Micropatterns in Grammars

Vadim Zaytsev, vadim@grammarware.net

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

Abstract. Micropatterns and nanopatterns have been previously demonstrated to be useful techniques for object-oriented program comprehension. In this paper, we use a similar approach for identifying structurally similar fragments in grammars in a broad sense (contracts for commitment to structure), in particular parser specifications, metamodels and data models. Grammatical micropatterns bridge the gap between grammar metrics, which are easy to implement but hard to assign meaning to, and language design guidelines, which are inherently meaningful as stemming from current software language engineering practice but considerably harder to formalise.

1 Introduction

Micropatterns are mechanically recognisable pieces of design that reside on a significantly lower level than design patterns, hence being closer to the implementation than to an abstract domain model, while still representing design steps and decisions [14]. They have been proposed in 2005 by Gil and Maman as a method of comparing software systems programmed in the object-oriented paradigm — the original paper concerned Java as the base language for its experiments, but the presence of similar classification methods for considerably different languages like Smalltalk [26] leads us to believe that the approach is applicable to any object-oriented programming language at the least. In this paper, we investigate whether micropatterns can become a useful tool for grammarware.

Grammatical micropatterns are similar in many aspects to the OOP micropatterns, in particular in (cf. [14, §4]):

- **Recognisability.** For any micropattern, we can construct an algorithm that recognises if the given grammar matches its condition. Our approach toward this property is straightforward: we implement all micropattern recognisers in Rascal [21] and expose them at the public open source code repository [42]. Unlike design patterns, there are no two micropatterns with the same structure.
- **Purposefulness.** Even though there are infinitely many possible micropatterns ("name starts with A", "number of terminals is a prime number", "uses nonterminals in alphabetical order", etc), we collect only those which intent can be reverse engineered and clearly identified ("name starts with

uppercase" — because the metalanguage demands it; "no terminals used" — because it defines abstract syntax; etc).

- **Prevalence** is the fraction of nonterminals that satisfy the micropattern condition. It is a property that strengthens the purposefulness, showing whether the condition happens in practice and if so, how often. We tend to ignore micropatterns with zero prevalence or with prevalence greater than 50 %, with a few notable exceptions.
- **Simplicity** is a requirement that stops us from concocting overcomplicated micropatterns like "uses a nonterminal that is not used in the rest of the grammar", even if they are useful. Mostly we pursued two forms of micropattern conditions: ones that can be formulated with a single pattern matching clause, and ones that assert one simple condition over all its children. (When inspecting the implementation, one can notice multiline definitions as well, which are only made so for readability and maintainability purposes, and utilise advanced Rascal techniques like pattern-driven dispatch).
- **Scope.** Each micropattern concerns one nonterminal symbol, and can be automatically identified based on the production rules of that nonterminal symbol. It does not have to bear any information about how this nonterminal is used or what the real intent was behind its design.
- **Empirical evidence.** The micropatterns from our catalogue are validated against a corpus of grammars in a broad sense. Even if the corpus is not curated and not balanced to yield statistically meaningful results, we have a stronger claim of evidential usage of micropatterns in the practice of grammarware engineering than any software language design patterns or guidelines might have (simply because their claims rely on manual harvest).

However, there are some notable differences in our work:

- **Usability of isolated micropatterns.** One of the distinctive feature of micropatterns versus design patterns and implementation patterns pointed out by Gil and Maman in [14, §4.2], was that a single micropattern is not useful on its own, and only the entire catalogue is a worthy instrument. However, as we found out, isolated micropatterns (single ones and small subsets of the catalogue) can also be useful indicators of grammar properties, triggers for grammar mutations, assertions of technical compatibility, etc.
- **Coverage** is measured as a combined prevalence of a thematic group of micropatterns (it is not equal to the sum of their prevalences, since micropatterns in most groups are not mutually exclusive) and computed separately for each group. For OOP micropatterns, coverage was calculated for the whole catalogue, but per system: we do it the other way around, to emphasize conceptual gaps between groups and to avoid issues with a non-curated corpus. For groups with low coverage we also report on *frequency*, which is prevalence within the group.
- **Grammar mining** is much less popular than software mining and data mining [40], and hence the fact that we derived our catalogue my mining a repository of versatile grammars, is a unique contribution in that sense.

2 Grammar corpus

Grammar Zoo and Grammar Tank are twin repositories that together aim at collecting grammars in a broad sense (per [20]) from various sources: abstract and concrete, large and small, typical and peculiar [40]. Technically and historically, they are a part of the larger initiative titled Software Language Processing Suite (SLPS) and available as a publicly accessible repository online since 2008 [42]. The SLPS project also includes experiments and tools relating to the activities of grammar extraction, recovery, documentation, convergence, maintenance, deployment, transformation, mutation, migration, testing, etc.

The conceptual difference between the two sibling collections is that Grammar Zoo is meant to display big beasts occurring in real life, while Grammar Tank collects flocks of smaller prey which quite often cannot tell the users much on their own. The border between them is not clearly defined, and so for the purpose of this paper we will simply refer to the entire collection of grammars as "the corpus" or "the Zoo". Contrary to prior practice, we will also not include pointers to individual sources per grammar in the paper, plainly due to sheer impossibility of delivering over 500 bibliographic references. An interested reader is referred to the frontend of the Zoo at `http://slps.github.io/zoo` and `http://slps.github.io/tank` to inspect any of the grammars or all of them, together with the metadata concerning their authors, original publication dates, extraction and recovery methods and other details properly structured and presented there.

The corpus mainly consists of the following kinds of grammars:

- grammars extracted from parser specifications composed by students
  - for example, 32 TESCOL grammars were used in [9] for grammar testing
- grammars extracted from language documents
  - standardisation bodies like ISO, ECMA, W3C, OMG publish standards [41] that are possible to process with notation-parametric grammar recovery methodology [37]
- grammars extracted from document schemata
  - for example, XML Schema and RELAX NG definitions of MathML, SVG, DocBook are available and ready to be researched and compared to definitions of the same languages with other technologies like Ecore
- grammars extracted from metamodels
  - the entire Atlantic metamodel zoo[1] is imported into Grammar Zoo by reusing their Ecore metamodel variants with our extractor
- grammars extracted from concrete syntax specs
  - for example, the ASF+SDF Meta-Environment and the TXL framework have their own repositories for concrete grammars, which have been extracted and added to the Grammar Zoo
- grammars extracted from DSL grammars in a versioning system (BGF)

[1] AtlantEcore Zoo: `http://www.emn.fr/z-info/atlanmod/index.php/Ecore`.

- various DSL were spawned by the SLPS itself during its development: they are not interesting on their own, but the presence of many versions of the same grammar is a rare treasure; for example, there are 35 version available of the unified format for language documents from [41].

With 121 grammars in the Grammar Zoo and 412 in the Grammar Tank[2], they are the biggest collection of grammars in a broad sense; the grammars are obtained from heterogeneous sources; they are all properly documented, attributed to their creators and annotated with the data available about their extraction process — the combination of these three factors may set the Zoo apart from its competitors [40], yet it does not make it perfect.

We could not emphasize strong enough that empirical investigation is not the primary contribution of this paper. All presented evidence about prevalence of proposed micropatterns serves as a mere demonstration that they indeed occur in practice. Our grammar corpus consists of as many grammars as we could secure, obtained by different means from heterogeneous sources, and we calculate prevalence and coverage as an estimate of ever encountering the same micropatterns in other real life grammars, not as a prediction of the probability of that. At this point, it is not yet feasible to construct a representative versatile corpus of grammars: even though Grammar Zoo is the largest of its kind, it does not have enough content to claim any kind of balance between different technologies, grammar sizes, quality levels, etc. However, this effort is an ongoing work.

3 Grammatical micropatterns

The process of obtaining the micropatterns catalogue is identical to the one undertaken by Gil and Maman [14], and we will spare the space on its details. In short, all possible combinations of metaconstructs were considered and tried on a corpus of grammars; those with no matches were either abandoned or kept purely for symmetrical considerations; the intent behind each of them was manually investigated, leading to naming a micropattern properly; and finally the named micropattern was connected to its context by pointing out key publications related to it.

3.1 Metasyntax

It has been shown before [36] that many metalanguages existing for context-free grammars, commonly referred to as BNF dialects or "Extended Backus-Naur Forms"[3], can be specified by a small set of indicators for their metasymbols,

[2] Counted at the day of paper submission: the actual website may contain more.

[3] By "the EBNF", people usually mean the most influential extended variant of BNF, proposed in 1977 by Wirth [34] as a part of his work on Wirth Syntax Notation. However, almost each of the metalanguages used in language documentation ever since, uses its own concrete notation, which sometimes differs even in expressivity from Wirth's proposal — see [36] for more details.

Category	Pattern	Matches	Prevalence
Metasyntax	ContainsEpsilon	4,185	10.20%
	ContainsFailure	69	0.17%
	ContainsUniversal	825	2.01%
	ContainsString	1,889	4.60%
	ContainsInteger	343	0.84%
	ContainsOptional	6,554	15.97%
	ContainsPlus	4,586	11.18%
	ContainsStar	3,080	7.51%
	ContainsSepListPlus	55	0.13%
	ContainsSepListStar	142	0.35%
	ContainsDisjunction	2,804	6.83%
	ContainsSelectors	17,328	42.22%
	ContainsLabels	132	0.32%
	ContainsSequence	19,447	47.39%
	AbstractSyntax	29,299	71.39%
	Total coverage	**36,522**	**89.00%**

Table 1. Metasyntax micropatterns

which correspond both to the "grammar for grammars" and to human-perceived aspects like "do we quote terminals in this notation?" or "how do we write down multiple production rules for one nonterminal?".

For every feature of the internal representation of a grammar in a broad sense, we define a ContainsX micropattern, where X is that feature:

- ContainsEpsilon for the empty string metaconstruct (ε),
- ContainsFailure for the empty language metaconstruct (φ),
- ContainsUniversal for the universal metaconstruct (α),
- ContainsString for a built-in string value,
- ContainsInteger for a built-in integer value,
- ContainsOptional for an optionality metasymbol,
- ContainsPlus for the transitive closure,
- ContainsStar for the Kleene star,
- ContainsSepListPlus for a separator list with one or more elements,
- ContainsSepListStar for a separator list with zero or more elements,
- ContainsDisjunction for inner choice metasymbol,
- ContainsSelectors for named subexpressions,
- ContainsLabels for production labels,
- ContainsSequence for sequential composition metaconstruct.

Category	Pattern	Matches	Prevalence
Global	Root	563	1.37%
	Leaf	9,467	23.07%
	Top	3,245	7.91%
	MultiRoot	1	0.002%
	Bottom	1,311	3.19%
	Total coverage	**12,459**	**30.36%**
Structure	Disallowed	69	0.17%
	Singleton	29,134	70.99%
	Vertical	3,697	9.01%
	Horizontal	6,043	14.73%
	ZigZag	784	1.91%
	Total coverage	**39,727**	**96.81%**

Table 2. Global position micropatterns

Furthermore, we add one extra micropattern AbstractSyntax for nonterminals which definitions do not contain terminal symbols — mainly because investigations of abstract data types and abstract syntax vs concrete syntax [32] form a valuable subdomain of grammarware research. As can be observed on Table 1, the prevalence of AbstractSyntax is quite high, which can be explained by many Ecore metamodels and XML Schema schemata in our corpus.

3.2 Global position and structure

Since the very beginning of grammar research, even when grammars were still considered as structural string rewriting systems and not as commitments to structure, the was a need to denote the initial state for rewriting [5, §4.2]. Such an initial state was quickly agreed to be specified with a *starting symbol*, or a *grammar root* — the nonterminal symbol that initiates the generation, or a root of a parse tree. Not being able to overlook this, we say that a nonterminal exercises the Root micropattern, when it is explicitly marked as a root of its grammar. Contrariwise, we define the Leaf micropattern for nonterminals that do not refer to any other nonterminals — they are the leaves of the nonterminal connectivity graph, not of the parse tree.

In some frameworks, the roots are not specified explicitly: either because such metafunctionality is lacking (such as in pure BNF), or because the information was simply lost during engineering or knowledge extraction. For such cases, found quite often in grammar recovery research, we could speak of the Top micropattern, named after "top sorts" from [23, p.19] and "top nonterminals" from [22, §2.2], which are nonterminals defined by the grammar, but never used. A previously existing heuristic technique in semi-automated interactive grammar

adaptation, reported rather reliable, is to establish missing connections to all top nonterminals, until only one non-leaf top remains, and assume it to be the true root [22]. Methods such as this would become much easier to explain in terms of micropatterns and relations between them.

In practical grammarware engineering, grammars are commonly allowed to have multiple starting symbols, while most publications about formal languages use a representation with a single root. The reason behind this is simple: one can always imagine adding another nonterminal that becomes a new starting symbol, defined with a choice of all nonterminals that are the "real" starting symbols. Hence, we define a MultiRoot micropattern for catching such definitions explicitly encoded. Surprisingly, it was not very popular: only one match in the whole Grammar Zoo. However, if we were to investigate an XML-based framework that relied heavily on the fact that each element defined by an XSD is allowed to be the root, then such information can be decided to be propagated by the `xsd2bgf` grammar extractor, which would then lead to all grammars extracted from XML Schema schemata, to have one MultiRoot nonterminal each. The current implementation of the `xsd2bgf` grammar extractor leaves the roots unspecified, since it is hardly an intent of every XMLware developer to explicitly rely on such diversity.

Complementary to Top, we propose the Bottom micropattern, which is exhibited by a nonterminal that is used in a grammar but never defined — again, we adopt these terminology from [22,23]. Usually in the same context another property of a nonterminal is tested, called "fresh" [24, §3.4], for nonterminals that are not present in the grammar in any way, but this property does not convert well into a micropattern for obvious reasons.

For each nonterminal that is not bottom, there are only four possible ways that it can be defined, and so we make four micropatterns from them: Disallowed (defined by an empty language[4]), Singleton (defined with a single production rule), Vertical (defined with multiple production rules) and Horizontal (defined with one production rule that consist of a top level choice with alternatives). We also introduce a separate ZigZag micropattern for definitions that are both horizontal and vertical (multiple production rules, with at least one of them having a top level choice). These five micropatterns together with Bottom are mutually exclusive and together always cover 100 % of any set of nonterminals, and for the Zoo it can be seen on Table 2. The terms "horizontal" and "vertical" are borrowed from the XBGF grammar transformation framework and publications related to it [25, §4.1], other sources also relate to them as "flat" and "non-flat" [24].

As for the global position micropatterns, unsurprisingly, most of nonterminals do not belong to any of these classes, and this group of micropatterns has a meager total coverage of 30.36 % (Table 2). As an example of how Top and

[4] NB: an empty language should not be confused with an empty string/term language. The former means $L(G) = \varnothing$ and means unconditional failure of parsing and impossibility of generation. The latter means $L(G) = \varepsilon$ and means successful parsing of an empty string (or a trivial term) and immediate successful halting of generation.

Category	Pattern	Matches	Prevalence	Frequency
Sugar	FakeOptional	134	0.33%	10.89%
	FakeSepList	624	1.52%	50.69%
	ExprMidLayer	349	0.85%	28.35%
	ExprLowLayer	39	0.10%	3.17%
	YaccifiedPlusLeft	354	0.86%	28.76%
	YaccifiedPlusRight	6	0.01%	0.49%
	YaccifiedStarLeft	0	0.00%	0.00%
	YaccifiedStarRight	0	0.00%	0.00%
	Total coverage	**1,231**	**3.00%**	

Table 3. Sugary micropatterns

Bottom micropatterns encapsulate grammar quality and design intent, we quote Lämmel and Verhoef [23, p.20]:

> In the ideal situation, there are only a few top sorts, preferably one corresponding to the start symbol of the grammar, and the bottom sorts are exactly the sorts that need to be defined lexically.

In the scope of disciplined grammar transformation [25], a ZigZag nonterminal could also be considered a bad style of grammar engineering, but we have no evidence of what dangers it brings along, only an observation of its surprisingly high prevalence.

3.3 Metasyntactic sugar

There are several micropatterns that are conceptually similar to those from the previous section, but without the metafunctionality explicitly present in the metalanguage. When a particular metaconstruct is available in the metalanguage, we can check its use, as we have done in subsection 3.1; when it is not a part of the metalanguage, we can still check if any usual substitute for it, is used. For example, the optionality metasymbol is in fact metasyntactic sugar for "this or nothing" — i.e., a choice with one alternative representing the empty language (ε). We call such explicit encodings FakeOptionals (see Table 3), they mostly indeed found occurring in grammars extracted from technical spaces that lack the optionality metasymbol. Similarly, a FakeSepList micropattern explicitly encodes a separator list, and its prevalence is much higher since there are more metalanguages without separator list metasymbols.

For all metalanguages that do not allow to specify expression priorities explicitly, there exists a commonly used implementation pattern:

```
logical-or-expression ::= logical-and-expression
 | logical-or-expression "||" logical-and-expression ;
logical-and-expression ::= inclusive-or-expression
 | logical-and-expression "&&" inclusive-or-expression ;
                    ... (12 layers skipped) ...
primary-expression ::= literal | "this"
 | "(" expression ")" | id-expression ;
```

(ISO/IEC 14882:1998(E) C++)

Based on multiple occurrences of such an *implementation pattern* in the Grammar Zoo, we have designed the following two *micropatterns*:

- ExprMidLayer: one alternative is a nonterminal, the others are sequences of a nonterminal, a terminal and another nonterminal;
- ExprLowLayer: one alternative is a sequence of a terminal, a nonterminal and another terminal, where the two terminals form a symmetric bracketing pair, the others are solitary terminals or solitary nonterminals.

As one can see, these micropatterns are defined locally and do not enforce any complicated constraints (e.g., concerning the nonterminal between brackets in ExprLowLayer), which could possibly result in false positives, but satisfies our requirements from section 1.

Similarly, we can look for "yaccified" definitions that emulate repetition metasymbols with recursive patterns. A yaccified definition [18,22] is named after YACC [17], a compiler compiler, the old versions of which required explicitly defined recursive nonterminals. Instead of writing:

```
X ::= Y+ ;
```

one would write:

```
X ::= Y ;
X ::= X Y ;
```

because in LALR parsers like YACC, left recursion was preferred to right recursion (contrary to recursive descent parsers, which are unable to process left recursion directly at all). The use of metalanguage constructs `X+` and `X*` is technology-agnostic, and the compiler compiler can make its own decisions about the particular way of implementation, and will neither crash nor have to perform any transformations behind the scenes. However, as can be seen from Table 3, many existing grammars contain yaccified definitions, and usually the first step in any project that attempts to reuse such grammars for practical purposes, starts with deyaccification [22,25,35, etc].

3.4 Naming

Research on naming conventions has enjoyed a lot of interest in the scopes of program analysis and comprehension [4] and code refactorings that recommend

Category	Pattern	Matches	Prevalence
Naming	CamelCase	16704	40.70%
	LowerCase	3323	8.10%
	MixedCase	1706	4.16%
	MultiWord	31816	77.53%
	UpperCase	2073	5.05%
	Total coverage	**40,562**	**98.84%**
Naming, lax	CamelCaseLax	18332	44.67%
	LowerCaseLax	17840	43.47%
	MixedCaseLax	1969	4.80%
	MultiWordLax	32290	78.68%
	UpperCaseLax	2412	5.88%
	Total coverage	**41,038**	**100.00%**

Table 4. Naming micropatterns

renaming misspelt, synonymous and inaccurate variable names [29]. Naming conventions have not yet been thoroughly investigated in grammarware engineering, but were noted to be useful to consider as a part of metalanguage for notation-parametric grammar recovery [37] and were used as motivation for some automated grammar mutations [38], usually preceding unparsing a grammar in a specific metalanguage. In the scope of grammar recovery, mismatches like `digit` vs `DIGIT` or `newline` vs `NewLine` were reported as common in recovering grammars with community-created fragments [35].

Let us distinguish four naming conventions to be recognised by micropatterns, namely: CamelCase (LikeThis), MixedCase (almostTheSame), LowerCase (apparentlyso) and UpperCase (OBVIOUSLY). Given that most of current research on naming conventions in software engineering focuses on tokenisation and disabbreviation, we add one more micropattern called MultiWord. A nonterminal conforms to MultiWord, when its name is either written in camelcase or mixed case and has two or more words; or when its name consists of letter subsequences separated by a space, a dash, a slash, a dot or an underscore, — in other words, when its name can be easily tokenised without any dictionary-based heuristics nor heavy machine learning. Something akin to a SingleWord micropattern would have been useful as well, but we failed to obtain a reasonable definition for it: a single mixed case word name is indistinguishable from a single lower case word; both lower case and upper case names may have no word delimiters; a single word camelcase name could in fact also be a multi word capitalised name; etc.

By looking at the top half of Table 4, one quickly realises that the constraints for naming notations could be formulated in a more relaxed way. The nonterminal `Express_metamodel::Core::GeneralARRAYType` from the EXPRESS meta-

model is a nice example of an unclassifiable nonterminal name: it combines four capitalised words, one lowercase and one uppercase one, with three different kinds of concatenation (by an underscore, double colons and an empty separator). Arguably, though, its name can be considered CamelCase, with underscore being a "neutral letter" and word boundaries being either empty or "`::`". Hence, we define a set of five more lax naming convention micropatterns, that together easily cover the whole corpus by using "neutral letters" (underscores and numbers) and being more tolerant with separators.

In particular, one could notice a remarkably high prevalence of MultiWord micropatterns, both strict and lax. These micropatterns have no directly noticeable use right away, but can become a central part of future research on mining and tokenising nonterminal symbol names in grammars.

3.5 Concrete syntax

We inherit the term Preterminal from the natural language processing field, where it is used for syntactic categories of the words of the language. Preterminals are the immediate parents of the leaves of the parse tree, and usually define keywords of the language, identifier names, etc. Prevalence of the Preterminal micropattern is impressively high in our corpus — 7.92 % — despite the fact that more then half of its grammars have been extracted from metamodels and thus contain few or no terminal symbols at all. This can be explained by many concrete syntax definitions and parser specifications in the corpus as well — in particular, the common practice in ANTLR is to wrap every terminal symbol in a separate nonterminal with an uppercased name, so the prevalence of the Preterminal micropattern in such grammars can climb up to 46.9 % for big languages (Java 5 grammar by Dieter Habelitz) and up to 71.19 % for small ones (TESCOL grammar 10000).

Mining concrete grammars from the corpus led us to discover several steadily occurring patterns of terminal usage (all subcases of the Preterminal micropattern, reported on Table 5):

- Keyword: defined with one production rule, which right hand side is an alphanumeric word:

  ```
  non_end_of_line_character ::= "character" ;
  ```
  (LNCS 4348, Ada 2005)
  ```
  Retry ::= "retry" ;
  ```
  (ISO/IEC 25436:2006(E) Eiffel)
  ```
  this-access ::= "this" ;
  ```
  (Microsoft C# 3.0)
- Keywords: a horizontal or vertical (recall subsection 3.2) definition with all alternatives being keywords:

  ```
  ConstructorModifier ::= "public" ;
  ConstructorModifier ::= "private" ;
  ConstructorModifier ::= "protected" ;
  ```
  (JLS Second Edition, readable Java grammar)

```
exit_qualifier ::= ("__exit" | "exit__" | "exit" | "__exit__") ;
```
(TXL C Basis Grammar 5.2)

– Operator: defined with one production rule, which right hand side is a strictly non-alphanumeric word:

```
formal_discrete_type_definition ::= "(<>)" ;
```
(Magnus Kempe Ada 95)

```
right-shift-assignment ::= ">>=" ;
```
(Microsoft C# 4.0)

```
empty-statement ::= ";" ;
```
(ECMA-334 C# 1.0)

– Operators: a horizontal or vertical definition with all alternatives being operators:

```
relational_operator ::= ("=" | "/=" | "<" | "<=" | ">" | ">=") ;
```
(Lämmel-Verhoef Ada 95)

```
PostfixOp ::= "++" ;
PostfixOp ::= "--" ;
```
(JLS Third Edition Java, implementable)

```
equalityOperator ::= ("==" | "!=" | "===" | "!==") ;
```
(Google Dart 0.01)

– OperatorsMixed: a horizontal or vertical definition with some alternatives being operators and some being keywords:

```
typeModifier ::= ("opt" | "repeat" | "list" | "attr" | "see" | "not"
 | "push" | "pop" | ":" | "~" | ">" | "<") ;
```
(TXL Basis Grammar for TXL 10.5)

```
op ::= (">" | "<" | "<=" | ">=" | "<>" | "=" | "in" | "is" | "+" | "-"
 | "or" | "xor" | "*" | "/" | "div" | "mod" | "and" | "shl" | "shr"
 | "DIV" | "AND") ;
```
(TXL Basis Grammar for Borland Delphi Object Pascal 1.1)

```
overloadable_unary_operator ::= ("+" | "-" | "!" | "~" | "++" | "--"
 | "true" | "false") ;
```
(Validated TXL Basis Grammar for C# Edition 3)

– Words: a sequential and/or repetitive composition of keywords:

```
simpleDerivationSet ::= "#all" | ("list" | "union" | "restriction")*
```
(RELAX NG schema for XML Schema)

```
mml.lines.datatype ::= ("none" | "solid" | "dashed")+
```
(TESCOL 10001)

– Tokens: a sequential and/or repetitive composition of nontrivial non-keywords:

```
WS ::= (" " | "\t" | "\r" | "\n")+ ;
```
(TESCOL 10100)

– Modifiers: a horizontal or vertical definition with all alternatives being combinations of same keywords:

```
mode ::= ("in"? | ("in" "out") | "out") ;
```
(LNCS 4348, Ada 2005)

```
static_constructor_modifiers ::=
 (("extern"? "static") | ("static" "extern"?)) ;
```
(Validated TXL Basis Grammar for C# 3)

Category	Pattern	Matches	Prevalence	Frequency
Concrete	Preterminal	3249	7.92%	100.00%
	Keyword	906	2.21%	27.89%
	Keywords	1774	4.32%	54.60%
	Operator	1001	2.44%	30.81%
	Operators	1190	2.90%	36.63%
	OperatorsMixed	110	0.27%	3.39%
	Words	40	0.10%	1.23%
	Tokens	34	0.08%	1.05%
	Modifiers	19	0.05%	0.58%
	Range	730	1.78%	22.47%
	NumericLiteral	51	0.12%	1.57%
	LiteralSimple	15	0.04%	0.46%
	LiteralFirstRest	62	0.15%	1.91%
	EmptyStatement	30	0.07%	0.92%
	Total coverage	**3,249**	**7.92%**	

Table 5. Concrete syntax micropatterns

- Range: a choice of trivial terminals:

```
Integer_base_letter ::= ("b" | "c" | "x" | "B" | "C" | "X") ;
```

(ISO/IEC 25436:2006(E) Eiffel)

```
DIGIT ::= ("0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9") ;
```

(ANTLR Google Dart)

- NumericLiteral: a possibly signed repetition of choice of digits:

```
HEX_DIGIT ::= ("0" | "1" | ... | "9" | "A" | ... | "F" | "a" | ... | "f") ;
```

(Michael Studman Java 5)

```
INT ::= ("+" | "-")? ("0" | (("1" | ... | "9") ("0" | "1" | ... | "9")*)) ;
```

(TESCOL 00011)

- LiteralSimple: a repetition of a range of trivial terminals:

```
[NT-Digits] Digits ::= ("0" | "1" | "2" | "3" | ... | "8" | "9")+ ;
```

(W3C XPath 1.0)

- LiteralFirstRest: a choice of terminals followed by a Kleene star over a choice of terminals:

```
IDENT ::= ("a" | ... | "z" | "A" | ... | "Z" | "_" | "$")
 ("a" | ... | "z" | "A" | ... | "Z" | "_" | "0" | ... | "9" | "$")* ;
```

(Michael Studman Java 5)

```
VARID ::= ("A" | ... | "Z" | "a" | ... | "z")
 ("A" | ... | "Z" | "a" | ... | "z" | "0" | ... | "9" | "_")* ;
```

(TESCOL 10110)

- EmptyStatement: a keyword followed by a semicolon:

```
terminate_alternative ::= "terminate" ";" ;
null_statement ::= "null" ";" ;
```

(ISO/IEC 8652/1995(E) LNCS 2219 Ada 95)

Category	Pattern	Matches	Prevalence
Normal	CNF	5,365	13.07%
	GNF	3,074	7.49%
	ANF	26,269	64.01%
	Total coverage	**28,168**	**68.64%**

Table 6. Normal form micropatterns

```
continue-statement ::= "continue" ";" ;
break-statement ::= "break" ";" ;
```

(ISO/IEC 23270:2003(E) C# 1.0)

3.6 Normal forms

A lot can be said about normal forms in formal grammar theory, and in the context of micropatterns we can also view normal forms as conditions on nonterminals and their definitions. In particular, we have implemented Chomsky Normal Form, CNF [6]; Greibach Normal Form, GNF [15]; and Abstract Normal Form, ANF [39], as micropatterns — unsurprisingly, the prevalence of ANF is rather high due to many abstract syntax definitions in the corpus (Table 6).

3.7 Folding/unfolding

Not all nonterminals are introduced to the grammar because of the impossibility to express the same language differently: many are simply results of folding/unfolding transformations on a minimal grammar, and are meant to improve readability, maintainability or modularity of the language definition. In this group we collected micropatterns for nonterminals that can be removed from the grammar with relative ease (examples are given only for less intuitive micropatterns):

- Empty: a nonterminal is defined as an empty term;
- Failure: a nonterminal is explicitly undefined or prohibited;
- JustOptional: a nonterminal defined only with just an optional reference to another nonterminal;
- JustPlus: a one-or-more repetition of a reference to another nonterminal;
- JustStar: a zero-or-more repetition of a reference to another nonterminal;
- JustSepListPlus: a non-empty separator list;
- JustSepListStar: a possibly empty separator list;
- JustChains: a nonterminal defined only with chain production rules (right hand sides are nonterminals);
- JustOneChain: a nonterminal defined only with exactly one chain production rule (right hand side is a nonterminal);
- ReflexiveChain: a nonterminal is circularly defined as itself (should only happen as an intermediate transformation result);

Category	Pattern	Matches	Prevalence	Frequency
Folding	Empty	3,028	7.38%	32.56%
	Failure	69	0.17%	0.74%
	JustOptional	48	0.12%	0.52%
	JustPlus	199	0.48%	2.14%
	JustStar	130	0.32%	1.40%
	JustSepListPlus	28	0.07%	0.30%
	JustSepListStar	32	0.08%	0.34%
	JustChains	1,045	2.55%	11.24%
	JustOneChain	2,065	5.03%	22.20%
	ReflexiveChain	0	0.00%	0.00%
	ChainOrTerminal	145	0.35%	1.56%
	ChainsAndTerminals	290	0.71%	3.12%
	Total coverage	**9,300**	**22.66%**	

Table 7. Folding/unfolding micropatterns

– ChainOrTerminal: a choice of a nonterminal and a terminal:
```
return-type ::= (type | "void") ;
```
(Microsoft C# 4.0)
– ChainsAndTerminals: a choice where the all alternatives are either isolated nonterminals or isolated terminals:
```
class-type ::= (type-name | "object" | "dynamic" | "string") ;
```
(Microsoft C# 4.0)
```
TypeDeclaration ::= ClassDeclaration ;
TypeDeclaration ::= ";" ;
TypeDeclaration ::= InterfaceDeclaration ;
```
(JLS Second Edition Java, readable)
– AChain: one production rule for the nonterminal is a chain production rule.

Table 7 summarises the prevalence observations of these micropatterns. The ChainsAndTerminals nonterminals mostly tend to have a terminal as the first alternative and nonterminals as the other ones, or vice versa, but we decided to combine such cases into one micropattern due to their extremely low prevalence (under 0.05 %).

3.8 Templates

In previous sections, we have already seen some micropatterns defined as templates like "opening-bracket, nonterminal, closing bracket" (part of ExprLowLayer), "single terminal" (Keyword or Operator), etc. In fact, there are 2673 such templates in total found in the corpus of grammars, and in this section we present the most prevalent ones of them (Table 8):

Category	Pattern	Matches	Prevalence	Frequency
Template	Constructor	657	1.60%	13.56%
	Bracket	132	0.32%	2.73%
	BracketedFakeSepList	56	0.14%	1.16%
	BracketedFakeSLStar	10	0.02%	0.21%
	BracketedOptional	117	0.29%	2.42%
	BracketedPlus	6	0.01%	0.12%
	BracketedSepListPlus	8	0.02%	0.17%
	BracketedSepListStar	24	0.06%	0.50%
	BracketedStar	15	0.04%	0.31%
	Delimited	81	0.20%	1.67%
	ElementAccess	25	0.06%	0.52%
	PureSequence	2,999	7.31%	61.91%
	DistinguishByTerm	933	2.27%	19.26%
	Total coverage	**4,844**	**11.80%**	

Table 8. Template micropatterns

- Constructor: a named (non-empty production label or a top-level selector) empty term (ε);
- Bracket: a bracket-delimited nonterminal:

```
Explicit_creation_type ::= "{" Type "}" ;
Actual_generics ::= "[" Type_list "]" ;
Parenthesized ::= "(" Expression ")" ;
External_system_file ::= "<" Simple_string ">" ;
```

(ISO/IEC 25436:2006(E) Eiffel)

- BracketedFakeSepList: a bracket-delimited explicitly encoded separator list:

```
typeParameters ::= "<" typeParameter ("," typeParameter)* ">" ;
namedFormalParameters ::= "[" defaultFormalParameter
              ("," defaultFormalParameter)* "]" ;
```

(ANTLR Google Dart)

```
template ::= "{{" title ("|" part)* "}}" ;
tplarg ::= "{{{" title ("|" part)* "}}}" ;
```

(EBNF MediaWiki)

- BracketedFakeSLStar: a bracket-delimited possibly empty separator list;
- BracketedOptional: a bracket-delimited optional reference to another nonterminal;
- BracketedPlus: a bracket-delimited one-or-more repetition of a nonterminal;
- BracketedSepListPlus: a bracket-delimited separator list;
- BracketedSepListStar: a bracket-delimited possibly empty separator list;
- BracketedStar: a bracket-delimited zero-or-more repetition of a nonterminal;

– Delimited: a sequence of symbols delimited by non-bracketing terminals:

```
RecordType ::= "RECORD" Fields "END" ;
LoopStmt ::= "LOOP" Stmts "END" ;
```

(SDF Modula 3)

– ElementAccess: a nonterminal followed by a bracketed nonterminal:

```
slice ::= prefix "(" discrete_range ")" ;
```

(LNCS 4348, Ada 2005)

```
libraryDefinition ::= LIBRARY "{" libraryBody "}" ;
```

(ANTLR Google Dart)

```
ArrayDeclarator ::= VariableName "(" ArraySpec ")" ;
StructureConstructor ::= TypeName "(" ExprList ")" ;
```

(TXL Fortran 77/90)

– PureSequence: a definition that uses purely sequential composition;

– DistinguishByTerm: a choice where each alternative starts with a terminal:

```
wildcard_type_bound ::= ("extends" type_specifier)
 | ("super" type_specifier) ;
```

(TXL Java 1.5 Basis Grammar)

```
default_expression_OR_nodefault ::= ("default" expression)
 | "nodefault" ;
```

(TXL Basis Grammar for Borland Delphi Object Pascal 1.1)

```
image-mode-manual-thumb ::= ("thumbnail=" image-name)
 | ("thumb=" image-name) ;
```

(BNF MediaWiki)

4 Related work

An obviously related research topic to micropatterns are design patterns [13], implementation patterns [3] and architectural patterns [11]. In the software language engineering community, there is no widely accepted collection of "DSL design patterns", but there is no shortage on papers and books with guidelines on language design and implementation [31,16,33,1,27,19,12,30, 1965–2013]. Most of these guidelines encapsulate their authors' vision and experience, but are still waiting to be formally organised, algorithmically expressed and verified. We hope that the catalogue of micropatterns is a step toward that goal, even if a small one. In [10], the main focuses of tool support for patterns were identified as application, validation and discovery — of these three, micropatterns mostly contribute to discovery.

Extending software metrics line of thinking to grammars can also be identified as a related domain to grammatical micropatterns. However, there are three main differences between our work and grammar metric suites like gMetrics [7] and SynC [28]. First, grammar metrics are used mostly for measurements, while the main purpose of micropatterns is classification. One can compare grammars based on their metrics, and one can cluster them by size, McCabe complexity and other computed values, so this gap is not unbridgeable, but it is present. The second issue is that grammar metrics work on the level of grammars, while micropatterns in this paper are formulated on the level of nonterminals. The third

difference is that some metrics like Varju height are very complicated and require lengthy computations, which clearly contradicts with the simplicity requirement we formulated in section 1. It remains to be seen whether micropatterns carry a value for grammar metrics in a form of "how many nonterminals in grammar X satisfy the condition of micropattern Y?".

In [8], it is noted that the expressiveness of the software language that is used to define (micro)patterns, severely affects the complexity of their validation and discovery. By using state of the art technology like Rascal [21], we were able to fit the entire system of classifiers and all the experimental code around it, in 760 lines of code, which is about as concise as one could hope.

Being formulated on the level of nonterminals, which is arguably the most fine-grained level of details one could get when working with grammars, puts grammatical micropatterns closer to OOP nanopatterns [2]. However, there is still enough space for grammatical nanopatterns — one could think of them as continuation of the ContainsX micropatterns from subsection 3.1 and operate with patterns like "contains a semicolon terminal", "contains two consecutive terminals", etc.

Grammar mutations are large scale intentional grammar transformations [38, §3.8.1] that involve enforcing a new naming convention over the entire grammar, performing massive folding/unfolding rewritings, removing all terminal symbols in one sweep, etc. Micropatterns are related to them because they can be used as triggers for actual transformation steps, as preconditions and postconditions. For instance, we can say that some grammar mutation works on all nonterminals satisfying the micropattern LowerCase, and as a result they start being UpperCase. The change itself can be either inferred or programmed, but still with a lot of control and a strict specification around it.

5 Conclusion

We have identified **85** algorithmically recognisable, purposeful, notable, simple micropatterns, by analysing **41038** nonterminal symbols of **533** software language definitions. Many of these micropatterns have been previously researched, used or considered in publications in the domain of grammarware engineering. Both the original corpus of grammars and the implementation of micropattern recognisers is publicly exposed through a GitHub project.

References

1. L. Ammeraal. On the Design of Programming Languages Including MINI ALGOL 68. In J. Mülbacher, editor, *GI 5. Jahrestagung*, volume 34 of *LNCS*, pages 500–504. Springer, 1975.
2. F. Batarseh. Java Nano Patterns: a Set of Reusable Objects. In *Proceedings of the 48th Annual Southeast Regional Conference*, ACM SE '10, pages 60:1–60:4. ACM, 2010.
3. K. Beck. *Smalltalk. Best Practice Patterns.* Prentice Hall, 1996.

4. S. Butler. Mining Java Class Identifier Naming Conventions. In *Proceedings of the International Conference on Software Engineering*, ICSE'12, pages 1641–1643. IEEE Press, 2012.
5. N. Chomsky. *Syntactic Structures*. Mouton, 1957.
6. N. Chomsky. On Certain Formal Properties of Grammars. *Information and Control*, 2(2):137–167, 1959.
7. M. Črepinšek, T. Kosar, M. Mernik, J. Cervelle, R. Forax, and G. Roussel. On Automata and Language Based Grammar Metrics. *Computer Science and Information Systems*, 7(2), 2010.
8. P. van Emde Boas. Resistance is Futile; Formal Linguistic Observations on Design Patterns. Technical Report ILLC-CT-97-02, Institute for Logic, Language and Computation, University of Amsterdam, 1997.
9. B. Fischer, R. Lämmel, and V. Zaytsev. Comparison of Context-free Grammars Based on Parsing Generated Test Data. In U. Aßmann and A. Sloane, editors, *Post-proceedings of the Fourth International Conference on Software Language Engineering (SLE 2011)*, volume 6940 of *LNCS*, pages 324–343. Springer, Heidelberg, 2012.
10. G. Florijn, M. Meijers, and P. Winsen. Tool Support for Object-Oriented Patterns. In M. Akşit and S. Matsuoka, editors, *ECOOP'97*, volume 1241 of *LNCS*, pages 472–495. Springer, 1997.
11. M. Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley Professional, 2002.
12. M. Fowler. *Domain Specific Languages*. Addison-Wesley Professional, 2010.
13. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
14. J. Gil and I. Maman. Micro Patterns in Java Code. In *Proceedings of OOPSLA'05*, pages 97–116. ACM, 2005.
15. S. A. Greibach. A New Normal-Form Theorem for Context-Free Phrase Structure Grammars. *Journal of the ACM*, 12(1):42–52, Jan. 1965.
16. C. A. R. Hoare. Hints on Programming Language Design. Technical report, Stanford University, Stanford, CA, USA, 1973.
17. S. C. Johnson. *YACC—Yet Another Compiler Compiler*. Computer Science Technical Report 32, AT&T Bell Laboratories, Murray Hill, New Jersey, 1975.
18. M. de Jonge and R. Monajemi. Cost-Effective Maintenance Tools for Proprietary Languages. In *Proceedings of ICSM 2001*, pages 240–249. IEEE, 2001.
19. A. Kleppe. *Software Language Engineering: Creating Domain-Specific Languages Using Metamodels*. Addison-Wesley Professional, 2008.
20. P. Klint, R. Lämmel, and C. Verhoef. Toward an Engineering Discipline for Grammarware. *ACM Transactions on Software Engineering Methodology (ToSEM)*, 14(3):331–380, 2005.
21. P. Klint, T. van der Storm, and J. Vinju. EASY Meta-programming with Rascal. In J. M. Fernandes, R. Lämmel, J. Visser, and J. Saraiva, editors, *GTTSE*, volume 6491 of *LNCS*, pages 222–289. Springer, January 2011.
22. R. Lämmel. Grammar Adaptation. In *Proceedings of the International Symposium of Formal Methods Europe on Formal Methods for Increasing Software Productivity*, volume 2021 of *LNCS*, pages 550–570. Springer, 2001.
23. R. Lämmel and C. Verhoef. Semi-automatic Grammar Recovery. *Software—Practice & Experience*, 31(15):1395–1438, December 2001.
24. R. Lämmel and G. Wachsmuth. Transformation of SDF Syntax Definitions in the ASF+SDF Meta-Environment. In *Proceedings of LDTA'01*, volume 44 of *ENTCS*. Elsevier Science, 2001.

25. R. Lämmel and V. Zaytsev. Recovering Grammar Relationships for the Java Language Specification. *Software Quality Journal (SQJ)*, 19(2):333–378, Mar. 2011.
26. M. Lanza and S. Ducasse. A Categorization of Classes based on the Visualization of their Internal Structure: the Class Blueprint. In L. M. Northrop and J. M. Vlissides, editors, *Proceedings of OOPSLA'01*, pages 300–311. ACM, 2001.
27. M. Mernik, J. Heering, and A. M. Sloane. When and How to Develop Domain-Specific Languages. *ACM Computing Surveys*, 37(4):316–344, 2005.
28. J. F. Power and B. A. Malloy. A Metrics Suite for Grammar-based Software. *Journal of Software Maintenance and Evolution: Research and Practice*, 16:405–426, Nov. 2004.
29. A. Thies and C. Roth. Recommending Rename Refactorings. In *Proceedings of the Second International Workshop on Recommendation Systems for Software Engineering*, RSSE'10, pages 1–5. ACM, 2010.
30. M. Völter, S. Benz, C. Dietrich, B. Engelmann, M. Helander, L. C. L. Kats, E. Visser, and G. Wachsmuth. *DSL Engineering: Designing, Implementing and Using Domain-Specific Languages*. dslbook.org, 2013.
31. A. van Wijngaarden. Orthogonal Design and Description of a Formal Language. MR 76, SMC, 1965.
32. D. S. Wile. Abstract Syntax from Concrete Syntax. In *ICSE*, pages 472–480. ACM, 1997.
33. N. Wirth. On the Design of Programming Languages. In *IFIP Congress*, pages 386–393, 1974.
34. N. Wirth. What Can We Do about the Unnecessary Diversity of Notation for Syntactic Definitions? *Communications of the ACM*, 20(11):822–823, 1977.
35. V. Zaytsev. MediaWiki Grammar Recovery. *Computing Research Repository (CoRR)*, 4661:1–47, July 2011.
36. V. Zaytsev. BNF WAS HERE: What Have We Done About the Unnecessary Diversity of Notation for Syntactic Definitions. In S. Ossowski and P. Lecca, editors, *SAC/PL 2012*, pages 1910–1915. ACM, Mar. 2012.
37. V. Zaytsev. Notation-Parametric Grammar Recovery. In A. Sloane and S. Andova, editors, *Post-proceedings of LDTA 2012*. ACM, June 2012.
38. V. Zaytsev. The Grammar Hammer of 2012. *Computing Research Repository (CoRR)*, 4446:1–32, Dec. 2012.
39. V. Zaytsev. Abstract Normal Form for Grammars in a Broad Sense. Submitted to the Information Processing Letters (IPL). Under review. http://dx.doi.org/10.6084/m9.figshare.643391, May 2013.
40. V. Zaytsev. Grammar Zoo: A Repository of Experimental Grammarware. Fifth Special issue on Experimental Software and Toolkits of Science of Computer Programming (SCP EST5). Currently under major revision, 2013.
41. V. Zaytsev and R. Lämmel. A Unified Format for Language Documents. In B. A. Malloy, S. Staab, and M. G. J. van den Brand, editors, *Post-proceedings of the Third International Conference on Software Language Engineering (SLE 2010)*, volume 6563 of *LNCS*, pages 206–225, Berlin, Heidelberg, Jan. 2011. Springer-Verlag.
42. V. Zaytsev, R. Lämmel, T. van der Storm, L. Renggli, and G. Wachsmuth. Software Language Processing Suite[5], 2008–2013. http://slps.github.io.

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