Towards a Taxonomy of Grammar Smells

Mats Stijlaart
Universiteit van Amsterdam
Amsterdam, The Netherlands
mstijlaart@gmail.com

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

Abstract

Any grammar engineer can tell a good grammar from a bad one, but there is no commonly accepted taxonomy of indicators of required grammar refactorings. One of the consequences of this lack of general smell taxonomy is the scarcity of tools to assess and improve the quality of grammars. By combining two lines of research—on smell detection and on grammar transformation—we have assembled a taxonomy of smells in grammars. As a pilot case, the detectors for identified smells were implemented for grammars in a broad sense and applied to the 641 grammars of the Grammar Zoo.

CCS Concepts • **Theory of computation → Grammars and context-free languages**; • **Software and its engineering → Software defect analysis**; *Patterns*; *Compilers*;

Keywords Smell detection, grammar engineering

ACM Reference Format:
Mats Stijlaart and Vadim Zaytsev. 2017. Towards a Taxonomy of Grammar Smells. In *Proceedings of 2017 ACM SIGPLAN International Conference on Software Language Engineering (SLE'17).* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3136014.3136035

1 Introduction

As it has been pointed out over a decade ago [22], the underlying goal behind establishing the engineering discipline of grammarware is to improve the quality of grammarware: grammars, compilers, IDEs, APIs—software languages and their processors. Since then, the goal has been getting closer, but the progress was more measurable in some domains than in others. The quality of grammars as such, has always been and remains, an ephemeral concept: everyone agrees that it should be high, many experts easily tell a good grammar from a bad one when they see it, but quantifying the difference and turning it into an executable tool that detects when a grammar is in need of refactoring and whether such a refactoring indeed improved it, is way out of reach. Researchers and tool implementers from various domains use proxies for quality such as size and complexity metrics [4], the number of differences to the master grammar [30, 65], the number and nature of detectable ambiguities [7], the connectedness and the factual correspondence with executable tooling [29], the point in the lifecycle between dead text in a manual and the source artefact for tool generation [63], etc.

Language design smells as a research domain within software language engineering, were proposed by Tijs van der Storm in February 2012, as documented by *The Grammar Hammer* [59] at the end of that year. The problems of this domain are socio-technical by nature, and range from attempts to introduce an indentation-sensitive software language for the domain traditionally filled with C++ programmers (will fail even if there is nothing technically wrong with it), to having a Rascal grammar with a terminal partly defined with a `lexical` production rule and partly with a `syntax` one (technical and idiosyncratic up to being obscure for non-Rascal experts). With this paper, we contribute to this domain in an a priori focused way: by providing a list and a classification for grammar smells, a subset of software language smells.

By *"smells"* we mean here symptoms of a possible problem that indicate that something is not quite right, but not necessarily point out an error and thus require human touch for verification [17, 46, 48]. Smells in code are often results of undisciplined work, hasty development and rushed design decisions. Similarly, smells in grammars are fuzzily described symptoms that may indicate imperfections in their engineering. By *"grammars"* we will understand extended Boolean grammars in a broad sense. Thus, they may contain any kind of repetition [54], disjunction [5], conjunction, negation [38], as well as other advanced metaconstructs like separator lists; and these grammars are not necessarily related to parsing (even in a broad sense [68]), instead representing structural commitment [22] and binding contracts [13]. By choosing this viewpoint, we possibly embrace all grammarware technologies and more, covering syntax specifications along with document schemata, metamodels and whatnot.

The next section briefly describes prior results related to our work. Sections 3–5 define smells as such, with examples when space permits. http://slebok.github.io/grass contains a more complete and constantly growing counterpart of those sections. Next, section 6 shows how to apply the taxonomy by detecting the smells on the Grammar Zoo corpus [64] with detectors written in GrammarLab [67] and Rascal [23], of which one example is described with a discussion of its results. The paper is concluded by section 7.

SLE'17, October 23–24, 2017, Vancouver, Canada

2 Related Work

The project related the closest, is the grammar assessment framework developed by Sellink and Verhoef in 2000 [45]. During their own grammar engineering activities around several grammars of industrial DSLs, they noticed that certain patterns were common, harmful and detectable; implemented them in a collection of small tools and defined semi-formally. The focus of their work was to quantify the progress of grammar recovery. All the harmful patterns Sellink and Verhoef have identified, are covered by the taxonomy of this paper (Echo, Dead, Bottom, etc).

Many initiatives around grammar adaptation [29, 32], grammar programming [11, 14], grammar convergence [30, 31], grammar transformation [34], grammar mutation [62], grammar recovery [33, 58], grammar deployment [26], etc., include grammar manipulation languages that are for the most part motivated by the changes that are commonly required in imperfect grammars. We looked at all those for inspiration, reverse engineering what kind of problems may be identified so that they can be solved later by transforming or mutating the smelly grammar.

There are many metrics defined on grammars, counting rules and symbols, assessing complexity and calculating properties of inferred structures (tree impurity, grammatical levels, etc), it was a popular topic of research in the 1970s and 1980s. The main fruits of those research activities are summarised well by Power and Malloy [40, 41]. We will define a number of metric-based smells in subsection 5.3. One step further from metrics are micropatterns that have already been well-defined for grammars [60]. Grammar micropatterns are the counterparts of micropatterns [19], nanopatterns [8], usage patterns [37, 69] and code idioms [2] in software engineering. For example, "Preterminal" (a nonterminal which definition contains only terminals) is a micropattern for nonterminals defined by trivially combining terminals.

Grammar engineering case studies are not often being published, but there are quite a few of them nevertheless: by van den Brand et al. [50, 51], Sellink and Verhoef [45], Lämmel and Verhoef [28, 29], Tratt [49], Visser [53], Alves and Visser [4], Zaytsev [55, 56, 58, 64]. All of them talk about improving the quality of grammars and explain what patterns they find harmful and worth removing or refactoring. We will refer to these studies from the smell descriptions to establish their prevalence and acceptance among acknowledged grammar engineers.

Fowler et al. [17] wrote a book on smells in code developed in object-oriented languages. Their work has become such a classic that most smell catalogues are literal mappings ("technological space travel" [27]) thereof or at least grow from such a mapping [3, 18, 21, and over 9000 others]. OOP code is as far from grammars in a broad sense as one could be, but we do borrow Fowler et al.'s definitions and attitude towards smells as open-ended (not necessarily precise), expert-recognisable (not always automatically detectable), resolvable (useless to detect clones in a DSL without reuse) patterns, somehow representative of bad practices and antipatterns. The online version of sections 3–5 refers back to the classic and near-classic smells in code from definitions of our smells in grammars, by using one of the most recent and up to date sources: the *Refactoring for Software Design Smells* [48] book and the online resource derived from it and an accompanying literature survey, called *A Taxonomy of Software Smells* [46], regularly updated throughout 2017.

3 Organisation Smells

Let us start by considering the most global smells, related to the metalanguage, conventions of its use, as well as other problems with the entire way that the grammar was created.

3.1 Convention Smells

Convention smells are about violations of policies that a grammar engineer reading through the grammar, is supposed to know or can get to know if the reading goes on long enough. Once the convention is suspected, it will be expected to be followed, so when it is suddenly not followed, we claim a smell to be found.

3.1.1 Misformat

There are many formatting mistakes one can make when creating a grammar without proper tool support [31, 58]. Mostly they revolve around mistypings, misspellings, misalignments, etc, and result in actual incorrect constructs in extracted grammars. However, there can be other, more subtle smells within formatting of a grammar, that do not change the way the machine processes it, but do change the way a tired grammarware engineer may understand it. The most canonic example of misleading formatting would be (mind the colon, not a semicolon, following `ghi`):

```
abc :
        def;
        ghi:
        jkl;
mno :
        pqr;
```

3.1.2 Misnomer

There are a lot of potential problems with names used within a grammar, mostly concerning nonterminal names and labels. Many grammar notations do not support labels (decorative names for production rules or right hand side subexpressions) [57], but realistic metalanguages tend to have them in some form. Nonterminal names, on the other hand, are essential—they are optional only in notations for regular expressions, and present in all grammar notations of the context-free kind and beyond.

One can blame names to be *uncommunicative*, like the names from the last example: `abc` or `pqr` are much worse

for the readability and maintainability of the grammar than `if_statement`, `CompilationUnit` or `DIGIT`, similarly to how this is a problem in programming in general [9]. One can also investigate whether naming *policies* are present and how they are respected. For instance, if all nonterminals are camelcased, but one is lowercase with an underscore separator, it was probably a misspelling—cases like this were reported in a MediaWiki grammar which was created by several unrelated grammar engineers [56]. It can also be the case that the naming policy carries semantic meaning: typically lexical nonterminals and/or preterminals are named in uppercase, to distinguish them visually when they are used next to others like this:

```
if_stmt ::= IF condition THEN expression ENDIF;
```

Sometimes naming policies represent namescoping, which is considered a bad smell in OOP but is much less so in grammars because all names are global (at least up to a module level, if we have modules). An example:

```
ConstDef ::= ConstName DefKeyword ConstType;
```

Finally, names can be *misleading* and contain words that contradict the definition of the named entity. For example:

```
WhileStatement ::= "while" Condition Block
      | "repeat" Block "until" Condition ;
```

3.1.3 SayMyName

The information is conveyed both by the structure specified in the grammar notation and by natural language used for naming nonterminals and modules. In small grammars misspellings and misnamings are easy to overlook since humans are naturally capable of that. When the grammar size increases, primitive automation techniques are used like plain text search, and such a search query looking for all statements will not find the one labelled with "staetment".

3.1.4 ZigZag

ZigZag was a previously identified micropattern of a nonterminal defined in a style that mixes horizontal production rules (the ones with a top-level choice) with vertical production rules (with several rules per nonterminal) [60]. In this fragment `h` is horizontal, `v` is vertical and `z` is zigzag:

```
h ::= a | b;                    z ::= e | f;
v ::= c;          v ::= d;      z ::= g;
```

When it comes to smells, we have at least two ways to define and detect ZigZags: the *local* one within a nonterminal (equal to the micropattern) and the *global* within a grammar. The latter would mean that some nonterminals are defined horizontally while others are defined vertically, which may not be technically detrimental, but is still sloppy.

3.1.5 Splat

Since definitions of vertical nonterminals (see ZigZag) consist of several production rules, these rules can be distributed over the grammar and not focused in one place. This may be bad, especially if most of the rules are collected together, and only one or two are elsewhere.

3.2 Notation Smells

The second group of organisation smells directly concerns the notation (the metalanguage) used to write the grammar. Common properties of notations for syntactic definitions has been investigated and modelled before [57], but in reality their infinite diversity comes in infinite combinations.

3.2.1 Underuse

The original BNF used for early ALGOL, did not yet borrow Kleene star (`x*` for zero or more `x`s) and Kleene cross (`x+` for one or more) from regular expressions, and early parser specification notations were just as limited. Grammars written with those notations in mind (not necessarily for using them!), suffer from "yaccification" [12, 32], when all repetitions are written out explicitly as additional left-recursive nonterminals. This pattern is well-known to be harmful since it reduces grammar's both readability (being basically an encoding move, not a modelling one) and portability (a left-recursive grammar is often useless or suboptimal for top-down parsing). There can be other similar patterns falling under the same smell description: for instance, separator lists are concisely and efficiently handled by many grammar notations, and if one of those is used, should not be written out with group repetition or recursion.

3.2.2 Overspec

In many notations, there are various ways to achieve the same effect, and information in those should not be duplicated or contradictory, since it only confuses grammar engineers and leads to grammars with very subtle bugs. The simplest example of Overspec is something like `!"a" & ("b" | "c")`, where the choice between terminals `"b"` and `"c"` is preceded by a negative conjunctive clause saying that they at the same must not be `"a"`. Naturally, this cannot happen in either case, so the clause is either disposable or erroneous.

3.2.3 Priorities

A typical *layered* grammar [30] treats highly recursive language constructs with sophisticated priorities by explicitly encoding them in a long streak of nonterminals[1]:

```
expression ::= assignment-expr
    expression "," assignment-expr
assignment-expr ::= conditional-expr
    logical-or-expr assignment-operator assignment-expr
    throw-expr
conditional-expr ::= logical-or-expr
    logical-or-expr "?" expression ":" assignment-expr
logical-or-expr ::= logical-and-expr
```

[1] ISO/IEC 14882:1998(E), *Programming languages — C++*, extracted [64], for spacial considerations some nonterminal names have been shortened and some alternatives removed or factored.

```
    logical-or-expr "||" logical-and-expr
logical-and-expr ::= inclusive-or-expr
    logical-and-expr "&&" inclusive-or-expr
inclusive-or-expr ::= exclusive-or-expr
    inclusive-or-expr "|" exclusive-or-expr
exclusive-or-expr ::= and-expr
    exclusive-or-expr "^" and-expr
and-expr ::= equality-expr
    and-expr "&" equality-expr
equality-expr ::= relational-expr
    equality-expr ("==" | "!=") relational-expr
relational-expr ::= shift-expr
    relational-expr ("<" | ">") shift-expr
shift-expr ::= additive-expr
    shift-expr ("<<" | ">>") additive-expr
additive-expr ::= multiplicative-expr
    additive-expr ("+" | "-") multiplicative-expr
multiplicative-expr ::= pm-expr
    multiplicative-expr ("*" | "/" | "%") pm-expr
pm-expr ::= cast-expr
    pm-expr (".*" | "->*") cast-expr
cast-expr ::= unary-expr
    "(" type-id ")" cast-expr
unary-expr ::= postfix-expr
    unary-operator cast-expr
    "sizeof" unary-expr
    new-expr
    delete-expr
postfix-expr ::= primary-expr
    postfix-expr "[" expression "]"
    postfix-expr "(" expression-list? ")"
    postfix-expr ("++" | "--")
primary-expr ::= literal
    "this"
    "(" expression ")"
    id-expr
```

This example is from an obviously complicated programming language (C++), and many extra nonterminals increases this complexity. A cleaner way would have been to merge all definitions into one nonterminal (or a few conceptually grouped ones) and to define priorities between them. Priorities can be specified in a separate notation or by using ordered choices, depending on the notation. Once these priorities are defined, there can be other variations of this smell in them: circular dependencies, missing elements, etc.

3.2.4 Singleton

The designers of grammar notations, as all DSL designers, try to make them fit the domain, but never achieve absolute perfection. In particular, multiary symbols with arity of 2 and up, are commonly expressed in such a way that allows their use on an empty or trivial list of arguments, such as disjunction expressed by a prefix "one of" [57]: e.g., `a := one of b c`. Single-element sequences, disjunctions and conjunctions like this are easy to detect and remove by rewriting: e.g., for any `x`, a rule `a := one of x` is the same as `a := x`.

3.2.5 Combo

Grammar combinators (metasymbols of arity 1 and up) such as the Kleene star and cross, or an optional, can be combined in an improper way. For example, a grammar engineer who defines `A ::= B?;` and `B ::= C?;`, may mean well, but creates a confusing contract if `A` is used to bind a textual structure with a tree structure: it is ambiguous what an empty string corresponds to—an empty node `A` or a node `A` containing an empty node `B`. Some of these issues may be harder to detect due to indirection, but they are all automatically fixable.

3.2.6 Chant

During grammar recovery projects in the past we were occasionally stumbling across nonterminals that were "defined" in natural language instead of the actual grammar notation: "defined similarly to...", "all Unicode characters of class...", "any of the following", etc. These are drastic examples of this smell, since they make the grammar completely useless for automatic machine consumption, and require a human expert to either fix the grammar or devise a semiparsing [61] workaround, and possibly an extra person to translate the description to a natural language understandable by the expert. However, having improper constructions in one's grammar that are covered up by an extensive comment explaining why it is not that bad, is still an instance of this smell.

3.2.7 Deprecated

Similarly to deprecated statements and methods in programming, grammar notations may have some functionality that is no longer considered viable and proper in the new version. This does not happen all that often, but it may.

3.2.8 Exotic

This smell is in contradiction with Underuse, and states that using notational features that are uncommon, obscure or overly exotic, should be limited. Excessive use of features idiosyncratic for one particular notation, will result in a vendor lock-in. For example, if a notation allows context handling (pushing the grammar outside the comfort zone of CFGs), using it is only fully justified when the result is too cumbersome otherwise.

3.3 Parsing Smells

Even though we have stated that the subject of our investigation is Boolean grammars in a broad sense, we should not close our eyes at cases when grammars are used as parser specifications. There are at least three smells that we can identify here. The interesting aspect is that the state of the art in SLE is to try to avoid excessive impact of parsing technologies on grammar engineering [51], and, as a consequence, language workbenches tend to apply transformations that remove the smells automatically [15].

3.3.1 Factoring

In classic by-the-book [20] non-memoising parsing, if alternative production rules from the same nonterminal start from the same symbols, these would have to be reparsed in each of the branches. As a real example[2]:

```
open_if_statement
  : IF boolean_expression THEN statement
  | IF boolean_expression THEN closed_statement
                          ELSE open_statement;
```

Interestingly, this example is a false positive: factoring the first three symbols into a separate nonterminal will clutter the grammar without bringing any noticeable benefits (and will introduce the Weak smell). In other cases, this smell has been avoided/removed[3]:

```
ifStatement ::= "if" "(" expression ")" statement
                ("else" statement)?
```

3.3.2 1SidedRecursion

It is well-known that left-recursive definitions are deadly for by-the-book top-down parsing technologies [20], since they create an infinite loop and cause the parser to crash from stack overflow. There are many approaches to solve the problem by grammar refactoring or parser tweaking, available from late 1960s, but most of them, if not all, increase the size and complexity of the grammar significantly. Hence, we can imagine some scenarios when left recursion should be recognised as a smell to be reported to the grammar engineer who will fix the issue manually. This is an example[4]:

```
expression ::= expression op expression
             | id expression+
```

It must be noted here that indirect recursion (when A's right hand side starts with B whose right hand side starts with A) is just as deadly for top-down parsing as direct recursion.

Right recursion in general is less harmful, but it does lead to bottom-up parsers being slower than otherwise [20], so it should still be avoided whenever possible.

3.3.3 Superset

Some grammars represent a superset of the intended language. This may become a problem if the parser based on the grammar is to be used as a correctness oracle, since in this role it is inadequate. Overly relaxed grammars are routinely used in other scenarios such as software analytics and inter-language translation, and can be very useful there.

3.3.4 Shotgun

Shotgun parsing is a term used in cybersecurity to represent an architecture where a proper parser is substituted with lightweight treatment (by regular expression matches and direct string manipulation) [10]. The name comes from the fact that in a pipeline of tools built with such defects, the problems quickly multiply with each step when the receiver is applying Postel's Law in trying to be relaxed with its input [44], and is known to cause all kinds of subtle bugs in software language processing [66]. We define the Shotgun smell as a situation when the grammatical bind is too loose on one of its ends. For example, imagine function arguments in a C-like language to be parsed as a parenthesis-enclosing string which is expected to be split into a proper list by the code that uses the resulting tree. This smell was not seen within grammars of the Grammar Zoo, but was observed in the industrial setting when time pressure got the best of grammar engineers.

3.3.5 NoDefault

Parser generators are great for everything, except for one aspect: error handling. There are many methods that use heuristics in an attempt to improve the situation, but error detection, localisation and reporting in manually written parsers is always incomparably better than in generated ones. However, there are certain tricks experienced grammar engineers use to improve the situation. For example, consider a DSL where each statement starts with a keyword and ends with a period. An obvious improvement to the naïve approach would be to, for instance, once a keyword `MAP` is recognised, have a panic mode setup or some other semiparsing [61] machinery to fail locally and report on an "error in a map statement" rather than pointlessly try to backtrack and fail at the general statement level. To do this, one has to have a special *default* case among the rules for each particular statement kind. This recipe is more often observed in grammars written for frameworks where ordered choice is more natural (TXL, PEGs, etc), and could take the form of `Stmt := "MAP" MapStmt "." / "IF" IfStmt "." / ··· / Id (!".")* "."`. The lack of the last alternative in this example would be an indication of a NoDefault smell.

3.3.6 Action

Many realistic language workbenches draw the line to prevent endless growing of their notations, and introduce a concept of a "semantic action", which is written like an annotation in the grammar and acts as a doorway to the mainstream language typical for the target platform [15]. Obviously, since this action consists of code, the code can suffer from one of the numerous code smells.

3.4 Duplication Smells

3.4.1 Echo

A nonterminal definition is echoed if it is included in the grammar several times, each of which is identical to any other. Echoes were found in the Java Language Specification as a result of manual (not tool supported) creation of both the

[2] Doug Cooper, Scott Moore, *Pascal grammar in Yacc format*, fetched [64].
[3] The Dart Team, *Dart Programming Language Specification*, extracted [64].
[4] Vadim Zaytsev, *FL.Txl*, extracted [64].

grammar and language documentation. They were merged in the grammar extractor—that is, during the phase of converting the original HTML document to the first version of the grammar [31, p.348]. A similar error was observed earlier in the C# standard [55] and later in other languages [64].

3.4.2 Clone

Nonterminals that have exactly the same definitions, are only cluttering the grammar, and can be painlessly united. In clone management research, these are called "type 1 clones", we will revisit this classification later when defining Lookalike.

3.4.3 Foldable

This smell occurs when the clone is formed not between full definitions of two nonterminals, but when the right hand side of one nonterminal occurs as a subexpression in the right hand side of another nonterminal. Conceptually they are still clones and suffer from all known consequences of coupled evolution, but the solution is different: instead of merging the nonterminals, the subexpression needs to be folded into a nonterminal that defines it.

3.4.4 Common

One step further, we may observe clones between two or more subexpressions found in different places in the grammar. The detection pattern is almost the same as with Foldable, but the solution must involve creating a new nonterminal and then folding it (*extract* in the terminology of [30, 31]). Since creating a new nonterminal implies inventing a new name for it, and only suboptimal heuristics are available, the removal of this smell cannot be properly fully automated. It may also not be that desirable to refactor *all* common subexpressions, since doing this may introduce Weak nonterminals. Appropriate thresholds need to be investigated.

3.4.5 Permuted

Clones modulo permutations (e.g., `A | B` vs `B | A`) are confusing: if the choice used in the notation is commutative, they are just Clones, otherwise if the choice is ordered, having both `A / B` and `B / A` within the same grammar is even more confusing for everyone.

3.4.6 Lookalike

In mainstream clone detection research, people distinguish between clones of different types: exact clones ("type 1") with identical literal duplicates; parametrised clones ("type 2") with variations in identifier names, literals, even variable types; near miss clones ("type 3") where statements are allowed to be changed, added or removed up to some extent; semantic clones ("type 4") as the same computation with a different syntax and possibly even different algorithms; structural clones—higher level similarities, conceptually bottom-up-detected implementation patterns; and artefact clones—function clones and file clones [43]. For grammars, there is no developed theory of clone management, so the questions are open on what constitutes a proper clone, what classes of clones are there, which ones are useful to detect and which to refactor, etc. It makes sense to assume that clone detection in grammars will bear some similarity to contextual clones [35] that worked for another DSL with relatively few constructs, where clones were detected based on the context of clone candidate fragments, and not on their structure per se.

4 Navigation Smells

Navigation smells relate to problems that grammar engineers experience in locating entities within the grammar.

4.1 Spaghetti Smells

The most straightforward family of navigation smells is those that make grammar engineers move around too much when working with a grammar.

4.1.1 Uncluster

Nonterminals that refer to one another, should be located close to one another. The longer the distance between the use of a nonterminal from its definition, the more the reader of the grammar will have to switch context. A lot of scrolling always means there is something smelly about how the grammar is set up. Moving the production rules that cause the scrolling closer to each other to form a cluster, will result in an easier grammar with more coherent structure.

Automated removal of this smell is problematic, since the grammar engineer should decide which nonterminals to move where, but there are a lot of heuristics one can develop and test their effectiveness empirically.

4.1.2 Unsequence

The order of the production rules in the grammar should be set up in a consistent manner, such that referred nonterminals in production rules refer either up or down in the grammar. If Uncluster and Splat are concerned with general placement of production rules, this smell is about how they need to be structured to keep the reading experience optimal.

In the past we have been using a grammar mutation called *SubGrammar* to improve the readability of grammars extracted from language manuals [62]. It would reorder the rules in the following way: grab the starting symbol (usually the root of the grammar) and list all its production rules, and then go through all nonterminals used in their right hand sides one by one in the same sequence that they occur, add their production rules to the target grammar and add the newly used nonterminals to the backlog. Once the backlog was empty, the mutation stopped. This was one of the possible strategies to get rid of both Splat and Unsequence, and to some extent of Uncluster, but not the only one.

4.1.3 StartInTheMiddle

To improve the readability and navigability of the grammar, its starting symbol should be on top or the bottom of the grammar, not lost somewhere in the middle.

4.2 Shortage Smells

This subcategory of navigational smells covers complaints about missing pieces of the grammar. Depending on the notation and the particular language workbench, some of these are undeniable defects and not just smells.

4.2.1 AlmostAlphabet

The completeness of some character classes and terminal choices can be predicted, and compared to the actual value given by the grammar. For example, if a character class includes all Latin letters except one or all whitespace characters except `\r`, it may be an error. Similarly, if a preterminal is defined as a choice of all other alphanumeric terminals in the grammar (common for keyword definitions), then not a single one of them should be skipped.

4.2.2 ConfusingEntry

With StartInTheMiddle we have already addressed positioning of the starting symbol of the grammar, but there can be three more problems with it. **(1)** Some grammars do not have any start specified at all, having it to be inferred by heuristics (e.g., the only top nonterminal). **(2)** There can be multiple starts, especially for notations that exceed classic CFGs. This can indicate several independent grammars that got merged into one, or just several entry points into the grammar (which would allow, for example, to parse statements or expressions out of context—it is a not quite challenging exercise in theory, but extremely useful in practical grammarware engineering when integrating software languages with an IDE and a debugger). **(3)** The root symbol is properly marked as such, but is also referenced from other nonterminals in the grammar (so the starting symbol is not a top nonterminal).

The exact harmfulness of this smell heavily depends on the grammar handling framework.

4.2.3 Dead

All top (unused) nonterminals identified as not being the starting symbol(s) of the grammar, represent unreachable fragments. The programming counterpart of this smell is dead code and its variants. Both dead nonterminals and dead code are occasionally useful for testing and mock-ups.

4.2.4 Bottom

The lack of definition for nonterminals that are used within the grammar, is an obvious mistake that must be reported one way or another, and also possibly as a smell. There are three main reasons for undefined nonterminals: **(1)** they were forgotten by the grammar engineer; **(2)** they are defined in a different module; **(3)** they are defined on a separate conceptual layer. For scenario (1), we cannot do anything to fix the problem automatically (beyond attempting heuristics).

4.2.5 Debt

Similarly to Chant that covers up imperfect fragments with comments in natural language, there could be pieces missing entirely from the grammar and replaced with comments. If the comments admit clearly what is missing, use searchable tags like "TODO" or "FIXME" and are intended to use as a backlog, the current practice is to refer to them as "self-admitted technical debt" [39].

4.3 Mixture Smells

The last category of navigational smells is about mixing the grammar with foreign fragments that look like a grammar but have a special relationship to the rest of it.

4.3.1 BadLayout

Dealing with layout and whitespace can be very tricky, and, as any tricky process, there may be issues with it. Some language workbenches offer default layout, which, again, may be smelly to use it or not to use it—we cannot provide any general guidelines. Not specifying any layout may be harmful in some cases as well.

One particular issue with layout can be explained in a bit of more detail. Usually there are two naturally different things covered by layout: whitespace (in software languages that ignore it) and comments (that do not influence behaviour of the system but can have an impact on its understanding). Mixing those two indiscriminately in the grammar may eventually lead to the point where it is required but impossible or overly complex to get one but not the other (e.g., for handling structured comments or preserving it through transformations).

4.3.2 Preprocessor

A preprocessor [16] is a curious thing: it is essentially, for all intents and purposes, a compiler that processes the input text, expands macros, connects additional textual sources, performs variant compilation and other similar activities. On the other hand, it is so common to use it before the "actual" compiler, that some studybooks regard it as a separate phase of compilation. Some language manuals contain production rules belonging to the preprocessor, and, since the preprocessor is a separate compiler with its own grammar, those should not be mixed with the rest of the main grammar.

5 Structure Smells

Harmful relationships among grammar components.

5.1 Proxy Smells

Smells about excessive abstraction with too many entities.

5.1.1 Chain

Chain rules are a well-known smell in grammar engineering [30, 31]: it happens when a nonterminal is defined with only one production rule which has exactly one nonterminal as its right hand side. The "inner" nonterminal acts like a middle man and does not play a significant rule in structural commitments of the grammar. In Fowler's words, "after a while it's time to cut out the middle man and talk to the object that really knows what's going on" [17].

5.1.2 Throwaway

A nonterminal that is used only once, may be useful to shorten production rules (see TooWide), and may occasionally convey a useful abstraction with its name. Beyond those circumstances it is a smell and a candidate for refactoring.

5.1.3 Weak

When the right hand side of a nonterminal is formed from several symbols that happen to occur one after another, without forming a proper abstraction, this can hinder grammar's understanding. The terminology is borrowed from [36] which distinguished *strong* nonterminals (used during parsing and present in the resulting tree) from *weak* ones (used during parsing but flattened into single nodes in the tree).

5.1.4 Ghost

If an expression, especially a Common subexpression, could have formed a proper abstraction, but is not made into a separate nonterminal, we speak of it as a Ghost. An example of a Ghost could be an omnipresent qualified identifier that is always used as `Id ("." Id)*`.

5.1.5 Multitool

This takes place when a nonterminal violates the single responsibility principle, and represents several (hopefully related) abstractions, such as a type name and a variable name.

5.2 Dependency Smells

5.2.1 Diamond

A well-known pattern in dependency and inclusion is when a class A inherits from class X and class B also inherits from class X, but class C inherits from both A and B and thus gets to see double of each of X's elements. The problem is solved differently in different programming and modelling languages—in grammars, it causes an ambiguity[5]:

```
reference-type ::= class-type | interface-type
                 | delegate-type | ... ;
class-type ::= type-name | "object" | "string";
interface-type ::= type-name ;
delegate-type ::= type-name ;
```

[5]ISO/IEC 23270:2003(E), *Information technology — Programming languages — C Sharp*, extracted [64].

Parsing something recognisable as `type-name` will cause at least a triple ambiguity since it will be an acceptable `class-type`, `interface-type` as well as `delegate-type` [55].

5.2.2 Rivalry

Traditionally grammar notations advertised to have a sequence combinator and a choice combinator as the ways to compose complex expressions from atomic symbols (terminals and nonterminals) [5]. Modern frameworks are more advanced and versatile, they feature several kinds of negation, conjunction, ordered choices, precede and follow restrictions, etc. However, there is none that explicitly provides an exclusive disjunction combinator [57], even though it was the original intent behind the choice: thus, a statement may be a conditional statement or a print statement, but not both at the same time. There is an entire research domain dedicated to *disambiguation* of grammars—that is, detecting when something intended as an exclusive choice, leads to several successful parses, only one of which must remain in the final tree [1]. This smell is about such situations: it occurs whenever languages of alternative siblings overlap and create an ambiguity. Ambiguity detection and removal techniques can be powerful but not perfect, and the general case is undecidable because it requires decidability of language equivalence.

5.2.3 Ouroboros

If nonterminals are mutually, say, left recursive, and have no non-recursive alternatives, they are useless and cannot express a proper syntactic commitment. However, a similar issue may be encountered on the level of modules, and it is harder to detect for a human because modular grammars are already stretching comprehension capabilities of a grammar engineer. Such circular dependencies are fairly easy to detect automatically with a tool.

5.2.4 Soulmates

If enough information is available about the evolution of the grammar (e.g., in a form of a versioned repository or a piece of documentation describing all changes), one can notice two nonterminals having a so-called *co-change relationship* when each revision that changes one, also changes the other. The smell occurs when this co-change relationship in the revision log does not correspond to explicit dependencies between modules and nonterminals.

In the absence of change history, a thought experiment may serve the same purpose: if a nonterminal `X` is to change in a particular way, what other nonterminals will have to be inevitably co-updated to preserve the consistency of the grammar? All of those, if any, are its Soulmates. It deserves mentioning that the programming counterpart of this smell, called "Feature Envy", is detectable automatically through static code analysis. The Soulmates smell, however, is only detectable with revision mining or conceptual analysis.

5.2.5 Spillover

Spillover happens when some symbols that should have been a part of the nonterminal definition, are not included in its right hand side, and appended every time to its use. This creates a co-change relationship between the nonterminal and the context of its use. With Spillover, every time a definition of a nonterminal is changed in a particular way, all occurrences of the same nonterminal needs to be updated.

5.2.6 Mythic

Formal language theory defines a language *extensionally*, as a set of all possible programs written in it. A grammar is an *intensional* definition, which is nicer because it is a finite specification of an infinitely large entity, but it also makes it harder to see some relations between that and the instances of the language. In particular, the actual codebase of the software language, if available and comprehensive enough, can serve as a good approximation of the language features used by programmers. If a grammar contains a feature that is never exercised by any program in the actual codebase, it is a Mythic feature that, does not have to be supported for an analysis or migration tool to be useful and applicable.

5.3 Complexity Smells

This subcategory collects issues that make a grammar seem big and complex. There are two state of the art approaches to declaring something "too big": either by setting a threshold (e.g., "big" is above 20), or by searching for statistical outliers (e.g., "big" is bigger than 90% others).

5.3.1 TooWide

This smell is designed to recognise production rules which are too wide—that is, their right hand side is too long. There could be at least three ways to define what is "too long":

- The number of consequent terminals is too high, which is harmful because long streaks of consequent terminals obscure the syntactic structure.
- The number of nonterminals is too high, which is harmful because it requires knowledge about referenced nonterminals to debug a grammar (so we should not count preterminals).
- The number of metasymbols (stars, optionals, crosses, separator lists and other combinators) is too high, which is harmful because the importance of knowing the notation is stressed when a grammar engineer needs to understand such a production rule.

The last option also correlates to the omnipresent notion of cyclomatic complexity (covered by the TooRamose smell), because many metasymbols imply branching that is done during parsing or analysing an instance.

5.3.2 TooRamose

McCabe's cyclomatic complexity has received a lot of critique over the years, but nevertheless is present in many code analysis tools either directly or conceptually through improvements like cognitive complexity [42] or control flow patterns [52]. In grammars cyclomatic complexity is easy to estimate if we think of the parsing semantics, and it will be rather close for any other concrete application of the grammar. Alternatives and all kinds of disjunction obviously contribute to its increase, as well as repetition metasymbols. Conjunction, if present, also contributes to the branching since a construct like `A & B` means that both the parser and the grammar engineer will have to explore both branches related to `A` as well as to `B`.

It is interesting to consider how this smell can be eliminated: a similar "Wide Hierarchy" code smell suggests to introduce intermediate hubs for groups of nonterminals, but others blame such a solution from other points of view because there is a chance of those nonterminals to be Weak.

5.3.3 TooRecursive

If recursion and mutual recursion are too prevalent in a grammar, it can be confusing even if it is not 1SidedRecursion.

5.3.4 TooNested

Subsequences (often called *groups*) may be used to avoid Misformat, but really hamper the understanding of the system if used excessively. This smell often correlates with production rules being TooRamose.

5.3.5 TooTall

For each nonterminal we can calculate its minimal distance from the starting symbol, as the minimal number of productions in a full derivation that contains it. The maximum of all these distances for all nonterminals, is what is referred to as the height of the grammar. Out of two grammars of comparable size with respect to number of terminals, nonterminals and production rules, a taller grammar will be more complex to understand—thus, it is advisable to refactor a grammar that has grown too tall.

5.3.6 Lonely

A variant of the well-known Insufficient Modularisation smell, ported to grammars: if the size of a grammar is much larger than expected, the time has come to split it up in modules. Old-fashioned notations did not have any explicit modularisation capabilities and treated a collection of production rules as a set, but modern language workbenches have advanced frameworks with namespaces, dependence management, etc [6, 15].

5.3.7 TooModular

On the other side of the spectrum from TooLonely, a grammar can be too modular and split into so many modules that each of them is meaninglessly tiny, yet their combination is unbearably unintelligible.

5.3.8 Greedy

Similarly to TooLonely but not quite identical to it, there is a scenario when a grammar is modularised, but still insufficiently: in particular, if there is one module that is much greedier than the rest and does too much compared to any other module. In realistic grammars this smell is quite common, and the culture of proper modularisation with close to uniform distribution of responsibilities among modules, has not yet developed. The harmfulness of this smell has also never been shown, and also not been investigated properly (to the best of our knowledge).

5.3.9 Lazy

The opposite of Greedy, a Lazy module is the one that does not do much: it is empty or contains just one nonterminal.

5.3.10 TooCoupled

Modularity can not only be broken or insufficient, it can also be weakened. A grammar split into several modules that have high coupling among them and low cohesion inside each of them, had better stayed as Lonely.

6 Application

So far we have attempted to evaluate our results by relating to the existing smells, their classification and taxonomies, on one side, and to parts of previous grammar engineering projects which were directly linked to quality complaints or quality improvements, on the other[6]. However, none of this points to possible usefulness of the result. In this section we present a snapshot of the ongoing project (nearing its completion, after which it will be fully merged into GraSs) on specifying instances of the grammar smells we described above as executable metaprograms. We say "instances" because of the realisation that detecting, say, Soulmates in an API through mining the versioned repository of its definition, will look drastically different and produce vastly different results that detecting the same smell on XML Schema schemata with genetic methods and the number of co-changed nonterminals being a fitness function.

6.1 Pilot Case Setup

We choose GrammarLab [67] as the grammar handling framework because it fits our requirements and supports Boolean grammars in a broad sense of the box. The framework itself is written in Rascal [23] which is also quite one of the best tool

[6]The former is occasional in the paper due to space constraints but systematic on the GraSs project page at http://slebok.github.io/grass.

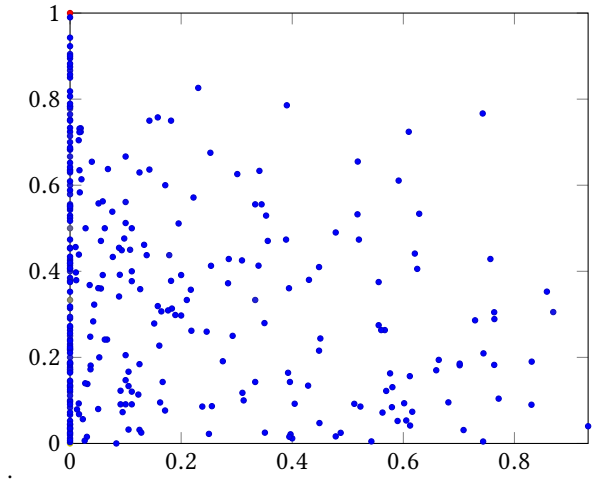

Figure 1. Fraction of Unsequenced nonterminals (Ox) vs referencing ratio (Oy) in *extracted* grammars in the corpus.

choices to express software analysis algorithms [24, 25]. As the corpus, we take the *extracted* grammars of the Grammar Zoo [64], which are the result of extraction-by-abstraction grammar recovery. It means that all grammars are in the same format, and every detail in the original extraction source that does not "fit" into this format and cannot be expressed in its notation, is removed (abstracted from): ordered and unordered choices are not distinguished, semantic actions are not modelled, etc. There are 641 of such grammars in total, that have the following distribution by the number of nonterminals (the VAR metric):

VAR	1–10	11–25	26–50	51–100	101–200	201–500	501+
#	158	125	76	46	44	39	11

Since Rascal visualisation library is not as mature as its metaprogramming core, we generated JSON files with collected data, and processed them further with JavaScript and node.js scripts. Performance of such a setup is adequate: the running time for detecting all smells in all grammars, is under 10 minutes on a good developer's laptop.

6.2 Detecting Unsequence

To detect the Unsequence smell, we define the downward reference count d to be equal to the number of nonterminal-to-nonterminal relations where at least one production rule defining a nonterminal, is located after the production rule using this nonterminal. Similarly, u is the upward reference count. If $d > u$, we call a grammar downward referencing; if $d < u$, upward referencing; and if $d = u$, even referencing. (Alternatively, a margin of proximity could have been used instead of equality). The core metric for this detection will be the referencing ratio, defined as $\frac{|d-u|}{d+u}$. Ideally, the referencing ratio should be close to 1, and is equal to 0 for even referencing grammars. However, it may be impossible to reach 1 due to mutually recursive nonterminals. To filter them out, we fine-tune the detector by allowing referencing

against the flow within grammatical levels (which is grammar engineering term for a clique, a complete subgraph in the nonterminal use graph).

A significant number of even referencing grammars are only found in the group of smallest grammars (1–10 nonterminals), where they make up 15%. In all groups, down referencing grammars are the most prevalent (67%–91%). The entire picture looks like further clustering techniques would not be effective (Figure 1).

Manual inspection of a selected subset of grammars guilty of Unsequence, shows at least the following categories of them (examples are referenced in the footnotes):

- **Perfect**[7]: no violations modulo grammatical levels; no permutation of production rules that could have increased the referencing ratio.
- **Fixable**[8]: a few violations with a very concrete solution (for the PNML example, the solution to the only violation is moving `NumberConstant` between `BuiltInConstant` and `Number`).
- **Broken by a fix**[9]: since the transition from *fetched* to *extracted* was often done by a fairly sophisticated heuristic-based recovery tool [58], the fixes it applied could cause smells (for the C# example, the original problem was the Echo of `array-type`: the extractor removed the first occurrence but it should have removed the second one).
- **Disarray**[10]: a grammar needs a complete overhaul to become consistently referencing (in the Dart example, the statement grammatical level did not correspond well technically to the documentation structure).

7 Conclusion and Future Work

With the information on smells collected from Sharma's taxonomy [46], plus with the available corpus of grammar engineering papers published by the SLE community [4, 6, 7, 10–12, 15, 22, 24–26, 28, 29, 32–34, 40, 41, 44, 45, 49–51], and occasionally with the use of general papers on software quality [2, 3, 8, 9, 17–19, 21, 35, 37, 39, 42, 43, 48, 69], we have assembled a taxonomy of grammar smells. They are defined for grammars in a broad sense in an attempt to cover the most ground and to make the results partially applicable to a wide range of domains. We see this is in a broader context as a contribution to the SLE Body of Knowledge (SLEBoK) initiative, so an important part is that the taxonomy is released in a form of a website as well: http://slebok.github.io/grass. The website will keep growing, we plan to enhance it with comfortable editing functionality, a discipline in linking examples of smells, metaprogram samples of possible detectors, etc. Besides all these obvious improvements:

- We need a structured literature review on code smells and coding convention violations. For this project we used Sharma's taxonomy [46] as a proxy, but that does not guarantee complete coverage of the field and does not link related smells together.
- We need feasibility studies of applying this taxonomy to popular language workbenches to create notation-specific smell detection toolkits.
- We need to extend the notion of a grammar smell to a language definition smell, and extend the taxonomy to cover bad type systems, database schemata, metamodels and other artefacts defining software languages.

We hope to get to all this in the future, but would be grateful if someone in the SLE community gets there faster.

Acknowledgement

The second author would like to express his gratitude to Tijs van der Storm and Jurgen Vinju of CWI/Rascal fame, for several discussions in 2013 around this topic, with a hope that this paper will form a good foundation on which to build future collaboration; as well as to Tushar Sharma for his umbrella taxonomy of all smells and discussions about it during SATToSE 2017. The technical content of this paper is based on the first author's thesis [47].

[7] Hugo Brunelière, *C# 1.0, a simplified metamodel*, extracted [64].
[8] Lom Hillah, *RELAX NG implementation of Integers grammar*, extracted [64].
[9] ISO/IEC 23270:2003(E), *Information technology — Programming languages — C# 1.0*, extracted [64].
[10] The Dart Team, *Dart Programming Language Specification, Draft Version 0.61*, extracted [64].

References

[1] Alfred V. Aho, Stephen C. Johnson, and Jeffrey D. Ullman. 1973. Deterministic Parsing of Ambiguous Grammars. In *Conference Record of the First Symposium on Principles of Programming Languages*, Patrick C. Fischer and Jeffrey D. Ullman (Eds.). ACM Press, 1–21. https://doi.org/10.1145/512927.512928

[2] Miltiadis Allamanis and Charles A. Sutton. 2014. Mining Idioms from Source Code. In *Proceedings of the 22nd Symposium on the Foundations of Software Engineering (FSE)*. ACM, 472–483. https://doi.org/10.1145/2635868.2635901

[3] Diogo Almeida, José Creissac Campos, João Saraiva, and João Carlos Silva. 2015. Towards a Catalog of Usability Smells. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing (SAC)*. ACM, 175–181. https://doi.org/10.1145/2695664.2695670

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

[5] John W. Backus. 1960. The Syntax and Semantics of the Proposed International Algebraic Language of the Zurich ACM-GAMM Conference. In *Proceedings of the International Conference on Information Processing*, S. de Picciotto (Ed.). Unesco, Paris, 125–131.

[6] Bas Basten, Jeroen van den Bos, Mark Hills, Paul Klint, Arnold Lankamp, Bert Lisser, Atze van der Ploeg, Tijs van der Storm, and Jurgen J. Vinju. 2015. Modular Language Implementation in Rascal — Experience Report. *Science of Computer Programming* 114 (2015), 7–19. https://doi.org/10.1016/j.scico.2015.11.003

[7] Bas Basten and Jurgen J. Vinju. 2011. Parse Forest Diagnostics with Dr. Ambiguity. In *Revised Selected Papers of the Fourth International Conference on Software Language Engineering (LNCS)*, Anthony M. Sloane and Uwe Aßmann (Eds.), Vol. 6940. Springer, 283–302. https://doi.org/10.1007/978-3-642-28830-2_16

[8] Feras Batarseh. 2010. Java Nano Patterns: a Set of Reusable Objects. In *Proceedings of the 48th Annual Southeast Regional Conference (SE'10)*. ACM, Article 60, 4 pages.

[9] Gal Beniamini, Sarah Gingichashvili, Alon Klein Orbach, and Dror G. Feitelson. 2017. Meaningful Identifier Names: The Case of Single-letter Variables. In *Proceedings of the 25th International Conference on Program Comprehension (ICPC'17)*. IEEE Press, 45–54. https://doi.org/10.1109/ICPC.2017.18

[10] Sergey Bratus and Meredith L. Patterson. 2012. Shotgun Parsers in the Cross-hairs. In *Brucon*. http://langsec.org.

[11] James R. Cordy. 2006. The TXL Source Transformation Language. *Science of Computer Programming* 61, 3 (2006), 190–210. https://doi.org/10.1016/j.scico.2006.04.002

[12] Merijn de Jonge and Ramin Monajemi. 2001. Cost-Effective Maintenance Tools for Proprietary Languages. In *Proceedings of the 17th International Conference on Software Maintenance (ICSM)*. IEEE Computer Society, 240–249.

[13] Merijn de Jonge and Joost Visser. 2000. Grammars as Contracts. In *Revised Papers of the Second International Symposium on Generative and Component-Based Software Engineering (LNCS)*, Vol. 2177. Springer-Verlag, 85–99. https://doi.org/10.1007/3-540-44815-2_7

[14] Thomas Roy Dean, James R. Cordy, Andrew J. Malton, and Kevin A. Schneider. 2002. Grammar Programming in TXL. In *Proceedings of the Second International Workshop on Source Code Analysis and Manipulation (SCAM)*. IEEE Computer Society, 93–102. https://doi.org/10.1109/SCAM.2002.1134109

[15] Sebastian Erdweg, Tijs van der Storm, Markus Völter, Meinte Boersma, Remi Bosman, William R. Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, Gabriël D. P. Konat, Pedro J. Molina, Martin Palatnik, Risto Pohjonen, Eugen Schindler, Klemens Schindler, Riccardo Solmi, Vlad A. Vergu, Eelco Visser, Kevin van der Vlist, Guido Wachsmuth, and Jimi van der Woning. 2013. The State of the Art in Language Workbenches — Conclusions from the Language Workbench Challenge. In *Proceedings of the Sixth International Conference on Software Language Engineering (SLE'13) (LNCS)*, Martin Erwig, Richard F. Paige, and Eric Van Wyk (Eds.), Vol. 8225. Springer, 197–217. https://doi.org/10.1007/978-3-319-02654-1_11

[16] Jean-Marie Favre. 1996. Preprocessors from an Abstract Point of View. In *Proceedings of the 12th International Conference on Software Maintenance*. IEEE Computer Society, 329–None.

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

[18] Joshua Garcia, Daniel Popescu, George Edwards, and Nenad Medvidović. 2009. Toward a Catalogue of Architectural Bad Smells. In *Proceedings of the Fifth International Conference on Quality of Software Architectures: Architectures for Adaptive Software Systems (LNCS)*, Vol. 5581. Springer, 146–162. https://doi.org/10.1007/978-3-642-02351-4_10

[19] Joseph Yossi Gil and Itay Maman. 2005. Micro Patterns in Java Code. In *Proceedings of the 20th Annual Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Ralph E. Johnson and Richard P. Gabriel (Eds.). ACM, 97–116. https://doi.org/10.1145/1094811.1094819

[20] Dick Grune and Ceriel J. H. Jacobs. 2008. *Parsing Techniques — A Practical Guide* (second ed.). Addison-Wesley. https://dickgrune.com/Books/PTAPG_2nd_Edition/

[21] Felienne Hermans, Martin Pinzger, and Arie van Deursen. 2012. Detecting Code Smells in Spreadsheet Formulas. In *Proceedings of the 28th International Conference on Software Maintenance*. IEEE Computer Society, 409–418. https://doi.org/10.1109/ICSM.2012.6405300

[22] Paul Klint, Ralf Lämmel, and Chris Verhoef. 2005. Toward an Engineering Discipline for Grammarware. *ACM Transactions on Software Engineering Methodology (ToSEM)* 14, 3 (2005), 331–380.

[23] Paul Klint, Davy Landman, Mark Hills, Jurgen Vinju, Anastasia Izmaylova, Tijs van der Storm, Atze van der Ploeg, Bert Lisser, Ali Afroozeh, Anya Helene Bagge, Vadim Zaytsev, and Ashim Shahi. 2013. Rascal 0.6.x. http://www.rascal-mpl.org. (2013).

[24] Paul Klint, Tijs van der Storm, and Jurgen J. Vinju. 2009. EASY Meta-programming with Rascal. In *Revised Papers of the Third International Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE'09) (LNCS)*, João M. Fernandes, Ralf Lämmel, Joost Visser, and João Saraiva (Eds.), Vol. 6491. Springer, 222–289. https://doi.org/10.1007/978-3-642-18023-1_6

[25] Paul Klint, Tijs van der Storm, and Jurgen J. Vinju. 2009. RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation. In *Proceedings of the Ninth International Working Conference on Source Code Analysis and Manipulation (SCAM'09)*. IEEE Computer Society, 168–177. https://doi.org/10.1109/SCAM.2009.28

[26] Jan Kort, Ralf Lämmel, and Chris Verhoef. 2002. The Grammar Deployment Kit — System Demonstration. *Electronic Notes in Theoretical Computer Science* 65, 3 (2002), 117–123. https://doi.org/10.1016/S1571-0661(04)80430-4

[27] Ivan Kurtev, Jean Bézivin, and Mehmet Akşit. 2002. Technological Spaces: an Initial Appraisal. In *Proceedings of CoopIS, DOA'2002, Industrial track*. https://research.utwente.nl/en/publications/technological-spaces-an-initial-appraisal

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

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

[30] Ralf Lämmel and Vadim Zaytsev. 2009. An Introduction to Grammar Convergence. In *Proceedings of the Seventh International Conference on*

Integrated Formal Methods (iFM 2009) (LNCS), Michael Leuschel and Heike Wehrheim (Eds.), Vol. 5423. Springer-Verlag, Berlin, Heidelberg, 246–260. https://doi.org/10.1007/978-3-642-00255-7_17

[31] Ralf Lämmel and Vadim Zaytsev. 2011. Recovering Grammar Relationships for the Java Language Specification. *Software Quality Journal (SQJ); Section on Source Code Analysis and Manipulation* 19, 2 (March 2011), 333–378. https://doi.org/10.1007/s11219-010-9116-5

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

[33] Ralf Lämmel. 2005. The Amsterdam Toolkit for Language Archaeology. *Electronic Notes in Theoretical Computer Science* 137, 3 (2005), 43–55. https://doi.org/10.1016/j.entcs.2005.07.004

[34] Ralf Lämmel and Guido Wachsmuth. 2001. Transformation of SDF syntax definitions in the ASF+SDF Meta-Environment. *Electronic Notes in Theoretical Computer Science* 44, 2 (2001), 9–33. https://doi.org/10.1016/S1571-0661(04)80918-6

[35] Douglas Martin and James R. Cordy. 2011. Analyzing Web Service Similarity Using Contextual Clones. In *Proceedings of the Fifth International Workshop on Software Clones (IWSC'11)*. ACM, 41–46. https://doi.org/10.1145/1985404.1985412

[36] Michael C. McCord. 1985. Modular Logic Grammars. In *Proceedings of the 23rd Annual Meeting on Association for Computational Linguistics (ACL)*. Association for Computational Linguistics, 104–117. https://doi.org/10.3115/981210.981223

[37] Tung Thanh Nguyen, Hoan Anh Nguyen, Nam H. Pham, Jafar M. Al-Kofahi, and Tien N. Nguyen. 2009. Graph-based mining of multiple object usage patterns. In *Proceedings of the Seventh joint meeting of the 12th European Software Engineering Conference and the 17th International Symposium on Foundations of Software Engineering*, Hans van Vliet and Valérie Issarny (Eds.). ACM, 383–392. https://doi.org/10.1145/1595696.1595767

[38] Alexander Okhotin. 2013. Conjunctive and Boolean Grammars: The True General Case of the Context-Free Grammars. *Computer Science Review* 9 (2013), 27–59. https://doi.org/10.1016/j.cosrev.2013.06.001

[39] Aniket Potdar and Emad Shihab. 2014. An Exploratory Study on Self-Admitted Technical Debt. In *Proceedings of the 30th International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 91–100. https://doi.org/10.1109/ICSME.2014.31

[40] James F. Power and Brian A. Malloy. 2000. Metric-Based Analysis of Context-Free Grammars. In *Proceedings of the Eighth International Workshop on Program Comprehension*. IEEE Computer Society, 171–178. https://doi.org/10.1109/WPC.2000.852491

[41] James F. Power and Brian A. Malloy. 2004. A Metrics Suite for Grammar-based Software. *Journal of Software Maintenance and Evolution* 16 (Nov 2004), 405–426. Issue 6. https://doi.org/10.1002/smr.v16:6

[42] Juergen Rilling and Tuomas Klemola. 2003. Identifying Comprehension Bottlenecks Using Program Slicing and Cognitive Complexity Metric. In *Proceedings of the 11th International Workshop on Program Comprehension*. IEEE Computer Society, 115–124. https://doi.org/10.1109/WPC.2003.1199195

[43] Chanchal Kumar Roy, James R. Cordy, and Rainer Koschke. 2009. Comparison and Evaluation of Code Clone Detection Techniques and Tools: A Qualitative Approach. *SCP* 74, 7 (2009), 470–495. https://doi.org/10.1016/j.scico.2009.02.007

[44] Len Sassaman, Meredith L. Patterson, and Sergey Bratus. 2012. A Patch for Postel's Robustness Principle. *IEEE Security and Privacy* 10, 2 (March 2012), 87–91. https://doi.org/10.1109/MSP.2012.31

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

[46] Tushar Sharma. 2017. A Taxonomy of Software Smells. http://tusharma.in/smells/. (2017).

[47] Mats Stijlaart. 2017. *Towards a Catalogue of Grammar Smells*. Master's thesis. Universiteit van Amsterdam, Amsterdam, The Netherlands.

[48] Girish Suryanarayana, Ganesh Samarthyam, and Tushar Sharma. 2014. *Refactoring for Software Design Smells: Managing Technical Debt*. Morgan Kaufmann.

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

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

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

[52] Jurgen J. Vinju and Michael W. Godfrey. 2012. What Does Control Flow Really Look Like? Eyeballing the Cyclomatic Complexity Metric. In *Proceedings of the 12th International Working Conference on Source Code Analysis and Manipulation*. IEEE Computer Society, 154–163. https://doi.org/10.1109/SCAM.2012.17

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

[54] Niklaus Wirth. 1977. What Can We Do about the Unnecessary Diversity of Notation for Syntactic Definitions? *Communications of the ACM* 20 (Nov. 1977), 822–823. Issue 11. https://doi.org/10.1145/359863.359883

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

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

[57] Vadim Zaytsev. 2012. BNF WAS HERE: What Have We Done About the Unnecessary Diversity of Notation for Syntactic Definitions. In *Programming Languages Track, Volume II of the Proceedings of the 27th ACM Symposium on Applied Computing (SAC 2012)*, Sascha Ossowski and Paola Lecca (Eds.). ACM, Riva del Garda, Trento, Italy, 1910–1915. https://doi.org/10.1145/2245276.2232090

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

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

[60] Vadim Zaytsev. 2013. Micropatterns in Grammars. In *Proceedings of the Sixth International Conference on Software Language Engineering (SLE 2013) (LNCS)*, Martin Erwig, Richard F. Paige, and Eric Van Wyk (Eds.), Vol. 8225. Springer, Switzerland, 117–136. https://doi.org/10.1007/978-3-319-02654-1_7

[61] Vadim Zaytsev. 2014. Formal Foundations for Semi-parsing. In *Proceedings of the Software Evolution Week (IEEE Conference on Software Maintenance, Reengineering and Reverse Engineering), Early Research Achievements Track (CSMR-WCRE 2014 ERA)*, Serge Demeyer, Dave Binkley, and Filippo Ricca (Eds.). IEEE, 313–317. https://doi.org/10.1109/CSMR-WCRE.2014.6747184

[62] Vadim Zaytsev. 2014. Software Language Engineering by Intentional Rewriting. *Electronic Communications of the EASST; Software Quality and Maintainability* 65 (March 2014). https://doi.org/10.14279/tuj.eceasst.0.903

[63] Vadim Zaytsev. 2015. Grammar Maturity Model. In *Post-proceedings of the Ninth Workshop on Models and Evolution (ME'14) (CEUR Workshop Proceedings)*, Alfonso Pierantonio, Bernhard Schätz, and Dalila Tamzalit (Eds.), Vol. 1331. CEUR-WS.org, 42–51. http://ceur-ws.org/Vol-1331/p5.pdf

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

[65] Vadim Zaytsev. 2015. Guided Grammar Convergence. In *Poster proceedings of the Sixth International Conference on Software Language Engineering (SLE 2013)*. https://arxiv.org/abs/1503.08476

[66] Vadim Zaytsev. 2015. Taxonomy of Flexible Linguistic Commitments. In *Workshop on Flexible Model-Driven Engineering (FlexMDE) (CEUR Workshop Proceedings)*, Davide Di Ruscio, Juan De Lara, and Alfonso Pierantonio (Eds.), Vol. 1470. CEUR-WS.org. http://ceur-ws.org/Vol-1470/FlexMDE15_paper_7.pdf

[67] V. Zaytsev et al. 2013. GrammarLab. Software. http://grammarware.github.io/lab. (2013).

[68] Vadim Zaytsev and Anya Helene Bagge. 2014. Parsing in a Broad Sense. In *Proceedings of the 17th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2014) (LNCS)*, Jürgen Dingel, Wolfram Schulte, Isidro Ramos, Silvia Abrahão, and Emilio Insfran (Eds.), Vol. 8767. Springer, 50–67. https://doi.org/10.1007/978-3-319-11653-2_4

[69] Hao Zhong, Tao Xie, Lu Zhang, Jian Pei, and Hong Mei. 2009. MAPO: Mining and Recommending API Usage Patterns. In *Proceedings of the 23rd European Conference on Object-Oriented Programming (ECOOP) (LNCS)*, Sophia Drossopoulou (Ed.), Vol. 5653. Springer, 318–343. https://doi.org/10.1007/978-3-642-03013-0_15