Grammar Zoo:
A Corpus of Experimental Grammarware

Vadim Zaytsev

Software Analysis & Transformation Team (SWAT),
Centrum Wiskunde & Informatica (CWI), The Netherlands

Abstract

In this paper we describe a way to compose a corpus of grammars in a broad sense in order to enable reuse of knowledge accumulated in the field of grammarware engineering. The Grammar Zoo, which displays the results of grammar hunting for big grammars of mainstream languages, as well as collecting grammars of smaller DSLs, is already operational and publicly supplies its users with grammars that have been recovered from different sources of grammar knowledge, varying from official language standards to community-created wiki pages.

We summarise recent achievements in the discipline of grammarware engineering, that made the creation of such a corpus possible. We also describe in detail the technology that is used to build and extend such a corpus. The current contents of the Grammar Zoo are listed, as well as some possible future uses for them.

Keywords: grammarware engineering, grammar recovery, experimental infrastructure, curated corpus

1. Introduction

This paper contains a method to compose a corpus of grammars in a broad sense. Having such a corpus could be profitable for mining new properties and patterns from the existing body of grammatical knowledge, for comparing grammar-based techniques and developing new ones. Formal grammars are inherently complex software artefacts, and until recently it was technically unfeasible to create such a large scale corpus, so in existing literature most case studies involve one, two or no more than a handful of grammars, and many statements about software language design remain statistically unchecked and empirically unvalidated or even unprovable.

The main contributions of this paper are:

Email address: `vadim@grammarware.net` (Vadim Zaytsev)

- The Grammar Zoo as the big and still growing corpus of 569 grammars in a broad sense.
- An open source toolkit for supporting the creation and expansion of the Grammar Zoo.
- A Grammar Zoo entry metadata model for enabling efficient sampling and reuse.

The paper is organised as follows: §2 explains the problem in detail, motivates the need for its solution, sets goals, defines context and envisions possible problems. In §3 we revisit those grammarware engineering challenges that have already been addressed in prior work, and have made the current development possible. In §4, the metadata model for the corpus is presented and the tools available for grammar extraction, recovery and evolution are highlighted. §5 lists the current contents of the Grammar Zoo and sketches directions for future work. §6 concludes the paper.

2. A repository of grammars

In [1], Klint et al. have defined the field of "grammarware" and identified its set of problems, promises, principles and challenges. The foundation of their work was formed by the vast existing body of knowledge about formal grammars, compiler construction, metaprogramming, source code analysis, term rewriting, parsing techniques, generative programming, attribute grammars, graph transformation and other adjacent fields. In the years after that, there have appeared many publications contributing directly to this domain, and the "engineering discipline for grammarware" from the ideal long term goal has turned into a technically achievable and partially even achieved objective. However, comparison of different grammar-based methods is still hindered by the relative lack of stability in grammar metrics and their sensitivity to many factors ranging from grammar development style (e.g., horizontal or vertical style of writing production rules has a substantial impact on the number of rules) to the choice of grammar-based technology (some syntactic notations are more expressive than others; some technologies implicitly expect grammars to be written with a specific kind of recursion, etc).

In contemporary software engineering, especially in empirical studies thereof, a similar problem has been addressed by introducing a curated collection of code artefacts [2]. In model-driven engineering, many metamodels — artefacts commonly compared to grammars in the literature — have been collected in one place to form a corpus available in many formats [3]. Such reference corpora can be used as an input of various newly proposed analysis and transformation techniques, allowing their output to be measured and reported in a systematic manner.

In [4], van Wijngaarden states that a powerful and elegant language should not contain many concepts and should be explainable in few words. In [5], Wirth concludes that language simplicity should be achieved through modularity and not through generalisation. In [6], Hoare claims that in programming language design, simplicity is different from and more important than modularity. In [7], Mernik et al. argue that language modularity is positively influenced by the presence of a textual notation. In [8], Völter et al. tie the multitude and diversity of general purpose programming languages to their domain-specific optimisations. In [9], Chomsky elaborates that semantic and statistical considerations should be of no consequence to the grammatical structure of the language. In [10], Erwig and Walkingshaw maintain that language design should be semantics driven. In [11], Tratt establishes that the evolution of a domain specific language mostly involves adding functionality found in general purpose programming languages. In [12], Herrmannsdörfer et al. observe that modeling languages evolution is bound to requirements creep and technological progress. In [13], Hutchinson et al. claim that effectiveness of a domain specific language and narrowness of its domain are in inverse proportion. Many more claims like these can be found in academic and engineering publications about grammarware and related topics — most are backed by expert opinions and case studies of manageable size. However, we still lack the luxury of (re)formulating them as research hypotheses and subsequently validating against (a chosen part of) the corpus of grammars and languages. We construct Grammar Zoo in order to enable such activities in the future.

It has been previously noted by Do et al. that obtaining the right kind of infrastructure for setting up experiments is nontrivial and labour intensive, and its usefulness has huge impact on future experiments [14]. According to Do et al., the users of such infrastructure mostly face the following challenges:

Supporting replicability across experiments. Homogeneity or well-documented heterogeneity of the collected artefacts and completeness of metadata are the key factors for the creators of the infrastructure, to help addressing this challenge [14, 15, 16].

Supporting aggregation of findings. Systematic capture of the experimental context is required to complement high replicability, in order to guarantee correct aggregation of findings from different experiments [14, 17].

Reducing the cost of controlled experiments. In order to facilitate painless artefact reuse, artefact organisation needs to be standardised, they need to be complete in some sense (preferably by conforming to a well-defined completeness level) and require as little manual handling as possible [14].

Obtaining sample representativeness. The main problems foreseen by [14] in allowing the users to acquire representative samples, are small sample sizes and sampling bias. These are to be addressed by including many artefacts obtained from different heterogeneous sources.

Isolating the effects of individual factors. Isolating software language design concerns and decoupling conceptual modules within one software language have always been challenging problems, and still pose great difficulty. Since this is an open research question, we will not be able to prevent all problems that it leads to.

2.1. Illustration: Grammar Zoo utilisation

Suppose that we have assembled the Grammar Zoo as a collection of various grammars. What kind of research questions we can answer with it and what kind of problems can we address? We provide some scenarios below.

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 [18] 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.

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 [19]. The more grammars can be recovered with such heuristics, the better validated and motivated they become.

Empirical grammar analysis. Grammar metrics are a mature field of research, but more elaborate characterisations such as "top" or "bottom" nonterminals are common in grammar-based papers. Given a large enough repository of various grammars, one can reapply the micropattern mining methodology from [20] and infer such characterisations by mining this repository — this is the most recent successful result obtained by using the Grammar Zoo [21].

2.2. Software Language Processing Suite

The creation of the Grammar Zoo was facilitated by the tools accumulated within the Software Language Processing Suite project, also abbreviated as SLPS [22]. It started in 2008[1] on the Sourceforge platform and migrated to GitHub in 2012.

In particular, SLPS contains the following groups of tooling:

[1] Ralf Lämmel, *Software Language Processing Samples (SLPS)*. MSDN blog post in *Grammarware, Haskellware, XMLware*, May 2008, http://blogs.msdn.com/b/ralflammel/archive/2008/05/23/software-language-processing-samples-slps.aspx.

- *Formats* for storing grammars [23], grammar transformations [24], metalanguage notations [25], grammar documentation [26], etc. These formats have evolved by being used in methods and tools working with grammars in a broad sense and thus abstract from many technical idiosyncrasies. The formats are designed to describe essential grammatical knowledge and are therefore compatible with many different platforms and technologies.

- *Transformers* for grammars [24], metasyntactic specifications [27], language documents [26], etc. These can be used either as first class entities to encapsulate grammar evolution or correction steps, or just as a technical aid for changing grammar-related artefacts.

- *Megamodels* are an emerging technology for modelling linguistic architecture of a system — that is, a megamodel is model of a software system with some elements denoting very complex notions like languages, grammars, technologies and stakeholders. Megamodels are used especially well when nontrivial differences between similar technologies need to be spotted, modelled and resolved [28]. SLPS contains a Rascal megamodelling library that is compatible with the most recently developed general purpose megamodelling language MegaL [29].

- *Extractors* for locating and obtaining fragments of grammatical knowledge from various software artefacts such as parser specifications, data type definitions, grammars of various kinds, data schemata, etc. Since extractors are essential for the process of creation of the Grammar Zoo, they will be described in more detail in §4.5.

- *Recovery tools* are advanced extractors that use heuristics or context conditions to infer corrections to the source during or immediately after the extraction of a grammar. In the past, grammar recovery has been done manually [30] or semi-automatically [31]. The state of the art tools allow fully automatic grammar recovery to work based on a specification of the expected metalanguage [19].

- *Analysers* for investigating grammar properties, calculating grammar metrics [32, 33], expressing and collecting micropatterns [21], often aid in other grammar-related activities.

- *Exporters* take care of grammar visualisation and producing grammars in a form recognisable by other language workbenches and metaprogramming environments.

- *Test generators* can automatically support a test suite exercising all features of a software language described by a grammar and possibly limited by coverage criteria [34, 18].

- *Documentation* support is present as well: in its foundation we have a unified format for language documents [26] that was constructed by analysing and comparing hundreds of software language specifications, descriptions, standards, studybooks and alike.

- *Frameworks* for working with the above are present in languages like Prolog, Python and Rascal.

Software Language Processing Suite is a noncommercial effort mainly aimed at developing tool prototypes and proofs of concepts [22]. It is publicly available on the internet for distribution through a free and open software license (CC-BY: Attribution[2]).

2.3. Main objectives

We define the **main goals** of the Grammar Zoo as follows:

- Collecting grammars in the broad sense — structural definitions of software languages.
- Annotating each grammar with information about its source, original format and authors.
- Complementing each grammar with details about how it was obtained, extracted, recovered, transformed, etc.
- Documenting usages of each grammar — its derivatives, tools, documents and other grammars.
- Making all grammars publicly available in a variety of formats.

What the Grammar Zoo is **not** about:

- It is not about collecting parsers. Collecting a large number of them systematically would mean committing to a specific parsing technology or even a specific grammar manipulation framework (metagrammarware). Also, not all grammars are meant to be used for syntactic analysis of textual data.
- It is not about unifying syntactic notation for grammars. Numerous attempts to unify textual representations of context-free grammars have failed in the past. Instead of fighting the diversity of notations, we embrace it and develop tools that can deal with it. We ultimately aim at storing the pure grammatical knowledge and exporting it on demand according to the users' needs.
- It is not about enforcing the quality and level of grammars. Heterogeneous content is also inherent to the field of grammarware engineering in its current state: different tasks expect different properties from grammars. Instead, we aim at properly documenting such differences so that the

[2] CC Attribution 3.0 Unported, http://creativecommons.org/licenses/by/3.0/.

corpus can be used to obtain representative sets of grammars that are similar in some required sense.

2.4. Summary

Recent advances in the field of grammarware engineering make it possible to design and engineer a corpus of grammars in a broad sense. Having such a repository of grammars, each annotated with metadata about its source, means of extraction and recovery, its evolution and linked tooling, would allow us to mine it for similarities and singularities, as well as to use it as a common testing ground for grammar-based methods. Based on the experience gained from several such experiments in the past, in the next sections we will compose the Grammar Zoo, which will collect as many grammars as we can secure.

3. Previously addressed challenges

The Grammar Zoo is a relatively new initiative which it was unfeasible to create a long time ago. The following subsections are dedicated to the challenges which were solved during the last years by various researchers, and helped us to found this initiative on top of their methodologies. We credit the following research directions:

Grammar extraction. Automated recovery of grammars from existing artefacts is required, otherwise adding each grammar to the corpus will always be a separate project with its own specifics.

Grammar evolution. When grammar recovery goes decidedly beyond extraction, it involves bringing systematic changes to the grammar. Proper documentation of such adjustments relies on an advanced transformational infrastructure.

Metalinguistic evolution. With many notations for syntactic definitions being used in various grammar-based toolkits, it is crucial to be able to export each grammar in a variety of notations, or perhaps even in a new notation defined on-the-fly.

Generating browsable documentation. One of the most common uses for a grammar, beside generating grammarware code, is facilitating its inspection.

3.1. Grammar extraction and automated recovery

Basically, there are three main challenges in developing grammar extraction: the unnecessary diversity of notation [47, 25]; the error-prone manual process of grammar creation and typesetting; and the semantic gap between the metalanguages. The first (notational) challenge is addressed by the notation-parametric approach [19] that requires a formal specification of a notation and can base the extraction steps on it. The second (lexical) challenge is solved by heuristics expressing relaxed conformance to the intended notation. The third (semantic)

Language	Source	To L1	To L2	To L3	To L4	To browse	Result	Ref
ANSI COBOL III	spec in “general format”	manual	manual	manual	—	—	no	[35]
IBM COBOL	SC26-9046-03 spec	Prolog	—	—	—	Prolog	[36]	
IBM OS PL/I V2R3	SC26-4308-02 spec	Prolog	—	—	—	Prolog	[36]	
Ericsson MSC	ITU Z.120 (MS-Word document)	Perl	manual	—	—	Perl	no	[37]
ANSI COBOL 85	hard copy	reused	reused	reused	ASF	—	no	[30]
Ada 95	ISO/IEC 8652:1995 spec	Prolog	—	—	—	Prolog	[36]	
Ericsson SSL	hypertext documentation	bnf2sdf	manual	manual	ASF	Box	no	[37]
Ericsson PLEX	BNF rules in compiler sources	6×SDF	interactive	manual	ASF	—	no	[38]
IBM VS COBOL II	language spec [39]	Prolog	interactive	manual	ASF	Prolog	[36]	[31]
C#	ECMA-334 spec [40]	manual	FST	manual	FST	ASF	[36]	[41]
FL	various grammars in a broad sense	automated	XBGF	XBGF	—	XSLT	[22]	[23]
Java	Sun language specs [42, 43, 44]	Python	XBGF			BGF	[22]	[24]
MediaWiki	community-created grammar	5×EDD	XBGF	—	—	XSLT	[22]	[45]
SLPS Zoo	42 sources	varies	varies	—	—	XSLT	[22]	[46, ...]
SLPS Tank	53 sources	varies	varies	—	—	XSLT	[22]	[46, ...]
Grammar Zoo	569 sources	varies	w.i.p.	—	—	XSLT	[22]	here

Table 1: The summary of grammar recovery activities that took place before Grammar Zoo. The early SLPS Zoo and SLPS Tank, as well as the current Grammar Zoo, are added for comparison to the bottom of the table. By “To L1” we denote a method of obtaining a “level 1 grammar” [31], a freshly extracted CFG corrected from misspellings. Similarly, by “L2” we mean a “level 2 grammar”, which has undergone some at least superficial analysis and has been maximally connected. A level 3 grammar (L3) is complete with a lexical part and any other idiosyncratic additions that allows to generate useful grammarware from it. A level 4 grammar (L4) defines a tool that has been exercised on large codebase. §4.3 elaborates on the notion of a grammar level.

challenge is avoided by using the "extraction through abstraction" [23] approach of extracting pure grammatical knowledge from the available artefacts while abstracting from the technology-specific details. All three challenges are addressed by automated tools from SLPS, a more detailed overview of them will follow in §4.5.

The progress of the grammar extraction methods with automated error recovery, also summed up in Table 1, was as follows[3] [19]:

Message Sequence Charts was a DSL described in a Word document, which was converted to Postscript due to the lack of appropriate API in 1996 when the recovery took place. The Postscript document was converted to an ASCII file which was processed by a Perl script and produced BNF rules, which were in turn manually edited with all 14 changes claimed to be documented. Another script was used to generate a hypertext form of a grammar suitable for browsing [37].

COBOL grammar capable of handling a range of language dialects, was recovered in 1997. The help of a Master student was used to convert 1100 production rules of the ANSI COBOL 85 standard to SDF [49]. A long and sophisticated process of forced coupling followed, leading to (disciplined) changes brought both to the codebase and to the grammar, and resulting in capability of the adjusted grammar to parse the adjusted source code [30].

Switching System Language was a proprietary DSL documented in a set of HTML files containing its grammar in an BNF dialect they called SBNF. The recovery endeavour was reported in 2000 and is a remarkable milestone in a way that it was an attempt to use precise parsing on an unreliable source. A range of (as we are now aware) typical issues arose such as naming convention violations and non-matching brackets, and significant amount of interactive grammar adjustments was needed. The project succeeded also due to development support of the ASF+SDF Meta-Environment, resulting again in the situation where an adjusted SBNF grammar was used to parse adjusted syntax rules [37].

Programming Language for EXchanges was a complex DSL consisting of 20 sublanguages ("sectors") and having over 60 Mb of grammarware source code. The mining process delivered fragments of BNF found in the comments, which with the help of six parsers were transformed to pure aggregated BNF and subsequently to SDF, which was combined with a lexer. The project took only two weeks and resulted in parsing 8 MLOC of unmodified PLEX, as reported in 2001 [38].

[3] NB: we limit the mentioned initiatives to those which were extracting grammatical knowledge from software artefacts that were already known to contain it and to *define structures*: context-free grammars, compiler sources, parser definitions, XML schemata, etc. For a comprehensive overview of methods and initiatives of extracting/inferring grammatical knowledge from a collection of *language instances*, we redirect the reader to [48].

IBM VS COBOL II recovery project is one of the most complicated among those reported in academic sources. A raw grammar was extracted from the language documentation, which was not trivial since it used "railroad track" kind of syntax diagrams instead of purely textual BNF. After static errors were taken into account and the lexical syntax was added, the project entered the phase of test-driven correction and completion. Several phases of grammar recovery followed, including beautification, modularisation, disambiguation and adaptation. The recovery project was reported in 2001 [31], and its outcome was made freely available for reuse from the authors' website [36].

C# recovery gave rise to very similar problems, even though it targeted a newly developed language in 2005, unlike COBOL which had existed for almost half a century by then. In order to parse C# code, the project involved manual transition from the ECMA-produced PDF to LLL and intensive grammar transformation with FST and GDK, as reported in [41] and [50, §3].

FL was an artificial toy functional language used to demonstrate the principles of grammar convergence in [23]. The emphasis of that work was put to the extraction itself (i.e., how to obtain a grammar in a broad sense from an existing artefact), not the recovery part (i.e., how to reconstruct missing or damaged information). This lightweight extraction work with choosing only reliable sources as starting points: concrete syntax definitions in SDF [51], parser specifications in ANTLR [52], definite clause grammars in Prolog [53], grammars in TXL [54], object models in Java [44], document schemata of XML [55]. This opened the possibility of incorporating existing grammar repositories like ANTLR Grammar List [56], TXL Grammar Collection [57], Altantic Metamodel Zoo [3], etc.

Java grammar recovery required tolerance to overcome layout inconsistencies and other lexical deviations of the source grammars, which was expressed as a list of heuristics and described in detail in [24]. The same technology was used later for grammars of C, C++ and C# found in other ISO standards written in the same syntactic notation [50].

Notation-parametric recovery method [19] relies on encapsulating commonly varying details of the syntactic notation in a notation specification [25] and binding the recovery heuristics to those variation points. This approach allows to extract a grammar in a never-seen-before notation in a matter of several minutes required to compose such a specification.

3.2. Grammar evolution support

The ability to express grammar evolution steps as first class executable entities, was identified as one of the crucial components of the engineering discipline for grammarware [1]. Below we try to cover the existing spectrum of the grammar evolution support[4]:

Attribution as a claim that one grammar is “derived from” or “written from” some other source like another extracted grammar or language documentation, is a common way to represent grammar evolution in many existing grammar repositories [56, 57, 3]. However, these derivation steps are rarely documented, so this level of detail is not informative enough for any automated verification of such claims or even for their consistency management.

Documented patches as lists of changes that were applied to the grammar in order to get from the original to the final version, are much more useful, even if they are not entirely formal. They have been used in early grammar engineering projects [37], and are also not uncommon in grammar-related bug reports [62]. What is often missing in such lists, is justification for the proposed solution: for example, compare [62] with [24].

Grammar transformation operators are a functional way of representing patches. Each change step is expressed as a function application, with a function being one of the predefined operators from a grammar transformation operator suite. First such operator suites that were published, are FST [63] and the one used for COBOL grammar recovery [64, 31]. The differences between them are insignificant for the current paper — a discussion about them can be found in [24]. An ideologically similar approach was demonstrated by TXL [54], a framework where grammar specialisation for each task is an essential part of the grammarware engineering paradigm.

Higher-level grammar transformation operators like “fold a nonterminal” or “perform a safe refactoring” were shown to be more useful and better maintainable than the low-level ones like “remove any part of a grammar” or “replace any expression by another expression everywhere” [65]. This approach was used in Grammar Deployment Kit, experimental metagrammarware that was used successfully in a number of projects [66]. A similarly advanced operator suite for the modelware technological space, has also been developed and published [67]: even though always taking coupled evolution of models into account when dealing with metamodel changes, has its challenges [68], they seem to be addressed in recent research [69].

Recovery domain-specific operators were introduced for Grammar Recovery Kit as a demonstration of how a successful grammar recovery project can be undertaken and documented with a minimal number of them [70].

General purpose grammar evolution was the opposite attempt to cover all possible use cases for grammar evolution, recovery, convergence, adaptation,

[4]NB: we limit our overview on *programmable* grammar evolution, without considering adjacent initiatives such as incremental grammar refinement [58, 59, 60, 61].

etc. The resulting grammar manipulation language is called XBGF, for Transformations of BNF-like Grammar Format [50, §7], and was also used extensively throughout the Software Language Processing Suite, for many tasks in many projects [22], some of which involved operating on grammars of industrial size (Java, C#, C++, etc).

Bidirectional grammar evolution operator suite ΞBGF is a bidirectional variation of XBGF, that has shown its usefulness for metalinguistic evolution and derivation of transformation steps [27].

3.3. Metalinguistic evolution

Even though many metagrammarware tools and language documents claim to use EBNF [47], it does not mean that they agree on a metalanguage. EBNF has grown to become a family of textual notations for defining context-free grammars with possible extensions. We speak of "metalinguistic evolution" as a specific case of grammar evolution, when the language defined by the grammar, does not change, but the metalanguage in which it was written, does. This is a known problem at least since [37] (translation between SBNF and SDF) and [41] (translation from EBNF used by ISO standards to LLL), which received a relatively straightforward solution with the introduction of the notation specification [25].

Given a grammar $G_N(L)$ written in a metalanguage N, we can express metalinguistic evolution as a metalanguage transformation σ that transforms specification $S(N)$ into a specification $S(N')$. A new metalanguage N', defined by the transformed specification; its own grammar $G(N')$ and an updated grammar $G_{N'}(L)$ can all be automatically derived from this σ, as shown in [27].

3.4. Generating browsable artefacts

In [37], it was claimed to be possible with the Box functionality of the ASF+SDF Meta-Environment [71]. In [70], the author went to great length to inject the changes in the grammar back to its documentation. As a part of research on language documentation in [26], a case study was completed that involved extracting all the available information from a language manual for the purpose of regenerating it after necessary manipulations.

In general, we have expected and achieved the following qualities in order to claim useful browsability:

Metasyntax highlighting means that different entities are displayed differently — for example, terminal and nonterminal symbols use different colours.

Interactiveness means that if an element can be observed, it can also be interacted with, if such interaction makes sense — for example, one can get to a definition of a nonterminal from its occurrence.

Metrics as simple as top and bottom nonterminals proposed by [37, 72] or as complex as grammatical level depth and others listed in [32, 33], aid comprehension of a grammar just as much as traditional software metrics help estimating code quality and detecting code smells.

Full automation is expected to take care of the abovementioned qualities without asking the end user to set up any complex infrastructure or configuration.

All four objectives can be achieved in at least two different ways: hypertext and IDEs. Hypertextual rendering of grammars is a pretty straightforward mapping from the language of their internal representation to XHTML supported by a predefined CSS. There is hardly anything scientifically challenging in this mapping, but we have of course engineered and automated it within the SLPS to support the Grammar Zoo [22]. The other way to achieve browsability is relying on a sufficiently advanced IDE framework such as Rascal language workbench [73]. Using the framework functionality, one can quickly prototype a powerful grammarware engineering environment, which would be highly domain-specific and yet extensible and programmable.

4. Creating a grammar repository

Having set up the objectives in §2 and revisited previously addressed challenges that enable the creation of the Grammar Zoo in §3, we move on to explain the repository creation process. In particular, we will define the model of the metadata that is supplied for each grammar in the repository (§4.1), explain different kinds of sources for grammar extraction and the levels of grammars extracted from them (§4.3), list the currently available automated extractors (§4.5) and exporters (§4.6) and explain the recovery process (§4.8). Several intervening sections will present illustrative examples to demonstrate our approach.

4.1. The model of metadata

We present the data model behind the Grammar Zoo frontend on Figure 1. (An intuitively readable dialect of EBNF is used with `?` denoting "zero or one", `*` denoting "zero or more", `+` denoting "one or more" and using `|` for choice; `x:y` means referencing a nonterminal `y` with the name `x`). Details follow:

`repository` — the root element;

`language` — one language or language family (e.g., "Java", "C#", "(E)BNF");

`version` — one version of a language with all its grammars and metadata (e.g., "Java 5");

`grammar` — one grammar with all its metadata (e.g., "a level 2 grammar of Java 5 extracted from the third edition of the Java Language Specification with Grammar Hunter and certain grammar transformation scripts");

`name` — a string naming something; important for identification of a language, a language version or a grammar within the repository, although does not have to be unique: as Table 3 will show, it is possible and useful to have several grammars of the same intended language in a repository;

```
// A repository consists of many languages or language families
repository ::= language+;
// Each language (e.g., Java) can have several versions
language ::= name version+;
// Each version (e.g., Java 5) can have several grammars of it
version ::= name grammar+;
// Each grammar denotes its name and status and lists sources, extractors,
// renderers and possibly related tools, publications and other resources
grammar ::= name status source:resource extraction:(link+)
            (render:link)+ tool* resource*;
// Each (re)source refers to its origin and may state title, subtitle,
// file names, publication venue, edition number, etc.
resource ::= origin title? subtitle? file* venue? edition? date
             specific? link+;
// Resource origin is a list of authors, a reference number of a standard
// or an organisation name that produced it.
origin ::= author+ | standard | organisation;
// Each link can be named and is in general a URI with shorthand notations.
link ::= name? (uri|doi|xbgf|wiki|readme|slps|...);
// Each related grammarware tool has a name (e.g., parser),
// a technology it is based on (e.g., ANTLR) and a list of links.
tool ::= name technology link+;
```

Figure 1: The data model of Grammar Zoo displayed as an annotated grammar.

`status` — the quality status of the recovered grammar; discussed in detail in §4.3;

`source` — the primary resource used for grammar extraction (several resources are allowed if the grammar fragments needed to be collected); discussed in detail in §4.3;

`resource` — a publication related to the grammar or using it extensively, a website dedicated to its recovery process, etc;

`extraction` — usually a grammar refers to at least one set of extraction tools, but there can possibly be more (recovery toolkits, disambiguation tools, grammar correction scripts, etc); discussed in detail in §4.5 for extractors and §4.8 for recovery tools;

`render` — a format or notation in which this grammar should be automatically rendered (pretty-printed); as of now, the Grammar Zoo provides HTML (a browsable variant), BGF (the internal easy-to-parse XML representation), EBNFs (the one used in a range of ISO standards and the one used inside the DMS Software Reengineering Toolkit [74]), SDF (syntax notation used by the Meta-Environment) [49, 51] and Rascal metaprogramming language [73]; as discussed in detail in §4.6, this is also a future extension point, since complex scenarios of rendering a grammar may involve grammar mutations and matching;

`tool` — a grammarware tool coupled with the grammar or related to it; for example, if this grammar was extracted from a parser specification, this will link to the executable parser; it can be a validator for data models, a refactoring tool, a migration tool, etc.;

`origin` — any entity responsible for creating a resource;

`title` — the title of a resource;

`subtitle` — a string necessary to identify the resource, but not a part of the title itself;

`file` — the name of the particular file used for extraction: especially crucial for multipart sources such as modularised grammars;

`venue` — the name of a conference, a workshop or a journal;

`date` — when the resource was created, usually the year of publication: timestamps can be important (among other things) to identify the version of the intended language, if it cannot be derived through other means;

`edition` — language specifications often have editions or versions;

`specific` — specific coordinates for the extraction source within the generally identified resource (i.e., a chapter of a book, page numbers, an important note);

`link` — essentially a customary named URI;

`author` — the name of each of the authors stored as a string: possibly can be matched with DBLP, ORCID or similar framework and appropriately linked;

`standard` — a reference number for a language standard (e.g., for ISO standard of EBNF it is "ISO/IEC 14977:1996(E)" [75]);

`organisation` — the name of a company or a standardisation body responsible for the creation of the resource (and possibly the holder of the copyright);

`uri` — a uniform resource identifier, a link to a webpage;

`doi` — a digital object identifier, the easiest way to refer to most academic publications: can be easily resolved to an official publisher's page with `http://dx.doi.org`;

`xbgf` — a shorthand notation for referencing grammar transformation scripts, which are then also automatically rendered as hypertext; such link does not need a name since it can be inherited from the transformation script;

wiki — a shorthand notation for referencing an page on the SLPS wiki at http://github.com/grammarware/slps/wiki, usually such a page is automatically generated and contains links to multiple files in the repository that comprise one tool;

readme — a shorthand notation for referencing a verbose README.txt file supplied with more details described in natural language;

slps — a shorthand notation for referencing other files in the SLPS repository on GitHub [22];

The specifications of the Grammar Zoo conforming to the data model described above, can be found as zoo.xml files in SLPS in directories with each extracted grammar. The root zoo.xml document[5] in the GitHub project slps.github.com collects links to all those. The table with a brief overview of the current contents of the Grammar Zoo will be presented later as Table 3.

4.2. Illustration: notation-parametric recovery of the C++ grammar

Consider Figure 2 as an example of one of the 569 entities comprising the Grammar Zoo. In this example, dots represent omissions. Additional resources refer to [24] and other related publications. The (post-)extraction grammar transformation scripts recover.xbgf and correct.xbgf are inherited from the Java grammar convergence case study in [24] and pretty-printed in hypertext as a part of the Grammar Zoo. Both represent grammar evolution steps that lift this Java grammar from the L1 status (freshly extracted) to L2 (checked and connected) — we will list all supported statuses later in §4.3. The main difference between them is that correct.xbgf fixes the mistakes made by the creators of the grammar source, while recover.xbgf fixes the mistakes that were not automatically handled by recovery heuristics. For instance, the nonterminal FormalParameter was erroneously left undefined in the specification — hence, correct.xbgf contains a call to a **define** operator with appropriate arguments. On the other hand, on several occasions curly bracket terminal symbols were mistaken by the extractor to be repetition metasymbols (i.e., in the JLS notation "{x}" means "zero or more x") — hence, these mistakes were fixed in the recover.xbgf script.

4.3. Grammar sources

So far, we have encountered the following kinds of grammar sources:

Language standard is a language document that was developed under supervision or received acknowledgment from a standardisation body (ANSI, ECMA, IEEE-SA, ISO, IEC, ITU, IETF, OASIS, OMG, WSA, W3C, etc). There are two additional factors that play important roles:

[5] http://github.com/slps/slps.github.com/blob/master/_dev/zoo.xml

```
<grammar>
    <name>JLS3 Implementable</name>
    <status>L2</status>
    <source>
        <author>James Gosling</author>
        <author>Bill Joy</author>
        <author>Guy Steele</author>
        <author>Gilad Bracha</author>
        <title>Java Language Specification</title>
        <edition>3</edition>
        <date>2004</date>
        <specific>Ch.18: Syntax, pages 585-596</specific>
        <readme/>
        <link>
            <uri>http://java.sun.com/.../syntax.html</uri>
            <name>HTML</name>
        </link>
        <link>
            <uri>http://java.sun.com/.../download/langspec-3.0.pdf</uri>
            <name>PDF</name>
        </link>
    </source>
    <extraction>
        <wiki>HTML2BGF</wiki>
        <xbgf>recover</xbgf>
        <xbgf>correct</xbgf>
    </extraction>
    <resource>
        ...
    </resource>
</grammar>
```

Figure 2: A typical entry in the Grammar Zoo, corresponding to the "more implementable" grammar of Java 5 extracted from the Java Language Specification [44] for the purpose of reverse engineering its relationships with other Java grammars [24].

Centralised or distributed? Grammar knowledge can be concentrated in an appendix or a specific section of the language standards, but it can also be distributed all over the document (e.g., when it is used for explaining language constructs one at a time). In the second case, the extraction process is prone to missing grammar fragments due to incorrect markings and other reasons.

Open or closed? When a standardisation body commits to public disclosure of a language standard, it goes through a certain process which usually comprises sanitising the contents at least to some extent: clean up, mark up, linking and similar activities improve the quality of the grammar source. If the standard is a close publication, it can be unavailable for inspection for a broad audience (require payment or special subscription), and there is an additional step of reentering the data from its printed copy back into a computer. Both manual retyping and automated text recognition processes are error-prone.

Industrial specification is in many aspects the same as a language standard, but it is developed inside a commercial company (Ericsson, Google, IBM, Microsoft, Oracle, etc). The same additional factors from language standards apply, with conditions for disclosure usually being even more strict.

Browsable documentation is often found in many corners of the Internet. Many people spend their own time on extracting grammar knowledge from the artefacts they were able to obtain, sanitising it and reformatting the resulting grammar as hypertext. Some of such endeavours that we mentioned before, are well-documented and linked to a published scientific report [36], others contain conformance and validity claims that require thorough verification.

Parser specification is an executable grammar that contains many annotations that often take it beyond the context-free class. Grammars specified in ANTLR [52, 56], Bison [76], JavaCC [77], Kiama [78], Rascal [73, 79], SDF [49, 51, 80], TXL [54, 57], YACC [81] and many other metagrammarware frameworks can be located in their corresponding repositories or just anywhere close to end users of these products. Such specifications can be stripped from excessive information and extracted in the form expected by the representation central for the repository.

Metamodel is a grammar in a broad sense used in the modelware technological space. Just like a parser specification, it can contain details that transcend structural definitions: constraints, certain relations, etc. However, that information can be abstracted from, and the grammars can be extracted. AtlanMod already started an initiative of accumulating metamodels from model-driven open source projects: we have referenced the EMF XMI part of it as [3], but the same repository also contain metamodels in KM3 [82], MSchema [83], Clojure [84], SBVR [85], UML 2.1 [86], GraphML [87], OWL [88], MOF [89], etc.

Wiki pages can also be a source for grammar extraction, if the grammar was developed by a community. So far we have encountered only one such initiative, with several reverse engineered grammars of MediaWiki, and reported it in detail in [45].

Scientific papers often contain small grammars or grammar fragments. We have not yet attempted a big scale mining process of recovering all possible grammar fragments published in a certain set of venues. However, at least once [27] it was useful to compare the grammar published in a workshop paper [65] with its updated version published electronically in a programmer's manual [66].

As a starting point for identifying a status of a grammar, we have decided to adopt the notion of *grammar levels*[6] from [31] and extend it with a zero level:

- A *level 0 grammar* (L0) is a text that is assumed to directly represent grammatical knowledge (usually written in an (E)BNF-like notation). It can contain unreadable symbols, incorrect indentation, or even be present in a form of an image or a manuscript.
- A *level 1 grammar* (L1) is a raw grammar that has just been extracted from a language definition, corrected of all typographical, text recognition and similar errors and converted into a context-free grammar (or, more broadly speaking, to a Boolean grammar [91]).
- A *level 2 grammar* (L2) is a maximally connected level 1 grammar: it does not contain unwanted top sorts (nonterminals that are defined but never used) and bottom sorts (nonterminals that are used but not defined). These two quality indicators were proposed in [72, 37] and discussed in more detail in [64].
- A *level 3 grammar* (L3) is a level 2 grammar complemented with a lexical part: on the second level only those top sorts remained that are either true root sorts or lexical. Some language documents have a special section dedicated to a lexical grammar, while others do not, and it must be created manually. Typically, a parser or at least a recogniser can be automatically generated from such a grammar.
- A *level 4 grammar* (L4) is a level 3 grammar that has been tested on a scale of considerable volume of code coming from different sources, companies, countries and coding traditions — in languages such as C or COBOL a codebase used for testing can contain millions of lines of code.
- A *level 5 grammar* (L5) is a grammar that bidirectionally corresponds to a real piece of grammarware such as a compiler or a code analysis or transformation tool. Level 5 grammars are the ones most close to the ideal since they describe everything that can be accepted by a compiler and nothing more.

We do acknowledge apparent imperfection of this model. For example, a lot of research attention has been directed recently to disambiguation of grammars [92, 93] and to grammar-based testing [94, 18] — while these levels do not account for the presence or the lack of ambiguities in the grammar or any of its test coverage criteria. For the Grammar Zoo, we intentionally adopt the hierarchy as is, and leave its extension to future work.

[6] These "grammar levels" are essentially CMM-like levels applied to grammars, unrelated to well-known "grammatical levels" using for a range of grammar metrics [90, 32].

Collection	Contents	Format
Atlantic Metamodel Zoo [3]	303 abstract metamodels	Ecore [95]
TXL World: Grammars [57]	21 concrete PL grammars	TXL [54]
RelaxNG Schemas [96]	90 schemata	RELAX NG [97]
SDF Library [80]	14 concrete syntax specifications	SDF [51]
Rascal Language Library [79]	13 syntax definitions	Rascal grammar[a]
	8 algebraic data types	Rascal ADT[b]
	1 grammar found in comments	EBNF dialect
SLPS [22], prior experiments	12 grammars of FL [23, 98]	various
	33 grammars of TESCOL [18]	ANTLR3 [99]
	35 versions of the LDF grammar [26]	BGF [23]
	23 other assorted grammars	various
ISO/IEC JTC1/SC22 [100]	13 grammars from language standards	EBNF dialects
Ecma International [101]	4 grammars from language standards	EBNF dialect

[a] http://tutor.rascal-mpl.org/Rascal/Declarations/SyntaxDefinition/SyntaxDefinition.html
[b] http://tutor.rascal-mpl.org/Rascal/Declarations/AlgebraicDataType/AlgebraicDataType.html

Table 2: Existing collections of grammars in a broad sense that have been incorporated, to the best of our ability, to the Grammar Zoo. For each notation, a crucial component is a grammar extractor from that particular format — §4.5 will elaborate on them.

4.4. Illustration: inheriting grammar knowledge

Apart from grammar sources occasionally found in various places, we have systematically inherited and extracted grammars from the existing collections, listed in Table 2.

Atlantic Metamodel Zoo is web-based, so obtaining an L1 grammar is easy with semi-standard widespread tools like `wget` or `curl`. All the metadata about the grammars (i.e., authorship, creation dates, etc) was recovered by parsing the front webpage. Most of the metamodels yielded L2 grammars, and those that did not, were deliberately designed to consist of several non-connected components. *TXL World* is similarly accessed through the internet, but it deploys grammars in tarballs of varying structure, so the process involved less automation: each source must have been explored individually to locate the files relevant for grammar extraction. TXL grammars typically are at L3 or L4, and can be considered L5 if shipped together with a corresponding tool. *RelaxNG Schemas* is a collection of links: some of them became dysfunctional over time and needed to be located via The Wayback Machine[7], otherwise processing it was similar to TXL grammars, with the obvious exception being that data schemata are L2 grammars. *Meta-Environment* (that includes a grammar library in its distribution), *Rascal* (ibid.) and *SLPS* are open source projects with their repositories also freely exposed to the public — this allows us not only to extract grammars from them in a very robust way, but also collect and analyse the whole history of commits concerning each of them (this was successfully performed as an experiment for some SLPS grammars). *ISO* and *ECMA* standards contain grammars in textual form, written in a variety of informally

[7] The Wayback Machine, http://archive.org/web/web.php.

described (E)BNF dialects (far from ISO EBNF), and are often compromised by typesetting artefacts, misspellings, etc. Still, advanced notation-parametric grammar recovery [19] combined with some basic cleanup actions specific for each grammar, was enough to extract a number of such L1 grammars.

4.5. Grammar extractors

As of now, we have the following grammar extractors available in the Software Language Processing Suite:

ADT to BGF. Without loss of generality, one can assume that abstract data types in Rascal are conceptually the same as in Haskell or any other advanced functional language. In this extractor, types are mapped to nonterminals and constructors are mapped to alternative right hand sides.

Example — Bert Lisser, Dot, `lang::dot::Dot`, 2012

```
public data DotGraph = graph(Id id, Stms stmts)
                     | digraph(Id id, Stms stmts);
```

```
DotGraph ::= [graph]::(id:Id stmts:Stms)
          |  [digraph]::(id:Id stmts:Stms);
```

(All types are treated as nonterminals, constructors as labels and arguments as expression selectors).

ANTLR to BGF. In order to be able to extract grammars from ANTLR parser definitions, we reused the standard ANTLR grammar for ANTLR grammars by attaching appropriate semantic actions to it. The semantic actions were programmed for using XML API to serialise the parse tree as a BGF grammar and abstract from the parsed semantic actions.

Example — Oliver Kellogg, Ada, `ada.g`, 2003

```
use_clause : u:USE^
        ( TYPE! subtype_mark ( COMMA! subtype_mark )*
            { Set(#u, USE_TYPE_CLAUSE); }
        | c_name_list { Set(#u, USE_CLAUSE); }
        )
    SEMI!
    ;
```

```
use_clause ::= u:USE ((TYPE subtype_mark (COMMA subtype_mark)*)
   | c_name_list) SEMI;
```

(Both context-free and lexical nonterminals become nonterminals, all AST-building annotations are abstracted from, as are semantic predicates and actions).

DCG to BGF. Definite clause grammars are a way of specifying a parser in Prolog [53]. Their clauses are mapped straightforwardly to production rules by an extractor written also in Prolog.

Example — Ralf Lämmel, FL, `Parser.pro`, 2008

```
function((N,Ns,E)) -->
       name(N),
       +(name,Ns),
       @("="),
       expr(E),
       +(newline).
```

```
function ::= name name+ "=" expr newline+;
```

(Left hand sides of definite clauses become left hand sides of a grammar, all predicates on the right hand side that are not recognised as metasyntax, are treated as nonterminals, except for "`@`", "reserved" and "keyword" that are treated as specifying terminal symbols. All information on variables is abstracted from).

Ecore to BGF. Since Ecore models are by default serialised as XMI, we only needed to express the mapping between Ecore and BGF, which was done in XSLT.

Example — Hugo Bruneliére, Cobol, `COBOL.ecore`, 2005

```
<eClassifiers xsi:type="ecore:EClass" name="COBOL88Element">
  <eStructuralFeatures xsi:type="ecore:EAttribute" name="name"
  ordered="false" unique="false" lowerBound="1" eType="/1/String"/>
  <eStructuralFeatures xsi:type="ecore:EReference" name="has"
  ordered="false" lowerBound="1" upperBound="-1"
  eType="/0/COBOL88ElementValue" containment="true"/>
</eClassifiers>
```

```
COBOL88Element ::= name:String has:COBOL88ElementValue+;
```

(The extractor works with the XMI serialisation of an Ecore model. All non-abstract `eClassifiers` tags of `EClass` type become nonterminals, with each of `eStructuralFeatures` referring to a named element in the sequence on the right hand side in the grammar production rule).

Java to BGF. The object model of a Java program is extracted from a Java source by the use of reflection. Classes are treated as nonterminals, and their visible interfaces (public members and getters/setters) serve as the right hand side. This mapping helped to trivially converge the structure defined by the Java source generated by a data binding framework (JAXB) with the structure defined by the original schema (XSD) in [23].

Example	Ralf Lämmel, FL, `types/Binary.java`, 2008

```
public class Binary extends Expr {
        public Ops ops;
        public Expr left, right;
        public Binary(Ops o, Expr left, Expr right) {
                this.ops = o;
                this.left = left;
                this.right = right;
        }
        public void accept(Visitor v) {
                v.visit(this);
        }
}
```

```
Expr ::= Binary;
Binary ::= ops:Ops left:Expr right:Expr;
```

(Each class is mapped to a nonterminal. Inheritance is treated as a chain production rule. Every public element of a class becomes a sequentially composed element on the right hand side of the grammar production rule, with types becoming nonterminals and names becoming selectors).

LDF to BGF. Since we assume that any language document does contain grammar knowledge explicitly, i.e., in BGF, we use a special extractor to take out the BGF bits and compose a grammar from them. In the past, the LDF to BGF extractor was mostly used for testing purposes. We leave it without an example since extraction from LDF is basically selective copying.

LLL to BGF. The first extractor from LLL was developed as a means of importing grammars manipulated by GDK [65]. Later it has been retired in favour of an EBNF Dialect Definition (EDD) of LLL that serves as a parameter for Grammar Hunter (see below).

Example	Vadim Zaytsev, C#, `Final.lll` extracted from ECMA-344, 2005

```
goto-statement
  : "goto" lex-csharp/identifier ";"
  | "goto" "case" expression ";"
  | "goto" "default" ";"
  ;
```

```
goto-statement ::= "goto" lex-csharp/identifier ";"
                |  "goto" "case" expression ";"
                |  "goto" "default" ";";
```

(The notations are very close: BGF covers strictly more metasyntactic features than LLL).

Python to BGF. A Python library called PyParsing allows to define a PEG inside Python code. This extractor, written in Rascal, relies on the structure expected by PyParsing, in order to recover grammar knowledge from a Python program.

Example — Ruwen Hahn, FL, `parser.py`, 2008

```
ifThenElse = (
        _IF + expr + _THEN + expr + _ELSE + expr
        ).setParseAction(lambda tok: t.IfThenElse(*tok))
```

```
ifThenElse ::= _IF expr _THEN expr _ELSE expr;
```

(The extractor abstracts from all parse actions of the PyParsing library and treats all components as nonterminals, while also recognising combinators for sequential composition ("^" and "+"), optionality (`Optional`), negation (`NotAny`), Kleene star (`ZeroOrMore`), etc).

Rascal grammar to BGF. By reusing Rascal grammar for Rascal and internal interfaces for accessing it, this extractor delivers a platonic grammar extracted from a Rascal [73] concrete syntax definition (which is essentially an annotated parser specification).

Example — Jurgen Vinju, Pico, `lang::pico::syntax::Main`, 2011–2012

```
syntax Expression
  = id: Id name
  | strcon: String string
  | natcon: Natural natcon
  | bracket "(" Expression e ")"
  > left concat: Expression lhs "||" Expression rhs
  > left ( add: Expression lhs "+" Expression rhs
         | min: Expression lhs "-" Expression rhs
         )
  ;
```

```
Expression ::= [id]::(name:Id)
            |  [strcon]::(string:String)
            |  [natcon]::(natcon:Natural)
            |  "(" e:Expression ")"
            |  [concat]::(lhs:Expression "||" rhs:Expression)
            |  [add]::(lhs:Expression "+" rhs:Expression)
            |  [min]::(lhs:Expression "-" rhs:Expression);
```

(The extractor abstracts from associativity rules and priority specifications and treats names of alternatives as labels. Any annotations are also neglected during extraction).

RelaxNG schema to BGF. RELAX NG is a schema language for XML, alternative to a much more popular XML Schema [97]. It is in general believed to be simpler and thus more understandable and making the users less prone to mistakes, but all commonly used features are present in both languages, making the choice between them usually platform-dependent or library-dependent. The extractor was developed incrementally by covering the functionality found in the schemata referenced at the RELAX NG home page [96]; it was not a scientifically challenging process.

Example Norman Walsh, DocBook, `dbhier.rng`, 2002

```
<define name="tocfront">
    <element name="tocfront">
      <ref name="tocfront.attlist"/>
      <zeroOrMore>
        <ref name="para.char.mix"/>
      </zeroOrMore>
    </element>
  </define>
```

```
tocfront ::= [tocfront]::(tocfront.attlist para.char.mix*);
```

(All `define` clauses with a nested `element` are mapped to labelled production rules. References to other definitions are treated as nonterminals, repetition kinds are also mapped straightforwardly. The `interleave` construct is abstracted from and mapped to a sequence, which is still a reasonably good representation in the case of an abstract grammar. Mixed content is mapped to a Kleene star over the choice of the actual content and a raw string).

SDF to BGF. We encoded the necessary traversal functions for crawling the parse trees of SDF grammars and producing BGF and reused the SDF module and the XML module from the standard package of the Meta-Environment. Separate command line tools are used to make a parse table, to compile ASF formulæ, to parse the input grammar, to rewrite the parse tree and to serialise the transformed parse tree into a file. They are bundled together and wrapped in a black box extraction script.

Example Jurgen Vinju, Taeke Kooiker, Mark van den Brand, C, `ansi-c/syntax/Declarations.sdf`, 2006–2008

```
context-free syntax
Specifier+ {InitDeclarator ","}+ ";" -> Declaration
Specifier+ ";"                       -> Declaration {avoid}
```

```
Declaration ::= Specifier+ {InitDeclarator ","}+ ";";
Declaration ::= Specifier+ ";";
```

(The notations are very similar, except the right hand side and the left hand sides are flipped to a more common position, and priority specifications are neglected. Constructor names, if present, becomes labels for production rules).

W3C Specification to BGF. As a part of the initiative to create a unified data model for language documentation, a case study was completed to map the W3C specification of XPath to it [26]. This extractor looks for all `<scrap>` elements inside a W3C standard specification and maps its `<prod>` elements to production rules.

Example — W3C, XPath, `REC-xpath-19991116.xml`, 1999

```
<prod id="NT-LocationPath">
    <lhs>LocationPath</lhs>
    <rhs>
        <nt def="NT-RelativeLocationPath">RelativeLocationPath</nt>
    </rhs>
    <rhs>
        |
        <nt def="NT-AbsoluteLocationPath">AbsoluteLocationPath</nt>
    </rhs>
</prod>
```

```
LocationPath ::= [NT-LocationPath]::(RelativeLocationPath
             | AbsoluteLocationPath);
```

(Production rules are formed from `<prod>` tags, with their IDs becoming production labels and each reference to a nonterminal being mapped to a nonterminal).

TXL to BGF. We reused the TXL grammar for TXL grammars and made use of the TXL engine's option of returning the parse tree in an XML form. The mapping between TXL XML and our XML (i.e., BGF) was straightforwardly encoded in XSLT.

Example — William Waite, James Cordy, Fortran, `fortran.grm`, 2009

```
define BlockDataStmt
  [LblDef] 'blockdata [opt BlockDataName] [EOS]
end define
```

```
BlockDataStmt ::= LblDef "blockdata" BlockDataName? EOS;
```

(Each `define` and `redefine` clause is mapped to a production rule. All combinators of the latest version of TXL (`opt`, `repeat`, `list`, etc) are supported. All `rule` clauses are ignored).

XML Schema to BGF. Not all elements of the XML Schema can be mapped to grammar concepts efficiently, but the most used ones easily find their counterparts. The mapping is thus partial and bidirectional at best (e.g., XML elements and XSD complex types are both mapped to nonterminal symbols).

Example — Vadim Zaytsev, LDF, `ldf.xsd`, 2010

```
<xsd:complexType name="named-link">
    <xsd:sequence>
        <xsd:element name="title" type="xsd:string"/>
        <xsd:choice minOccurs="0">
            <xsd:element name="version" type="xsd:string"/>
            <xsd:element name="edition" type="xsd:string"/>
        </xsd:choice>
        <xsd:element name="uri" type="xsd:anyURI" minOccurs="0"/>
    </xsd:sequence>
</xsd:complexType>
```

```
named-link ::= title:str (version:str | edition:str)? uri:str?;
```

(All elements, groups, complex and simple types become nonterminals. Combinations of `minOccurs` and `maxOccurs` constraints are mapped to the closest possible variant available in BGF: "`+`", "`*`" or "`?`").

4.6. Grammar renderers

Technically, a grammar renderer (pretty-printer, exporter) is a part of a bidirectional transformation [102, 103] between the source grammar specified in a notation of a particular framework or methodology, and the platonic piece of grammatical knowledge stored in the repository. When the grammar is recovered, corrected and otherwise changed, the two become desynchronised and can be brought back in sync by a renderer. A more simplistic view would be to assume that the source grammar is lost or empty, and that the synchronisation process only entails generating the corresponding definition entirely. To illustrate this, suppose that we have a syntax definition D in SDF [49]: it contains a context-free part, a lexical part, possibly variable definitions, priority descriptions, AST constructor annotations and similar details, which are together enough to parse textual data unambiguously to form a tree correctly representing its structure in the described language. A Grammar Zoo entry G extracted from that source, would have less information, because it has undergone through the "extraction through abstraction" process [23] and lost its idiosyncratic details. Still, if something changes — for instance, a production rule is removed from the extracted grammar G, we can automatically detect which rule needs to be removed from D if we wish them synchronised. If we represent the bidirectional transformation classically as a relation and two update functions, then the pair of D and G will be an element of this relation and two functions would be the extractor and the renderer.

For the sake of simplicity we adopt the simplistic view and consider that a rendered grammar is generated from scratch from a repository entity and does not need to matched with its older version and updated. SLPS contains such simple renderers for exporting grammars in a readable EBNF dialect like the one displayed in this paper; in the BNF dialect used in the DMS Toolkit [74]; in Graphviz dot format for visualisation purposes; in Rascal as a syntax definition [73]; in SDF [49]; in TXL [54]; in LaTeX for publishing purposes, etc. Conceptually each one of those is a reverse of an extractor — that is, a pretty-printer. We will only consider hypertext rendering in detail.

Browsable grammars are not a new idea: in particular, they have been motivated in [31] and [104], and even the bare observation of the abundance of such grammars being displayed all over the internet, with metasyntax highlighting and hyperlinked nonterminals, demonstrates practical need for them.

The hypertext rendering of a grammar in the Grammar Zoo, besides the main part, contains the following additional computed information:

- Number of production rules — the actual number of different definitions for nonterminals physically present in the grammar;
- Number of top alternatives — the PROD metric from [32], calculated as the number of top level alternatives in order to account for "horizontal" definitions (i.e., "`X ::= Y | Z;`" instead of "`X ::= Y; X ::= Y;`");
- Number of defined nonterminal symbols — the number of unique left hand sides of all production rules;

- Root nonterminal symbols — nonterminals explicitly marked as starting symbols of a grammar;
- Other top nonterminal symbols — defined nonterminals not used anywhere in a grammar and not marked as roots; in [31] it is suggested that top nonterminals should be examined during grammar recovery and either elevated to the root status or removed from the grammar;
- Bottom nonterminal symbols — nonterminals forming the difference between the abovementioned number of defined nonterminal symbols and the VAR metric from [32]; undefined nonterminals encountered exclusively within the right hand sides of production rules;
- Number of used terminal symbols — the TERM metric from [32].

The question of how to render a grammar perfectly, is an open one, but these properties have been quoted as helpful for grammar comprehension: they help a human engineer to estimate the grammar's complexity and perform some superficial analyses like examination and elimination of multiple top nonterminals.

4.7. Illustration: convergence of JLS grammars

The method of grammar convergence is used to reverse engineer true relationships between grammars by transforming them toward equality and examining the properties of these transformation steps (in particular, preservation of the assumed semantics). For example, if the only changes between two grammars concern renaming nonterminals, we conclude that the grammars are equivalent in the sense of generating or accepting the same string language, but not equivalent in the sense of XML document types. When the method of grammar convergence was first proposed in [23], it was demonstrated on a small case study with six grammars of such a small size that all of them would fit on one page. Obviously, the next step in demonstrating the viability of the method was to perform a full scale case study with industrial size grammars of a mainstream language: Java was chosen as one of the languages having three published editions of the official language specification, each containing two formally unrelated alternative grammars for the same language version.

In the Java convergence paper [24], one quarter of it, 11 full pages out of 42 (not counting the bibliography), is dedicated to the extraction and recovery process. However, the recovery of the six grammars from three editions of the Java Language Specification is not among the main contributions of the paper, so the amount of attention it received is due to the sheer complexity of the process, which entailed:

- tag elimination — source L0 grammars were typed in manually in ill-formed HTML;
- indentation processing — the input notation relied on indentation to separate top level choices in production rules;

- robust parsing — in order to deal with incorrect markup and unexpected artefacts;
- matching parentheses — to disambiguate brackets-as-terminals from brackets-as-metasymbols;
- adjusting symbol roles — incorrect markup and improper context could lead to metasymbols being treated as terminal symbols or vice versa, or terminals as nonterminals;
- composing sibling symbols and decomposing compound symbols — to address the issue of erroneous markup being interjected in the middle of a token;
- removing duplicates — also those with slightly different typesetting.

If anyone is to replicate the case study and, for instance, to converge several C# grammars taken from different sources (ECMA standards, ISO standards and Microsoft language specifications), the whole effort would need to be reinvested both in the engineering side of the case study (the actual programming of an appropriate extractor) and in the description of it (even if only for the sake of honest report). By saving this kind of information together with the extracted grammar in a publicly accessible place, we save the space in any future replication and allow such papers to fully focus on their main contributions.

4.8. Grammar recovery and evolution

All of the tools listed so far in §4.5, did not go beyond simple extraction of a level 1 grammar: that is, in the presence of an error in the grammar source used for extraction, they give up after reporting it to the user, which then has to go back to the source and figure out a way to solve it. Unlike them, the tools listed below are capable to identify and even resolve some of the commonly encountered issues.

BNF to LLL. After we have noticed that many ISO standards of programming languages (C, C++, C#) share the same metalanguage, this was the first tool to be developed. It normalised some lexical singularities and translated the EBNF dialect used by ISO grammar developers to LLL used in GDK [65] (for which we have another extractor ready). It should be noted here that the EBNF used in ISO standards is not the same as ISO EBNF defined by [75].

PDF to BGF. A grammar copy-pasted from a PDF of an ISO standard of C, C++, C# or any other that uses the same metalanguage, can be extracted with some tolerance regarding lexical imperfections. This extractor is essentially a composition of BNF to LLL and LLL to BGF.

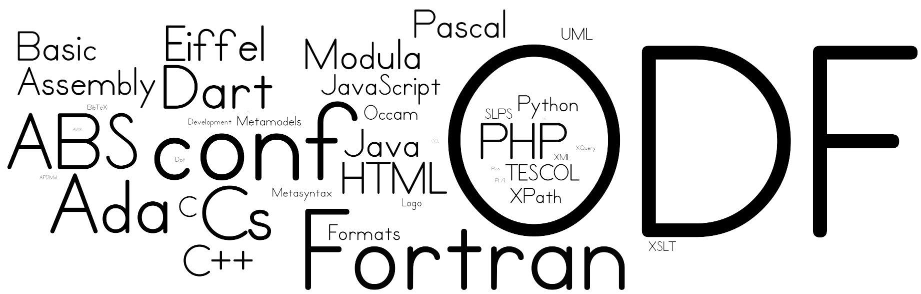

Figure 3: A tag cloud showing language and language group names according to the average sizes of their grammars (made with Wordle). "Cs" denotes C#, "conf" is a collection of metamodels for the domain of conference organisation. "ODF" is Open Document Format.

HTML to BGF. This advanced extractor had to work on a manually and loosely hypertext source. It comprised a set of generalised heuristics in a pattern form that it tried to apply for automated recovery. In the Java Language Specification case study [24] its use was prolific, the extractor fixed 669 errors before we started to program the main body of grammar transformations.

EDD to Rascal. This tool aids semi-automated interactive grammar recovery [19]. It requires a specification of the input metalanguage in a form of EDD, an EBNF Dialect Definition [25], from which it generates a Rascal plugin that enables manipulation of grammars written in the specified EBNF dialect with standard means of Eclipse. This method does not automatically solve any problems, but it helps identifying them and facilitates a grammar engineer in fixing them.

Grammar Hunter. This tool used for notation-parametric grammar recovery [19], requires a specification of the input metalanguage in a form of EDD, an EBNF Dialect Definition [25]. Then it consumes the input text, treating it as a grammar text written in the specified EBNF dialect, applies all appropriate heuristics and delivers a recovered grammar automatically.

XBGF scripts are more or less a standard way of programmable grammar manipulations in SLPS: they can be generated or manually developed, and always can be re-executed or pretty-printed for inspection. The XBGF language is described in detail in [24], [50, §7] and [22, XBGF Manual]. Essentially it allows to use predefined operators for (un)folding nonterminals, factoring choices, renaming symbols, etc.

5. Using the Zoo

Figure 3 visualises the software languages currently represented in the Grammar Zoo. In order to avoid multiple diagrams, we have used a cumulative size

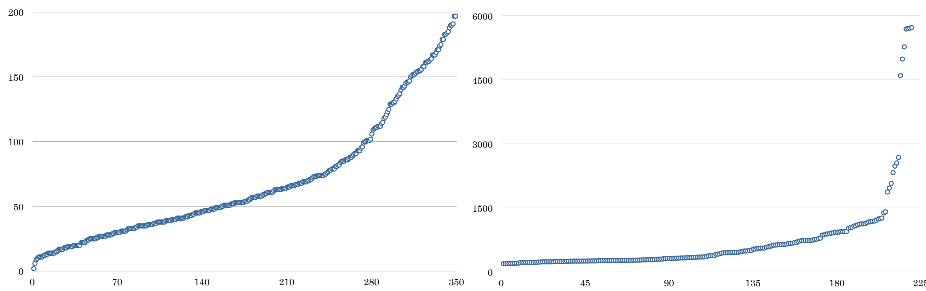

Figure 4: Sizes of individual grammars in Grammar Zoo, in ascending order (up to 200 on the left, from 200 up on the right).

metric conceptually akin to Halstead vocabulary [105] and calculated as the sum of the number of unique nonterminal symbols, the number of unique terminal symbols, the number of unique labels for production rules and top alternatives, and the number of selector names for grammatical subexpressions. As one can see from Figure 4, the Grammar Zoo contains grammars of all sizes, so clustering based on this cumulative metric is unfeasible. The outrageously large outliers on both figures are Open Document Format grammars, all containing around 1000 nonterminals, around 700 terminals and around 2000 production rules.

A more detailed summary of the minimum, average and maximum sizes of grammars per language group, can be found on Figure 5: the smallest grammars of ODF are those of its Manifest Schema and its Digital Signature Schema, so a "language" in this context should be understood as a domain, not as a string language in the formal grammar sense.

Below we list the entire contents of the Grammar Zoo in the form as compact as possible for the list with 569 items. "To L0" in the table denotes the way that we have obtained the grammar source: "git" or "cvs" means getting it from a repository, while "copy-paste" or "tar -xf" assumes some manual work. "To L1" is the extraction process: most of the entries refer to the extractors listed in §4.5. "To L2", if present, describes how the grammar was changed to be considered maximally connected — usually it happens with a grammar transformation script in XBGF, since it has the framework support in Rascal, Prolog and Python, and can be pretty-printed to a readable form to be consumed naturally as a part of the Grammar Zoo itself.

Language	**Number**	**Source**	**To L0**	**To L1**	**To L2**
ABS	1	HATS	private	rscs2bgf	—
Ada	2	ANTLR Grammar List	antlrpp	antlr2bgf	—
Ada	2	ISO/IEC 8652	regexp	hunter	xbgf
Ada	1	TXL Grammar Collection	tar -xf	txl2bgf	—
Ada	1	VU Browsable Grammars	copy-paste	hunter	xbgf
API2MoL	1	API2MoL website	wget	ecore2bgf	—
API2MoL	1	API2MoL website	wget	antlr2bgf	—
Assembly	1	TXL Grammar Collection	tar -xf	txl2bgf	—

AWK	1	AtlantEcore Metamodel Zoo	wget	ecore2bgf	—
Basic	1	TXL Grammar Collection	tar -xf	txl2bgf	—
Basic	1	AtlantEcore Metamodel Zoo	wget	ecore2bgf	—
BibTeX	4	AtlantEcore Metamodel Zoo	wget	ecore2bgf	—
C	3	ISO/IEC 9899	copy-paste	pdf2bgf	xbgf
C	1	SDF Library	git	sdf2bgf	—
C	1	AtlantEcore Metamodel Zoo	wget	ecore2bgf	—
C	1	Rascal Language Library	git	rscs2bgf	—
C	4	TXL Grammar Collection	tar -xf	txl2bgf	—
C++	1	ISO/IEC SC22/WG21 N2723	copy-paste	pdf2bgf	—
C++	1	TXL Grammar Collection	tar -xf	txl2bgf	—
C++	1	AtlantEcore Metamodel Zoo	wget	ecore2bgf	—
C++	1	ISO/IEC 14882:1998(E)	copy-paste	hunter	—
C#	1	VU Browsable Grammars	cvs	hunter	—
C#	2	TXL Grammar Collection	tar -xf	txl2bgf	—
C#	4	Microsoft Lang. Spec.	copy-paste	pdf2bgf	xbgf
C#	4	ECMA-334	copy-paste	pdf2bgf	xbgf
C#	1	AtlantEcore Metamodel Zoo	wget	ecore2bgf	—
C#	2	ISO/IEC 23270	copy-paste	pdf2bgf	—
Conferences	14	AtlantEcore Metamodel Zoo	wget	ecore2bgf	—
Dart	1	Google Code	wget	antlr2bgf	—
Dart	2	Dart Specification	copy-paste	hunter	xbgf
Development	10	AtlantEcore Metamodel Zoo	wget	ecore2bgf	—
Dot	2	AtlantEcore Metamodel Zoo	wget	ecore2bgf	—
Dot	1	Rascal Language Library	git	rscs2bgf	—
Dot	1	Rascal Language Library	git	rscd2bgf	—
Dot	1	Rascal Language Library	git	hunter	—
Eiffel	1	TXL Grammar Collection	tar -xf	txl2bgf	—
Eiffel	1	Gobosoft website	copy-paste	hunter	xbgf
Eiffel	1	ISO/IEC 25436	copy-paste	hunter	xbgf
FL	1	SLPS	git	sdf2bgf	—
FL	1	SLPS	git	xsd2bgf	—
FL	1	SLPS	git	py2bgf	—
FL	1	SLPS	git	dcg2bgf	—
FL	1	SLPS	git	rscd2bgf	—
FL	1	SLPS	git	antlr2bgf	—
FL	2	SLPS	git	ecore2bgf	—
FL	2	SLPS	git	java2bgf	—
FL	1	SLPS	git	rscs2bgf	—
FL	1	SLPS	git	txl2bgf	—
Formats	3	Rascal Language Library	git	rscs2bgf	—
Formats	2	SLPS	git	rscs2bgf	—
Formats	2	Rascal Language Library	git	rscd2bgf	—
Formats	4	Furubayashi website	wget	rng2bgf	—
Formats	1	SLPS	git	rscd2bgf	—
Formats	2	MediaWiki website	wget	hunter	xbgf
Formats	1	GitHub Gist	wget	rng2bgf	—
Formats	24	PNML website	wget	rng2bgf	—
Formats	1	OASIS	wget	rng2bgf	—
Formats	19	AtlantEcore Metamodel Zoo	wget	ecore2bgf	—
Fortran	1	SDF Library	git	sdf2bgf	—
Fortran	1	TXL Grammar Collection	tar -xf	txl2bgf	—
HTML	2	AtlantEcore Metamodel Zoo	wget	ecore2bgf	—
HTML	3	Thai Open Source	wget	rng2bgf	—
HTML	1	TXL Grammar Collection	tar -xf	txl2bgf	—
Java	1	SDF Library	git	sdf2bgf	—

Java	4	ANTLR Grammar List	wget	antlr2bgf	—
Java	5	AtlantEcore Metamodel Zoo	wget	ecore2bgf	—
Java	1	Rascal Language Library	git	rscs2bgf	—
Java	2	Rascal Language Library	git	rscd2bgf	—
Java	2	TXL Grammar Collection	tar -xf	txl2bgf	—
Java	6	Java Language Specification	wget	html2bgf	xbgf
JavaScript	1	TXL Grammar Collection	tar -xf	txl2bgf	—
Logo	1	SDF Library	git	sdf2bgf	—
Metamodels	212	AtlantEcore Metamodel Zoo	wget	ecore2bgf	—
Metasyntax	1	TXL Grammar Collection	copy	txl2bgf	—
Metasyntax	3	AtlantEcore Metamodel Zoo	wget	ecore2bgf	—
Metasyntax	1	OASIS	wget	rng2bgf	—
Metasyntax	1	Tennison website	wget	rng2bgf	—
Metasyntax	8	SDF Library	git	sdf2bgf	—
Metasyntax	2	Grammar Deployment Kit	copy-paste	hunter	—
Metasyntax	3	ISO/IEC 14977	copy-paste	hunter	—
Metasyntax	1	TXL Grammar Collection	tar -xf	txl2bgf	—
Metasyntax	1	SLPS	git	copy	—
Metasyntax	1	Rascal Language Library	git	rscd2bgf	—
Metasyntax	1	SLPS	git	rscs2bgf	—
Metasyntax	5	Rascal Language Library	git	rscs2bgf	—
Modula	1	SDF Library	git	sdf2bgf	—
Modula	1	SRC 052	copy-paste	hunter	xbgf
Modula	1	TXL Grammar Collection	tar -xf	txl2bgf	—
Occam	1	μOCCAM book	copy-paste	hunter	—
Occam	2	AtlantEcore Metamodel Zoo	wget	ecore2bgf	—
OCL	5	AtlantEcore Metamodel Zoo	wget	ecore2bgf	—
ODF	9	OASIS	wget	rng2bgf	—
Pascal	1	AtlantEcore Metamodel Zoo	wget	ecore2bgf	—
Pascal	3	TXL Grammar Collection	tar -xf	txl2bgf	—
PHP	1	TXL Grammar Collection	tar -xf	txl2bgf	—
Pico	1	SDF Library	git	sdf2bgf	—
Pico	1	Rascal Language Library	git	rscs2bgf	—
PL/I	1	AtlantEcore Metamodel Zoo	wget	ecore2bgf	—
Python	2	TXL Grammar Collection	tar -xf	txl2bgf	—
SAF	1	Rascal Language Library	git	rscs2bgf	—
SAF	1	Rascal Language Library	git	rscd2bgf	—
SLPS DSLs	10	SLPS	git	rscd2bgf	—
SLPS DSLs	48	SLPS	git	xsd2bgf	—
TESCOL	33	COMP2010	private	antlr2bgf	—
UML	17	AtlantEcore Metamodel Zoo	wget	ecore2bgf	—
XML	1	AtlantEcore Metamodel Zoo	wget	ecore2bgf	—
XML	1	TXL Grammar Collection	tar -xf	txl2bgf	—
XML	1	Rascal Language Library	git	rscd2bgf	—
XPath	1	W3C XPath Rec	wget	spec2bgf	—
XSLT	1	Thai Open Source website	wget	rng2bgf	—
XSLT	1	AtlantEcore Metamodel Zoo	wget	ecore2bgf	—
XQuery	1	AtlantEcore Metamodel Zoo	wget	ecore2bgf	—

Future possible uses for the Grammar Zoo and any other grammar repository built on the same principles we have proposed in this paper, are diverse and include, but are not limited to, the following:

- Improving the overall accessibility of the repository by providing grammars in more notations, with better visualisation and filtering strategies.

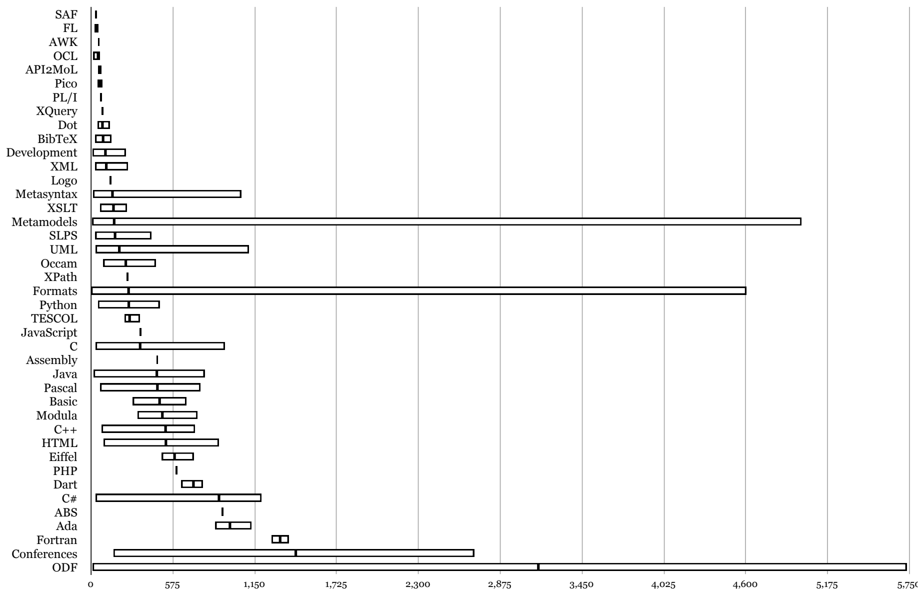

Figure 5: Grammar sizes: minimum, average and maximum — per language group. By "grammar size" we understand a cumulative metric summing the number of nonterminals, number of terminals and number of expression labels in a grammar. Such a remarkably small size for PL/I can be explained by the fact that at the moment we only have one simplified metamodel of the language.

- Assimilating other grammar collections — for instance, the Grammar Zoo contains several ANTLR grammars from the ANTLR Grammar List [56], but not yet all of them.
- Curating the corpus and raising the level of grammars there for the sake of enabling bulk processing of only L2 or only L3 grammars, in the manner that was performed in [18] with differential testing of parsers.
- Adding a time dimension: for grammars which source is a repository, all versions can be extracted and somehow presented in the Grammar Zoo. This has been done as an experiment for some of the DSL grammars used within SLPS: BGF, a BNF-like grammar format, and LDF, a unified format for language documentation.
- Mining the corpus for various properties — for example, defining a collection of micropatterns based on the current content of the Grammar Zoo has been done in [21].
- Performing comparative studies on grammar-based techniques.
- Empirical studies of grammar improvement: user studies or genetic algorithms.

- Developing methods of inferring notation specifications.
- Adding interactive on demand grammar export instead of providing a static collection of predefined notations.
- Bridging technological spaces by investigating grammars from them, megamodelling the corpus itself and grammars in it.
- Improving interoperability of metagrammarware.

6. Conclusion

Many claims about design of software languages and their descriptions (grammars in a broad sense, per [1]) can be found in existing literature [4, 5, 6, 7, 8, 9, 10, 11, 12, 13, et al.], all backed up by case studies and expert opinions. In order to support, challenge or generalise them, one could collect relevant statistical evidence, based on a sufficiently large corpus of diverse language definitions. Such a corpus, referred to as the **Grammar Zoo**, was proposed in this paper. This corpus will support replicability across grammar-based experiments (by providing open access to its grammars to the public), support aggregation of findings (by annotating the grammars extensively), reduce the cost of controlled experiments (by shifting the focus of future research from obtaining the grammars for case studies to newly proposed techniques), aid to obtain sample representativeness (by making the grammarbase large and versatile) and help to isolate the effects of individual factors (by metadata-based filtering). The Grammar Zoo initiative is modelled after successful projects like Qualitas Corpus [2] in source code analysis and Atlantic Metamodel Zoo [3] in modelware.

By relying on previous experiences in grammar extraction and grammar recovery [37, 30, 38, 31, 36, 41, 23, 24, 19, 25], we provide methods and tools for relatively semi-automatic easy extraction of grammars in a broad sense from various software artefacts: syntax specifications, type definitions, data schemata, etc. These methods and tools rely systematic manipulation of syntactic notations, on reproducible specification of grammar evolution steps, on advanced IDE support, as well as on some other less recently developed technologies. A unified data model for systematic accumulation of grammar knowledge has been designed, presented and exemplified.

The Grammar Zoo, publicly available as `http://slps.github.io/zoo`, is a collection of big grammars of mainstream languages such as Ada, C, C++, C#, Dart, Modula, Fortran, as well as small grammars used for various purposes, mostly for demonstrating certain software language engineering techniques on a proof-of-concept scale. Its name stems from the activity known as "*grammar hunting*" [19] or "grammar stealing" [38] and hints at the fact that the result of a hunt is not cooked, eaten and gone, but rather carefully put on display. The Grammar Zoo at the time of submission has 569 grammars, and continues growing.

References

[1] P. Klint, R. Lämmel, C. Verhoef, Toward an Engineering Discipline for Grammarware, ACM Transactions on Software Engineering Methodology (TOSEM) 14 (3) (2005) 331–380.

[2] E. Tempero, C. Anslow, J. Dietrich, T. Han, J. Li, M. Lumpe, H. Melton, J. Noble, Qualitas Corpus: A Curated Collection of Java Code for Empirical Studies, in: Asia Pacific Software Engineering Conference (APSEC 2010), 2010, pp. 336–345.

[3] J. Cabot, M. Tisi, H. Brunelière, et al., AtlantEcore Metamodel Zoo, http://www.emn.fr/z-info/atlanmod/index.php/Ecore (2003).

[4] A. van Wijngaarden, Generalized ALGOL, in: R. Goodman (Ed.), Annual review in automatic programming 3, Pergamon Press, 1963, pp. 17–26.

[5] N. Wirth, On the Design of Programming Languages, in: IFIP Congress, 1974, pp. 386–393.

[6] C. A. R. Hoare, Hints on Programming Language Design, Tech. rep., Stanford University, Stanford, CA, USA (1973).

[7] M. Mernik, J. Heering, A. M. Sloane, When and How to Develop Domain-Specific Languages, ACM Computing Surveys 37 (4) (2005) 316–344. doi:10.1145/1118890.1118892.

[8] M. Völter, S. Benz, C. Dietrich, B. Engelmann, M. Helander, L. C. L. Kats, E. Visser, G. Wachsmuth, DSL Engineering: Designing, Implementing and Using Domain-Specific Languages, dslbook.org, 2013.

[9] N. Chomsky, Syntactic Structures, Mouton, 1957.

[10] M. Erwig, E. Walkingshaw, Semantics first!, in: Proceedings of the 4th international conference on Software Language Engineering, SLE'11, Springer-Verlag, Berlin, Heidelberg, 2012, pp. 243–262. doi:10.1007/978-3-642-28830-2_14.
URL http://dx.doi.org/10.1007/978-3-642-28830-2_14

[11] L. Tratt, Evolving a DSL Implementation, in: Generative and Transformational Techniques in Software Engineering, Vol. 5235 of Lecture Notes in Computer Science, Springer Berlin / Heidelberg, 2008, pp. 425–441. doi:10.1007/978-3-540-88643-3_11.

[12] M. Herrmannsdörfer, S. D. Vermolen, G. Wachsmuth, An extensive catalog of operators for the coupled evolution of metamodels and models, in: Proceedings of the Third international conference on Software language engineering, SLE'10, Springer-Verlag, Berlin, Heidelberg, 2011, pp. 163–182.
URL http://dl.acm.org/citation.cfm?id=1964571.1964585

[13] J. Hutchinson, J. Whittle, M. Rouncefield, S. Kristoffersen, Empirical assessment of mde in industry, in: Proceedings of the 33rd International Conference on Software Engineering, ICSE '11, ACM, New York, NY, USA, 2011, pp. 471–480. doi:10.1145/1985793.1985858.
URL http://doi.acm.org/10.1145/1985793.1985858

[14] H. Do, S. Elbaum, G. Rothermel, Supporting Controlled Experimentation with Testing Techniques: An Infrastructure and its Potential Impact, Journal of Empirical Software Engineering 10 (4) (2005) 405–435.

[15] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, A. Wesslén, Experimentation in Software Engineering: an Introduction, Kluwer Academic Publishers, Norwell, MA, USA, 2000.

[16] V. R. Basili, F. Shull, F. Lanubile, Building Knowledge through Families of Experiments, IEEE Transactions on Software Engineering 25 (4) (1999) 456–473.

[17] L. M. Pickard, B. A. Kitchenham, P. W. Jones, Combining Empirical Results in Software Engineering, Journal of Information and Software Technology 40 (14) (1998) 811–821.

[18] B. Fischer, R. Lämmel, V. Zaytsev, Comparison of Context-free Grammars Based on Parsing Generated Test Data, in: U. Aßmann, A. Sloane (Eds.), Post-proceedings of the Fourth International Conference on Software Language Engineering (SLE 2011), Vol. 6940 of LNCS, Springer, Heidelberg, 2012, pp. 324–343. doi:10.1007/978-3-642-28830-2_18.

[19] V. Zaytsev, Notation-Parametric Grammar Recovery, in: A. Sloane, S. Andova (Eds.), Post-proceedings of the 12th International Workshop on Language Descriptions, Tools, and Applications (LDTA 2012), ACM Digital Library, 2012, pp. 9:1–9:8. doi:10.1145/2427048.2427057.

[20] J. Gil, I. Maman, Micro Patterns in Java Code, in: Proceedings of OOPSLA'05, ACM, 2005, pp. 97–116.

[21] V. Zaytsev, Micropatterns in Grammars, accepted to the Sixth International Conference on Software Language Engineering (SLE 2013). In print (Jun. 2013).

[22] V. Zaytsev, R. Lämmel, T. van der Storm, L. Renggli, G. Wachsmuth, Software Language Processing Suite[8], http://slps.github.io. Contains, among other works: Grammar Zoo (V. Zaytsev, 2009–2013), http://slps.github.io/zoo. (2008–2013).

[23] R. Lämmel, V. Zaytsev, An Introduction to Grammar Convergence, in: M. Leuschel, H. Wehrheim (Eds.), Proceedings of the Seventh International Conference on Integrated Formal Methods (iFM 2009), Vol. 5423 of LNCS, Springer-Verlag, Berlin, Heidelberg, 2009, pp. 246–260. doi:10.1007/978-3-642-00255-7_17.

[24] R. Lämmel, V. Zaytsev, Recovering Grammar Relationships for the Java Language Specification, Software Quality Journal (SQJ) 19 (2) (2011) 333–378. doi:10.1007/s11219-010-9116-5.

[25] V. Zaytsev, BNF WAS HERE: What Have We Done About the Unnecessary Diversity of Notation for Syntactic Definitions, in: S. Ossowski, P. Lecca (Eds.), Programming Languages Track, Volume II of the Proceedings of the 27th ACM Symposium on Applied Computing (SAC 2012), ACM, Riva del Garda, Trento, Italy, 2012, pp. 1910–1915. doi:10.1145/2245276.2232090.

[26] V. Zaytsev, R. Lämmel, A Unified Format for Language Documents, in: B. A. Malloy, S. Staab, M. G. J. van den Brand (Eds.), Post-proceedings of the Third International Conference on Software Language Engineering (SLE 2010), Vol. 6563 of LNCS, Springer-Verlag, Berlin, Heidelberg, 2011, pp. 206–225. doi:10.1007/978-3-642-19440-5_13.

[27] V. Zaytsev, Language Evolution, Metasyntactically, Electronic Communications of the European Association of Software Science and Technology (EC-EASST) 49.
URL http://journal.ub.tu-berlin.de/eceasst/article/view/708

[28] J. Bézivin, F. Jouault, P. Valduriez, On the Need for Megamodels, OOPSLA & GPCE, Workshop on best MDSD practices.

[8]The authors are given according to the list of contributors at http://github.com/grammarware/slps/graphs/contributors.

[29] J.-M. Favre, R. Lämmel, A. Varanovich, Modeling the Linguistic Architecture of Software Products, in: Proceedings of the 15th international conference on Model Driven Engineering Languages and Systems (MoDELS), LNCS, Springer, 2012, pp. 151–167.

[30] M. G. J. van den Brand, M. P. A. Sellink, C. Verhoef, Obtaining a COBOL Grammar from Legacy Code for Reengineering Purposes, in: M. P. A. Sellink (Ed.), Proceedings of the Second International Workshop on the Theory and Practice of Algebraic Specifications, Springer-Verlag, Berlin, 1997, pp. 6–17.

[31] R. Lämmel, C. Verhoef, Semi-automatic Grammar Recovery, Software—Practice & Experience 31 (15) (2001) 1395–1438.

[32] J. F. Power, B. A. Malloy, A Metrics Suite for Grammar-based Software, Journal of Software Maintenance and Evolution: Research and Practice 16 (2004) 405–426.

[33] C. Julien, M. Črepinšek, R. Forax, T. Kosar, M. Mernik, G. Roussel, On Defining Quality Based Grammar Metrics, in: Proceedings of the International Multiconference on Computer Science and Information Technology, IMCSIT 2009, 2009, pp. 651–658.

[34] R. Lämmel, W. Schulte, 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), Vol. 3964 of LNCS, Springer, 2006, pp. 19–38.

[35] P. I. Manuel, ANSI Cobol III in SDF + an ASF Definition of a Y2K Tool, Master's thesis, Universiteit van Amsterdam, The Netherlands (Nov. 1996).

[36] R. Lämmel, C. Verhoef, Browsable grammars, http://www.cs.vu.nl/grammarware/browsable, contains: VS COBOL II grammar Version 1.0.4 (Lämmel, Verhoef, 1999–2003), http://www.cs.vu.nl/grammarware/browsable/vs-cobol-ii; COBOL grammar Version 0.1.1 (Lämmel, Verhoef, 1999), http://www.cs.vu.nl/grammarware/browsable/cobol; OS PL/I V2R3 grammar Version 0.1 (Lämmel, Verhoef, 1999), http://www.cs.vu.nl/grammarware/browsable/os-pli-v2r3; Browsable Ada 95 Grammar (Lämmel, Verhoef, 2000), http://www.cs.vu.nl/grammarware/browsable/ada; C# Grammar Recovered (Zaytsev, 2005), http://www.cs.vu.nl/grammarware/browsable/CSharp (1999).

[37] M. P. A. Sellink, C. Verhoef, Development, Assessment, and Reengineering of Language Descriptions, in: J. Ebert, C. Verhoef (Eds.), Proceedings of the Fourth European Conference on Software Maintenance and Reengineering (CSMR 2000), IEEE Computer Society, 2000, pp. 151–160.

[38] R. Lämmel, C. Verhoef, Cracking the 500-Language Problem, IEEE Software (2001) 78–88.

[39] IBM Library, SX26-3721-05: VS COBOL II Application Programming Reference Summary, Release 4, available at publibz.boulder.ibm.com/cgi-bin/bookmgr_OS390/BOOKS/IGYR1101 (1987).

[40] Standard ECMA-334, C# Language Specification, 4^{th} Edition, available at http://ecma-international.org/publications/standards/Ecma-334.htm (June 2006).

[41] V. Zaytsev, 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, 2005, pp. 154–155, extended abstract.

[42] J. Gosling, B. Joy, G. L. Steele, The Java Language Specification, Addison-Wesley, 1996, available at http://java.sun.com/docs/books/jls.

[43] J. Gosling, B. Joy, G. L. Steele, G. Bracha, The Java Language Specification, 2nd Edition, Addison-Wesley, 2000, available at http://java.sun.com/docs/books/jls.

[44] J. Gosling, B. Joy, G. L. Steele, G. Bracha, The Java Language Specification, 3rd Edition, Addison-Wesley, 2005, available at http://java.sun.com/docs/books/jls.

[45] V. Zaytsev, MediaWiki Grammar Recovery, Computing Research Repository (CoRR) 1107.4661 (2011) 1–47.

[46] V. Zaytsev, The Grammar Hammer of 2012, Computing Research Repository (CoRR) 1212.4446 (2012) 1–32.

[47] N. Wirth, What Can We Do about the Unnecessary Diversity of Notation for Syntactic Definitions?, Communications of the ACM 20 (11) (1977) 822–823.

[48] A. Stevenson, J. R. Cordy, Grammatical Inference in Software Engineering: An Overview of the State of the Art, in: K. Czarnecki, G. Hedin (Eds.), Proceedings of the Fifth International Conference on Software Language Engineering (SLE 2012), Vol. 7745 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2013, pp. 204–223.

[49] J. Heering, P. R. H. Hendriks, P. Klint, J. Rekers, The Syntax Definition Formalism SDF—Reference Manual, ACM SIGPLAN Notices 24 (11) (1989) 43–75.

[50] V. Zaytsev, Recovery, Convergence and Documentation of Languages, Phd thesis, Vrije Universiteit, Amsterdam, The Netherlands (Oct. 2010).

[51] E. Visser, Syntax Definition for Language Prototyping, Ph.D. thesis, University of Amsterdam (Sep. 1997).

[52] T. Parr, The Definitive ANTLR Reference: Building Domain-Specific Languages, 1st Edition, Pragmatic Programmers, Pragmatic Bookshelf, 2007.

[53] F. Pereira, D. Warren, Definite Clause Grammars for Language Analysis, in: B. J. Grosz, K. Sparck-Jones, B. L. Webber (Eds.), Readings in Natural Language Processing, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1986, pp. 101–124.

[54] T. R. Dean, J. R. Cordy, A. J. Malton, K. A. Schneider, Grammar Programming in TXL, in: Proceedings of the Second IEEE International Conference on Source Code Analysis and Manipulation (SCAM 2002), IEEE, 2002, pp. 93–102.

[55] S. Gao, C. M. Sperberg-McQueen, H. S. Thompson, N. Mendelsohn, D. Beech, M. Maloney, W3C XML Schema Definition Language (XSD) 1.1 Part 1: Structures, W3C Candidate Recommendationhttp://www.w3.org/TR/2009/CR-xmlschema11-1-20090430.

[56] T. Parr, et al., ANTLR Grammar List, http://www.antlr.org/grammar/list (2003).

[57] J. R. Cordy, et al., TXL World: Grammars, http://www.txl.ca/nresources.html (2003).

[58] O. Nierstrasz, M. Kobel, T. Gîrba, M. Lanza, H. Bunke, Example-Driven Reconstruction of Software Models, in: R. Krikhaar, C. Verhoef, G. D. Lucca (Eds.), Proceedings of the 11th European Conference on Software Maintenance and Reengineering (CSMR 2007), IEEE Computer Society, 2007, pp. 275–284.

[59] M. Herrmannsdörfer, D. Ratiu, M. Kögel, Metamodel Usage Analysis for Identifying Metamodel Improvements, in: B. A. Malloy, S. Staab, M. G. J. van den Brand (Eds.), Post-proceedings of the Third International Conference on Software Language Engineering (SLE 2010), Vol. 6563 of LNCS, Springer-Verlag, Berlin, Heidelberg, 2011, pp. 62–81.

[60] M. Lungu, M. Lanza, O. Nierstrasz, Evolutionary and Collaborative Software Architecture Recovery with Softwarenaut, Science of Computer Programming (2012) 20 pages. In press. Available online since 16 May 2012.

[61] M. Črepinšek, M. Mernik, F. Javed, B. R. Bryant, A. Sprague, Extracting Grammar from Programs: Evolutionary Approach, SIGPLAN Notices 40 (4) (2005) 39–46.

[62] R. Bosworth, Syntax specification (section 18) is inconsistent with other sections, http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=6442525 (2006).

[63] R. Lämmel, G. Wachsmuth, Transformation of SDF Syntax Definitions in the ASF+SDF Meta-Environment, Electronic Notes in Theoretical Computer Science 44 (2) (2001) 9–33.

[64] R. Lämmel, Grammar Adaptation, in: Proceedings of the International Symposium of Formal Methods Europe on Formal Methods for Increasing Software Productivity, Vol. 2021 of LNCS, Springer-Verlag, 2001, pp. 550–570.

[65] J. Kort, R. Lämmel, C. Verhoef, The Grammar Deployment Kit. System Demonstration, Electronic Notes in Theoretical Computer Science 65 (3) (2002) 117–123, Second Workshop on Language Descriptions, Tools and Applications (LDTA 2002).

[66] J. Kort, Grammar Deployment Kit: Reference Manual, Universiteit Amsterdam, http://gdk.sourceforge.net/gdkref.pdf (May 2003).

[67] G. Wachsmuth, Metamodel Adaptation and Model Co-adaptation, in: E. Ernst (Ed.), Proceedings of the 21st European Conference on Object-Oriented Programming (ECOOP 2007), Vol. 4609 of LNCS, Springer, 2007, pp. 600–624.

[68] A. Cicchetti, F. Ciccozzi, T. Leveque, A. Pierantonio, On the concurrent versioning of metamodels and models: challenges and possible solutions, in: Proceedings of the Second International Workshop on Model Comparison in Practice (IWMCP 2011), ACM, New York, NY, USA, 2011, pp. 16–25.

[69] M. Herrmannsdörfer, S. Vermolen, G. Wachsmuth, An Extensive Catalog of Operators for the Coupled Evolution of Metamodels and Models, in: B. A. Malloy, S. Staab, M. G. J. van den Brand (Eds.), Post-proceedings of the Third International Conference on Software Language Engineering (SLE 2010), Vol. 6563 of LNCS, Springer-Verlag, Berlin, Heidelberg, 2011, pp. 163–182.

[70] R. Lämmel, The Amsterdam Toolkit for Language Archaeology, Electronic Notes in Theoretical Computer Science 137 (3) (2005) 43–55.

[71] M. van der Graaf, A Specification of Box to HTML in ASF+SDF, Master's thesis, Universiteit van Amsterdam, The Netherlands (Aug. 1997).

[72] A. Sellink, C. Verhoef, Generation of Software Renovation Factories from Compilers, in: Proceedings of 15th International Conference on Software Maintenance (ICSM 1999), 1999, pp. 245–255.

[73] P. Klint, T. van der Storm, J. Vinju, EASY Meta-programming with Rascal, in: J. M. Fernandes, R. Lämmel, J. Visser, J. Saraiva (Eds.), Post-proceedings of the Third International Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE 2009), Vol. 6491 of LNCS, Springer-Verlag, Berlin, Heidelberg, 2011, pp. 222–289.

[74] I. D. Baxter, C. Pidgeon, M. Mehlich, Dms: Program transformations for practical scalable software evolution, in: Proceedings of the 26th International Conference on Software Engineering, ICSE '04, IEEE Computer Society, Washington, DC, USA, 2004, pp. 625–634.

[75] ISO/IEC 14977:1996(E), Information Technology—Syntactic Metalanguage—Extended BNF, available at http://www.cl.cam.ac.uk/~mgk25/iso-14977.pdf.

[76] J. Levine, flex & bison, O'Reilly Media, 2009.

[77] T. Copeland, Generating Parsers with JavaCC: An Easy to Use Guide for Programmers, Centennial Books, 2007.

[78] A. M. Sloane, L. C. Kats, E. Visser, A Pure Embedding of Attribute Grammars, Science of Computer Programming (2011) 18 pages. In press. Available online since 29 November 2011.

[79] A. Izmaylova, A. H. Bagge, A. Lankamp, A. van der Ploeg, B. Basten, B. Lisser, D. Landman, J. Timmer, J. van den Bos, J. J. Vinju, M. Hills, M. Steindorfer, P. Klint, T. van der Storm, V. Zaytsev, Rascal Language Library[9], http://github.com/cwi-swat/rascal, the src/org/rascalmpl/library/lang directory (2010–2013).

[80] J. Vinju, M. van den Brand, T. van der Storm, M. Bravenboer, SDF Library[10], http://github.com/cwi-swat/meta-environment, the sdf-library directory (1996–2012).

[81] S. C. Johnson, YACC—Yet Another Compiler Compiler, Computer Science Technical Report 32, AT&T Bell Laboratories, Murray Hill, New Jersey, 1975.

[82] F. Jouault, J. Bézivin, KM3: A DSL for Metamodel Specification, in: R. Gorrieri, H. Wehrheim (Eds.), Proceedings of the International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS 2006), Vol. 4037 of Lecture Notes in Computer Science, Springer, Bologna, Italy, 2006, pp. 171–185.

[83] Microsoft Corporation, The Microsoft code name "M" Modeling Language Specification, http://msdn.microsoft.com/en-us/library/dd285271.aspx, obsolete.

[84] M. Fogus, C. Houser, The Joy of Clojure, Manning, 2010.

[85] Object Management Group, Semantics Of Business Vocabulary And Business Rules, v1.0 Edition, available at http://www.omg.org/spec/SBVR/1.0 (2008).

[86] Object Management Group, Unified Modeling Language, 2nd Edition, available at http://schema.omg.org/spec/UML/2.1.1 (2007).

[87] U. Brandes, M. Eiglsperger, I. Herman, M. Himsolt, M. S. Marshall, Graphml progress report: Structural layer proposal, in: P. Mutzel, M. Jnger, S. Leipert (Eds.), Revised Papers of the 9th International Symposium on Graph Drawing (GD 2001), Vol. 2265 of Lecture Notes in Computer Science, Springer, 2001, pp. 501–512.

[88] D. L. McGuinness, F. van Harmelen, OWL: Web Ontology Language. Overview, W3C Recommendation. http://www.w3.org/TR/owl-features.

[89] Object Management Group, Meta-Object Facility (MOF™) Core Specification, 2nd Edition, available at http://www.omg.org/spec/MOF/2.0 (January 2006).

[90] A. Kelemenová, Grammatical Levels of the Position Restricted Grammars, in: Proceedings on Mathematical Foundations of Computer Science, Springer-Verlag, London, UK, 1981, pp. 347–359.

[9] The authors are given alphabetically according to the list of contributors produced by the `git log` command.

[10] The authors are given according to the `AUTHORS` file provided in the distribution.

[91] A. Okhotin, Boolean Grammars, Information and Computation 194 (1) (2004) 19–48, http://users.utu.fi/aleokh/papers/boolean_grammars_ic.pdf. doi:10.1016/j.ic.2004.03.006.

[92] M. G. J. van den Brand, J. Scheerder, J. J. Vinju, E. Visser, Disambiguation Filters for Scannerless Generalized LR Parsers, in: N. Horspool (Ed.), Compiler Construction 2002 (CC 2002), 2002, pp. 143–158.

[93] H. J. S. Basten, Tracking Down the Origins of Ambiguity in Context-free Grammars, in: Proceedings of the Seventh International colloquium conference on Theoretical Aspects of Computing, ICTAC'10, Springer-Verlag, Berlin, Heidelberg, 2010, pp. 76–90.

[94] P. Purdom, A Sentence Generator for Testing Parsers, BIT 12 (3) (1972) 366–375.

[95] Eclipse, Eclipse Modeling Framework Project (EMF 2.4), http://www.eclipse.org/modeling/emf/ (2008).

[96] M. Murata, RELAX NG Schemas, http://relaxng.org/#schemas (2013).

[97] J. Clark, M. Murata, RELAX NG Specification, OASIS Committee SpecificationAvailable at http://relaxng.org/spec-20011203.html, also being standardised by ISO as multiple parts of ISO/IEC 19757.

[98] V. Zaytsev, Guided Grammar Convergence, accepted to the poster session of the Sixth International Conference on Software Language Engineering (SLE 2013) (Jun. 2013).

[99] T. Parr, ANTLR v3 — ANother Tool for Language Recognition, http://antlr3.org (2008).

[100] ISO/IEC JTC1/SC22 — International Standardization Subcommittee for Programming Languages, Their Environments and System Software Interfaces, http://www.open-std.org/jtc1/sc22.

[101] Ecma International — Standards @ Internet Speed, http://www.ecma-international.org, accessed in June 2009.

[102] K. Czarnecki, J. Foster, Z. Hu, R. Lämmel, A. Schürr, J. Terwilliger, Bidirectional Transformations: A Cross-Discipline Perspective, in: R. Paige (Ed.), Theory and Practice of Model Transformations, Vol. 5563 of LNCS, Springer Berlin / Heidelberg, 2009, pp. 260–283.

[103] J. N. Foster, M. B. Greenwald, J. T. Moore, B. C. Pierce, A. Schmitt, Combinators for Bidirectional Tree Transformations: A Linguistic Approach to the View-Update Problem, ACM Transactions on Programming Languages and Systems (TOPLAS) 29.

[104] S. Klusener, V. Zaytsev, Language Standardization Needs Grammarware, JTC1/SC22 Document N3977, ISO/IEC (2005).
URL http://www.open-std.org/jtc1/sc22/open/n3977.pdf

[105] M. H. Halstead, Elements of Software Science, Elsevier Science Inc., New York, NY, USA, 1977.