

Consistency of the Java Language Specification

Ralf Lämmel and Vadim Zaytsev

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

Abstract. The Java Language Specification (JLS) is an industrial standard that is critical to the Java platform. Each of the 3 versions of the JLS contains 2 different grammars — a “more readable” one, and a “more implementable” one. The JLS does not describe the correspondences between the 6 grammars in any systematic, detailed manner. We have found that there are various accidental differences between the grammars. We combine ideas of grammar recovery (reverse engineering) and grammar transformation (re-engineering) for capturing all accidental or intended differences between the 6 JLS grammars in a precise and mechanized manner. Our work provides a method for consistency management of language specifications.

1 Introduction

The *expected* consistency of the Java standard

The Java Language Specification (JLS; [9,10,11]) is an industrial standard that is critical to the Java platform. For instance, it is a foundation for Java compilers, code generators, transformation tools, and IDEs. There are 3 JLS versions (1, 2, 3); they record the backwards-compatible extension history of the Java language. Each of the 3 JLS versions contains 2 different grammars which address different needs (language users vs. language implementors) — there is a “more readable” grammar (referred to as G_1^R , G_2^R , G_3^R), and a “more implementable” one (referred to as G_1^I , G_2^I , G_3^I).

One should expect the two grammars per version to be equivalent in the formal sense of the generated language, i.e., $L_j \stackrel{\text{def}}{=} L(G_j^R) = L(G_j^I)$ for $j = 1, 2, 3$. Here, $L(G)$ denotes the language generated by the grammar G . (This equivalence may need to be relaxed for practicality, as we will discuss in the paper.) Further, one should expect language-inclusion ordering for the JLS versions, i.e., $L_1 \subseteq L_2 \subseteq L_3$.

These straightforward properties for *language inclusion or equivalence* should be coupled with *correspondences on derivation trees* in order to be operationally useful. For instance, consider a Java tool based on G_1^I ; how would one be expected to revise the tool so that it is in compliance with G_2^I , perhaps even without yet covering constructs that are new in L_2 ? The revision would be feasible if there was a correspondence (in fact, a relation) $C_{1,2}^I \subseteq T(G_1^I) \times T(G_2^I)$. Here, $T(G)$ denotes the set of all derivation trees for the grammar G . The intensional definition of such a correspondence would allow one to understand one grammar *structurally* in terms of the other. Any pair of the 6 JLS grammars should be related in that manner.

The *actual* consistency of the Java standard

We have found that *none* of the expected language-inclusion or -equivalence properties hold, not even when making reasonable concessions to practicality. There are at least 34 “correctness issues” in the JLS (subject to a counting scheme to be discussed). It appears that the producer of the standard should have been very interested in avoiding such flaws. So given the importance of the Java standard, also given the amount of scrutiny that went into its preparation, one wonders how such a systematic lack of consistency could have happened? The answer is relatively simple. First, language inclusion (and hence equivalence) is not decidable for context-free grammars. Hence, there is no straightforward check that could have been applied. Second, in practice, grammar design and evolution is a manual process: knowledgeable engineers design or evolve grammar productions, as they see fit. They may use simple tools to check the grammar for basic well-formedness or compliance with a grammar class. They may also test a parser derived from the grammar. However, such measures cannot guarantee the expected language-inclusion or -equivalence properties, and oversights can happen all too easily. It goes without saying that the JLS also lacks any sort of explicit representation for the desirable correspondences between derivation trees that we motivated above.

A strong case for model-driven techniques

The consistency of a system of grammars can be maintained by means of grammar transformations as follows. For any version j , the more implementable grammar G_j^I is to be derived, transformationally, from G_j^R . If this transformation, referred to as $X_j^{R,I}$, only uses “language-preserving” operators, then language equivalence $L(G_j^R) = L(G_j^I)$ holds by construction. Also, the two grammars would be immediately related in structural terms because $X_j^{R,I}$ directly defines a relation $C_j^{R,I} \subseteq T(G_j^R) \times T(G_j^I)$. (Here we assume that any operator for grammar transformation also offers an interpretation of a transformation at the level of derivation trees.) Likewise, the more readable grammar of any version $j = 2, \dots$ is to be derived, transformationally, from the previous version $j - 1$. If this transformation, referred to as $X_{j-1,j}^R$, only uses “language-preserving and -increasing” operators, then language inclusion $L(G_{j-1}^R) \subseteq L(G_j^R)$ holds again by construction.

In the present paper, we describe our efforts to apply this model-driven approach retrospectively. That is, we have recovered the various transformations $X_1^{R,I}$, $X_2^{R,I}$, $X_3^{R,I}$, $X_{1,2}^R$ and $X_{2,3}^R$. These recovered transformations always consist of some part that describes intended differences (in particular, style differences and evolutionary differences) as well as accidental differences (i.e., inconsistencies). We also had to recover the JLS grammars in the first place because they are buried in the JLS documents.¹

¹ The complete JLS effort including all the involved sources, transformations, results, and tools is publicly available through <http://sourceforge.net/projects/slps/>; see [topics/java/lci](http://sourceforge.net/projects/slps/topics/java/lci) in particular.

Contribution This is the first time that *correspondences between sized grammars have been mechanized*. In this sense, the present work is the principled case study for the method of grammar convergence that we have introduced recently [20]. Grammar convergence is our lightweight verification method for relationships between scattered grammar knowledge. The method combines a suitable grammar format with means for grammar comparison and transformation.

Roadmap §2 introduces the JLS corpus including the contained grammars. §3 describes that part of our reverse engineering effort which extracts the raw grammars from the JLS documents. §4 is the main section of paper; it describes the transformational approach to recovering and representing correspondences between the JLS grammars. §5 measures the JLS case by means of simple metrics. §6 discusses related work, and the paper is concluded in §7.

2 The JLS corpus

The “more readable” grammar (i.e., G_j^R) is always scattered over the main chapters of a JLS document; we refer to them as *doc1–3* from here on. The “more implementable” grammar (i.e., G_j^I) is always given, *en bloc*, in a late section — a de-facto appendix; we refer to them as *app1–3* from here on. In the following, we gather some basic knowledge about these grammars, and their embedding into the JLS.

2.1 Grammar classes and correspondences

JLS1: The grammar class of *app1* is described by saying that [9, §19] the grammar has “*been mechanically checked to insure that it is LALR(1)*”. The correspondence between *doc1* and *app1* is briefly described by saying [9, §2.3] that *doc1* is “*very similar to*” *app1* “*but more readable*”.

JLS2: The second edition of the JLS [10, “Preface to the 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 grammars of JLS1 and JLS2 are nowhere related explicitly. Upon cursory examination we came to conclude that *doc1* and *doc2* are strikingly 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. Instead, *app2* is only said [10, §18] to be “*the basis for the reference implementation*”.

JLS3: JLS3 extends JLS2 in numerous ways [11, Preface]: “*Generics, annotations, asserts, autoboxing and unboxing, enum types, foreach loops, variable arity methods and static imports have all been added to the language*”. Again, the grammars of JLS2 and JLS3 are nowhere related explicitly, and again, cursory examination suggests that *doc2* and *doc3* are strikingly similar (modulo the extensions to be expected). This time, *app2* and *app3* also bear strong resemblance. No definitive grammar-class claim is made, but an approximation thereof that suggests that *app3* has definitely departed from LALR(1), i.e., *app3* is said [11, §18] to be “*not an LL(1) grammar, though in many cases it minimizes the necessary look ahead.*”

	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

We list all spotted grammar-class claims; we also add observations about iteration style (“lists”) that we made during cursory examination. This table makes clear that we need to bridge the gap between different iteration styles (which is relatively simple) but also different grammar classes (which is more involved) — if we want to establish the correspondences between the different grammars by effective transformations.

Fig. 1. Grammar classes and iteration style for the JLS grammars.

	Productions	Nonterminals	Tops	Bottoms
<i>app1</i>	282	135	1	7
<i>doc1</i>	315	148	1	9
<i>app2</i>	185	80	6	11
<i>doc2</i>	346	151	1	11
<i>app3</i>	245	114	2	12
<i>doc3</i>	435	197	3	14

The metrics were automatically derived from the extracted grammars. Terminology: a *top nonterminal* is a nonterminal that is defined but never used; a *bottom nonterminal* is a nonterminal that is used but never defined; see [18,23] for these terms.

Fig. 2. Simple metrics for the extracted JLS grammars.

2.2 Simple grammar metrics

The major differences between the numbers of productions and nonterminals for the two grammars of any given version (see Fig. 2) is mainly implied by the different grammar classes and iteration styles. The decrease of numbers for the step from *app1* to *app2* is explainable with the fact that an LALR(1) grammar was replaced by a new development (which does not aim at LALR(1)). Otherwise, the obvious trend is that the numbers of productions and nonterminals go up with the version number.

The *difference in numbers of top-nonterminals is definitely a problem indicator*. There should be only one top-nonterminal: the actual start symbol of the Java grammar. Any additional case of an “unused” nonterminal does not make sense. At first glance, the *difference in numbers of bottom-nonterminals may be reasonable* because a bottom nonterminal may be a lexeme class — those classes are somewhat of a grammar-design issue. However, a review of the nonterminal symbols rapidly reveals that some of them correspond to (undefined) categories of compound syntactic structures.

2.3 The source format for grammars

A JLS document is basically a structured text document with embedded grammar sections. The JLS is available electronically in HTML and PDF format. Neither of these formats was designed with convenient access to the grammars in mind. We will favor the HTML format here. The grammar format slightly varies for the different JLS grammars and versions; we collect bits from different documents and sections — in particular from [9,10,11, §2.4] and [10,11, §18].

Grammar fragments are hosted by `<pre> . . . </pre>` blocks in the JLS documents. Terminal symbols appear in fixed font (as in `<code>class</code>`). Nonterminal symbols appear in italic type (as in `<i>Expression</i>`). Nonterminal symbols are

alphanumeric and start in upper case. Alternatives start in a new line and they are indented. However, there is also a form of top-level choices with atomic branches; they are to be announced with the keywords “one of”. Further, there is yet another form for choices: $x_1 | \dots | x_n$ denotes a choice over the x_i ; such a choice can appear in a nested position subject to grouping “(...)”. Optionality can be expressed by using a subscripted suffix “opt” for nonterminal symbols (as in `Expression_{opt}`). Alternatively, one can use the form $[x]$ which denotes zero or one occurrences of x . There is also a form for lists: $\{x\}$ denotes zero or more occurrences of x . JLS makes no use of a construct for one or more occurrences (think of “+” vs. “*”). Finally, we should mention line continuation; it allows spreading one top-level alternative over several lines [11, §2.4]: “A very long right-hand side may be continued on a second line by substantially indenting this second line”.

Example 1. Here is a grammar fragment taken from [10, §4.2]:

```

<i>NumericType:
    IntegralType
    FloatingPointType

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

```

The fragment illustrates the vertical aggregation of alternatives, c.f., nonterminal `NumericType`. It also puts to use the “one-of”-based aggregation of alternatives, c.f., nonterminal `IntegralType`. 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 tag ends up on a new line.

Assumed grammar format for exposition The markup-based source format is inconvenient for presentation in the paper. We prefer to show grammar fragments in a pretty-printed format with nonterminals in italic type, terminals enclosed in double quotes, and operators “?”, “*” and “+” for optionality and lists. Top-level choices (alternatives) are represented as a set of productions with the same left-hand side. Elisions are shown as “...”. For instance, Ex. 1 is pretty-printed as follows:

<i>NumericType: IntegralType</i>	<i>IntegralType: "byte"</i>
<i>NumericType: FloatingPointType</i>	<i>IntegralType: "short"</i>
	...

3 Grammar extraction

Ideally, a language specification should be developed in a model-driven fashion, where the final document contains a strongly checked grammar that has not been tampered with. In contrast, the JLS case requires the extraction of the raw grammars from the manually fabricated JLS documents. We will briefly summarize this part of the project. The grammar segments of the document have to be analysed by a *non-classic parser* in

order to deal with irregularities of the input format — such as those illustrated in Ex. 1. The parser also needs to work around violations of XML/HTML well-formedness. Below, we discuss the essential phases of the non-classic parser.

3.1 The preprocessing phase

Upon collecting the `<pre>...</pre>` blocks, they are processed such that a dictionary is constructed: the keys are the left-hand side nonterminal symbols; the value of each entry is a collection (an array) of all the top-level alternatives for the nonterminal symbol at hand. Here we assume the following requirements:

Tag elimination The input notation interleaves tags with proper grammar structure. In order to prepare for classic parsing, we eliminate the tags by constructing distinct lexemes for terminals, nonterminals and metasymbols.

Indentation elimination The input notation relies on indentation to mark off alternatives and continuation lines. In order to prepare for classic parsing, we fuse multi-lines, and we record lists of alternatives as arrays.

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 classic (precise) parsing is expected to succeed.

We use a stateful scanner to meet “tag elimination”. There are three states: *italic* upon opening `<i>` tag (or ``), *fixed* upon opening `<code>`, *default* when no tag is open. For any given (space-separated) token, the stateful scanner needs to decide whether it recognizes a terminal, a nonterminal or a metasymbol (of the grammar notation). 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 inevitable “error-recovering” decision is to map a non-alphanumeric token (that does not match any metasymbol) to a terminal — even when it is tagged with `<i>...</i>`. An example of an “arbitrary” decision is to map bars without markup to metasymbols rather than terminals. The “arbitrary” decisions were optimized to recover syntactical correctness and to necessitate the least number of subsequent transformations for correction.

3.2 The error-recovery phase

At this point, we face syntax errors (with regard to the syntax of the grammar notation) as well as “obvious” semantic errors (in the sense of the language generated by the grammar). We speak of obvious errors because they are detected mechanically without taking into account Java knowledge. The parser addresses both kinds of errors by using an error-recovery approach which relies on a uniform, rule-based mechanism to transform nonterminal definitions exhaustively.

For instance, consider a rule “nonterminal to terminal” which applies when the symbol n of an undefined nonterminal starts in lower case (where they would normally start in upper case). Such a nonterminal reference is converted to a terminal. Another rule is “recover optionality” which applies when markup for the subscript “opt” is missing. That is, when there is an undefined nonterminal with a symbol n that ends on “opt”,

and there is a defined nonterminal with a symbol n' that equals n with the “opt” postfix removed, then all references to n are replaced by optional references to n' .

Example 2. Consider the following fragment that appears in [10, §15.9]:

```
ClassInstanceCreationExpression: Primary " . " "new" Identifier ( ... ) ClassBodyopt
```

The rule “recover optionality” applies and results in the following definition:

```
ClassInstanceCreationExpression: Primary " . " "new" Identifier ( ... ) ClassBody?
```

3.3 The removal-of-duplicates phase

The JLS documents (deliberately) repeat grammar segments. Hence, there is a phase to detect and remove such duplication. This is a simple problem because all alternatives for a given nonterminal symbol from anywhere in the JLS document have been aggregated in the same array, and hence doubles can be spotted easily.

Example 3. The following definition appears both in [10, §8.3] and [10, §14.4]:

```
VariableDeclaratorId: Identifier
```

```
VariableDeclaratorId: VariableDeclaratorId " [ " " ] "
```

It is worth mentioning that both occurrences used different markup in the JLS document, but those differences were neutralized during the preprocessing phase. Also, both occurrences were subject to error recovery because the “[...]” symbols had been parsed as metasymbols initially (due to the lack of markup for terminals).

3.4 The classic parsing phase

Finally, the dictionary structure (mapping nonterminals to arrays of alternatives) can be analysed by a classic parser. Our implementation dumps the extracted grammar immediately in accordance to an XML-based grammar interchange format so that generic grammar tools for comparison and transformation can take over.

4 Grammar convergence

We will now describe the transformational approach to establishing the correspondences between the JLS grammars. We will illustrate intended and accidental differences between the different JLS grammars and show the grammar transformations that neutralize those differences.

4.1 The convergence tree

The plan must be to devise transformations such that the two grammars per JLS version are “converged to a common denominator” (see the nodes *jls1–3* in Fig. 3), and all three versions are “converged” (in pairwise fashion) to account for inter-version differences — the extensions to the Java language in particular (see the 4 nodes *jls12* and *jls123* as well as *doc12* and *doc123* in the figure).

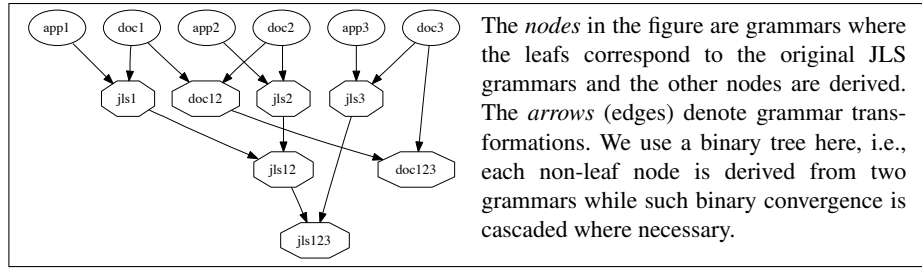


Fig. 3. Binary convergence tree for the JLS grammars.

When deriving *jls1–3*, we give preference to the direction of deriving the “more implementable” grammar from the “more readable” grammar while some refactoring and correction may still be applied to the “more implementable” grammar. This preference reflects the general rule that an implementation-oriented artifact should be derived from a design-oriented artifact — rather than the other way around. (Incidentally, this direction is also easier to handle by the available transformation operators.)

When relating the different JLS versions, we adopt the redundant approach to relate the common denominators *jls1–3* in one cascade (see the nodes *jls12* and *jls123*), but also the “more readable” grammars *doc1–3* in another cascade (see the nodes *doc12* and *doc123*). The latter cascade is presumably more important because *doc1–3* are known to be structurally similar; hence convergence can be expected to be cheap. The additional cascade is merely seen as a sanity and scalability check for the method.

4.2 Workflow of grammar convergence

Fig. 4 sketches the overall workflow which we apply to any pair of grammars that need to be related to each other. Foremost, this is an iterative process that stops when the two involved grammars have become structurally equal. The transformations are accumulated along the way; they present the actual result of the convergence process. The transformation steps are driven by the differences revealed by grammar comparison. In each step, one has to pick a difference as well as one of the two grammars to adapt.

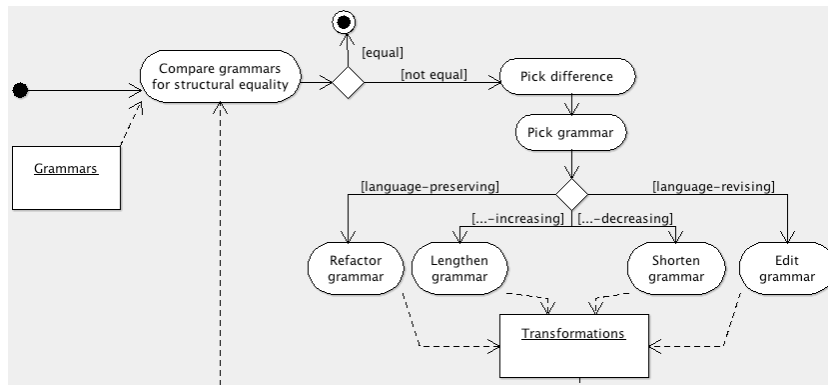


Fig. 4. (A UML activity diagram for the informal) workflow of grammar convergence.

As of writing, grammar comparison is trivially defined as follows. *Nonterminals are matched by name (i.e., nonterminal symbols). Productions (alternatives) are matched by name (i.e., production labels), if present, and by precise structure otherwise. All unmatched nonterminals and productions are reported.*

When the two given grammars are structured differently, then they can be aligned by *language-preserving* transformations (say, *refactorings*). When one grammar is more permissive than the other with regard to a syntactic category or a specific construct, then such a difference is neutralized by a *language-increasing* or *-decreasing* transformation. To parallel the notion of refactoring, we say that we *lengthen* or *shorten* the grammar. When the two given grammars disagree more arbitrarily, then a less disciplined transformation for *grammar editing* must be used, e.g., for replacing a production.

Refactoring, lengthening, shortening, and editing are not to be mixed at will. Instead, refactoring should be exhausted first to align the syntactical categories of the two grammars by refactoring, and to minimize the scopes of unresolvable differences. Only then, non-refactoring operations should be invoked to resolve those unresolvable differences in the smallest possible scopes.

4.3 Grammar refactoring

There are operators to *fold* and *unfold* nonterminal definitions, to *extract* and *inline* specific nonterminals, to *factor* and *distribute* grammar expressions, to *massage* grammar expressions according to algebraic laws, and to *alter iteration style* (recursion vs. “*”).

Several of the grammar transformation operators can be viewed as disciplined forms of “replacement”, i.e., they are invoked by the form $o(x, x')$ where o is the operator in question, x is the grammar expression to be located in the input, and x' is the corresponding replacement. For instance, the *factor* operator is applied to an expression and a *factored* variation; the *massage* operator is applied to an expression and an *algebraically equivalent* variation based on a fixed set of laws.

Other grammar transformation operators apply a fixed operation to a specific nonterminal, and hence, they can be invoked by the form $o(n)$ where o is the operator in question, and n is the nonterminal to be affected. For instance, *inlining* a nonterminal can be requested in this manner. Also, the conversion from a recursive definition-based style of iteration to the use of the regular operators “*” and “+” can be requested in this manner. (We call the latter step “deyaccification” [17,14].)

The following two samples of transformations are taken from a refactoring script that aligns *doc2* and *app2*. The JLS case involves many hundreds of such small refactoring steps; see §5 for the discussion of measures. Grammar convergence, for the larger part, is a refactoring effort.

Example 4 (Transformation operators factor and massage).

```
factor(
  (Block | ("static" Block)),
  ((ε | "static") Block));
massage(
  (ε | "static"),
  "static?");
```

In *doc2*, there are distinct alternatives for blocks vs. static blocks. In contrast, in *app2*, these forms appear in a factored manner. Hence, the *factor* operator is used to factor out the shared reference to *block*. Then, the *massage* operator changes the style of expressing optionality of the keyword “static”.

Example 5 (Transformation operators deyaccify and inline).

```
deyaccify(  
  ClassBodyDeclarations);  
inline(  
  ClassBodyDeclarations);  
massage(  
  ClassBodyDeclaration+?,  
  ClassBodyDeclaration* );
```

In *doc2*, recursion-based style of iteration is used. For instance, there is a recursively defined nonterminal *ClassBodyDeclarations* for lists of *ClassBodyDeclaration*. In contrast, in *app2*, the list form “*” is used. Deyaccification replaces the recursive definition of *ClassBodyDeclarations* by *ClassBodyDeclaration*⁺. The nonterminal *ClassBodyDeclarations* is no longer needed, and hence inlined. In fact, the list of declarations was optional, and hence “+” and “?” can be simplified to “*”.

4.4 Grammar lengthening / shortening

There are operators to *widen* and *narrow* occurrence constraints (e.g., to change “+” to “*” and vice versa), to *add* and *remove* alternatives (say, productions), and to replace a nonterminal occurrence by one of its productions and vice versa (to which we refer as *downgrading* and *upgrading*).

Example 6 (Widening an occurrence constraint).

```
widen(  
  "static",  
  "static"?,  
  ClassBodyDeclaration);
```

This transformation is part of a script that captures the delta between JLS1 and JLS2. The particular widening step enables *instance* initializers in class bodies (where only *static* initializers were admitted before).

The example also demonstrates that transformation operators may carry an extra argument to describe the *scope of replacement*. By default, the scope is universal: all matching expressions in the input grammar would be affected. Selective scopes are nonterminal definitions (specified by a nonterminal — as in the example) or productions (specified by a production label).

Example 7 (Adding an alternative).

```
add(  
  ConstantModifier: Annotation);
```

This transformation is part of a script that captures the delta between JLS2 and JLS3. In JLS2, a constant modifier can be “public” or “static” or “final”. JLS3 offers the additional option *Annotation*.

When grammars of different versions are related, then language-increasing or -decreasing transformations are clearly to be expected. As a matter of discipline, we prefer to describe the delta by “lengthening” a given version to a successor rather than by “shortening” in the other direction.

However, the JLS case also requires the use of language-increasing or -decreasing transformations in other situations. Consider the overall situation that the two grammars of a given JLS version do not achieve language equivalence for some given syntactic category, but only language inclusion. In some cases, such a difference may be purely accidental, and it needs to be *corrected* by shortening or lengthening the grammar. However, there are also cases where lack of equivalence may be related to *grammar optimization*. We use this term to refer to deliberate design decisions by knowledgeable

engineers who want to make a grammar more readable, more implementable, or more compact at times while the deviation of the actual language from the intended language is considered more or less negligible.

Example 8 (Grammar optimization).

<i>app2</i>		
<i>Modifier</i> : "public" ...		
<i>doc2</i>		
<i>ClassModifier</i> : "public" ...		
<i>FieldModifier</i> : "public" ...		
<i>InterfaceModifier</i> : "public" ...		
<i>MethodModifier</i> : "public" ...		

Grammar optimizations are in fact the reason that language equivalence is too strict of a requirement for a pair of “more readable” and “more implementable” grammars, as we assumed in the introduction. We suggest that a language specification should explicitly call out optimizations so that they are not confused with correctness issues. In this manner, one is also prepared to reason about different implementations that may or may not apply certain optimizations.

4.5 Grammar editing

There are operators to *undefine* a nonterminal (i.e., to abandon its definition), to *replace* a grammar expression in a unconstrained manner, to *inject* new components into a production and to *project* away existing components. The operators *inject* and *project* can be invoked by a form such that a *grammar expression with markers* (as in $a c$) is passed as a parameter. These markers highlight the components to be added or removed, respectively, and thereby state the intention of the operator application more explicitly.

Example 9 (Correcting expression syntax in app2 and app3).

<i>Incorrect expression syntax</i>			
<i>Expression2</i> : <i>Expression3</i> [<i>Expression2Rest</i>]			The <i>app2</i> and <i>app3</i> grammars define the Java expression syntax by means of layers, i.e., there are several nonterminals <i>Expression1</i> , <i>Expression2</i> , ... for the different priorities. We are concerned with one layer here. The second rule for <i>Expression2Rest</i> contains an offending occurrence of <i>Expression3</i> which needs to be projected away. This issue was revealed by comparison with the <i>doc2</i> and <i>doc3</i> grammars (subject to prior refactoring).
<i>Expression2Rest</i> : (<i>InfixOp</i> <i>Expression3</i>)*			
<i>Expression2Rest</i> : <i>Expression3</i> "instanceof" <i>Type</i>			
<i>Language-revising transformation</i>			
project (<i>Expression2Rest</i> : < <i>Expression3</i> > "instanceof" <i>Type</i>);			
<i>Corrected expression syntax</i>			
<i>Expression2</i> : <i>Expression3</i> [<i>Expression2Rest</i>]			
<i>Expression2Rest</i> : (<i>InfixOp</i> <i>Expression3</i>)*			
<i>Expression2Rest</i> : "instanceof" <i>Type</i>			

Example 10 (Correcting statement syntax in app2).

inject(
Statement: "break" Identifier? < ";" >);

The production for the *break* statement lacks the semicolon which is injected accordingly.

The two examples above are concerned with incorrect syntax of the kind that the intended language is not captured proper. There are also situations where incorrect syntax merely arises due to the limits of grammar extraction. That is, when markup is incorrect or missing, then error recovery is not always able to guess the right decision. Hence, “error recovery” needs to be continued to some extent in the transformation phase; see the following example.

Example 11 (Extraction — post-processing for app3).

replace(
BlockStatements* ,
" { " BlockStatements " } ");

The source format defines curly brackets without markup to express iteration. In the example at hand, the curly brackets were meant as terminals though, but markup `<code> . . . </code>` was missing. The incorrect list construct is replaced accordingly.

5 Measurements of the JLS case

Figures 5–7 measure the extraction effort, the derived grammars, and the involved grammar transformations. Overall, the extraction measures indicate how much work is needed to obtain the raw grammars in the first place. A model-driven approach to language specification would eliminate all such effort.

The measures for the involved transformations include “lines of code” just as a very informal indication. (We count lines of code for the XML representation of the grammar transformations.) More usefully, the number of transformations directly refers to the number of *applications of transformation operators*. 32 different operators are used in the JLS case; many of them were mentioned in sections 4.3–4.5. About three quarters of the transformations are semantics-preserving. The remaining quarter is mainly dedicated to semantics-increasing or -decreasing transformations with only 2% of semantics-revising transformations.

The (large) number of transformations illustrates the simplicity of the transformation operators at hand; it does not usefully quantify the number of “issues” that had to be addressed. Roughly, we think of an “issue” as a difference to be reported by grammar comparison — assuming that comparison operates in as small scopes as possible, and summarizes recurring differences (such as name differences) as one. We have manually tagged all transformations with issues without though including any “pure refactoring” issues — also because they are hard to count precisely. We speak of a *recovery* issue when the grammar extractor was misled by inconsistent markup. We speak of an *extension* issue when semantics-increasing transformations are required to model a new construct added by a version. We refer to §4.4 for *optimization* issues. We speak of *correction* for all remaining issues that do not suffice with refactoring.

The counting of issues in Fig. 7 is affected by the JLS convergence tree that uses two redundant cascades. That is, part of the convergence transformations (and hence,

	app1	app2	app3	doc1	doc2	doc3	Total
Arbitrary lexical decisions	2	109	60	1	90	161	423
Well-formedness violations	5	0	7	4	11	4	31
Indentation violations	1	2	7	1	4	8	23
Recovery rules	3	12	18	2	59	47	141
o Match parentheses	0	3	6	0	0	0	9
o Metasymbol to terminal	0	1	7	0	27	7	42
o Merge adjacent symbols	1	0	0	1	1	0	3
o Split compound symbol	0	1	1	0	3	8	13
o Nonterminal to terminal	0	7	3	0	8	11	29
o Terminal to nonterminal	1	0	1	1	17	13	33
o Recover optionality	1	0	0	0	3	8	12
Purge duplicate definitions	0	0	0	16	17	18	51
Total	11	123	92	24	181	238	669

The number of “arbitrary” lexical decisions (c.f., §3.1) is considerable; the favored decisions minimize the work to be done during transformation. The numbers of (XML/HTML) well-formedness and indentation violations provide other measures of the consistency of format usage. We list the recovery rules (c.f., §3.2) without further explaining them here. Finally, the number of definitions with duplicates is also listed (c.f., §3.3).

Fig. 5. Formatting issues resolved by the extraction of the JLS grammars.

	Productions	Nonterminals	Tops	Bottoms
<i>jls1</i>	278	132	1	7
<i>jls2</i>	182	75	1	7
<i>jls3</i>	236	109	1	7
<i>jls12</i>	182	75	1	7
<i>jls123</i>	236	109	1	7
<i>doc12</i>	347	152	1	7
<i>doc123</i>	440	201	1	7

We show the same, simple metrics for the derived grammars as we originally presented for the leafs of the convergence tree; c.f., Fig. 2. Top- and bottom-nonterminals are consolidated now. In the case of the “common denominators” *jls1–3*, the numbers of nonterminals and productions reflect that these grammars were derived to be similar to *app1–3*. Similar correlations hold for the “inter-version” grammars in the rest of the table.

Fig. 6. Simple metrics for the derived JLS grammars.

	jls1	jls2	jls3	jls12	jls123	doc12	doc123	Total
Number of lines	600	4807	9469	4285	2934	1491	3072	26658
Number of transformations	62	367	538	287	120	70	133	1577
o semantics-preserving	40	278	398	235	87	25	73	1136
o semantics-increasing or -decreasing	22	78	127	50	32	38	56	403
o semantics-revising	—	11	13	2	1	7	4	38
Number of issues	8	38	47	25	17	32	40	207
o recoveries	—	7	8	—	—	7	4	26
o corrections	5	22	22	2	—	10	7	68
o extensions	—	—	—	17	14	15	28	74
o optimizations	3	9	17	6	3	—	1	39

Fig. 7. Transformation of the JLS grammars — effort metrics and categorization

part of the issues that they address) are conceptually shared among the two convergence cascades. (Because of reusability limits for the transformations, two variations are sometimes needed — one for each cascade.) A safe lower bound for the number of correctness issues is therefore $\lceil 68/2 \rceil = 34$. We also note that our proposal for a category of “optimizations” helps to keep down the number of correctness issues.

6 Related work

The JLS effort is most directly related to previous work on *grammar recovery* or re-engineering of syntax definitions [23,18,14,7,1]. The main difference is that the JLS case required the recovery of *correspondences* between different grammars, whereas grammar recovery focuses on the engineering process of deriving a quality grammar (e.g., one that can be used for parsing) from some raw grammar. That is, the JLS case (and grammar convergence, in general) involves elements of comparison and “transformation towards structural grammar equality” that are not present in grammar recovery. A notable counterexample is [3] where *precedence rules* are recovered from *multiple* grammars and checked for consistency.

Grammar transformations [4,5,17,19,27] provide the heavy lifting of grammar convergence. In fact, we leverage grammar transformations that are similar to those in our earlier work on grammar recovery [17,19]. The actual operator suite has been designed to maximize the number of grammar correspondences that can be expressed as (nearly) language-preserving transformations. For instance, in §4.3, we mentioned *factor* and *distribute* operators that specifically arise from the need to align a less factored (i.e., more readable) grammar with a more factored (i.e., more implementable) grammar. While the JLS situation required the *recovery* of grammar correspondences, ultimately, those correspondences should be maintained along grammar development. For instance, one may consider the “more readable” grammar as the primary artifact, and generate the “more implementable” grammar in a semi-automatic, model-driven manner [15].

The JLS case involves another element of grammar recovery, i.e., the *extraction of raw grammars from some other software artifact* — from the language documentation in this case, which is comparable to the situation of [18]. Each such extraction effort tends to require unique techniques due to the idiosyncrasies of the artifact at hand. The JLS case required a designated scheme of error recovery which deviates from standard practice of syntax and semantics error recovery in compiler construction [13,2,25]. The extractor begins by constructing an intermediate data structure for the grammar segments without yet engaging in precise parsing. Such a liberal scheme is reminiscent of robust parsing as needed in IDEs or reverse and re-engineering [16,21,24]. Recovery rules are applied to the intermediate data structure; they are implemented by regular expression matching on plain token sequences for the definitions of nonterminals. The extraction parser also involves recovery behavior to handle ill-formed XML/HTML — this is a folklore idea [12].

Grammar convergence involves *grammar comparison* in order to detect structural differences, and to inform thereby the process of transformation. Grammar comparison can be compared with schema matching in ER/relational modeling [22,6] (with data integration as an important application) as well as model and metamodel matching/diffing in model-driven engineering [28,26,8] (specifically in the context of model/metamodel evolution) — except that our current approach to comparison (see §4.2) is trivial compared to the cited work. A trivial approach is sufficient here because we only expect help in devising grammar transformations; we do not expect any sort of (semi-) automatic restructuring and conflict resolution at this point. Obviously, a more automated convergence approach may be an interesting topic for future work.

7 Concluding remarks

This is the first published record of representing the correspondences between sized grammars that serve different audiences (language users and implementers) and that capture different versions of the language — in a mechanized fashion. The approach carefully distinguishes language extensions, grammar restructuring, grammar optimization, and correction of inconsistencies.

While the approach succeeded at scrutinizing all correspondences in the Java standard including the discovery of presumably all inconsistencies, more work is needed to arrive at a best practice. The effort for developing the transformation framework and for accumulating the actual transformations is perhaps too high and requires quite specific skills. We were able to leverage substantial background on grammar engineering, and it still took us about 10 men months to get to the closure that we report here.

There is a *productivity problem*. The transformation part of grammar convergence requires substantial effort to recover “structural grammar equality under fold/unfold manipulation” by the use of low-level transformation operators. Higher-level operators are needed. A related subject is the partial inference of transformations. Further, the tool support for grammar transformations would need to go beyond current batch execution of transformations. Interactive and incremental transformations are needed. Also, comparison, transformation, and error diagnosis would need to be integrated.

There is also a *formal problem*. Currently, the justification of using operators other than for refactoring is based on informal reasoning. In principle, one may accidentally engage in editing operations even when the grammars are equivalent. Also, the scope of editing may not be minimal. We are working on effective, practically meaningful, formal criteria to address this problem. Here we are obviously challenged by the fundamental property that grammar equivalence is undecidable.

References

1. T. Alves and J. Visser. A case study in grammar engineering. In *Post-proceedings of 1st International Conference on Software Language Engineering (SLE'08)*, LNCS. Springer, 2009. To appear.
2. D. Barnard and R. Holt. Hierarchic Syntax Error Repair for LR Grammars. *International Journal of Computer and Information Sciences*, 11(4):231–258, 1982.
3. E. Bouwers, M. Bravenboer, and E. Visser. Grammar Engineering Support for Precedence Rule Recovery and Compatibility Checking. *ENTCS*, 203(2):85–101, 2008.
4. T. Dean, J. Cordy, A. Malton, and K. Schneider. Grammar Programming in TXL. In *Proceedings, Source Code Analysis and Manipulation (SCAM'02)*. IEEE, 2002.
5. T. Dean, J. Cordy, A. Malton, and K. Schneider. Agile Parsing in TXL. *Journal of Automated Software Engineering*, 10(4):311–336, 2003.
6. H. H. Do and E. Rahm. Matching large schemas: Approaches and evaluation. *Information Systems*, 32(6):857–885, 2007.
7. E. B. Duffy and B. A. Malloy. An Automated Approach to Grammar Recovery for a Dialect of the C++ Language. In *Proceedings, 14th Working Conference on Reverse Engineering (WCRE 2007)*, pages 11–20. IEEE, 2007.
8. J.-R. Falleri, M. Huchard, M. Lafourcade, and C. Nebut. Metamodel Matching for Automatic Model Transformation Generation. In *Proceedings of Model Driven Engineering Languages and Systems (MoDELS 2008)*, volume 5301 of LNCS, pages 326–340. Springer, 2008.

9. J. Gosling, B. Joy, and G. L. Steele. *The Java Language Specification*. Addison-Wesley, 1996. Available at java.sun.com/docs/books/jls.
10. 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.
11. 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.
12. Q. T. Jackson. Efficient formalism-only parsing of XML/HTML using the λ -calculus. *ACM SIGPLAN Notices*, 38(2):29–35, 2003.
13. 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.
14. 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.
15. F. Jouault, J. Bézivin, and I. Kurtev. TCS:: a DSL for the specification of textual concrete syntaxes in model engineering. In *GPCE '06: Proceedings of the 5th international conference on Generative programming and component engineering*, pages 249–254. ACM, 2006.
16. 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.
17. 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.
18. R. Lämmel and C. Verhoef. Semi-automatic Grammar Recovery. *Software—Practice & Experience*, 31(15):1395–1438, 2001.
19. 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.
20. R. Lämmel and V. Zaytsev. An Introduction to Grammar Convergence. In *Proceedings of 7th International Conference on integrated Formal Methods (iFM 2009)*, LNCS. Springer, 2009. Proceedings to appear.
21. L. Moonen. Generating Robust Parsers using Island Grammars. In *Proceedings, Working Conference on Reverse Engineering (WCRE'01)*, pages 13–22. IEEE, Oct. 2001.
22. E. Rahm and P. A. Bernstein. A survey of approaches to automatic schema matching. *VLDB Journal*, 10(4):334–350, 2001.
23. 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.
24. 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.
25. P. N. van den Bosch. A bibliography on syntax error handling in context free languages. *SIGPLAN Notices*, 27(4):77–86, 1992.
26. S. Wenzel and U. Kelter. Analyzing model evolution. In *ICSE '08: Proceedings of the 30th international conference on Software engineering*, pages 831–834. ACM, 2008.
27. D. Wile. Abstract syntax from concrete syntax. In *Proceedings, International Conference on Software Engineering (ICSE'97)*, pages 472–480. ACM Press, 1997.
28. Z. Xing and E. Stroulia. Refactoring Detection based on UMLDiff Change-Facts Queries. In *WCRE '06: Proceedings of the 13th Working Conference on Reverse Engineering*, pages 263–274. IEEE Computer Society, 2006.