Grammar Maturity Model

Vadim Zaytsev

Universiteit van Amsterdam, The Netherlands, vadim@grammarware.net

Abstract. The evolution of a software language (whether modelled by a grammar or a schema or a metamodel) is not limited to development of new versions and dialects. An important dimension of a software language evolution is maturing in the sense of improving the quality of its definition. In this paper, we present a maturity model used within the Grammar Zoo to assess and improve the quality of *grammars in a broad sense* (structural models) and give examples of activities possible for each level.

1 Motivation

Grammar Zoo [41] is an initiative aimed at systematic collection of *grammars in the broad sense* — structural definitions of software languages; annotation of each grammar with information about its source, original format and authors; complementing each grammar with details about how it was obtained; documenting usages of each grammar — its derivatives, tools, documents and other grammars; and finally making all these grammars publicly available in a variety of formats. It has many possible uses:

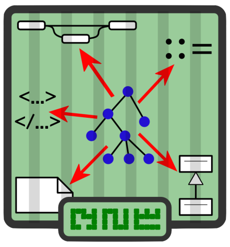

Fig. 1. Grammar Zoo, http://slps.github.io/zoo

Interoperability testing. Suppose that we have identified multiple grammars of the same intended language that correspond to its different frontends. To test their interoperability, one can do code reviews or develop a test suite, but a better, more systematic, way is to generate such a suite and compare or converge those grammars automatically. An approach for that has been proposed in [9] and evaluated by two case studies involving 4 Java grammars and 33 TESCOL grammars, which were extracted from parser specifications and became one of the early fragments of the Grammar Zoo. Its replication with a simplified algorithm applied to adapted grammars, used 28 grammars of different languages from the ANTLR grammar repository [3].

Grammar recovery heuristics. There have been many successful attempts of reusing grammatical knowledge embedded in various software artefacts like parser specifications, data format descriptions or metamodels. Some focused on idiosyncratic properties of the source notation, others tried to generalise the relaxed ways of treating the baseline artefact with heuristic rules for splitting/combining names, matching parentheses, etc [39]. The more grammars can be recovered with heuristics, the better validated and motivated they become.

Empirical grammar analysis. Grammar metrics is a mature field of research, but more elaborate characterisations such as "top" or "bottom" nonterminals are common in grammar-based papers. When the repository of various grammars has grown to a reasonable size and comprised over 500 grammars, the micropattern mining methodology [11] was applied to infer characterisations by mining this repository [40]. They can in turn be used for clustering grammars based purely on statistical data about sets of indicators and for triggering grammar mutations [3].

Sample-based grammar engineering. Crafting a good grammar suitable for the intended task is not an easy activity: is must not be too restrictive or too permissive, must be compatible with the intended technology, reasonable in terms of performance, readable for a human expert, machine processable, etc (some of the aspects obviously being more relevant under different circumstances). Having access to a large corpus of existing successful grammars (together with the information about their actual applications, of course), could aid the grammar engineer either implicitly (essentially by "code reading") or explicitly (by reusing an existing grammar as a starting point for developing a domain-specific language).

Grammar components. There is ongoing work on identifying *semantic components* of software languages that correspond to concepts like loops, variables, exception throwing, etc [16]. By using a combination of program slicing and clone detection techniques on a large enough corpus, we can identify *syntactic components* of software languages and investigate whether there is any correspondence with semantic components.

Since the first days of the Grammar Zoo it became painfully clear that there is no general agreement about maturity of grammars: even a grammar in a narrow sense (say, a parser specification) can be optimised with one parsing technology in mind while rendering it useless for other technologies, and for grammars in a broad sense the meaning behind measurements becomes even more obscure. In early publications around the project there was also mentioning of the Grammar Tank, a sibling project collecting smaller grammars of DSLs — it seemed unreasonable to put a tiny grammar extracted from a ten-line Haskell ADT definition right next to a "real" grammar defining a programming language like C# or C++, extracted from a 1000+ page manual. Later we imported grammars from the Atlantic Metamodel Zoo, from the Relax NG collection and many other places, adding to the complexity and heterogeneity. In the end, all grammars have been merged into one collection, each annotated with all kinds of metadata. One of the important annotations is a maturity level, which we define essentially as the level of details and consistency and a measure of how close the grammar is to actual use, notwithstanding its domain and intent.

The next session introduces the usual tools available for grammar engineering activities. §3 actually presents the levels of the maturity model. In §4, we give an example of how a grammar can be used in practice, referring to its possible maturity level. §5 discusses related work and §6 concludes the paper.

2 Grammar Manipulation

The arsenal of grammar manipulation is rather large, and for the sake of better understanding of the rest of the paper, we name a few important methods. All of them are to some extent represented in the GrammarLab[1] library and useful for programming in the Rascal metaprogramming language [21].

XBGF [25] is an operator suite for programmable grammar transformations. It consists of more than 50 operators like ***renameN*** for renaming a nonterminal or ***unfold*** for unfolding a nonterminal reference to its definition. Operators have universally defined semantics and known properties — some can be proven to be language-preserving refactorings, others extend the defined language or restrict it. Transformations are programmed against this operator suite by choosing an operator and providing appropriate arguments for it to make it applicable to the grammar being transformed — i.e., ***unfold***(n) is a valid transformation step, if n is a nonterminal present in the grammar. Similar techniques exist for other grammar manipulation frameworks [6,22,23].

SLEIR [42] is a similar suite for grammar mutations [38], which are generalised transformations automatically applicable on a large scale: enforcing a normal form, changing a naming convention, systematically refactoring harmful constructions into equivalent harmless ones are examples of grammar mutations.

Grammar analysis is a research direction on its own, where based on observable facts obtained from grammar metrics [31] or micropatterns [40] we can make estimations and draw conclusions about a grammar's suitability for specific tasks, compatibility with a technology, backwards compatibility through versions, interoperability among various tools, etc.

The last mentioned group of instruments was important for creating the Grammar Zoo and growing it, and less so for using it as such. Grammar extraction obtains grammars in a broad sense from software artefacts containing grammatical knowledge (source code, documentation, protocol definitions, algebraic data types, databases, etc). Grammar recovery [24,34,39] works similarly with sources using deceivingly familiar notations in an inconsistent or otherwise unexpected way — manually written, out of date, incomplete, etc.

3 A Quality/Maturity Model for Grammars

We distinguish among the following grammar levels:

- ⧫ A grammar is **fetched** if it can be put in a file which we claim to contain grammatical knowledge. A fetched grammar is usually written in an (E)BNF-like notation, but it can also be an XML Schema schema, an Ecore model, a parser specification, etc. A grammar from an undisclosed ISO standard or a grammar built in a proprietary tool is *not* fetched, since we have no possible way to extract the knowledge from it. A compiler is therefore not a fetched grammar since the grammatical knowledge is ingrained too deep

[1] GrammarLab: `http://grammarware.github.io/lab`.

in it and requires special techniques to be fetched. The sources of a compiler, however, can be considered fetched, since further extraction can be semi-automated, and the result will depend mostly on the source and not on the extraction algorithm. Hence, a corpus of programs in a given language is also *not* a fetched grammar, but an APTA (Augmented Prefix Tree Acceptor) or a DFA (Deterministic Finite Automaton) constructed from it by a grammatical inference algorithm, is a fetched grammar. A fetched grammar can contain unreadable symbols, incorrect indentation, parts written in an unknown notation or a natural language, or even be present in a form of an image or a manuscript.

- ⧫ A grammar is **extracted**, if it was fetched and then successfully processed by a (hopefully automatic) grammar extractor, possibly also corrected of typographical, text recognition and similar errors and converted into a context-free grammar or, more broadly speaking, to a Boolean grammar [29]. (Not venturing beyond context freedom is simply a consequence of the current lack of theoretical foundations for linking classic context-dependent grammars to generalised types — in general, aligning the Chomsky hierarchy [4] with Barendregt λ-cube [2]). An extracted grammar is suitable for automated processing: it can be pretty-printed in a range of different ways and transformed by general means, without writing a tool specific for its peculiar notation or format: syntax diagrams, Relax NG schemata, algebraic data types, parser specifications all lose their notational differences when they are being extracted, but they retain all structural peculiarities such as using a particular style of recursion (syntax diagrams are incapable of expression left recursion, and some parsing algorithms tend to avoid it as well), the lack or presence of terminal symbols (anything that defines an abstract syntax, has no terminals) or nonterminal symbols (classic regular expressions [20] have no notion of named subcomponents), etc.
- ⧫ A grammar is **connected**, if it was extracted and then processed to not contain unwanted top sorts (defined but never used) and bottom sorts (used but not defined). These two quality indicators were proposed in [33,34] and discussed in more detail in [23] before being formalised as micropatterns [40]. Connecting a grammar can be done automatically with a mutation [38,42], semi-automatically with a sequence of transformation steps, or by editing it inline. Connecting is a simple procedure that allows to start making some claims about the grammar, since it enables its formalisation (the classic $\langle \mathcal{N}, \mathcal{T}, \mathcal{P}, s\rangle$ model of a grammar requires it to have one known starting symbol) and possible application of grammar-based algorithms (in particular grammar-based test data generation expects the grammar to be connected because otherwise it is futile to use any coverage criteria). In a broader sense, a connected grammar always relies on some underlying mechanism of testing or validation which ensures its general quality — as opposed to the extracted grammar which can be an output of an automated extractor and never checked nor inspected further.
- ⧫ A grammar is **adapted**, if it is connected and then transformed towards satisfying some constraints: it could be complemented with a lexical part,

or its naming convention can be adjusted, or certain metaconstructs can be introduced to or removed from its syntax. The adaptation has a clear intent: adding a lexical part can lead to automated generation of a parser or at least a recogniser; conforming to a naming convention can enable the use of the grammar in specific language workbenches, etc.

- A grammar is **exported**, if it was adapted and then a piece of grammarware was generated from it. An exported grammar bidirectionally and possibly nontrivially corresponds to a real piece of grammarware such as a compiler or a code analysis or transformation tool.

Each Grammar Zoo entry has one *fetched* grammar: ones with less than one are "non-entries" that can perhaps be referred to, but can under no circumstances be machine processed; having more than one fetched grammar can happen for cases such as multiple websites mirroring one another, because an additional check is required to assert them to be equal. If several extractors are available (e.g., one straightforward one and one heuristic-based error-correcting one), there can be multiple *extracted* grammars per entry. Similarly, there can be several grammars of level *connected* and up per entry, varying per their extracted source and methods of processing. Especially different *adapted* grammars per entry are common, since each of them corresponds to a specific intentional adaptation project. At this point in the history of the Grammar Zoo we have not yet experienced the need to explicitly distinguish the reason for adaptation of each grammar: some are massaged for better readability, some adjusted with parsing in mind, some are disambiguated [35], some adapted for testing purposes [3,9,32], etc. We intentionally leave the hierarchy as general as it is, and leave its extension to future work.

4 A Maturation Path

Suppose we would like to have a piece of grammarware to parse and analyse programs in a particular software language — say, COBOL or PHP. Being constrained in time, we usually start by looking for existing grammars: once we find one that seems reasonably suitable for our needs, we can declare it *fetched*. If a fetched grammar of our intended language is already in the Grammar Zoo, it can save us the search, the frustration from websites having been taken down, as well as the ambiguity about the true source of the grammar.

In the simplest cases, grammar extraction methods and tools can be applied to a fetched grammar with reasonable success. There is quite a collection of them readily available within GrammarLab, and it is fairly straightforward to use notation-parametric grammar extraction [39], if the input notation is anything like BNF or EBNF; write out XSLT mapping templates, if the input notation is XMI, XSD or anything from the XMLware technical space. If all available methods fail, we can attempt to apply grammar recovery tools, which have heuristics known to overcome frequent erroneous patterns. Once some reasonable kind of non-empty grammar is obtained or if it was in the Grammar Zoo to begin with, the grammar can be considered to be *extracted*.

An extracted grammar can be processed further, analysed, transformed, exported, imported, visualised etc — there are many tools in the GrammarLab that can do it directly, and they can also help to export it to a format readily consumable by other metagrammarware. However, it does not mean that this grammar would "work" there, whatever that might mean. There are some sensible metrics, constraints and grammar analyses established in state of the art grammar recovery [24,34], that are almost universally useful in improving the quality of a grammar. For instance, we would like to identify the starting symbol of a grammar, establish it being unique. Furthermore, for each parts unreachable from it, we would like to make a decision and either remove them or connect to the rest of the grammar. This is usually done by programming the corresponding steps in XBGF [25], SLEIR [42], GDK [22], TXL [6] or any other grammar manipulation language. This usually implies manual examination of a grammar and its metrics by an expert, making the appropriate decisions and then documenting the changes. Once this is completed, we speak of having a *connected* grammar.

The next step is grammar adaptation [23]: a goal-specific continuation of grammar transformation activities. For example, if we have decided to parse and analyse code in COBOL or PHP, this is our goal, and in case of Rascal [21] it will mean having a complete concrete syntax specification and a suitable algebraic data type. Both can be obtained from a connected grammar, but the adaptation strategies are different. For a syntax specification, we need to add the lexical part, specify layout, increasingly disambiguate the grammar, etc. For a data type, we should think of its suitability for specifying our analyses later, and we can easily eliminate all terminal symbols and massage the remaining abstract grammar to enable more concise and readable patterns. These streaks of activity end up with an *adapted* grammar each.

Finally, our two grammars (or a syntax specification and a data type, or a grammar and a schema — terminology may vary) are ready to be *exported* — we do this with out of the box renderers, possibly followed by manual polishing such as adding documenting annotations and inserting copyright notices. It is not unusual for an exported grammar to be linked to a specific tool which it forms a part of.

5 Related Work

Lämmel and Verhoef [24] were the first to propose the notion of a *grammar level*[2] to specify a quality level or a recovery status of a grammar. We have conceptually inherited that hierarchy and extended it to accommodate more important details. Their level 1 broadly corresponds to an extracted grammar in our model, level 2 to connected, levels 3 and 4 (depending on how thorough it has been tested) to adapted. An exported grammar is at level 5 if it either demonstrates the absence of manual steps in grammar deployment, or documents them by its

[2] These "grammar levels" are essentially CMM-like levels applied to grammars, unrelated to well-known "grammatical levels" used for a range of grammar metrics [18,31].

existence. Our contribution lies in rethinking and generalising this hierarchy for all grammars in a broad sense and empirically applying it to hundreds of grammars (as opposed to CFGs of one or two programming languages).

El-Attar and Miller [7] have shown how antipattern detection can be used to improve the quality of models (in their case, use case models, but a similar technique for metamodels is not unthinkable). This approach is conceptually very similar to a typical grammar engineering activity when the language engineer identifies which metaconstructs are incompatible with the technology indended for use, and refactors them away. At the current state of software language engineering, the first part is the most appropriate to do with micropatterns [40] and the second part with grammar mutations [42].

France and Rumpe [10] have investigated the relation between quality and MDE from the pragmatic point of view and found out that one of the biggest advantages comes from the opportunity to reuse previously assessed models (and submodels) of known quality in the development of new ones. The same argumentation, lifted to the metalevel, can be found in the first section of this paper when we show some possible uses for the Grammar Zoo.

It is impossible to talk about quality without mentioning ISO/IEC 9126 [15], an official standard specifying quality of a software system as a product (hence, also a grammarware system). It identifies quality characteristics such as functionality, reliability, usability, efficiency, maintainability, portability, and continues to break them down into smaller pieces. There have been some attempts to formalise and detail parts of it up the the point of implementability — in particular, maintainability has received a lot of attention [13], but the general agreement is to treat the standard as a set of guidelines and considerations, not as an immediately implementable model. An extensive review of research activities concerning the quality models in the particular context of model-driven software development, was made about papers in top conferences in 2000–2009 [28].

Welker's maintainability index in the Coleman-Oman model is often claimed useful for quickly assessing the maintainability (and hence also quality, per ISO 9126 [15]) of software. Its formula exists in various slightly adjusted incarnations in the academic literature and is commonly denoted as either just "maintainability" [1, p.155][5, p.46] or "maintainability index" / "MI" [12, p.255][26, p.15]. As it turns out, MI is inappropriate for grammarware purposes. Suppose we apply aggressive normalisation and unchain all chain productions and inline all nonterminal symbols that are used only once and have only one production rule. Such a transformation preserves stability of the grammar, but obviously reduces its analysability, changeability and testability. Yet MI shows improvement. Since stability, analysability, changeability and testability are the main components of maintainability per ISO 9126 [15], MI does not adequately measure maintainability.

Discussions on language quality are abundant in the context of general purpose programming languages [36,14,37], modelling languages [30,19] and domain-specific languages [27,17]. Their contributions are mostly in a form of sets of guidelines that are with some evidence and expertise linked to the quality of the final product. Our maturity model can be seen as an attempt to formalise and standardise a part of software language quality — namely, its structural model. There is some strong evidence that this syntactic aspect is not dominating when considering software language quality in general [8].

6 Conclusion

We have briefly introduced the field of grammarware manipulation and a project of collecting grammars in a broad sense — structural definitions of structure found in software systems. We have also presented a maturity model of several distinct levels on which grammars can reside — the model gradually came into existence during several years of research on grammar analysis and improvement. The renovated Grammar Zoo with this new maturity model is about to be deployed and made available for public use. This model was essential in its growth from a dozen self-made grammars to over 1500 entries of fetched level and above.

There are many reasons for models to evolve: some are externally motivated and concern the natural evolution (improvement as a response to contextual changes), some concern the actual use of the models. In this particular paper we have treated quality as basically the level of details in a model extracted from its real-life counterpart, which was directly linked to the possibility of automated processing. This made sense in our context — collecting and analysing grammars in a broad sense — but it stands to reason that the same considerations would apply for any fact extraction models and software models in general.

There is some evidence in adjacent fields that models which evolved in one dimension (e.g., a programming language grammar extracted from a book, cleaned up, polished and turned into a validation tool) can be very profitably (re)used for improving or constructing models that evolved in another dimension (e.g., a programming language grammar from the next edition of the book). For us, proving this or even collecting substantial evidence by convincing case studies, remains future work.

References

1. D. Ash, J. Alderete, P. W. Oman, and B. Lowther. Using Software Maintainability Models to Track Code Health. In *Proceedings of the International Conference on Software Maintenance (ICSM' 94)*, pages 154–160. IEEE Computer Society, 1994.
2. H. P. Barendregt. Introduction to Generalized Type Systems. *Journal of Functional Programming*, 1(2):125–154, 1991.
3. E. Butrus. Satisfying Coverage Criteria by Grammar Mutations and Purdom's Sentence Generator. Master's thesis, Universiteit van Amsterdam, Aug. 2014.
4. N. Chomsky. On Certain Formal Properties of Grammars. *Information and Control*, 2(2):137–167, 1959.
5. D. Coleman, D. Ash, B. Lowther, and P. Oman. Using Metrics to Evaluate Software System Maintainability. *Computer*, 27:44–49, 1994.

6. T. R. Dean, J. R. Cordy, A. J. Malton, and K. A. Schneider. Grammar Programming in TXL. In *Proceedings of the Second IEEE International Conference on Source Code Analysis and Manipulation (SCAM 2002)*, pages 93–102. IEEE, 2002.
7. M. El-Attar and J. Miller. Improving the Quality of Use Case Models Using Antipatterns. *Software and System Modeling*, 9(2):141–160, 2010.
8. M. Erwig and E. Walkingshaw. Semantics First! In *Proceedings of the Fourth International Conference on Software Language Engineering*, SLE'11, pages 243–262. Springer, 2012.
9. B. Fischer, R. Lämmel, and V. Zaytsev. Comparison of Context-free Grammars Based on Parsing Generated Test Data. In U. Aßmann and A. Sloane, editors, *Post-proceedings of the Fourth International Conference on Software Language Engineering (SLE 2011)*, volume 6940 of *LNCS*, pages 324–343. Springer, 2012.
10. R. B. France and B. Rumpe. Modeling to Improve Quality or Efficiency? An Automotive Domain Perspective. *Software and System Modeling*, 11(3):303–304, 2012.
11. J. Gil and I. Maman. Micro Patterns in Java Code. In *Proceedings of the 20th ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA'05)*, pages 97–116. ACM, 2005.
12. J. H. Hayes, S. C. Patel, and L. Zhao. A Metrics-Based Software Maintenance Effort Model. In *Proceedings of the Eighth Euromicro Working Conference on Software Maintenance and Reengineering (CSMR'04)*, pages 254–258. IEEE Computer Society, 2004.
13. I. Heitlager, T. Kuipers, and J. Visser. A Practical Model for Measuring Maintainability. In *Proceedings of the Sixth International Conference on Quality of Information and communications Technology (QUATIC'07)*, pages 30–39. IEEE, 2007.
14. C. A. R. Hoare. Hints on Programming Language Design. Technical report, Stanford University, Stanford, CA, USA, 1973.
15. ISO/IEC. *ISO/IEC 9126. Software Engineering — Product Quality.* ISO/IEC, 2001.
16. A. Johnstone, P. D. Mosses, and E. Scott. An Agile Approach to Language Modelling and Development. *Innovations in Systems and Software Engineering*, 6(1-2):145–153, 2010.
17. G. Karsai, H. Krahn, C. Pinkernell, B. Rumpe, M. Schneider, and S. Völkel. Design Guidelines for Domain Specific Languages. In M. Rossi, J. Sprinkle, J. Gray, and J.-P. Tolvanen, editors, *Proceedings of the Ninth OOPSLA Workshop on Domain-Specific Modeling (DSM 2009)*, pages 7–13, Mar. 2009.
18. A. Kelemenová. Grammatical Levels of the Position Restricted Grammars. In *Proceedings on Mathematical Foundations of Computer Science*, pages 347–359. Springer, 1981.
19. S. Kelly and R. Pohjonen. Worst practices for domain-specific modeling. *IEEE Software*, 26(4):22–29, July 2009.
20. S. C. Kleene. Representation of Events in Nerve Nets and Finite Automata. *Automata Studies*, pages 3–42, 1956.
21. P. Klint, T. van der Storm, and J. Vinju. EASY Meta-programming with Rascal. In J. M. Fernandes, R. Lämmel, J. Visser, and J. Saraiva, editors, *Post-proceedings of the Third International Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE 2009)*, volume 6491 of *LNCS*, pages 222–289. Springer, Jan. 2011.

22. J. Kort, R. Lämmel, and C. Verhoef. The Grammar Deployment Kit. System Demonstration. *Electronic Notes in Theoretical Computer Science*, 65(3):117–123, 2002. Workshop on Language Descriptions, Tools and Applications (LDTA).
23. R. Lämmel. Grammar Adaptation. In *Proceedings of the International Symposium of Formal Methods Europe on Formal Methods for Increasing Software Productivity*, volume 2021 of *LNCS*, pages 550–570. Springer, 2001.
24. R. Lämmel and C. Verhoef. Semi-automatic Grammar Recovery. *Software—Practice & Experience*, 31(15):1395–1438, Dec. 2001.
25. R. Lämmel and V. Zaytsev. Recovering Grammar Relationships for the Java Language Specification. *Software Quality Journal (SQJ)*, 19(2):333–378, Mar. 2011.
26. A. Liso. Software Maintainability Metrics Model: An Improvement in the Coleman-Oman Model. *Software Engineering Technology*, pages 15–17, August 2001.
27. M. Mernik, J. Heering, and A. M. Sloane. When and How to Develop Domain-Specific Languages. *ACM Computing Surveys*, 37(4):316–344, 2005.
28. P. Mohagheghi, V. Dehlen, and T. Neple. Definitions and Approaches to Model Quality in Model-based Software Development - A Review of Literature. *Information and Software Technology*, 51(12):1646–1669, Dec. 2009.
29. A. Okhotin. Boolean Grammars. *Information and Computation*, 194(1):19–48, 2004. http://users.utu.fi/aleokh/papers/boolean_grammars_ic.pdf.
30. R. F. Paige, J. S. Ostroff, and P. J. Brooke. Principles for Modeling Language Design. *Information and Software Technology*, 42(10):665–675, 2000.
31. J. F. Power and B. A. Malloy. A Metrics Suite for Grammar-based Software. *Journal of Software Maintenance and Evolution: Research and Practice*, 16:405–426, Nov. 2004.
32. P. Purdom. A Sentence Generator for Testing Parsers. *BIT*, 12(3):366–375, 1972.
33. A. Sellink and C. Verhoef. Generation of Software Renovation Factories from Compilers. In *Proceedings of 15th International Conference on Software Maintenance (ICSM 1999)*, pages 245–255, 1999.
34. M. P. A. Sellink and C. Verhoef. Development, Assessment, and Reengineering of Language Descriptions. In *Proceedings of the Fourth European Conference on Software Maintenance and Reengineering (CSMR 2000)*, pages 151–160, Mar. 2000.
35. M. G. J. van den Brand, J. Scheerder, J. J. Vinju, and E. Visser. Disambiguation Filters for Scannerless Generalized LR Parsers. In N. Horspool, editor, *Compiler Construction 2002 (CC 2002)*, pages 143–158, 2002.
36. A. van Wijngaarden. Orthogonal Design and Description of a Formal Language. MR 76, SMC, 1965.
37. N. Wirth. On the Design of Programming Languages. In *IFIP Congress*, pages 386–393, 1974.
38. V. Zaytsev. Language Evolution, Metasyntactically. *Electronic Communications of the EASST; Bidirectional Transformations*, 49, 2012.
39. V. Zaytsev. Notation-Parametric Grammar Recovery. In A. Sloane and S. Andova, editors, *Post-proceedings of the 12th International Workshop on Language Descriptions, Tools, and Applications (LDTA 2012)*. ACM Digital Library, 2012.
40. V. Zaytsev. Micropatterns in Grammars. In M. Erwig, R. F. Paige, and E. V. Wyk, editors, *Proceedings of the Sixth International Conference on Software Language Engineering (SLE 2013)*, volume 8225 of *LNCS*, pages 117–136. Springer, 2013.
41. V. Zaytsev. Grammar Zoo: A Repository of Experimental Grammarware. *Fifth Special issue on Experimental Software and Toolkits of Science of Computer Programming (SCP EST5)*, 2014. In print.
42. V. Zaytsev. Software Language Engineering by Intentional Rewriting. *Electronic Communications of the EASST; Software Quality and Maintainability*, 65, 2014.