An Industrial Case Study in Compiler Testing (Tool Demo)

Vadim Zaytsev
Raincode Labs, Brussels, Belgium, vadim@grammarware.net

Abstract

Compiler construction is one of the oldest areas of software engineering, yet despite its maturity it has underdeveloped sides such as compiler testing. There exist many disparate methods for testing parsers, optimisers and other components, but no unified methodology that consumable by practitioners from a book to be directly applied to fulfil their needs.

Instead of striving to cover all theoretical aspects of compiler testing in one paper, we present a case study for an ongoing project of a relatively large size for our company (2 years, 3–6 devs, ~500kLOC), a clean room compiler development effort in replicating a 4GL. We built a model-based test data generator, consuming manually written specs and generating necessary test code in the 4GL, in the host language, and in auxiliary DSLs (batch files, XML project descriptions), to both the developers' and the customer's satisfaction. The number of specs is 927 at the publication time, while the number of test cases generated from them, is 6268. All these tests have been run prior to shipping for the last 49 releases of the compiler, both to ensure the lack of regression and to report on the project overall progress. The generated tests are separated into 11 categories which the paper details in the hope that the classification will aid in seeking related work and in pushing this line of research forward.

CCS Concepts • **Software and its engineering → Compilers**; **Software testing and debugging**;

Keywords compiler testing, legacy, 4GL

ACM Reference Format:
Vadim Zaytsev. 2018. An Industrial Case Study in Compiler Testing (Tool Demo). In *Proceedings of the 11th ACM SIGPLAN International Conference on Software Language Engineering (SLE '18), November 5–6, 2018, Boston, MA, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3276604.3276619

1 Introduction

There are two cardinally opposite views on software testing. One can be defined as Dijkstra's famous *"testing shows the presence, not the absence of bugs"* [9, p.21]. The other one was advocated by Goodenough as *"properly structured tests are capable of demonstrating the absence of errors in a program"* [13], which puts testing on the same level as verification which has always been viewed as its bigger and smarter cousin. (*"[If] you have [been] given the proof of correctness, [you] can dispense with testing altogether"* [28, p.51]). The three middle ground sweet spots commonly found in software engineering, are:

- Best effort: especially for certification purposes, it is important to demonstrate the intent to break claimed functionality, even if such attempts ultimately fail. In practice, however, it is relatively rare to invest in testing significantly without finding any bugs at all, since in general an average software system is of imperfect quality [31, 39, 49].
- Coverage-driven: defining some metric of how good a test suite is, and working towards increasing it up to some exhaustion point. It has been known for a long time that *"tests based solely on the internal structure of a program are likely to be unreliable"* [13]. Instead, we should focus on conditions that can be observably violated, and test for all combinations of them.
- Refactoring support: test cases can encapsulate existing or desired behaviour of the system before its internal structure is about to change, and then used to ensure that the change did not affect the execution semantics [12]. This path is commonly taken when dealing with legacy code [10].

Compiler testing is an interesting subtopic with many challenges. There is definitely industrial need and demand for it, but the usual time pressure does not allow for in depth investigations and methodological explorations. In the rest of the paper we will explain how such challenges were faced in one standalone project.

As an example, we take an ongoing project of Raincode Labs. Its origins and peculiarities will be briefly described below—for a more extended version the readers are invited to explore *Parser Generation by Example* [45, §1]. For legal reasons we will continue calling our primary client of this project, 𝔄. It is a company working in the banking sector, which owns a multi-million line codebase. It was developed over decades of company growth and contains most of its business rules and IT assets. Besides COBOL and PL/I which are routinely encountered in our line of business, the codebase contains almost 70k modules in a fourth-generation

SLE '18, November 5–6, 2018, Boston, MA, USA

language we will call 𝔅. Even though 𝔄 has over 150 developers actively creating new software in that language on a daily basis, 𝔅 has been classified by the top management as a liability for the future and scheduled for retirement in its current incarnation. We are now in the process of writing a full-fledged compiler for 𝔅 targeting the .NET Framework. When the project is completed, it will allow 𝔄 to deploy their products on commonplace hardware or modern platforms such as Azure, to write hand-tweaked components in modern programming languages such as C$^\sharp$ and, most importantly, to hire young professionals otherwise frightened off by the prospect of learning an obscure dying language as the first job requirement. Currently we have reached the point of compiling the entire codebase and performing the successful end-to-end runs that originate in a window on an end user's computer, go through all the layers of architecture to the database, and display the query result back to the user.

The documentation of 𝔅 is partly non-existent, partly outdated and ultimately protected legally by an explicit disclaimer that only paying customers of 𝔅's current rights owner are allowed to read it (which rules us out). Hence, the development relies primarily on the analysis of the code owned by 𝔄, as well as on collaboration with 𝔄's domain experts. The source artefacts are written in several interrelated domain-specific notations, ranging from JSON-like and XML-like notations to "proper" algorithmic imperative textual coding syntaxes. The details of those notations are irrelevant for the focus of this paper, but their existence and plurality are relevant signs of the problem complexity.

2 Test Kinds

Tests are used for at least three purposes within this project:

- Measure the progress of the project, providing solid figures to 𝔄 to demonstrate planned compiler development milestones. We can appreciate how this purpose might seem mundane and distastefully political, but it plays an important, hard to overrate, role in industrial projects. For instance, shipping timestamp formatting was not enough, but stating that it works based on the fact that it has been tested on all format strings present in the codebase, and providing a collection of compilable and executable test cases to support the claim, had a stronger and more lasting effect.
- Catch regressions and monitor preparedness of the compiler to be shipped. In this project we have been delivering new versions of our compiler on an approximately weekly basis 49 times by now, and it quickly became a mandatory procedure to run all available test cases, including the overall six-hour long recompilation of the entire codebase of 𝔄, to uncover unexpected regressions. On many occasions (~30%), such regressions caused a delay of a day or two in the next version delivery, which was almost always better than shipping a defective version knowingly.
- Implement new functionality, TDD (test-driven development [4, 29]) style, or refactor existing functionality the classic way [12], by having tests cases represent the current behaviour and their non-regression serve as the evidence of preserving it.

To satisfy these three emerged objectives, we need to make the test cases easy to explain and communicate and as exhaustive as possible. Formally, we see each test case specification as a definition of a *binding* between some form of input and some sort of output, and explain it below as such. Within this project, we are actively using the following ten kinds of test cases:

- **R (recognised)** binds a textual input of the compiler to the fact that it is parsable (i.e., the parser must succeed to produce a non-empty smell-free parse tree).
- **P (parsed)** binds a textual input of the compiler to the graph, with the intention of having it parsed exactly as the structure provided.
- **Q (roundtrip)** technically has the same binding as the R-tests, but during the execution of the test it is parsed, unparsed and reparsed again, with equivalent result of the two parses.
- **N (normalised)** has the same binding as for P-tests, but the equivalence of the resulting tree with the expected baseline is done later in the pipeline, after the symbol table has been created and certain syntactic sugar constructs have been desugared.
- **T (typed)** binds a compileable program to some piece of its metadata. Classically it is a symbol table, which associates the name of an entity such as a variable, with its derived attributes such as a type and scope. The T-tests contain predominantly typing assertions.

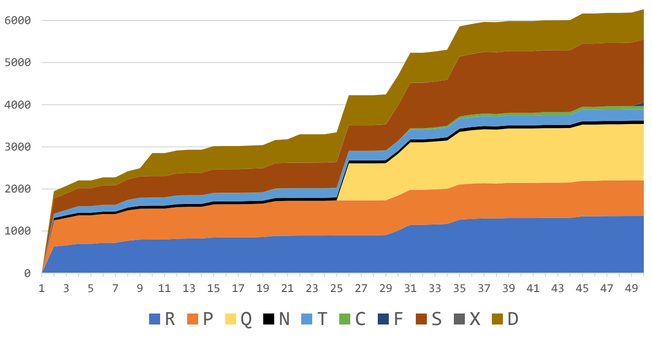

Figure 1. A plot showing how the number of test cases increases per release, as reported by the language documentation accompanying each shipment. Older figures are harder to track but according to the version control, the work on test cases started at least 50 weeks before the first official release.

- **C (compiled)** binds a textual input with the fact that this program is supposed to be consumed by the compiler which should produce a sensible DLL (a dynamic-link library, an atomic deployment entity on Windows). The resulting DLL is not checked for executability, but its bytecode is checked for verification with PEVerify [30]. If the input program is erroneous to some extent, then the warning(s) and the error(s) produced by the compiler, are also checked by such a C-test.
- **F (failed)** has the same binding as C-tests but the compilation is expected to fail yielding a particular error specified by the test case definition (i.e., the errors are fatal: missing declarations, type inference incompatibilities, parsing failures, etc).
- **S (successful)** binds a textual input that can be fully parsed, normalised, typed, compiled and executed, with an a priori known outcome. This is the most advanced kind of tests we have, which stresses several phases of the compiler and the language runtime. All tests are written in a style to be self-sufficient for execution (i.e., there are no dependencies among test modules) and to print all the data that can help determining whether the execution was normal. Then, the testing framework catches the output produced by executing a test case, and compares it with the baseline value specified in the test case definition.
- **X (exception)** has the same binding as S-tests but the output usually contains an error code of the exception that is supposed to manifest itself.
- **D (direct)** is a piece of bare C$^\sharp$ code that emulates some small fragment of a runtime library function usage. Technically all D-tests can be lifted to be S-tests, but sometimes it is more comfortable for debugging purposes to have the testing code as close to the code under test as possible. We seize this opportunity to write D-tests in an elaborate way: for instance, a date formatting test case loops over *all* valid days, months and years, and compares the output of the library formatter with the one given by .NET.

Figure 1 shows how the number of tests grows with time. In particular, we see that R-tests grow continuously, P-tests used to follow them and stabilised mid-project, Q-tests were introduced at a late stage, F-test and X-tests are underdeveloped and D-tests grow in spikes.

3 Usefulness

R-tests were noticeably helpful during the early stages of the project when not all of the codebase can be parsed yet. The theoretical basics of incremental grammar engineering have already been covered previously by Klint et al. [18] as well as by Lämmel [23, 25], with case studies dissected by van den Brand et al. [5, 6], Sellink and Verhoef [33], Lämmel and Verhoef [20, 27], Tratt [36], Visser [38], Alves and Visser [1], Zaytsev [40–44], Vavrová and Zaytsev [37]. Later in the project the R-tests were mostly used to control possible regressions after grammar adaptation tweaks, and otherwise were always expected to stay in the green. At the time of the publication we had 1359 R-tests.

The **P-tests** were used for the same purpose in the first stages of the project, as an aid to tweak usual problematic places in the grammar: ambiguities, priorities, associativity, etc. However, they were useful later in the project as well to fine-tune certain parsing oddities. For instance, in $\mathfrak{B}$ there is a multiple branching construct that can look like this:

```
CASEOF A
    CASE B C D
ENDCASE
```

In this construction, `A` is compared with `B` and the branch is entered if the equality is observed. However, `C` can also turn out out to be a call to a user-defined procedure. In this case, if the procedure has a non-void return type and that type is compatible with the type of `A`, it is treated as a part of the condition (so the branch is also entered if `A` is equal to `C`). If `C` returns nothing, it is treated as the first statement of the branch. If it returns a value that cannot be converted into a type compatible with `A`, then it is still treated as the first statement of the branch, but the return value is simply discarded. Such details became known to us many months after the start of the project, because they represent collaboration of several features of the software language (space as a separator between values of the same case and the ability to call non-void procedures as void ones) and thus are hard to anticipate. Our eventual implementation of this context-sensitive parsing involved parsing its first context-free approximation and performing subsequent tree rewriting [3] based on the symbol table once it can be constructed. Having P-tests with an evident binding between a source code input and the expected tree structure that the parser is expected to deliver, was a great help in both developing and debugging the solution.

The further into the project, the more the emphasis shifted from P-tests towards **Q-tests** which could be inferred automatically by the framework. Specifying a tree structure manually was made as comfortable as possible with a special DSL, but was still rather labour intensive. Hence, in later phases of the project, when parsing was not expected to fail or misbehave, it was better to invest the effort elsewhere.

The **S-tests** received most attention both from our side and from $\mathfrak{A}$ (who even developed their own test programs, which we included in the suite but did not deem all that useful for modularity purposes: it is essentially one big 3000 LOC program). The state of the art in S-tests is biased towards testing optimising compilers and is trying to answer the question of how to ensure that the optimised compiled code has the same observable behaviour as the unoptimised compiled code. Approaches vary from formal specification of optimiser patterns to a straightforward brute force comparison checking of outputs of the two versions [16].

F-tests sounded like a good idea, especially in the context of well-known drawbacks of positive-only test suites [2, 47], but were not of much use until recently. Their runtime counterparts **X-tests** are planned to be exhaustively covered in the near future because the design of exceptions is stabilising and it is known that reliability is only reachable when *"all exceptions are testable and tested"* [13].

4 Future Plans and Related Work

For comprehensive coverage of related work on compiler testing [19] and verification [8] we refer to corresponding elaborate surveys. In this section we collect a few directions that have been explored in the past in related research endeavours but not utilised to their fullest within this project.

One of the first things worth exploring is *R-testing with generated test data*. There is a famous algorithm of generating a small set of short test sentences from a context-free grammar, initially proposed by Purdom [32] and later extended with backtracking [17], actions [7], controlled coverage [26], etc. Just as with mundane software testing, in parser testing rule coverage is known to be important but fundamentally insufficient [24]. Coverage can be seen as two-dimensional [14] with a syntactic axis (nonterminals, rules, etc.) and a semantic one (computations, calculations, attributes, etc). The coverage issue even can be and is ignored by some of the approaches, but in that case it is unclear when the test case adding process can/must end [19].

Fischer et al. [11] use grammar-based test data generators to compare several alternative grammars for reportedly "the same" language. It was advocated to be used to choose among available grammars, but we can rely on it to tweak the parser for performance without regression (differentially comparing alternative implementations), or simply for increasing the collection of short debuggable programs that are expected to be parsable (perhaps with manual post-processing by a language expert filtering out pointless generated test data).

There is a lot of related work on *N-testing of optimisers*. For example, OTK [48] is a system that allows for abstract specification of an optimiser in such a way that the metamodel of its input data can be constructed, from which test data is generated and fed into the optimiser. Without the break of generality, we can consider it a unidirectional scenario synchronising the test suite when the compiler is changed, with the algorithm specification being manually inferred from the compiler and the rest of the steps are automated.

CompCert [22] uses a similar approach by constructing semantic preservation proofs in the calculus of inductive and coinductive constructions. This is mostly useful in making so-called "verified compilers" where most program transformations (typing, stack/register allocating, instruction scheduling, etc) are formally verified and thus more secure. This approach is mostly applied to code generation and is aimed at proving that the source program and the code it compiled to, have the same observable behaviour (by proxy: if a source program satisfies some specification, the code does the same). CompCert is aimed at a subset of C, it has direct competitors like Verisoft [21], and there are similar initiatives for other languages, such as VerifiCard [35] aimed at a subset of Java. Each one of them is a massive effort since it includes a complete formal specification of the underlying language semantics. To the best of our knowledge, there are neither research teams nor companies currently investing substantial effort into building a certified compiler for C$^\sharp$ or for any of the 4GLs.

In our setup, the compiler infrastructure (e.g., the specification of the language) and the testing infrastructure (test specifications themselves plus the framework to run them plus the generator that does most of the scrupulous heavy lifting) are mostly disjoint, but they do not have to be. For example, $\Lambda\Delta\Lambda$ [15] is a system for tool-aided construction of such a detailed language specification (in the form of modular attribute grammars, a typical language constructs library and a knowledge base) that both a prototype interpreter and test data generator can be inferred from it automatically. Looking away from the prototypical nature of the $\Lambda\Delta\Lambda$, we can count it as an early attempt to encapsulate the *intent* behind each test. Whether we will merge the infrastructures for this project or not, remains to be decided, but intent encapsulation and refinement is going to be the crucial core if we ever hope to achieve a comfortable setup with test suite coverage and bidirectional evolution.

5 Conclusion

In this paper we have shared lessons learnt from writing an industrial compiler for a legacy 4GL. From the first day of the project, testing of the compiler was among the things that enjoyed most of our attention. After a short introduction into the topic and the project in §1, we presented 10 kinds of test cases mostly used by us in §2. Their value for the project was discussed and compared to prior research in §3. More related work was pointed out in §4 along with future plans.

Finding the right balance between occasional ad hoc tests and running massive overnight recompilation jobs, is not easy, but we believe there are many lessons to be learnt on this way, and hope to contribute a few steps there.

Acknowledgement

The work on the test generator presented in this paper was mostly a solitary venture, but the author would like to acknowledge other developers who contributed their code to this project, in alphabetical order: Darius Blasband, Kelly Hollis, Thierry Miceli, Sudipta Mukherjee, Daan Nijs, Benoît Ragoen, as well as 𝔄's senior developers H.F.E., S.O.H., B.J.J., P.S. and J.S.S., who served as 𝔅's domain experts and compiler alpha testers. Early thoughts on the topic of compiler testing were also shared by the author at LangDevCon [34] in March 2018 and at CoCoDo [46] in April 2018. The author is grateful to participants of both events for feedback.

References

[1] Tiago Laureano Alves and Joost Visser. 2008. A Case Study in Grammar Engineering. In *Revised Selected Papers of the First International Conference on Software Language Engineering (SLE) (LNCS)*, Dragan Gašević, Ralf Lämmel, and Eric Van Wyk (Eds.), Vol. 5452. Springer, 285–304. https://doi.org/10.1007/978-3-642-00434-6_18

[2] Dana Angluin. 1980. Inductive Inference of Formal Languages from Positive Data. *Information and Control* 45, 2 (1980), 117–135. https://doi.org/10.1016/S0019-9958(80)90285-5

[3] Franz Baader and Tobias Nipkow. 1998. *Term Rewriting and All That.* Cambridge University Press.

[4] Kent Beck. 2003. *Test-Driven Development by Example.* Addison-Wesley.

[5] Mark van den Brand, M. P. A. Sellink, and Chris Verhoef. 1997. Obtaining a COBOL Grammar from Legacy Code for Reengineering Purposes. In *Proceedings of Second International Workshop on the Theory and Practice of Algebraic Specifications (Electronic Workshops in Computing)*, M. P. A. Sellink (Ed.). Springer.

[6] Mark van den Brand, M. P. A. Sellink, and Chris Verhoef. 1998. Current Parsing Techniques in Software Renovation Considered Harmful. In *Proceedings of the Sixth International Workshop on Program Comprehension.* IEEE Computer Society, 108–117. https://doi.org/10.1109/WPC.1998.693325

[7] Augusto Celentano, S. Crespi Reghizzi, P. Della Vigna, C. Ghezzi, G. Granata, and F. Savoretti. 1980. Compiler Testing Using a Sentence Generator. *Software: Practice and Experience* 10, 11 (1980), 897–918. https://doi.org/10.1002/spe.4380101104

[8] Maulik A. Dave. 2003. Compiler Verification: A Bibliography. *ACM SIGSOFT Software Engineering Notes* 28, 6 (2003), 2. https://doi.org/10.1145/966221.966235

[9] P. Ercoli and F. L. Bauer. 1970. *Software Engineering Techniques.* Conference Report. J. N. Buxton and B. Randell (Eds.). NATO Science Committee. http://homepages.cs.ncl.ac.uk/brian.randell/NATO/nato1969.PDF

[10] Michael Feathers. 2004. *Working Effectively with Legacy Code.* Prentice Hall.

[11] Bernd Fischer, Ralf Lämmel, and Vadim Zaytsev. 2012. Comparison of Context-free Grammars Based on Parsing Generated Test Data. In *Post-proceedings of the Fourth International Conference on Software Language Engineering (SLE 2011) (LNCS)*, Uwe Aßmann and Anthony Sloane (Eds.), Vol. 6940. Springer, 324–343. https://doi.org/10.1007/978-3-642-28830-2_18

[12] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. 1999. *Refactoring: Improving the Design or Existing Code.* Addison-Wesley.

[13] John B. Goodenough and Susan L. Gerhart. 1975. Toward a Theory of Test Data Selection. In *Proceedings of the International Conference on Reliable Software.* ACM, 493–510. https://doi.org/10.1145/800027.808473

[14] Jörg Harm and Ralf Lämmel. 2000. Two-dimensional Approximation Coverage. *Informatica Journal* 24, 3 (2000).

[15] Jörg Harm, Ralf Lämmel, and Günter Riedewald. 1997. The Language Development Laboratory ($\Lambda\Delta\Lambda$). In *Selected Papers of the Eighth Nordic Workshop on Programming Theory (NWPT'96): Research Report No. 248*, Olaf Owe, Magne Haveraaen, and Olav Lysne (Eds.). Department of Informatics, University of Oslo.

[16] Clara Jaramillo, Rajiv Gupta, and Mary Lou Soffa. 2002. Debugging and Testing Optimizers through Comparison Checking. *Proceedings of the Workshop on Compiler Optimization Meets Compiler Verification (COCV), ENTCS* 65, 2 (2002), 83–99. https://doi.org/10.1016/S1571-0661(04)80398-0

[17] Uwe Kastens. 1980. *Studie zur Erzeugung von Testprogrammen für Übersetzer.* Bericht 12/80. Institut für Informatik II, University Karlsruhe.

[18] Paul Klint, Ralf Lämmel, and Chris Verhoef. 2005. Toward an Engineering Discipline for Grammarware. *ACM Transactions on Software Engineering Methodology (ToSEM)* 14, 3 (2005), 331–380. https://doi.org/10.1145/1072997.1073000

[19] A. S. Kossatchev and M. A. Posypkin. 2005. Survey of Compiler Testing Methods. *Programming and Computing Software* 31 (Jan. 2005), 10–19. Issue 1. https://doi.org/10.1007/s11086-005-0008-6

[20] Ralf Lämmel and Chris Verhoef. 2001. Semi-automatic Grammar Recovery. *Software — Practice & Experience* 31 (December 2001), 1395–1438. Issue 15. https://doi.org/10.1002/spe.423

[21] D. Leinenbach and E. Petrova. 2008. Pervasive Compiler Verification — From Verified Programs to Verified Systems. *ENTCS* 217 (2008), 23–40. https://doi.org/10.1016/j.entcs.2008.06.040

[22] Xavier Leroy. 2009. Formal Verification of a Realistic Compiler. *Commun. ACM* 52, 7 (2009), 107–115. http://gallium.inria.fr/~xleroy/publi/compcert-CACM.pdf

[23] Ralf Lämmel. 2001. Grammar Adaptation. In *Proceedings of the International Symposium of Formal Methods Europe: Formal Methods for Increasing Software Productivity (FME) (LNCS)*, Vol. 2021. Springer, 550–570. https://doi.org/10.1007/3-540-45251-6_32

[24] Ralf Lämmel. 2001. Grammar Testing. In *Proceedings of the Fourth International Conference on Fundamental Approaches to Software Engineering (FASE) (LNCS)*, Vol. 2029. Springer, 201–216. https://doi.org/10.1007/3-540-45314-8_15

[25] Ralf Lämmel. 2018. *Software Languages: Syntax, Semantics, and Metaprogramming.* Springer. https://doi.org/10.1007/978-3-319-90800-7

[26] Ralf Lämmel and Wolfram Schulte. 2006. Controllable Combinatorial Coverage in Grammar-Based Testing. In *Proceedings of the 18th IFIP TC6/WG6.1 International Conference on Testing of Communicating Systems (TestCom'06) (LNCS)*, Umit Uyar, Mariusz Fecko, and Ali Duale (Eds.), Vol. 3964. Springer Verlag, 19–38. https://doi.org/10.1007/11754008_2

[27] Ralf Lämmel and Chris Verhoef. 2001. Cracking the 500-Language Problem. *IEEE Software* 18 (2001), 78–88. Issue 6. https://doi.org/10.1109/52.965809

[28] P. Naur and B. Randell (Eds.). 1969. *Software Engineering.* Scientific Affairs Division, NATO.

[29] James W. Newkirk and Alexei A. Vorontsov. 2004. *Test-Driven Development in Microsoft .NET.* Microsoft Press.

[30] Ron Petrusha, Maira Wenzel, Luke Latham, Tom Pratt, and Yisheng Jin. 2017. Peverify.exe (PEVerify Tool). https://docs.microsoft.com/en-us/dotnet/framework/tools/peverify-exe-peverify-tool

[31] David S. Platt. 2007. *Why Software Sucks... and what You Can Do about it.* Addison-Wesley.

[32] Paul Purdom. 1972. A Sentence Generator for Testing Parsers. *BIT Numerical Mathematics* 12, 3 (1972), 366–375.

[33] M. P. A. Sellink and Chris Verhoef. 2000. Development, Assessment, and Reengineering of Language Descriptions. In *Proceedings of the Fourth Conference on Software Maintenance and Reengineering.* IEEE Computer Society, 151–160. https://doi.org/10.1109/CSMR.2000.827323

[34] Tijs van der Storm and Markus Völter. 2018. LangDevCon: a Language Developers Meetup. http://langdevcon.org.

[35] Martin Strecker. 2002. Formal Verification of a Java Compiler in Isabelle. In *Proceedings of the 18th International Conference on Automated Deduction (CADE) (LNCS)*, Vol. 2392. Springer, 63–77. https://doi.org/10.1007/3-540-45620-1_5

[36] Laurence Tratt. 2007. Evolving a DSL Implementation. In *Revised Papers of the Second International Summer School on Generative and Transformational Techniques in Software Engineering (LNCS)*, Ralf Lämmel, Joost Visser, and João Saraiva (Eds.), Vol. 5235. Springer, 425–441. https://doi.org/10.1007/978-3-540-88643-3_11

[37] Nicole Vavrová and Vadim Zaytsev. 2017. Does Python Smell Like Java? *The Art, Science and Engineering of Programming (‹Programming›)* 1 (April 2017), 11–1–11–29. Issue 2. https://doi.org/10.22152/programming-journal.org/2017/1/11

[38] Eelco Visser. 2007. WebDSL: A Case Study in Domain-Specific Language Engineering. In *Revised Papers of the Second International Summer School on Generative and Transformational Techniques in Software Engineering (LNCS)*, Ralf Lämmel, Joost Visser, and João Saraiva (Eds.), Vol. 5235. Springer, 291–373. https://doi.org/10.1007/978-3-540-88643-3_7

[39] Joost Visser, Sylvan Rigal, Pascal van Eck, Gijs Wijnholds, and Rob van der Leek. 2016. *Building Maintainable Software, C♯ Edition. Ten Guidelines for Future-Proof Code.* O'Reilly Media.

[40] Vadim Zaytsev. 2005. Correct C♯ Grammar too Sharp for ISO. In *Participants Workshop, Part II of the Pre-proceedings of the International Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE 2005).* Technical Report, TR-CCTC/DI-36, Universidade do Minho, Braga, Portugal, 154–155. Extended abstract.

[41] Vadim Zaytsev. 2011. MediaWiki Grammar Recovery. *Computing Research Repository (CoRR)* 1107.4661 (July 2011), 1–47. http://arxiv.org/abs/1107.4661

[42] Vadim Zaytsev. 2012. Notation-Parametric Grammar Recovery. In *Post-proceedings of the 12th International Workshop on Language Descriptions, Tools, and Applications (LDTA 2012)*, Anthony Sloane and Suzana Andova (Eds.). ACM Digital Library. https://doi.org/10.1145/2427048.2427057

[43] Vadim Zaytsev. 2012. The Grammar Hammer of 2012. *Computing Research Repository (CoRR)* 1212.4446 (Dec. 2012), 1–32. http://arxiv.org/abs/1212.4446

[44] Vadim Zaytsev. 2015. Grammar Zoo: A Corpus of Experimental Grammarware. *Fifth Special issue on Experimental Software and Toolkits of Science of Computer Programming (SCP EST5)* 98 (Feb. 2015), 28–51. https://doi.org/10.1016/j.scico.2014.07.010

[45] Vadim Zaytsev. 2017. Parser Generation by Example for Legacy Pattern Languages. In *Proceedings of the 16th International Conference on Generative Programming: Concepts and Experience (GPCE)*, Matthew Flatt and Sebastian Erdweg (Eds.). ACM, 212–218. https://doi.org/10.1145/3136040.3136058

[46] Vadim Zaytsev. 2018. CoCoDo: Raincode Labs Compiler Coding Dojo. https://cocodo.github.io.

[47] Sergey Zelenov and Sophia Zelenova. 2006. Automated Generation of Positive and Negative Tests for Parsers. In *Formal Approaches to Software Testing*, Wolfgang Grieskamp and Carsten Weise (Eds.). LNCS, Vol. 3997. Springer Berlin / Heidelberg, 187–202. https://doi.org/10.1007/11759744_13

[48] Sergey Zelenov and Sophia Zelenova. 2007. Model-Based Testing of Optimizing Compilers. In *Testing of Software and Communicating Systems*, Alexandre Petrenko, Margus Veanes, Jan Tretmans, and Wolfgang Grieskamp (Eds.). LNCS, Vol. 4581. Springer, 365–377. https://doi.org/10.1007/978-3-540-73066-8_25

[49] Andreas Zeller. 2009. *Why Programs Fail: A Guide to Systematic Debugging.* Elsevier Science.