}(X)$, let $\eth(p,\beta) = A \to \alpha \bullet \omega$ be the result of deleting β at the designated position. Then $p \rightsquigarrow \eth(p,\beta)$ is a *synchronization patch*.

(b) If ω is nullable and $T_p^- \subseteq \mathrm{follow}(A)$, let $\eth(p,\beta\omega) = A \to \alpha\bullet$ be the result of deleting $\beta\omega$ at the designated position. Then $p \rightsquigarrow \eth(p,\beta\omega)$ 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 5.2.2 (synchronization validation)**.** Let $p = A \to \alpha \bullet \beta\omega$ be an item in $P^\bullet$. The synchronization patch $G \rightsquigarrow_{\eth(p,\beta)} G'$ is *validated* over $TS_{\mathcal{L}}$ if $\mathrm{left}_{G'}(\eth(p,\beta)) \times \mathrm{right}_{G'}(\eth(p,\beta)) \subseteq \Gamma_2(TS_{\mathcal{L}})$.

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

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

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

The rules are created from the corresponding baseline rules by the deletion of a single symbol at the identified fault locations shown above, and are added to the grammar, rather than replacing the baseline rules, because the latter are used in other passing tests. gfixr finds this fix with three patches in roughly two minutes, generating 26 candidate grammars.[2] Note that the initially top-ranked item $param \to \bullet\, type$ **array** `id` induces a rule $param \to \varepsilon$ through a panic mode synchronization, but this fails the patch validation and gets ruled out because $\Gamma_2(TS'_{test})$ does not contain the induced bigram (**(** , **,**).

In the variant G'_{test} 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_{test} (see Fig. 5.1) 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.

5.2.2 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_{test}). 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., $T_p^- \subset \mathrm{right}(A \to \alpha \bullet \omega)$) and the patch thus covers *all* failing tests associated with the item, we check here only for a non-empty intersection (i.e., $T_p^- \cap \mathrm{right}(A \to \alpha \bullet \omega) \neq \emptyset$), 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 5.2.3 (symbol insertion)**.** Let $p = A \to \alpha \bullet \omega$ be an item in $P^\bullet$ with $\mathrm{left}(p) \subseteq T_p^+$, and $\mathrm{i}(p, X) = A \to \alpha \bullet X\omega$ be the result of inserting $X \in V$ at the designated position of p. If $T_p^- \cap \mathrm{right}(\mathrm{i}(p,X)) \neq \emptyset$, then $p \rightsquigarrow \mathrm{i}(p,X)$ is an *insertion patch.*

We validate insertion patches by checking the same condition as for synchronization patches, with the designated position *before* the inserted sym-

[2]All runtimes given in Section 5.2 to Section 5.4 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 5.6 uses a different computational setup, and times are not necessarily comparable.

bol; 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 5.2.4 (insertion validation)**.** Let $p = A \to \alpha \bullet \omega$ be an item in $P^{\bullet}$ and $X \in V$. The insertion patch $G \leadsto_{\mathfrak{i}(p,X)} G'$ is *validated* over $TS_{\mathcal{L}}$ if $\text{left}_{G'}(\mathfrak{i}(p,X)) \times \text{right}_{G'}(\mathfrak{i}(p,X)) \subseteq \Gamma_2(TS_{\mathcal{L}})$.

Example Repair. If we remove the rule

$$\mathit{fdecl} \to \textbf{define}\ \texttt{id}\ \textbf{(}\ \textbf{)}\ \textbf{->}\ \mathit{type}\ \texttt{id}\ \mathit{body}$$

from G_{test}, 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.

5.2.3 Symbol Substitutions

Substitution patches fix bugs where grammar developers have used a wrong symbol, as shown in the example from the introduction (see Section 1.1). Such bugs are particularly difficult to detect when the grammar is either too permissive (e.g., $\mathit{name} \to \texttt{id}\ \textbf{[}\ \mathit{expr}\ \textbf{]}$) or too restrictive, in a way that is only uncovered by structurally complex tests (e.g., $\mathit{paramlist} \to \mathit{param} \mid \mathit{param}\ \textbf{,}\ \mathit{param}$). A substitution patch replaces the symbol at the designated position by another one that "eats up" at least one of the bad tokens.

Definition 5.2.5 (symbol substitution)**.** Let $p = A \to \alpha \bullet X\omega$ be an item in $P^{\bullet}$ with $\text{left}(p) \subseteq T_p^+$, $Y \in V$, and $\mathfrak{s}(p,Y) = A \to \alpha \bullet Y\omega$ be the result of replacing X at the designated position by Y. If $T_p^- \cap \text{right}(A \to \alpha \bullet Y\omega) \neq \emptyset$, then $p \leadsto \mathfrak{s}(p,Y)$ 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 5.2.6 (substitution validation)**.** Let $p = A \to \alpha \bullet X\omega$ be an item in $P^{\bullet}$ and $Y \in V$. The substitution patch $G \leadsto_{\mathfrak{s}(p,Y)} G'$ is *validated* over $TS_{\mathcal{L}}$ if

(*i*) $\text{left}_{G'}(\mathfrak{s}(p,Y)) \times \text{right}_{G'}(\mathfrak{s}(p,Y)) \subseteq \Gamma_2(TS_{\mathcal{L}})$, and

(*ii*) $\text{left}_{G'}(A \to \alpha Y \bullet \omega) \times \text{right}_{G'}(A \to \alpha Y \bullet \omega) \subseteq \Gamma_2(TS_{\mathcal{L}})$.

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 $A \to \alpha \bullet X\omega \leadsto A \to$

$\alpha YZ \bullet \omega$ 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_{test} where the rule: *factor* $\to$ **(** *expr* **)** is missing, so that it rejects bracketed expressions. The following five tests fail from TS_{test}

```
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* $\to$ $\bullet$`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* $\to$ $\bullet$ **not** *factor* $\leadsto$ *factor* $\to$ $\bullet$ **(** *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*).

5.2.4 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_{test} that has the following bug in the *idlist*-rule

$$\begin{aligned} \textit{idlist} &\to \texttt{id}\ \textit{idlisttail} \\ \textit{idlisttail} &\to \bullet\ \texttt{id}\ \textbf{,}\ \textit{idlisttail} \mid \varepsilon \end{aligned}$$

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 5.2.7 (symbol transposition). Let $p = A \to \alpha \bullet XY\omega$ be an item in $P^\bullet$ with $\mathrm{left}(p) \subseteq T_p^+$, and $\mathrm{t}(p) = A \to \alpha \bullet YX\omega$ be the result of swapping the symbols X and Y at the designated position. If $T_p^- \cap \mathrm{right}(\mathrm{t}(p)) \neq \emptyset$, then $p \leadsto \mathrm{t}(p)$ 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 5.2.8 (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*) $\mathrm{left}_{G'}(\mathfrak{t}(p)) \times \mathrm{right}_{G'}(\mathfrak{t}(p)) \subseteq \Gamma_2(TS_{\mathcal{L}})$, and

(*ii*) $\mathrm{left}_{G'}(A \to \alpha Y \bullet X\omega) \times \mathrm{right}_{G'}(A \to \alpha Y \bullet X\omega) \subseteq \Gamma_2(TS_{\mathcal{L}})$.

(*iii*) $\mathrm{left}_{G'}(A \to \alpha YX \bullet \omega) \times \mathrm{right}_{G'}(A \to \alpha YX \bullet \omega) \subseteq \Gamma_2(TS_{\mathcal{L}})$.

5.3 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*.

5.3.1 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_{test} submitted by a student where the the *body*-, *vdecllist*-, and *vdecl*-rules in G_{test} are replaced by the following rules:

$$\begin{aligned} body &\to \texttt{begin}\ vdecls\ stmts\ \texttt{end} \\ vdecls &\to type\ \texttt{id}\ idlist\ \texttt{;} \bullet \mid \varepsilon \end{aligned}$$

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 $\bullet$-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 5.3.1 (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 $\mathrm{first}(A) \subseteq T^-$.

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

In the example, gfixr finds the single patch fix $\mathit{vdecls} \rightarrow \mathit{type}\ \texttt{id}\ \mathit{idlist}\ \texttt{;}\ \mathit{vdecls}$ in roughly 30 seconds, generating only five candidate grammars.

5.3.2 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, the *body*-rule in G_{test} is replaced by the following rules:

$$\begin{aligned} \mathit{body} \rightarrow\ & \textbf{begin}\ \mathit{stmts}\ \textbf{end} \\ \mid\ & \textbf{begin}\ \mathit{type}\ \texttt{id}\ \texttt{;}\ \bullet\ \mathit{stmts}\ \textbf{end} \end{aligned}$$

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_{test} 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 nonterminal 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 5.3.3 (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, $T_p^+ \subseteq \text{left}(A \to \alpha\gamma \bullet \omega)$ and $T_p^- \cap \text{right}(A \to \alpha\gamma \bullet B\omega) \neq \emptyset$.

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

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 5.2.2 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_{test}

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

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 $T^+ = \{\,\textbf{)}\,,\textbf{a}\,,\textbf{0}\,\}$, $T^- = \{\,\textbf{,}\,\}$ and $T^* = \{\,\textbf{(}\,,\textbf{a}\,,\textbf{0}\,\}$ respectively.

gfixr reconstructs it correctly to the following rules.

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

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

5.4 Language Tightening Transformations

The example grammar in Figure 5.1 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*:

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

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.

5.4.1 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 5.4.1 (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.

5.4.2 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* $\to$ **`read`** *name*, which only fails in combination with *name* $\to$ `id` **`(`** *arglist* **`)`** .

We therefore need an enabling patch that moves rules into the right contexts (similar in spirit to CDRC coverage [82]) and so separates out passing and failing rule applications.

Definition 5.4.2 (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 $G \rightsquigarrow_{\mathfrak{S}(p,B)} G'$ is a *non-terminal splitting patch* for B.

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

Example Repair. We repaired the idiosyncrasies in G_{test} with a $step_2$ [150] test suite with 159 positive tests and seven negative tests, including the test

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

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

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

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

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

5.4.3 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.* [99] 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 5.4.3 (token splitting)**.** Let $G = (N, T, P, S)$, $p = A \to \alpha a \omega \bullet \in P^\bullet$ a reduction item and $\mathrm{fail}(p) \subseteq TS^-$. 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 $G \leadsto_{\mathfrak{T}(p,a)} G'$ 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`

$$\begin{aligned} \textit{simple} &\rightarrow \texttt{ADDOP}\ \textit{termlist} \bullet \mid \textit{termlist} \\ \textit{termlist} &\rightarrow \textit{term} \mid \textit{termlist}\ \texttt{ADDOP}\ \textit{term} \\ \texttt{ADDOP} &\rightarrow \texttt{-} \mid \texttt{+} \mid \texttt{or} \end{aligned}$$

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 three lexemes. This gives us

$$\begin{aligned} \textit{simple} &\rightarrow \texttt{-}\ \textit{termlist} \mid \texttt{+}\ \textit{termlist} \mid \texttt{or}\ \textit{termlist} \mid \textit{termlist} \\ \textit{termlist} &\rightarrow \textit{term} \mid \textit{termlist}\ \texttt{-}\ \textit{term} \mid \textit{termlist}\ \texttt{+}\ \textit{term} \mid \textit{termlist}\ \texttt{or}\ \textit{term} \end{aligned}$$

In the second iteration, a rule deletion patch is applied to rules $\textit{simple} \rightarrow \texttt{+}\,\textit{termlist}$ and $\textit{simple} \rightarrow \texttt{or}\,\textit{termlist}$, which leaves us with full fix:

$$\begin{aligned} \textit{simple} &\rightarrow \texttt{-}\ \textit{termlist} \mid \textit{termlist} \\ \textit{termlist} &\rightarrow \textit{term} \mid \textit{termlist}\ \texttt{-}\ \textit{term} \mid \textit{termlist}\ \texttt{+}\ \textit{term} \mid \textit{termlist}\ \texttt{or}\ \textit{term} \end{aligned}$$

5.4.4 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 5.3. Consider for example a variant of G_{test} submitted by a student where the *expr*-rules in G_{test} are replaced by the following rules:

$$\textit{expr} \rightarrow \textit{expr relop simple} \mid \textit{simple}$$

The target language restricts relational operators (*relop*) to only two simple terms as arguments, but the first alternative *expr*-rule over-generalizes this and allows any number of simple terms and therefore 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
```

Below, we introduce two versions of the recursion elimination patches that target immediate left (resp. right) recursion. These patches introduce a fresh non-terminal that is defined over the non-recursive alternatives and replaces the recursive occurrence.

Definition 5.4.4 (immediate left recursion elimination)**.** Let $G = (N, T, P, S)$, $p = A \to A\omega\bullet \in P^\bullet$ a reduction item with other alternatives for A, $P_A = \{A \to \alpha_i\}_{i>1}$, $ep(p) > 0$, $ef(p) > 0$, and $\text{fail}(p) \subseteq TS^-$. If $G' = (N', T, P', S)$ is a CFG with $N' = N \cup \{B\}$, $B \notin N$, and $P' = P \setminus \{p\} \cup \{A \to B\omega, A \to B, B \to \alpha_i\}$ then $G \rightsquigarrow_{\mathfrak{E}_l(p,A)} G'$ is the *immediate left recursion elimination patch* for A.

Definition 5.4.5 (immediate right recursion elimination)**.** Let $G = (N, T, P, S)$, $p = A \to \alpha A\bullet \in P^\bullet$ a reduction item with other alternatives for A, $P_A = \{A \to \beta_i\}_{i>1}$, $ep(p) > 0$, $ef(p) > 0$, and $\text{fail}(p) \subseteq TS^-$. If $G' = (N', T, P', S)$ is a CFG with $N' = N \cup \{B\}$, $B \notin N$, and $P' = P \setminus \{p\} \cup \{A \to \alpha B, A \to B, B \to \beta_i\}$ then $G \rightsquigarrow_{\mathfrak{E}_r(p,A)} G'$ is the *immediate right recursion elimination patch* for A.

We can easily extend the definitions for general immediate recursion of the form $A \to \alpha A \beta$, but we leave its support and evaluation for future work.

Example repair. gfixr prevents the over-generalization in the *expr*-rule by transforming the rules as follows:

$$\begin{aligned} expr &\to rest\ relop\ simple \mid rest \\ rest &\to simple \end{aligned}$$

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

5.4.5 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.

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

The above rules capture all syntactically valid function calls but also generalizes to ill-formed tests, the grammar 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 5.4.6 and Definition 5.4.7 formulate the left and right recursive variations, respectively.

Definition 5.4.6 (push-down list elements)**.** Let $G = (N, T, P, S)$, $p = A \to A\gamma v \bullet \in P^{\bullet}$ a reduction item with other alternatives for A, $P_A = \{A \to v, A \to \omega_i\}_{i>1}$, $ep(p) > 0$, $ef(p) > 0$, and fail$(p) \subseteq TS^{-}$. If $G' = (N', T, P', S)$ is a CFG with $N' = N \cup \{B\}$, $B \notin N$, and $P' = P \setminus \{p\} \cup \{A \to B, A \to \omega_i, B \to B\gamma v, B \to v\}$ then $G \rightsquigarrow_{\mathfrak{P}_l(p,A)} G'$ is the *push-down list elements patch* for A.

Definition 5.4.7 (push-down list elements)**.** Let $G = (N, T, P, S)$, $p = A \to \alpha\beta A \bullet \in P^{\bullet}$ a reduction item with other alternatives for A, $P_A = \{A \to \alpha, A \to \gamma_i\}_{i>1}$, $ep(p) > 0$, $ef(p) > 0$, and fail$(p) \subseteq TS^{-}$. If $G' = (N', T, P', S)$ is a CFG with $N' = N \cup \{B\}$, $B \notin N$, and $P' = P \setminus \{p\} \cup \{A \to B, A \to \gamma_i, B \to \alpha\beta B, B \to \alpha\}$ 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 repairs*.** gfixr finds the fix for the above over-generalization in about 4 minutes and generated 106 candidate patches.

$$\begin{array}{lll} \mathit{expr\_list} & \to & \mathit{exprs} \mid \varepsilon \\ \mathit{exprs} & \to & \mathit{exprs}\ \textbf{,}\ \mathit{expr} \mid \mathit{expr} \end{array}$$

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

5.5 Implementation

We have prototyped both passive and active repair approaches, as described in the previous sections in the gfixr tool.

***System Architecture*.** gfixr implements the repair loop variations shown in Algorithms 2 and 3. 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 re-configured easily.

The input oracle $\mathcal{O}$ 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 the Section 5.2, Section 5.3, and Section 5.4 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 case, gfixr evaluates the performance of each candidate patch over the original input test suite; in the active 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 5.1: (*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 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 5.2.3). 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 over-matching 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 all 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.

5.6 Evaluation

In this section, we answer our third main research question, i.e., can we use fault localization to drive automatic repair of faults in grammars? We refine this question into three specific sub-questions:

RQ3a How effective is our proposed passive repair approach in fixing faults in grammars?

RQ3b How effective is our proposed active repair approach in fixing faults in grammars?

RQ3c Does the active repair approach induce better fixes than the passive repair approach?

5.6.1 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. Since the current version of gfixr does not support new token creation we discarded submissions with known lexical issues (e.g., wrong regular expressions for strings). The first ten grammars (#1 to #10, see Table 5.1) 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 2.3.1, 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 5.1.8.

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 metrics used to evaluate the accuracy of the learned grammars in the Arvada system [80]. 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 5.2 and 5.3 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 overfitting 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, that indicates that the repaired grammar is too permissive and according to our set up above, we get low recall.

5.6.2 Passive Repair Results (RQ3a)

Table 5.1 summarizes the results of our passive repair approach for the student grammars. $\mathcal{L}$ is the target language, with *test* the running example (see Figure 5.1), 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 5.1 and 5.2. *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 can-

[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 5.1: 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	*test*	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

didate, 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 5.1 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 [7, 162]. Moreover, it also indicates that the combined patches are sufficiently expressive. In the failing cases, however, the localization ranked the faulty location too low, and the repair kept trying to fix correct rules (see Section 5.7 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 5.2 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 input grammar G, 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 overgeneralize beyond the target language.

The corresponding recall, precision, and F1 score values for the repaired variants are shown in columns R_p, P_p, and $F1_p$, 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. Over-

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

Table 5.2: Summary of results showing accuracy of the passive repair approach and the number of applied patches for each repaired grammar.

			accuracy						symbol edit				list.	
#	$\mathcal{L}$	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	*test*	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						

all, 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 5.6.3).

Applied Patches. The right-most columns of Table 5.2 give insight on the interaction of the grammar transformations discussed in Sections 5.2, 5.3 and 5.4 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 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. More specifically, deletion is applied 22 times, insertion 27 times, substitution 25 times, and two listification patches are applied 7 and 17 times respectively.

RQ3a: 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.

5.6.3 Active Repair Results (RQ3b)

Table 5.3 summarizes the repair results using the active repair approach described in Section 5.1.8. Here, TS_{init} comprises the CDRC test suite $TS_{\mathcal{L}}$ that is generated from the target language (that also serves as oracle $\mathcal{O}$), 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 5.3 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 5.3 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 here, an increase from just two in the passive repair experiments. Grammar #2 in particular exhibits

Table 5.3: 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	*test*	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

the biggest jump from one bug revealed in the passive case to seven here. 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.

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 5.3 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 5.7 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 [132]. In fact, we generate test suites using the same generic cover algorithm (see Section 2.3.3) they used 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. We show how it compares to the passive repair approach in the next section.

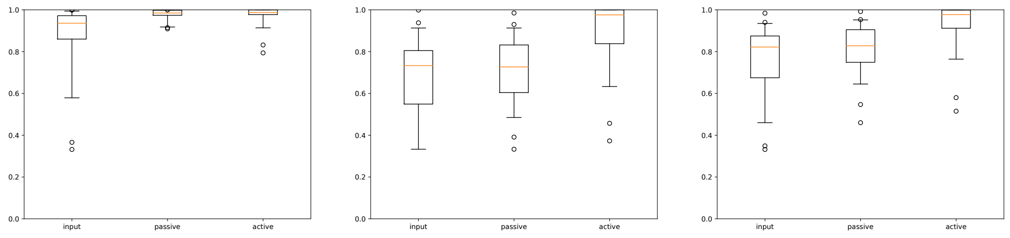
Applied Patches. Table 5.4 gives the details on the The patch types are labeled as before but we now also include language tightening patches here: $\mathfrak{S}$ refers to non-terminal and token splitting patches, $\mathfrak{D}$ rule deletion patch, $\mathfrak{P}$ push-down list elements patches and $\mathfrak{E}$ to immediate recursion elimination patches.

Like in the passive case, most of the patch types are used widely, but symbol transposition remains unused. More specifically, deletion is applied

Table 5.4: 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	*test*	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

16 times, insertions 8 times, 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



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

Figure 5.2: 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.

3 and 10 times respectively.

RQ3b: 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.

5.6.4 Passive Repair vs Active Repair (RQ3c)

In this section, we compare the passive and active repair approaches "like-for-like" for all 33 grammars. Figure 5.2 summarizes the results of this comparison through a series of boxplots. We see that active repair produces repairs with slightly better recall, but 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 to the oracle into the repair loop, prevents some over-generalization to some degree even when using weaker test suites like CDRC as repair specifications.

Despite high runtime costs and naturally more memory usage, we can conclude that the active repair approach produces patches of higher quality and lower F1 scores of the passive repair approach show that over-generalization is prevalent when repairing against a fixed test suite.

RQ3c: The active repair approach produces patches that generalize better than the passive repair approach.

5.7 Limitations

In this section, we discuss some limitations that affect the efficacy of our proposed repair approaches. 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 approaches in gfixr.

5.7.1 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 5.1, where *startVarIDs*- and *variableIDs*-rules are defined as follows:

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

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 5.1.

5.7.2 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, we got along fine with enabling `-non_lr`, in some cases LR(1) parsing restrictions did not allow the search to stop with the required high quality patches. In particular, in grammar #8 where the target language has the same expression structure as our example grammar in Figure 5.1. The *simple*- and *simple_list*-rules are written as

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

The oracle rejects some 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

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

The expectation was that, in the next iteration, gfixr would apply rule deletion patches to the second, third, fourth, fifth and sixth rules of *simple_list*, as they get 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

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

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.

5.7.3 Loop Restart

In active repair, the basic idea of a repair loop restart is to stop the search after some iteration i when some specified condition is met, 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 $\mathcal{O}$ on the initial test suite (recall from Table 5.3 that this includes a user provided seeds and test suite generated from the faulty input grammar G), but rejects a test case $w \in L(\mathcal{O})$ that was added to TS in some later iteration. Patches from grammars #15, #18, and #19 were prompted to restart the search.

5.8 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, 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 tested our implementation and scripts, and by using well-established tools for item-level spectra collection and test suite generation.

5.9 Conclusion

This chapter introduced and described our generic grammar repair framework with all its necessary ingredients. We presented two variants of our general repair approach. In the passive repair variant, we repair an input grammar against a fixed test suite specification, while the active repair variant exploits a boolean oracle that is capable of answering membership queries on the test suite enrichment adding tests derived from each generated repair candidate.

These ideas are then implemented in a prototype tool gfixr that takes as input a grammar G in CUP-format and a test suite $TS_{\mathcal{L}}$, and automatically constructs a "similar" grammar G' that accepts all positive and rejects all negative tests for the intended target language.

We have successfully used gfixr to fix 33 grammars students submitted as homeworks in compiler engineering course. We showed how both repair configurations produce grammars improved in quality over the input grammars. We have also demonstrated that the active repair setting produces more grammars that capture the original intent of the grammar better than the passive repair.

Chapter 6

Related Work

We are not aware of any work that directly shares our high-level goal of reporting the exact locations of bugs in context-free grammars and automatically fixing those bugs. However, there are related work from several areas that inspired our approach.

6.1 Spectrum-Based Fault Localization

Dating back to the late 1970s, the field of *software fault localization* has been a popular area of research and continues to be an important track in today's major software engineering conferences and journals, with new techniques and the application of existing ones to other areas such as *automated program repair* still being proposed. Many techniques have been applied to the problem of fault localization, e.g., program slicing, machine learning, model-based techniques, program spectrum-based techniques, etc. Wong *et al.* [158] gives a very comprehensive overview of the entire field and highlights advantages and disadvantages, limitations, issues, and concerns for the different techniques. This thesis focuses exclusively on spectrum-based techniques [32, 158].

Spectrum-based fault localization (SFL) employs data gathered during the execution of test cases to identify faulty program fragments. The origins of SFL methods can be traced back to work by Collofello and Cousins [28] who analysed executed paths to identify faulty sites. As a subfield of fault localization, SFL attracted a great deal of attention, with different kinds of techniques being proposed. However, this work borrows heavily from metric-based SFL techniques that formulate different ranking metrics. These ranking metrics utilize coverage data (or program spectra, see Section 2.4 for more details) to compute a suspiciousness score for each program entity. Higher scores indicate higher bug likelihood.

There have been more than 40 different ranking metrics proposed for SFL [32, 57, 112, 158]. Tarantula [69, 70] is considered to be one of the first metrics to be used for SFL and it has been extensively used as a baseline to benchmark new techniques [109, 157]. Naish *et al.* [109] propose two metrics O and O^P optimized for single-fault programs and empirically evaluate their performance with respect to other metrics using the Siemens and Space benchmarks. While we do not use these two specific metrics in this work, the authors highlight the very important insight that most of the metrics yield identical results. Using equivalence relations, Debroy and Wong [34] theoretically show that some metrics do indeed produce similar rankings. In this work, we use four of the widely used metrics that have performed well in other experimental evaluations [6, 89]: Tarantula [69, 70], Ochiai, which was first applied to problems in Botany [115], Jaccard [26], which was used in the information retrieval domain, and DStar [157].

It is worth noting that Tarantula takes into account all four basic counts (see Section 2.4.1), while the other three metrics used in this work omit program elements *not* executed in passing tests. Experimental results [6, 157] show that Ochiai, Jaccard and DStar outperform Tarantula. Our evaluation results also confirms that Tarantula performs the worst of the four metrics and that Ochiai, Jaccard and DStar cannot experimentally be told apart in terms of performance under different scenarios.

There exists work that departs from proposing new ranking metrics, but exploits spectral information in various controlled ways in order to improve SFL. Zhang *et al.* [172, 173] present methods that incorporate both dimensions of spectral analysis (i.e., program elements and the test suite) to boost SFL. They specifically use the PageRank algorithm [117] to recompute execution traces, and traditional ranking metrics can then be used to assign rankings. We adapted these ideas and used the PageRank algorithm for our fault localization task. We however did not obtain significant gains and some metrics' (esp. DStar) performance suffered. Santelices *et al.* [137] combine different levels of granularity (statements, control flow, and data flow) to improve SFL. These ideas do not directly translate to the domain of grammars, although we experimentally showed that our item-level fault localization gives more precise results than the base rule-level fault localization. Gopinath *et al.* [52] propose using SAT technology to improve SFL by annotating program elements with specifications and returns elements that violate some constraints. Again, this does not translate into our domain, and our work does not apply specification-based methods in tandem with SFL.

Our experiments in Sections 3.6 and 4.4 evaluate the efficacy of our SFL approach using grammars with seeded faults. Fault seeding has been used extensively in the literature [5, 109, 156] although we use different, domain-specific mutation operators. We further successfully evaluate our solution over grammars with multiple faults in Sections 3.6 and 5.6 even though it is well known that SFL techniques are based on a single-fault assumption and that their accuracy deteriorates for programs with multiple faults [7, 162].

One of the open questions in SFL is tie resolution. Xu *et al.* [161] present an evaluation of three heuristics for breaking ties viz., statement order-, confidence- and data dependency- based strategies. Our item-level fault localization uses a simple strategy that prefers the right-most item of a rule which can be seen as a domain-specific version of a statement-order based tie breaking strategy. Our attempts to resolve ties by further exploiting the hierachical structure of the grammar did not produce favourable results. Finally, Steimann *et al.* [145] study the threats of validity for SFL. Our work inherits most of the threats of validity outlined in their study.

6.2 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. This area of research is still relatively young, and seminal work can be traced back to the late 1990s [148] and early 2000s. APR comes in different flavours, e.g., generate-and-validate, semantics- and data-driven techniques. We identify three studies that give a comprehensive overview of the field [48, 49, 108].

We discuss generate-and-validate (see 6.2.1) and semantic-driven (see Section 6.2.2) approaches in this thesis because they are the most studied techniques. These techniques have been shown to suffer from overfitting and generated patches may not generalize to different, previously unseen tests [55, 92, 93, 127]. Some studies propose ways to mitigate overfitting and generalization issues [91, 107, 159, 160], but this work is out of our scope.

6.2.1 Generate-and-Validate Techniques

Like our repair task, many approaches are based on generating candidate patches using different search strategies such as genetic programming [12, 13, 14, 47, 94, 154, 155], random search [67, 125, 126], or bug templates [62, 75, 79, 96, 97, 98, 103, 135] and validating each candidate patch over a test suite.

The most notable approach in this category is GenProg [47, 94, 154, 155] that searches for candidate patches using genetic programming operations. It specifically modifies the input program by applying crossover and mutation (deleting, inserting, or transposing statements) operations on all statements flagged as suspicious. Application of these modifications is based on the *redundancy assumption* [93] that follows the intuition that the fix already exists elsewhere in the program. GenProg defines its own simple fault localization strategy that assigns each statement executed exclusively in failing test cases a value of one, statements executed only in passing tests a value of zero and a fractional value for statements executed in both failing and passing test cases. Each generated candidate patch is validated by compiling and running it over the same set of test cases and a fitness score, defined as a weighted sum of passing and failing tests, is computed for each candidate. Candidates with higher fitness scores are retained for further modifications, while those with a lower fitness score are discarded. We use similar ideas in our repair approach, specifically, our patch selection strategy is similar to that used by GenProg. We can easily re-configure our repair tool gfixr to use GenProg's fault localization strategy as well.

More recent work adapts the core ideas of GenProg and applies them to problems in other domains and programming languages. JGenProg2 [101, 102] is a Java re-implementation of GenProg. The original tool targets C code. Gissurarson *et al.* [50] describe PropR, a tool that utilizes GenProg's core algorithm to suggest fixes for buggy Haskell programs. Ahmad *et al.* [8] propose a genetic programming based tool, CirFix that repairs defects in hardware designs implemented in Verilog.

RSRepair [126] is built on top of GenProg and uses the same mutation operators, but substitutes the genetic programming search with a random search. The authors show that it outperforms GenProg both in the quality of repair and speed of search. It also employs the same fault localization strategy but differs in the patch selection for mutation. Each iteration only considers one candidate patch and instead is discarded immediately if it does not improve over the parent. RSRepair boasts substantial speed-ups because the algorithm it implements also does away with fitness score computation for each candidate patch, but invokes a test prioritization based strategy for its validation step. However, the optimizations employed by RSRepair make it ill-suited for repairs that require multiple iterations.

Moving on from the GenProg family of approaches, some studies apply *mutation testing* ideas to program repair [33, 49, 102]. Debroy and Wong [33] propose an approach that heavily relies on the spectrum-based fault localization metric Tarantula to identify repair sites. They apply two kinds of mutation operators at each site: replacement of an operator (arithmetic, relational, logical, assignments and pre- (resp. post-) increment (resp. decrement) with one from the same class, and inversion of conditions in if- and while-statements. Martinez and Monperrus [102] present a modular Java re-implementation of the ideas presented by Debroy and Wong [33]. Ghanbari *et al.* [49] present PRaPR that applies mutation operators on the JVM bytecode at the suspicious locations computed using the Ochiai metric. Our grammar transformations can be seen as domain-specific mutations and those used in these studies are not easily transferrable to our work.

Finally, other generate-and-validate methods use code templates. Kim *et al.* [75] present another GenProg derivative called PAR that replaces the mutations with carefully extracted code patterns. The authors manually inspected a little over 60K human code patches, from which they derive ten fix templates. They show that the PAR system outperforms GenProg in the quality of patches and also finds more fixes. Other notable studies that mine fix templates for program repair include [98] (which mines code patterns from StackOverflow), and work by Koyuncu *et al.* [79] which defines a reusable template mining library that can be used by other tools.

6.2.2 Semantics-Driven Techniques

In semantics-driven approaches, the faulty code fragments (dentified using spectrum-based fault localization approaches) are executed symbolically while the non-faulty fragments are executed using concrete values using symbolic execution. One such approach that is implemented by the tool SemFix [113, 133], first, uses the Tarantula ranking metric to identify faulty program locations. Second, for each location, the symbolic execution engine KLEE [23] is used to generate a repair constraint (a tuple of path conditions and symbolic output). Lastly, a program synthesis routine is invoked to solve the repair constraint. Mechtaev *et al.* [107] address the scalability issues in semantics-driven approaches by using a lightweight representation of a repair constraint called an *angelic forest* that is well suited for fixes in multiple locations. These ideas are implemented in a tool called Angelix which targets C code. JFix [90] is an extension of Angelix for Java programs; it uses Symbolic PathFinder [121], a widely used symbolic execution engine for Java. Ke *et al.* [72] describe a repair approach implemented in a SearchRepair tool that maintains a database of human-written patches encoded as SMT constraints. Each faulty statement is executed symbolically to extract repair constraints. The repair constraints are then resolved through a semantic search in that database.

Semantics-driven repair approaches share the use of spectrum-based fault localization methods with our work, but because grammars do not have an equivalent executable semantic model, their underlying ideas are not obviously applicable to our domain.

6.3 Grammar-Based Test Suite Generation

Since the fault localization and repair tasks described in this thesis rely on test suites, we need to ensure that these test suites sufficiently cover the syntactic structure of the target language $\mathcal{L}$. 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, even if is not accessible to the grammar developers (e.g., the instructor's grammar that is hidden from the students), or if it is the wrong formalism (e.g., grammar migration) and automatically generate detailed test suites.

Systematic Positive Test Suite Construction. Purdom [124] in 1972 first proposed an algorithm to test grammars and grammar-aware tools using automatically generated test suites from a context-free grammar. The algorithm aims to generate a minimal set of test cases that exercise all the rules of the grammar. Malloy and Power [100] provide a modernised, structured and modular re-implementation of Purdom's algorithm. The algorithm can be summarized in three steps. The first two steps collect static information about the grammar, in particular, the minimal yield for each symbol and the shortest derivation for each grammar rule. In the final step, the algorithm starts derivations from the start rule and expands unexercised rules, one at a time, while also marking the corresponding non-terminal symbols as used. We however do not use Purdom's algorithm in our work, based on the conjecture that the generated test cases are not specially suitable for fault localization because of the complexity of each individual test cases.

We evaluate both solutions proposed in this work over a series of test suites that are systematically constructed using the generic cover algorithm used in approaches by Fischer *et al.* [43] and Havrikov and Zeller [56]. The algorithm guarantees test suites with the required syntactic coverage and variance. These generated test suites also satisfy several grammar coverage criteria, e.g., the most straightforward criterion and perhaps the simplest intuitively is *rule* coverage that ensures, just like Purdom's algorithm, that every rule of the grammar is exercised. The main difference is that the maximum coverage of the grammar achievement is concentrated on the individual test cases in Purdom's algorithm, i.e., the algorithm "stuffs" as many rules into a test as possible. Lämmel [82] defines a generalization of rule coverage called *context dependent rule coverage (CDRC)* which applies each A-rule to all occurrences of the non-terminal A in the grammar. CDRC produces richer test suites and has become a baseline coverage criterion in the field.

There exist variations of CDRC that induce longer and deeper derivations. We use k-step derivations (also known as k-path coverage [56]). The basic idea behind k-step derivations is, for every embedding $S \Rightarrow^* \alpha X \omega$, to find a derivation for a pair $(X, Y) \in V \times V$ in at most k steps. van Heerden *et al.* [150] evaluate *derivable pair coverage* that the authors describe as a fix point version of k-step. Another CDRC variant that explores deeper derivations is called, bfs_k [150] that simultaneously covers all occurrences of a rule A.

Some work take an analytical view of grammar-based test suite construction: this test suite generation mechanism works from the automata representation of the grammar and generates tests through exploring all state transitions of the automata [131, 170]. More recently Rossouw and Fischer [131, 132] describe two LR graph traversal based algorithms that initially translate $LR(0)$ automata produced by HYacc [142], a variant of Yacc [68] that accepts all $LR(1)$ grammars, into LR graphs. The authors define explicit conditions on path selection that lead to positive tests. Their approach can be thought to simulate LR parsing. Evaluation over two large real world grammars shows their approach is scalable. In our work, (in particular in the active repair approach), the test suite generation is used as a black-box algorithm. We achieved good results with the generic cover algorithm and we leave a comparison to LR-based algorithms as future work.

Systematic Negative Test Suite Construction. In the literature, the generation of positive test suites that contain syntactically valid tests has enjoyed more attention than negative test suite generation. Zelenov and Zelenova [170] described the first algorithm to generate negative tests. Rossouw and Fischer [131, 132] also give a LR-based algorithm for negative test suite generation that uses local (edge level edits) and global (stack operations) mutations. However, these algorithms are complex. Raselimo *et al.* [130] propose simple word and rule mutation algorithms that produce tests that contain single and well-defined errors. The basic idea behind negative test suite construction via word mutation is to introduce a *poisoned pair* (defined as a pair of tokens that can never occur next to each other in every derivation from the start rule) to a positive test. Rule mutation systematically modifies the rules of the grammar so that any derivation from the start rule that applies the mutated rule results in an invalid word. We evaluate our approach using these word and rule mutation test suites, and rely on the active repair algorithm to prevent overgeneralizations.

Random Test Suite Construction. Beyond systematic test suite construction methods, other approaches are based on random rule selection and application in derivations until a complete test case is generated. Such approaches use various control parameters (e.g., derivation depth, length, rule probabilities, balance, dependence and construction restrictions and many others) to avoid combinatorial explosion and generate test suites with certain characteristics [21, 58, 59, 61, 83, 104, 105, 122]. van Heerden *et al.* [150] experimentally show that purely random test suites outperform systematic test suites both in code coverage and triggering more crashes in hand-rolled one pass compilers written by students. We use random test suite construction to complement the systematic test suite construction. Random sentence generation approaches have enabled many grammar-based fuzzing tools that have been used to discover many bugs and security vulnerabilities in software. Some notable work apply these random methods in tandem with *differential testing* [106] to effectively address the test oracle problem (i.e., the difficulty to confirm if the observed output agrees with the expected outcome). CSmith [163] employs such techniques and generates well-formed programs by construction, and has been used to test C compilers. The RAGS system [141] has successfully found many bugs in many SQL systems. Java runtime environments have been subjected to similar techniques [164]. Many other grammar-based fuzzing tools exist and target different use cases. LangFuzz [164] and IFuzzer [151] randomly generate sentences from a generic grammar and often exploit a given corpus to extract seed code fragments. Nautilus [15] uses feedback from the system under test to further guide sentence generation. Some grammar-based fuzzers (e.g., Skyfire [152] and Superion [153]) are built specifically to be used as frontends to popular coverage-guided grey-box and grammar-blind fuzzers such as AFL [166] and libFuzzer [63].

6.4 Grammar Engineering

Grammar engineering denotes the systematic application of software engineering techniques to the development of CFGs. In that case, our work can be seen as grammar engineering. Below we discuss further grammar engineering approaches.

Grammar Transformations. Lämmel and Zaytsev [81, 167] have defined general grammar transformations and used them for grammar construction, refactoring, and adaptation [85, 168], including the extraction and comparison of several complete grammars from different language specifications [84, 86]. Such transformations could also be used for grammar repair, although our experiments have shown that our small set of transformations is already sufficient. Jain et. al [66] 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.

Grammar Convergence. Lämmel and Zaytsev [85] propose *grammar convergence*; a transformation-based method that systematically modifies any pair of input grammars G' and G'' until they are structurally equivalent. Grammar convergence builds on two carefully designed grammar engineering disciplines, first, *grammar comparison*: which comprises scanning both input grammars and flagging nominal and structural differences between them, and second, *grammar transformation*: which builds on the previous step, and defines and applies meta transformation function f to G' so that it structurally approximates G''. The result $f(G')$ is said to be *f-equal* to G''. In hindsight, the problem this thesis addresses seems to be trivial – after all, we could simply apply grammar convergence methods to automate grammar debugging. However, resorting to such methods presents a myriad of challenges that invalidate their suitability. Firstly, and perhaps most importantly, grammar convergence does not generalize; it requires white-box access to both the grammar under test and the reference grammar (and they should presumably be in the same format). This makes it almost incompatible with some application scenarios, for example, when we have a test suite TS for an unknown target language (which is possibly described by an unavailable grammar) and an input grammar G that fails at least one test case $t \in TS$ and the goal is repair G to be consistent with TS. This is typically the case in teaching, where students develop grammars from a textual description of a target language and a few given test seeds. Secondly, the grammar comparator used would judiciously flag input grammars as "different" even in simpler cases where different naming convention mechanisms are used. For example, one of the two Pascal grammars that claim to define the same specification (which is rebutted [99, 128]) uses the snake case naming convention, while the other chooses Pascal casing. And finally, the cost of grammar convergence is high, as it requires human expertise to select from numerous expressive grammar transformations.

Grammar Smells. Faults in grammars can manifest themselves in non-terminal symbols that are non-productive or unexpectedly nullable, or in ambiguities that could be resolved unexpectedly by a (deterministic) parser. While non-productivity and nullability are easily checkable, ambiguity is undecidable in general [24], although several practical approximations have been developed [18, 22, 138, 139, 140]. Basten's approach [18] identifies rules that are provably not involved in an ambiguity and so helps with localization. LR parser generators typically report any shift/reduce and reduce/reduce conflicts that they encounter; Isradisaikul and Myers [65] produce "unifying counterexamples" for such situations that can help users to debug their grammars.

However, none of these approaches can really be seen as fault localization, because the situations that they detect are *grammar smells* rather than necessarily faults. Consider for example the traditional "dangling else" problem [10]. Most LR parsers resolve the ambiguity indicated through shift/reduce conflict by shifting, and so accept the intended language. Stijlaart and Zaytsev [147] provide a comprehensive classification of different grammar smells.

Grammar Equivalence. Proving the equivalence of the grammar under test to a given "golden" grammar can be seen as an alternative to fault localization, similar to the way proving a program correct is an alternative to testing. CFG equivalence is of course well-known to be undecidable in general, but decision algorithms have been developed for several relevant subclasses, e.g., simple [19, 78], LL(k) [116], or LL-regular grammars [114]. Madhavan et al. [99] describe a system that implements several of these algorithms and can produce counter-examples when it finds that the grammars are not equivalent. Fischer et al. [43] use systematic test case generation and parsing to identify which non-terminals accept the most similar languages, which can be seen as an approximate, fine-grained equivalence check. Such approaches could be used to validate repairs that are found by our repair approach.

6.5 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, in this thesis, 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 [37] and modification of EBNF operators [30] in a single rule, global mutations such as merging and splitting of non-terminal symbols [123], mutated rule duplication [37], or different rule generalizations [123], 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 over-generalization [30]. Scoring functions are typically based on some version of balanced accuracy, sometimes taking the length of the longest recognized fragment into account [88]. 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 $\mathcal{O}$ to answer membership queries on the generated sentences, are intrinsically incorporated in our main repair loop, the GA-based algorithm by Crepinsek *et al.* [30] leave sentence generation from candidates only after the plausible candidate that parses all positive examples is returned to determine the need for further introduction of negative examples. The eg-GRIDS system by Petasis *et al.* [123] 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) algorithm is rather employed that ensures "compact" learned grammars with respect to encoded training examples.

Di Penta and Taneja [38] and Di Penta *et al.* [37] 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 work [129] 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., [143]) and has been widely addressed, both in theory and in practice (see [31, 95, 136, 146] 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* [51], where observations are presented in sequence (and approaches are often order-sensitive, e.g., [77]) and Angluin's *query model* [11], 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 [41, 136].

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 [64]. We focus on heuristic approaches here.

Several systems such as Synapse [110, 111] or Gramin [134] 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 [17] 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 [143]. 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.* [80] introduce and evaluate a non-deterministic grammar learning tool, Arvada, that takes as input (like Glade) training examples $\mathcal{S}$ and an oracle $\mathcal{O}$ 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 [20]. The replication study disputes some of the claims of the original paper such as "overly optimistic" F1 scores and raises scalability concerns of Glade.

6.6 Error Recovery and Correction

Syntax error recovery and correction algorithms are invoked when the parser detects syntax violations in the input, and try to enable parsing to continue by either manipulating the parse stack and/or modifying input tokens. There are several error recovery algorithms, and the discussions in these studies [25, 35, 39] give a good overview of the field.

Note that error recovery algorithms address a different problem than our work. Error recovery algorithms assume that the input is correct with respect to the underlying correct grammar, while our work assumes the complete opposite: we use the input test suite as specification to localize faults in the grammar and automatically modify the grammar to fix these faults.

One of the most common error recovery algorithms that is easily implementable with almost any grammar is "panic mode" recovery [60]. The basic idea of error recovery by the panic mode algorithm is to skip input tokens until a *synchronization* token (e.g., a semicolon in Java or C) is reached or inserted to enable parsing to continue. Our symbol deletion transformation (see Definition 5.2.1 in Section 5.2) takes inspiration from panic mode error recovery: starting at the designated position (within a rule), 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.

Another class of error recovery algorithms modify the underlying grammars themselves. The grammars are augmented by introducing *error productions* which allow appropriate handling and correction of syntax errors [9, 45, 54]. However, practical adoption of these techniques can only be found in the YACC system [68], where a special "error" token is used to allow the parsing to continue when an error is detected. The grammar augmented with these error productions can be ambiguous, just like our grammar transformations, which may introduce some conflicts.

Starting with work by Fischer *et al.* [44], there have been approaches that manipulate the states left on the LR parse stack at the time of syntax error and find the minimum cost sequence of insertions and deletions on input symbols to find a configuration that would enable parsing to continue. Corchuelo *et al.* [29] incorporate parsing elements in their algorithm, they specifically introduce shift sequences in addition to insertion and deletion repair sequences. Diekmann and Tratt [39] recently presented $CPCT^+$, an improved version of the algorithm by Corchuelo *et al.* [29] that recovers from errors, returns multiple minimum cost repair sequences in less time compared to the approach by Corchuelo *et al.* [29], and their experiments show that it is less prone to the cascading error problem than the popular panic mode error recovery mechanism. Our LR spectra collection described in Section 3.4 also takes a parse stack as input and simply pulls the (partially) applied rules or items from states left on the stack at time of syntax error, hence we do not modify the contents of the parse stack. However, the error recovery algorithms described here can modify the contents of the stack (e.g, by shifting symbols) in order to find a configuration that enables parsing to continue normally.

Summary of Related Work

Our discussion of related work shows that our work is inspired by elements from a wide variety of well studied fields. We first draw related work from spectrum-based fault localization for general software systems, from which our fault localization approaches heavily borrow ideas from. The automatic repair approach proposed in this thesis shares a lot in common with many generate-and-validate program repair approaches: we use the same patch selection strategies and define a patch validation function to compute the fitness of each generated repair candidate. Our work can also be seen as grammar learning, however, our repair approach solves a much simpler problem because our input grammars are closer to the unknown target language while grammar learning infers grammars from scratch. We also evaluate our approaches over test suites that satisfy different coverage criteria, and finally, we end our discussion with syntax error recovery mechanisms because our illustration of the manual find-and-fix in Section 1.1, relies on these syntax messages and recovery attempts from parsers.

Chapter 7

Conclusions and Future Work

7.1 Conclusions

Context-free grammars (CFGs) are a widely used mechanism to concisely specify the structure of complex objects, e.g., JSON objects, XML files, and of course computer programs. In software engineering, CFGs are used in testing, specifically in test-suite generation and fuzzing. However, in many cases CFGs are not available, incomplete, or outdated.

Several approaches and techniques that regard grammars as proper software artefacts address some of the engineering challenges in grammar development. In this work, we also follow the view that grammars are proper software artefacts. We demonstrate that they can also contain bugs, like any other software. Testing, which remains the most widely used method for software quality assurance in general, is also applicable to grammars, but it does not give any further information about the location of bugs, and much less about how to automatically fix them. We therefore develop and evaluate new techniques to automatically localize and repair bugs in grammars.

Fault Localization. We described and evaluated the first method specifically aimed at finding faulty rules in a grammar. It uses spectrum-based fault localization techniques that have been used successfully to identify faulty program elements in software. Our key insight is that the same framework applies to grammars with minimal changes. We only need to replace the concept of "executed statements" by that of "used rules" and can keep the remaining established framework in place.

We describe two variants of our spectrum-based fault localization method that work at two different levels of granularity; a coarse-grained rule-level localization that we use as a baseline and a more fine-grained item-level localization that localizes faults more precisely at the level of individual symbols in a rule. We gave formal definitions for grammar spectra for both configurations: *rule spectra* summarize which of the grammar rules have been (partially) applied in an attempt to parse an input, and *item spectra* enable us to determine which positions within the applied rules have been successfully processed and which await further processing; hence we only consider symbols on the right side of these boundaries within flagged rules.

We showed for both fault localization approaches how grammar spectra can be collected for LL and LR parsers and described how popular parser generator tools such as ANTLR, JavaCC, and CUP can be extended in order to collect these grammar spectra. We also described a "flipped" approach where synthetic grammar spectra are constructed directly from test cases derived from a grammar and used these to localize differences between the grammar and the language accepted by the black box system.

Our evaluation showed that our method can identify grammar bugs with a high precision. In a large fault seeding experiment, it ranked the seeded faults within the top five rules in more than half of the cases and pinpointed them (i.e., uniquely ranked them as most suspicious) in 10%–30%. On average, it ranks the faulty rules within about 25% of all rules, and within less than 15% for a very large test suite containing both positive and negative test cases. Our method pinpoints far fewer of the seeded faults down to the exact symbol position, or even ranks them within the top five positions, due to the larger number of possible locations and corresponding larger ties (i.e., groups of equally suspicious locations). However, a simple tie-breaking strategy that prefers the right-most position amongst the rules in a tie proves remarkably effective: it typically ranks the seeded faults within the top five positions in about 30%–60% of the cases, and pinpoints them in about 15%–40% of the cases. On average, it ranks the seeded faults within about 10%–20% of all positions. The specialized symbol-level localization also significantly outperforms a simplistic extension of the rule-level localization, where all positions within a rule are given the same score. We also showed that fault localization techniques based on synthetic spectra work even better at identifying these seeded single faults. More specifically, the faulty rule is uniquely localized in at least 40% of the cases and in about 85% of the cases the prediction is within the top five rules.

We were also able to identify deviations and faults in real world and student grammars, which contain multiple, real faults. Finally, the flipped version of our fault localization method found four locations where a large production-quality SQLite grammar deviates from the language accepted by the black-box SQLite system.

Automatic Grammar Repair. We have described the first approach to automatically repair bugs in context-free grammars. This approach alternates over two key steps and gradually improves the grammar until it passes all tests in a given test suite: (*i*) We use fine-grained spectrum-based fault localization to identify suspicious items (i.e., specific positions in rules) as potential repair sites. (*ii*) We use small-scale transformations to patch the grammar and formulate with each transformation explicit pre- and postconditions that are necessary for it to improve the grammar. Both steps significantly reduce the number of potential repair patches to be applied.We further use a priority queue to keep improving the most promising candidate grammars. At the high level, our repair approach relies on two basic principles, the *competent programmer hypothesis* which assumes that the grammars to be repaired are approximating the target language already sufficiently well and *Occam's razor* which expresses itself in the fact that the repair uses the vocabulary and the structure of the original grammar, and minimizes the number of applied patches.

We developed and evaluated two variants of our general grammar repair approach: a *passive repair* approach, where we repair the grammar against a fixed test suite specification, and an *active repair* approach which leverages an input oracle $\mathcal{O}$ that answers membership queries for the test suite enrichment, where we judiciously generate (positive and negative) tests from the patch candidates, and use the oracle to obtain the expected outcome.

We have prototyped the repair approach in the highly configurable gfixr tool that uses the Ochiai spectrum-based fault localization technique to repair faulty CUP grammars. This can be easily extended to use other fault localization techniques, and the same underlying ideas are easily transferable to other parser generator tools.

We successfully used gfixr's passive and active repair approaches to repair 33 grammars that contain multiple, real faults. The passive repair approach found patches in all but four cases where it returned partially repaired variants after 150 iterations. We showed that even these partially repaired variants have improved in quality over their corresponding faulty input grammars. We also showed that passive repair produces grammars that generalize well to new unseen tests that were generated from oracle grammars (i.e., the fixed grammars improved recall score). However, minor improvements in the achieved F1 scores by these patches induced by passive repair showed that the approach sometimes produces patches that over-generalize beyond the target language.

We developed and evaluated the active repair approach to address this over-generalization. Active repair substantially improves over passive repair and produced high quality patches that capture the original (human) intent of the grammar. We showed that we achieve 100% F1 score in eight grammars, 100% recall in ten cases and 100% precision in thirteen grammars.

7.2 Future Work Directions

We discuss possible extensions that address some interesting open questions that arose during our experimental evaluation. These additional avenues for future work could make our approaches achieve even higher precision and generalize to more parsing technologies.

7.2.1 Stronger Oracles for Negative Tests

Currently, our handling of negative test may be too simplistic; we mark negative tests as passed if the input grammar reports the syntax violations and hence its observed output matches the expected boolean output. In order to obtain precise negative grammar spectra, negative tests should be considered passed if the input grammar returns the same error location as the oracle. This requires us to implement a better oracle that tells us what the location is. We could perhaps achieve this in two ways, (i) for the fault seeding experiments we can simply run the golden parser and extract the location from that error message, and (ii) a more general handling would be to modify our negative test suite construction algorithms (see Section 2.3.3) in order to generate test suites augmented with location information.

7.2.2 Multi-Rule Repair

The current version of gfixr iteratively generates repair candidates for each identified repair site and validates each candidate against a test suite. This automates the find and fix cycle we illustrated earlier in Chapter 1. This is analogous to the popular "one-bug-at-a-time" (OBA) debugging technique. However, in our experimental evaluation, we also observed that multiple faults can sometimes be independently repaired in a single step. This is easily observable in Tables 5.1 and 5.4 where the number of bugs revealed by the test suites equal the number of iterations gfixr took to find the repair.

In automated program repair, such multi-line repair approaches have been shown to improve scalability in tools like Angelix [107]. We plan to extend gfixr to repair multiple rules in a single iteration. The starting point would be to introduce a different patch selection strategy, and to analyse the spectra in more detail, using a "multiple-bugs-at-a-time" (MBA) strategy in order to achieve the required separation of faults.

7.2.3 Patch Priorities

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) [81, 167].

Some patch types are more general than others (e.g., list synthesis has a wider scope than right recursion introduction) and some work better in different contexts than others (e.g., non-terminal splitting patches only consider a local context while push down list elements can be thought of global patches). As future work, we plan to define some priority function over the different patch types because we observed in our experiments that, for example, list synthesis patches tend to over-generalize beyond the target language more often than the right recursion introduction patches. The priority function should give preference to right recursion introduction to list synthesis in cases like these.

7.2.4 Migration to Modern Compiler-Compiler Tools

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 [42].

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. Moving to a generalized parser generator is perhaps the most interesting future work direction because from a theoretical and practical point of view, it may be a difficult task to learn or migrate a grammar from one formalism to another. The discussion of parsing restrictions in Section 5.7 also motivates this move because coping with the restrictions that CUP places on grammars significantly limits the style of grammar transformations we could use.

7.3 Final Remarks

We have developed two automated approaches which identify faulty rules in context-free grammars and repair the faults fully automatically, with little to no human intervention. These techniques can have direct or indirect impact and influence in other related fields; they can reduce costs and improve the quality of new language development, specifically for domain-specific languages that lack a large community of contributors. The automated methods proposed in this thesis can also influence and improve research in the field of software testing, since many advanced methods (such as fuzzing) require CFGs. The work proposed here leads to a novel iterative generate-localize-repair approach for mining grammars that are practically useful for grammar-based testing and other applications in software engineering. This approach constructs an initial candidate CFG, then generates test suites from the candidate, evaluates them via the teacher and uses its feedback to localize and repair suspicious rules.

This research can also impact teaching of compiler engineering courses, through automated feedback and grading tools, since the fault localization approach proposed here can identify any differences in CFGs and propose repairs to students.

Appendix A

CDRC Test Suite

The test suite TS_{test} we use in the running example in Chapter 5 to illustrate different types of grammar transformations.

```
 1 program a begin boolean array a; relax end
 2 program a begin boolean a, a, a; relax end
 3 program a begin boolean a, a; relax end
 4 program a begin boolean a; boolean a; relax end
 5 program a begin boolean a; relax end
 6 program a begin a() end
 7 program a begin a(0) end
 8 program a begin a(0, 0) end
 9 program a begin a(0, 0, 0) end
10 program a begin a := array 0 end
11 program a begin a := 0 end
12 program a begin a[0] := 0 end
13 program a begin if 0 then leave end end
14 program a begin if 0 then relax else leave end end
15 program a begin if 0 then relax else relax end end
16 program a begin if 0 then relax elsif 0 then leave end end
17 program a begin if 0 then relax elsif 0 then relax elsif 0 then relax end end
18 program a begin if 0 then relax elsif 0 then relax end end
19 program a begin if 0 then relax end end
20 program a begin integer a; relax end
21 program a begin leave; a() end
22 program a begin leave; if 0 then relax end end
23 program a begin leave; leave; leave end
24 program a begin leave; leave end
25 program a begin leave; read a end
26 program a begin leave; while 0 do relax end end
27 program a begin leave; write "" end
28 program a begin leave end
29 program a begin read a[0] end
30 program a begin read a end
31 program a begin relax end
32 program a begin while 0 do leave end end
33 program a begin while 0 do relax end end
34 program a begin write(0) end
35 program a begin write - 0 end
36 program a begin write false end
37 program a begin write a() end
38 program a begin write a[0] end
39 program a begin write a end
40 program a begin write not(0) end
41 program a begin write not false end
42 program a begin write not a end
43 program a begin write not not 0 end
44 program a begin write not 0 end
```

```
45 program a begin write not true end
46 program a begin write 0 \# 0 end
47 program a begin write 0 \% 0 end
48 program a begin write 0 * (0) end
49 program a begin write 0 * false end
50 program a begin write 0 * a end
51 program a begin write 0 * not 0 end
52 program a begin write 0 * 0 * 0 end
53 program a begin write 0 * 0 end
54 program a begin write 0 * true end
55 program a begin write 0 + 0 + 0 end
56 program a begin write 0 + 0 end
57 program a begin write 0 - 0 end
58 program a begin write 0 / 0 end
59 program a begin write 0 < 0 end
60 program a begin write 0 <= 0 end
61 program a begin write 0 = 0 end
62 program a begin write 0 > 0 end
63 program a begin write 0 >= 0 end
64 program a begin write 0 and 0 end
65 program a begin write 0 end
66 program a begin write 0 or 0 end
67 program a begin write "" . 0 end
68 program a begin write "" . "" . "" end
69 program a begin write "" . "" end
70 program a begin write "" end
71 program a begin write true end
72 program a define a(boolean array a) begin relax end begin relax end
73 program a define a(boolean a) -> boolean a begin relax end begin relax end
74 program a define a(boolean a) -> integer a begin relax end begin relax end
75 program a define a(boolean a) begin relax end begin relax end
76 program a define a(boolean a) begin relax end define a(boolean a) begin relax end begin relax end
77 program a define a(boolean a, boolean a) begin relax end begin relax end
78 program a define a(boolean a, boolean a, boolean a) begin relax end begin relax end
79 program a define a(integer a) begin relax end begin relax end
```

List of References

[1] (1990). Ieee standard glossary of software engineering terminology. *IEEE Std 610.12-1990*, pp. 1–84.

[2] (2014). CUP 0.11b.
Available at: `http://www2.cs.tum.edu/projects/cup/`

[3] (2020). JavaCC 7.0.5.
Available at: `https://javacc.github.io/javacc/`

[4] (2021). Sqlite.
Available at: `https://sqlite.org`

[5] Abreu, R. (2009). *Spectrum-based Fault Localization in Embedded Software*. Ph.D. thesis, Delft University of Technology, Netherlands.
Available at: `http://resolver.tudelft.nl/uuid:78aa2510-acff-4acb-85ec-15852aa08e5c`

[6] Abreu, R., Zoeteweij, P. and 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*, pp. 39–46. IEEE Computer Society.
Available at: `https://doi.org/10.1109/PRDC.2006.18`

[7] Abreu, R., Zoeteweij, P. and 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*, pp. 88–99. IEEE Computer Society.
Available at: `https://doi.org/10.1109/ASE.2009.25`

[8] Ahmad, H., Huang, Y. and Weimer, W. (2022). Cirfix: automatically repairing defects in hardware design code. In: Falsafi, B., Ferdman, M., Lu, S. and Wenisch, T.F. (eds.), *ASPLOS '22: 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, 28 February 2022 - 4 March 2022*, pp. 990–1003. ACM.
Available at: `https://doi.org/10.1145/3503222.3507763`

[9] Aho, A.V. and Peterson, T.G. (1972). A minimum distance error-correcting parser for context-free languages. *SIAM J. Comput.*, vol. 1, no. 4, pp. 305–312.
Available at: `https://doi.org/10.1137/0201022`

[10] Aho, A.V., Sethi, R. and Ullman, J.D. (1986). *Compilers: Principles, Techniques, and Tools*. Addison-Wesley series in computer science / World student series edition. Addison-Wesley. ISBN 0-201-10088-6.
Available at: `https://www.worldcat.org/oclc/12285707`

[11] Angluin, D. (1987). Queries and concept learning. *Mach. Learn.*, vol. 2, no. 4, pp. 319–342.
Available at: `https://doi.org/10.1007/BF00116828`

[12] Arcuri, A. (2009). *Automatic software generation and improvement through search based techniques*. Ph.D. thesis, University of Birmingham, UK.
Available at: `http://etheses.bham.ac.uk/400/`

[13] Arcuri, A. (2011). Evolutionary repair of faulty software. *Appl. Soft Comput.*, vol. 11, no. 4, pp. 3494–3514.
Available at: `https://doi.org/10.1016/j.asoc.2011.01.023`

[14] Arcuri, A. and 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*, pp. 162–168. IEEE.
Available at: `https://doi.org/10.1109/CEC.2008.4630793`

[15] Aschermann, C., Frassetto, T., Holz, T., Jauernig, P., Sadeghi, A. and 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.
Available at: `https://www.ndss-symposium.org/ndss-paper/nautilus-fishing-for-deep-bugs-with-grammars/`

[16] Barraball, C., Raselimo, M. and Fischer, B. (2020). An interactive feedback system for grammar development (tool paper). In: Lämmel, R., Tratt, L. and 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*, pp. 101–107. ACM.
Available at: `https://doi.org/10.1145/3426425.3426935`

[17] Bastani, O., Sharma, R., Aiken, A. and Liang, P. (2017). Synthesizing program input grammars. In: Cohen, A. and 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*, pp. 95–110. ACM.
Available at: `https://doi.org/10.1145/3062341.3062349`

[18] Basten, H.J.S. (2010). Tracking down the origins of ambiguity in context-free grammars. In: Cavalcanti, A., Déharbe, D., Gaudel, M. and Woodcock, J. (eds.), *Theoretical Aspects of Computing - ICTAC 2010, 7th International Colloquium, Natal, Rio Grande do Norte, Brazil, September 1-3, 2010. Proceedings*, vol. 6255 of *Lecture Notes in Computer Science*, pp. 76–90. Springer. ISBN 978-3-642-14807-1.
Available at: `https://doi.org/10.1007/978-3-642-14808-8_6`

[19] Bastien, C., Czyzowicz, J., Fraczak, W. and Rytter, W. (2006). Prime normal form and equivalence of simple grammars. *Theor. Comput. Sci.*, vol. 363, no. 2, pp. 124–134.
Available at: `https://doi.org/10.1016/j.tcs.2006.07.021`

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

[21] Bird, D.L. and Munoz, C.U. (1983). Automatic generation of random self-checking test cases. *IBM Systems Journal*, vol. 22, no. 3, pp. 229–245. ISSN 0018-8670.

[22] Brabrand, C., Giegerich, R. and Møller, A. (2010). Analyzing ambiguity of context-free grammars. *Sci. Comput. Program.*, vol. 75, no. 3, pp. 176–191.
Available at: `https://doi.org/10.1016/j.scico.2009.11.002`

[23] Cadar, C., Dunbar, D. and Engler, D.R. (2008). KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In: Draves, R. and van Renesse, R. (eds.), *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, pp. 209–224. USENIX Association.
Available at: `http://www.usenix.org/events/osdi08/tech/full_papers/cadar/cadar.pdf`

[24] Cantor, D.G. (1962). On the ambiguity problem of backus systems. *J. ACM*, vol. 9, no. 4, pp. 477–479.
Available at: `https://doi.org/10.1145/321138.321145`

[25] Cerecke, C. (2003). *Locally least-cost error repair in LR parsers*. Ph.D. thesis.
Available at: `http://dx.doi.org/10.26021/1600`

[26] Chen, M.Y., Kiciman, E., Fratkin, E., Fox, A. and 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*, pp. 595–604. IEEE Computer Society.
Available at: `https://doi.org/10.1109/DSN.2002.1029005`

[27] Clun, D., van Heerden, P., Filieri, A. and Visser, W. (2020). Improving symbolic automata learning with concolic execution. In: Wehrheim, H. and Cabot, J. (eds.), *Fundamental Approaches to Software Engineering - 23rd International Conference, FASE 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings*, vol. 12076 of *Lecture Notes in Computer Science*, pp. 3–26. Springer.
Available at: `https://doi.org/10.1007/978-3-030-45234-6_1`

[28] Collofello, J.S. and Cousins, L. (1987). Towards automatic software fault location through decision-to-decision path analysis. In: *In Proceedings of the AFIP 1987 National Computer Conference*, pp. 539 – 544.

[29] Corchuelo, R., Pérez, J.A., Cortés, A.R. and Toro, M. (2002). Repairing syntax errors in LR parsers. *ACM Trans. Program. Lang. Syst.*, vol. 24, no. 6, pp. 698–710.
Available at: `https://doi.org/10.1145/586088.586092`

[30] Crepinsek, M., Mernik, M., Bryant, B.R., Javed, F. and Sprague, A.P. (2005). Inferring context-free grammars for domain-specific languages. *Electron. Notes Theor. Comput. Sci.*, vol. 141, no. 4, pp. 99–116.
Available at: `https://doi.org/10.1016/j.entcs.2005.02.055`

[31] de la Higuera, C. (2010). *Grammatical Inference: Learning Automata and Grammars*. Cambridge University Press. ISBN 0521763169.

[32] de Souza, H.A., Chaim, M.L. and Kon, F. (2016). Spectrum-based software fault localization: A survey of techniques, advances, and challenges. *CoRR*, vol. abs/1607.04347. `1607.04347`.
Available at: `http://arxiv.org/abs/1607.04347`

[33] Debroy, V. and Wong, W.E. (2010). Using mutation to automatically suggest fixes for faulty programs. In: *Third International Conference on Software Testing, Verification and Validation, ICST 2010, Paris, France, April 7-9, 2010*, pp. 65–74. IEEE Computer Society.
Available at: `https://doi.org/10.1109/ICST.2010.66`

[34] Debroy, V. and Wong, W.E. (2011). On the equivalence of certain fault localization techniques. In: *Proceedings of the 2011 ACM Symposium on Applied Computing*, pp. 1457 – 1463. Association for Computing Machinery, New York, NY, USA. ISBN 9781450301138.
Available at: `https://doi.org/10.1145/1982185.1982498`

[35] Degano, P. and Priami, C. (1995). Comparison of syntactic error handling in LR parsers. *Softw. Pract. Exp.*, vol. 25, no. 6, pp. 657–679.
Available at: `https://doi.org/10.1002/spe.4380250606`

[36] DeMillo, R.A., Lipton, R.J. and Sayward, F.G. (1978). Hints on test data selection: Help for the practicing programmer. *Computer*, vol. 11, no. 4, pp. 34–41.
Available at: `https://doi.org/10.1109/C-M.1978.218136`

[37] Di Penta, M., Lombardi, P., Taneja, K. and Troiano, L. (2008). Search-based inference of dialect grammars. *Soft Comput.*, vol. 12, no. 1, pp. 51–66.
Available at: `https://doi.org/10.1007/s00500-007-0216-5`

[38] Di Penta, M. and 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*,

pp. 241–244. IEEE Computer Society.
Available at: https://doi.org/10.1109/CSMR.2005.52

[39] Diekmann, L. and Tratt, L. (2020). Don't panic! better, fewer, syntax errors for LR parsers. In: Hirschfeld, R. and Pape, T. (eds.), *34th European Conference on Object-Oriented Programming, ECOOP 2020, November 15-17, 2020, Berlin, Germany (Virtual Conference)*, vol. 166 of *LIPIcs*, pp. 6:1–6:32. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.
Available at: https://doi.org/10.4230/LIPIcs.ECOOP.2020.6

[40] Dijkstra, E.W. (1972). The humble programmer. *Commun. ACM*, vol. 15, no. 10, pp. 859–866.
Available at: https://doi.org/10.1145/355604.361591

[41] Drewes, F. and Högberg, J. (2003). Learning a regular tree language from a teacher. In: Ésik, Z. and Fülöp, Z. (eds.), *Developments in Language Theory, 7th International Conference, DLT 2003, Szeged, Hungary, July 7-11, 2003, Proceedings*, vol. 2710 of *Lecture Notes in Computer Science*, pp. 279–291. Springer.
Available at: https://doi.org/10.1007/3-540-45007-6_22

[42] Economopoulos, G., Klint, P. and Vinju, J.J. (2009). Faster scannerless GLR parsing. In: de Moor, O. and 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*, vol. 5501 of *Lecture Notes in Computer Science*, pp. 126–141. Springer.
Available at: https://doi.org/10.1007/978-3-642-00722-4_10

[43] Fischer, B., Lämmel, R. and Zaytsev, V. (2011). Comparison of context-free grammars based on parsing generated test data. In: Sloane, A.M. and Aßmann, U. (eds.), *Software Language Engineering - 4th International Conference, SLE 2011, Braga, Portugal, July 3-4, 2011, Revised Selected Papers*, vol. 6940 of *Lecture Notes in Computer Science*, pp. 324–343. Springer.
Available at: https://doi.org/10.1007/978-3-642-28830-2_18

[44] Fischer, C.N., Dion, B. and Mauney, J. (1979). A locally least-cost lr-error corrector.
Available at: http://digital.library.wisc.edu/1793/58168

[45] Fischer, C.N. and Mauney, J. (1980). On the role of error productions in syntactic error correction. *Comput. Lang.*, vol. 5, no. 3, pp. 131–139.
Available at: https://doi.org/10.1016/0096-0551(80)90006-5

[46] Ford, B. (2004). Parsing expression grammars: a recognition-based syntactic foundation. In: Jones, N.D. and Leroy, X. (eds.), *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004, Venice, Italy, January 14-16, 2004*, pp. 111–122. ACM.
Available at: https://doi.org/10.1145/964001.964011

[47] Forrest, S., Nguyen, T., Weimer, W. and 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*, pp. 947–954. ACM.
Available at: `https://doi.org/10.1145/1569901.1570031`

[48] Gazzola, L., Micucci, D. and Mariani, L. (2019). Automatic software repair: A survey. *IEEE Transactions on Software Engineering*, vol. 45, no. 1, pp. 34–67.

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

[50] Gissurarson, M.P., Applis, L., Panichella, A., van Deursen, A. and Sands, D. (2022). Propr: Property-based automatic program repair.

[51] Gold, E.M. (1967). Language identification in the limit. *Inf. Control.*, vol. 10, no. 5, pp. 447–474.
Available at: `https://doi.org/10.1016/S0019-9958(67)91165-5`

[52] Gopinath, D., Zaeem, R.N. and Khurshid, S. (2012). Improving the effectiveness of spectra-based fault localization using specifications. In: Goedicke, M., Menzies, T. and Saeki, M. (eds.), *IEEE/ACM International Conference on Automated Software Engineering, ASE'12, Essen, Germany, September 3-7, 2012*, pp. 40–49. ACM.
Available at: `https://doi.org/10.1145/2351676.2351683`

[53] Gopinath, R., Jensen, C. and Groce, A. (2014). Mutations: How close are they to real faults? In: *25th IEEE International Symposium on Software Reliability Engineering, ISSRE 2014, Naples, Italy, November 3-6, 2014*, pp. 189–200. IEEE Computer Society.
Available at: `https://doi.org/10.1109/ISSRE.2014.40`

[54] Graham, S.L., Haley, C.B. and Joy, W.N. (1979). Practical LR error recovery. In: Johnson, S.C. (ed.), *Proceedings of the 1979 SIGPLAN Symposium on Compiler Construction, Denver, Colorado, USA, August 6-10, 1979*, pp. 168–175. ACM.
Available at: `https://doi.org/10.1145/800229.806967`

[55] Gu, Z., Barr, E.T., Hamilton, D.J. and Su, Z. (2010). Has the bug really been fixed? In: Kramer, J., Bishop, J., Devanbu, P.T. and Uchitel, S. (eds.), *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE 2010, Cape Town, South Africa, 1-8 May 2010*, pp. 55–64. ACM.
Available at: `https://doi.org/10.1145/1806799.1806812`

[56] Havrikov, N. and 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*, pp. 189–199. IEEE.
Available at: `https://doi.org/10.1109/ASE.2019.00027`

[57] Heiden, S., Grunske, L., Kehrer, T., Keller, F., van Hoorn, A., Filieri, A. and Lo, D. (2019). An evaluation of pure spectrum-based fault localization techniques for large-scale software systems. *Softw. Pract. Exp.*, vol. 49, no. 8, pp. 1197–1224.
Available at: `https://doi.org/10.1002/spe.2703`

[58] Hodován, R., Kiss, Á. and Gyimóthy, T. (2018). Grammarinator: a grammar-based open source fuzzer. In: Prasetya, W., Vos, T.E.J. and Getir, S. (eds.), *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation, A-TEST@SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 05, 2018*, pp. 45–48. ACM.
Available at: `https://doi.org/10.1145/3278186.3278193`

[59] Hoffman, D., Ly-Gagnon, D., Strooper, P.A. and Wang, H. (2011). Grammar-based test generation with yougen. *Softw., Pract. Exper.*, vol. 41, no. 4, pp. 427–447.
Available at: `https://doi.org/10.1002/spe.1017`

[60] Holub, A.I. (1990). *Compiler design in C*. Prentice Hall. ISBN 978-0-13-155151-0.

[61] Homer, W. and Schooler, R. (1989). Independent testing of compiler phases using a test case generator. *Softw., Pract. Exper.*, vol. 19, no. 1, pp. 53–62.
Available at: `https://doi.org/10.1002/spe.4380190106`

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

[63] Infrastructure, L.C. (2016). libfuzzer.
Available at: `https://llvm.org/docs/LibFuzzer.html`

[64] Isberner, M. (2015). *Foundations of active automata learning: an algorithmic perspective*. Ph.D. thesis, Technical University Dortmund, Germany.
Available at: `http://hdl.handle.net/2003/34282`

[65] Isradisaikul, C. and Myers, A.C. (2015). Finding counterexamples from parsing conflicts. In: Grove, D. and Blackburn, S. (eds.), *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, pp. 555–564. ACM. ISBN 978-1-4503-3468-6.
Available at: `https://doi.org/10.1145/2737924.2737961`

[66] Jain, R., Aggarwal, S.K., Jalote, P. and Biswas, S. (2004). An interactive method for extracting grammar from programs. *Softw. Pract. Exp.*, vol. 34, no. 5, pp. 433–447.
Available at: `https://doi.org/10.1002/spe.568`

[67] Ji, T., Chen, L., Mao, X. and 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*, pp. 197–202. IEEE Computer Society.
Available at: `https://doi.org/10.1109/COMPSAC.2016.69`

[68] Johnson, S. (1975). Yacc.
Available at: `http://dinosaur.compilertools.net/yacc/`

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

[70] Jones, J.A., Harrold, M.J. and Stasko, J.T. (2002). Visualization of test information to assist fault localization. In: Tracz, W., Young, M. and Magee, J. (eds.), *Proceedings of the 24th International Conference on Software Engineering, ICSE 2002, 19-25 May 2002, Orlando, Florida, USA*, pp. 467–477. ACM.
Available at: `https://doi.org/10.1145/581339.581397`

[71] Just, R., Jalali, D., Inozemtseva, L., Ernst, M.D., Holmes, R. and Fraser, G. (2014). Are mutants a valid substitute for real faults in software testing? In: Cheung, S., Orso, A. and Storey, M.D. (eds.), *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*, pp. 654–665. ACM.
Available at: `https://doi.org/10.1145/2635868.2635929`

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

[73] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M. and Irwin, J. (1997). Aspect-oriented programming. In: Akşit, M. and Matsuoka, S. (eds.), *ECOOP'97 — Object-Oriented Programming*, pp. 220–242. Springer Berlin Heidelberg, Berlin, Heidelberg. ISBN 978-3-540-69127-3.

[74] Kiers, B. (2016). ANTLR4 grammar for SQLite 3.8.x.
Available at: `https://github.com/bkiers/sqlite-parser`

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

[76] Klint, P., Lämmel, R. and Verhoef, C. (2005). Toward an engineering discipline for grammarware. *ACM Trans. Softw. Eng. Methodol.*, vol. 14, no. 3, pp. 331–380.
Available at: `https://doi.org/10.1145/1072997.1073000`

[77] Knobe, B. and Knobe, K. (1976). A method for inferring context-free grammars. *Inf. Control.*, vol. 31, no. 2, pp. 129–146.
Available at: `https://doi.org/10.1016/S0019-9958(76)80003-4`

[78] Korenjak, A.J. and Hopcroft, J.E. (1966). Simple deterministic languages. In: *7th Annual Symposium on Switching and Automata Theory, Berkeley, California, USA, October 23-25, 1966*, pp. 36–46. IEEE Computer Society.
Available at: `https://doi.org/10.1109/SWAT.1966.22`

[79] Koyuncu, A., Liu, K., Bissyandé, T.F., Kim, D., Klein, J., Monperrus, M. and Traon, Y.L. (2020). Fixminer: Mining relevant fix patterns for automated program repair. *Empir. Softw. Eng.*, vol. 25, no. 3, pp. 1980–2024.
Available at: `https://doi.org/10.1007/s10664-019-09780-z`

[80] Kulkarni, N., Lemieux, C. and 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*, pp. 456–467. IEEE.
Available at: `https://doi.org/10.1109/ASE51524.2021.9678879`

[81] Lämmel, R. (2001). Grammar adaptation. In: Oliveira, J.N. and Zave, P. (eds.), *FME 2001: Formal Methods for Increasing Software Productivity, International Symposium of Formal Methods Europe, Berlin, Germany, March 12-16, 2001, Proceedings*, vol. 2021 of *Lecture Notes in Computer Science*, pp. 550–570. Springer.
Available at: `https://doi.org/10.1007/3-540-45251-6_32`

[82] Lämmel, R. (2001). 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*, vol. 2029 of *Lecture Notes in Computer Science*, pp. 201–216. Springer.
Available at: `https://doi.org/10.1007/3-540-45314-8_15`

[83] Lämmel, R. and Schulte, W. (2006). Controllable combinatorial coverage in grammar-based testing. In: Uyar, M.Ü., Duale, A.Y. and Fecko, M.A. (eds.), *Testing of Communicating Systems, 18th IFIP TC6/WG6.1 International Conference, TestCom 2006, New York, NY, USA, May 16-18, 2006, Proceedings*, vol. 3964 of *Lecture Notes in Computer Science*, pp. 19–38. Springer.
Available at: `https://doi.org/10.1007/11754008_2`

[84] Lämmel, R. and Verhoef, C. (2001). Semi-automatic grammar recovery. *Softw. Pract. Exp.*, vol. 31, no. 15, pp. 1395–1438.
Available at: `https://doi.org/10.1002/spe.423`

[85] Lämmel, R. and Zaytsev, V. (2009). An introduction to grammar convergence. In: Leuschel, M. and Wehrheim, H. (eds.), *Integrated Formal Methods, 7th International Conference, IFM 2009, Düsseldorf, Germany, February 16-19, 2009. Proceedings*, vol. 5423 of *Lecture Notes in Computer Science*, pp. 246–260. Springer.
Available at: `https://doi.org/10.1007/978-3-642-00255-7_17`

[86] Lämmel, R. and Zaytsev, V. (2009). Recovering grammar relationships for the java language specification. In: *Ninth IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2009, Edmonton, Alberta, Canada, September 20-21, 2009*, pp. 178–186. IEEE Computer Society.
Available at: `https://doi.org/10.1109/SCAM.2009.29`

[87] Lämmel, R. and Zaytsev, V. (2011). Recovering grammar relationships for the java language specification. *Softw. Qual. J.*, vol. 19, no. 2, pp. 333–378.
Available at: `https://doi.org/10.1007/s11219-010-9116-5`

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

[89] Le, T.B., Thung, F. and Lo, D. (2013). Theory and practice, do they match? A case with spectrum-based fault localization. In: *2013 IEEE International Conference on Software Maintenance, Eindhoven, The Netherlands, September 22-28, 2013*, pp. 380–383. IEEE Computer Society.
Available at: `https://doi.org/10.1109/ICSM.2013.52`

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

[91] Le, X.D., Chu, D., Lo, D., Le Goues, C. and Visser, W. (2017). S3: syntax- and semantic-guided repair synthesis via programming by examples. In: Bodden, E., Schäfer, W., van Deursen, A. and Zisman, A. (eds.), *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*, pp. 593–604. ACM.
Available at: `https://doi.org/10.1145/3106237.3106309`

[92] Le, X.D., Thung, F., Lo, D. and Le Goues, C. (2018). Overfitting in semantics-based automated program repair. In: Chaudron, M., Crnkovic, I., Chechik, M. and Harman, M. (eds.), *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, p. 163. ACM.
Available at: `https://doi.org/10.1145/3180155.3182536`

[93] Le Goues, C., Forrest, S. and Weimer, W. (2013). Current challenges in automatic software repair. *Softw. Qual. J.*, vol. 21, no. 3, pp. 421–443.
Available at: `https://doi.org/10.1007/s11219-013-9208-0`

[94] Le Goues, C., Nguyen, T., Forrest, S. and Weimer, W. (2012). Genprog: A generic method for automatic software repair. *IEEE Trans. Software Eng.*, vol. 38, no. 1, pp. 54–72.
Available at: `https://doi.org/10.1109/TSE.2011.104`

[95] Lee, L. (1996). Learning of context-free languages: A survey of the literature. Tech. Rep. Computer Science Group Technical Report TR-12-96, Harvard University.

[96] Liu, K., Koyuncu, A., Bissyandé, T.F., Kim, D., Klein, J. and Traon, Y.L. (2019). You cannot fix what you cannot find! an investigation of fault localization bias in benchmarking automated program repair systems. In: *12th IEEE Conference on Software Testing, Validation and Verification, ICST 2019, Xi'an, China, April 22-27, 2019*, pp. 102–113. IEEE.
Available at: `https://doi.org/10.1109/ICST.2019.00020`

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

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

[99] Madhavan, R., Mayer, M., Gulwani, S. and Kuncak, V. (2015). Automating grammar comparison. In: Aldrich, J. and 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*, pp. 183–200. ACM.
Available at: `https://doi.org/10.1145/2814270.2814304`

[100] Malloy, B.A. and Power, J.F. (2001). An interpretation of Purdom's algorithm for automatic generation of test cases. In: *1st ACIS Annual International Conference on Computer and Information Science*.
Available at: `http://eprints.maynoothuniversity.ie/6434/`

[101] Martinez, M., Durieux, T., Sommerard, R., Xuan, J. and Monperrus, M. (2018). Automatic repair of real bugs in java: A large-scale experiment on the defects4j dataset. *CoRR*, vol. abs/1811.02429. `1811.02429`.
Available at: `http://arxiv.org/abs/1811.02429`

[102] Martinez, M. and Monperrus, M. (2016). ASTOR: a program repair library for java (demo). In: Zeller, A. and Roychoudhury, A. (eds.), *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016*, pp. 441–444. ACM.
Available at: `https://doi.org/10.1145/2931037.2948705`

[103] Martinez, M. and Monperrus, M. (2018). Ultra-large repair search space with automatically mined templates: The cardumen mode of astor. In: Colanzi, T.E. and McMinn, P. (eds.), *Search-Based Software Engineering - 10th International Symposium, SSBSE 2018, Montpellier, France, September 8-9, 2018, Proceedings*, vol. 11036 of *Lecture Notes in Computer Science*, pp. 65–86. Springer.
Available at: `https://doi.org/10.1007/978-3-319-99241-9_3`

[104] Maurer, P.M. (1990). Generating test data with enhanced context-free grammars. *IEEE Software*, vol. 7, no. 4, pp. 50–55.
Available at: `https://doi.org/10.1109/52.56422`

[105] Maurer, P.M. (1992). The design and implementation of a grammar-based data generator. *Softw., Pract. Exper.*, vol. 22, no. 3, pp. 223–244.
Available at: `https://doi.org/10.1002/spe.4380220303`

[106] McKeeman, W.M. (1998). Differential testing for software. *Digit. Tech. J.*, vol. 10, no. 1, pp. 100–107.
Available at: `http://www.hpl.hp.com/hpjournal/dtj/vol10num1/vol10num1art9.pdf`

[107] Mechtaev, S., Yi, J. and Roychoudhury, A. (2016). Angelix: scalable multiline program patch synthesis via symbolic analysis. In: Dillon, L.K., Visser, W. and Williams, L.A. (eds.), *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, pp. 691–701. ACM.
Available at: `https://doi.org/10.1145/2884781.2884807`

[108] Monperrus, M. (2018). Automatic software repair: A bibliography. *ACM Comput. Surv.*, vol. 51, no. 1, pp. 17:1–17:24.
Available at: `https://doi.org/10.1145/3105906`

[109] Naish, L., Lee, H.J. and Ramamohanarao, K. (2011). A model for spectra-based software diagnosis. *ACM Trans. Softw. Eng. Methodol.*, vol. 20, no. 3, pp. 11:1–11:32.
Available at: `https://doi.org/10.1145/2000791.2000795`

[110] 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. and Tomita, E. (eds.), *Grammatical Inference: Algorithms and Applications, 8th International Colloquium, ICGI 2006, Tokyo, Japan, September 20-22, 2006, Proceedings*, vol. 4201 of *Lecture Notes in Computer Science*, pp. 72–83. Springer.
Available at: `https://doi.org/10.1007/11872436_7`

[111] Nakamura, K. and 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*, vol. 1891 of *Lecture Notes in Computer Science*, pp. 186–195. Springer.
Available at: `https://doi.org/10.1007/978-3-540-45257-7_15`

[112] Neelofar, Naish, L. and Ramamohanarao, K. (2018). Spectral-based fault localization using hyperbolic function. *Softw. Pract. Exp.*, vol. 48, no. 3, pp. 641–664.
Available at: `https://doi.org/10.1002/spe.2527`

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

[114] Nijholt, A. (1982). The equivalence problem for LL- and lr-regular grammars. *J. Comput. Syst. Sci.*, vol. 24, no. 2, pp. 149–161.
Available at: `https://doi.org/10.1016/0022-0000(82)90044-7`

[115] 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*, vol. 22, no. 9, pp. 526–530.

[116] Olshansky, T. and Pnueli, A. (1977). A direct algorithm for checking equivalence of ll*(k)* grammars. *Theor. Comput. Sci.*, vol. 4, no. 3, pp. 321–349.
Available at: `https://doi.org/10.1016/0304-3975(77)90016-0`

[117] Page, L., Brin, S., Motwani, R. and Winograd, T. (1999). The pagerank citation ranking: Bringing order to the web. Tech. Rep., Stanford InfoLab.

[118] Parr, T. (1990). Antlr.
Available at: `https://www.antlr.org/download/antlr-4.10.1-complete.jar`

[119] Parr, T. and Fisher, K. (2011). Ll(*): the foundation of the ANTLR parser generator. In: Hall, M.W. and Padua, D.A. (eds.), *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, pp. 425–436. ACM.
Available at: `https://doi.org/10.1145/1993498.1993548`

[120] Parr, T., Harwell, S. and Fisher, K. (2014). Adaptive ll(*) parsing: the power of dynamic analysis. In: Black, A.P. and Millstein, T.D. (eds.), *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014*, pp. 579–598. ACM.
Available at: `https://doi.org/10.1145/2660193.2660202`

[121] Pasareanu, C.S., Visser, W., Bushnell, D.H., Geldenhuys, J., Mehlitz, P.C. and Rungta, N. (2013). Symbolic pathfinder: integrating symbolic execution with model checking for java bytecode analysis. *Autom. Softw. Eng.*, vol. 20, no. 3, pp. 391–425.
Available at: `https://doi.org/10.1007/s10515-013-0122-2`

[122] Payne, A.J. (1978 January). A formalised technique for expressing compiler exercisers. *SIGPLAN Not.*, vol. 13, no. 1, pp. 59–69. ISSN 0362-1340.
Available at: `http://doi.acm.org/10.1145/953428.953435`

[123] Petasis, G., Paliouras, G., Spyropoulos, C.D. and Halatsis, C. (2004). eg-grids: Context-free grammatical inference from positive examples using genetic search. In: Paliouras, G. and Sakakibara, Y. (eds.), *Grammatical Inference: Algorithms and Applications, 7th International Colloquium, ICGI 2004, Athens, Greece, October 11-13, 2004, Proceedings*, vol. 3264 of *Lecture Notes in Computer Science*, pp. 223–234. Springer.
Available at: `https://doi.org/10.1007/978-3-540-30195-0_20`

[124] Purdom, P. (1972). A sentence generator for testing parsers. *BIT*, pp. 366–375.

[125] Qi, Y., Mao, X. and 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*, pp. 180–189. IEEE Computer Society.
Available at: `https://doi.org/10.1109/ICSM.2013.29`

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

[127] Qi, Z., Long, F., Achour, S. and Rinard, M.C. (2015). An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In: Young, M. and Xie, T. (eds.), *Proceedings of the 2015 International Symposium on Software Testing and Analysis, ISSTA 2015, Baltimore, MD, USA, July 12-17, 2015*, pp. 24–36. ACM.
Available at: `https://doi.org/10.1145/2771783.2771791`

[128] Raselimo, M. and Fischer, B. (2019). Spectrum-based fault localization for context-free grammars. In: Nierstrasz, O., Gray, J. and 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*, pp. 15–28. ACM.
Available at: `https://doi.org/10.1145/3357766.3359538`

[129] Raselimo, M. and Fischer, B. (2021). Automatic grammar repair. In: Visser, E., Kolovos, D.S. and Söderberg, E. (eds.), *SLE '21: 14th ACM SIGPLAN International Conference on Software Language Engineering, Chicago, IL, USA, October*

17 - 18, 2021, pp. 126–142. ACM.
Available at: `https://doi.org/10.1145/3486608.3486910`

[130] Raselimo, M., Taljaard, J. and Fischer, B. (2019). Breaking parsers: mutation-based generation of programs with guaranteed syntax errors. In: Nierstrasz, O., Gray, J. and 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*, pp. 83–87. ACM.
Available at: `https://doi.org/10.1145/3357766.3359542`

[131] Rossouw, C. and Fischer, B. (2020). Test case generation from context-free grammars using generalized traversal of lr-automata. In: Lämmel, R., Tratt, L. and 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*, pp. 133–139. ACM.
Available at: `https://doi.org/10.1145/3426425.3426938`

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

[133] Roychoudhury, A. (2016). Semfix and beyond: semantic techniques for program repair. In: Naik, R., Medicherla, R.K. and Banerjee, A. (eds.), *Proceedings of the International Workshop on Formal Methods for Analysis of Business Systems, ForMABS@ASE 2016, Singapore, Singapore, September 4, 2016*, p. 2. ACM.
Available at: `https://doi.org/10.1145/2975941.2990288`

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

[135] Saha, R.K., Lyu, Y., Yoshida, H. and Prasad, M.R. (2017). ELIXIR: effective object oriented program repair. In: Rosu, G., Penta, M.D. and 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*, pp. 648–659. IEEE Computer Society.
Available at: `https://doi.org/10.1109/ASE.2017.8115675`

[136] Sakakibara, Y. (1997). Recent advances of grammatical inference. *Theor. Comput. Sci.*, vol. 185, no. 1, pp. 15–45.
Available at: `https://doi.org/10.1016/S0304-3975(97)00014-5`

[137] Santelices, R.A., Jones, J.A., Yu, Y. and Harrold, M.J. (2009). Lightweight fault-localization using multiple coverage types. In: *31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings*, pp. 56–66. IEEE.
Available at: `https://doi.org/10.1109/ICSE.2009.5070508`

[138] Schmitz, S. (2007). Conservative ambiguity detection in context-free grammars. In: Arge, L., Cachin, C., Jurdzinski, T. and Tarlecki, A. (eds.), *Automata, Languages and Programming, 34th International Colloquium, ICALP 2007, Wroclaw, Poland, July 9-13, 2007, Proceedings*, vol. 4596 of *Lecture Notes in Computer Science*, pp. 692–703. Springer. ISBN 978-3-540-73419-2.
Available at: `https://doi.org/10.1007/978-3-540-73420-8_60`

[139] Schmitz, S. (2008). An experimental ambiguity detection tool. *Electr. Notes Theor. Comput. Sci.*, vol. 203, no. 2, pp. 69–84.
Available at: `https://doi.org/10.1016/j.entcs.2008.03.045`

[140] Schröer, F.W. (2001). Amber, an ambiguity checker for context-free grammars.
Available at: `http://accent.compilertools.net/Amber.html`

[141] Slutz, D.R. (1998). Massive stochastic testing of SQL. In: Gupta, A., Shmueli, O. and Widom, J. (eds.), *VLDB'98, Proceedings of 24rd International Conference on Very Large Data Bases, August 24-27, 1998, New York City, New York, USA*, pp. 618–622. Morgan Kaufmann.
Available at: `http://www.vldb.org/conf/1998/p618.pdf`

[142] Smith, P. (2017). Hyacc.
Available at: `http://hyacc.sourceforge.net/`

[143] 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*, pp. 285–289. UNESCO (Paris).

[144] Sommerville, I. (2010). *Software Engineering (Ninth Edition)*. Pearson.

[145] Steimann, F., Frenkel, M. and Abreu, R. (2013). Threats to the validity and value of empirical assessments of the accuracy of coverage-based fault locators. In: Pezzè, M. and Harman, M. (eds.), *International Symposium on Software Testing and Analysis, ISSTA '13, Lugano, Switzerland, July 15-20, 2013*, pp. 314–324. ACM.
Available at: `https://doi.org/10.1145/2483760.2483767`

[146] Stevenson, A. and Cordy, J.R. (2014). A survey of grammatical inference in software engineering. *Sci. Comput. Program.*, vol. 96, pp. 444–459.
Available at: `https://doi.org/10.1016/j.scico.2014.05.008`

[147] Stijlaart, M. and Zaytsev, V. (2017). Towards a taxonomy of grammar smells. In: Combemale, B., Mernik, M. and Rumpe, B. (eds.), *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2017, Vancouver, BC, Canada, October 23-24, 2017*, pp. 43–54. ACM.
Available at: `https://doi.org/10.1145/3136014.3136035`

[148] Stumptner, M. and Wotawa, F. (1996). Model-based program debugging and repair. In: Tanaka, T., Ohsuga, S. and Ali, M. (eds.), *Industrial and Engineering Applications of Artificial Intelligence and Expert Systems, Proceedings of the Ninth International Conference, Fukuoka, Japan, June 4-7, 1996*, pp. 155–160. Gordon and Breach Science Publishers.

[149] Taneja, K., Xie, T., Tillmann, N., de Halleux, J. and Schulte, W. (2009). Guided path exploration for regression test generation. In: *31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Companion Volume*, pp. 311–314. IEEE.
Available at: `https://doi.org/10.1109/ICSE-COMPANION.2009.5071009`

[150] van Heerden, P., Raselimo, M., Sagonas, K. and Fischer, B. (2020). Grammar-based testing for little languages: an experience report with student compilers. In: Lämmel, R., Tratt, L. and 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*, pp. 253–269. ACM.
Available at: `https://doi.org/10.1145/3426425.3426946`

[151] Veggalam, S., Rawat, S., Haller, I. and Bos, H. (2016). Ifuzzer: An evolutionary interpreter fuzzer using genetic programming. In: Askoxylakis, I.G., Ioannidis, S., Katsikas, S.K. and 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*, vol. 9878 of *Lecture Notes in Computer Science*, pp. 581–601. Springer.
Available at: `https://doi.org/10.1007/978-3-319-45744-4_29`

[152] Wang, J., Chen, B., Wei, L. and 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*, pp. 579–594. IEEE Computer Society.
Available at: `https://doi.org/10.1109/SP.2017.23`

[153] Wang, J., Chen, B., Wei, L. and Liu, Y. (2019). Superion: grammar-aware greybox fuzzing. In: Atlee, J.M., Bultan, T. and Whittle, J. (eds.), *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, pp. 724–735. IEEE / ACM.
Available at: `https://doi.org/10.1109/ICSE.2019.00081`

[154] Weimer, W., Forrest, S., Le Goues, C. and Nguyen, T. (2010). Automatic program repair with evolutionary computation. *Commun. ACM*, vol. 53, no. 5, pp. 109–116.
Available at: `https://doi.org/10.1145/1735223.1735249`

[155] Weimer, W., Nguyen, T., Le Goues, C. and 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*, pp. 364–374. IEEE.
Available at: `https://doi.org/10.1109/ICSE.2009.5070536`

[156] Wen, W., Li, B., Sun, X. and Li, J. (2011). Program slicing spectrum-based software fault localization. In: *Proceedings of the 23rd International Conference on Software Engineering & Knowledge Engineering (SEKE'2011), Eden Roc Renaissance, Miami Beach, USA, July 7-9, 2011*, pp. 213–218. Knowledge Systems Institute Graduate School.

[157] Wong, W.E., Debroy, V., Gao, R. and Li, Y. (2014). The dstar method for effective software fault localization. *IEEE Trans. Reliab.*, vol. 63, no. 1, pp. 290–308.
Available at: `https://doi.org/10.1109/TR.2013.2285319`

[158] Wong, W.E., Gao, R., Li, Y., Abreu, R. and Wotawa, F. (2016). A survey on software fault localization. *IEEE Trans. Software Eng.*, vol. 42, no. 8, pp. 707–740.
Available at: `https://doi.org/10.1109/TSE.2016.2521368`

[159] Xin, Q. and Reiss, S.P. (2017). Identifying test-suite-overfitted patches through test case generation. In: Bultan, T. and Sen, K. (eds.), *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, Santa Barbara, CA, USA, July 10 - 14, 2017*, pp. 226–236. ACM.
Available at: `https://doi.org/10.1145/3092703.3092718`

[160] Xiong, Y., Wang, J., Yan, R., Zhang, J., Han, S., Huang, G. and Zhang, L. (2017). Precise condition synthesis for program repair. In: Uchitel, S., Orso, A. and Robillard, M.P. (eds.), *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*, pp. 416–426. IEEE / ACM.
Available at: `https://doi.org/10.1109/ICSE.2017.45`

[161] Xu, X., Debroy, V., Wong, W.E. and Guo, D. (2011). Ties within fault localization rankings: Exposing and addressing the problem. *Int. J. Softw. Eng. Knowl. Eng.*, vol. 21, no. 6, pp. 803–827.
Available at: `https://doi.org/10.1142/S0218194011005505`

[162] Xue, X. and 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*, pp. 113–122. IEEE Computer Society.
Available at: `https://doi.org/10.1109/ESEM.2013.22`

[163] Yang, X., Chen, Y., Eide, E. and Regehr, J. (2011). Finding and understanding bugs in C compilers. In: Hall, M.W. and Padua, D.A. (eds.), *Proceedings of the*

32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011, pp. 283–294. ACM.
Available at: `https://doi.org/10.1145/1993498.1993532`

[164] Yoshikawa, T., Shimura, K. and Ozawa, T. (2003). Random program generator for java JIT compiler test system. In: *3rd International Conference on Quality Software (QSIC 2003), 6-7 November 2003, Dallas, TX, USA*, p. 20. IEEE Computer Society.
Available at: `https://doi.org/10.1109/QSIC.2003.1319081`

[165] Zakari, A., Lee, S.P., Abreu, R., Ahmed, B.H. and Rasheed, R.A. (2020). Multiple fault localization of software programs: A systematic literature review. *Inf. Softw. Technol.*, vol. 124, p. 106312.
Available at: `https://doi.org/10.1016/j.infsof.2020.106312`

[166] Zalewski, M. (2017). American fuzzy lop.
Available at: `https://lcamtuf.coredump.cx/afl/`

[167] Zaytsev, V. (2009). Language convergence infrastructure. In: Fernandes, J.M., Lämmel, R., Visser, J. and Saraiva, J. (eds.), *Generative and Transformational Techniques in Software Engineering III - International Summer School, GTTSE 2009, Braga, Portugal, July 6-11, 2009. Revised Papers*, vol. 6491 of *Lecture Notes in Computer Science*, pp. 481–497. Springer.
Available at: `https://doi.org/10.1007/978-3-642-18023-1_16`

[168] Zaytsev, V. (2014). Negotiated grammar evolution. *J. Object Technol.*, vol. 13, no. 3, pp. 1: 1–22.
Available at: `https://doi.org/10.5381/jot.2014.13.3.a1`

[169] Zaytsev, V.V. (2010). *Recovery, convergence and documentation of languages.* Ph.D. thesis, Vrije Universiteit.

[170] Zelenov, S.V. and Zelenova, S.A. (2005). Generation of positive and negative tests for parsers. *Program. Comput. Softw.*, vol. 31, no. 6, pp. 310–320.
Available at: `https://doi.org/10.1007/s11086-005-0040-6`

[171] Zeller, A. (2009 01). Why programs fail. *Why Programs Fail.*

[172] Zhang, M., Li, X., Zhang, L. and Khurshid, S. (2017). Boosting spectrum-based fault localization using pagerank. In: Bultan, T. and Sen, K. (eds.), *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, Santa Barbara, CA, USA, July 10 - 14, 2017*, pp. 261–272. ACM.
Available at: `https://doi.org/10.1145/3092703.3092731`

[173] Zhang, M., Li, Y., Li, X., Chen, L., Zhang, Y., Zhang, L. and Khurshid, S. (2021). An empirical study of boosting spectrum-based fault localization via pagerank. *IEEE Trans. Software Eng.*, vol. 47, no. 6, pp. 1089–1113.
Available at: `https://doi.org/10.1109/TSE.2019.2911283`