

# Java is Not Syntax-safe – Apparently\*

Ralf Lämmel and Vadim Zaytsev

Software Languages Team, The University of Koblenz-Landau, Germany

## Abstract

Using a mechanized process, we have reverse-engineered the relationships between the 6 different grammars that are contained in 3 versions of the Java Language Specification (JLS). To this end, we have extracted the grammars from (the HTML representation of) the JLS, and we have systematically resolved or captured all accidental or intended differences by grammar transformations. The project reveals a considerable number of problems with the grammars such as productions that did not define the intended language.

## 1 Introduction

The Java Language Specification (JLS; [5, 6, 7]) is a/the foundation for Java compilers, Java reverse and re-engineering tools, if not all Java technologies. There exist 3 versions of the JLS as a result of the Java language’s evolution. Each version contains 2 grammars: a “more readable” one and a “more implementable” one. The 2 grammars per version are meant to be (essentially) equivalent in the formal sense of the generated language, but the JLS neither provides any formal evidence that equivalence holds nor any structured description of the differences between the grammars. Each newer version of the JLS describes a “grown” (say, an extended) Java language, when compared to the previous version, but the JLS does not formalize the “language delta” or other differences between the grammars of the different versions.

### Contributions

1. We have recovered the actual relationships between all these grammars in the form of grammar trans-

formations. This approach distinguishes fold/unfold variations, iteration style, language extensions, and other kinds of grammar differences explicitly. This is the first time that relationships between sized grammars have been mechanized in such a systematic and comprehensive manner.

2. Our approach guarantees to spot any sort of differences between the grammars at hand. We have indeed spotted a considerable number of issues in the JLS: productions that did not define the intended language, weakly or undocumented cases of using more liberal constructs in one of the grammars, not even to mention irregularities in the HTML format.
3. The project is a case study for the method of grammar convergence [14] — a lightweight verification method for relationships between scattered grammar knowledge. The scale and the results of the project clarify the effectiveness of the method as well as its original contribution, when compared to earlier work on grammar recovery [12]. We have measured various aspects of this project, and report and interpret all these numbers in this paper.<sup>1</sup>

**Roadmap** §2 introduces the JLS corpus of specifications and the contained grammars; the “distance” between the grammars is informally illustrated. §3 describes an original grammar extractor that scraps grammars from JLS documents that use an HTML representation. §4 describes the critical part of this project: the use of transformations for refactoring, correction, extension, and others. Related work is discussed in §5, and the paper is concluded in §6.

<sup>1</sup>The complete case study including all the involved sources, transformations, results, and tools are publicly available through <https://sourceforge.net/projects/slps/>; see [topics/java/lci](#) in particular.

\*The title is a pun and an homage on a series of papers [4],[16],...

## 2 The JLS corpus

Each of the 3 versions of the JLS develops the “more readable” grammar over the main chapters of the document, and summarizes the “more implementable” grammar, *en bloc*, in a late section (a de-facto appendix). We refer to these various grammars as *doc1*, *app1*, *doc2*, *app2*, *doc3*, and *app3*. In the following, we gather some basic knowledge about the grammars.

**JLS1** According to [5, §19], the *app1* grammar “has been mechanically checked to insure that it is LALR(1)”. The *doc1* grammar is also referred to as “syntactic grammar” and its relationship to *doc1* is briefly described as follows [5, §2.3]: “A LALR(1) version of the syntactic grammar is presented in Chapter 19. The grammar in the body of this specification is very similar to the LALR(1) grammar but more readable.” Presumably, the grammars are supposed to generate (essentially) the same language in a formal sense. While the two grammars *doc1* and *app1* are said to be similar, a more precise relationship is not in sight though.

**JLS2** According to [6, “Preface to the Second Edition”], “[...] the language has grown [...] This second edition [...] integrates all the changes made to the Java programming language since [...] the first edition in 1996. The bulk of these changes [...] revolve around the addition of nested type declarations.” The 2nd version does not explicitly relate its grammars to those of the 1st version. Upon cursory examination we came to conclude that *doc1* and *doc2* are similar (modulo the extensions to be expected), whereas surprisingly, *app1* and *app2* appeared as different developments. Also, the LALR(1) claim for *app1* is not matched for *app2*, but instead the following, less precise statement is made [6, §18]: “The grammar presented piecemeal in the preceding chapters is much better for exposition, but it is not ideally suited as a basis for a parser. The grammar presented in this chapter is the basis for the reference implementation.”

**JLS3** JLS3 extends JLS2 in numerous ways [7, Preface]: “Generics, annotations, asserts, autoboxing and unboxing, enum types, foreach loops, variable arity methods and static imports have all been added to the language

	Grammar class	Iteration style
<i>app1</i>	LALR(1)	left-recursive
<i>doc1</i>	none	left-recursive
<i>app2</i>	unclear	EBNF (*)
<i>doc2</i>	none	left-recursive
<i>app3</i>	“nearly” LL(k)	EBNF (*)
<i>doc3</i>	none	left-recursive

Figure 1: Properties of the JLS grammars.

	# Productions	# Nonterminals	# Tops	# Bottoms
<i>app1</i>	135	135	1	7
<i>doc1</i>	148	148	1	9
<i>app2</i>	80	80	6	11
<i>doc2</i>	151	151	1	11
<i>app3</i>	114	114	2	12
<i>doc3</i>	197	197	3	15

Figure 2: Metrics for the JLS grammars.

recently.” Again, the 3rd version does not explicitly relate its grammars to those of the 2nd version. Upon cursory examination we came to conclude that *doc2* and *doc3* are similar, just as much as *app2* and *app3* (modulo the extensions to be expected). No definitive grammar-class claim is made, but an approximation thereof [7, §18] that suggests that *app3* has definitely departed from LALR(1), i.e., *app3* is said to be “not an LL(1) grammar, though in many cases it minimizes the necessary look ahead.”

**Grammar data** Fig. 1 lists the grammar-class claims that we gathered; we also add observations about iteration style (“lists”) that we made during cursory examination.

Fig. 2 summarizes some simple metrics for the JLS grammars. (We automatically derived these numbers from the extracted grammars.) One can expect that the major differences between the number of productions and nonterminals is a result of different styles of grammar structuring, e.g., different iteration styles as discussed above. The difference in numbers of top-nonterminals [17, 12] is a problem indicator. There should be only one top-nonterminal: the actual start symbol of the Java grammar. Any additional case of an “un-

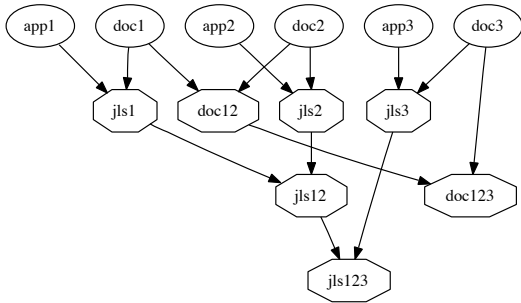


Figure 3: *Convergence tree for the JLS grammars.*

used” nonterminal does not make sense. The difference in numbers of bottom-nonterminals *may* be reasonable because the choice of the classes of lexemes is somewhat of a design issue. However, a review of the nonterminal names rapidly reveals that a good number of them corresponds to undefined syntactic categories as opposed to lexeme classes.

**Grammar convergence** Fig. 3 shows the desired convergence tree [14] for the JLS grammars; the JLS grammars (“sources”) are the leaf nodes; the results of convergence (“targets”) are the inner nodes and the root.

The 2 grammars of each JLS version are “converged” to account for the differences between the “more readable” and the “more implementable” grammar (see *jls1*, *jls2*, and *jls3*). All 3 versions are “converged” in a cascade to account for inter-version differences such as extensions in particular (see *jls12* and *jls123*).

Since we observed that the 3 “more readable” grammars are similar, it makes sense to attempt a redundant path, i.e., to converge *doc1* . . . *doc3* without any influence from *app1* . . . *app3*. Hence, there is another cascade with targets *doc12* and *doc123*. We mention that, in the derivation of the targets *jls1*, *jls2* and *jls3*, we lean towards the “more implementable” grammar because it is typically more permissive, and easier to reach transformationally than the other way around.

### 3 Grammar extraction

The JLS is available electronically in HTML and PDF. Neither the HTML nor the PDF formats were designed

*Production:*

*Nonterminal* " : [ "one" "of" ] *CR Line* { *Line* } *CR*

*Line:*

*Indent Symbols CR*

*Symbols:*

*Symbol* { *Symbol* }

*Symbol:*

*Nonterminal*

*Terminal*

" ( " *Symbols* " | " *Symbols* { " | " *Symbols* } " ) "

" [ " *Symbols* " ] "

" { " *Symbols* " } "

*CR:*

... carriage return ...

Figure 4: *Relevant grammar expressiveness given in a self-descriptive manner; for clarity, terminals are enclosed in double quotes as opposed to the use of markup; the markup-based form of optionals is also neglected.*

with convenient access to the grammars in mind. We opted for the HTML route. Our effort resulted in an original extractor design, and it revealed a number of irregularities in (the HTML representation of) the JLS. Arguably, others have attempted the (then less automated) recovery of a Java grammar from the JLS, but these grammars are of no use to us because we strive to examine the *verbatim* grammars as shown in the JLS.

**Assumed grammar format** Grammar fragments are hosted by `<pre>...</pre>` blocks in the JLS documents. According to [5, 6, 7, §2.4]: terminal symbols are shown in fixed font (as in `<code>class</code>`); nonterminal symbols are shown in italic type (as in `<i>Expression</i>`); a subscripted suffix “opt” indicates an optional symbol (as in `Expression<sub>opt</sub>`); alternatives start in a new line and they are indented; “one of” marks a top-level choice with atomic branches. (We have also observed that nonterminals are expected to be alphanumeric and start in upper case.) Further notation and expressiveness is described in [6, 7, §18]:  $[x]$  denotes zero or one occurrences of  $x$ ;  $\{x\}$  denotes zero or more occurrences of  $x$ ;  $x_1 | \dots | x_n$  forms a choice over the  $x_i$ . The JLS documents consistently suffice with “\*” lists (zero or more occurrences); there are no uses of “+” lists. Refer to Fig. 4 for a summary.

We should also mention line continuation; it allows to

spread one alternative over several lines [7, §2.4]: “A very long right-hand side may be continued on a second line by substantially indenting this second line”.

**Example 1** A grammar fragment as of [6, §4.2]:

```
<i>NumericType:
    IntegralType
    FloatingPointType

IntegralType: one of</i>
    <code>byte short int long char
</code>
```

For comparison, in pretty-printed format:

```
NumericType:
    IntegralType
    FloatingPointType

IntegralType : one of
    "byte" "short" "int" "long" "char"
```

The fragment illustrates two different kinds of “choices”, i.e., multiplicity of vertical alternatives, and “one of” choices. (The third form, which is based on “|”, is not illustrated.) The fragment also clarifies that markup tags are used rather liberally. The “nonterminal” tag (i.e., `<i>...</i>`) spans more than one production. The terminal tag (i.e., `<code>...</code>`) spans several terminals and the closing tags ends up on a new line.

**Phase 1 — Preprocessing** The tiny Ex. 1 is a good indication of the many irregularities that are found in the HTML representation. We need a non-classic grammar parser to deal with these irregularities. Our extractor therefore works in several phases. The first phase, which we call a preprocessing phase, has the following I/O behavior:

- Input: the `<pre>...</pre>` blocks.
- Output: a dictionary
  - Keys: Left-hand side nonterminals
  - Values: Arrays of top-level alternatives

The phase is subject to the following requirements:

**Tag elimination** The input notation interleaves tags with proper grammar structure. In order to prepare for classic parsing, we need to eliminate the tags in the process of constructing properly typed lexemes for terminals and nonterminals.

**Indentation elimination** The input notation relies on indentation to express top-level choices and line continuation. The output format stores top-level choices in arrays, and fuses multi-line alternatives.

**Robustness** The inner structure of top-level alternatives is parsed simply as a sequence of tokens in the interest of robustness so that recovery rules can be applied separately, before, finally the precise grammar structure is parsed.

The preprocessor relies on a stateful scanner (to meet “tag elimination”) and a robust parser (to meet “robustness”). The parser recognizes sequences of productions, each one essentially consisting of a sequence of alternatives; it parses alternatives as sequences of tokens terminated by CR. The scanner uses three states:

**italic** upon opening `<i>` tag (or `<em>`)

**fixed** upon opening `<code>` tag

**default** when no tag is open

That is, we treat each tag as a special token that changes the global state of the scanner, which in turn can be observed when creating morphemes for terminals and nonterminals. We also deal with violations of XML and HTML well-formedness in this manner. Here is the decision table of the scanner; classes of strings on the y-axis; states on the x-axis; we also show the number of times each decision is taken for all JLS documents:

	<i>italic</i>	<i>fixed</i>	<i>default</i>
Alphanumeric	N (2342)	T (173)	T? (194)
	M (2)	T (2)	M? (29)
{,},[,],(,.)	M (707)	T (174)	T? (200)
otherwise	T (198)	T (165)	T (205)

(T ... terminal, N ... nonterminal, M ... metasympol)

Most of these decisions are inevitable, even though some of them pinpoint markup errors. An example of an “error-free” decision is to map an alphanumeric string in italic mode to a nonterminal. An example of an “error-recovering” decision is to map a non-alphanumeric token (that does not match any metasympol) to a terminal — even when it is tagged with `<i>...</i>`. Several decisions in the “default” column involve an element of choice (as indicated by “?”). The shown decisions give the best results, that is, they require the least subsequent transformations of the extracted grammar. For instance, it turned

out that bars without markup were supposed to be BNF bars, but other metasymbols were better mapped to terminals, whenever markup was missing. Also, alphanumeric strings without markup turned out to be mostly terminals, and hence that preference was implemented as a decision by the scanner.

**Phase 2 — Error recovery** We face (a few) syntax errors (with regard to the syntax of the grammar notation). We also face a number of “obvious” semantic errors (in the sense of the language generated by the grammar). We call them obvious errors because they can be spotted by simple, generic grammar analyses that involve only very little Java knowledge, if any. We have opted for an error-recovery approach that relies on a uniform, rule-based mechanism that performs transformations on each sequence of tokens that corresponds to an alternative. The rules are applied until they are no longer applicable. We describe the rules informally; they are implemented by regular expression matching.

**Rule 1 (Match up parentheses)** *When there is a group (a bar-based choice) that misses an opening or closing parenthesis, such as in “(a—b”, then a nearby terminal “(” or “)” (if available) is to be converted to the parenthesis. If there is still a closing parenthesis that cannot be matched, then it is dropped. (We have not seen the case of an opening parenthesis to remain unmatched.)*

**Rule 2 (Metasymbol to terminal)** *(a) When “|” was scanned as a BNF metasymbol, but it is not used in the context of a group, then it is converted to a terminal. (b) When “[” and “]” occur next to each other as BNF symbols, then they are converted to terminals. (c) When “{” and “}” occur next to each other as BNF symbols, then they are converted to terminals. (d) When an alternative makes use of the metasymbols for grouping, but there is no occurrence of the metasymbol “|”, then the parentheses are converted to terminals.*

**Rule 3 (Compose sibling symbols)** *When two alphanumeric nonterminal or terminal tokens are next to each other where one of the symbols is of length 1, then they are composed as one symbol.*

**Example 2** *Multiple terminals to compose [5, §19.11]:*

```
<code>continu</code><i>e
```

*Multiple nonterminals to compose [5, §14.9]:*

```
S<i>witchBlockStatementGroups</i>
```

**Rule 4 (Decompose compound terminals)** *When a terminal consists of an alphanumeric prefix, followed by “.”, possibly followed by some postfix, then the terminal is taken apart into several ones.*

**Example 3** *Consider this phrase [6, §15.9]:*

```
Primary.new Identifier ( ... ) ClassBodyopt
```

*The decomposition results in the following:*

```
Primary . new Identifier ( ... ) ClassBodyopt
```

**Rule 5 (Nonterminal to terminal)** *Lower-case nonterminals that are not defined by the grammar (i.e., that do not occur as a key in the dictionary coming out of phase 1), and are in lower case, are converted to terminals.*

**Rule 6 (Terminal to nonterminal)** *Alphanumeric terminals that start in upper case, and are defined by the grammar (when considered as nonterminals) are converted.*

**Rule 7 (Recover optionality)** *When a nonterminal’s name ends on “opt”, as in “fooopt”, and the grammar defines a nonterminal “foo”, then the nonterminal “fooopt” is replaced by [foo]. (Hence, markup for subscript “opt” was missing.)*

**Example 4** *Consider again the result of Ex. 3:*

```
Primary . new Identifier ( ... ) ClassBodyopt
```

*The recovered optional looks as follows:*

```
Primary . new Identifier ( ... ) [ ClassBody ]
```

These are all the rules that have stabilized over the project’s duration. Several other rules were investigated but eventually abandoned because the corresponding issues could be efficiently addressed by grammar transformations. We used experimental rules to test for the recurrence of any issue we had spotted. We quantify the use of the rules shortly.

	app1	app2	app3	doc1	doc2	doc3	Total
Non-well-formedness issues	5	0	7	4	11	4	31
Unusual line continuations	1	2	7	1	4	8	23
Matched up parentheses	0	3	6	0	0	0	9
Metasymbol to terminal	0	1	7	0	27	7	42
Sibling symbols	1	0	0	1	1	0	3
Decomposed symbols	0	1	1	0	3	8	13
Nonterminal to terminal	0	7	3	0	8	11	29
Terminal to nonterminal	1	0	1	1	17	13	33
Recovered optionality	1	0	0	0	3	8	12
Duplicate definitions	0	0	0	16	17	18	51
Total	9	14	32	23	91	77	246

Figure 5: Irregularities resolved by grammar extraction.

**Phase 3 — Removal of doubles** The JLS documents (deliberately) repeat grammar parts. Hence, we have added a trivial phase for removal of double alternatives. That is, when a given right-hand side nonterminal is encountered several times in a source, then phase 1 accumulates all the alternatives via one entry of the dictionary, and phase 3 compares alternatives (i.e., sequences of tokens) to remove any doubles.

**Example 5** Consider the following definition [6, §8.3]:

```
VariableDeclaratorId:
  Identifier
  VariableDeclaratorId [ ]
```

The same definition appears elsewhere in the document, even though the markup is different, but these differences are already neutralized during phase 1 [6, §14.4]:

```
<em>VariableDeclaratorId:
  Identifier
  VariableDeclaratorId</em> [ ]
```

Phase 3 preserves 2 alternatives out of 4. As an aside, this particular example also required the application of Rule 2.b because [ ] must be converted to terminals.

**Phase 4 — Precise parsing** Finally, the dictionary structure of phase 1, after the recovery of phase 2, and double removal of phase 3, is trivially parsed according to the (E)BNF for the grammar notation; c.f., Fig. 4. In fact, our implementation dumps the extracted grammar immediately in an XML-based grammar interchange format so that generic grammar tools for comparison and transformation can take over [14].

**Extraction data** Fig. 5 summarizes the frequency of using recovery rules, handling “unusual” continuation lines<sup>2</sup>, and removal of doubles. The extractor has fixed 246 problems that otherwise would have prevented straightforward parsing to succeed with extraction, or implied loss of information, or triggered substantial grammar transformations.

## 4 Grammar transformation

The overall goal of the transformation process is to establish structural equivalence of the transformed sources along the tree structure of Fig. 3. We use an operator suite for grammar transformation akin to [11, 13]. The precise operator suite is beyond the scope of this publication, but we categorize the transformations that are needed, we illustrate those categories, and we compile some data about the transformation process.

**Category “semantic error recovery cont’d”** The extractor recovered from all syntax errors and simple, recurring semantic errors. There are a few more semantic errors that are essentially implied by notational irregularities of the HTML documents. Each and every such irregularity is fixed by a designated transformation.

**Example 6** A transformation for “semantic error recovery”: curly brackets were not recognized as terminals.

<sup>2</sup>Initially, our guess was that “substantially indenting” means more spaces or tabs than the previous line, but it turned out there are cases when continuation lines were not indented at all.

---

*doc2*

```

ClassBody:
  { [ ClassBodyDeclarations ] }

```

---

*corrected*

```

ClassBody:
  "{" [ ClassBodyDeclarations ] }"

```

**Category “language correction”** A profound form of errors in the JLS grammars are those that require a transformation for “language correction” in the sense of changing the generated language due to replacements of unintended definitions of nonterminals or provisions of missing definitions of nonterminals.

**Example 7** A difference calling for “language correction”: the fragment contributes to the layered definition of expression forms. The nonterminals *Expression1*, *Expression2*, ... serve for the expression of priorities. The upper definition of the “instanceof” form contains an incorrect component, c.f., *Expression3*.

---

*app2 and app3*

```

Expression2:
  Expression3 [ Expression2Rest ]
Expression2Rest:
  { InfixOp Expression3 }
  Expression3 "instanceof" Type

```

---

*corrected*

```

Expression2:
  Expression3 [ Expression2Rest ]
Expression2Rest:
  { InfixOp Expression3 }
  "instanceof" Type

```

**Category “language extension”** When one language version is supposed to be a backward-compatible extension of another, then the corresponding grammars need to be aligned by adding alternatives for existing syntactic categories and by injecting new (possibly optional) components within existing structures. (Cases of language extension clearly should not be expected for grammars of the same language version.)

**Example 8** A difference calling for “language extension”: an alternative needs to be added to upgrade the *jls2* notion of a *ConstantModifier* to the one of *jls3*.

---

*doc2*

```

ConstantModifier:
  "public"
  "static"
  "final"

```

---

*doc3*

```

ConstantModifier:
  Annotation
  "public"
  "static"
  "final"

```

**Category “grammar optimization”** There are many cases of variation between the various grammars that should not necessarily be tagged as “language correction” issues because they may be the result of deliberate decisions by the grammar engineers to make the grammars more readable, more implementable, or more compact. These grammar optimizations are semantically extensions or restrictions but with the intention that the change of semantics is negligible. The use of such non-strictly semantics-preserving transformations is debatable because it may obviously lead to compliance problems between implementations and specifications, but it is common practice, and hence we use a separate category not to unfairly count such issues as correction issues.

**Example 9** A difference explainable as “grammar optimization”: the second definition assumes one all-inclusive nonterminal of *Modifier* whereas the first definition is precise about the specific modifiers that can be associated with a specific declaration.

---

*doc2*

```

InterfaceModifiers:
  InterfaceModifier
  InterfaceModifiers InterfaceModifier
MethodModifiers:
  MethodModifier
  MethodModifiers MethodModifier
FieldModifiers: ...
ClassModifiers: ...

```

---

*app2*

```

ModifiersOpt:
  { Modifier }

```

**Category “Grammar refactoring”** When grammars are not structurally equivalent, the generated languages of course may still be. The property of generating the same

	<b>jls1</b>	<b>jls2</b>	<b>jls3</b>	<b>jls12</b>	<b>jls123</b>	<b>doc12</b>	<b>doc123</b>	<b>Total</b>
Number of lines	620	4359	6569	3979	124	1517	3157	20325
Number of transformations	64	350	401	278	11	91	171	1366
◦ semantics-preserving	42	274	313	234	7	46	110	1026
◦ semantics-increasing or -decreasing	22	72	81	43	4	44	59	325
◦ semantics-revising	—	4	7	1	—	1	2	15
Number of issues	8	37	38	25	3	32	40	183
◦ recoveries	—	7	5	—	—	7	3	22
◦ corrections	5	21	19	2	—	10	8	65
◦ extensions	—	—	—	17	3	15	28	63
◦ optimizations	3	9	14	6	—	—	1	33

Figure 6: *Effort metrics and categorization of the convergence transformations for the JLS.*

language is not generally decidable for context-free grammars, but we can use refactoring transformations, which are semantics-preserving by definition, to constructively prove equivalence. The bulk of work in grammar convergence amounts to grammar refactoring, indeed. Here are some scenarios for refactoring:

- *Fold/unfold manipulation*: a proof of structural equality may involve the expansion of a nonterminal’s definition (unfold) or vice versa (fold). It may also involve the extraction or inlining.<sup>3</sup>
- *De-/yaccification*: grammars may differ with regard to style of iteration. Deyaccification transforms YACC-style of recursion to EBNF-style lists [11]. There is also the opposite direction.
- *Factoring/distribution*: one grammar may apply left factoring to facilitate parsing with limited look-ahead; another grammar may avoid any such consideration of implementation aspects.

All of these and other scenarios are relevant for the JLS.

**Example 10** *Let us encounter refactoring in Ex. 9. We start with the “semantics-increasing” part that “unites” all the nonterminals for specific kinds of modifiers as well as the nonterminals for iterations thereof. By normalization, all the resulting identical copies of the productions disappear. It remains to construct EBNF-style iteration from the left-recursive style of recursion. There is a corresponding deyaccify refactoring to this end. We also need*

<sup>3</sup>Extraction basically introduces a new nonterminal to capture some existing structure, and then uses the new nonterminal immediately in a fold step [11].

*to rename the nonterminals for modifiers and sequences thereof to Modifier and ModifiersOpt. Renaming is a form of refactoring, too.*

**Transformation data** Fig. 6 summarizes the transformation process in numbers. Obviously, the numbers of “lines” and “transformations” do not convey much information due to the lack of a good point of reference. We mention that the transformation operators provide primitive adaptations as opposed to any sort of higher-level scheme.<sup>4</sup> The bulk of the transformations are semantics-preserving (75%), but there is also a considerable number of semantics-increasing/-decreasing transformations (24%)<sup>5</sup> to model extensions, some of the corrections, and optimizations. The number of semantics-revising transformations is very small (1%).

We have tagged all transformations to contribute to a specific “issue” (say, a recovery, correction, extension or optimization issue). An issue is the adaptation of a specific production, a specific definition, a specific occurrence of a given expression or a nonterminal, or all occurrences of an expression or a nonterminal. For instance, the number of extension issues maps to the sum of all language constructs that need to be added. The table does not count “issues of grammar refactoring” because we found it intrinsically hard to split these refactorings up in a transparent (countable) manner.

<sup>4</sup>For instance, one could imagine a generic scheme of deyaccification, but our operators necessitate separate transformation for each recursively defined nonterminal.

<sup>5</sup>Given a transformation  $T$  from a grammar  $G$  to a grammar  $G'$ ,  $T$  is semantics-increasing (decreasing resp.) if  $L(G) \subset L(G')$  ( $L(G') \subset L(G)$  resp.);  $T$  is semantics-revising if  $L(G) \not\subseteq L(G')$  and  $L(G') \not\subseteq L(G)$  [11].

	# Productions	# Nonterminals	# Tops	# Bottoms
<i>jls1</i>	147	132	1	7
<i>jls2</i>	75	75	1	7
<i>jls3</i>	131	115	1	10
<i>jls12</i>	75	75	1	7
<i>jls123</i>	81	76	1	11
<i>doc12</i>	152	152	1	7
<i>doc123</i>	201	201	1	8

Figure 7: Metrics for the transformed grammars.

**Grammar metrics** Fig. 7 summarizes some simple metrics for the targets of convergence — the same metrics that were shown for the sources in Fig. 2. There is only one top-nonterminal for each of these grammars. Likewise, the bottom-nonterminals have been consolidated; the remaining small difference accounts for an extension of the 3rd version of the JLS.

## 5 Related work

The present paper, in essence, describes a (major) case study on grammar convergence. Hence, we refer to the introductory text [14] on grammar convergence for a broader related work discussion, and we focus here on the discussion of issues that are particularly relevant for the case study at hand.

Prior work on *re-engineering syntax definitions* [17, 12, 9] typically focuses on the derivation of *one* “quality” grammar from *one* source artifact (such as a language reference or a proprietary implementation), where quality typically translates to the ability of parsing an existing harness that acts as proxy for the intended language. In contrast, our JLS case study derives *several* (7) “quality” grammars from *several* (6) sources, where quality translates to the property of the derived grammars to be proven variations of one another and the sources — subject to refactorings and other disciplined editing operations. To this end, grammar convergence relies on grammar comparison to report grammar differences that can be addressed by transformations. It would be a complemen-

tary project to validate the sources (such as *app1*, *app2*, *app3* that are meant to be implementable) to be fit for parsing. [1] describes a re-engineering approach that applies to multiple grammars but it focuses on the variation of precedence rules rather than the problem of showing grammars to be equivalent modulo more general variations that are all expressible by transformations.

Prior work on *programmable grammar transformations* [20, 11, 13, 2, 3] has crucially influenced the transformation language (namely, XBGF) that we have used and advanced in this case study. Compared to prior work, the case study and XBGF reach a new landmark with regard to size of the problem and the involved effort, c.f., Fig. 6. The case study uses 28 different transformation operators of XBGF in some different modes. In principle, it would be possible to use a simpler transformation language (e.g., the one of [11]), but this would make the transformations considerably more tedious and reduce the number of steps that are known to be semantics-preserving. For instance, we have access to a message operator that accumulates a list of algebraic identities on EBNF expressions. Without this operator, many steps would require “unchecked” replace operations.

The design of the extractor combines very simple ideas of robust parsing and error recovery [19, 15, 10, 18]. That is, parts of the input are initially parsed like the “ocean” in the terminology of island grammars; specific patterns of syntactically erroneous input are associated with repair rules. The handling of markup tags is reminiscent of liberal HTML parsing in browsers. Several of our correction rules actually serve semantic error recovery. In compilation, such recovery typically focuses at the improvement of error diagnosis, e.g., by modifying the symbol table at hand [8], whereas in our case the goal is really to construct an alternative syntactically correct program that does not expose symptoms of semantic errors.

## 6 Concluding remarks

We have shown, for the first time, that it is possible to represent the relationships between sized grammars that serve different audiences (language users and implementers) in a mechanized fashion based on an operator suite of grammar transformations; ditto for relationships between major versions of sized grammars.

Our mechanized approach revealed a considerable number of problems in the grammars at hand. For instance, the 2nd version of the JLS required corrections in 21 locations. Here we note that we aimed to count corrections minimalistically; we invented a category of “optimizations” to give the benefit of doubt. Obviously, the owner of the JLS would have been interested in avoiding correctness problems. Since these problems exist, we take the liberty to assume that all conservative measures failed, and that grammar transformations combined with grammar comparison (in fact, grammar convergence) are essential.

We see this case study as a valuable testbed to improve our realization of grammar convergence. First, the considerable size of transformation scripts suggests that the grammar engineer should receive more help in developing and maintaining them. We see clear potential for inferring part of the scripts, and for providing substantially improved interactive support (when compared to plain grammar comparison). Second, some patterns of grammar variation are still not conveniently supported by the available transformation operators. As of writing, we use a small number of “aggressive” or “unchecked” transformations to treat these patterns efficiently. Third, we need effective ways of reasoning about non-refactoring operations which are indispensable when two grammar sources disagree. Currently, we can only informally quantify the impact of such operations.

## References

- [1] E. Bouwers, M. Bravenboer, and E. Visser. Grammar Engineering Support for Precedence Rule Recovery and Compatibility Checking. *ENTCS*, 203(2):85–101, 2008.
- [2] T. Dean, J. Cordy, A. Malton, and K. Schneider. Grammar Programming in TXL. In *Proceedings, Source Code Analysis and Manipulation (SCAM'02)*. IEEE, 2002.
- [3] T. Dean, J. Cordy, A. Malton, and K. Schneider. Agile Parsing in TXL. *Journal of Automated Software Engineering*, 10(4):311–336, 2003.
- [4] S. Drossopoulou and S. Eisenbach. Java is Type Safe – Probably. In *ECOOP'97—Object-Oriented Programming, 11th European Conference, Jyväskylä, Finland, June 9–13, 1997, Proceedings*, volume 1241 of *LNCS*, pages 389–418. Springer, 1997.
- [5] J. Gosling, B. Joy, and G. L. Steele. *The Java Language Specification*. Addison-Wesley, 1996. Available at [java.sun.com/docs/books/jls](http://java.sun.com/docs/books/jls).
- [6] J. Gosling, B. Joy, G. L. Steele, and G. Bracha. *The Java Language Specification*. Addison-Wesley, second edition, 2000. Available at [java.sun.com/docs/books/jls](http://java.sun.com/docs/books/jls).
- [7] J. Gosling, B. Joy, G. L. Steele, and G. Bracha. *The Java Language Specification*. Addison-Wesley, third edition, 2005. Available at [java.sun.com/docs/books/jls](http://java.sun.com/docs/books/jls).
- [8] C. W. Johnson and C. Runciman. Semantic errors — diagnosis and repair. In *SIGPLAN '82: Proceedings of the 1982 SIGPLAN symposium on Compiler construction*, pages 88–97. ACM, 1982.
- [9] M. de Jonge and R. Monajemi. Cost-effective maintenance tools for proprietary languages. In *Proceedings, International Conference on Software Maintenance (ICSM'01)*, pages 240–249. IEEE, 2001.
- [10] A. Klusener and R. Lämmel. Deriving tolerant grammars from a base-line grammar. In *Proceedings, International Conference on Software Maintenance (ICSM'03)*, pages 179–189. IEEE, 2003.
- [11] R. Lämmel. Grammar Adaptation. In J. Oliveira and P. Zave, editors, *Proceedings, Formal Methods Europe (FME) 2001*, volume 2021 of *LNCS*, pages 550–570. Springer, 2001.
- [12] R. Lämmel and C. Verhoef. Semi-automatic Grammar Recovery. *Software—Practice & Experience*, 31(15):1395–1438, 2001.
- [13] R. Lämmel and G. Wachsmuth. Transformation of SDF syntax definitions in the ASF+SDF Meta-Environment. In *Proceedings, Language Descriptions, Tools and Applications (LDTA'01)*, volume 44 of *ENTCS*. Elsevier Science, 2001.
- [14] R. Lämmel and V. Zaytsev. An Introduction to Grammar Convergence. Submitted for publication. Available online <http://www.uni-koblenz.de/~laemmel/convergence/>, Sept. 2008.
- [15] L. Moonen. Generating Robust Parsers using Island Grammars. In *Proceedings, Working Conference on Reverse Engineering (WCRE'01)*, pages 13–22. IEEE, Oct. 2001.
- [16] T. Nipkow and D. von Oheimb. *Java<sub>light</sub>* is Type-Safe—Definitely. In *POPL '98: Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 161–170. ACM, 1998.
- [17] M. Sellink and C. Verhoef. Development, Assessment, and Reengineering of Language Descriptions. In *Proceedings, Conference on Software Maintenance and Reengineering (CSMR'00)*, pages 151–160. IEEE, 2000.
- [18] N. Synytskyy, J. Cordy, and T. Dean. Robust Multilingual Parsing Using Island Grammars. In *Proceedings CASCON'03, 13th IBM Centres for Advanced Studies Conference, Toronto*, pages 149–161, 2003.
- [19] P. N. van den Bosch. A bibliography on syntax error handling in context free languages. *SIGPLAN Notices*, 27(4):77–86, 1992.
- [20] D. Wile. Abstract syntax from concrete syntax. In *Proceedings, International Conference on Software Engineering (ICSE'97)*, pages 472–480. ACM Press, 1997.