Recovery, Convergence and Documentation of Languages

by
Vadim Zaytsev

September 14, 2010

VRIJE UNIVERSITEIT

Recovery, Convergence and Documentation of Languages

ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad Doctor aan
de Vrije Universiteit Amsterdam,
op gezag van de rector magnificus
prof.dr. L.M. Bouter,
in het openbaar te verdedigen
ten overstaan van de promotiecommissie
van de faculteit der Exacte Wetenschappen
op woensdag 27 oktober 2010 om 15.45 uur
in de aula van de universiteit,
De Boelelaan 1105

door

Vadim Valerievich Zaytsev

geboren te Rostov aan de Don, Rusland

promotoren: prof.dr. R. Lämmel
prof.dr. C. Verhoef

Dit onderzoek werd ondersteund door de Nederlandse Organisatie voor Wetenschappelijk Onderzoek via:
This research has been sponsored by the Dutch Organisation of Scientific Research via:

NWO 612.063.304 *LPPR: Language-Parametric Program Restructuring*

Acknowledgements

Working on a PhD is supposed to be an endeavour completed in seclusion, but in practice one cannot survive without the help and support from others, fruitful scientific discussions, collaborative development of tools and papers and valuable pieces of advice.

My work was supervised by Prof. Dr. Ralf Lämmel and Prof. Dr. Chris Verhoef, who often believed in me more than I did and were always open to questions and ready to give expert advice. They have made my development possible.

LPPR colleagues — Jan Heering, Prof. Dr. Paul Klint, Prof. Dr. Mark van den Brand — have been a rare yet useful source of new research ideas.

All thesis reading committee members have dedicated a lot of attention to my work and delivered exceptionally useful feedback on the late stage of the research: Prof. Dr. Jean Bézivin, Dr. Jean-Marie Favre, Prof. Dr. Willem Jan Fokkink, Prof. Dr. Paul Klint, Dr. Steven Klusener. I am also grateful for Cor-Paul Bezemer and Toon Verwaest who provided proofreading and correcting services for the Dutch part of this thesis. There have been a lot of insightful discussions in the rooms and hallways of the Vrije Universiteit with Dr. Niels Veerman, Ernst-Jan Verhoeven, Łukasz Kwiatkowski and Johan Vincent de Vries.

I would like to thank my family that backed me up with complete support and encouragement through the years of research, especially my mother, Dr.ir. Liudmila Zaytseva; my grandmother, Dr. Svetlana Bocheva; my grandfather, Prof. Dr.ir. Alexander Bochev; my uncle, Dr. Michael Bochev and my godfather, Prof. Dr. Yuri Bashmakov, MD.

My close friends' understanding, respect and interest in my work was also among the most important things that kept me going: Dr. Alexander Gufan, Dr. Stanislav Tsykavy and Stanislav Rezhabek.

I have also been saved many times from depression and writer's blocks by good music. I cannot name all the artists responsible for that, but the most credit goes to Huddie Ledbetter, William Broonzy, Fulton Allen, Thomas McClennan and Bruce Springsteen.

Contents

List of Tables

List of Figures

List of Listings

Chapter 1

Introduction

1.1 Research context

A *language* is formally defined as a countable set of finite sequences of symbols from a given alphabet [2]. Conversely, anything that can be expressed or perceived as a set of symbol sequences, can be considered a language and treated with similar technology:

- ◇ programming languages such as Java, C#, COBOL, Python,
- ◇ mark-up languages such as XML, YAML, HTML, RTF, DITA, LaTeX, MediaWiki,
- ◇ data definition languages like DDL, XML Schema, ASN.1,
- ◇ command issuing languages such as JCL, DCL, Unix shells,
- ◇ fourth generation languages such as Informix, RPG, FoxPro,
- ◇ presentation and transformation languages such as CSS, XSL,
- ◇ query languages such as SQL, .QL, XQuery, JGrok, Rscript, JRelCal, JGraLab,
- ◇ modelling languages such as UML, SDL, Z notation, Promela,
- ◇ other domain-specific languages,
- ◇ application programming interfaces,
- ◇ libraries.

A language is commonly defined by a *grammar*. Most of language processing tools and methodologies rely on the *parsing* process, which analyses the source code according to the rules of the grammar. This places the grammar at the foundation of almost any language processing infrastructure. The software which input can be described by a grammar, is called grammar-based software or *grammarware*. There exist many kinds of grammarware, they will be listed in section 2.2. The grammars are engineered from scratch, reverse engineered from the tools that contain them implicitly, extracted from available sources. The list of possible artefacts bearing grammar knowledge includes language processor source code, language *documentation*, a codebase. The extraction process often comprises more than simple mapping activities, in this case we speak of grammar *recovery*, of which we will see several case studies in this thesis.

In practice there often exist several grammars for one language. There are those that are implicitly encoded in grammarware (e.g., source code of the parser), others are readily

available in explicit form (e.g., parser definition for a compiler generator). Grammars are also commonly *adapted* for their users' needs: for instance, during development of source code transformation tools [39]. In order to reduce engineering and maintenance effort, *grammar transformation* can be used to derive adapted grammars. In the case of one primary grammar and a set of secondary grammars derived from it, the former is usually called a *base-line* grammar.

Recovering several grammar variants from different sources is also common practice. These grammars are not necessarily equivalent in the formal sense. It is possible to employ grammar transformation, among other goals:

- ◇ to make sure that the compiler indeed implements the features listed in the specification of the language,
- ◇ to provide strong evidence that a code analysis tool set operates on the same dialect as the compiler,
- ◇ to argue that the X/O mapping (e.g., in data binding) is correct,
- ◇ to see that the new version of a language is indeed backwards compatible, or
- ◇ to prove that the base-line grammar covers a set of dialect grammars.

If such transformations make the grammars *converge*, one can interpret the properties of these transformations as a form of representation of relationships between the grammars. For instance, if all utilised transformations preserve the semantics, the grammars define identical languages. This method proposed in this thesis is presented in detail in Chapters 4 and 5.

Language *evolution* is a process of changing the language over time by reflecting language users' needs, advances in language engineering and design paradigm shifts. In the current practice language evolution happens as a combination of parallel activities, and the language *documentation* is updated occasionally by using error-prone methods. We propose to replace a range of these methods by *language engineering* technology: manual updates of grammar productions can be superseded by grammar extraction and insertion; inline code samples can be pretty-printed when needed from an operational test suite; hand-made change documents are not needed when disciplined language document transformations take their place. Introducing new language features and changing the syntax of existing language constructs can be mapped to transformation operators, as will be shown in chapter 7 for grammar transformation and in chapter 6 for language documentation transformation. The original language documents need to be adjusted accordingly by adding new sections, updating textual paragraphs, regenerating the samples, renewing internal and external links.

1.2 Motivation and objectives

For each of the language engineering areas mentioned in the previous section there exist multiple well-known approaches to problem solving. This section proposes the vision that we adopt in this thesis in order to explain our later methodological choices. After that a real specification excerpt will be presented to exemplify grammar extraction, transformation, documentation and life-cycle for the reader.

We strive towards *automated* grammar extraction. This means that the mapping between a real grammar artefact in the form of a parser specification, a data format definition or a syntax diagram, and the executable and transformable grammar in the form accepted by the chosen infrastructure, should be completely automated. We think in terms of general *extractors*, tools that implement the recovery activities, perform necessary abstraction activities and map the core grammar knowledge. These extractors should be sufficiently customisable in order to be adapted, possibly with an interference of a human expert, to the concrete source of grammar knowledge, but they should be universal per type of grammar knowledge, not per source. The list of such extractors is included in the deliverables list in section 8.3. The detailed description of one of the non-trivial extractors is given in section 5.4.

There are different approaches to grammar transformation. We opt for semi-automated, *programmable* grammar transformation. This means that the infrastructure provides the language engineer with an operator suite, a set of parametric transformation operators that help to refactor or edit the base-line grammar by introducing changes consistently and retaining the relationship with the original. Then, it is a manual job of a language engineer to pick an operator from the suite, to assign values to its parameters so that it becomes applicable and accomplishes the desired goal, and to save this information for the automated transformation engine to execute. Transforming grammars this way is very much alike conventional programming. Our transformation suite for grammar engineering is presented in chapter 7, another suite for language documentation transformation is give in section 6.6.

Thus, language evolution and language document evolution can be technically related and the tool support for one can simplify the other. If updating language documentation is also semi-automated and relies on a set of well-defined transformation operators with predictable behaviour, it will surface and strengthen the relationship between the grammar(s) and the specification(s). We utilise our own infrastructure for our needs in order not only to help ourselves, but also to strengthen the proposed approach with case studies. In particular chapter 7, subsection 4.10.1, section 6.4 and section 6.6 have been fully generated with our prototype implementation.

1.3 Example scenario

Having stated the design intent in the first two sections, we now take a concrete example and observe the meaning and the impact of the above mentioned technologies. Consider Listing 1.1, which presents a piece of a language document, namely *Java Language Specification* (a deliberately incomplete section). Our points of interest here are:

- ◇ The section describes one domain concept (one programming language construct).
- ◇ It contains a marked up grammar production that defines its syntax.
- ◇ It contains a written explanation of the language construct in unstructured English.
- ◇ The text contains references to other related sections.
- ◇ The section contains an executable example.
- ◇ It contains written explanation of the example in unstructured English.

8.1 Class Declaration

A *class declaration* specifies a new reference type:

ClassDeclaration:
ClassModifiers$_{opt}$ `class` *Identifier* *Super*$_{opt}$ *Interfaces*$_{opt}$ *ClassBody*

If a class is declared in a named package (§7.4.1) with fully qualified name *P* (§6.7), then the class has the fully qualified name *P*`.`*Identifier*. If the class is in an unnamed package (§7.4.2), then the class has the fully qualified name *Identifier*. In the example:

```
class Point { int x, y; }
```

the class `Point` is declared in a compilation unit with no `package` statement, and thus `Point` is its fully qualified name, whereas in the example:

```
package vista;
class Point { int x, y; }
```

...

Listing 1.1: Class declaration section from JLS 1 [77, p.128].

Suppose this document is stored in a format that we have the functionality to access and to parse—say, in HTML. Then, we can start by retrieving a grammar production for ClassDeclaration. This is an example of *grammar extraction*. Once we have extracted all grammar productions from the whole document and collected them in the form that we can further operate on, *grammar recovery* is completed. The original production can be then pretty-printed in our BNF-like notation (see section 7.4) as follows:

```
ClassDeclaration:
     ClassModifiers? "class" Identifier Super? Interfaces? ClassBody
```

Despite the notational differences of the presentation, this is the same piece of grammar knowledge that was in the original language document. The mapping between the original grammar artefact type (i.e., hypertext multi-font language document) and our target BNF-like format is established at this point. Having this mapping available means that we can repeat the same (now automated) process to obtain the productions—i.e., we have the *extractor*. The extracted grammar knowledge will now be used in *grammar transformations* to be beautified, deyaccified, disambiguated, optimised and otherwise adapted to the needs of the language end user.

Meanwhile let us return to the original document. Suppose we have an automated extractor that is able to recover not only the grammar productions scattered in the text, but also all other heterogeneous artefacts: blocks of text, keywords, samples, cross-references, etc. The result of its application to Listing 1.1 would be Listing 1.2. The extraction process is a fully automated action. By migrating the document from the original format to the common format for language documentation, we abstract from presentation idiosyn-

```
<subtopic id="S.8.1">
  <title>Class Declaration</title>
  <description>
    <content>
      <text>
        A <ldf:keyword>class declaration</ldf:keyword> specifies
        a new reference type:
      </text>
      <bgf:production>
        . . .
      </bgf:production>
      <text>
        If a class is declared in a named package
        (<ldf:link>
          <text>§7.4.1</text>
          <reference>S.7.4.1</reference>
        </ldf:link>)
        with fully qualified name <ldf:code>P</ldf:code>
        (<ldf:link>
          <text>§6.7</text>
          <reference>S.6.7</reference>
        </ldf:link>),
        then the class has the fully qualified name
        <ldf:code>P.</ldf:code><ldf:keyword>Identifier</ldf:keyword>.
        If the class is in an unnamed package
        (<ldf:link>
          <text>§7.4.2</text>
          <reference>S.7.4.2</reference>
        </ldf:link>),
        then the class has the fully qualified name
        <ldf:keyword>Identifier</ldf:keyword>.
      </text>
      <text>In the example:</text>
      <sample>class Point { int x, y; }</sample>
      <text>
        the class <ldf:code>Point</ldf:code> is declared in a compilation unit
        with no <ldf:code>package</ldf:code> statement, and thus
        <ldf:code>Point</ldf:code> is its fully qualified name, whereas
        in the example:
      </text>
      <sample>
        package vista;
        class Point { int x, y; }
      </sample>
      . . .
    </content>
  </description>
</subtopic>
```

Listing 1.2: LDF as extracted. Three blocks are selected on the figure—the simple extractor was not able to recognise them. One can see that the third block depicts an example, and it contains samples as such, as well as the accompanying text.

```
<subtopic id="S.8.1">
  <title>Class Declaration</title>
  <synopsis>
    <content>
      <text>
        A <ldf:keyword>class declaration</ldf:keyword> specifies
        a new reference type:
      </text>
    </content>
  </synopsis>
  <syntax>
    <content>
      <bgf:production>
        . . .
      </bgf:production>
      <text>
        If a class is declared in a named package
        (<ldf:link>
          <text>§7.4.1</text>
          <reference>S.7.4.1</reference>
        </ldf:link>)
        with fully qualified name <ldf:code>P</ldf:code>
        (<ldf:link>
          <text>§6.7</text>
          <reference>S.6.7</reference>
        </ldf:link>),
        then the class has the fully qualified name
        <ldf:code>P.</ldf:code><ldf:keyword>Identifier</ldf:keyword>.
        If the class is in an unnamed package
        (<ldf:link>
          <text>§7.4.2</text>
          <reference>S.7.4.2</reference>
        </ldf:link>),
        then the class has the fully qualified name
        <ldf:keyword>Identifier</ldf:keyword>.
      </text>
    </content>
  </syntax>
  <example>
    <content>
      <text>In the example:</text>
      <sample>class Point { int x, y; }</sample>
      <text>
        the class <ldf:code>Point</ldf:code> is declared in a compilation unit
        with no <ldf:code>package</ldf:code> statement, and thus
        <ldf:code>Point</ldf:code> is its fully qualified name, whereas
        in the example:
      </text>
      <sample>
        package vista;
        class Point { int x, y; }
      </sample>
      . . .
    </content>
  </example>
  . . .
</subtopic>
```

Listing 1.3: LDF as intended. The blocks identified on the previous figure now belong to different subsections of §8.1, namely to Synopsis, Syntax and Example.

8.1 Class Declaration

A *class declaration* specifies a new named reference type:

ClassDeclaration:
ClassModifiers$_{opt}$ `class` *Identifier* *Super*$_{opt}$ *Interfaces*$_{opt}$ *ClassBody*

The *Identifier* in a class declaration specifies the name of the class. A compile-time error occurs if a class has the same simple name as any of its enclosing classes or interfaces.

. . .

Listing 1.4: Class declaration section from JLS 2 [78, p.136].

crasies. Once the document is inside our infrastructure, we do not depend on irrelevant storage format peculiarities and can focus completely on crucial points of interest.

If we take a closer look at Listing 1.2, we will notice that the synopsis, the syntax definition and the example—all three parts of the original section—ended up as consequent content blocks of one "description" section. If we had a smarter context-aware extractor or asked a human domain expert to look into the document's internal structure, the language documentation ontology would have been utilised more consistently, representing examples, syntax definitions and other parts as on Listing 1.3. This semantic shift could be achieved by means of *language document transformations*, an extension of grammar transformations. The old functionality of producing a seemingly homogeneous language document is retained. However, by keeping the internal representation well-structured, the result of transformation will also be easily useful to perform sophisticated actions like extracting test cases or listing all subsections of the same type.

We now compare Listing 1.1 with 1.4. The documentation excerpts displayed on them express the same language construct in different versions of the Java Language Specification. However, they are not syntactically identical. The grammar production in both versions is the same, but the differences are:

- ◇ In 1.4 there is no longer a code example.
- ◇ The text in 1.1 and 1.4 is different.
- ◇ The role of the text has also changed: cf. the semantic details on naming in 1.1 and the compile-time errors in 1.4.

It is quite normal for language documentation to evolve, and the technology support for that process can also be provided by the same language document transformations.

In the previous example we have seen the language document changed. Also the language itself can evolve. For instance, consider Listing 1.5. It is different from both 1.1 and 1.4 in the sense that their grammar fragments are not equal:

8.1 Class Declaration

A *class declaration* specifies a new named reference type. There are two kinds of class declarations - *normal class declarations* and *enum declarations*:

ClassDeclaration:
NormalClassDeclaration
EnumDeclaration

NormalClassDeclaration:
ClassModifiers$_{opt}$ `class` *Identifier* *TypeParameters*$_{opt}$ *Super*$_{opt}$ *Interfaces*$_{opt}$ *ClassBody*

The rules in this section apply to all class declarations unless this specification explicitly states otherwise. In many cases, special restrictions apply to enum declarations. Enum declarations are described in detail in §8.9.

The *Identifier* in a class declaration specifies the name of the class. A compile-time error occurs if a class has the same simple name as any of its enclosing classes or interfaces.

. . .

Listing 1.5: Class declaration section from JLS 3 [79, p.175].

Listing 1.1 and 1.4

```
ClassDeclaration:
     ClassModifiers? "class" Identifier Super? Interfaces?
          ClassBody
```

Listing 1.5

```
ClassDeclaration:
     NormalClassDeclaration
     EnumDeclaration
NormalClassDeclaration:
     ClassModifiers? "class" Identifier TypeParameters? Super?
          Interfaces? ClassBody
```

Hence, grammar transformations are needed to align Listing 1.5 with 1.1 and 1.4.

Beside evolutionary aspects there is another scenario to consider. Suppose the language manual exists in different forms: for instance, one compiler vendor version, one international standard and one parser definition. There should be a relationship between these documents and the grammars in them, which is usually hidden or undefined. To make this relationship explicit, we use the process of *language convergence*. Language document transformations are used to establish the relationship between the documents.

An uninitiated reader would assume that the relationship between different grammars of presumably the same language is guaranteed, i.e. grammar convergence would show their equivalence. At this moment this is not the case, because the grammars were not engineered with automated tools and can contain inconsistencies that grammar convergence

aids to locate. For instance, subsection 3.2.7 presents this mismatch between different parts of the C# specification:

C# standard, §22.1, page 297, lines 15–16

```
delegate-declaration:
       attributes?  delegate-modifiers?  "delegate"
              return-type identifier
              "(" formal-parameter-list?  ")" ";"
```

C# standard, Appendix A.2.11, page 357, lines 34–35

```
delegate-declaration:
       attributes?  delegate-modifiers?  "delegate"
              type identifier
              "(" formal-parameter-list?  ")" ";"
```

The difference in these two productions is structural: one nonterminal symbol is used instead of the other. Since we know from §17.5 on page 282 that `return-type` is defined as `type` or `"void"`, we also know that the former production is more permissive than the latter.

Another example for the Java specification is presented in subsection 5.6.3:

JLS2, §14.14, page 297

```
BreakStatement:
     "break" Identifier? ";"
```

JLS2, §18.1, page 453

```
Statement:
     "break" Identifier?
```

We see two differences here: one nominal which is intentional and is caused by the way each grammar was engineered, whereas the other one is structural—the absence of a semicolon in the second excerpt is a bug.

We found these mismatches with our engineering approach proposed in this thesis. If this technology is adopted for usage on early stages of language documentation development, it can prevent the appearance of many errors. A strong and well-designed infrastructure is needed, and our work contains such a prototype.

The contributions that we propose in this thesis are dedicated to the recovery, evolution and convergence of language documents. The most prominent structures that repeatedly appear in the analysed language specifications and manuals, are captured in the prototype ontology for language documentation. We have also developed a range of tools to support the recovery, evolution and convergence of languages. The tools themselves, as well as the working grammars that were derived with their help, were made freely available on the internet.

1.4 Thesis outline and contributions

The entire organisation of the thesis will be sketched in a single paragraph, and then we will proceed with a chapter by chapter overview of its contents, its main contributions and details on publication and collaboration.

The rest of the thesis is organised as follows. There are six core chapters: one background chapter; one about a grammar recovery case study; one on convergence; one on a bigger case study in recovery and convergence; one on documentation and one on transformation. Each of them presents the research goals, methods and contributions on its subtopic, relying on the results of the previous chapters. For instance, in chapter 3 there is a case study with one grammar that is manually extracted by copy and paste from a language definition and transformed with very simple instruments that basically redefine parts of it. In chapter 4 we already employ automated extraction and start developing notions of semantic preservation to introduce and utilise more accurate transformation operators. In chapter 5 another case study is presented and evaluated, containing six grammars, each of the size of the first case study, and thousands of carefully planned and laid out transformation steps. In chapter 6 we are in a position to report on how language documents are currently organised and to propose how to improve on that using our proposed methods that we support with an implementation. By arrival at chapter 7 we are ready to draw conclusions about categories of grammar transformation operators and claim to have a comprehensive transformation suite capable of handling grammars of industrial size. Besides that, we already have a language documentation infrastructure and use it ourselves for making this chapter.

1.4.1 Chapter 2 overview: additional background

Keywords: *grammarware, grammar engineering, grammar recovery, grammar convergence, language documentation, grammar transformation, language evolution.*

Summary: After the first chapter having introduced the basic terms briefly, additional background material is discussed in Chapter 2. It starts by introducing prominent background concepts about grammar engineering and lists the methods, notions and considerations important for all three main topics of the thesis: recovery, convergence and documentation of languages.

1.4.2 Chapter 3 overview: case study on language recovery

Keywords: *C#, grammar engineering, semi-automatic grammar recovery, grammar adaptation.*

Summary: Chapter 3 is an application of grammar engineering methodology to a real life case study. A comparatively new language standard at the time of research, ECMA-334 or ISO/IEC 23270:2003, which defines the C# programming language, was taken in order to extract a grammar from it and use that grammar for parsing C# code. It turned out that the grammar recovery of C# was far from trivial. It took us 217 transformations to achieve a correct and complete C# grammar. Intuitively one would expect such an effort for older languages like COBOL due to their decades-long evolution, but this seems to be a problem of all times. The presented process is not limited to C#. We provide a general step-wise plan to recover a correct working formal language grammar from an existing language specification in a semi-automated way. At the end of the chapter the research objectives are revisited to conform to the lessons learnt from the C# experiment.

Contribution: Using existing technology, a working grammar of C# was recovered directly from the corresponding language specification and tested on small scale (100KLOC). After our initial idea of the research objectives we carried out a first case study with promising results. This case study further refined and shaped the research objectives, as displayed by subsection 3.4.2.

Publication: The technical results of this chapter were briefly described, published and presented at the participants' workshop of the International Summer School on Generative and Transformational Techniques in Software Engineering as *Correct C# Grammar too Sharp for ISO* [257]. In the next version of the standard some of the errors were removed [115]. We have no information about Microsoft actually using our results in their work, but since not all errors were fixed, we presume they did not use an engineering approach for this. Our C# grammar was later made publicly available via `http://www.cs.vu.nl/grammarware/browsable/CSharp` [241] and forms now a part of the *SLPS Grammar Zoo* at `http://slps.sf.net/zoo` [260]. General discussion on EBNF as a grammar definition formalism, as well as on grammar engineering techniques like de-layering and relaxation overlaps somewhat with the ISO note titled *Language Standardization Needs Grammarware* [143].

Acknowledgement: [143] was written in collaboration with Dr. Steven Klusener (Vrije Universiteit Amsterdam), who also presented the work to the international programming languages subcommittee ISO/IEC JTC1/SC22.

1.4.3 Chapter 4 overview: language convergence

Keywords: *grammar convergence, grammar re-engineering, grammar extraction, programmable grammar transformations, grammar synchronisation, domain specific languages.*

Summary: Chapter 4 introduces the method of grammar convergence. Our proposal is to take two or more grammars that are expected to be related, use grammar transformation to make them identical, then analyse these transformation steps and draw conclusions about the real relationship between the grammars. The chapter illustrates our viewpoint with a small case study, namely FL, the Factorial Language. Grammar convergence is a lightweight verification method for establishing and maintaining the correspondence between grammar knowledge ingrained in different kinds of software artefacts such as object models, XML schemata, parser descriptions or language documents. The central idea of the approach is to extract grammar knowledge, and to use programmable grammar transformations as means to constructively prove the convergence of different sources to a shared limit.

Contribution: Grammar convergence using programmable grammar transformations is a grammar re-engineering approach that comprises not only recovery of individual grammars, but also comparing them, applying benchmarks and choosing the right places for changes. Overall this is a methodology chapter that introduces an approach for solving a particular class of problems, namely those where several grammars of a language can be obtained but cannot be explicitly related based on any other hard evidence—which grammar convergence process aims to provide.

Publication: Parts of this chapter were published in condensed form as *An Introduction to Grammar Convergence* [166] and *Recovering Grammar Relationships for the Java Language Specification* [167, 168]. The former is an introductory paper presented at the 7th International Conference on Integrated Formal Methods. The version included in the thesis is substantially enhanced with respect to the original paper, and also includes full diagrams and listings presented in a unified notation. The notion of phases of convergence (section 4.7) has emerged later during the work on [167] for the 9th IEEE International Working Conference on Source Code Analysis and Manipulation where it received the Best Paper Award. A significantly extended version is being printed in the special issue of Software Quality Journal [168]. An implementation of grammar convergence is publicly available through the *Software Language Processing Suite* [263].

Acknowledgement: The idea of using programmable grammar transformations to converge grammars has emerged in collaboration with Prof. Dr. Ralf Lämmel (Software Languages Team, Universität Koblenz), as was the subsequent development of the prototype. All three papers mentioned above [166, 167, 168] are co-authored with him.

1.4.4 Chapter 5 overview: case study on recovery and convergence

Keywords: *Java, language documentation, grammar extraction, programmable grammar transformations, grammar convergence*.

Summary: Chapter 5 describes a completed effort to recover the relationships between all the grammars that occur in the different versions of the Java Language Specification (JLS). According to the method from the previous chapter, the relationships are represented as grammar transformations that capture all accidental or intended differences between the JLS grammars. The process is mechanised and it is driven by simple measures of nominal or structural differences between any pair of grammars involved.

Contribution: A large case study such as this one (1627 transformation steps) serves as stress test for any methodology. While working on it, we were able to justify the grammar convergence method as well as to stress it enough to redesign certain elements. In particular the overall workflow of grammar convergence got shaped in this chapter, benchmarks were introduced to allow a language engineer to measure the progress. The engineering contribution of this chapter comprises three grammars that correspond to JDK 1.0 "Oak", J2SE 1.2 "Playground" and J2SE 5.0 "Tiger", as well as transformation scripts documenting their relationship to one another. A number of inconsistencies were observed, some of them known to exist, but some yet unreported.

Publication: A condensed version of this case study description was published as *Recovering Grammar Relationships for the Java Language Specification* [167] in the proceedings of the 9th IEEE International Working Conference on Source Code Analysis and Manipulation. The paper received the Best Paper Award. A significantly extended version is being printed in the special issue of Software Quality Journal [168]. This chapter is a further extension of both papers. We included important information on the extractor (section 5.4) and displayed detailed tables such as Table 5.4 and Table 5.6. The tables contain important information about the extraction process and the recovered relationship; it was mentioned in the paper but not included there due to space constraints. An application of our grammar convergence method to various versions of the Java Language Specification is publicly available through the *Software Language Processing Suite* [263], including all Java grammars being displayed as a part of *SLPS Grammar Zoo* [260].

Acknowledgement: As stated above, Prof. Dr. Ralf Lämmel (Software Languages Team, Universität Koblenz) was a co-author of [167, 168].

1.4.5 Chapter 6 overview: language documentation

Keywords: *language documentation, grammarware, language evolution, language adaptation, language manuals, language specifications, data model, programmable transformations*.

Summary: Chapter 6 dives into the issues of consistency management of language specifications that were mentioned and discussed in previous chapters. The current state of affairs is presented and examined based on freely available standards, as well as internal knowledge of the ISO working groups. The chapter explains the meaning of standardisation for language engineering and defines a unified data model for language specifications and a suite of transformations that operate on it. Previously presented advancements in grammar notation and programmable grammar transformations are utilised here as well.

Contribution: This chapter reports on an elaborate domain analysis in the field of language documentation. It contains several tables that collect the information about how the language specifications are composed, what they consist of and what differences various syntax definition formalisms demonstrate. The core of the reverse engineering contribution is the summary of language definition notations and the summary of language documentation elements collected by examination of a range of existing language documents. The technical contribution of the chapter is generalising grammar transformation and grammar convergence concepts and processes to cover the conceptual prerequisites. Full technical support for automating activities related to language documentation is also given. The document schema is of particular interest, the chapter shows how it can be used to map any of the existing language specifications. It is also explained how the technology was prototyped and applied to the following chapter.

Publication: The concepts of handling language documentation with technical discipline and providing proper tool support were formulated on an early stage of research as *Language Standardization Needs Grammarware* [143]. The paper titled *Language Convergence Infrastructure* [259] being published the post-proceedings of 3rd International Summer School on Generative and Transformational Techniques in Software Engineering included a more up-to-date view, with an extended abstract already published in the pre-proceedings as [258] and reported at the aforementioned summer school. *Language Documentation: Survey and Synthesis of a Unified Format* [262] is accepted for publication at the 3rd International Conference on Software Language Engineering. The complete description of LDF and XLDF is publicly available through the *Software Language Processing Suite* [263].

Acknowledgement: The conceptual prerequisites of this chapter were formulated in collaboration with Dr. Steven Klusener (Vrije Universiteit Amsterdam) and delivered with his help to the ISO SC22 committee in 2005. [262] was co-authored with Prof. Dr. Ralf Lämmel.

1.4.6 Chapter 7 overview: XBGF language manual

Keywords: *programmable grammar transformations, transformation suite, language manual, grammar refactoring, language semantics, term-oriented semantics, string-oriented semantics, semantic preservation.*

Summary: The purpose of Chapter 7 is twofold. On the one hand, it elaborates on the XBGF transformation suite that was fleshed out during the research in chapters 4 and 5. All the operators that have been mentioned or even used so far, are presented here systematically, classified and exemplified by test cases. On the other hand, Chapter 7 is also a demonstration of our tools. It is a case study where the chapter was fully generated from its input using our implementation.

Contribution: The conceptual contribution of this chapter is a complete and thorough definition and explanation of the XBGF transformation suite utilised in the previous two chapters. This chapter is also a correct language document by itself, and as such demonstrates the way of dealing with language documents and serves as a case study for the methods that were explained in Chapter 6.

Publication: The contents of this chapter are identical with the XBGF online manual [261] that can be found online at `http://slps.sf.net/xbgf`. Both the HTML manual and the LaTeX source of the chapter were generated from the data extracted from the XML Schema for XBGF with the transformational scripts—the technology explained in the Chapter 6. Beside language transformations, two XSLT pretty-printers were used to produce XHTML and TEX.

Acknowledgement: The XBGF transformation suite was designed in collaboration with Prof. Dr. Ralf Lämmel (Software Languages Team, Universität Koblenz) who also was the lead developer in its Prolog implementation.

Figure 1.1: Tag cloud of the text of this thesis.

Chapter 2

Additional background

The previous chapter, especially section 1.2, has already introduced the most important background concepts. However, we need to provide additional, more detailed information on the research context. This chapter will present the notions used throughout the thesis, explain the existing methods of grammar engineering and define our view on them. Both purpose and importance of grammar transformation, recovery, convergence and documentation should become clear, enabling the remaining chapters to focus directly on the contributions.

2.1 Terminology

There are notions that will be used extensively in this chapter yet could have remained unclear from the previous sections:

A **grammar** is a strict and precise definition of a language in its formal sense (as a set of allowed words). Hence, the grammar defines the structure of a piece of source code. Grammars for mainstream languages used in industry are big, they are not supposed to be read by humans and be manually checked for completeness, correctness and other properties. Instead, an automated approach is taken with an infrastructure accepting a formal grammar as an input and producing a parser, a transformational tool or other grammarware as an output.

The words "schema", "ontology" or "metamodel" are used instead of the word "grammar" in different areas. Schemata and data models are notions related to grammars in database and data manipulation research, although not all data models can be easily mapped to grammars. XML also calls its grammars schemata, whether they conform to XML Schema [75, 208], RELAX NG [34], DTD [20] or any other standard. Ontologies are used in complex matters such as semantic web, business process modelling or artificial intelligence [221]. They mostly fall outside the scope of this thesis because of their complicate nature. Grammar domain is smaller, simpler and does not face the kind of challenges that are typical for ontology alignments.

Grammar definition formalism is any kind of notation for modelling the syntax of a language. It can be textual with only a few basic features for denoting terminals,

nonterminals and productions, or it can be a complicated graphic notation with arrows and boxes.

BNF [9] is a very well-known format for describing grammars, standing for Backus-Naur Form or **Backus Normal Form**. John Backus introduced the format back in 1960s as a part of his work on Algol 60 [10]. Peter Naur contributed to it later, significantly improving readability [144]. Niklaus Wirth made another improvement in 1977 with Extended BNF (EBNF) that included more advanced and more abstract iteration constructs together with other features described in detail in section 6.3 and section 2.7.

The structure of the source code is typically represented by a data structure called a **parse tree**. A **parser** is an application that takes a piece of source code as an input and constructs the corresponding parse tree. If the source code does not conform to the grammar, a parser reports a **parse error**. Parsers can be generated from grammars automatically by so called **parser generators** [2, 83, 128].

2.2 Grammarware

The term **grammarware** is introduced in [141], widely used nowadays [64, 153, 251] and will be encountered numerous times on the pages of this thesis. In natural linguistics, the term "lingware engineering" is used [191]. According to the definition, grammarware comprises grammars and all related grammar-dependent software. Examples of grammarware include, but are not limited to:

- A *parser* (a hierarchical analyser or a syntactic analyser) that combines tokens of a source program into grammar phrases in a form of a parse tree [2, 83].
- A *compiler* that translates a program in the source language to its equivalent in the target language [2, 23].
- A *pretty-printer* (a formatter, a code beautifier) that transforms a program in such a way that the whitespace is organised for better human comprehension and making its structure explicit [24, 29].
- A *scanner* (a lexical analyser) that composes a stream of tokens out of a program text [2].
- A *browser* that shows web pages from internet with adjusting the layout for the particular user needs and loading additional files, menus, movie clips and other plug-ins [256].
- A *static checker* (e.g., a code optimiser or a type checker) that performs certain verification techniques without actually executing the program [71].
- A *development environment* together with built-in tools such as structural editors, debuggers or profilers that aid programmers in their job by suggesting the right libraries or modules, pointing out errors or possible bug sources and otherwise organising all daily activities [88].

- A *preprocessor* and a *postprocessor* that manipulate the code before and after the compilation respectively [63, 223].
- An *interpreter* or any other run-time environment [215].
- A *code refactoring* tool that aids the developer or programmer to restructure the source code in order to improve its readability, maintainability, integrability or extendability without changing its behaviour [70, 138, 156, 200].
- A *program slicing* tool that identifies the source code that can affect or can be affected by a value of a given variable [247].
- A *reverse engineering* tool that helps to extract business logic from a system which source code is unavailable or untrusted [84]
- *XMLware*, or software that is based on XML: an XML parser, an XML validator, data binding tools, Extensible Stylesheet Language Transformations (XSLT), XPath and XQuery processors — everything that uses XML Schema schemata or Document Type Definitions (they describe structure of XML data and are therefore in fact grammars) [64].
- Numerous (re)documentation tools and software analysis tools [65, 214].
- Even API (Application Programming Interfaces) can be seen as languages that have grammars. Domain specific languages (DSL) and Modelware (MDE-related technology) can also be treated as grammarware [153, 251].

2.3 Techniques for grammars

A number of techniques and approaches will be mentioned and utilised on the pages of this thesis:

- **Grammar adaptation** [154] refers to grammar transformations employed for the sake of adapting it for grammar development, grammar maintenance, grammar reengineering, grammar recovery, etc.
- **Grammar extension** [167] is a semantic increasing grammar transformation that makes explicit additions to existing functionality by introducing previously nonexistent operators, clauses, statement types, etc.
- **Grammar relaxation** is a semantic increasing variant of grammar adaptation where parts of the grammar are removed or generalised in order to simplify the structure. As a result, functionality is partially lost, but usually the language engineer is aware of that. Tolerant parsing [142] and island grammars [186, 187, 227, 236] are examples of technological support for grammar relaxation, but it can also take form of simpler adjustments.

- ◇ **Grammar hacking** [155, 219] is a process when a grammar is changed for the purpose of making it fit into certain parsing technology limitations. The name implies some degree of error-proneness, uncontrollability and the lack of testing, as opposed to disciplined grammar engineering.

- ◇ **Yaccification** [132, 154] is a variation of it that involves YACC parser generator [128] or tendencies and traditions triggered earlier by the use of YACC [139]. In a broader sense, by yaccification we will mean conscious introduction of idiosyncratic constructs for the sake of fitting the grammar within limitations of a certain parsing technology, be it LR(1), LALR or recursive descent.

- ◇ **De-yaccification** [27, 217] is a complementary process that removes the binding to any particular parsing technology to form a generic base-line grammar from a yaccified or otherwise "optimised" one.

- ◇ **Grammar beautification** is a name for grammar transformation steps taken for the sake of improving its performance and readability. There is sometimes no clear practical goal set for beautification, but rather a general idea of a better way to implement the same functionality.

- ◇ **Grammar correction** is a grammar transformation step that removes an error. The nature of the error can be such that the grammar was still operational, but the semantics of the resulting parse tree was wrong and insufficient testing did not show it.

2.4 Language evolution: versions and dialects

The making of grammars and parsers is an activity that is neither easy nor common. Parsers are usually constructed not only on basis of a standard, but also on different considerations of a specific vendor. Besides that, parsers are developed as a back-end for bigger grammarware (most commonly a compiler) and sometimes they become non-detachable and, therefore, non-reusable. If someone wants a solid grammar, it should be re-engineered from such compiler's sources.

Often grammars are constructed in a way that they belong to a specific class of grammars. While Chomsky's grammar classes [32] are compositional in a sense that two context-free grammars combined make a context-free grammar, the grammars that conform to a specific parsing technology are not always compositional. That means that two LR grammars, when combined, make a context-free grammar that may or may not be LR. Inability to fit into a specific class can make the grammar useless for the developer's goal, so that usually means grammars are not modular: they are built as a whole and not constructed from small components.

The practice shows us that only very few languages can stay in the same form as they were first invented. Most languages evolve or die. We can name several reasons for that:

- ◇ Languages evolve because new features need to be introduced. Software development methods have not been the same during the last years, and language evolution

must keep up with them. For example, object-oriented analysis and design methods [18] led to introducing object-oriented programming constructs into many languages (actually, to all high level mainstream languages). The last years emphasis on modularisation and re-use lead to introducing language constructs supporting design patterns [73] and component-oriented development [228].

◇ Languages also evolve when contemporary development methods prove themselves harmful and need to be extracted from the language to avoid their further abuse. The most famous example in this category is elimination of an unconditional transfer of control statement, **GO TO** [48, 133, 146, 188, 216]. Almost every language has "all sorts of programming language features that seem better ignored than exploited" [47]. Recent work on *bug patterns* [4] and *anti-patterns* [229] collects other examples of the commonly used design and development moves that were deemed to be inappropriate or error-prone.

◇ Languages just evolve because their authors want to make it easier for the programmers to use them, since language usability issues have never been really researched anyway. One of the frequently encountered improvements is introducing so called *syntactic sugar* constructs that do not formally make the language more powerful in a sense of declarativity, but rather let the programmers do their job more efficiently. The `yield` statement introduced in C# 2.0 is a good example of such a language change [115], as is the coalesce operator considered in section 2.9.

◇ The languages co-evolve with their compilers and IDEs. No modern language can exist in vacuum, or at least it will have no chance of being used in industry. They all have supporting libraries that play important role in making a design choice in favour of this or that programming language. Introducing a new API library is not a change to the language in a strict sense, but by our definition it is—simply because it changes the way the programs are written. Besides that, the APIs give rise to grammars themselves. Sometimes the platform and the application programming interface it provides are more important than the underlying language as such—a source code of a web-service written in C# will have much more common with the same web-service created in any other .NET language than with a web-service made in Java with CORBA.

Language specifications that represent languages, are of dynamic nature as well. In an ideal situation the grammarware and the specification it conforms to are co-evolving, but in practice one often makes the first step with the other one hopefully following. Early classical language standards were developed by a special committee of experts with grammarware developers starting their work only after receiving a completed standard document, while compiler specific language reference manuals are apparently updated only after the changes take place in the compiler itself. If properly and flawlessly organised, this process can be considered co-evolution anyway.

2.5 Grammar levels

In [164] the notion of *grammar levels* is introduced, which we utilise here and adopt to the broader context as follows:

- ◇ A **Level 1 Grammar** is a raw grammar that is a starting point of whole grammar recovery process: it has just been extracted from a language definition, corrected of all typographical, text recognition and similar errors and converted into a BNF dialect.
- ◇ A **Level 2 Grammar** is maximally connected Level 1 Grammar. That is, it does not contain unwanted top sorts (nonterminals that are defined but never used) and bottom sorts (nonterminals that are used but not defined). These two quality indicators were proposed in [217, 219] and discussed in more detail in [154].
- ◇ A **Level 3 Grammar** is a Level 2 Grammar complemented with a lexical part: on the second level only those top sorts remained that are either true root sorts or lexical. Some language standards such as [77] (as well as [114, 115, 225] that are heavily based on it) have a special part dedicated to a lexical grammar, while others such as [90] do not (that means it must be created manually).
- ◇ A **Level 4 Grammar** is a Level 3 Grammar that has been tested on a scale of considerable volume of code coming from different sources, companies, countries and coding traditions—in languages such as C or COBOL a codebase used for testing can contain millions of lines of code.
- ◇ A **Level 5 Grammar** is a grammar that was fully recovered from a source code of a compiler in an automated way. If it has been done correctly, it can be claimed that the Level 5 Grammars are the ones most close to the ideal since they accept everything that can be accepted by a compiler and nothing more.

2.6 Grammar recovery methodology

Grammar recovery process can be broken down roughly into two distinct phases: grammar extraction and grammar transformation. Grammar extraction process involves taking a grammar artefact: language specification, parser source code, XML schema, etc, and producing a Level 1 Grammar that contains the same grammar knowledge as the one engraved there. Grammar transformation process operates on the extracted grammar and produces a Level 2 or 3 Grammar.

Practice shows that automated software maintenance, analysis, migration and renovation can be much more profitable in terms of costs and human effort than the alternatives (manual changes, legacy rebuild, etc), especially on large scale [234]. However, automated methods do require special foundation for their successful usage. The basis is formed by two things here: the grammar and the transformational rules.

A working grammar is needed to parse the code, to get it from a textual form that the programmers created into a specialised generational and transformational infrastructure

that usually utilises a tree-like internal format. In spite of the fact that the formal grammar theory is quite an established area since 1956 [32], the grammars themselves are rarely freely obtainable, they are complex artefacts that are seen as valuable assets, require considerable effort and expertise to compose and therefore are not always readily disclosed to public by those who develop or reverse engineer them. A syntax grammar is basically nothing more than a formal description of what can and what cannot be considered valid in a language. Thus, the most obvious source for this kind of information is a language documentation.

However, documentation is neither ever complete nor error-free. Special techniques are needed to obtain correct grammars, such as grammar adaptation [154], grammar recovery [164], grammar engineering, grammar reverse engineering, grammar re-engineering [141], grammar archæology [158], grammar extraction, assessment, correction, completion, testing, and so on.

Grammar recovery—the methodology for extracting a complete grammar from an existing programming language's manual, a specification or a compiler's source code, assessing the grammar, correcting it, testing, making it complete, and so on—is not a new approach. This technique has been used in 1998 for the first time with Programming Language for EXchanges (PLEX), a proprietary DSL for real time embedded software systems by Ericsson [219]. A successful application of the same technology to COBOL followed [28].

It is a widespread misunderstanding that grammar recovery is needed only if the language has decades-long evolution and legacy as in the aforementioned examples of COBOL and PLEX. It is easy to assume that newer standards such as the C# specification [115, 225] are of much better quality; that they are easily accessible, and that their formal contents can be used in their existing condition (i.e., without any grammar adaptation steps whatsoever). Unfortunately, this is not the case. The research presented in the chapter 3 shows that C#, a fairly modern programming language, needed recovery as well. In that project the grammar was adapted by engineering more than 200 transformation rules.

2.7 Grammar definition formalism

There are many metasyntaxes in use. BNF (Backus-Naur Form or Backus Normal Form) [9] and EBNF (Extended BNF) [252] are used practically everywhere, but other formalisms exist as well. One good example of a completely different grammar definition formalisms would be PEG, Parsing Expression Grammar [69]. Visual formalisms exist as well, in the form of "railroad tracks" [182, 183] and other kinds of syntax diagrams [76, 113, 176]. Graphical form is believed to be easier to understand, but it poses technical problems for translation, such forms are usually more difficult to parse than plain (E)BNF. The work in this thesis mostly confines itself to (E)BNF, because other forms are either easily transformed to it (as syntax diagrams) or do not have enough experience and acceptance behind them (as PEGs).

BNF itself is not well-standardised, and every ENBF dialect extends it in its own way. For example, the original proposition [252] suggested the use of square brackets to mark

optional parts of a rule: `[a]` stood for `a`$|\varepsilon$. Nowadays the question sign is used more often instead: `a?` stands for `a`$|\varepsilon$. Every standard usually includes the definition, formal or otherwise, of the "BNF-like notation" it will use throughout the text. EBNF was standardised by ISO [112], but the standard is not used. Sadly enough, the question Niklaus Wirth put into his title *"what can we do about the unnecessary diversity of notation for syntactic definitions?"* [252] in 1977 is yet to be answered.

Ultimately, BNF offers basic ways of defining nonterminals by combining them with terminal symbols. All EBNF dialects make use of the following extensions:

- ◇ They distinguish clearly between terminals and nonterminals by quoting terminal symbols or putting them in another font. It was not done in BNF, so the meaning of `a b c` would depend on whether there is a definition for `b`, and then `b` is a nonterminal symbol, or there is no definition, and `b` is a terminal symbol (text).

- ◇ They contain an explicit iteration construct to avoid heavy use of recursion for such simple cases. Usually it is {`a`} or `a`$\star$, both can stand for $\varepsilon|$`a`$|$`aa`$|$`aaa`$|\cdots$. The original proposition did not have the iteration construct, forcing the grammar engineer to choose between right and left recursion right away: for the example given above instead of `b ::= a`$\star$`;` one would write either `b ::=` $\varepsilon|$`ab;` or `b ::=` $\varepsilon|$`ba;`. Later it deemed to be inappropriate [27].

- ◇ They allow bracketing for putting things together in order to decrease the number of unnecessary nonterminals and rules for them. Earlier it was believed to be better to have simpler productions.

- ◇ They provide means to mark a symbol or a group of symbols as optional to avoid the explicit use of ε or `<EMPTY>`. For example, `a?` or $\texttt{a}_{opt}$ may be defined equal to `a`$|\varepsilon$. A frequently used alternative is definition of `[a b]` equal to `(a b)`$|\varepsilon$. This alternative was used on early stages more often due to the lack of normal brackets, nowadays a question sign is more commonly used, see Table 6.1.

- ◇ They explicitly mark the end of a rule by introducing a final character, a dot or a semicolon, or by paragraphing. In the early days of BNF a production was only observed by having a "::=" or similar sign. This is hardly appropriate for automatic language processing.

The section 6.3 contains a more detailed overview of various grammar definition notations used in language documentation.

2.8 Grammar idiosyncrasies and parsing technology

In order to perform syntactic analysis (parsing) of a program with a given grammar, a *parsing algorithm* needs to be used. Various approaches co-exist, like top-down parsing: LL(1), LL(k), LL($\star$); bottom-up parsing: CYK [255], LR(0)/SLR, LR(1), LR(k) [145], LALR(1) [204], LALR(k) [44, 128, 171], GLR [231], SGLR [26, 238]—each having its own restrictions on the way the grammar must be engineered to enable it. Several

normalisation strategies were introduced to better define the classes a grammar needs to fit for a specific parsing algorithm—such as Kuroda normal form [152], Greibach normal form [3], Chomsky normal form [170]. Backus-Naur Form [9] and Extended Backus-Normal Form [252] are de facto notational standards, but they exist in a variety of dialects that can be considerably different not only in terms of concrete notation, but also in terms of expressive power. When extracting a grammar from a software engineering artefact containing grammar knowledge, one needs to take care of them.

In [27] it is explained that the grammars are often not declarative but already geared towards a specific parsing technology or a grammar class. It is also motivated there that the declarativity and readability of such grammars is sacrificed considerably by the idiosyncratic constructs. For a more in-depth discussion about this issue the reader is referred to the quoted paper, whose title *Current Parsing Techniques in Software Renovation Considered Harmful* says enough. Its message is that nowadays one should use a parser generator that supports all context free grammars, i.e., a parser generator that puts no limitations on the grammar.

In this thesis we try to abstain from favouring one particular technology. In the next chapter, however, we will use scannerless generalised LR (SGLR) parser engine of the ASF+SDF Meta-Environment [16, 45, 140, 169, 213]. Since this is not the main topic of the thesis, the reader is referred for an overview of them to standard textbooks like [2, 83] or scientific papers like [17, 27].

It is important to be aware of existing grammar classes and their relations when constructing a unified grammar formalism as we will do in subsection 4.3.3. They are also an inevitable issue when developing a grammar extractor that maps a context-free grammar in a source notation to our unified formalism.

Grammars are usually considered extremely static objects, made once and not subject to any change. This is apparently not completely true, especially in the modern practice of dynamic and iterative engineering of domain specific languages that can be found everywhere. Continuous changing and maintenance of a grammar is extremely expensive in terms of man-effort, and in many tool building organisations the required knowledge is limited to a small number of specialists. By consistently using a well-developed grammar definition notation while keeping the grammar engineering activities free from idiosyncrasies, these costs can be reduced.

2.9 Grammarware and tool generation

A grammar forms the very basis of a language definition [143]. From a grammar, tools like parsers can be generated automatically. We have seen in section 2.2 and in papers like [141] that there are much more examples of grammarware than just parsers. Explicit or implicit, the grammars are omnipresent. However, there is a large difference between a formal executable grammar and a human readable language reference, either an ISO Language Standard [36, 113, 114, 115] or a compiler vendor specific Language Reference Manual [1, 37, 72, 183, 215, etc]. While language references are widely available on the Internet, formal grammars are not.

Lämmel and Verhoef have shown in their paper [164] how an IBM VS II COBOL

Coalesce operator introduction	

`if (a==null)`
`  return b;`
`else`
`  return a;`
	`return a ?? b;`

Figure 2.1: Code example of a transformation caused by changes in the language.

grammar can be obtained from the language reference manual made by IBM. The result of that research was made available via [241], it was the first COBOL grammar freely available on the web.

We have already mentioned that parsers can be generated from grammars. Many other tools can be generated from grammars and language definitions in general, directly or indirectly, such as:

- ◇ Tools that check that certain constructs, like *obsolete* keywords, are not used;
- ◇ Tools that migrate source code from one language variant to another, as in [218] where a collection of rules was introduced to go from COBOL'74 to COBOL'85 and [235] where these rules were extended and integrated into an algorithm that can be used on a large scale (millions of lines of code);
- ◇ Tools that enforce certain layout standards, which are called *pretty-printers* [24, 29];
- ◇ Browsable versions of a language definition [241], as described above.

For example, consider C# programming language updating its language specification to C# 2.0. Change in language features implies change in the source code. Transformations like the one shown on Figure 2.1 can be shipped with the specification and deployed semi-automatically in order to update the source code with C# 2.0 features.

Changes in a programming language lead to changes in the source code written in that programming language, this is a process called co-evolution [64, 33]. New constructions are introduced, old solutions are deemed unreliable or inefficient, API changes, new design methods come into play, etc. C# is still a new language and its code is never considered legacy, but sooner or later even C# code will require restructuring and renovation.

New language features can be roughly classified into two groups: syntactic sugar and paradigm shifting. Replacing nested enumerator classes with a `yield` construct (C# 2.0, §22.4), replacing traditional database access with LINQ (C# 3.0, §26.7), introduction of nullable types (C# 2.0, §24) or anonymous types (C# 3.0, §26.5) can serve as examples of paradigm shifts. The automation of those requires considerable research effort. One of the examples of the syntactic sugar is shown in Figure 2.1: §19.5 introduces a null coalescing operator `a ?? b`, which results in `a` if `a` is not `null` and `b` otherwise. `null` could be used in earlier versions of C# for objects and therefore the pattern on the left can possibly be found in the source code, with `a` and `b` being any arbitrary expressions. Replacing this pattern with new operator will increase code effectiveness and future maintainability and

extendability. The actual corresponding transformational rules can be written in a special grammar-based language like XBGF or ASF.

2.10 Language documentation qualities

The intended purpose of language documentation is not always stated clearly and explicitly, but most of the time it is twofold: to define the language and to explain it. The first objective is intended for computers and is traditionally achieved by presenting a grammar and sometimes a formal semantic definition. The second objective is intended for humans, mostly it is plain English text with tables and figures. The document itself is a combination of parts defining the language for computers and parts explaining it for human users. They are mixed with each other and practically non-detachable.

As explained above, the defining part should be good enough to provide a language engineer with sufficient material to adapt the existing infrastructure to deal with the code in the defined language: to parse it, transform it, detect errors, perform analyses, generate grammarware. The explaining part, however, has significantly different goals.

The human readable part of a language definition is meant for people to read or employ otherwise in order to learn about certain language characteristics. It can be utilised as a complementary part to the formal grammar when it is unclear, ambiguous or just overcomplicated. It can be utilised to learn the language as a whole. It should also be possible to utilise it to understand small aspects of the language when an adjustment about them is about to be made in the IT portfolio. The requirements for this part are rather different. This thesis is mainly aimed at solving technical problems for the formal part, due to the fact that its author is a computer scientist and neither a linguist nor a psychologist, but the rest of this section is dedicated to this issue nevertheless.

Usability is a vaguely defined term that refers to the overall clarity with which the users learn how to use the product and the satisfaction of such use. ISO 9126 [116] defines it as "a set of attributes that bear on the effort needed for use, and on the individual assessment of such use, by a stated or implied set of users". When talking about language specifications, good usability means that anyone can take the standard and deploy it. A standard that is several thousand pages long, contains incorrect grammar, incompatible code samples and vague explanations, ranks low on usability.

Searchability means locating the wanted items quickly and efficiently without having to browse manually through big volumes of data. *Browsability* [241] and the use of other cross-references addresses this problem very well by separating content from the presentation with making the presentation highly interactive on a basis of information devised from the content. Most of the standards that exist nowadays and continue to evolve are becoming more searchable and browsable, usually distributed as hyperlinked PDF files or an on-line manual.

Reusability means that existing pieces of code and knowledge can be used to build new ones. Software reuse is an everyday practice and one of the basic and crucial foundations of software engineering industry since 1968 [177]. Language standards can be reused as well; right now they are not reused explicitly even within the same organisation and among similar languages. For example, ISO C [119] and ISO C++ [110] standards have tons of

subtle differences in places that should be identical and stored in a separate repository. The structure of language specifications—sectioning, text organisation, etc.—can certainly be reused for all languages almost without any adjustments. Syntax definition formalism should be re-used, as will be discussed later in section 6.3.

Readability (understandability) refers to the ease with which a human reader can comprehend the content. While there do exist formal methods of measuring the readability (such as Automated Readability Index, Harry McLaughlin's Simple Measure of Gobbledygook, Flesch–Kincaid Readability Test and others), this issue will not be pursued in this thesis.

Learnability[1] is an abstract measure of how fast and effective it is for a human to learn new concepts from the documentation. Consistent presentation of the data does help to achieve this goal and make the language standard learnable.

Operability and *maintainability* refer to the ability to keep system operational or be able to quickly return it to operational state once a bug is found or experienced. Language standards that are kept up to date with the latest development and language engineering results and have good supportive infrastructure are maintainable. *Extensibility, adaptability, integrability* and similar properties measure the extent to which maintenance activities can include major improvements.

While it is up to usability experts and cognitive scientists to fully judge the quality of the completed manual, it is certainly possible for grammarware technology to pinpoint technical problems that can be found and fixed in a semi-automated fashion in order to raise the quality level of language specifications.

2.11 Standardisation bodies

There are many organisations in the world that are dedicated to developing, quality checking, approving and maintaining language documents. The complete list follows, it will be used in the language document schema in subsection 6.5.2 directly, and specific remarks will be made in chapter 6 about the work of different subcommittees and working groups.

ANSI (American National Standards Institute) is a private non-profit organisation that is based in the USA [7]. It oversees the development of a wide range of American standards, and coordinates that those standards can be used worldwide. ANSI is an ISO/IEC representative via United States National Committee (USNC), and it participates in most ISO/IEC activities. Several programming language specifications started out as ANSI standards and were later adopted by ISO (most notably ISO C [119] and ISO COBOL [113]).

ECMA (European Computer Manufacturers Association), or **Ecma International** (new name since 1994), is an international private standards development organisation that takes care of ICT-related issues [60]. It started in Europe in the early sixties, now it operates worldwide. Most well-known Ecma specifications are those for C# [225] and JavaScript [224]. The Java programming language specification was submitted to ECMA

[1] Not to be confused with "language learnability" [210] which is a linguistic synonym for "grammar inference" or "fuzzy parsing" [148].

but withdrawn afterwards without receiving official status. Some Ecma standards are ISO standards as well.

IEEE-SA (Institute of Electrical and Electronics Engineers Standards Association) is a standardising branch of IEEE [93]. It maintains almost a thousand of active industry standards in electrical and electronics engineering and computer science, and processes around 200 new proposals each year. However, its activity in the field of languages is limited to the Scheme programming language that was standardised as IEEE Std 1178-1990 (later it became an ANSI standard as well).

ISO (International Organization for Standardization) [97] and **IEC (International Electrotechnical Commission)** [92] are huge international standards development and distribution bodies. While the former concentrates its effort more or less on software-related issues and the latter one on hardware ones, there was quite an overlap that led to establishing a Joint Technical Committee (JTC1) with 18 sub-committees dealing with different areas. The work of SC22 [100] a sub-committee on programming Languages, their environments and systems software interfaces, and its components were described in earlier sections.

ISO (International Organization for Standardization) is one of the biggest standardisation bodies that maintains a standardisation committee or a working group for every significant topic. ISO/IEC JTC1/SC22 [100] is the international standardisation subcommittee that deals with programming languages, their environments and system software interfaces. Inside the subcommittee there are active working groups for several mainstream programming languages:

- ◇ **WG4** [101]: COBOL is the most used programming language in industry. While the part of modern programming languages like Java is growing rapidly, they still have to beat more than 225 billion lines of COBOL code [129, 163] which are already in use. WG4 maintains the ISO/IEC 1989:2002 standard [113] keeping it up to date and constantly introducing new features like object-oriented programming or Unicode support.

- ◇ **WG14** [104]: C is a widely used general-purpose middle-level imperative procedural programming language. The unsatisfactorily low level of the language and drastic maintainability deterioration due to heavy use of the C Preprocessor [223] narrow its application area considerably (mainly as an alternative to the assembly language in system programming and embedded software).

- ◇ **WG21** [108]: C++ started as a high-level object-oriented extension to C, later evolved into a much bigger independent language with data abstraction, generics, runtime typing and multiple inheritance. C++ is slowly being replaced by modern successors like Java or C#. If counted together, C/C++ is the second most used programming language with 20% of installed software and 180 billion lines of code [130].

- ◇ **WG9** [103]: Ada was created as a language for embedded and real-time systems. It was a result of extensive research done by the High Order Language Working Group, a group of professionals employed by the United States Department of Defense. The use of Ada in DoD projects was mandatory for years, but is not anymore.

The failure of Ariane 5 launch system due to improper run-time error handling in Ada programming language [52] raised some questions for its suitability.

- ◇ **WG5** [102]: Fortran [11] is less used in industry, but still a lot of computational code was and is being written in it. Popular Fortran libraries LINPACK and EISPACK evolved into an even more widely used programming language and a numerical computing environment called MATLAB. Recently (on 15th of January, 2008) WG5 released a new beta version of the ISO Fortran specification that is as of now still in the internal discussion phase.

- ◇ **WG19** [107]: The working group for standardising extended Backus Naur Form and Vienna Development Method for declaring syntax and semantics correspondingly. WG19's notable output used in this research is [112] that will be discussed in the following chapters together with other (E)BNF dialects.

There are also working groups for mainstream industry-unrelated programming languages like ISLisp [105] (they have already been outrun by ANSI [81]) or Prolog [106]. All other numbers from WG1 throughout WG22 belong to the working groups which are no longer active for some reasons (PLIP, Pascal, APL, Algol, PL/I, Basic, Modula-2, Java).

ITU (International Telecommunication Union) is the oldest standardisation body in the world [122]. It handles most of standards in radio and telecommunications: for example, Integrated Services Digital Network (ISDN) and Asymmetric Digital Subscriber Line (ADSL) are ITU standards.

IETF (Internet Engineering Task Force) is an open standardisation body with no formal membership [96]. It operates on Requests for Comments (RFCs) that can be submitted by anyone, are freely passed around for commenting. There is a very fixed structure of an RFC and there are several strict levels of approval (proposed, draft, full internet standard). TCP/IP protocol stack, Unicode and Universal Resource Locators (URLs) are examples of approved RFCs. IETF often collaborates with W3C and **ISO/IEC**.

OASIS (Organization for the Advancement of Structured Information Standards) (ex-"SGML Open") is a global consortium for standardisation on a field of web services and e-business [194]. DocBook [245] and DITA [195] are OASIS standards.

WSA (Website Standards Association) is a group of industry experts having several common goals, one of which is to identify and promote the minimum standards that all websites should meet [253]. This recently created organisation does not have any significant standards yet, but it seems to strive for a wide range of them, from marketing to usability.

W3C (World Wide Web Consortium) is one of the largest standardisation bodies active in the Internet-related areas. It develops inter-operable technologies like specifications, guidelines and grammarware to "lead the Web to its full potential" [242]. Within the thesis we do not use W3C recommendations for grammar recovery, but we did use many of its web-related standards. Basically everything that is used to present, retrieve, transmit or store information in the world wide web has been or is being standardised by W3C. CSS [19], XHTML [8], XML [30], XSLT [137], XML Schema [75, 208] and XPath [15] are W3C Working Drafts and W3C Recommendations used in this thesis.

Language	Input, LOC	Programmed, LOC	Generated, LOC
ANTLR	122	345	
ASF		763	
BGF	5436077	6860	5432766
C#	∼100000		
Dot			3206
Ecore	278	51	227
FST		1540	
(X)HTML	40089		4926
LCF		816	343
LDF			27616
LLL	731	731	5877
Makefile		2067	
Prolog	404	6413	
Python		5594	
SDF	82	649	
TEX		8187	14346
XBGF		34069	28502
XLDF		6746	
XML Schema	95	4717	
XSLT		5926	
XQuery		372	

Table 2.1: Languages used in the thesis, in lines of "code". "Input" means data that has been used as an input for our tools, whether taken from third parties, programmed or generated. "Programmed" means the code that we have written. "Generated" means code produced by our automated tools. It is possible for generated code to be used as an input on another step (e.g., it is common with BGF). The table is generated automatically from the repository of SLPS [263].

2.12 Languages used in the thesis

The approach taken involves a string of different languages itself, we briefly introduce them in this section. Languages to which we have applied our methods, are also listed. For the sake of convenience we introduce all languages first in lexicographical order, and then we provide a graph to depict the role of all the languages in our approach. Table 2.1 summarises the number of lines of code per language.

ANTLR — one of the sources for the Factorial Language extractor prototype suite was ANTLR [202] grammar file, see Listing 4.5.

ASF — Algebraic Specification Formalism [16] is a language for specifying grammar transformation formulæ in ASF+SDF Meta-Environment [22]; it was used to extract BGF from SDF and BTF from AsFix [25].

BGF — BNF-like Grammar Format, our XML-enabled version of Backus Normal Form that was used for representing grammar knowledge, see subsection 4.3.3 and section 7.4.

BTF — BGF Tree Format, was used as a complementary XML format for storing parse trees. It was needed for coupled transformations (test set convergence), the part of research that did not end up in the thesis.

C# was one of the case studies [257], its grammar was semi-automatically extracted from [225] and further refactored, see chapter 3.

Dot was generated by Language Convergence Infrastructure from LCF as a cheap way to produce a PDF diagram with Graphviz [74]. All convergence graphs, both abridged and detailed ones, were generated in Dot form with LCI.

EBNF dialects were used while conducting research on language definitions, see section 6.3, as well as for pretty-printing BGF, see section 7.4.

Ecore — Eclipse Modeling Framework's format for models can also be treated as a grammar knowledge source. One of the implementations of Factorial Language was made in Ecore, extracted and converged with other FL grammars, see Listing 4.3 and Listing 4.8.

FL — Factorial Language, was used for prototyping grammar convergence techniques in [166]. For that project FL was implemented in ANTLR (see Listing 4.5), SDF (see Listing 4.7), C#, C++, Converge, Ecore, Haskell, Java, LDF, Prolog (see Listing 4.6), Python, Smalltalk, XQuery and XSD (see Listing 4.10). The only extraction sources really used for the thesis were ANTLR, Ecore, JAXB (see Listing 4.12), Java object model (see Listing 4.11), SDF, XSD and Prolog.

FST — the grammar transformation language of Grammar Deployment Kit [149] was used for C# case study detailed in section 3.2.

HTML — non-well-formed HyperText Markup Language documents [77, 78, 79] were used as a source of Java grammar knowledge in [167] and chapter 5.

Java was used in a large language convergence case study [167]: six different Java programming language grammars were extracted from specifications, transformed and converged.

LCF — LCI Configuration File, an XML-based configuration file for Language Convergence Infrastructure, see section 4.10.

LDF — Language Documentation Format, a BGF-using format for language definitions, see chapter 6.

LLL was used in Grammar Deployment Kit [149] for storing grammars for input, output and all intermediate steps, see subsection 3.2.2.

Makefiles were used as a low-level means of expressing dependencies. When in the context of grammar convergence the makefiles started to become too complicated and obfuscated, we developed a domain specific language (see LCF) that delivered both simplicity and controllability.

Prolog was the main implementation language for XBGF and BGF/BTF-based coupled transformations (courtesy of Prof. Dr. Ralf Lämmel); Prolog definite clause grammar implementation of Factorial Language was also one of the sources for language convergence in [166].

PROTO was an ISO toy language we adopted for demonstrating some grammar discipline concepts in [143].

Python was used to wrap regular expressions for ANTLR grammar extractor and in various other tools that required scripting; Language Convergence Infrastructure [166] was completely implemented in Python; the scanner and the recovering parser for the Java Language specification was also done in Python.

SDF — Syntax Definition Formalism [86] is a language for representing grammar knowledge in ASF+SDF Meta-Environment [22]; was used as one of the sources in Factorial Language extractor prototype suite, in the SDF to BGF extractor, and as a target format for C# case study in Grammar Deployment Kit [149].

TEX — the thesis and the papers were typed in LATEX, as well as, naturally, all automatically generated tables and language documentation.

XBGF was the main transformation language for grammar convergence, see chapter 7. The total volume of XBGF files in Table 2.1 does not include XBGF calls from XLDF.

XHTML was used always when hypertext content needed to be generated (e.g., from LDF)—most importantly in [261].

XLDF was the transformation language behind chapter 6. The total volume of XLDF files in Table 2.1 includes built-in XBGF calls, as in subsection 6.6.19.

XML was used in form of BGF, BTF, LCF, LDF, XBGF, XLDF, XMI, XSD, XSL.

XML Schema was used for all evolved XML formats: BGF, BTF, XBGF, LDF and LCF; one of the Factorial Language grammars was also extracted from its XSD, see Listing 4.10.

XSL:FO was used as an intermediate stage in one of the ways to generate PDF from LDF.

XSLT sheets were used to pretty-print BGF grammars and XBGF transformation scripts presented in this thesis and in corresponding publications; XSLT was also a presentation language for transforming LDF to XHTML and XHTML further to XSL:FO.

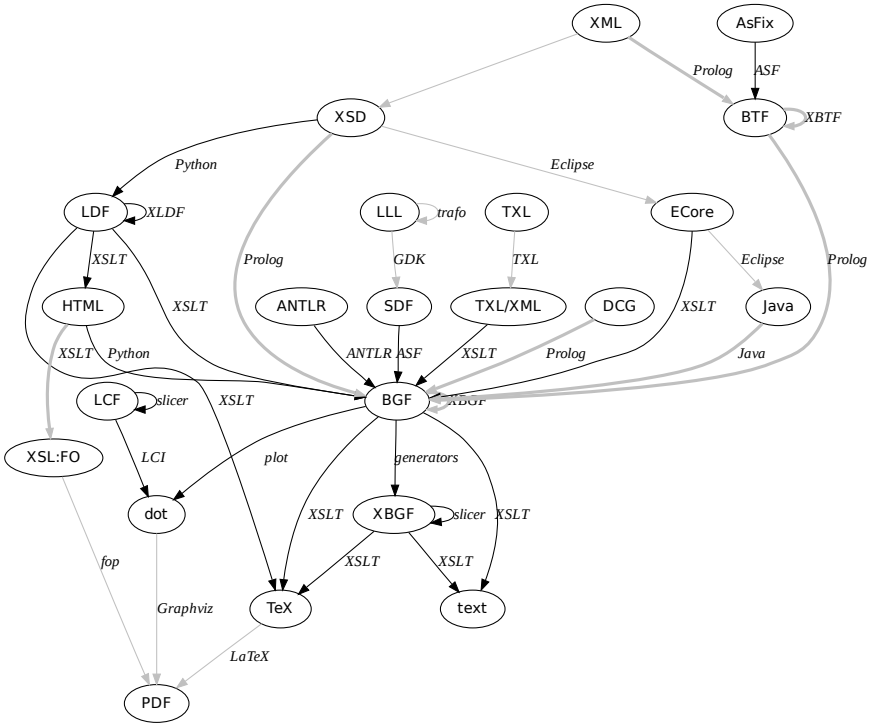


Figure 2.2: Dependency graph showing all language transformations. Thin grey lines denote tools present prior to this research. Thick grey edges are for co-developed transformations.

XPath dialect [173] was used internally by all Python scripts that were working with XML (in particular, Language Convergence Infrastructure), and classic XPath as a standalone tool for analyses, metrics and presentation.

XQuery scripts were generating output when XPath was not able to express the designed benchmarks.

2.13 Transformations used in the thesis

Figure 2.2 shows a dependency graph of all languages and other artefacts of grammar knowledge used in this work. Every node in this graph demonstrates a format, a form or a view of a language or its part. Every edge is a transformation. All solid black edges were designed and implemented as a part of this research project. Thin grey edges

represent external transformation facilities such as those provided by Eclipse Modelling Framework [59], Graphviz [74] or LaTeX. Thick grey edges are for transformations that were co-developed or taken from third parties. We list them below:

- ◇ XBGF is a collaboration effort with Prof. Dr. Ralf Lämmel.
- ◇ Java2BGF, XSD2BGF, XML2BTF, DCG2BGF and BTF2BGF are courtesy of Prof. Dr. Ralf Lämmel.
- ◇ HTML2XSLFO is courtesy of Antenna House, Inc., use permission granted.

Chapter 3

Case study on recovery

We found more errors than one would expect from a language reference manual.

Ralf Lämmel, Chris Verhoef,
2000 [164]

As a starting point for our research we have conducted an experiment by carrying out a case study with the most modern language document available at the time, and producing a working grammar for parsing source code in that language. A modern programming language was intended to be used to avoid claims that the problems we encounter are specific to legacy languages only, since prior research was about COBOL and PLEX. It was during the execution of this experiment that we realised how much effort was required just due to the simple lack of disciplined grammar engineering in the process of grammar making and standard making. A range of questions has been raised accordingly to be answered by the research done in the later chapters of this thesis.

This chapter utilises already existing research results from [154, 164] to show how they can be incorporated in the scope of section 1.2 in language specification development, evolution and maintenance. It makes the reader familiar with a range of issues that are important in such activities and demonstrates how such questions should be dealt with, providing a step-wise plan illustrated with examples from the C# specification. The aim of this research was not only to reconstruct the grammar for C#, but also to collect grammar recovery experience and to provide guidelines that can profitably be used by any scientist or engineer attempting a similar recovery task.

3.1 Contributions

- ◇ This chapter serves as an initial experiment in grammar recovery that used contemporary methods for grammar extraction and transformation and utilised a language specification as a source of grammar knowledge.

- ◇ A case study is presented where a grammar of C# is derived from the corresponding language specification and transformed into a working one capable of parsing C# code.
- ◇ The C# case study is generalised as a plan for semi-automatic grammar recovery from a language standard, language specification or language reference for its further use in the standardisation framework that will be elaborated into a prototype in the next chapters.

The technical results of this chapter were briefly described, published and presented at the participants' workshop of the International Summer School on Generative and Transformational Techniques in Software Engineering as *Correct C# Grammar too Sharp for ISO* [257]. In the next version of the standard some of the errors were removed [115]. We have no information about Microsoft actually using our results in their work, but since not all errors were fixed, we presume they did not use an engineering approach for this. Our C# grammar was later made publicly available via `http://www.cs.vu.nl/grammarware/browsable/CSharp` [241] and forms now a part of the *SLPS Grammar Zoo* at `http://slps.sf.net/zoo` [260]. General discussion on EBNF as a grammar definition formalism, as well as on grammar engineering techniques like de-layering and relaxation overlaps somewhat with the ISO note titled *Language Standardization Needs Grammarware* [143]. [143] was written in collaboration with Dr. Steven Klusener (Vrije Universiteit Amsterdam), who also presented the work to the international programming languages subcommittee ISO/IEC JTC1/SC22.

3.2 Semi-automated recovery of C# grammar

This section is an application of grammar recovery transformations from [154, 164] to the C# 1.0 grammar from [114, 184, 225]. It demonstrates how such questions should be dealt with, executing all steps needed to get the working grammar out of the C# specification. However, the objective will be to provide guidelines that can profitably be used by any scientist or engineer attempting a similar recovery task.

There is a complementary web page that contains the results of this work in a browsable form: `http://www.cs.vu.nl/grammarware/browsable/CSharp` [241].

3.2.1 Step 1: Obtaining the standard

Some standards are distributed for free, but others require a fee to be paid. In the case of C#, both ISO/IEC 23270:2003 [114] and ECMA-334 [225] standards are freely downloadable from their corresponding web sites (see bibliography or the aforementioned complementary web page for URLs).

The original language specifications for C# are distributed in the form of Microsoft Word documents (`.doc`). Jon Jagger used a simple lexical tool to generate a hyperlinked version of Appendix A of ECMA-334 [125]. Later we will use Jagger's notation to generate our hypertext version, but it seemed more trustworthy to rely on the original language document for the extraction step.

```
specification : rule+;
rule          : ident ":" disjunction ";";
disjunction   : {conjunction "|"};
conjunction   : term+;
term          : basis repetition?;
basis         : ident
              | literal
              | "%epsilon"
              | alternation
              | group
              ;
repetition    : "+" | "*" | "?";
alternation   : "{" basis basis "}" repetition;
group         : "(" disjunction ")" ;
```

Listing 3.1: EBNF of LLL grammar format in LLL notation [149, page 3].

3.2.2 Step 2: Extracting the grammar

Once the grammar is at hand, it must next be fed into the grammar engineering infrastructure that was chosen to be used throughout the rest of the project. Many tools use dialects of extended Backus-Naur form, but as we will see in section 6.3, no two (E)BNFs are alike.

In this case study it was observed that the C# grammar used font face to mark terminal symbols, and that information was inevitably to be lost, and quotation marks must be added by hand, if the grammar were to be copy-pasted from a PDF of the standard. However, Microsoft has an hypertext version of the specification [184], which can be parsed with W3C standards. Jagger's browsable version mentioned above could have also been used, but we wanted to eliminate even the slightest chance for introducing mistakes by preferring the specification itself or the sources that have been derived directly from it by the trusted parties, such as Microsoft.

LLL (Lightweight LL parsing, pronounced as "el-cube") is a language used by the Grammar Deployment Kit [149], which was used for this C# grammar recovery. It is an EBNF-like language that uses `?` to denote optional symbols, double quotes terminals, treats `;` as the end of a rule and regards the bar (traditional | for choice) as the lowest priority operation. Listing 3.1 gives the full formal definition of LLL.

SDF (Syntax Definition Formalism) [86] is a more advanced formalism employed in the ASF+SDF Meta-Environment, which was also used in this project (GDK generates SDF automatically from LLL). It allows for specifying in much more detail how the source code must be parsed, providing a grammar engineer with access to lexical rules and production priorities.

LLL and SDF also make a distinction between "one or more" ($+$) and "zero or more" ($\star$) repetitions (closures). Another extension of both LLL and SDF gives support for "comma-separated lists" as a special kind of repetition when a terminal—a dot, a comma or a semicolon in most cases—is inserted between entities, but not at the end. The syntax

is:

$$\{\texttt{a ","}\}^{+} \quad \equiv \quad a\ (\texttt{","}\ \texttt{a})^{\star}$$

3.2.3 Step 3: Fixing misprints

As long as language specifications are created by hand, they are to bound contain some amount of misprints, mistypings, multiple different names for the same entity, duplicate definitions, etc. For instance, in the recovery of the COBOL grammar from the IBM manual [28] a misspelled "DEGUGGING MODE" terminal was found considerably later than the recovered result was put online.

The `boolean-expression` nonterminal symbol was defined twice in the C# grammar: one time with other kinds of expressions in §A.2.4 of [225], and the other time along with conditional operators in §A.2.5 of [225]. This issue has been fixed by removing the second definition in the next versions of the standard. A couple of other mistakes found and described in the forthcoming sections were most probably mistypings as well.

3.2.4 Step 4: Completing a formal part

The C# grammar had informal words scattered inside it:

```
letter-character::
  A Unicode character of classes
    Lu, Ll, Lt, Lm, Lo, or Nl
  A unicode-character-escape-sequence representing
    a character of classes Lu, Ll, Lt, Lm, Lo, or Nl

decimal-digit:: one of
  0 1 2 3 4 5 6 7 8 9

integer-type-suffix:: one of
  U u L l UL Ul uL ul LU Lu lU lu
```

The "one of" looks quite similar to "the bar" in BNF, and indeed it can be automatically translated to the bar notation in the case of limited number of choices (like above). The textual references to the Unicode standard should be either removed or formally expanded. While the latter solution is certainly cleaner and better if one had the Unicode standard already available, in this project we removed such references at all. They were replaced with lexical rules that were written manually to be as close to the original as possible[1]. Those rules were used for parsing, yet the browsable version generated as the result of this project contains the original verbal descriptions put back—they are better readable and make the grammar identical even in Unicode-enabled environments.

The completed lexical part of C# is shown on Listings 3.2–3.4.

3.2.5 Step 5: Relaxation

Consider the following part of the C# grammar (A.2.2 Types):

[1] ASF+SDF Meta-Environment is not Unicode-friendly.

```
%% C# Language Specification, section A.1.6, pages 336–337
[A-Za-z\_][A-Za-z01-9\_]* → Identifier
%% C# Language Specification, section A.1.8, pages 338–339
"true" | "false" → Boolean-literal
"null" → Null-literal
[FfDdMm] → Real-type-suffix
[\"][\"] → Quote-escape-sequence
[x][01-9a-fA-F]([01-9a-fA-F]([01-9a-fA-F][01-9a-fA-F]?)?)?
 → Hexadecimal-escape-sequence
%% Optimisation by VVZ:
%% Simple–escape–sequence +
%% Hexadecimal–escape–sequence +
%% Unicode–character–escape–sequence =
[\'\"\\0abfnrtv]
| Hexadecimal-escape-sequence
| Unicode-character-escape-sequence → Escape-sequence
~[\'\\\n] | ( [\\] Escape-sequence ) → Character
[\'] Character [\'] → Character-literal
~[\"\\\n] | ( [\\] Escape-sequence ) → Regular-string-character
~[\"] | Quote-escape-sequence → Verbatim-string-character
[\"] Regular-string-character* [\"] → String-literal
[\@][\"] Verbatim-string-character* [\"] → String-literal
%% C# Language Specification, section A.1.3, page 335
([uUlL]|"UL"|"Ul"|"uL"|"ul"|"LU"|"Lu"|"lU"|"lu") → UL
[01-9]+UL? → Integer-literal
("0x"|"0X")[01-9a-fA-F]+UL? → Integer-literal
[01-9]*[\.][01-9]+Exponent-part? Real-type-suffix? → Real-literal
[01-9]+ Exponent-part Real-type-suffix? → Real-literal
[01-9]+ Real-type-suffix → Real-literal
[eE][\+\-]?[01-9]+ → Exponent-part
%% C# Language Specification, section A.1.5, page 336
[u][01-9a-fA-F][01-9a-fA-F][01-9a-fA-F][01-9a-fA-F]
 → Unicode-character-escape-sequence
[U][01-9a-fA-F][01-9a-fA-F][01-9a-fA-F][01-9a-fA-F]
[01-9a-fA-F][01-9a-fA-F][01-9a-fA-F][01-9a-fA-F]
 → Unicode-character-escape-sequence
```

Listing 3.2: Lexical part of C# grammar in SDF, part 1: lexical syntax

```
"null"  → Identifier {reject}
"true"  → Identifier {reject}
"false" → Identifier {reject}
%% C# Language Specification, section A.1.8, pages 338−339
Boolean-literal
| Integer-literal
| Real-literal
| Character-literal
| String-literal
| Null-literal → Literal
lexical restrictions
 Identifier ⊬ [A-Za-z01-9\_]
 Literal ⊬ [A-Za-z01-9\_]
 Integer-literal ⊬ [A-Za-z01-9\_]
 Hexadecimal-escape-sequence ⊬ [01-9A-Fa-f]
 Unicode-character-escape-sequence ⊬ [01-9A-Fa-f]
```

Listing 3.3: Lexical part of C# grammar in SDF, part 2: restrictions

```
non-array-type:
  type

array-type:
  non-array-type rank-specifiers

rank-specifiers:
  rank-specifier
  rank-specifiers rank-specifier

type:
  value-type
  reference-type

value-type:
  struct-type
  enum-type

struct-type:
  type-name
  simple-type

enum-type:
  type-name

reference-type:
  class-type
  interface-type
  array-type
  delegate-type

class-type:
  type-name
  "object"
  "string"

interface-type:
  type-name

delegate-type:
  type-name
```

```
module Comment-layout
imports Layout
exports sorts Asterisk Comment LineComment
exports lexical syntax
      "/*" ( ~[\*] | Asterisk )* "*/" → Comment
      [\*] → Asterisk
      "//" ~[\n]* [\n] → LineComment
      Comment → LAYOUT {prefer}
      LineComment → LAYOUT {prefer}
 lexical restrictions
      Asterisk ⁄[\/]
 context-free restrictions
      LAYOUT? ⁄[\/].[\*]
      LAYOUT? ⁄[\/].[\/]
```

Listing 3.4: Lexical part of C# grammar in SDF, part 3: comments

In this part of the grammar, we can find five different ways to get from `type` to `type-name`, which is very ambiguous and unnecessary for parsing. However, for someone who would like to run type checks later with the working grammar, this information is useful and should be kept (perhaps by tweaking definitions of those types if the grammarware cannot deal with ambiguities). For someone who needs the C# grammar for other purposes, this is irrelevant and should be removed. This is how the same grammar piece looks in the refactored C# grammar (in LLL):

```
type
  : non-array-type rank-specifier*
  ;
```

```
non-array-type
  : qualified-identifier
  | built-in-type
  ;
```

```
built-in-type
  : integral-type
  | built-in-class-type
  | "bool"
  | "decimal"
  | "float"
  | "double"
  ;
```

```
built-in-class-type
  : "object"
  | "string"
  ;
```

This instance actually covers more than the previous one. `qualified-identifier` is a dot-separated list of names of namespaces like `System.Windows.Forms.Button` which replaces `type-name`. There is place for other solutions: for example, we could both keep type information and remove ambiguities by introducing additional nonterminal symbols in all places where `type-name` was used.

3.2.6 Step 6: Removing idiosyncrasies from the grammar

As we explained in section 2.3, a "yaccified" grammar uses recursive definitions to introduce naturally iterative language constructs. Since nowadays it is maintained that declarativity and readability of a grammar deteriorates if parsing techniques influence the grammar engineering process [27], we should try to spot such idiosyncratic constructions and replace them by more general ones. The grammars found in the C# specifications [114, 115, 184, 225] use explicit left recursion, two examples follow:

```
argument-list:
  argument
  argument-list "," argument
```

```
statement-list:
  statement
  statement-list statement
```

SDF notation has postfix repetition metasymbols "$\star$" and "$+$", as well as confix notation $\{\cdots\}$ for terminal-separated lists. These concepts are abstract enough to be easily mapped to any technology-specific variation, be it repetition, left or right recursion. After refactoring the same productions will look like this (in LLL):

```
argument-list:
  {argument ","}+;
```

```
statement-list:
  statement*;
```

While the result is independent of parsing technology, it is not independent of the grammarware toolkit one is using: namely, the support for comma-separated lists must be present, otherwise it is not possible to perform the first of the last two refactorings.

A similar issue is what we call *layering* [166], where a hierarchy of definitions is introduced, each on referring to the next one and recursively to itself. This approach is one of the popular solutions to introducing operator priorities, but it comes with a disadvantage of having heavily cluttered and unreadable parse trees. In order to understand the issue completely, let us consider one example in detail; here is how expressions are defined in the C# specification [115, pages 454–456] (productions re-ordered for reader's convenience):

```
expression:
  conditional-expression
  assignment
```

```
conditional-expression:
  null-coalescing-expression
  null-coalescing-expression ? expression : expression
```

```
null-coalescing-expression:
  conditional-or-expression
  conditional-or-expression ?? null-coalescing-expression
```

```
conditional-or-expression:
  conditional-and-expression
  conditional-or-expression || conditional-and-expression
```

```
conditional-and-expression:
  inclusive-or-expression
  conditional-and-expression && inclusive-or-expression
```

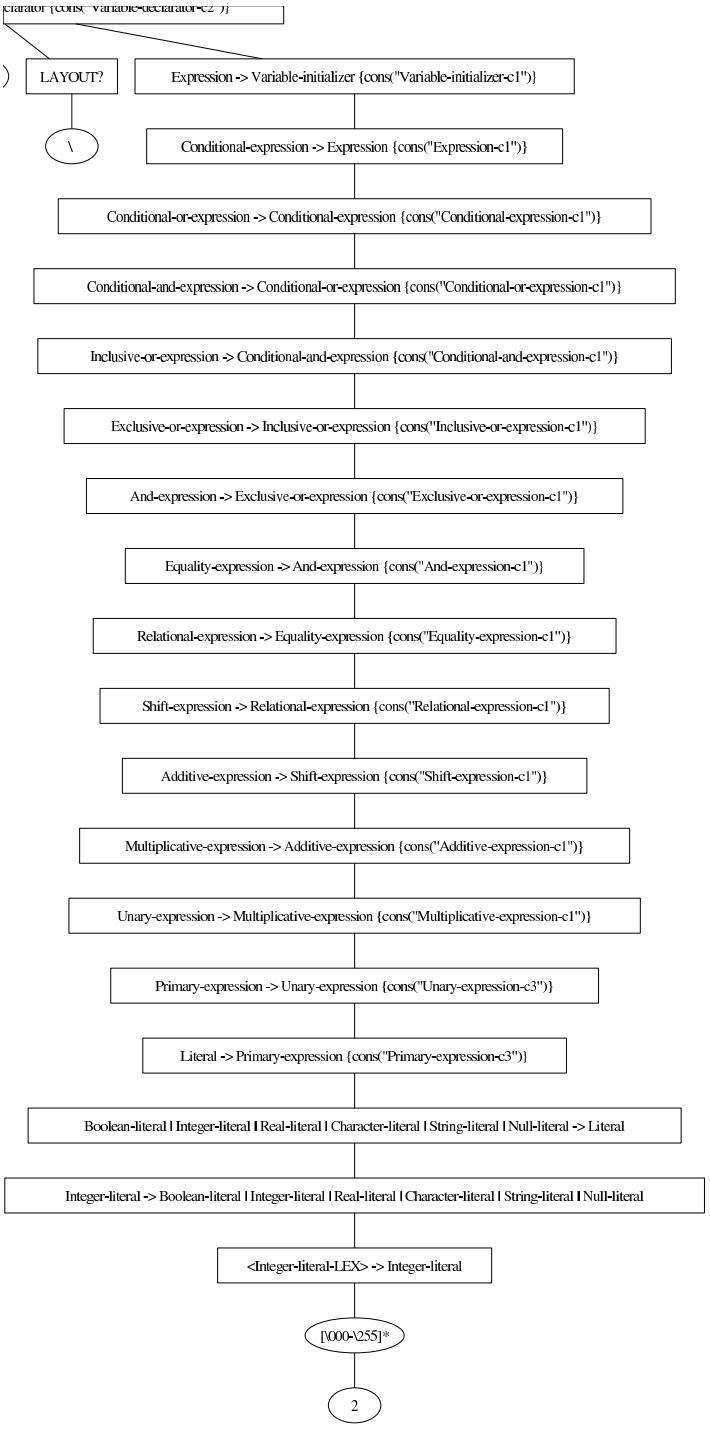


Figure 3.1: A number parsed as an expression.

```
inclusive-or-expression:
  exclusive-or-expression
  inclusive-or-expression | exclusive-or-expression
```

```
exclusive-or-expression:
  and-expression
  exclusive-or-expression ^ and-expression
```

```
and-expression:
  equality-expression
  and-expression & equality-expression
```

```
equality-expression:
  relational-expression
  equality-expression == relational-expression
  equality-expression != relational-expression
```

```
relational-expression:
  shift-expression
  relational-expression < shift-expression
  relational-expression > shift-expression
  relational-expression <= shift-expression
  relational-expression >= shift-expression
  relational-expression is type
  relational-expression as type
```

```
shift-expression:
  additive-expression
  shift-expression << additive-expression
  shift-expression >> additive-expression
```

```
additive-expression:
  multiplicative-expression
  additive-expression + multiplicative-expression
  additive-expression - multiplicative-expression
```

```
multiplicative-expression:
  unary-expression
  multiplicative-expression * unary-expression
  multiplicative-expression / unary-expression
  multiplicative-expression % unary-expression
```

```
unary-expression:
  primary-expression
  + unary-expression
  - unary-expression
  ! unary-expression
  ~ unary-expression
  pre-increment-expression
  pre-decrement-expression
  cast-expression
```

```
pre-increment-expression:
  ++ unary-expression
```

```
pre-decrement-expression:
  -- unary-expression
```

```
cast-expression:
  ( type ) unary-expression
```

```
primary-expression:
  array-creation-expression
  primary-no-array-creation-expression
```

```
module UnaryExpressionPriorities
exports
 context-free priorities
  Primary-expression → Unary-expression >
  {non-assoc:
   Expression-unary-operator Unary-expression → Unary-expression
   "(" Type ")" Unary-expression → Unary-expression}
module ExpressionPriorities
exports
 context-free priorities
  Unary-expression → Expression >
  {left: Expression "*" Expression → Expression
   Expression "/" Expression → Expression
   Expression "%" Expression → Expression} >
  {left: Expression Plus Expression → Expression
   Expression Minus Expression → Expression} >
  Expression Expression-shift-operator Expression → Expression >
  {Expression Expression-relational-operator Expression → Expression
   Expression "is" Built-in-type → Expression} >
  Expression Expression-equality-operator Expression → Expression >
  Expression Ampersand Expression → Expression >
  Expression "^" Expression → Expression >
  Expression Bar Expression → Expression >
  Expression "&&" Expression → Expression >
  Expression "||" Expression → Expression >
  Expression "?" Expression ":" Expression → Expression
```

Listing 3.5: The priorities for the C# grammar externally specified by SDF

`primary-no-array-creation-expression` evaluates then to `literal` and similarly simple things, as well as to postfix expressions (post-increment, member access, object creation). Almost 20 steps separate a literal from an expression with a lone literal in it, as shown in Figure 3.1 for C# 1.0 (without null-coalescing-expression). A part of a parse tree corresponding to the parse path from `2` to `expression` is cut from a graph automatically generated by the ASF+SDF Meta-Environment: every box on it corresponds to a grammar production in SDF, and the right hand side always matches with the left hand side of the next rule (in the box immediately above current).

As it has been shown, this approach of introducing expressions leads to very cluttered parse trees even for the simplest expressions, because the parser has to find its way through various kinds of expressions. This hinders human understanding and decreases productivity when working with parse trees (looking for ambiguities or otherwise). SDF [86], as many other notations used for parser generation, has other mechanisms for dealing with this problem: for example, introducing priorities and specifying associativity. These methods are much easier to understand for a human reader, as they make a grammar smaller and are fully supported by parsing technology.

Here is how the refactored piece of the same functionality looks:

```
expression:
  expression "?" expression ":" expression
  expression "||" expression
  expression "&&" expression
  expression bar expression
  expression "^" expression
  expression ampersand expression
  expression expression-equality-operator expression
  expression expression-relational-operator expression
  expression "is" built-in-type
  expression expression-shift-operator expression
  expression plus expression
  expression minus expression
  expression "*" expression
  expression "/" expression
  expression "%" expression
  unary-expression
  unary-expression assignment-operator expression
```

All newly-introduced nonterminal symbols have straightforward definition as collections of terminals: e.g., `expression-shift-operator` is a choice between "<<" and ">>".

The priorities are specified on a lower level in a separate SDF file, see Listing 3.5.

3.2.7 Step 7: Resolving conflicts

A **conflict** is a situation when two grammar knowledge pieces differ when they are supposed to agree. This concept will be elaborated and broadened into the notion of **structural difference** in section 4.7 in the scope of grammar convergence. For now we limit ourselves to singular mismatches between identical grammars.

One example of a conflict will be shown here. §22.1 Delegate declarations on page 297 (lines 15–16 in Microsoft version [184]) says (in this case new line does not mean choice):

```
delegate-declaration:
  attributes?  delegate-modifiers?  "delegate"
     return-type identifier
     "(" formal-parameter-list?  ")" ";"
```

However, Appendix A.2.11 Delegates on page 357 (lines 34–35), which actually contains the formal grammar, looks like this:

```
delegate-declaration:
  attributes?  delegate-modifiers?  "delegate"
     type identifier
     "(" formal-parameter-list?  ")" ";"
```

The difference is that `type` is a real type, while `return-type` is either a type or `"void"`. After encountering this inconsistency, other sources were examined and it was found out that the Microsoft compiler does allow delegates to be void; and even source code examples from the specification show void delegates [114, 184, 225]. So it was

decided that the formal grammar in the appendix is wrong and has to be corrected towards §22.1[2].

We will return to the issue of resolving grammar conflicts and surfacing the relations between different grammars with introduction of the method of grammar convergence in chapter 4.

3.2.8 Step 8: Improving the grammar

Let us look at these definitions from the C# grammar:

```
method-body:
  block
  ;
```

```
accessor-body:
  block
  ;
```

```
operator-body:
  block
  ;
```

```
constructor-body:
  block
  ;
```

```
static-constructor-body:
  block
  ;
```

```
destructor-body:
  block
  ;
```

We can see strong similarities there. One can assume these clones were needed for explaining semantics and for potential growth of the grammar. However, semantics can handle terms like "`body` of a method", and not all grammar users aim at grammar extensions distinguishing between constructor and destructor bodies. There are no other bodies in the grammar, so one can certainly simplify the matter by introducing a sort `body` and using it everywhere throughout the grammar. This will not just shrink future parse trees, but will also improve human intelligibility of the grammar.

3.2.9 Step 9: Generating the parser

Around 1500 lines of FST code (217 transformation rules) plus 274 lines of pure SDF with lexical definitions and priorities were needed to eliminate all irregularities, introduce absent definitions, disambiguate equivocal formulæ (by reformulating them), de-yaccify and de-layer expressions and beautify the grammar. The C# grammar was cut down from 281 nonterminals and 710 BNF formulæ to about 172 sorts and 466 LLL rules. Not only it got smaller, but also significantly gained in readability without losing in declarative power.

[2] After discovering this bug and reporting it on a conference [257] it has been fixed by Microsoft and in the new version of C# language specification §22.1 and A.2.11 give identical definitions of `delegate-declaration` [115].

An SGLR [26, 169, 213, 238] parser was generated by the ASF+SDF Meta-Environment [22, 23, 140]. The grammar was not thoroughly tested, only around 100000 lines of C# code were successfully parsed with it. This level 3 grammar was made publicly available at `http://www.cs.vu.nl/grammarware/browsable/CSharp` [241].

3.3 Proposed solution generalisation and evaluation

The stepwise plan for semi-automated grammar recovery presented below is a refinement of prior work [163, 164, 219] with additional experience gained from the C# case study [257]. The grammar recovery method described in this section consists of ten steps. It takes the raw grammar presented in a language definition as an input, and its outcome is production of grammarware. Most of the steps are semi-automated, which means there can be a significant improvement by using the right technology, but it still must be operated by a language engineer. The evaluation of the method is located at the end of the section.

Obtaining the standard

The starting point for the whole recovery process is getting the relevant standard. It is important to decide which particular *dialect* of the language is needed. On a theoretical level, we can operate with a notion of a language in general (e.g., COBOL), but in practice, source code is intended for a certain compiler, which set of rules can differ from the books and standards. For example, by "COBOL" one can mean any vendor specific dialect (like AcuCobol [1], DEC COBOL [46], IBM OS/VS COBOL, IBM VS COBOL II [164], IBM SAA COBOL, COBOL/370, Micro Focus COBOL [182], Object COBOL [183], Microsoft COBOL, NetCOBOL [72], RM/COBOL, Compaq COBOL [37], Unisys COBOL or COBOL.NET), or any standardised COBOL (like ANSI COBOL: COBOL'68, COBOL'74, COBOL'85; X/Open COBOL or ISO COBOL [113]), or any version thereof.

Usually some kind of documentation on a chosen dialect is available. It can take the form of a programmer's manual, a language reference, a compiler's help file, an on-line tutorial, a list of differences with another standard, etc. Almost any documentation can be used on the next step for extraction. The completeness of information contained in the chosen specification will influence the simplicity of the extractor.

Extracting the grammar

As a result of this step one would probably like to have a grammar inside one's tooling. The process of translation from the EBNF dialect of the language specification to the EBNF dialect of the tool involves mapping that can be fully automated despite some non-trivialities. For example, when going from a YACC [128] grammar to an SDF [86] grammar, one should turn the rules around (`a: b c` becomes `b c` $\rightarrow$ `a`) and drop all semantic actions, error handling, etc.

It is not necessary to use grammarware for lightweight translations. For example, "*opt*" marks can be replaced with "?" with a simple regular expression or a text processor facility. On the other hand, going from non-ASCII notation, or an ASCII representation of a visual form, to pure ASCII can require considerable manual labour or a creative approach like in [28] where a special extractor was developed to recover the grammar of IBM VS COBOL II. Another example is the ISO COBOL [113] grammar that exists in an Adobe FrameMaker form, which is a proprietary format and cannot be easily parsed; however, one can export it as a MIF document, which specification is available.

For our detailed analysis on various metasyntaxes found in different language documents, the reader is referred to section 6.3.

Fixing misprints

Obvious kinds of misprints include misspelled nonterminal symbol names, misspelled keywords (terminals), duplicate definitions, forgotten definitions which can be found in the standard but not gathered in the appendix with a grammar, or vice versa. Once spotted, all these kinds of problems are easy to fix. The very presence of such errors indicates that the grammar is hand-made; it was composed without the use of appropriate tools which can check the consistency of the content.

In section 5.4 we will see markup problems such as misplaced HTML tags, misleading indentation, mismatched parentheses, metasymbols marked up as terminals, terminals marked up as nonterminals and vice versa, nonterminal names being split or combined the wrong way, etc. Most of them can be considered "misprints" in a broad sense. In the JLS case study these misprints were fixed by a sophisticated automated extractor.

Completing a formal part

There are particular things that are easily described and understood when explained in human language, but which pose difficulties when being formalised. For example, one can state in a standard that statements are separated by semicolons, but forget to include the corresponding BNF formulae. Problems like these can usually be automatically detected from errors and warnings of the grammarware used, but tend to require human attention for fixing.

A classic example of such problems is Python [215]: its manual only notes that "indentation is Python's way of grouping statements", but it does not provide any means to deal with this issue in a grammar. Actually, it would seem rather unjustified to solve indentation problems within a grammar, and most existing Python parsers use a special preprocessor that inserts tokens which mark changes in indentation; any program preprocessed in such a way can be parsed using conventional indentation insensitive algorithms afterwards.

Modern programming languages are very liberal with comments, which may occur almost everywhere and sometimes even be nested. Obviously, a parser has to deal with these. However, specifying rules for comments informally is very easy while integrating them into the formal grammar may prove to be quite tricky. As a result,

the rules for comments are often omitted from the language documents. To be more precise, they can be explained only in an informal descriptive manner and never show up in the grammar itself. This can lead not only to the need of re-expressing it formally, but also to a possibility of introducing mistakes by erroneously interpreting the text.

In many cases another disjunction mechanism is used: "one of the following" ($\bigvee_k x_k$ in mathematics) instead of "x or y" expected by BNF ($x_1 \vee x_2 \vee \cdots$). They are equivalent in the case of limited number of alternatives, modulo the concrete syntax (infix notation vs. prefix notation).

Other standards can also be referenced, and it is done sometimes in textual form without any verification or formalisation attempts. For example, most books on C++ programming language tell us that it is an extension for C programming language, yet the work of standardisation subcommittees for C [104] and C++ [108] is separated for years, and there were no known attempts recently to validate that statement. In general, most references to other standards should be rewritten formally either to use those standards directly or to incorporate their needed content, or removed at all.

It is easy to check if all the nonterminals have been defined in the grammar. We construct a directed graph, where each nonterminal symbol is a node. An edge goes from one node to another if the corresponding target nonterminal is used in the definition of the corresponding starting nonterminal. If the resulting graph is connected—that is, all the nodes are reachable from the top node—then there are no missing rules. Any node with no incoming edges means a nonterminal that was never used in any other definitions. These unexpected top nodes notify us about missing definitions.

Relaxation

One of the most difficult things in *composing* a formal grammar is knowing where to stop. Usually pulling too much semantic issues into the grammar makes it unnecessarily overcomplicated. However, small bits of semantics always find their place in a grammar. Some people might want to use them; others do not need them. In general, it is natural that the grammar given in a text documenting a language is not the grammar one would like when parsing.

Other grammar parts that will not be necessary for the tool under development may also be removed at this step. In order to make the parser more tolerant and cover a range of language dialects, one can decide to remove significant and considerably big parts of the baseline grammar.

At this step the grammar should be operational, it should be possible to parse source code with it. This step is semi-automatic in the sense that it requires a language engineer to choose and execute the transformations. The next steps help to improve the grammar and to locate any problems it might still have.

Removing idiosyncrasies

Grammar class and parsing technology limitations like preference or ban on right or left recursion should not play a role in grammar comprehension, evolution and maintenance [27]. Finding out the use patterns that are peculiar to the notation or the tool used by the language document creators, and refactoring them out happens on this step.

Resolving conflicts

Even before starting the grammar recovery process, one should decide on priorities of the sources. For example, if the formal grammar states something, while the complementary text says something else, what should be taken as a "correct" variant? Should it be concluded that the text is written by someone who misunderstood the grammar, or should it be assumed that the grammar's composer misinterpreted the text? In fact, it is not uncommon to have more than one "standard" of the same status, C# 1.0 had three of them: [114, 184, 225]. There is often at least one compiler, which represents the "real" language as it is used by programmers ("natives" of the language). In the worst case one would have diverse compilers, their manuals, a whole bookshelf of reference books and several standards. Apparently, one should prioritise and decide which of the grammar knowledge sources takes precedence in case of inconsistencies.

Improving the grammar

After all the steps described above one will have a perfectly working grammar. It can still have details that ask for removal or refactoring, like the ones described in subsection 3.2.8 for C#. However, the grammar is already functional at this point, and further modifications will be *improvements*, as opposed to *corrections* or removing idiosyncrasy. Modularisation, beautification, renovation and similar activities can be performed after this step. Rules of grammarware engineering do exist and do indeed make life easier [141, 154]. Since some of them are questions of taste, this issue will not be pursued here.

Re-establishing links with the specification text

It is important for the whole process of standardisation to keep track of the relation between the grammar being transformed and other parts of the language definition: explanatory text, code examples, visualisations, outside and inside references, etc. In order to do so consistently, it is important to re-establish all the links that were in the original standard, but now with a corrected grammar. If it is not done at this point, later the relations will tend to fade and become less transparent, which can yield two entities—a specification and a grammar—that need to be maintained separately.

This process cannot be fully automated due to important decisions that have to be made by a grammar engineer, but it can certainly be semi-automated, when actions are triggered by a human expert and performed by a tool.

Generating grammarware

Having all these steps carried out, one can generate a parser and use it for transformations. Different parsing techniques (see section 2.8) can be used at this stage [2],

and even "yaccification"-like activities may be performed in order to optimise the grammar with a particular parser generator in mind. The presence of a general base-line grammar has its advantages here as well: switching to a specific technology is easier than refactoring something that was already optimised for another technology. Hence, no vendor lock-in for one parsing technology.

Browsable hyperlinked versions of a grammar [241] can also be automatically generated. The simplest facilities are already included in tools like GDK [149], but it is possible to generate the manual or specification in the way we will be ready to discuss in chapter 6.

3.4 Conclusion

This research experiment shows that grammar recovery techniques are needed with new languages like C# as sharp as with the old ones like COBOL or PL/I. The quality of both the language documentation as such and the grammar available as a part of it, was not found satisfactory. However, grammars taken directly from language documentation composed with "grammar hacking" can be re-engineered into a correct grammar and a working parser. This process can be partially automated. Conversely, language specification in the form of a manual or an hyperlinked on-line documentation can be generated from a grammar. The issue is pursued further in chapter 6, where a prototype is also developed.

As compared to [164] as one of the predecessors of this experiment, neither the C# case study nor our general plan involve test-driven correction. Instead we strive for an approach similar to differential testing [178] rather than conventional one: the approach will be much better demonstrated and explained in the next chapter when we will transform several grammars simultaneously. Also, we encountered less connectivity problems and more disambiguation and priority-related problems. To summarise, this case study can be considered an application of the same methodology to another language specification in order to prove that the original proposal was not specific to Cobol or to older programming languages, and to find its weak spots to encourage the next steps of research (see subsection 3.4.2).

3.4.1 Discussion on the method automation

Step 1 (obtaining the standard) cannot be automated, it must be accomplished manually, usually by finding the right document and preparing the grammar-related part of its content for extraction. The particular metasyntax utilised by the standard must also be obtained, usually informally defined on defined in itself. The language document may also be found to contain unparsable fragments: e.g., switching between EBNF and English.

Step 2 (extracting the grammar) difficulty depends on a difference between the grammar definition formalisms encountered in the standard and used by the tooling. There are no known metrics for measuring that distance. If the standard found on the previous step contained a formal grammar, step 2 can be automated. Otherwise, step 2 is not really about extracting the grammar anymore, but rather about engineering it. Successful

outcome of this step is a lexically correct grammar. The automation of this step is repeatable for newer versions of the same base-line specification, but not generalisable due to diversity of notations for syntactic definitions.

Steps 3 (fixing misprints) and **4** (completing a formal part) cannot be automated, but having good tools helps to spot the problems faster. The corresponding section explained some of the checks that can help, like constructing a connected graph of nonterminal symbols. In general, the activities that are performed during step 4 are similar to the last stages of grammar engineering. The result of step 4 is a syntactically correct grammar that is accepted by the chosen tooling—not necessary executable.

Step 5 (relaxation) is semi-automated: the grammarware technology gives a language engineer the tools to do the relaxation, but does not show immediately which parts of a grammar can be cut out. The relaxation step is optional, it is not uncommon to want to keep the whole grammar as it was found in the standard. If the step is skipped, it may lead to ambiguities and conflicts later on.

Steps 6 (de-yaccification) and **8** (grammar improvement) are semi-automated as well. It is advised to add a test set as early as step 6, even if it was impossible to reach at step 4. Consistently running increasingly thorough tests after every enhancement justifies the refactorings and assures they did not damage the grammar. Step 8 is optional, and its objectives may vary depending on the language engineer's point of view on what is and what is not a good grammar.

Step 7 (resolving conflicts between a grammar and a text) is purely manual, since its nature lies in spotting differences between a grammar at hand and the original specification or its alternatives. It takes place only when such differences are occasionally found out, or as a part of disambiguation process.

Step 9 (re-establishing links with the text of a spec) is important in the scope of language specification life cycle. If the grammar is only recovered to run one-time checks or to enable code migration—in other words, if its value as an IT asset is low or declining, this step is not worth the trouble. However, for an actual specification it can enable all kinds of positive issues discussed in chapter 6.

Step 10 (grammarware generation) is automated. Various grammarware factories, compiler compilers, formatters and generators can create tools out of a working grammar. Developing a new tool either based on a grammar or for generation itself, is much easier, faster and more efficient with a correct grammar at hand than with a language manual.

3.4.2 Research objectives revisited

We have referred to this case study as an "experiment" for a reason. At the beginning of this project we did not know how much effort would it take to recover an operational grammar from the ECMA C# specification. At the end, we did not only get the grammar, but also realised a couple of important points that became the foundation for the next research steps.

Grammar extraction was observed to be very sensible to the notation used in the language documentation. Thus, it should be automated in the most generalised form possible. Even if it is impossible to implement a unified grammar extractor, as proven by demonstration of grammar definition formalisms differences in section 6.3, we can

still design it carefully each time to cover the widest range of input grammar artefacts. To avoid manual steps like copy-pasting completely, we need to consider the lexical details of the language documents and other sources of grammar knowledge, as will be done in section 5.4.

Grammar transformations are needed to complete the process of grammar recovery after the grammar has been extracted. The facilities provided by the existing framework we used (GDK and the Meta-Environment) are not as fine-grained as we would wish and not nearly as accurate. GDK basically allows us to *redefine* grammar productions rather than *transform* them. As a result, for the next case studies and research, we relied on more advanced publications on grammar transformation suites such as [154] and gradually developed our own grammar transformation language called XBGF, as seen in section 4.8, section 5.6 and ultimately in chapter 7.

Grammar comparison is required in order to test what kind of differences two grammars possess. For instance, in this experiment we did not have any automated tools to validate the statement that ISO/IEC 23270:2003 C# Specification [114] defines exactly the same language as ECMA-334 Standard [225], as they claim. Other reasons for having two or more grammars can be language evolution or vendor-specific dialects (section 2.4), conflicting documentation (subsection 3.2.7), tool-specificity in "the grammar net" [164] or just independent design and development of idiosyncratic grammars (section 4.1). By answering this research question thoroughly, we arrived at the notion of **grammar convergence** which will be presented in detail in the next chapter.

Language documentation was shown to be imperfect, it contained mistypings, inconsistencies, undefined nonterminals, incorrect productions. If we aim at using its contents literally, we also need to account for that, and to research the possibilities to improve the situation. Since we wanted to analyse a range of language documents in order to understand better the process of recovering grammars from them and automating that process, we also reverse engineered the general structure those language documents had in common. The resulting model and other conclusions drawn from that effort will be presented in section 6.4 and adjacent sections.

Chapter 4

Language convergence

Grammar convergence is a lightweight verification method for establishing and maintaining the correspondence between grammar knowledge ingrained into different kinds of software artefacts. The central idea of grammar convergence is to extract grammar knowledge, and to use programmable grammar transformations [42, 154] as means to constructively prove the convergence of different sources to a shared limit. The transformations model refactoring and more liberal editing operations. For instance, when doing grammar convergence for different parser descriptions, then one would need to neutralise the specifics of any given parsing technology as well as accidental differences between parser descriptions. The method of grammar convergence entails the following core ingredients:

1. All kinds of software artefacts vastly differ in terms of the notation, idiosyncrasies and capabilities used internally to represent (E)BNF-like grammar knowledge. A unified *grammar format* is needed to cover the various use cases. We define such a format: BGF—BNF-like Grammar Format.

2. For any kind of software artefact of interest, there should be a corresponding *grammar extractor* which can strip off the grammar knowledge from a given artefact. Extraction has already been discussed in the previous chapter.

3. If there are two or more grammars which are intended to "agree upon each other", a *grammar comparer* can check for trivial syntactical equivalence and report differences in a way to provide helpful data for the manual (or possibly automated) proof of correspondence.

4. When grammars from different sources differ (more or less profoundly), then programmable *grammar transformations* have to constructively establish the degree of correspondence.

Ingredients 1.-4. are enough to constructively prove that grammars extracted from different sources converge. Note that the *use* of the grammar knowledge within the software artefact is not checked in any way: for some kinds of software artefacts, one can elaborate

grammar convergence to operate at the "instance level" (e.g., parse trees). This sophistication is beyond the scope of this thesis.

4.1 Motivation

Here are few use cases for grammar convergence:

- ◇ There is a handcrafted object model (say, Java classes) for a specific domain, be it financial exchange, while there is also an XML schema [75, 208] that is meant to standardise that domain. It is valuable to establish (the degree of) agreement between the object model and the schema.
- ◇ Given is a compiler for a mainstream and nontrivial programming language such as gcc for C++. Given is also a reverse or re- engineering tool for the same language based on a different parsing infrastructure. How can we establish that both front ends indeed agree on (the grammar of) the language at hand?
- ◇ Given is a language documentation that contains a declarative, perhaps simplified grammar of a language, say Java. This grammar is not really meant to be executable. Given is also a reference parser for the language; its grammar appeals to the specifics of a particular parsing technology. How can we establish that the documentation (such as [77]) is in sync with the reference implementation?
- ◇ A developer of an XML-data binding technology wants to design test cases for a mapping from XML schemas to object models [161]. The mapping may also provide customisation capabilities to users. How could the generated object models be usefully compared with the original XML schemas so that the mapping is validated?

4.2 Contributions

- ◇ This is the first relatively general attempt to verify the correspondence between scattered grammar knowledge. The particular strength of grammar convergence is that it relates ingrained grammar knowledge in diverse forms of software artefacts; it complements the use of generative (or "model-driven") approaches, when they are not used, have not been used, or cannot (yet) be used.
- ◇ Grammar convergence delivers conceptually simple grammar transformations to software artefacts other than just (E)BNFs: XML schemas, object models, and algebraic signatures. The similarity of these kinds of artefacts belongs to computer science trivia, but grammar convergence truly remains in the simple grammars space due to an effective notion of grammar extraction ("abstraction").

Parts of this chapter were published in condensed form as *An Introduction to Grammar Convergence* [166] and *Recovering Grammar Relationships for the Java Language Specification* [167, 168]. The former is an introductory paper presented at the 7th International Conference on Integrated Formal Methods. The version included in the thesis

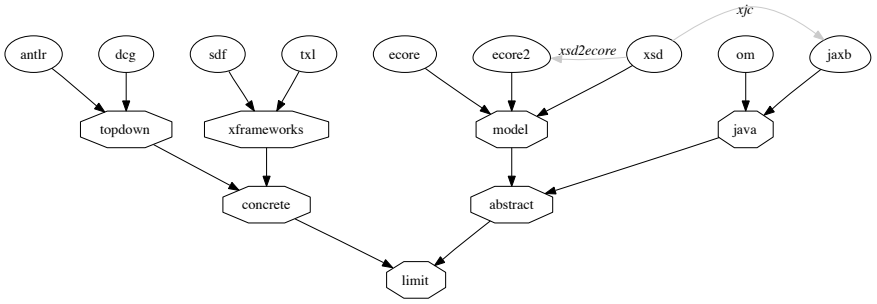


Figure 4.1: The overall convergence graph for the Factorial Language. The grey arrows show grammar relations that are expressed in LCF but not performed directly by the convergence infrastructure (the reason is that, for example, generating Ecore from XML Schema cannot be done from command line and must be performed via Eclipse IDE).

is substantially enhanced with respect to the original paper, and also includes full diagrams and listings presented in a unified notation. The notion of phases of convergence (section 4.7) has emerged later during the work on [167] for the 9th IEEE International Working Conference on Source Code Analysis and Manipulation where it received the Best Paper Award. A significantly extended version is being printed in the special issue of Software Quality Journal [168]. An implementation of grammar convergence is publicly available through the *Software Language Processing Suite* [263]. The idea of using programmable grammar transformations to converge grammars has emerged in collaboration with Prof. Dr. Ralf Lämmel (Software Languages Team, Universität Koblenz), as was the subsequent development of the prototype. All three papers mentioned above [166, 167, 168] are co-authored with him.

4.3 The domain

As a running example for convergence a trivial programming language FL is used. FL stands for "Factorial Language", it is available from the SLPS repository [263]. Six different grammars are extracted from various implementations of FL, and successfully converged. Following is an illustrative program in the FL language. It defines two functions: one for multiplication, another for the factorial function, the latter in terms of the former:

```
mult n m = if (n == 0) then 0 else (m + (mult (n − 1) m))
fac n = if (n == 0) then 1 else (mult n (fac (n − 1)))
```

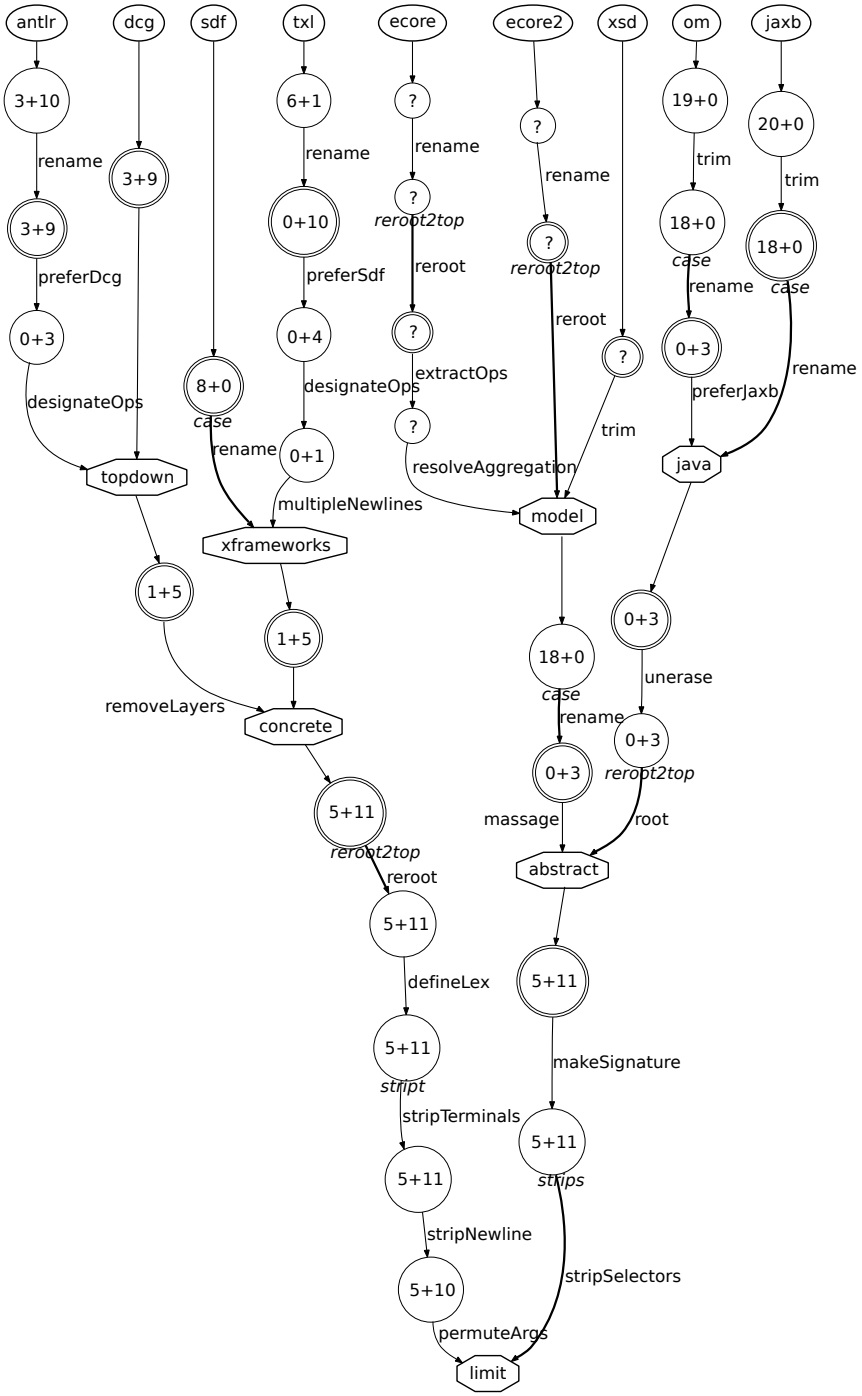


Figure 4.2: The detailed convergence graph for the Factorial Language. The numbers in each bubble are the number of nominal differences plus the number of structural differences. Edges that correspond to automated actions are bolder, with the generator's name in italics. The target *model* has been split in two in order to apply the metrics (otherwise it would be impossible to make a branch choice for synchronisation).

4.3.1 Sources of convergence

Figure 4.1 shows the sketch of a convergence tree for some of the existing FL implementations. The *leaves* of the tree (at the top of the figure) denote different *sources* for FL. We use the term source here to mean "software artefact containing grammar knowledge". Here is short description of the sources for FL:

antlr This is a parser description in the input language language of ANTLR [202]. Semantic actions (in Java) are intertwined with EBNF-like productions.

dcg This is a logic program written in the style of definite clause grammars; see Listing 4.2.

sdf This is a concrete syntax definition in the notation of SDF (Syntax Definition Formalism [86, 239]). It is parser description based on the SGLR implementation for SDF (Scannerless Generalised LR Parsing); see Listing 4.1.

txl This is another concrete syntax definition in the notation of TXL (Turing eXtender Language) transformational framework [39, 42, 43]. Unlike SDF, this framework uses a combination of pattern matching and term rewriting.

ecore This is an Ecore model [197], created manually in Eclipse [59] and represented in XMI; see Listing 4.3.

ecore2 This alternative Ecore model was automatically generated by Eclipse from the XML Schema and extracted from XMI [196].

xsd This is an XML schema [75, 208] for the abstract syntax of FL. In fact, this is the schema that served as the input for generating the object model of the *jaxb* source and the Ecore model of the *ecore2* source.

om This is a hand-crafted object model (Java classes) for the abstract syntax of FL. It is used by a Java-based implementation of an FL interpreter.

jaxb This is also an object model, but it was generated by the XML-data binding technology JAXB [126] from an XML schema for FL.

The sources are part of FL language processors, e.g., interpreters and optimisers.

4.3.2 Targets of convergence

Consider again Figure 4.1. The *inner nodes and the root* denote targets of the convergence process. These are grammars that are derived by transformation with the sole purpose of establishing grammar equality. There are the following targets:

topdown The sources *antlr* and *dcg* both involve top-down parsing. Their correspondence can be established by a few simple refactoring steps.

concrete This target converges all concrete syntax definitions. A noteworthy difference is that *sdf* uses one expression nonterminal, whereas *topdown* uses two "layers".

```
Function+                          → Program
Name Name+ "=" Expr Newline+       → Function
Expr Ops Expr                      → Expr  {left,prefer,cons(binary)}
Name Expr+                         → Expr  {avoid,cons(apply)}
"if" Expr "then" Expr "else" Expr  → Expr  {cons(ifThenElse)}
"(" Expr ")"                       → Expr  {bracket}
Name                               → Expr  {cons(argument)}
Int                                → Expr  {cons(literal)}
"−"                                → Ops   {cons(minus)}
"+"                                → Ops   {cons(plus)}
"=="                               → Ops   {cons(equal)}
```

Listing 4.1: SDF grammar for FL. Only (context-free) SDF productions are shown. SDF productions provide an alternative for the nonterminal on the right-hand side of the arrow. Productions can be annotated in certain ways between the braces, e.g., with constructor names (such as **cons**), or directions for disambiguation (such as **prefer** or **avoid**).

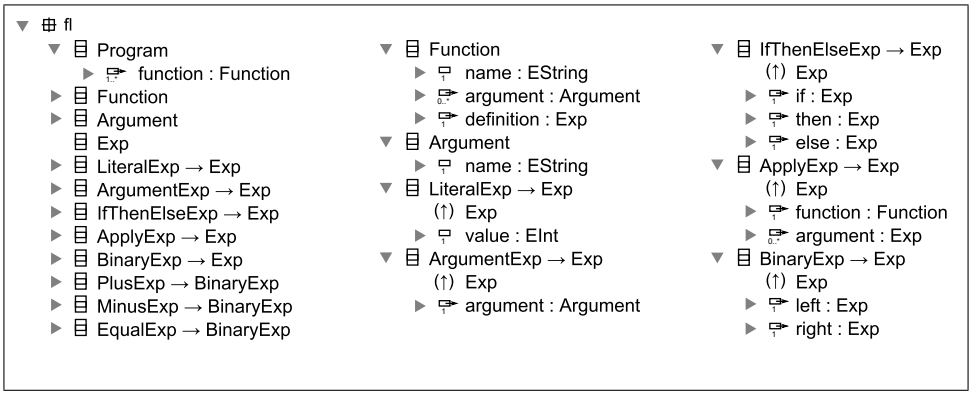
```
program(Fs) → +(function,Fs).
function(N,Ns,E)) → name(N), +(name,Ns), @("="), expr(E), +(newline).

expr(E)                          → lassoc(ops,atom,binary,E).
expr(apply(N,Es))                → name(N), +(atom,Es).
expr(ifThenElse(E1,E2,E3))       →
  reserved("if"), expr(E1), reserved("then"), expr(E2), reserved("else"), expr(E3).

atom(literal(I))      → int(I).
atom(argument(N))     → name(N).
atom(E)               → @("("), expr(E), @(")").

ops(equal) → @("==").
ops(plus)  → @("+").
ops(minus) → @("−").
```

Listing 4.2: Definite Clause Grammar for FL. The clauses construct a term representation; see the arguments of the various predicates. The DCG leverages higher-order predicates for EBNF-like expressiveness and left-associative tree construction (i.e., "+" and "lassoc"). The priorities on expression forms are expressed by means of a layered definition; i.e., "expr" vs. "atom".

java The sources *om* and *jaxb* are both object models whose correspondence can be established by refactoring.

abstract Eventually, the two object models and the XML schema (from which one of the object models was generated) can be converged to an abstract syntax definition. There are differences between object models and XSD because these are fundamentally different type systems [161].

limit All FL grammars are converged to one final target, which can be seen as the least



Listing 4.3: Ecore model for FL. It is particularly interesting to look at how binary expressions are represented here. In the world of EMF/MOF/UML modelling it is possible to leave the binary expression itself (*BinaryExp*) an abstract class and inherit its properties such as *left* and *right* with the subclasses *PlusExp, MinusExp* and *EqualExp*. The extractor takes good care of abstract classes, but the resulting grammar structure is still significantly different from the use of terminal symbols in concrete syntax-oriented grammars and also from the use of labels and selectors in other abstract syntax-oriented sources.

upper bound of all instances of ingrained grammar knowledge. This step involves the removal of terminals from *concrete*, and the removal of selectors from *abstract*.

4.3.3 BGF — BNF-like Grammar Format

In principle, we could base our work upon existing syntax definition formalisms (e.g., SDF [86, 239]) or metamodelling facilities (e.g., MOF [197] or EMF [59]). In contrast, we derive the grammar format BGF in a way that it is limited to the representation of grammar knowledge that we consider relevant and manageable across different kinds of software artefacts. Despite this rationale, there is still a trade-off to be made between simplicity of format and precision of extraction. At this point, we lean towards simplicity. The use of a new format also allows us to define grammar convergence without interfering with any existing operational semantics. For convenience, we can still represent BGF in other notations (using a generative approach as in [149]).

We start with the most trivial aspects for BGF:

- ◇ Terminals and nonterminals.
- ◇ Regular expression-like composition and grouping:
  - sequential composition (also called “sequence”).
  - alternative composition (also called “choice”).
  - *true* — also known as epsilon (ε).
  - *fail* — also known as “empty language”.

  - $\star$, $+$, ? — usual EBNF [252].

- ◇ A production is a pair of "defined nonterminal" and "defining expression".
- ◇ A grammar is a collection of productions.
- ◇ A grammar may also designate start symbols.

At this point, we have reached "representation capability" for textbook-style BNF and EBNF (when restricted to context-free syntax). Only few more concepts are needed to represent essential extras of XML schemata, object models, and algebraic signatures:

Production labels Concrete manifestations are names of subclasses, term constructors of an abstract syntax associated with a concrete one, or derived types or substitution-group members in an XML schema. Extraction should not abstract away such important nominal grammar knowledge. As a bonus, labels are convenient in addressing productions in programmable grammar transformations.

Expression selectors Object models use flat record-like grammar structure with selectors at the top level. However, local element declarations of XML Schema as well as component selectors of richer algebraic structuring formats (such as Haskell) dictate that selectors must be admitted more liberally.

Simple types In a context-free grammar, the terminals and the "morpheme classes" are the leaves of (context-free) grammatical structure. In an XML schema, an object model, or an applied functional language with algebraic data types, "simple" types are used at the leaves. Even in a parser description, morpheme classes may be associated with simple types (such as string and int).

Universal type There are situations when no precise grammatical structure can be extracted, but some record of the relevant component should be preserved in the extraction results. Examples include wildcards of XML Schema, or dynamics in functional languages, as well as "object" as the base type in OO languages.

Namespaces Software artefacts subject to extraction may use a namespace mechanism, or a module system: object models of Java are organised in packages; Haskell types are organised in a hierarchical module systems; XML schemata are organised in XML namespaces. Namespaces may need to be preserved in extracts.

Listing 4.4 defines the grammar of BGF in BGF. Based on an self-application of grammar convergence, starting from an XML schema of BGF, subject to recorded refactoring transformations, this self-representation has been *computed*. For more detailed presentation and explanation as well as for converging pretty-printed concrete syntax with the abstract data model please refer to section 7.4.

```
grammar:
      root::nonterminal* production*
production:
      label::label? nonterminal::nonterminal expression
expression:
      epsilon::ε
      empty::ε
      value::value
      any::ε
      terminal::terminal
      nonterminal::nonterminal
      selectable::(selector::selector expression)
      sequence::(expression+)
      marked::expression
      choice::(expression+)
      optional::expression
      plus::expression
      star::expression
value:
      int::ε
      string::ε
label:
      STR
nonterminal:
      STR
selector:
      STR
terminal:
      STR
```

Listing 4.4: BGF — BNF-like Grammar Format (The shown extent does not cover namespaces, and its simple type system is limited to integers and strings.)

4.4 Grammar extraction

4.4.1 Abstraction by extraction

The purposely restricted expressiveness of BGF implies an abstraction along grammar extraction. In addition, an extractor may purposely perform abstraction. (For instance, an extractor may merge namespaces of the source.) Arguably, abstraction simplifies proofs of grammar convergence. Here are examples of "implied" abstraction grouped by kinds of grammarware:

(E)BNF none.

Parser descriptions

- ◇ Semantic actions.
- ◇ Lexical syntax descriptions.

- ◇ Technology-specific idioms (e.g., precedence declarations).

Object models

- ◇ Constructors, static methods, initialisers
- ◇ Specific types of collection classes
- ◇ Classes vs. interfaces dichotomy
- ◇ Fields vs. methods dichotomy

Algebraic data types

- ◇ Nominal types vs. aliases (e.g., Haskell, SML, etc.)
- ◇ New types vs. constructor types (e.g., Haskell)
- ◇ Higher-order and quantified types (represented universally)

XML schemata

- ◇ Elements vs. attributes
- ◇ Elements vs. named complex types vs. groups
- ◇ Simple type system of XML Schema

4.4.2 Grammar extractors

An extractor is simply a software component that processes a software artefact and produces a (BGF) grammar. In the simplest case, extraction boils down to a straightforward mapping defined by a single pass over the input. Extractors are typically implemented within the computational framework of the kind of source, or in its affinity:

Kind of artefact	Computational framework
DCG	Prolog
SDF	ASF+SDF [22] or Stratego [240]
TXL	TXL [39] and XSLT [137]
ANTLR	ANTLR [202]
Ecore (XMI)	XSLT [137]
Java	Java compile- or run-time reflection
Haskell	One of several metaprogramming techniques for Haskell

On the output side, an extractor typically leverages the XML format for BGF.

Figures 4.5–4.12 contrast the extraction results obtained from different FL parsers and an XML schema for FL. The differences between the grammars can be summarised as follows:

- ◇ Only the SDF+DCG+ANTLR extracts contain terminals.
- ◇ The DCG+ANTLR extracts contain expressions layers expr and atom.
- ◇ The SDF+XSD extracts contain a single expression layer.

```
program:
      function+
function:
      ID ID+ "=" expr NEWLINE+
expr:
      binary
      apply
      ifThenElse
binary:
      atom (ops atom)*
apply:
      ID atom+
ifThenElse:
      "if" expr "then" expr "else" expr
atom:
      ID
      INT
      "(" expr ")"
ops:
      "=="
      "+"
      "-"
```

Listing 4.5: BGF extracted from an ANTLR front end for FL

```
program:
      function+
function:
      name name+ "=" expr newline+
[binary] expr:
      atom (ops atom)*
[apply] expr:
      name atom+
[ifThenElse] expr:
      "if" expr "then" expr "else" expr
[literal] atom:
      int
[argument] atom:
      name
atom:
      "(" expr ")"
[equal] ops:
      "=="
[plus] ops:
      "+"
[minus] ops:
      "-"
```

Listing 4.6: BGF extracted from the DCG for FL as of Listing 4.2.

```
Program:
      Function+
Function:
      Name Name+ "=" Expr Newline+
[binary] Expr:
      Expr Ops Expr
[apply] Expr:
      Name Expr+
[ifThenElse] Expr:
      "if" Expr "then" Expr "else" Expr
Expr:
      "(" Expr ")"
[argument] Expr:
      Name
[literal] Expr:
      Int
[minus] Ops:
      "-"
[plus] Ops:
      "+"
[equal] Ops:
      "=="
```

Listing 4.7: BGF extracted from the SDF for FL as of Listing 4.1.

- ◇ Only the XSD extract contains selectors.
- ◇ Only the XSD extract use choices.
- ◇ The XSD extract only uses definitions of a single production.
- ◇ There is also semantic variation on using production labels.
- ◇ More trivially, the grammars disagree on names, upper and lower case.

4.5 Grammar comparison

The grammar comparator is used to discover grammar differences, and thereby, to help with drafting transformations in a stepwise manner. We distinguish nominal vs. structural grammar differences. We face a *nominal difference* when a nonterminal is defined or referenced in one of the grammars but not in the other. We face a *structural difference* when the definitions of a shared nonterminal differ for two given grammars. Some of the nominal differences will be eliminated by a simple renaming, while others will disappear gradually when dealing with structural differences that involve folding/unfolding.

In order to give better results, we assume that grammar comparison operates on a slightly normalized grammar format. The assumed, straightforward normalisation rules are simple. If (x, y) represents sequential composition of symbols x and y, $(x; y)$ represents a choice with x and y as alternatives and $fail$ represents the "empty language", then the following formulæ are used for normalising grammars within our framework:

read2 [78, §8.1]

```
ClassDeclaration:
      ClassModifiers? "class" Identifier Super? Interfaces?
            ClassBody
```

read3 [79, §8.1, §8.9]

```
ClassDeclaration:
      NormalClassDeclaration
      EnumDeclaration
NormalClassDeclaration:
      ClassModifiers? "class" Identifier TypeParameters? Super?
            Interfaces? ClassBody
EnumDeclaration:
      ClassModifiers? "enum" Identifier Interfaces? EnumBody
```

Figure 4.3: Two similar grammar excerpts from different versions of the JLS. The second excerpt involves two more nonterminals than the first excerpt: `NormalClassDeclaration`, which looks similar to the nonterminal from the first grammar, and `EnumDeclaration`, which is completely new. Hence, we speak of two nominal differences (two nonterminals in *read3* that do not match *read2*), and of two structural differences (two unmatched branches in *ClassDeclaration*).

```
Program:
      function::Function+
Function:
      name::STR argument::Argument+ definition::Exp
Argument:
      name::STR
Exp:
      LiteralExp | ArgumentExp | IfThenElseExp | ApplyExp | BinaryExp
LiteralExp:
      value::INT
ArgumentExp:
      argument::Argument
IfThenElseExp:
      if::Exp then::Exp else::Exp
ApplyExp:
      function::Function argument::Exp+
BinaryExp:
      PlusExp | MinusExp | EqualExp
PlusExp:
      left::Exp right::Exp
MinusExp:
      left::Exp right::Exp
EqualExp:
      left::Exp right::Exp
```

Listing 4.8: BGF extracted from an Ecore model for FL as of Listing 4.3

```
Apply:
      name::STR arg::Expr+
Argument:
      name::STR
Binary:
      ops::Ops left::Expr right::Expr
Expr:
      Apply | Argument | Binary | IfThenElse | Literal
Function:
      name::STR arg::STR+ rhs::Expr
IfThenElse:
      ifExpr::Expr thenExpr::Expr elseExpr::Expr
Literal:
      info::INT
Ops:
      Equal::ε
      Plus::ε
      Minus::ε
ProgramType:
      function::Function+
```

Listing 4.9: BGF extracted from an Ecore model generated from an XML schema for FL

```
Program:
      function::Function+
Fragment:
      Expr
Function:
      name::STR arg::STR+ rhs::Expr
Expr:
      Literal | Argument | Binary | IfThenElse | Apply
Literal:
      info::INT
Argument:
      name::STR
Binary:
      ops::Ops left::Expr right::Expr
Ops:
      Equal::ε
      Plus::ε
      Minus::ε
IfThenElse:
      ifExpr::Expr thenExpr::Expr elseExpr::Expr
Apply:
      name::STR arg::Expr+
```

Listing 4.10: BGF extracted from an XML schema for FL

```
Apply:
      name::STR args::(ANY*)
Argument:
      name::STR
Binary:
      ops::Ops left::Expr right::Expr
Expr:
      Apply | Argument | Binary | IfThenElse | Literal
Function:
      name::STR args::(ANY*) rhs::Expr
IfThenElse:
      ifExpr::Expr thenExpr::Expr elseExpr::Expr
Literal:
      info::INT
Ops:
      Equal::ε
      Plus::ε
      Minus::ε
Program:
      functions::(ANY*)
Visitor:
      EMPTY
```

Listing 4.11: BGF extracted from a manually created Java object model

```
Apply:
      Name::STR Arg::(ANY*)
Argument:
      Name::STR
Binary:
      Ops::Ops Left::Expr Right::Expr
Expr:
      Apply
      Argument
      Binary
      IfThenElse
      Literal
Function:
      Name::STR Arg::(ANY*) Rhs::Expr
IfThenElse:
      IfExpr::Expr ThenExpr::Expr ElseExpr::Expr
Literal:
      Info::INT
ObjectFactory:
      ε
Ops:
      EQUAL::ε
      PLUS::ε
      MINUS::ε
package-info:
      EMPTY
Program:
      Function::(ANY*)
```

Listing 4.12: BGF extracted from a Java object model generated by JAXB [126].

$$
\begin{array}{rr}
(,) \Rightarrow \varepsilon & (;) \Rightarrow fail \\
(\ldots,(x,\ldots,z),\ldots) \Rightarrow (\ldots,x,\ldots,z,\ldots) & (x,) \Rightarrow x \\
(\ldots,x,\varepsilon,z,\ldots) \Rightarrow (\ldots,x,z,\ldots) & (x;) \Rightarrow x \\
(\ldots;(x;\ldots;z);\ldots) \Rightarrow (\ldots;x;\ldots;z;\ldots) & \varepsilon^{+} \Rightarrow \varepsilon \\
(\ldots;x;fail;z;\ldots) \Rightarrow (\ldots;x;z;\ldots) & \varepsilon^{\star} \Rightarrow \varepsilon \\
(\ldots;x;\ldots;x;z;\ldots) \Rightarrow (\ldots;x;\ldots;z;\ldots) & \varepsilon? \Rightarrow \varepsilon
\end{array}
$$

Let us borrow a simple example from the case study that will be presented in complete detail in the next chapter. Without paying attention to the specific grammar notation and transformation operators, we will observe a typical scenario of using a grammar comparator. Consider two grammar excerpts from the grammars of JLS2 and JLS3 (*read2* and *read3* from now on) in Figure 4.3. Conceptually, the grammars are different in the following manner. The *read3* grammar covers additional syntax for enumeration declarations; it also uses an auxiliary nonterminal `NormalClassDeclaration` for the class-declaration syntax that is declared directly by `ClassDeclaration` in the *read2*

grammar. The comparator reports four differences that are directly related to these observations:

- ◇ Nominal differences:
  - read2: nonterminal `NormalClassDeclaration` missing.
  - read2: nonterminal `EnumDeclaration` missing.
- ◇ Structural differences:
  - Nonterminal `ClassDeclaration`: no matching alternatives (counts as 2 because the definitions have a maximum of 2 alternatives).

Arguably, these differences should help the grammar engineer who will typically try to find definitions for missing nonterminals by extracting their inlined counterparts. The counterpart for `NormalClassDeclaration` is relatively obvious because of the combination of a nonterminal that is entirely missing on one side while it occurs in a structural different and unmatched alternative on the other side.

4.6 Grammar transformation

Since the goal of grammar convergence is to relate all sources to each other, the relationships between grammars will be represented as grammar transformations. We say that grammars g_1 and g_2 are f-equal, if $f(g_1) = g_2$ (where "=" refers to structural equality on grammars, and f denotes the meaning of a grammar transformation). When f is a refactoring (i.e., a semantics-preserving transformation), then f-equality coincides with grammar equivalence. If f is a semantics-increasing (-decreasing) transformation, then we have shown an inclusion ordering for the languages generated by the two grammars.

We use the terms "semantics-preserving", "-increasing" and "-decreasing" in the formal sense of the language generated by a grammar. Clearly, the composition of (sufficiently expressive) increasing and decreasing operators allows us to relate arbitrary grammars, in principle. Hence, more restrictions are needed for accumulating reasonable grammar relationships, as we will discuss below. We also mention that there is a rare need for operators that are neither semantics-increasing nor -decreasing. In this case, we speak of a semantics-revising operator. Consider, for example, an unconstrained **replace** operator for expressions in grammar productions that may be needed if we face conflicting definitions of a nonterminal in two given grammars.

The baseline scenario for grammar transformation in the context is convergence is as follows. Given are two grammars: g_1 and g_2. The goal is to find f such that g_1 and g_2 are f-equal. In this case, one has to gradually aggregate f by addressing the various differences reported by the comparator. In our current implementation of grammar comparison, we do not make any effort to propose any transformation operators to the user, but this is clearly desirable and possible.

In JLS, given the differences reported by the comparator and presented in the previous section, the grammar engineer authors an transformation to add an extra chain production

Using a grammar transformation operator to introduce a chain production

```
chain (
 ClassDeclaration:
      NormalClassDeclaration );
```

read2' (Intermediate grammar fragment)

```
ClassDeclaration:
      NormalClassDeclaration
NormalClassDeclaration:
      ClassModifiers? "class" Identifier Super? Interfaces?
            ClassBody
```

Using grammar transformation operators to add enumeration declarations

```
introduce ( ... );
addV (
 ClassDeclaration:
      EnumDeclaration );
```

read2'' (Intermediate grammar fragment)

```
ClassDeclaration:
      NormalClassDeclaration
      EnumDeclaration
NormalClassDeclaration:
      ClassModifiers? "class" Identifier Super? Interfaces?
            ClassBody
EnumDeclaration:
      ...
```

Using a grammar transformation operator to insert a nillable symbol

```
appear (
 NormalClassDeclaration:
      ClassModifiers? "class" Identifier ⟨TypeParameters?⟩ Super?
            Interfaces? ClassBody );
```

Figure 4.4: Transforming the grammar and proving (**chain** ∘ **introduce** ∘ **addV** ∘ **appear**)-equality.

for `NormalClassDeclaration`. This transformation and a few subsequent ones as well as all intermediate results are listed in Figure 4.4.

The idea is now that such compare/transformation steps are repeated. Hence, we compare the intermediate result, as obtained above, with the grammar *read3*. It is clear that the nominal difference for `NormalClassDeclaration` has been eliminated. The comparator reports the three following differences:

- ◇ Nominal difference:
  - read2': nonterminal *EnumDeclaration* missing.
- ◇ Structural difference: nonterminal *NormalClassDeclaration*
  - read2':
    ClassModifiers? `"class"` *Identifier Super*? *Interfaces*? *ClassBody*
  - read3:
    ClassModifiers? `"class"` *Identifier TypeParameters*? *Super*? *Interfaces*? *ClassBody*
- ◇ Structural difference: nonterminal *ClassDeclaration*
  - Unmatched alternatives of read2': none
  - Unmatched alternatives of read3: `EnumDeclaration`

We see that enumerations are missing entirely from *read2'*, and hence a definition has to be introduced, and a corresponding alternative has to be added to `ClassDeclaration`. Once we are done, the result is again compared to *read3*:

- ◇ Structural difference: nonterminal *NormalClassDeclaration*
  - read2'':
    ClassModifiers? `"class"` *Identifier Super*? *Interfaces*? *ClassBody*
  - read3:
    ClassModifiers? `"class"` *Identifier TypeParameters*? *Super*? *Interfaces*? *ClassBody*

Again, this difference is suggestive. Obviously, the definition of `NormalClassDeclaration` according to *read2"* does not cover the full generality of the construct, as it occurs in *read3*. The structural position for the type parameters of a class has to be added. (This has to do with Java generics which were added in the 3rd edition of the JLS.) There is a designated transformation operator that makes new components **appear** (such as type parameters) in existing productions; the newly inserted part is marked on Figure 4.4 with angle brackets. This is a downward-compatible change since type parameters are optional. Once these small transformations have been completed, all the discussed differences are resolved, and the comparator attests structural equality.

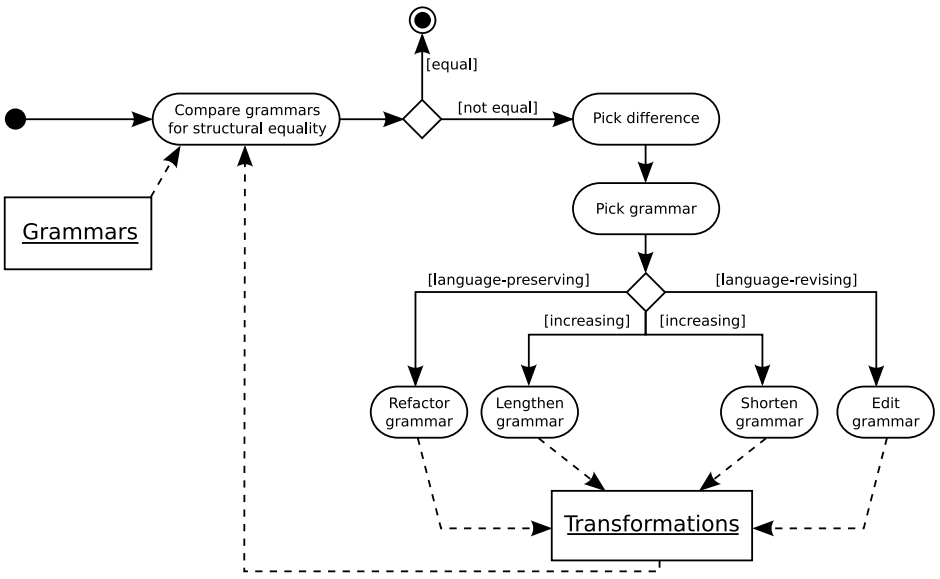


Figure 4.5: An activity diagram for the informal workflow of grammar convergence.

4.7 Convergence process

As we were discussing grammar comparison and transformation, we already alluded to a basic compare/transform cycle—this cycle is indeed the spline of the convergence process. However, it provides insufficient structure to a complex convergence situation. As a remedy, we identify *phases* for the convergence process in order to impose more structure and discipline onto it. These phases assume asymmetric, binary convergence trees: we favour one of the inputs for target such that all nominal and structural differences are resolved by changing one of the two grammars rather than the other one. An exception is needed if the favoured grammar is found to be incorrect or structurally unfavorable. There are five consecutive convergence phases: the initial extraction phase involves a mapping from an external grammar format and is therefore implemented as a standalone tool in our infrastructure; the other four convergence phases are directly concerned with transformation.

In a tiny project such as FL convergence the usage of this particular strategy does not pay back since the whole transformation chain can fit on one page after being pretty-printed, and it is possible to use a straightforward way presented on Figure 4.5. In chapter 5 and in [167, 168] we consider a much larger case study (more than 30000 lines of XML code of 1600+ transformations) that would not be technically feasible at all without any firm strategy.

Extraction: A starting point for grammar extraction is always a set of real grammar artefacts. A mapping is required for each kind of artefact so that grammar knowledge

can be extracted and represented in a uniform grammar format. (In in the case of our infrastructure, we use BGF—a BNF-like Grammar Format.) Each extractor may implement particular design decisions in terms of any normalization or abstraction to be performed along with extraction. Once extraction is completed, a (possibly incorrect or not fully interconnected) grammar is ready for transformation.

Convergence preparation: This convergence phase involves correcting immediately obvious or a priori known errors in the given sources. These corrections are represented as grammar transformations so that they can be easily revisited or re-applied in the case when the extractor is modified or the source changes.

Nominal matching: We perform asymmetric compare/transform steps. That is, the non-favored grammar is compared with the (prepared) favored grammar, which is the baseline for the (intermediate) target of convergence. The objective of this convergence phase is to align the syntactic categories of the grammars in terms of their nonterminals. The nominal differences, as identified by comparison, guide the grammar engineer in drafting transformations for renaming as well as extraction and inlining such that the transformations immediately reduce the number of nominal differences. It is important to notice that we restrict ourselves to operators for renaming, inlining, and extraction. These operators convey our intuition of (initial) nominal alignment. We make these assumptions:

- ◇ *When a nonterminal occurs in both grammars, then it models the same syntactic category (conceptually).* If the assumption does not hold, then this will become evident later through considerable structural differences, which will trigger a renaming to resolve the name clash. Such corrective renaming may be pushed back to the phase of convergence preparation.
- ◇ *Any renaming for nonterminals serves the purpose of giving the same name to the same syntactic category (in an conceptual sense).* If a grammar engineer makes a mistake, then this will become evident later, again, through considerable structural differences. In this case, we assume that the grammar engineer returns to the name matching phase to revise the incorrect match.

Structural matching: We continue with asymmetric compare/transform steps. This convergence phase dominates the transformation effort; it aligns the definitions of the nonterminals in a structural sense. The structural differences, as identified by comparison, guide the grammar engineer in drafting transformations for refactoring such that they immediately reduce the number of structural differences. As we continue to limit ourselves to refactoring, the order of the individual transformations does not matter due to its commutativity. The grammar engineer can simply pick any applicable refactoring operator, but the firm requirement is that the number of structural and nominal differences declines, which is automatically verified by our infrastructure.

Resolution: This convergence phase consists of three kinds of steps, as discussed in more detail in §5.6: *extension*, *relaxation* and *correction*. In the case of semantics-

> increasing operators, it is up to the grammar engineer to perform the classification. Semantics-decreasing operators serve correction on the grounds of a convention. That is, we assume a directed process of convergence where the grammars of extended (relaxed) languages are derived from the grammars of "sublanguages". However, if the grammars violate such an intended sublanguage relationship, then correction must be expressed through semantics-decreasing operators.

The correctness of the process relies on one assumption regarding the limited use of non-semantics-preserving operators. In particular, non-semantics-preserving operators should only be used, if the given grammars are not equivalent. Making equivalent grammars non-equivalent is clearly not desirable. Currently, we cannot verify this assumption, and in fact, it is generally impossible because of undecidability of grammar equivalence. However, a heuristic approach may be feasible, and provides an interesting subject for future work. Even when the given grammars are non-equivalent, we still need to limit the use of non-semantics-preserving operators for correctness' sake. That is, we should disallow zigzag transformations such that semantics-increasing and -decreasing transformations partially cancel each other.

We use the number of nominal and structural differences as means to **track progress** of grammar convergence. Each unmatched nonterminal symbol of either grammar counts as a nominal difference. For every nominally matched nonterminal, we add the maximum number of unmatched alternatives (of either grammar), if any, to the number of structural differences.

The main guiding principle for grammar convergence is to consistently reduce the number of grammar differences throughout the two matching convergence phases as well as the final resolution phase. Figure 5.2 will illustrate this principle in the next chapter for one edge in the convergence graph of the JLS case study. The figure also visualizes that nominal differences tend to be resolved earlier than structural differences.

In order to employ a differential approach [178] to benchmarking, we need to establish the notion of a *synchronisation point*. Suppose we have a target with two branches, A and B. For the branch A, the synchronisation point will be a grammar in the path B that is a direct result of nominal matching (if we put it earlier on path B, there will be irrelevant nominal mismatches; if we put it later, first structural matches will become unnoticeable). All the metrics are performed by comparing every grammar on the path A with this synchronisation point. Every step, be it a single transformation or an atomic transformation sequence, must not increase the number of mismatches between the "current" grammar and its synchronisation point.

Our transformation infrastructure is aware of the different phases of convergence, and it checks the incremental reduction of differences at runtime. As a concession to a simple design of the operator suite for grammar transformations, restructuring steps may also slightly increase structural differences as long as they are explicitly grouped in "transactions" whose completion achieves reduction.

4.8 Programmable grammar transformations

In the more preferable case, two different grammars can be refactored to become syntactically identical. We use the term grammar *refactoring* in the sense of semantics-preserving transformations, this meaning being well-spread and widely used in program refactoring [70, 138, 200]. We assume an intuitive understanding of the notion of grammar semantics until later in this section. In the less preferable case, non-semantics-preserving transformations are due, in which case weaker properties should limit the impact.

4.8.1 Transformation properties

We may refer to the semantics of a grammar as the language (set of strings) generated by the grammar, as it is common for formal languages—for context-free grammars, in particular. With the string-oriented semantics in mind, few transformations are semantics-preserving. Examples include renaming of nonterminals, and fold/unfold manipulations. To give an example where different semantics are needed consider the scenario of aligning a concrete and an abstract syntax.

When necessary, we may apply the algebraic interpretation of a grammar, where grammar productions constitute an algebraic signature subject to a term-algebraic model. In this case, the terminal occurrences in any given production do no longer carry semantic meaning; they are part of the function symbol. (Hence, abstract and concrete syntaxes can be aligned now.) Some transformations that were effortlessly semantics-preserving with respect to the string-oriented semantics, require designated bijective mappings with respect to the term-oriented semantics, e.g., fold/unfold manipulations, but generally, the term-oriented semantics admits a larger class of semantics-preserving transformations than the string-oriented one.

Transformations that are not semantics-preserving may still be "reasonable" if they model data refinement [87, 190]. We say that a data type (domain) A can be refined to a data type (domain) B, denoted by the inequality $A \leq B$, if there is an injective, total function $to : A \to B$ (the representation function), and a surjective, possibly partial function $from : B \to A$ (the abstraction function) such that $from.to = id_A$, where id_A is the identity function on A. A simple way to think of data refinement in our context is that a transformation increases or decreases the number of "representational options", e.g., by making a certain syntactic structure optional or mandatory. Here we assume the term-oriented semantics with its term-algebraically defined domains.

Some grammar differences may require more arbitrary replacements. In this case, one would want to be sure that *a)* indeed no more preserving transformation is possible, and *b)* the scope of replacement is as small as possible. To this end, we have developed an effective strategy, which however is beyond the scope of the present paper.

4.8.2 Grammar refactoring

Let us demonstrate a number of refactoring operators. In our running example, there are two sources that are very close to each other: *antlr* and *dcg*; see Listing 4.6. Both sources serve top-down parsing. The remaining differences are neutralized by the follow-

ing refactorings to be applied to the ANTLR grammar; we show the applications of the transformation operators combined with an explanatory comment:

```
renameN(NEWLINE, newline);
renameN(ID, name);
renameN(INT, int);
```

After aligning the nonterminals by introducing lower case and renaming *ID* to *name*, we can start refactorings, moving from one top-choice production to multiple productions for *expr*, *atom* and *ops*, inlining unneeded productions and giving labels to those productions that lack them in the ANTLR source.

```
vertical( in expr );
unchain(
 expr:
      apply
);
unchain(
 expr:
      binary
);
unchain(
 expr:
      ifThenElse
);
vertical( in atom );
```

```
designate(
 [literal] atom:
      int
);
designate(
 [argument] atom:
      name
);
vertical( in ops );
designate(
 [equal] ops:
      "=="
);
designate(
 [plus] ops:
      "+"
);
designate(
 [minus] ops:
      "-"
);
```

Figure 4.6 briefly describes a small suite of refactoring operators. All operators except ***permute*** are semantics-preserving with respect to string-oriented semantics. Without exception, the operators are semantics-preserving with respect to term-oriented semantics. For the statistics of their use we refer to Table 4.2, for detailed descriptions—to chapter 7.

renameN(N_1, N_2) renames all occurrences of the nonterminal N_1 to N_2, provided N_2 does not occur in G. There are also operators ***renameL*** and ***renameS*** for renaming labels and selectors. In ***renameS***(OL, S_1, S_2), OL is an optional label; if present, S_1 is renamed only in the scope of the identified production, or globally otherwise. See section 7.14.

permute(P) replaces a production say P' by P, where P and P' must agree on their defined nonterminal and (optional) label while their defining expressions must be permutations of each other (with regard to sequential composition). See subsection 7.10.3. Here is an example:

- ◇ A production: `[binary] expr: expr ops expr`
- ◇ A permutation: `[binary] expr: ops expr expr`

verticalN(N) converts the choice-based definition of N to multiple productions. Each alternative of the choice becomes another production. An outermost selector, if present, is reused as a production label (but must not yet be in use in G). The variation ***verticalP***(P) limits the conversion to a production P. There is the opposite operator ***horizontal***. See subsection 7.7.9.

unchain(P) replaces a chain production P and the production P' that defines the nonterminal of its defining expression by a production that inlines P' in P. See subsection 7.6.7. Here is an example:

- ◇ The chain production: `expr: literal`
- ◇ The referenced definition: `literal: int`
- ◇ The result of unchaining: `expr: int`

There is also the opposite operator ***chain***.

designate(P) replaces an unlabeled production say P' by its labeled variant P. See subsection 7.12.1. There is also the opposite operator ***unlabel***.

lassoc(P) replaces list-based recursion by binary recursion. (The 'l' in ***lassoc*** is for *left* association hinting at the expected effect at the instance level. There is also an operator ***rassoc*** hence.) Here, P describes binary recursion. Their must be a corresponding production in G that uses list-based recursion. See subsection 7.7.12. Here is an example:

- ◇ Binary recursion: `[binary] expr: expr ops expr`
- ◇ List-based recursion: `[binary] expr: expr (ops expr)*`

Figure 4.6: Operators for grammar refactoring (G refers to the input grammar.)

4.8.3 Grammar editing

Grammars may also differ in ways that cannot be neutralised by strict refactoring. We use the term grammar *editing* for these remaining transformation scenarios. Let us consider an example. The *antlr* and *dcg* sources of FL use two expression layers (expr and atom), whereas the *sdf* source only uses one expression layer (and deals with priorities by extra annotations). The following transformation uses an editing operator ***unite*** to merge the two layers (i.e., nonterminals) in one:

unite `(atom, expr);`

project(P) replaces a production say P' by P, where P and P' must agree on their defined nonterminal and (optional) label, and the defining expression of P must be a sub-sequence of the one of P' (with regard to sequential composition). See subsection 7.11.5.

abridge(P) removes a reflexive chain production P. See subsection 7.6.5.

unite(N_1, N_2) recursively merges the definitions of N_1 and N_2 into one by replacing all defining and using occurrences of N_1 by N_2. See subsection 7.8.5.

define(Ps) adds the productions Ps as a definition, assuming that all productions agree on a defined nonterminal that is used but not yet defined in G. We take the view that an undefined nonterminal is implicitly defined to be equal to the universal type. Hence, the ***define*** operator essentially 'narrows' a definition in a semantic sense. There is also the opposite operator ***undefine*** for discarding the explicit definition of a nonterminal. See subsection 7.11.1.

Figure 4.7: Operators for grammar editing (G refers to the input grammar.)

Consider another example. The grammars in Listings 4.5–4.12 differ with regard to the grammatical details regarding FL's literals and function or argument names. The sources which define the abstract syntax (such as *xsd* or *ecore*) use precise (simple) types int and string, whereas the grammars defining the concrete syntax (such as *antlr* or *dcg*) leave the corresponding nonterminals undefined because the extraction only returned immediate context-free structure in those cases. The following transformations resolve the undefined nonterminals in the *dcg* grammar in accordance to the *xsd* source:

```
define(
 name:
       STR
);
define(
 int:
       INT
);
inline(name);
inline(int);
```

Consider a final example. The convergence of concrete and abstract syntax definitions requires a transformation that removes all details that are specific to concrete syntax definitions. That is, we strip off all terminals (handled by an automated transformation generator, see section 4.9), project away the reference to newline, remove the bracketing production and permute the ingredients of binary expressions to resemble prefix instead of infix notation. Thus:

	abstract	concrete	java	limit	model	topdown	xframeworks	Total
Number of lines	194	26	36	147	547	105	175	1230
Number of transformations	9	2	8	14	28	14	16	91
○ semantics-preserving	3	—	8	10	27	14	15	77
○ semantics-increasing or -decreasing	6	2	—	2	—	—	1	11
○ semantics-revising	—	—	—	3	—	—	—	3
Number of steps	2	1	3	4	5	3	4	22

Table 4.1: Effort metrics and categorisation of the convergence transformations for FL.

```
abstractize (
 function:
      STR STR+ ⟨"="⟩ expr newline+
);
project (
 function:
      STR STR+ expr ⟨(newline+)⟩
);
abridge (
 expr:
      expr
);
permute (
 [binary] expr:
      ops expr expr
);
```

Figure 4.7 briefly describes a small suite of editing operators. For the statistics of their use we refer to Table 4.2, for detailed descriptions of all editing operators—to chapter 7. All but one of the operators model data refinement in one direction or the other, i.e., from input (I) to output (O), or vice versa:

- ◇ ***abridge***: $O \leq I$
- ◇ ***unite***: $I \leq O$
- ◇ ***define***: $O \leq I$
- ◇ ***undefine***: $I \leq O$

The operator ***project*** does not model data refinement; rather it models "data disposal". Its I-to-O mapping for ***project*** is total, surjective, non-injective; its O-to-I mapping is not generally defined.

4.9 Transformation generators

Grammar convergence research has started with an objective to use programmable grammar transformations to surface the relationships between grammars extracted from

	abstract	**concrete**	**java**	**limit**	**model**	**topdown**	**xframeworks**	**Total**
○ *rename*	—	—	5	—	13	3	5	26
○ *inline*	—	—	—	2	4	—	—	6
○ *extract*	—	—	—	—	1	—	—	1
○ *abridge*	—	—	—	1	—	—	—	1
○ *unchain*	—	—	—	5	—	3	—	8
○ *massage*	3	—	—	—	3	—	—	6
○ *factor*	—	—	—	—	1	—	—	1
○ *eliminate*	—	—	3	—	1	—	—	4
○ *vertical*	—	—	—	2	—	3	2	7
○ *lassoc*	—	1	—	—	—	—	—	1
○ *widen*	—	—	—	—	—	—	1	1
○ *unite*	—	1	—	—	—	—	—	1
○ *narrow*	6	—	—	—	—	—	—	6
○ *permute*	—	—	—	1	—	—	—	1
○ *define*	—	—	—	2	—	—	—	2
○ *project*	—	—	—	1	—	—	—	1
○ *replace*	—	—	—	—	1	—	—	1
○ *designate*	—	—	—	—	—	5	8	13
○ *deanonymize*	—	—	—	—	2	—	—	2
○ *anonymize*	—	—	—	—	2	—	—	2

Table 4.2: XBGF operators usage for FL convergence.

	Number of Productions	Number of Nonterminals	Number of Tops	Number of Bottoms
antlr	8	8	1	3
dcg	11	5	1	3
sdf	11	4	1	3
txl	4	4	1	3
ecore	12	12	1	0
ecore2	9	9	1	0
xsd	10	10	2	0
om	10	10	3	0
jaxb	11	11	4	0

Table 4.3: Metrics for the FL grammars.

	Number of Productions	Number of Nonterminals	Number of Tops	Number of Bottoms
topdown	11	5	1	3
xframeworks	11	4	1	3
concrete	11	4	1	3
model	9	9	1	0
java	9	9	2	0
abstract	9	9	1	0
limit	10	4	1	0

Table 4.4: Metrics for the transformed grammars.

sources of different nature. Hence, we mostly aimed to provide a comprehensive transformation suite, a convergence strategy and an infrastructure support. However, at some point we found it easier to generate the scripts to resolve specific mismatches rather than to program them manually. A full-scale research on this topic remains future work, yet below we present the results obtained so far and the considerations that can serve as foundation for the next research steps.

Consider an example of converging concrete and abstract syntax definitions. This situation requires a transformation that removes all details that are specific to concrete syntax definitions, i.e., first and foremost strips all the terminals away from the grammar. Given the grammar, it is always possible to generate a sequence of transformations that will remove all the terminal symbols. It will take every production in the grammar, search for the terminals in it and if found, produce a corresponding call to **abstractize**. For instance, given the production:

```
[ifThenElse] expr:
      "if" expr "then" expr "else" expr
```

the following transformation will be generated:

```
abstractize(
 [ifThenElse] expr:
      ⟨"if"⟩ expr ⟨"then"⟩ expr ⟨"else"⟩ expr
);
```

Other generators we used in the FL case study were meant for removing all selectors from the grammar (works quite similar to removing terminals), for disciplined renaming (e.g., aligning all names to be lower-case) and for automated setting of the root nonterminals by evaluating them to be equal to top nonterminals of the grammar.

Eliminating all unused nonterminals can also be a valuable generator in some cases. For us it was not particularly practical since we wanted to look into each nominal difference (which unused terminal is a subtype of) in order to better align the grammars.

More aggressive transformation generator example can be the one that **inline**s or **unchain**s all nonterminals that are used only once in the grammar. This can become a powerful tool when converging two slightly different grammars and thus can be considered a

form of aggressive normalisation. We did not work out such an application scenario for grammar convergence so far.

Deyaccification [217], which has already been discussed in section 2.8 and [143] and will be talked again in chapter 5 and subsection 7.7.4, can also be performed in an automated fashion. In general, all grammar transformations that have a precondition enabling their execution, can be generated—we only need to try to apply them everywhere and treat failed preconditions as identical transformations.

On various occasions we also talk about "horizontal" and "vertical" productions (called "flat" and "non-flat" correspondingly in [165]), where being vertical means having separate productions for one nonterminal and being horizontal means having one production with a top choice. There are also singleton productions that are neither horizontal nor vertical, and productions that can be made horizontal by distribution. According to this classification and to the need of grammar engineers, it is possible to define a range of generators of different aggressiveness levels that would search for horizontal productions and apply **vertical** to them; or search for vertical productions and apply **horizontal** to them; or search for potential horizontal productions and apply **distribute** and **vertical** to them; etc.

It is important to note here that even though complete investigation of the possible generators and their implementation remain future work, this alone will not be enough to replace human expertise. Semi-automation will only be shifted from "choose which transformation to apply" to "choose which generator to apply". A strongly validated strategy for automating the choice is needed, which is not easy to develop, even if possible.

4.10 Language Convergence Infrastructure

Language Convergence Infrastructure, or LCI, is the name of a tool that handles most of the functionalities needed to support grammar convergence. It was not called "Grammar Convergence Infrastructure" because there are some coupled transformation facilities implemented in it, and we intend to stress and enhance them in the future research projects. The necessary information about the concrete convergence scenario is provided to the LCI in a form of a configurational DSL. The DSL was named LCF for LCF Configuration Format, it is described below in detail, also annotated with remarks about LCI behaviour. The technology that helped generate the content of this section will be presented in chapter 6.

4.10.1 Main configuration elements

All DSL elements such as sources, targets and tools, are presented in a completely volatile order.

Each configurational entity is either defining a shortcut, specifying a tool or a generator, or defining a source, a target or a test set.

4.10.1.1 Syntax

```
configuration:
      definition+
```

```
definition:
      shortcut
      generator
      tool
      source
      target
      testset
```

4.10.2 Shortcuts

A shortcut in LCF is just a simple macro. If binds a longer definition to a concise name. For example, a path with long directory names that is used multiple times in the LCF document is a good shortcut.

4.10.2.1 Syntax

```
shortcut:
      name::STR expansion::xstring
```

4.10.3 Generators

A method for automated generation of XBGF scripts. The **command** is an executable that takes two more arguments: the input BGF and the output XBGF.

4.10.3.1 Syntax

```
generator:
      name::STR command::xstring
```

4.10.4 Sources

A source in LCF is the starting point for the grammar convergence. A source must have a name by which it will be referenced later. A source must have a defined extractor that provides LCI with a BGF. A source may have a parser and an evaluator defined as commands — if they are present, they will be tested.

4.10.4.1 Syntax

```
source:
      name::STR derived? grammar tree? testing?
```

```
grammar:
      extraction::xstring parsing::xstring? evaluation::xstring?
```

```
tree:
      extraction::xstring evaluation::xstring?
```

```
testing:
      set::xstring+
```

```
derived:
      from::STR using::STR
```

4.10.5 Targets

A target in LCF is the convergence point. A target must have a name by which it is referenced and displayed on a diagram. A target can have any number of branches. Each branch defines an input that can be either source name or a target name. Each branch also references a sequence of actions that are performed on that source or target to reach this target. Once all branches are completed, the grammar comparator is run to make sure all results converge. If there is one branch, no comparison takes place. If there are three or more branches, comparing occurs pairwise. Within one branch actions are sorted by convergence phase (see above), all of which are optional: preparation, nominal matching, structural matching and resolution.

4.10.5.1 Syntax

```
target:
      name::STR branch+
```

```
resolution:
      extension::phase
      relaxation::phase
      correction::phase
```

```
branch:
      input::STR preparation::phase? nominal-matching::phase?
            structural-matching::phase? resolution*
```

During convergence process, LCI regularly compares the result of each transformation step (XBGF file) of each branch to the reference points of adjacent branches. Two diagrams are drawn as a result, one contains only sources and targets with the edges signifying which target is based upon which targets or sources; and one is a very detailed overview of all the files that have been executed and all the numbers of nominal and structural differences.

In the special measurement mode all the transformational scripts are sliced in atomic steps before convergence. In this case, every step is followed by grammar comparison. The convergence process is stopped prematurely if the number of differences suddenly goes up. Running LCI in this mode can take considerable time (for six JLS grammars it is about an hour).

4.10.6 Phases

4.10.6.1 Syntax

```
phase:
      (perform::STR | automated)+
```

```
automated:
      method::STR result::STR
```

The interface to generators is universal. We used it only with **casexbgf**, **striptxbgf** and **stripsxbgf**, but it is possible to define any number of generators outside LCI and connect them here accordingly.

4.10.7 Test sets

A test set in LCF is a set of code samples in the given language. When testing, all .src files are parsed and all .run files are evaluated in the context of their .src to yield .val

4.10.7.1 Syntax

```
testset:
      name::STR command::xstring
```

4.10.8 Tools

A tool in LCF is an external script. At this point there are three tools which output is needed by LCI:

validation: The validator makes sure that the BGF grammars are valid against their schema. This includes extracted BGF files as well as all derived ones. If the validator is not defined, LCI works fine without it.

comparison: The diff tool takes two parameters and returns zero if both BGF files are deemed equal. If the diff tool is not defined, LCI can only work with one-branch targets.

transformation: The XBGF engine (or any other grammar transformation engine, actually) that takes one BGF and one XBGF, transforms the former according to the latter and produces one BGF as its result. Since this is the core of grammar convergence, LCI does not work unless transformation tool is defined.

4.10.8.1 Syntax

```
tool:
      name::toolType grammar::xstring tree::xstring?
```

```
[validation] toolType:
      ε
```

```
[comparison] toolType:
      ε
```

```
[transformation] toolType:
      ε
```

4.10.9 xstring

A universal type used throughout the LCF: a mix of strings and references to shortcuts.

4.10.9.1 Syntax

```
xstring:
      (expand::STR | STR)*
```

4.11 Related work

4.11.1 Interoperability

The consistent use of structural and nominal types (to be compared here with grammar knowledge) is typically supported by programming-language type systems, exchange formats, and interface definition languages (IDLs). IDLs are specifically used in distributed programming [61]. Exchange formats (e.g., based on XML Schema [75, 208]) are widely used for any sort of data- and communication-intensive programming. A domain with classic grammar-like exchange formats is reverse engineering [127, 151]; here, bridging between different formats has been of continued interest. In the broad context of interoperability, *grammar convergence provides verification power for the particular situation where diverse grammars, grammar-like types, interface definitions, or exchange formats are ingrained in different software artefacts*. By going through an extraction step, we do not need to insist on the consistent use of a common type system, IDL, or exchange format. By the anticipation of refactoring and more liberal transformations, we allow for flexible correspondence relationships.

4.11.2 Testing grammarware

The I/O behaviour of grammarware (e.g., the acceptor behaviour of a front end) can be tested by "sampling"—subject to test-data generation and test suites [155, 162, 174, 222]. Such approaches are specifically useful for differential testing of grammarware. *Grammar convergence is complementary in that it provides a static verification of the correspondence between different software artefacts based on access to the internal structure of the artefacts.* It can also be applied to specify the degree of deviation of grammars from each other.

4.11.3 Generators and synchronisers

If two artefacts are meant to use the same grammar (type, etc.) modulo its realisation in the software artefact, then, arguably one grammar (or software artefact) should be generated from the other, as in the case of XML-object mapping [161, 179, 244]. The strong version of this expectation is to even require bidirectional synchronisation between scattered grammar knowledge, akin to bidirectional model/model or model/code synchronisation in model-driven engineering [226]. As should be clear from the list of use cases in the introduction, *grammar convergence provides verification support in cases where generators or bidirectional synchronisers are not, have not been, or cannot (yet) be used for whatever technical or project-specific reason*. For instance, the derivation of a useful parser description for any possible parsing technology from a technology-independent baseline grammar (say, found in a language standard) requires a genuine effort; it is not a manageable scenario for a generator, i.e., a "parser-generator input generator" (except for the simplest cases [149]). As another example, consider the problem of different versions of a highly idiosyncratic parser description [203]. Bidirectional synchronisation is beyond current limits, whereas grammar convergence clearly applies.

4.11.4 Grammar recovery

Our work is heavily influenced by the idea of grammar recovery [21, 58, 132, 158, 164, 219], especially those forms that begin with the extraction of grammar knowledge from an artefact like a standard (containing syntax) or an implementation (based on an idiosyncratic parsing technology). Just like grammar convergence, grammar recovery involves (manual or automated) grammar transformations, which we discuss below. While grammar recovery has focused on (mostly concrete) syntax definitions, grammar convergence applies to a very broad interpretation of grammars (XML schemas, object models, etc.). Grammar recovery is a reverse-engineering method focused at a path from an original artefact to a declarative grammar. In contrast, *grammar convergence is a verification method that establishes and maintains grammatical correspondence constraints on software*.

4.11.5 Grammar transformation

(Automated) grammar transformation has seen a surge of interest over the last decade, but the concept is much older because parsing technologies tend to require internal transformations, cf. the classic example of left-recursion removal [2, 172, 189]. There are several modern use cases for grammar transformations that support automated software engineering and grammar-based programming in one way or another: grammar recovery (see above), derivation of an abstract from a concrete syntax [249], problem-specific customisation of grammars [43], mediation between different grammar classes [207]. Ultimately, we speak of grammar programming or programmable grammar transformations [42]. *Grammar convergence relies on an advanced operator suite for grammar transformation that is carefully designed around the expressiveness of a unified grammar format*. (Consider, for example, the operators ***vertical*** or ***designate*** that we have encountered.) The precise definition of that suite is beyond the scope of this introductory text.

4.11.6 Grammar convergence

Finally, we mention a few instances of grammar engineering techniques that can be seen as specific forms of grammar convergence. In [21], the compatibility of (different implementations of) precedence rules in grammars is checked. Our (current) grammar convergence approach does not fully address such scenarios that involve specifics of parsing techniques, but in return, it is more generic (with regard to the notion of grammar), and programmable (with regard to deltas between grammars). In [136], the correspondence between a concrete and abstract syntax definition is addressed by permitting incomplete specifications for both syntaxes, as long as they complement each other consistently. Grammar convergence, provides a general tool for "programming" such relationships and verifying them. In [203], the problem of proliferation of grammar-based artefacts (in fact, parser descriptions with semantic actions) due to grammar evolution or the need for different grammar use cases is addressed. Based on ideas of version control, a parser description always remains associated with its "prototype", so that extensions of the prototype can be signaled to derivatives. Grammar convergence also covers this scenario, except that it would be unable to detect modifications that are abstracted away during grammar extraction.

4.12 Concluding remarks

If unit testing is the simple, pragmatic, and effective method to generally validate the I/O behaviour of software modules, then grammar convergence is the simple, pragmatic, and effective method to keep scattered grammar knowledge in sync. In addition, the method can be used to capture the intended or the accidental differences between instances of scattered grammar knowledge. Further, grammar convergence also applies at the instance level (populated by XML trees, derivation trees, parse trees, etc.). That is, it can compare and converge "data" from different software artefacts. (See our implementation for the latter aspect that is not discussed in the present text.)

Here are some topics for ongoing and future work on grammar convergence:

- ◇ We have established the effectiveness of grammar convergence even for huge grammars (chapter 5 will present the report on a major case study), but arguably a semi-automatic approach to the inference and accumulation of grammar transformation would improve productivity.
- ◇ We have a good pragmatic understanding of the grammar transformations that are needed in practice, but more work is needed to deliver the ultimate (relatively complete, orthogonal, semantically well-defined) operator suite. The current state of the operator suite will be presented in chapter 7.
- ◇ BGF's provisions for coverage of XML schemata, object models and other non-standard grammars call for more research on the effectiveness of the approach for scenarios that would not count as grammar-centric (or schema-centric). This also necessitates a deeper comparison with existing metamodelling facilities.

Chapter 5

Case study on recovery and convergence

In this chapter a completed effort is described to recover the relationships between all the grammars that occur in the different versions of the Java Language Specification (JLS). The relationships are represented as grammar transformations that capture all accidental or intended differences between the JLS grammars. This process is mechanised and it is driven by simple measures of nominal or structural differences between any pair of grammars involved. Our work suggests a form of consistency management for the JLS in particular, and language specifications in general.

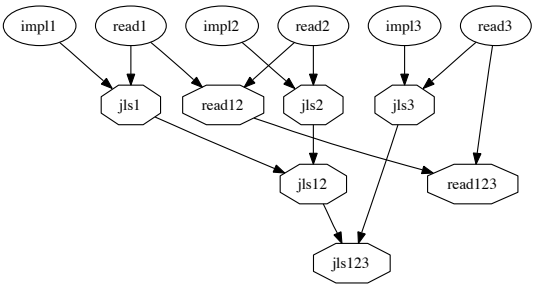
Grammar extraction is performed in an automated fashion with one general extractor being able to pick up any HTML document with pieces of EBNF-like productions scattered inside it, collect them all, parse them, convert to the BGF grammar and produce it to be used later in the convergence process. The extractor is not trivial in a sense that it applies certain recovery rules during the extraction process to fix the majority of errors that are found in language documentation due to its hand-made nature. The rules are empirically derived, subsequently generalised and presented in this chapter.

Six extracted grammars are then transformed to reach certain convergence points. Due to the delicate and complicated matter of this case study no generators were used in this chapter. Thus, all transformation steps were programmed by a human grammarware expert. Examples of them are provided on later pages, together with basic metrics and benchmarks that were utilised to guide the process of convergence.

5.1 Java is not syntax-safe—apparently[1]

Many software languages (and programming languages in particular) are described simultaneously by multiple grammars that reside in different software artefacts. For instance, one grammar may reside in a language specification; another grammar may be encoded in a parser specification. Many software languages are also subject to evolution, which

[1] The title is a pun and an homage on a series of papers [53], [193],...



impl1	JLS1, §19
read1	JLS1, §§4.1–15.27
impl2	JLS2, §18
read2	JLS2, §§4.1–15.28
impl3	JLS3, §18
read3	JLS3, §§4.1–15.28

Figure 5.1: Binary convergence tree for the JLS grammars. The *nodes* in the figure are grammars where the leaves correspond to the original JLS grammars and the other nodes are derived. The *directed edges* denote grammar transformation chains. We use a (cascaded) binary tree here, i.e., each forking node is derived from two grammars. The *implX* leaves are "implementable" grammars, the *readX* ones are "readable".

means that artefacts with embedded grammars may also occur in different versions. *This diversity of grammars for any single software language represents a fundamental consistency challenge.* Grammars (and hence grammar-dependent artefacts) may actually disagree on the software language in question in a hard-to-spot manner. The intended, evolution-related differences between two grammars may be obfuscated by other more accidental or superficial differences between the grammars.

In the previous chapter we have addressed the fundamental problem of grammar diversity by a contribution to the method of *grammar convergence* [166]; this method combines grammar extraction (to obtain raw grammars from artefacts and represent them uniformly), grammar comparison (to determine nominal and structural differences between grammars), and grammar transformation (to represent the relationships between the grammars at hand by transformations that make them structurally equal.

In this chapter we report on a major case study for grammar convergence, and we refine the method to provide better scalability and reproducibility. The case study concerns the 3 different versions of the Java Language Specification (JLS; [77, 78, 79]) where each of the 3 versions contains 2 grammars—one grammar is optimised for readability (see *read1–read3* in Figure 5.1), and another one is intended to serve as a basis for implementation (see *impl1–impl3* in Figure 5.1).

Here we note that the JLS is critical to the Java platform—it is a foundation for compilers, code generators, pretty-printers, IDEs, code analysis and transformation tools and other grammarware for the Java language. One would expect that the different grammars per version are essentially equivalent in terms of the generated language. (As a concession to practicality, i.e., implementability in particular, one grammar may be more liberal

than the other.) One would also expect that the grammars for the different versions engage in an inclusion ordering (again, in terms of the generated languages) because of the backwards-compatible evolution of the Java language.

Those expected relationships of (liberal) equivalence and inclusion ordering are significantly violated by the JLS grammars, as our case study shows.

The motivation of our work and its significance is not limited to the mere discovery of bugs in the Java standard. In fact, some of these bugs are known by now—based on informal grammar inspection or other brute-force methods. There are various accounts that have identified or fixed bugs in the JLS grammars or, in fact, in grammars that were derived from the JLS in some manner. We refer to the work of Richard Bosworth as a particularly operational account: `http://www.cmis.brighton.ac.uk/staff/rnb/bosware/javaSyntax/syntaxV2.html`. It is a clear list of bugs which is also endorsed by SUN: `http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=6442525`. We refer to this list as "known bugs" in our process.

The significance is amplified by two additional arguments. First, we provide a simple and mechanised process that is guaranteed to reveal accidental or intended differences between grammars. Second, we are able to represent the differences in a precise, operational and accessible manner—by means of grammar transformations. In different terms, we are essentially able to prove (or disprove) the equivalence for two grammars, and we can afford different grammars and different versions because we can effectively relate them.

The complete JLS effort including all the involved sources, transformations, results, and tools is publicly available through `http://slps.sf.net/` [263]; see `topics/convergence/java` in particular.

However, it is instructive to wonder why the importance of JLS combined with the scrutiny that went into its preparation still let inconsistencies go unnoticed. We call two challenges to account. First, we note that language equivalence and inclusion is not amenable to any straightforward check; in fact, it is undecidable for context-free grammars. Second, grammar design and evolution is a manual process in practice: grammar engineers design and evolve grammars, as they see fit. They may use simple tools to check the grammar for basic well-formedness or grammar-class compliance. They may also test a parser derived from the grammar. However, such measures cannot guarantee the expected properties for (relaxed) language equivalence and inclusion; oversights happen all too easily (as our case study shows). Our (refined) method of grammar convergence addresses both challenges.

5.2 Contributions

- ◇ We have recovered nontrivial relationships between sized grammars. (That is, we show that the grammars are equivalent modulo well-defined transformations.) Compared to related work on agile parsing [43] and grammar recovery [58, 132, 164], *two* grammars are given a priori as opposed to the derivation of a new grammar from a baseline say by customisation, correction, completion, or restructuring.

- ◇ We have implemented a mechanised and measurable and reproducible process for grammar convergence. Compared to our initial work [166], the process consists of well-defined phases and progress can be effectively tracked in terms of nominal and structural differences between the grammars at hand.

- ◇ We have worked out a comprehensive operator suite for grammar transformation that substantially extends our previous work on the subject.

A condensed version of this case study description was published as *Recovering Grammar Relationships for the Java Language Specification* [167] in the proceedings of the 9th IEEE International Working Conference on Source Code Analysis and Manipulation. The paper received the Best Paper Award. A significantly extended version is being printed in the special issue of Software Quality Journal [168]. This chapter is a further extension of both papers. We included important information on the extractor (section 5.4) and displayed detailed tables such as Table 5.4 and Table 5.6. The tables contain important information about the extraction process and the recovered relationship; it was mentioned in the paper but not included there due to space constraints. An application of our grammar convergence method to various versions of the Java Language Specification is publicly available through the *Software Language Processing Suite* [263], including all Java grammars being displayed as a part of *SLPS Grammar Zoo* [260]. As stated above, Prof. Dr. Ralf Lämmel (Software Languages Team, Universität Koblenz) was a co-author of [167, 168].

5.3 The JLS corpus

We recall that each version of the JLS provides a grammar that is optimised for readability (*read1–read3* in Figure 5.1), and another one that is intended to serve as a basis for implementation (*impl1–impl3* in Figure 5.1). We also refer to these grammars as being "more readable" or "more implementable". These notions are not strongly defined, but one can think, for example, of *left factoring* as being used in the more implementable grammars (but not in the more readable grammars).

In the following sections we gather basic knowledge about the grammars.

5.3.1 JLS1

According to [77, §19], the *impl1* grammar "*has been mechanically checked to insure that it is LALR(1)*". The *read1* grammar is also referred to as "syntactic grammar" and its relationship to *read1* is briefly described as follows [77, §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 *read1* and *impl1* are said to be similar, a more precise relationship is not in sight though.

	Grammar class	Iteration style
impl1	LALR(1)	left-recursive
read1	none	left-recursive
impl2	unclear	EBNF
read2	none	left-recursive
impl3	"nearly" LL(k)	EBNF
read3	none	left-recursive

Table 5.1: Basic properties of the JLS grammars.

5.3.2 JLS2

According to [78, "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 *read1* and *read2* are similar (modulo the extensions to be expected), whereas surprisingly, *impl1* and *impl2* appeared as different developments. Also, the LALR(1) claim for *impl1* is not matched for *impl2*, but instead the following, less precise statement is made [78, §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.*"

5.3.3 JLS3

JLS3 extends JLS2 in numerous ways [79, Preface]: "*Generics, annotations, asserts, autoboxing and unboxing, enum types, foreach loops, variable arity methods and static imports have all been added to the language 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 *read2* and *read3* are similar, just as much as *impl2* and *impl3* (modulo the extensions to be expected). No definitive grammar-class claim is made, but an approximation thereof [79, §18] that suggests that *impl3* has definitely departed from LALR(1), i.e., *impl3* is said to be "*not an LL(1) grammar, though in many cases it minimises the necessary look ahead.*"

5.3.4 Grammar data

In addition to grammar class claims for the JLS grammars we have also recorded iteration styles during cursory examination; see Table 5.1. This data already clarifies 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 recover the relationships between the different grammars by effective transformations.

	Number of Productions	Number of Nonterminals	Number of Tops	Number of Bottoms
impl1	282	135	1	7
read1	315	148	1	9
impl2	185	80	6	11
read2	346	151	1	11
impl3	245	114	2	12
read3	435	197	3	14

Table 5.2: Basic metrics of the JLS grammars.

Table 5.2 measures simple grammar metrics for the various JLS grammars. A *top nonterminal* is a nonterminal that is defined but never used; a *bottom nonterminal* is a nonterminal that is used but never defined; we adopt these terms from [164, 219]. We have eventually understood that the major differences between the numbers of productions and nonterminals for the two grammars of any given version is mainly implied by the different grammar classes and iteration styles. The decrease of numbers for the step from *impl1* to *impl2* 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. The *difference in numbers of bottom-nonterminals could 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.

Now the plan for Java grammar convergence should be obvious from Figure 5.1 and the observations made so far. 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 three 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 three "more readable" grammars are similar, it makes sense to attempt a redundant path, i.e., to converge *read1* ... *read3* without any influence from *impl1* ... *impl3*. Hence, there is another cascade with targets *read12* and *read123*. 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.

5.4 Automated grammar extraction

A JLS document is basically a structured text document with embedded grammar sections. In fact, the more readable grammar is developed throughout the document where the more implementable grammar is given, *en bloc*, in a late section—a de-facto appendix.

```
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 ...
Indent:
     ... indentation ...
```

Listing 5.1: 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.

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 have opted for the HTML format here. The grammar format slightly varies across the different JLS grammars and versions; we had to collect formatting rules from different documents and sections—in particular from [77, 78, 79, §2.4] and [78, 79, §18].

In order to deal with irregularities of the input format, such as liberal use of markup tags, misleading indentation, duplicate definitions as well as numerous smaller issues, we needed to design and implement a non-classic parser to extract and analyze the grammar segments of the documents and to perform a recovery. About 700 fixes were performed that way, as can be seen from Table 5.4. The insides of the extraction parser are explained below in detail.

5.4.1 Assumed grammar format

Grammar fragments are hosted by `<pre>...</pre>` blocks in the JLS documents. According to [77, 78, 79, §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 [78, 79, §18]: $[x]$ denotes zero or one occurrences of x; $\{x\}$ denotes zero or more occurrences of x; $x_1|\cdots|x_n$ forms a choice over the x_i. The JLS documents consistently suffice with "$\star$" lists (zero or more occurrences); there are no uses of "$+$" lists. Refer to Listing 5.1 for a summary.

We should also mention line continuation; it allows to spread one alternative over

several lines [79, §2.4]: "*A very long right-hand side may be continued on a second line by substantially indenting this second line*".

Example 5.1 *A grammar fragment as of [78, §4.2]:*

```
<i>NumericType:
      IntegralType
      FloatingPointType

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

It should be parsed as:

```
NumericType:
      IntegralType
      FloatingPointType

IntegralType:
      "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.*

Example 5.2 *The only place where the subscript "opt" is capitalised [79, §4.5.1]:*

```
Wildcard:
? WildcardBounds<sub>Opt</sub>
```

5.4.2 Phase 1 — Preprocessing

The tiny Example 5.1 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 behaviour:

- ◇ Input: the `<pre>...</pre>` blocks.
- ◇ Output: a dictionary
  - – Keys: Left-hand side nonterminals

	italic	`fixed`	default
Alphanumeric	nonterminal (2341)	terminal (173)	assumed terminal (194)
|	metasymbol (2)	terminal (2)	assumed metasymbol (29)
{,},[,],(,)	metasymbol (708)	terminal (174)	assumed terminal (200)
otherwise	terminal (198)	terminal (165)	terminal (205)

Table 5.3: Decision table of the extractor's scanner. Classes of strings are rows, scanner states are columns.

- 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 recognises 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. The decision table of the scanner is presented in Table **??** along with the number of times each decision is taken for all JLS documents.

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 metasymbol) 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.

5.4.3 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 in Python by regular expression matching.

Rule 5.1 (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, as in Example 5.3. If there is still a closing parenthesis that cannot be matched, then it is dropped, as in Example 5.4. We have not seen the case of an opening parenthesis to remain unmatched, but for the sake of completeness implemented the symmetric part of the rule nevertheless.*

Example 5.3 *A grammar production from [79, §18.1]: the closing bracket and the closing parenthesis need to be converted to metasymbols to match the opening ones:*

```
TypeArgument:
      Type
      "?" [ ( "extends" | "super" ")" "Type" "]"
```

Example 5.4 *A grammar production from [78, §18.1] and [79, §18.1]: non-matching square bracket needs to be removed:*

```
Expression:
      Expression1 [ AssignmentOperator Expression1 ] ]
```

Rule 5.2 (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, as in Example 5.5.*

(b) When "[" and "]" occur next to each other as BNF symbols, then they are converted to terminals, as in Example 5.6.

(c) When "{" and "}" occur next to each other as BNF symbols, then they are converted to terminals. (Not encountered so far, implemented for the sake of consistency).

(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, as in Example 5.7.

Example 5.5 *A grammar production from [78, §15.22]: there is no group, so the bar here is not a metasymbol, but a terminal:*

```
InclusiveOrExpression:
      ExclusiveOrExpression
      InclusiveOrExpression | ExclusiveOrExpression
```

Example 5.6 *A grammar production from [78, §8.3]: there is nothing to be made optional, so the square brackets here are not metasymbols, but terminals:*

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

Example 5.7 *A grammar production from [78, §14.19] and [79, §18.1]: there is no choice inside the group so the parentheses here are not metasymbols, but terminals:*

```
CatchClause:
      "catch" ( FormalParameter ) Block
```

Rule 5.3 (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, as in Example 5.8 and Example 5.9.*

Example 5.8 *Multiple terminals to compose [77, §19.11]:*

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

Example 5.9 *Multiple nonterminals to compose [77, §14.9]:*

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

Rule 5.4 (Decompose compound terminals) *When a terminal consists of an alphanumeric prefix, followed by ".", possibly followed by a postfix, then the terminal is taken apart into several ones, as in Example 5.10.*

Example 5.10 *Consider this phrase [78, §15.9]:*

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

The decomposition results in the following:

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

Rule 5.5 (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, as in Example 5.11.*

Example 5.11 *A grammar production from [78, §14.11]:* `default` *needs to be converted to a terminal:*

```
SwitchLabel:
      </em>case<em> ConstantExpression :
      default :
```

The same error is present in the later version of the specification [79, §14.11]:

```
SwitchLabel:</em>
      case <em>ConstantExpression </em>:
      case <em>EnumConstantName </em>:<em>
      default :
```

Note that in JLS3 the same production was changed (another alternative was added), and somewhere on the way a problem with ":" was fixed, but not the one with "`default`*".*

Rule 5.6 (Terminal to nonterminal) *Alphanumeric terminals that start in upper case, and are defined by the grammar (when considered as nonterminals) are converted, as in Example 5.12.*

Example 5.12 *A grammar production from [78, §7.5]:*

```
<em>ImportDeclaration</em>:
      SingleTypeImportDeclaration
      TypeImportOnDemandDeclaration
```

The decisive definitions are found in [78, §7.5.1] and [78, §7.5.2]:

```
SingleTypeImportDeclaration:
      "import" TypeName ";"
TypeImportOnDemandDeclaration:
      "import" PackageOrTypeName "." "*" ";"
```

Rule 5.7 (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 5.13 *Consider again the result of Example 5.10:*

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

After recovery it will be parsed as:

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

These are all the rules that have stabilised over the project's duration. Several other rules where 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.

	impl1	impl2	impl3	read1	read2	read3	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
◦ Match parentheses	0	3	6	0	0	0	9
◦ Metasymbol to terminal	0	1	7	0	27	7	42
◦ Merge adjacent symbols	1	0	0	1	1	0	3
◦ Split compound symbol	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
◦ 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

Table 5.4: Irregularities resolved by grammar extraction.

5.4.4 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 5.14 *Recall the following definition from Example 5.6 [78, §8.3]:*

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

The same definition appears elsewhere in the document, even though the markup is different, but these differences are already neutralised during phase 1 [78, §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 5.2.b because `[ ]` *must be converted to terminals.*

5.4.5 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; see Listing 5.1. 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 [166].

5.4.6 Extraction data

Table 5.4 summarises the frequency of using recovery rules, handling "unusual" continuation lines (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), and removal of doubles. The extractor has fixed 669 problems that otherwise would have prevented straightforward parsing to succeed with extraction, or implied loss of information, or triggered substantial grammar transformations.

5.5 The convergence graph

As it has ben explained in the previous chapter, the central idea of grammar convergence [166] is to extract grammars from diverse software artifacts, and to discover and represent the relationships between the grammars by chains of transformation steps that make the grammars structurally equal. We have defined the core ingredients of the method and now apply it to the JLS.

Grammar convergence always starts from the grammars that were extracted from the given software artifacts, to which we refer as *source grammars* or *sources* subsequently. In the present JLS study, we face 6 sources; we use *read1–read3* to refer to the "more readable" grammars, and *impl1–impl3* to refer to the "more implementable" grammars. It is reasonable to relate grammars through an additional grammar of which we think as the common denominator of the original grammars. We refer to such additional grammars as *targets*. The "distance" between source and target grammars may differ. In fact, it is not unusual, that one source—modulo minor transformations only—serves as common denominator.

The idea of the common denominator can be generalized such that we actually devise a tree of grammars with transformations as the edges. (We use arrows to express the direction of the transformation, and hence the trees appear inverted, when compared to common sense of drawing trees.) That is, the root of the tree (a target) is the common denominator of all grammars, but there may be additional intermediate targets that already serve as common denominators for some of the grammars. The source grammars are the leaves of such a tree.

Figure 5.1 showed the "convergence tree" for the present JLS case study. The original grammars from the JLS documents are located at the top. The tree states that the two grammars per JLS version are "converged to" a common denominator (see the nodes *jls1–3* in the figure), and all three versions are further "converged" to account for inter-version differences—the extensions to the Java language in particular (see the nodes *jls12* and *jls123* as well as *read12* and *read123* in the figure). For the JLS we use a binary tree, which means that we always limit the focus to two grammars, and hence a cascade is needed, if more than two grammars need to be converged.

When deriving *jls1–3*, we favor the "more implementable" grammar as the target of convergence, i.e., as the common denominator—except that some corrections may need to be applied, or some minimum restructuring is applied for the sake a more favorable grammar structure. 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 readable grammars *read1–3* in another cascade (see the nodes *read12* and *read123*) as a sort of sanity check. It turns out that *read1–3* are structurally quite similar, and accordingly, the additional cascade requires little effort.

5.6 Grammar transformation

In this section, we illustrate intended and accidental differences between the JLS grammars in more detail and show the grammar transformations that neutralise those differences. At the same time, we provide an overview of the major operators that are needed for the transformation of concrete syntax definitions in the context of grammar convergence. We distinguish semantics-preserving/-increasing/-decreasing and -editing operators [166] where semantics always refers to the language generated by the grammar (when considered as a set of strings).

5.6.1 Semantics-preserving operators

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. "$\star$"). We also say that all these operators serve *grammar refactoring*.

Several transformation operators serve disciplined "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.

Example 5.15 (*factor* and *massage* transformations)

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

In *read2*, there are distinct alternatives for blocks vs. static blocks. In contrast, in *impl2*, 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".

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 "$\star$" and "$+$" can be requested in this manner. We call the latter step "deyaccification" [132, 154].

Like the previous transformation sample, the next one is taken from a refactoring script that aligns *read2* and *impl2*. The JLS case study involves many hundreds of such small refactoring steps; see section 5.8.

Example 5.16 (*deyaccify* and *inline* transformations)

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

In *read2*, recursion-based style of iteration is used. For instance, there is a recursively defined nonterminal *ClassBodyDeclarations* for lists of *ClassBodyDeclaration*. In contrast, in *impl2*, the list form "$\star$" is used. Deyaccification replaces the recursive definition of *ClassBodyDeclarations* by *ClassBodyDeclaration*$^+$. The nonterminal *ClassBodyDeclarations* is no longer needed, and hence inlined. The list of declarations was optional, and hence "$+$" and "?" can be simplified to "$\star$".

5.6.2 Semantics-in/decreasing operators

There are operators to *widen* and *narrow* occurrence constraints (e.g., to change "$+$" to "$\star$" 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*). One can also want optional symbols (i.e., those with "?" or "$\star$") to *appear* or *disappear*.

Example 5.17 (Widening an occurrence constraint)

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

This transformation is part of a script that captures the delta between JLS1 and JLS2. The particular widening step enables *instance* initialisers in class bodies (where only *static* initialisers 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 5.18 (Adding an alternative)

```
addV(
 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 we seek relationships between grammars of different versions, then semantics-increasing/-decreasing transformations are clearly to be expected. As a matter of discipline, we prefer to describe the delta by a semantic-increasing transformation to map a version to its successor version (as opposed to the inverse direction). We speak of *grammar extension* in this case.

However, increase (or decrease) may also be needed when two grammars are essentially equivalent—except that one is more liberal than the other, merely as a concession to practicality of say parser development. We speak of *grammar relaxation* in this case. In the JLS case, the different purposes of the grammars (to be more or less readable or implementable respectively.) imply the need for relaxation. Similar issues arise with relationships between abstract and concrete syntaxes [166].

Finally, two grammars may differ (with regard to the generated language) in a manner that is purely accidental (read as “incorrect”). We speak of (transformations for) *grammar correction* in this case. Corrections may be expressed in terms of semantics-increasing/-decreasing operators. (Otherwise we have to use less disciplined operators; see below.)

Example 5.19 (Grammar relaxation) *The BGF snippets in this example are deliberately pretty-printed as horizontal productions for the sake of readability. In reality the extractor produces only vertical ones as usual.*

impl2

```
Modifier:
      "public" | "protected" | "private" | "static" | "abstract"
            | "final" | "native" | "synchronized" | "transient"
            | "volatile" | "strictfp"
```

read2

```
ClassModifier:
      "public" | "protected" | "private" | "abstract" | "static"
            | "final" | "strictfp"
ConstantModifier:
      "public" | "static" | "final"
ConstructorModifier:
      "public" | "protected" | "private"
InterfaceModifier:
      "public" | "protected" | "private" | "abstract" | "static"
            | "strictfp"
AbstractMethodModifier:
      "public" | "abstract"
MethodModifier:
      "public" | "protected" | "private" | "abstract" | "static"
            | "final" | "synchronized" | "native" | "strictfp"
FieldModifier:
      "public" | "protected" | "private" | "static" | "final" |
           "transient" | "volatile"
```

In *impl2*, there is only one category of (arbitrary) modifiers. In contrast, in *read2*, there are various precise categories of modifiers for classes, fields, interfaces and methods. Accordingly, the *impl2* grammar is more permissive as far as modifiers are concerned. We omit the neutralising transformation.

We suggest that a language specification should explicitly call out relaxations so that they are not confused with corrections. Neither relaxation and corrections must be confused with extension (in the sense of evolution).

5.6.3 Semantics-revising operators

There are operators to *undefine* a nonterminal (i.e., to abandon its definition), to *replace* a grammar expression in an 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 ⟨b⟩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 5.20 (Correcting statement syntax in *impl2*)

```
inject (
 Statement:
      "break" Identifier? ⟨";"⟩
);
```

The production for the *break* statement lacks the semicolon which is injected accordingly (left unnoticed in Bosworth's bug list, but obvious when converging with *read2*).

Example 5.21 (Correcting expression syntax)

Incorrect expression syntax in *impl2* and *impl3*

```
Expression2:
      Expression3 Expression2Rest?
Expression2Rest:
      (Infixop Expression3)*
Expression2Rest:
      Expression3 "instanceof" Type
```

Language-revising transformation

```
project (
 Expression2Rest:
      ⟨Expression3⟩ "instanceof" Type
);
```

Corrected expression syntax

```
Expression2:
      Expression3 Expression2Rest?
Expression2Rest:
      (Infixop Expression3)*
Expression2Rest:
      "instanceof" Type
```

The *impl2* and *impl3* grammars define the Java expression syntax by means of layers, i.e., there are several nonterminals *Expression1*, *Expression2*, ... for the different priorities. We are concerned with one layer here. The second rule for *Expression2Rest* contains an offending occurrence of *Expression3* which needs to be projected away. This issue was revealed by comparison with the *read2* and *read3* grammars (subject to prior refactoring), but also found in the "known bugs".

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 from representation anomalies of the HTML input used for extraction.

Example 5.22 (Extraction: post-processing for impl3)

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

The source format defines curly brackets to express iteration. However, in the example at hand they were meant as terminals, and were not recognised due to missing markup. The incorrect list construct is replaced accordingly.

Example 5.23 (Initial correction for impl2)

```
replace(
 Expr,
 Expression);
```

A misnamed nonterminal is found when examining the list of bottom nonterminals before the convergence process starts.

5.6.4 Grammar refactoring

When grammars are not structurally equivalent, the generated languages of course may still be. The property of generating the same 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 example 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. Extraction basically introduces a new nonterminal to capture an existing structure, and then uses the new nonterminal immediately in a fold step [154].
- ◇ *De-/yaccification*: grammars may differ with regard to style of iteration. Deyaccification transforms YACC-style of recursion to EBNF-style lists [154]. 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 5.24 *Let us encounter refactoring in Example 5.19. 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 normalisation, 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 to rename the nonterminals for modifiers and sequences thereof to* `Modifier` *and* `ModifiersOpt`*. Renaming is a form of refactoring, too.*

	jls1	**jls12**	**jls123**	**jls2**	**jls3**	**read12**	**read123**	**Total**
Number of lines	682	5116	2847	6772	10715	1639	3082	30853
Number of transformations	67	298	111	395	544	77	135	1627
○ semantics-preserving	45	239	80	283	381	31	78	1137
○ semantics-increasing or -decreasing	22	58	31	102	150	39	53	455
○ semantics-revising	—	1	—	10	13	7	4	35
Number of steps	8	9	8	22	26	6	6	85
Number of issues	8	25	17	40	53	32	44	219
○ recoveries	—	—	—	7	8	7	4	26
○ corrections	8	5	—	45	53	14	14	139
○ extensions	—	17	14	1	—	14	29	75
○ optimisations	3	6	3	9	19	1	—	41
Preparation phase	1	—	—	15	24	11	14	65
○ Known bugs (Example 5.21)	—	—	—	1	11	—	4	16
○ Post-extraction (Example 5.22)	—	—	—	7	8	7	5	27
○ Initial correction (Example 5.23)	1	—	—	7	5	4	5	22
Resolution phase	21	59	31	97	139	35	43	425
○ Extension (Example 5.18)	—	17	26	—	—	31	38	112
○ Relaxation (Example 5.19)	18	39	5	75	112	—	2	251
○ Correction (Example 5.20)	3	3	—	22	27	4	3	62

Table 5.5: Transformation of the JLS grammars: effort metrics and categorisation.

	jls1	**jls12**	**jls123**	**jls2**	**jls3**	**read12**	**read123**	**Total**
○ *rename*	9	4	2	9	10	—	2	36
○ *reroot*	2	—	—	2	2	2	1	9
○ *unfold*	1	10	8	11	13	2	3	48
○ *fold*	4	11	4	11	13	2	5	50
○ *inline*	3	67	8	71	100	—	1	250
○ *extract*	—	17	5	18	30	—	5	75
○ *chain*	1	—	2	—	—	1	4	8
○ *massage*	2	13	—	15	32	5	3	70
○ *distribute*	3	4	2	3	6	—	—	18
○ *factor*	1	7	3	5	24	3	1	44
○ *deyaccify*	2	20	—	25	33	4	3	87
○ *yaccify*	—	—	—	—	1	—	1	2
○ *eliminate*	1	8	1	14	22	—	—	46
○ *introduce*	—	1	30	4	13	3	34	85
○ *import*	—	—	2	—	—	—	1	3
○ *vertical*	5	7	7	8	22	5	8	62
○ *horizontal*	4	19	5	17	31	4	4	84
○ *add*	1	14	13	7	20	28	20	103
○ *appear*	—	8	11	8	25	2	17	71
○ *widen*	1	3	—	1	8	1	3	17
○ *upgrade*	—	8	—	14	20	2	2	46
○ *unite*	18	2	—	18	21	5	4	68
○ *remove*	—	10	1	11	18	—	1	41
○ *disappear*	—	7	4	11	11	—	—	33
○ *narrow*	—	—	1	—	4	—	—	5
○ *downgrade*	—	2	—	8	3	—	—	13
○ *define*	—	6	—	4	9	1	6	26
○ *undefine*	—	11	—	13	3	—	—	27
○ *redefine*	—	3	—	8	7	6	2	26
○ *inject*	—	—	—	2	4	—	1	7
○ *project*	—	1	—	1	2	—	—	4
○ *replace*	3	1	2	3	6	1	1	17
○ *unlabel*	—	—	—	—	—	—	2	2

Table 5.6: XBGF operators usage for JLS convergence.

5.7 Grammar convergence phases

5.7.1 Preparation phase: semantic error recovery

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. Every such irregularity is fixed by a designated transformation in a way that was explained in Example 5.22.

Interestingly enough, the original Java Language Specification [77] turned out to be marked up the most accurately — not only it had the least amount of errors (recall Table 5.4), but also all of those problems have been solved by applying all the rule set from the previous section. Manually programmed transformations were required for *read2*, *read3* and *impl3*.

5.7.2 Preparation phase: fixing known bugs

Richard Bosworth's Java specification bug report (`http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=6442525`) contains 18 issues that are immediately applicable to *impl3* and sometimes even to earlier versions of the standard. Example 5.25 demonstrates the way each of those issues was mapped to its XBGF representation. Among the previously presented examples Example 5.21 also belonged to this category.

Example 5.25 *Consider the following item from Richard Bosworth's list:*

> *18.1 does not permit the obsolescent array syntax in a method declaration of an annotation type:*
>
> ```
> AnnotationMethodRest:
>     () {[]} [DefaultValue]
> ```
>
> *where {[]} means 0 or more []'s. [DefaultValue] means 0 or 1 DefaultValue.*

After examining [79, §18.1], we find the production:

```
AnnotationMethodRest:
      "(" ")" DefaultValue?
```

In order to introduce the square brackets are required, we need a following transformation:

```
appear (
 AnnotationMethodRest:
      "(" ")" ⟨("[" "]")*⟩ DefaultValue?
);
```

5.7.3 Preparation phase: initial correction

Example 5.23 presented a good situation when initial correction—i.e., the one that happens before any grammar comparison—is possible. Lacking definitions, mistyped nonterminal names, superfluous definitions and similar problems can be spotted upon cursory examination of the grammar by a language engineer or after running some easily set up analyses like listing top and bottom nonterminals. As we predicted in [143], there were cases of nonterminals with "intuitive" names with lacking definitions (e.g., `ForInitOpt` and `ForUpdateOpt` in *impl2*).

Example 5.23 presents and explains another initial correction scenario.

Example 5.26 *By examining the list of bottom nonterminals in* read2*, it was discovered that* `ClassName` *was left undefined. Since* impl2 *never uses any nonterminal symbol that has the same name or could have been aligned with* `ClassName`*, we search for the same nonterminal in different versions of the Java Language Specification—it helped in other cases, but this time it is in vain. Then we open §6.5 of [78] to see similar definitions of* `PackageName`*,* `TypeName`*,* `ExpressionName`*, etc. All of them are in fact the same* `QualifiedIdentifier` *that we remember from C# grammar in chapter 3, but introduced in a left recursive way:*

```
PackageName:
      Identifier
      PackageName . Identifier
TypeName:
      Identifier
      PackageOrTypeName . Identifier
ExpressionName:
      Identifier
      AmbiguousName . Identifier
MethodName:
      Identifier
      AmbiguousName . Identifier
PackageOrTypeName:
      Identifier
      PackageOrTypeName . Identifier
AmbiguousName:
      Identifier
      AmbiguousName . Identifier
```

We use ***define*** *transformation to introduce a similar definition for* `ClassName` *to both* read2 *and* read3 *that happens to have the same problem:*

```
define (
 ClassName:
      Identifier
      AmbiguousName "." Identifier
);
```

If we were going beyond grammar convergence, one of the first steps in grammar beautification would be removal of all the above mentioned nonterminals in favour of

`QualifiedIdentifier` *as it was done for C#.*

5.7.4 Nominal matching phase

After all the preparatory actions are completed, one can start renaming and (un)folding to align syntactic categories of the converging grammars. Example 5.27 shows how it can be done.

Example 5.27 *We recall from Example 5.19 that modifiers are defined universally in* implX *part and separately for each case in* readX*. In terms of XBGF this will be achieved by a series of* ***unite*** *transformations. However, they all need a base nonterminal to unite* to *since there is no* `Modifier` *in* readX *grammars at all. We decide to use* `ClassModifier` *for that role:*

```
renameN(ClassModifier, Modifier);
renameN(ClassModifiers, Modifiers);
```

5.7.5 Structural matching phase

There are far too many refactorings to show them all here and even an endeavour to classify them into a reasonable number of groups seems unfeasible. Example 5.15 and Example 5.16 were already presented in the section above, let them serve the demonstration role. Example 5.28 continues Example 5.27 with all refactorings that can help convergence. Structural matching always continues until it is not possible anymore to bring the grammars together with semantic-preserving transformations.

Example 5.28 *Renaming alone is not enough to prepare us for uniting the nonterminals, we still need to* ***deyaccify*** *and* ***inline*** *them.*

```
deyaccify(Modifiers);
deyaccify(AbstractMethodModifiers);
deyaccify(InterfaceModifiers);
deyaccify(MethodModifiers);
deyaccify(FieldModifiers);
deyaccify(ConstructorModifiers);
deyaccify(ConstantModifiers);
inline(Modifiers);
inline(AbstractMethodModifiers);
inline(InterfaceModifiers);
inline(MethodModifiers);
inline(FieldModifiers);
inline(ConstructorModifiers);
inline(ConstantModifiers);
```

5.7.6 Resolution phase: extension

When an unresolved difference can be explained by language evolution, it is the simplest case for resolution phase actions. Example 5.17 and Example 5.18 were discussed above

and give a good impression of what to expect from grammar-lengthening steps. Note that there is no reduction phase anywhere in the Java case study nor in the general convergence recipe in section 4.7—it would have been easier to converge grammars by stripping non-matching parts away, but the result with such strategy would be much less meaningful.

5.7.7 Resolution phase: relaxation

Usually the grammar meant for implementation is somewhat more permissive than the one optimised for readability, as shown in Example 5.19. Example 5.29 finishes the work of Example 5.28 and all preceding transformation steps.

Example 5.29 *Finally the nonterminals are united. Note that we still need to run two refactoring transformations after the generality issue is resolved:*

```
unite(InterfaceModifier, Modifier);
unite(MethodModifier, Modifier);
unite(FieldModifier, Modifier);
unite(ConstructorModifier, Modifier);
unite(AbstractMethodModifier, Modifier);
unite(ConstantModifier, Modifier);
massage(
 Modifier+?,
 Modifier*);
extract(
 ModifiersOpt:
      Modifier*
);
```

5.7.8 Resolution phase: correction

Everything that is still unresolved and cannot be explained by language evolution nor permissiveness considerations, is a difference that was not meant to be—i.e., its removal is actually a correction of a newly found bug. Example 5.20 was already shown above as a bug-fixing scenario, see also Example 5.30 for a less obvious one.

Example 5.30 ***read3*** *contained the following production [79, §10.6]:*

```
ArrayInitializer:
      "{" VariableInitializers? ","? "}"
```

impl3 *contained a different one [79, §18] that after post-extraction correction looks as follows:*

```
ArrayInitializer:
      "{" (VariableInitializer ("," VariableInitializer)* ","?)? "}"
```

After unfolding `VariableInitializers`*, the former production is almost identical to the latter. However, the difference with the comma being inside the parenthesis or outside them, cannot be resolved with refactoring (easy to see with a counter-example: a*

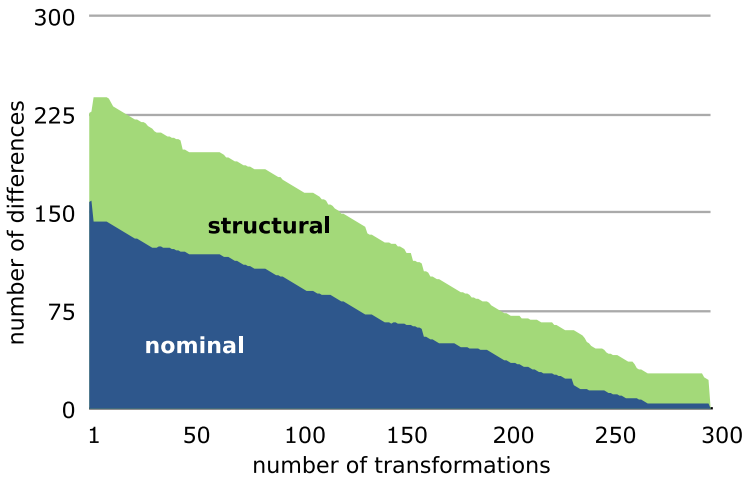


Figure 5.2: Difference reduction for *read2* towards the convergence target *jls2* in the convergence tree of Figure 5.1.

string "`"{" "," "}"`" is a valid expression that can be derived from the first definition but not from the second one). The following transformations resolve the issue:

```
disappear(
 ArrayInitializer:
     "{" (VariableInitializer ("," VariableInitializer)*)? ⟨","?⟩ "}"
);
appear(
 ArrayInitializer:
     "{" (VariableInitializer ("," VariableInitializer)* ⟨","?⟩)? "}"
);
```

5.8 Measuring grammars and transformations

As we recall from section 4.7, the guiding principle for grammar convergence is to consistently reduce the number of grammar differences throughout the two matching phases as well as the final resolution phase. Figure 5.2 illustrates this principle for one specific JLS grammar and the related convergence. The figure also visualises that nominal differences tend to be resolved earlier than structural differences.

Table 5.7 shows the same, simple metrics for the derived grammars as we originally presented for the leaves of the convergence tree; see Table 5.2. 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 *impl1–3*. Similar correlations hold for the "inter-version" grammars in the rest of the table.

	Number of Productions	Number of Nonterminals	Number of Tops	Number of Bottoms
jls1	278	132	1	7
jls2	178	75	1	7
jls3	236	109	1	7
jls12	178	75	1	7
jls123	236	109	1	7
read12	345	152	1	7
read123	438	201	1	7

Table 5.7: Simple metrics for the derived JLS grammars.

Table 5.5 measures the extraction effort and the involved grammar transformations. This information was obtained in an automated manner but it relies on some amount of semantic annotation of the transformations for the classifications and phases.

The number of transformations directly refers to the number of *applications of transformation operators*. Table 5.6 shows that 33 different operators are used in the JLS case; most of them were introduced in section 5.6. 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.

Let us make a few observations. For instance, one can observe that relaxation transformations indeed occur when a more readable and a more implementable grammar are converged. Further, one can observe that the overall transformation effort is particularly high for *jls12*—which signifies the fact mentioned above that *impl1* and *impl2* appear to be different developments. Finally, we have made an effort to incorporate SUN's bug list into the picture (see subsection 5.7.2). We note that some of the known bugs equally occur in both the more readable and the more implementable grammar, in which case we cannot even discover them by grammar convergence.

5.9 Related work

We organise the related work discussion in the following manner:

- ◇ grammar recovery (including grammar inference);
- ◇ programmable grammar transformations;
- ◇ other grammar engineering work;
- ◇ coupled transformations of grammar- or schema- or metamodel-like artefacts and grammar- or schema- or metamodel-dependent artefacts;
- ◇ comparison (including matching) of schemas or metamodels.

5.9.1 Grammar recovery

The main objective of the JLS study is to discover grammar relationships, but an "important byproduct" of the study is a consolidated Java grammar. Hence, this particular instance of grammar convergence (perhaps more than grammar convergence in general) relates strongly to other efforts on grammar recovery. This topic has seen substantial interest over the last 10 years because of the need for grammars in various software engineering scenarios. We categorise this work in the following.

5.9.1.1 Recovery option 1: Parser-based testing and improvement cycle

A by now classical approach to grammar recovery is to start from some sort of documentation that contains a raw grammar, which can be extracted, and then to improve the raw grammar through parser-based testing until all sources of interest can be parsed (such as test programs, or entire software projects) [6, 132, 163, 164, 219]. The actual improvement steps may be carried out manually [6, 132, 219] or by means of programmable grammar transformations [163, 164], as discussed in more detail in subsection 5.9.2.

The JLS study, in particular, and the basic paradigm of grammar convergence, in general, do not involve parser-based testing. Instead, the similarity between 2+ given grammars is used as the criterion for possibly improving correctness. Of course, it would be a viable scenario to actually try deriving a useful parser description from the converged Java grammar, and if additional problems were found, then the parser-based testing and improvement cycle of grammar recovery may be applied.

5.9.1.2 Recovery option 2: Grammar recovery from ASTs

Generally, raw grammars (as discussed above) may also be extracted from compilers. This is relatively straightforward, if the compiler uses a parser description to implement the parser. [58, 150] present another option, which relies on access to the parse trees or ASTs of a compiler. A grammar can be extracted from the ASTs for given sample programs. This approach is specifically meant to help with the recovery of language dialects for which precise grammars are often missing. In order to derive the grammar for the concrete syntax, one must discover the mapping between AST schema and concrete syntax. To this end, the approach also involves some verification infrastructure. If we assume that a baseline grammar is available (as opposed to a grammar for the specific dialect at hand), then grammar convergence may also be useful in providing the mapping between AST schema and concrete syntax.

5.9.1.3 Recovery option 3: Grammar inference

Different authors have approached grammar recovery for software languages through grammar inference techniques [54, 55, 56, 57, 181, 205, 206, 233]. Inference relies on language samples, typically on both positive and negative examples. Different inference scenarios have been addressed. [181, 233] infer more or less complete grammars, which is a very difficult problem. The approach applies to small languages, e.g., small domain-specific languages. [205, 206] begin from a baseline grammar, and infer modifications

to the grammar so that all sources of interest can be parsed. This search-based inference approach addresses the dialect problem in software engineering, where a grammar for the language of interest may be available, but not for the specific dialect at hand. Both of the approaches use *genetic algorithms*. [54, 55, 56, 57] use some mix of advanced parsing and inference techniques instead.

Just as in the case of Option 1, the approach uses parser-based testing as the correctness criterion, whereas grammar convergence leverages the similarity between 2+ given grammars as the criterion for possibly improving correctness. It is quite conceivable and interesting to combine grammar inference and grammar convergence. For instance, grammar inference techniques could be used to inform a semi-automatic grammar transformation approach. Also, it is interesting to understand whether transformation operators for convergence can usefully represent the modifications of the inference approach of [205, 206].

5.9.1.4 Recovery option 4: Special-purpose grammars

Rather than trying to recover the (full) grammar for a given language, one may also limit the recovery effort to specific samples, and more potently, to the specific purpose of the grammar. For instance, when the grammar is needed for a simple fact extractor, then there is no need to parse the full language, or to be fully aware of the dialect at hand. [186, 187] suggests so-called island grammars to only define as much syntactical structure as needed for the purpose and to liberally consume all other structure essentially as a token stream. [227] also pursues this approach specifically in the context of multilingual parsing. [192] also pursues a variant of special-purpose grammars, where sample programs are essentially modelled, and a grammar is computed from the samples. A disciplined and productivity-tuned, iterative approach is used to rapidly parse all the samples of interest. The approach also produces the right metamodel (object model) to represent parse trees tailored to the specific purpose at hand.

5.9.2 Programmable grammar transformations

Grammar convergence, and some forms of grammar recovery, but also some other software engineering problems rely on grammar transformations. In fact, we would like to limit the focus here to *programmable* grammar transformations as opposed to more hidden transformations as they may be performed implicitly by some software tools (such as a parser generator that removes left recursion, for example).

Cordy, Dean and collaborators have invented the notion of agile parsing [38, 41, 43] and the paradigm of grammar programming [42] in this context. Both concepts rely on language embedding of a grammar formalism into a programming language (TXL, in this case). Agile parsing basically suggests the customization of a baseline grammar for a specific use case (such as components for reverse engineering or re-engineering). The simpler programmable grammar transformations, which are sufficient for some scenarios, are **redefine** (to redefine a nonterminal), and **define** with the ability to extend the previous definition.

In [42], a range of additional grammar programming techniques are discussed, where some of these techniques may be naturally modelled as grammar transformations (or more

generally, as program transformations). These are the techniques: rule abstraction (so that grammar rules may be parametrised), grammar specialization (so that the semantics of specific uses cases can be incorporated into the grammar), grammar categorization (so that the resulting parser can effectively deal with context-free ambiguities), union grammars (so that one can have multiple grammars in the same namespace, perhaps even with a non-empty intersection), and markup (i.e., the use of markup syntax in combination with regular textual syntax).

In this thesis, we have been interested in operator suites for (programmable) grammar transformations, continuing the tradition of [154, 158, 165, 167]. The idea is to basically view the possible evolution of a grammar (along recovery or convergence) as a disciplined editing process such that each editing step is described in terms of an appropriate transformation operator. The use of an operator immediately documents a certain intention, and is subject to precondition checking—just like in other domains of program transformation. [249] has also suggested a small set of operators to specifically address the problem of computing abstract from concrete syntax.

5.9.2.1 The Amsterdam/Koblenz school of grammar transformation

To better understand the design space of programmable grammar transformations based on operator suites, we would like to compare several efforts; see Table 5.8 for an overview. The figure summarises known grammar transformation operators, and compares operator suites for grammar transformations:

VSC2 [154, 164]
The suite used for recovery of a Cobol grammar.[2]

FST [165]
A design experiment to define a comprehensive suite for SDF [239].[3]

GDK [149]
A suite that is part of a grammar-deployment infrastructure.[4]

GRK [158]
A suite that is part of an effort to reproduce our Cobol recovery case.[5]

XBGF
The suite of the present thesis; see chapter 7.[6]

A starred name in the figure (as in “restrict*”) means that the given operator covers the function at hand, but it is more general.

XBGF, the transformation language of the present paper, provides clearly the most comprehensive suite. There are a few empty cells in the XBGF column. Reasons for non-inclusion differ; either the operator is considered too low-level for the XBGF surface

[2] http://homepages.cwi.nl/~ralf/fme01
[3] http://www.cs.vu.nl/grammarware/fst
[4] http://gdk.sf.net
[5] http://slps.sf.net/grk
[6] http://slps.sf.net/xbgf

	VSC2	FST	GDK	GRK	XBGF
add a definition for a bottom nonterminal	resolve	resolve	resolve		define
add a new definition	introduce	introduce	introduce		introduce
add a new definition & fold it	extract			extract	extract
add a production to any nonterminal definition	include	include	include		addV
add a production to the grammar	add	add		add	
add alternatives to a choice					addH
change the order in a sequence	permute			permute	permute
do nothing	id	id			
give a production a label					designate
give a subexpression a selectable name					deanonymize
inject a nillable symbol					appear
inject a terminal symbol					concretize
inject symbols to a sequence					inject
inline a chain production					unchain
introduce a chain production					chain
introduce a reflexive chain production					detour
introduce several possibly interconnected definitions					import
merge two nonterminals					unite
merge two nonterminals if their definitions are equal		equate			equate
merge two nonterminals, one of which is bottom	unify	unify	unify		unite*
move a production across modules/sections		move			
perform factoring transformation			preserve*		factor
perform folding transformation	fold	fold	fold	fold	fold
perform massaging transformation	preserve	simplify	preserve		massage
perform narrowing transformation (as in x? to x)	restrict*	restrict*	restrict*		narrow
perform specialized automated factoring transformation					distribute
perform unfolding transformation	unfold	unfold	unfold	unfold	unfold
perform widening transformation (as in x+ to x*)	generalise	generalise	generalize		widen
project a nillable symbol	restrict*	restrict*	restrict*		disappear
project a terminal symbol					abstractize
project symbols from a sequence					project
remove a definition of a possibly used nonterminal	reject	reject	reject		undefine
remove a label from a production					unlabel
remove a production from the grammar	sub	sub			
remove a reflexive chain production					abridge
remove a selector in a subexpression					anonymize
remove alternatives from a choice					removeH
remove any part of a grammar	reset	reset			
remove unused definition	eliminate	eliminate	eliminate	reject	eliminate
removes one production of a nonterminal (not the last one)	exclude	exclude	exclude		removeV
rename a label					renameL
rename a nonterminal	rename	rename	rename		renameN
rename a nonterminal in a limited scope	substitute	substitute			replace*
rename a selector					renameS
replace a nonterminal with one of its definitions					downgrade
replace a nonterminal with ε	delete		delete		replace*
replace a terminal with another terminal					renameT
replace an expression by a nonterminal that can be evaluated to it					upgrade
replace any expression with another expression	replace	replace		replace	replace
replace iteration with left-associative equivalent					lassoc
replace iteration with recursion			preserve*		yaccify
replace iteration with right-associative equivalent					rassoc
replace recursion with iteration			preserve*		deyaccify
replace the current definition by a new one			redefine		redefine
separate one nonterminal into several (reverse of merge)	separate	seperate	separate		
terminate transformation sequence	fail	fail	write		dump
transpose a multi-production definition to the one with top-level choices					horizontal
transpose top-level choices to multiple productions					vertical
unfold & eliminate					inline

Table 5.8: Systematic comparison of grammar transformation operators provided by different frameworks

syntax (e.g., **substitute**, **reset**), or it is too low-level in the sense that all major application scenarios are covered by more specialised operators (e.g., **add**, **sub**), or it is not currently implementable (e.g., **move**—modules are not fully supported in our infrastructure), or it was simply not needed and perhaps debated so far (e.g., **delete**, **id**, **separate**, also known as FST's **seperate**; see the table for the typo).

There is generally a tension between the number of transformation operators vs. the achievable precision of a transformational program in terms of expressing intentions, and thereby enabling extra sanity checks by the transformation engine. Consider, for example, the line "add a production to the grammar". This low-level idiom may be used to **include** another production into an existing definition, or to add one or more productions in an effort to **resolve** a missing definition, or to **introduce** a definition for a so-far fresh nonterminal. In GRK, all these idioms are modelled by **add**, and hence no intentions are documented, and no extra checks can be performed automatically. In the case of XBGF, we have indeed tried to separate idioms aggressively. This approach also helps us with predicting the formal properties of each application of transformation operators (i.e., semantics-preserving, -increasing, -decreasing, -revising), and chains thereof.

5.9.3 Grammar engineering

Let us also discuss some additional related work on grammar engineering [141] in a broader sense. We begin with metrics which are used by various recovery approaches and other work on grammar engineering. We want to highlight [6, 58, 135, 150, 174]. Our work leverages simple grammar metrics (numbers of bottom and top nonterminals) and grammar-comparison metrics (numbers of nominal and structural differences) for providing guidance in a grammar convergence context.

An interesting blend of recovery and convergence (or consistency checking) is described in [21] where *precedence rules* are recovered from *multiple* grammars and checked for consistency. At this point, grammar convergence (in our sense) does not cover such sophisticated convergence issues. In fact, our approach is, as yet, oblivious to technology-specific representations of priority rules (as used in, say YACC or SDF). We could potentially detect priority layers in plain grammars, though.

An alternative to grammar recovery is the use of a flexible parsing scheme based on advanced error handling [12, 13, 142], subject to a baseline grammar. Because of flexible parsing, the grammar could also be used to parse a dialect; no precise grammar is needed. Also, code with syntax errors can be handled, which is important in some application areas such as reverse or re-engineering of legacy code.

There are approaches to connect the technical spaces of grammarware and modelware in a manner that can be viewed as a form of grammar convergence. That is, the parser may be obtained from the (meta)model based on appropriate metadata and mapping rules, using a generative approach [134, 192]. We also use the term model-driven parser development for these approaches. The point of grammar convergence is that it provides a very flexible means to represent relationships between grammar-like artefacts from different technical spaces—without enforcing a particular scheme of designing grammar-based artefacts.

5.9.4 Schema/metamodel comparison

Grammar comparison, as it is part of grammar convergence, can be loosely compared with *schema matching* in ER/relational modelling [49, 212] as well as model and *metamodel matching or comparison* in model-driven engineering [62, 248, 254] (specifically in the context of model/metamodel evolution). However, our current approach to comparison (as of section 4.5) is relatively trivial, and does not make any contribution to this subject, not even remotely.

A simple comparison approach was sufficient so far for two reasons:

- ◇ The metamodel of grammars is relatively simple. All of the possible grammar knowledge sources listed in subsection 4.4.1 contain less components than a typical ontology. Our conscious choice was to limit ourselves to the most widely used concepts: classes like "production", "expression", "nonterminal" and "terminal", attributes like "production label" and "expression selector" and the axioms of the formal grammar theory. This allows us to solve a relatively large set of problems with little effort.

- ◇ The metamodel of our alignments is extremely simple. As it was explained in section 4.5, we only require to determine nominal differences (subject to the comparison of defined nonterminal names) and structural differences (subject to matching alternatives). We do not make distinctions between *matchings* that represent similarity and *mappings* that can be proven. At this moment we also do not support heterogeneous alignments that can, for example, successfully and gracefully converge selectors in one grammar with nonterminals in another.

Currently we have shown some results that can be obtained with these two simple metamodels and semi-automated programmable transformations (alignments, relationships). We will need a more advanced comparison machinery once we aim at the partial inference of grammar transformations. In this case, grammar convergence should benefit from previous work on schema matching and metamodel comparison.

5.9.5 Coupled transformations

Grammar convergence relates to mappings in data processing [160, 230], specifically to the underlying theory of data refinement, and applications thereof [5, 40, 87, 190, 199]. In data refinement, one also considers certain well-defined operators for transforming data models. These operators must be defined immediately in a way that they can be also interpreted as mappings at the data level so that instance data can be converted back and forth between the data models that are related by the transformation.

Inspired by data refinement, all semantics-preserving and -decreasing operators for grammar transformation can also be interpreted at the AST level, and we experiment with such an interpretation, which opens up new applications for grammar convergence. For instance, one could replace the parser of a given program with another parser, even when their AST types are different. That is, the convergence transformations would be executed at the AST-level as a kind of conversion.

Data refinement is actually a specific and highly disciplined instance of so-called coupled transformations, which are characterised to involve multiple kinds of software artefacts (such as types vs. instance data vs. programs over those types) that depend on each other in the sense that the transformation of one entity (of one kind) necessitates a transformation of another entity (of another kind, potentially) [157]. For instance, [14, 33, 85, 159, 237, 243] are concerned with coupling for data models or metamodels vs. instance data or models; [35] are concerned with coupling for data models and programs over these data models. Again, we expect that the extension of grammar convergence to cover classes of coupled transformations may potently enable new application areas for the convergence method.

5.10 Concluding remarks

We have provided the first published record of recovering and representing the relationships between a priori given sized grammars that serve different audiences (language users and implementers) and that capture different versions of the language. Recovery is based on a systematic and mechanised process. The approach carefully distinguishes grammar refactoring, extension, correction and relaxation. Our results indicate that consistency among the different grammars and versions even for a language as complex as Java is achievable.

The recovery and representation of grammar relationships is based on a systematic and mechanized process that leverages a priori known grammar bugs, grammar metrics (e.g., for problem indication), grammar comparison for nominal and structural differences, and most notably, grammar transformations. We carefully distinguish transformations for grammar refactoring, extension, correction and relaxation.

While the JLS situation required the recovery of grammar relationships, ultimately, such relationships should be maintained for a given network of software artefacts with embedded grammars. That is, the relationships should be continuously checked and possibly updated along dependent or independent evolution of the involved artefacts. For instance, one may consider the "more readable" grammar as the primary artefact, and generate the "more implementable" grammar in a semi-automatic, model-driven manner [134].

The approach, as it stands, faces a *productivity problem*. The transformation part of grammar convergence requires substantial effort by the grammar engineer to actually map any given grammar difference into a (short) sequence of applications of operators for grammar transformation. Simple productivity gains can be expected from advanced tool support. We currently rely on basic batch execution of the transformations. Instead, the transformations could be done interactively and incrementally with good integration for grammar comparison, transformation and error diagnosis. Other productivity gains are known to be achievable by means of normalisation schemes (cf., de-/yaccification in [132, 154]).

However, ultimately, we need to provide inference of relationships (in fact, transformations). Such inference is involved because the process involves elements of choice that we need to better understand before we expect reasonable results. For instance, when two

syntactic categories are equivalent under fold/unfold modulations, then the grammar engineer is likely to favour one of the two forms—this calls for either an interactive approach or appropriate notions of normal forms or rule-based normalisation.

Perhaps the most exciting, remaining problem is to provide a proper formal argument for the "minimality" of the non-semantics-preserving transformations that are involved in a convergence. Currently, we use the pragmatic approach to first align nonterminals, then to align alternatives (by structure) as much as possible, and finally to break out of refactoring and allow ourselves presumably local non-semantics preserving transformations. However, there is no formal guarantee currently for not facing a false positive ("a presumed language difference that is none"). That is, one may accidentally engage in semantics-revising transformations even though the relevant syntactic categories are equivalent, but nonterminal symbols or alternatives are confused by the grammar engineer. Formally, the desired notion of minimality is limited by the undecidability of grammar equivalence, but we are confident that a practical strategy can be devised based on appropriate static analyses of the transformations and the involved grammars.

Chapter 6

Language documentation

Standards-making is not the deadly dull activity that it is popularly supposed to be.

Brian L. Meek, 1994 [180]

Language documents have been used in both case studies presented so far: chapter 3 recovered a working C# grammar from ECMA language standard, chapter 5 used Java Language Specification as a source of grammar knowledge. However, up to this point we considered language documentation a read-only artefact, we did not intend to create it, change its structure or delegate any grammar changes back to it. The purpose of this chapter is to cover these topics, motivated by conclusions drawn from software engineering theory and practice.

Upon analysing and reverse engineering a wide range of language documents from international ISO-approved standards to vendor-specific 4GL manuals we have designed a general schema capable of covering all features necessary for composition of such language documents. Having combined this document schema with a formalism for syntax definition presented in earlier chapters, we achieved a working prototype infrastructure to extract, create and transform consistent language definitions. Furthermore, we will discuss an organised process in which consecutive versions are obtained by employing changes to language documentation.

The next chapter will serve as a case study for us: it is a result of applying our infrastructure with semi-automated programmable language transformations to XBGF, the grammar transformation language mentioned, utilised and partially described in Chapters 4–5. XBGF is a language, and its manual was composed in a way that we propose for all language specifications. Thus, chapter 7 and its online version [261] were fully generated from the corresponding language document; section 6.7 explains the details of this process.

6.1 Motivation

Language grammars always exist in a context, even in the purest examples of grammar convergence from the previous chapters we had to dedicate considerable space to explain the details of grammar extraction. In practice all mainstream languages are somehow standardised, meaning that it is possible to obtain a grammar of any significant programming language from its documentation and use the textual annotations to resolve ambiguities and clear misunderstandings.

Defining a programming language in a standardised specification is often considered as a process that is executed just once. The dynamic and evolving nature of programming languages is underestimated and overlooked. Not only software itself, but programming languages that are used to make it, evolve over time. This process usually comes naturally in the sense that the first version of a language does not have all the features desired by its creator, or new features arise that seem quite appropriate to add, or better ways of achieving the same goals are invented or developed. However, it is important for that process to be guided and controlled in order to take full profit of it.

For certain languages this process is placed under governance of International Standardization Subcommittee for programming languages, their environments and system software interfaces: ISO/IEC JTC1/SC22 [100]. As described in detail in section 2.11, there are organisations like ANSI [7], ECMA [60], IEEE-SA [93], IEC [92], ITU [122], IETF [96], OASIS [194], WSA [253], W3C [242] that occasionally assist or replace ISO [97] in this field.

A language specification (programming language standard) is a complex document that may consist of hundreds of pages: the latest COBOL standard, ISO/IEC 1989:2002 [113], has more than 800 pages; the latest C [119] and C# [115] standards contain over 500 pages each, C++ draft is already well over 1100 pages [110]. It is important to note here that it has not always been like that. For example, the Algol 60 standard [10] is not much longer than 30 pages and yet it claimed to contain a complete definition of the language. However, programming languages evolve, their specifications grow in size and it is not possible anymore to keep them easily understood and maintainable while keeping them also short. Complicated structure of modern language documents reflects the complicated structure of modern programming languages. Writing and maintaining such a document and keeping it consistent is as complex as writing and maintaining a large software system—these processes have a lot in common.

There are tools like parsers and compilers which development is based on a language specification. If the language specification progresses with inconsistencies that may or may not be noticed and resolved in time, this can lead to non-conformance. If the evolution of the language specification is not fast enough, the developers or such derived tools have to make their own decisions and resolve the conflicts as they arise—this eventually yields a vendor-specific language dialect. As of now, it has not yet become a technological bottleneck to keep up with tool development (since making these tools is a very time consuming process performed by every compiler vendor independently), yet there are organisational bottlenecks that will have to be dealt with in this chapter.

6.2 Contributions

- ◇ We have deeply analysed a big number of language documentation artefacts, including language standards such as [30, 36, 112, 113, 114, 115, 185, 195, 225], language specifications such as [75, 184, 208, 209], language reference manuals such as [1, 86, 89, 98, 131, 139, 183, 215, 232], internal documents of standardisation bodies such as [7, 60, 97, 121, 194, 242], as well as more general yet well-structured software engineering books such as [73] and research attempts such as [9, 31, 50, 51, 66, 67, 76, 112, 123, 176, 144, 201, 250, 252], reverse engineered their intended internal structure and presented the comparison result in Table 6.1, Table 6.2 and accompanying text.
- ◇ We have designed Language Document Format, a generalised data model suitable for handling all features found in the analysed language documents. A transformation language capable of performing basic manipulations with language documents was also designed and presented. We have also laid out the mapping between XML Schema and Language Document Format.
- ◇ We have developed a prototype infrastructure for creating, maintaining and transforming language documents based on the above mentioned data model and the transformation language. An automated XSD to LDF extractor is also part of the tool set.
- ◇ A number of case studies have been examined by re-implementing them within the proposed infrastructure. The most remarkable ones were included in the thesis as its parts: chapter 7 (XBGF), subsection 4.10.1 (LCF), section 6.5 (LDF) and section 6.6 (XLDF).

The concepts of handling language documentation with technical discipline and providing proper tool support were formulated on an early stage of research as *Language Standardization Needs Grammarware* [143]. The paper titled *Language Convergence Infrastructure* [259] being published the post-proceedings of 3rd International Summer School on Generative and Transformational Techniques in Software Engineering included a more up-to-date view, with an extended abstract already published in the pre-proceedings as [258] and reported at the aforementioned summer school. *Language Documentation: Survey and Synthesis of a Unified Format* [262] is accepted for publication at the 3rd International Conference on Software Language Engineering. The complete description of LDF and XLDF is publicly available through the *Software Language Processing Suite* [263]. The conceptual prerequisites of this chapter were formulated in collaboration with Dr. Steven Klusener (Vrije Universiteit Amsterdam) and delivered with his help to the ISO SC22 committee in 2005. [262] was co-authored with Prof. Dr. Ralf Lämmel.

6.3 Grammar definition formalisms

It was noted as early as in 1958 [9] that having a metalanguage for defining the set of legal programs is necessary. Later work [252] on streamlining this metalanguage was standard-

ised as [112], unlike other attempts to extend BNF [31, 123, 250]. The results were widely accepted: the ISO standard is not being used directly, but rather changed and used in a form of various dialects. There are also later research attempts for extending BNF, such as Augmented BNF, or ABNF [201], which allows for rule-level modularity—alternatives can be added at separate lines without changing the original production and accurate limits on repetition—arbitrary numbers instead of just (0,∞) and (1,∞); or Translational BNF, or TBNF [175], which is basically EBNF enhanced with a possibility of specifying the structure of the abstract syntax tree.

Table 6.1 shows how different symbols and used in different ways in several real language definitions for well-known languages like C, C++, C#, Java, FORTRAN, etc. The top part of the table has metalanguage symbols proposed by the ISO standard [112], the bottom part is introduced here.

The table contains four different kind of arrows to show indentation. ↓ means the next line starts at the same position the right hand side of the previous line started. In the following example C and D, as well as E and F, are separated by ↓:

```
      ↓
A:  B C
    D E
    F G
```

↩ means the next line has less leading whitespace than the previous line, ↪ means it has more. For instance:

```
   ↩          ↪
A:  B  C | A:  B  C
D        |       D
```

↙ means the next line is empty and the line following the next one has no leading whitespace.

```
      ↙
A:  B C

D:  E F
```

The rest of this section discusses the table from Table 6.1 in detail.

6.3.1 Defining and separating symbols

There were three metasymbols of this category discussed in [112]: defining-symbol that signals the start of a new production; definition-separator-symbol that separates choices within a production; and terminator-symbol that shows the end of a production.

From Table 6.1 we can see the most widely used character for defining-symbol is a colon (`:`). Because of its de facto acceptance, it was also used in LLL [149] that was presented in the examples of one of the previous chapters, as well as in pretty-printed BNF representation of BGF files. BASIC standards are special here in a sense that they

Metalanguage symbol	Algol-58 [9]	Smalltalk [220]	C#[115, 114, 225]	C [119], C++ [110]	Java [77, 78, 79, read]	Java [77, 78, 79, impl]	BASIC [109]	FORTRAN [36]	Scheme [95]	Scala [198]	Pretty-printed BGF
defining	::≡	::=	:	:	:	:	=,>	**is**		::=	:
definition-separator	or	|	↓	↓	↓	↓, |	/	**or**		|	|
start-group		(				(	(			(	(
end-group		)				)	)			)	)
start-option		[				[		[		[	
end-option		]				]		]		]	
start-repeat						}				{	
end-repeat						{				}	
repetition		*,+					*	...	...		*,+
terminator	↩	↩	↙	↙	↙	↙	↩	↩		↩	↩
start-nonterminal	<	<<†							⟨		
end-nonterminal	>	>>†							⟩		
start-terminal		'								‘	"
end-terminal		'								’	"
nonterminal-font	a	a, *a*	*a*	*a*	*a*	*a*	a	*a*	**a**	a	a
terminal-font	a	a	**a**	**a**	a	a	A	a	**a**	**a**	a
optionality			*opt*	*opt*	*opt*		?				?
line-continuation	↓	↓	↪	↪	↪, ↩	↪, ↩	↓	↪, ■↓■		↓	↪

Notes:
† Smalltalk language manual uses <<a>> notation for syntactic categories of program grammar, <a> for syntactic categories of method grammar and simply a for lexical non-terminals.

Table 6.1: BNF dialects. The top part of the table has metalanguage symbols proposed by the ISO standard for EBNF [112], the bottom part is introduced by this thesis.

use two kinds of defining-symbols: the one that is meant to introduce a new definition and the one that adds alternatives to an existing one—a useful feature for language extensions.

Wirth's hope for a widespread tangible terminator-symbol was not met, as seen from the table. Most grammar engineering and transformation kits and methodologies like GDK [149], GRK [158], YACC [128], ANTLR [202], TXL [39] do have such a symbol, but it is rarely seen in language documents that prefer to rely on layout. Most of the time the terminator-symbol is just a newline character or double newline.

The need for a local choice metasymbol was not anticipated in early works like [9]. It is not shown directly in the table, but the definition-separator-symbols used in early standards are actually only meant for separating top choices, not inner choices. In modern standards this role is usually fulfilled with special indentation, while the vertical line character (`|`) is used only when local choices are needed. Also its positioning matters: for instance, Full BASIC standard [109] by ISO [100] is made almost unreadable by placing its definition-separator-symbol, the solidus (`/`), at the end of the line, making it very easy for a human reader to overlook and to misread as line continuation (compare Listing 6.1 with Listing 6.2).

Scheme programming language standard [95] by IEEE uses an approach that does not need defining and separating symbols. It presents only one formula at a time and has a unified notation for code examples, metaformulæ and evaluation examples. Thus, a BNF formula is identified by the `syntax` keyword placed at the end of it.

Listing 6.3 demonstrates the way BGF deals with productions and choices. The XML form shown here was derived automatically from [78, §4.2] by the extractor from chapter 5, with its plain text counterpart generated by an XSLT pretty-printer.

6.3.2 Optionality metasymbols

[112] had two kinds of metasymbols for this category: start-option-symbol and end-option-symbol, with the third one introduced in this thesis: an optionality-symbol.

An optionality-symbol is a character or a token-forming line of characters that designate one immediately preceding symbol as optional. I.e., if start-option-symbol is `[` and end-option-symbol is `]`, then optionality-symbol `?` can be defined as follows:

```
6.  ask-object           >  WINDOW boundary-variables /
                            VIEWPORT boundary-variables /
                            DEVICE WINDOW boundary-variables /
                            DEVICE VIEWPORT boundary-variables /
                            DEVICE SIZE numeric-variable comma
                            numeric-variable comma
                            string-variable /
                            CLIP string-variable
7.  boundary-variables   =  numeric-variable comma
                            numeric-variable comma
                            numeric-variable comma
                            numeric-variable
```

Listing 6.1: BNF in Full BASIC standard [109, page 8].

```
33  R202    program-unit    is  main-program
34                          or  external-subprogram
35                          or  module
36                          or  submodule
37                          or  block-data
38  R1101   main-program    is  [ program-stmt ]
39                              [ specification-part ]
39                              [ execution-part ]
39                              [ internal-subprogram-part ]
39                              end-program-stmt
```

Listing 6.2: BNF in Fortran standard [36, pages 26–27].

```
  <bgf:production>
    <nonterminal>PrimitiveType</nonterminal>
    <bgf:expression>
      <choice>
        <bgf:expression>
          <nonterminal>NumericType</nonterminal>
        </bgf:expression>
        <bgf:expression>
          <terminal>boolean</terminal>
        </bgf:expression>
      </choice>
    </bgf:expression>
  </bgf:production>
```

```
PrimitiveType:
        NumericType
        "boolean"
```

Listing 6.3: BGF: a sample production from [78, §4.2] in XML and pretty-printed forms.

$$a? \equiv [a]$$

In the case when a BNF dialect has optionality-symbol but lacks start-option-symbol and end-option-symbol, a nontrivial set of symbols can only be made optional by grouping them or extracting them into a separate nonterminal. Ergo, optionality-symbol is less expressive.

As seen from Table 6.1, BNF dialects tend to have exclusively either optionality-symbol or start- and end-option-symbols, but not both.

6.3.3 Iteration metasymbols

ISO EBNF [112] had all three kinds of metasymbols explained for this category: start-repeat-symbol, end-repeat-symbol and repetition-symbol.

The most used iteration type is, of course, the reflexive transitive closure, also called Kleene closure, that is traditionally represented by a star in the formal languages theory and by curly brackets as start- and end-repeat-symbols.

Some of the language specifications such as Smalltalk manual [220] also use transitive closure and designate it by a plus symbol. There is no counterpart for this in start- and end- symbols.

Another form of notation used for repetitive constructions is an ellipsis. For instance, in Scheme specification [95], there are two uses for it:

$$\langle \texttt{expression} \rangle \ \ldots \equiv \langle \texttt{expression} \rangle +$$

and

$$\langle \texttt{clause}_1 \rangle \ \langle \texttt{clause}_2 \rangle \ \ldots \equiv \langle \texttt{clause} \rangle +$$

In FORTRAN [36] committee draft a somewhat different usage was observed:

$$[digit] \ \ldots \equiv digit* \tag{6.1}$$

And, obviously,

$$digit \ [digit] \ \ldots \equiv digit^+$$

Since it is known from language theory that

$$\forall a \qquad a?^+ \equiv a^\star$$

$\ldots$ is the same as $+$, but in practice constructions like the left hand side of Equation 6.1 cannot be mapped directly as $?+$ because that would change the structure of the parse trees and introduce unnecessary ambiguities.

Advanced syntax definition formalisms also have lists with separators, which are identified by a type of iteration as well as by a separator between every entity in the list. For example, in SDF [86]:

$$\{a \ s\}^+ \equiv a \ (s \ a)^\star \equiv (a \ s)^\star \ a$$

Lists with separators as a notational extension were not encountered in language manuals, but lists with separators as a syntactic construct are quite common for many languages. The SDF's $\{a\ s\}^+$ was the only binary confix metasymbol that we have seen.

6.3.4 Grouping symbols

In order to enrich the metalanguage enough to enable local choices and collective closures, two metasymbols are used to mark the start and the end of a group. This feature does not add anything to the expressive power of the EBNF dialect, since any group can be extracted to serve as a separate nonterminal. However, it improves grammar readability and does not hinder metalanguage complexity because of its simplicity.

It is worth mentioning here that the grammars of C, C++ and C# avoid grouping and they also lack start-repeat-symbol and end-repeat-symbol, thereby enforcing a specific grammar engineering style (the above mentioned nonterminal extraction). In chapter 5, when grammars written in such style needed to be converged with grammars written with the use of repeating constructs, it led to a huge number of *inline* commands in XBGF scripts, as seen in Table 5.6.

6.3.5 Distinguishing terminals

Wirth [252] pointed out the necessity for quoting the terminal symbols of the language so that their set does not pose any limitations on the character set allowed for metasymbols and nonterminal symbol names. As we see from Table 6.1, this is rarely done in our days by using the conventional quotation marks. However, since we are no more limited to fixed type fonts and hand-written formulæ that can be easily misinterpreted, it is becoming common to distinguish between terminals and nonterminals by font face. The remaining two standards that have neither of the above are ISO BASIC [109] and IEEE Scheme [95]. The latter still has a mechanism for distinguishing nonterminals, so the only consequence for it concerns metalanguage symbols. In Full BASIC standard, all terminals must be typed in in upper case, while the nonterminals stay low case alpha-numeric, and that is how the distinguishing is made.

BGF is an XML dialect, and as such, it is self-documenting: any subtree belongs to a certain known type. Therefore, terminal symbols are also distinguished naturally: they will be encountered within `<terminal>` ··· `</terminal>` tags and pretty-printed with surrounding double quotes.

6.3.6 Distinguishing nonterminals

In order to simplify EBNF parsing even further, nonterminals can have a special syntactic form too. While Algol [9] and Scheme [95] just use angle brackets, Smalltalk [220] specification takes it to a new level. ANSI Smalltalk uses italic font for non-lexical bottom sorts (called "atoms") that are left undefined in the abstract grammar, but must be somehow defined in any concrete grammar that is based upon it. This is a unique feature—remember that in chapter 5 we needed an analysis script to collect the bottom sorts. That standard

also uses three different notations: the bracket-less one for lexical nonterminals; single-bracket and double-bracket ones for context-free nonterminals of different scope.

In BGF, nonterminals are wrapped in `<nonterminal>` ··· `</nonterminal>` tags.

6.3.7 Breaking up long lines

When the language structure is so sophisticated that one needs long EBNF formulæ, it is possible they will not fit on one physical line. In that case, there should be means to continue on the next line without stopping the production. Algol, Smalltalk and BASIC use ↓ to signal line continuation. Since that was already reserved for definition-separator-symbol in C, C++, C# and Java specifications, they increase indentation after the newline (denoted as ↪). Java Language Specifications [77, 78, 79] also let the indentation vanish after the newline—this is not a documented feature, it was discovered during the convergence of JLS grammars in chapter 5.

FORTRAN standard [36] defines line continuation explicitly by a halmos (■), but in the syntax rules themselves it is used quite inconsistently and often replaced with a newline and indentation shift (↪).

It should be noted here that the way we group the language documents according to their EBNF dialects, does not necessarily correspond to the trivial groups in real life (by language family, by standardisation body, etc). For example, ↓ is a definition-separator symbol in C specification, but a line-continuation symbol in BASIC specification, both being ISO standards designed by adjacent ISO committees.

6.3.8 Documenting the grammar

ISO EBNF standard [112] proposed to enrich the EBNF as such with comments. This was not adopted in any of the language specifications we have seen, since essentially the whole language specification is a commentary on the set of grammar productions. However, this feature is present in almost all grammar engineering and transformation frameworks that use EBNF dialects internally as a programming language.

BGF does not have any documentation facility except the standard XML one: `<!--` ··· `-->`.

6.3.9 Beyond BNF

As it was shown, there are huge differences even within something people call BNF or EBNF. Also note that the examples were chosen mainly from standards from the last decade: the divergence would have been explicable for artefacts from early 1970s when languages were invented on a rate of one per week [246] and there was an obvious unnecessary diversity of notations for syntactic definitions [252]. Some of the BNF dialects have names: e.g., BNF used in Switching System Language (SSL) was referred to as SBNF (SSL BNF) in [219], most are nameless and only exist in the scope of one language definition.

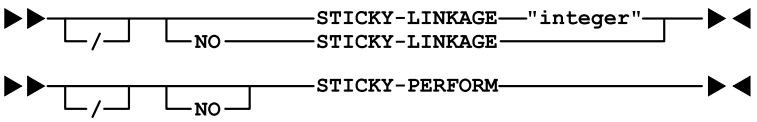


Figure 6.1: Railroad tracks from Micro Focus COBOL language manual [182, page 3-87].

ISO standards like STEP [99] or EXPRESS [98] still use Wirth syntax notation (WSN), which was a direct precursor of EBNF [252]. Most of the diversity we see now is nothing more than a legacy. Early Basic had all lines in the source code numbered, so the grammar rules were numbered as well, it seemed like an obviously good idea at the time. Early Fortran had tremendously strict formatting policy, hence the indentation rules for their BNF dialect. Java was born in mid 1990s, when the word editors, both WYSIWYG and otherwise, started to provide the user with lots of new functionalities, so they used different font faces and font sizes, and later it spread to specifications of C# and ISO C/C++.

All these notations and hundreds more are semantically equivalent to EBNF and are either as powerful as EBNF or slightly weaker in expression power due to idiosyncratic design choices. This allows for creation of grammar extractors that can help migrate grammars specified in different formalism to one standardised EBNF like [112]. In the scope of this thesis, a number of grammar extractors were developed and published online as a part of Software Language Processing Suite [263], for the complete list see subsection 8.3.5.

There is a number of syntax definitions that use a visual notation: e.g., Micro Focus COBOL [183] uses so called "railroad tracks" (Figure 6.1), syntax diagrams with terminals and nonterminals as nodes and their possible relations and combinations as arrows. These variations can be considered distant dialects of BNF in theory [76], but can prove to be very difficult to extract and transform in practice due to their graphical nature. There are also formalisms that cannot be easily mapped onto BNF. One of them that has a chance to enter the world of standardisation in the future is Parsing Expression Grammar [69] that can be converted directly into a recursive descent parser with exponential parsing time (unacceptable performance due to unlimited lookahead). Optimised versions use so called packrat parsing [68, 82] that makes the performance acceptable, but there are still different kinds of problems impeding its widespread use. Some of the obstacles, like the ban on left recursion, are being eliminated by recent research results [172].

6.4 Unified language document data model

The last section has shown how BNF [9] and EBNF [252] dialects are used in contemporary language documents. The ISO standard for EBNF [112] can become the consolidation point for those language engineers who want to use one generally accepted notation instead of including its variant description in every specification. We neither make solid proposals for adapting or improving ISO EBNF, nor provide any migration tools to extract

grammar knowledge in known notations. However, some of the extractors developed in the scope of this thesis can be considered as such tools: most importantly, BNF/HTML extractor used for Java Language Specification in section 5.4 is the closest of all. The comprehensive list of all extractors in included in the conclusion.

ISO EBNF in its current state lacks features utilised by some language standardisers (and those that the others can utilise), and programming languages subcommittee SC 22 does not enforce its usage within its working groups. It should also not be forgotten that ISO EBNF is a language specification itself, and should be created as such: consistently and with good semi-automated support.

From this point on we assume to have an EBNF-like notation to rely on when specifying grammar knowledge. This notation should support basic features discussed above: specifying nonterminal definitions, distinguishing terminal symbols, expression iteration, etc. In the working prototype BGF (see section 7.4) was used, but there have scarcely been any idiosyncratic steps.

6.4.1 Combining existing practices into one model

In Table 6.2 a comparison is presented of a set of language specifications. This selection has been specifically composed of sufficiently different language documents, coming from varying standardisation institutions and using different presentation traditions. Excerpts from these standards cannot be presented here due to copyright limitations.

- ◇ **ANSI Smalltalk** [220] is an NCITS J20 draft of 1997, 300+ pages long, it describes both the language (ANSI Smalltalk is derived from Smalltalk-80) and the Standard Class Library.

- ◇ Two **IBM** manuals for fourth generation languages: **JCL** [91] and **Informix** [89]—are perfect examples of industrial standards, they are extremely strictly structured, contain minimum extra sections and have impressive volume: 700+ and 1000+ pages. JCL spec uses EBNF dialect and Informix spec does it with railroad track syntax diagrams.

- ◇ **MOF Core Specification** [197] is a 90-pages long document describing Meta Object Facility. It uses UML and presents the information is a significantly different way, being oriented on diagrams, properties, operations and constraints. However, the overall information structuring turns out to be similar to conventional (E)BNF-based standards.

- ◇ **JOVIAL**, or J73 [185] is a Military Standard of 1984, which "has been reviewed and determined to be valid" in 1994. It is approved for use by the Department of the Air Force and is available for use by all other Departments and Agencies of the Department of Defense of USA. The version that was examined for Table 6.2 is a result of a second upgrade of the original language, of which we failed to acquire the specification. It is less than 200 pages and very strictly composed: basically every section has a syntax, semantics and constraints subsections, with rare notes or examples. A traditional BNF is uses for syntax, plain English for semantics.

LDF	Smalltalk [220]	JCL [91]	Informix [89]	MOF [197]	Jovial [185]	Design Patterns [73]	C# [115, 225]	XPath [15]
synopsis	synopsis	purpose	∼	∼	∼	intent	∼	definition
description	definition	description	usage	—	—	motivation	∼	—
syntax	∼	syntax	∼	—	syntax	structure	∼	[NN]
constraints	errors	requirements	restrictions	constraints	constraints	applicability	∼	constraint
references	—	references	references	—	—	related patterns	∼	HTML
relationship	return value, refinement	relationship, overrides	related	—	—	consequences	return type	∼
semantics	—	considerations	—	semantics	semantics	collaborations	∼	∼
rationale	rationale	locations	NLS	rationale	notes	implementation	note	note
example	—	examples	∼	—	examples	sample code, known uses	example	example
update	—	—	—	changes	—	—	—	—
default	—	defaults	note	—	—	—	default values	∼
value	conforms to	type	—	—	—	also known as	—	—
list	messages, parameters	subparameters	*terminals*	properties	—	—	—	∼
section	—	∼	∼	—	—	—	∼	∼
subtopic	—	parameters, fields	fields	operations	types	participants	parameters, methods	∼

Table 6.2: Comparison of eight existing language definitions against the presented model

- ◇ ***Design Patterns****: Elements of Reusable Object-Oriented Software* [73] is a well-known book by Erich Gamma et al., which defines 23 design patterns that every mainstream programmer is familiar with. Since design patterns can be considered a special language, their definition can be considered a language document—and the table only proves that, letting the 400 pages long book's structure fit in the general data model perfectly.

- ◇ **C#** standards [114, 115, 225] are known to the reader since chapter 3. The **ECMA** version used for this comparison is 550 pages long and very loosely structured, explaining a lot of issues in running text and using arbitrary subsectioning.

- ◇ The structure of **XPath W3C Recommendation** [15] is quite volatile, following the tradition of all other W3C recommendations. Each section contains one or several EBNF formulæ, the definition for a domain concept modelled by it and a body of text organised arbitrarily in lists and subsections. Examples are preceded by English introductory sentences like "here are some examples:".

The table contains our best effort in extracting the structure from the language documents listed above. We reflect on the structure as such and do not make any claims about the consistency of its usage—i.e., if there is a section called "default" somewhere in the C# specification, we record that fact without checking that indeed all default values are only encountered within such sections. "—" denotes the absence of the information that was meant to be in the corresponding section—i.e., the MOF standard contains no examples, the Informix standard contains no semantical explanations. "∼" denotes the lack of a separate structural element for the category when the information itself is still found in running text or as a part of neighbour or parent sections—i.e., the synopsis (brief description of the topic) was mostly present in all specifications, but rather as a first sentence of every section, preceding all others. For us the distinction between "—" and "∼" is important since the decision to not make a particular information portion into a separate section in the final document is a presentation decision, independent of the real internal structure.

The rows in the upper side of the column represent the subsections of any structured section (see subsection 6.5.13). The **value** is any fixed value that can be bound to a section but is not structured itself—for example, some sections in JCL specification about parameters have a designated "parameter type". The **list** is a simple list of items or a list of definitions. The **section** is a textual block that was for some reason shaped into a section but does not fall into any of the standard categories—for instance, §3.5 Comparison Expressions of XPath recommendation contains sections about value comparisons, general comparisons and node comparisons. This notion is close to the notion of a "topic" in DITA [195]. The **subtopic** is a fully structured subsection that can have its own synopsis, definition, etc.

The schema that is presented in the forthcoming sections deliberately does not impose any specific order of the sections that have just been explained. This is done because it has been seen in practice that the order is not the same for all existing language definitions, and insisting on any specific one would be limiting. Instead, we allow any order in the XML file and defer the ordering to the presentation layer (XSLT in the current infrastructure).

6.4.2 LDF benefits

Using our infrastructure has certain benefits that were mentioned earlier in passing, but the most important examples are collected in the following list. In parenthesis we give the names of the affected language documentation qualities from section 2.10.

- ◇ The grammar included in the language documentation is correct notation-wise, since it is defined with BGF. It is also correct parsing-wise, if completed with necessary details like definitions of lexical categories. (*Usability, integrability*).
- ◇ A working parser can be generated directly from the grammar taken from the LDF document. (*Usability, learnability*).
- ◇ Various checks and analyses can be performed on the grammar directly, without the tedious and nontrivial grammar recovery step. (*Integrability, searchability, maintainability*).
- ◇ Samples included in the LDF document are marked as such, which allows for their automated extraction and validation by parsing with the attached grammar. (*Usability, integrability*).
- ◇ An LDF document can be pretty-printed automatically to generate a classic PDF document at any time. Other output formats such as TEX, DocBook or HTML are also possible. (*Usability, readability*).
- ◇ If anyone decides to change the resulting PDF design, layout or any other presentation features, the document itself need not be touched. A separately developed pretty-printer can take the same document and present it differently (e.g., apply restrictions imposed by the standardisation body). (*Operability, maintainability*).
- ◇ Summary entities like tables of contents or collected grammar productions are stored as placeholders and generated on the fly. There is no redundancy and no manually performed error-prone actions involved. (*Operability, browsability*).
- ◇ Change documents encapsulating the delta between two consecutive versions of a language can refer to sections and paragraphs more exactly and be treated as sequences of technical commands, not as verbal descriptions. (*Maintainability, extensibility, learnability*).
- ◇ Generic actions can be performed error-free on subsets of LDF nodes like “all formulæ”, “all figures”, “all NB remarks”, etc. (*Maintainability, reusability*).

6.4.3 Alternative approaches

Publishing-oriented approaches operates in terms of “documents”. For them, a language specification is a textual document like any other, it is edited by a designated person who manually adds text to it, reorders sections, performs layout and formatting operations. The course of action for such an editor is described in a separated so-called “change document”. A change document can be created before the actual change takes place or

directly after it, and comprises a list of intended modifications. Once the editing process reaches a certain milestone, a new "revision" is delivered and stored in the archive. Once all the modifications approved by the language design committee are brought upon the main document, a new "version" is delivered and officially distributed within the terms of its license. This approach tends to utilise programs like Adobe Framemaker (ISO/IEC JTC1/SC22/WG4 [101]), Microsoft Word (Microsoft version of C# [184]), etc. It is also possible to use HTML (early W3C [211]) in such a way that the main document is edited manually and the changed are discussed and/or documented elsewhere.

This approach puts a lot of stress on the person or people responsible for editing the main document. All manual edits are error-prone and not easily tested. There is no verifiable link between the change documents and the main document revisions. Any structured content that is a part of a language document must be formatted in a way dictated by the medium: e.g., the formulæ can only use the symbols available in the font. It is also common to have several differently organised layers in the infrastructure: e.g., the main document is edited by one person following the instructions in the change document, but the change documents circulate in the form of co-authored Word documents.

Structure-oriented approaches operate in documentation domain concepts such as "chapters" and "sections". They make use of existing editing software to support maintenance activities. An example of such a documentation support system is DocBook [245]. It is a mature technology which has successful real-life uses and is well-documented. It is rather helpful as an aid in publishing help files for computer systems: e.g., Microsoft uses DocBook to generate help files for its Windows applications. PDF, LaTeX, Framemaker and other output formats are also possible — unlike the first type of approaches, this one uses one central repository which is filled with well-structured data, from which other formats can be generated on demand.

The separation between the content and its presentation can be sufficient in DocBook and similar systems. However, their orientation on books as the main medium prevents or hinders having documents that have several intertwined hierarchies (i.e., a syntax diagram is a part of a grammar as well as a part of a corresponding section). The instruments for representing language evolution steps as transformations exist (such as XSLT for DocBook) but rarely used for this purpose.

Topic-oriented approaches operate in terms of "topics" that should be covered in order for the documentation to be complete. The DocBook counterpart in this group of approaches is DITA (Darwin Information Typing Architecture) [195] which was designed specifically for authoring, producing and delivering technical information. IBM uses DITA for their hardware documentation. PDF, HTML, Windows help files and other output formats are possible. DocBook is a relatively modern technology (2004 versus DocBook's 1991), but the toolkits for it are being developed and the documentation is being written. A more lightweight approach is wiki technology that allows for topics to be left uncovered, showing explicitly which parts of the documentation are intended to be written in the future.

A specification of a programming language is not naturally organised in DITA topics and tasks. The upright investment in defining document schemata in detail before the first positive results can be seen is also an apparent drawback. For instance, necessary element types like grammar productions, code examples illustrating language features,

notes concerning version differences, optional feature descriptions, possible implementation remarks, language engineering explanations — must all be defined from scratch before DITA can be applied profitably.

LDF is the closest possible language documentation methodology one can have to literate programming [147]. We propose to use a data model narrowly tailored to the domain, yet to build one base line artefact which is meant for both understanding and formal specification. Any secondary artefacts such as grammars, test sets, web pages, language manuals and change documents should then be generated. We also have a prototype for **XLDF**, the transformation language for language documents.

6.5 LDF description

A document is essentially a sequence of several parts, such as title page, front matter, lexical and syntax sections, placeholders for generated content and various lists.

6.5.0.1 Syntax

```
document:
      titlePage frontMatter::anyTopSections lists? lexicalPart?
            (core::structuredSection+ |
            part::(core::structuredSection+)+)
            backMatter::anyTopSections?
            annex::structuredSection*
```

```
anyTopSections:
      topSection*
```

```
lists:
      frontList+
```

```
lexicalPart:
      lexicalSection+
```

6.5.1 Title page

The “title page” can in reality be rendered as several pages, but it contains the basic information that helps to identify this particular language definition and to distinguish it from similar documents. In our experience, language standards are either marked as organisation-created or person-authored ones. In the former case, the document must contain the name of the organisation and the standard reference number within it (e.g, ISO 10279). In the latter case, one or more authors are specified. It is also possible to mark some sections as having been authored by a specific set of authors, but the ones defined here are the principal authors that identify the specific standard.

The “topic” of the language document is its pure textual title without the reference number and status: e.g., “Programming Language REXX”, “Information technology — Programming languages — Full BASIC”, etc. Either version or edition follows.

According to W3C Process Document [124], each viable specification goes through the stages of Working Draft (WD), Candidate Recommendation (CR), Proposed Recommendation (PR), W3C Recommendation (REC), with possible continuation to Proposed Edited Recommendation and decline to Rescinded Recommendation. There has also been a “Note” status in the past for internal drafts.

IEEE uses different publication types, they are: Changed Designation, Collection, New Standard Project, Modified New Project, Modified Revision Project, Revision Project, Adoption in Progress, Approved Publication of IEEE, International Publication, Trial Use, Amendment, Corrigenda of Standard, Superseded, Withdrawn [94].

ISO/IEC operates with the following standard ratification stages: Approved Work Item (AWI), Working Draft (WD), Committee Draft (CD), Committee Draft Technical Report (CD TR), Committee Draft Technical Specification (CD TS), Committee Draft for Vote (CDV, only for IEC), Draft International Standard (DIS), Final Committee Draft (FCD, only for JTC1), Draft Technical Report (DTR), Draft Technical Specification (DTS), Final Draft International Standard (FDIS), International Standard (ISO), Technical Report (ISO TR), Technical Specification (ISO TS).

Some standardisation bodies like ANSI or ECMA do not have a long list of stages, the standard there is just either approved or not. Of course, it still can be revised, reaffirmed, withdrawn or be put into a category of technical reports.

We summarise these and possibly other sets of statuses by the following enumeration. Conceptually it provides functionality for the same categories, but the concrete wording may vary (i.e., “errata” vs “corrigenda”, “obsolete” vs “rescinded”). If necessary, the schema can be extended with more standard publication types or even adapted to fit completely in some specific standardisation body classification.

6.5.1.1 Syntax

```
titlePage:
      ((body number::INT) | author::STR+) topic::STR (version::STR
            | edition::STR) status date::STR
```

```
status:
      unknown::ε
      draft::ε
      candidate::ε
      proposed::ε
      approved::ε
      revised::ε
      obsolete::ε
      withdrawn::ε
      collection::ε
      trial::ε
      errata::ε
      report::ε
```

6.5.2 Standardisation bodies

6.5.2.1 Syntax

```
body:
      ansi::ε
      ecma::ε
      ieee::ε
      iso::ε
      iso/iet::ε
      itu::ε
      ietf::ε
      iec::ε
      oasis::ε
      wsa::ε
      w3c::ε
```

The comprehensive list of all standardisation bodies completed with descriptions of their work is given in section 2.11.

6.5.3 Placeholders for generated content

Entities such as table of contents can be generated by the infrastructure automatically, ensuring internal consistency and coherence. However, the language documentation creators should be able to specify the places where such data needs to be inserted.

6.5.3.1 Syntax

```
generated:
      index::ε
      fullgrammar::ε
      listoftables::ε
      listofauthors::ε
      listofcontents::ε
      listofreferences::ε
```

6.5.4 Front and back matter sections

Whatever the authors deem to be important enough to be put on one of the first pages. For example, in C# specifications Foreword is about the differences brought to the language by the current standard, while in the Scheme specification Foreword discusses programming languages design and demonstrates its principles applied to the forthcoming document.

6.5.4.1 Syntax

Technically speaking, Foreword is not a part of the specification. Instead, it precedes the specification and introduces it by putting in the proper context.

```
[foreword] topSection:
      simpleSection
```

The goals of language design are sometimes encountered being explicitly stated in the language document in one of the informative sections of the front matter part.

For example: ”C# is intended to be a simple, modern, general-purpose, object-oriented programming language.” (from ECMA 334 3rd edition, page xvii)

```
[designGoals] topSection:
      simpleSection
```

Scope section explains the context for the language document.

```
[scope] topSection:
      simpleSection
```

Conformance section defines several levels of compliance by explaining what is a conforming program and a conforming implementation with respect to this standard.

Definitions for meta-terms like ”shall” and ”should” and their relation to the compliance issue explained above.

```
[conformance] topSection:
      simpleSection
```

Same as conformance:

```
[compliance] topSection:
      simpleSection
```

While conformance/compliance define how external artefacts should conform to this standard, this section defines how this standard complies with previously existing ones.

```
[compatibility] topSection:
      simpleSection
```

Notation section defines grammar definition formalism used in the document: mostly it is about the EBNF dialect.

```
[notation] topSection:
      simpleSection
```

Formally lined up references to all other standards that are used or referenced to from within the document.

```
[normativeReferences] topSection:
      simpleSection
```

This section informally describes how the document is organised, divided into parts and chapters. Sometimes it explicitly states which sections are normative and which are informative.

```
[documentStructure] topSection:
      simpleSection
```

A list of changes brought to the language by the current specification replacing the previous one.

```
[whatsnew] topSection:
      simpleSection
```

Placeholders are used to designate places where generated content should be inserted.

```
[placeholder] topSection:
      generated
```

6.5.5 Simple sections

Innermost sections do not have to belong to a certain pre-defined type (like “synopsis” or “design goals”), but they can have a title and a possible list of authors which contributed directly to this section.

6.5.5.1 Syntax

```
simpleSection:
      title::STR? author::STR* content::simpleText id::STR?
```

6.5.6 Simple text

Simple text can include paragraphs, formulae, tables, lists, links, keywords, etc.

6.5.6.1 Syntax

```
simpleText:
      simpleTextElement+
```

```
simpleTextElement:
      empty::ε
      text::mixedType
      figure::simpleFigure
      table::simpleTable
      list::simpleList
      formula::(ANY+)
      sample::((ANY | STR)* src::STR)
      production
```

6.5.7 Simple figures

A figure in LDF can include several presentation variants in different formats. Any of them can be chosen by the rendering tool: for example, in our prototype, TEX generators prefer PDF figures while HTML generators tend to prefer PNG ones.

6.5.7.1 Syntax

```
simpleFigure:
      shortcaption::STR? caption::STR source::(type::figureType
            (localfile::STR | url::STR))+ id::STR?
```

```
figureType:
      PDF::ε
      PostScript::ε
      SVG::ε
      PNG::ε
      GIF::ε
      JPEG::ε
```

6.5.8 Simple tables

Tables have header rows (optional) and regular rows, with each row filled with table cells.

6.5.8.1 Syntax

```
simpleTable:
      header::tableRow* row::tableRow+
```

```
tableRow:
      cell::simpleText+
```

6.5.9 Simple lists

A list in LDF is nothing more sophisticated than a sequence of textual items.

6.5.9.1 Syntax

```
simpleList:
      item::mixedType+
```

6.5.10 Front matter lists

6.5.10.1 Syntax

This is a list of definitions for all basic terms needed to understand the document, but not especially introduced in it. Typical examples include explaining what is a ”program”, what is a ”namespace”, what is a ”library”, what is ”behaviour”.

```
[definitions] frontList:
      listOfTerms
```

Acronyms like IEEE, ISO or like CLI, BCL are frequently defined here in a separate definition list.

```
[abbreviations] frontList:
      listOfTerms
```

Such an informal introduction to the language is not present in all standards. However, some specifications contain a list of language concepts with their definitions and perhaps even some examples. The list is usually not claimed to be exhaustive.

```
[languageOverview] frontList:
      listOfTerms
```

```
listOfTerms:
      title::STR? author::STR* term+ id::STR?
```

```
term:
      name::STR definition::simpleText
```

6.5.11 Lexical part

Sections describing lexical structure tend to be shorter, less structured inside, and very limited in scope: there is usually one lexical section dedicated to whitespace, one lexical section about tokens, one about literals, one about comments, etc.

See the section on grammar notation for more details about how broad even the smallest aspects (e.g., about line continuations) can vary.

6.5.11.1 Syntax

```
lexicalSection:
      lineContinuations::simpleSection
      whitespace::simpleSection
      tokens::simpleSection
      preprocessor::simpleSection
      literals::simpleSection
      lexical::simpleSection
```

6.5.12 Main sections

The main bulk of the information the language document possesses lies in its core sections: they have more structure than other sections and than their own subsections, so we call them “structured sections”.

Beside the title, any section can include a list of authors that worked on it: this way even if the whole document is attributed to a particular company or a standardisation body, the main contributors can be attributed.

6.5.12.1 Syntax

```
structuredSection:
      title::STR author::STR* structuredSectionElement+ id::STR?
```

Some subsections can be generated, especially those that consist of structured content that is possible to derive automatically from the information stored in or collected from other sections.

```
[placeholder] structuredSectionElement:
      generated
```

Normative sections are obligatory and usually contain strict content that needs to be implemented by compiler developers or satisfied by language end users in order to comply to this standard.

For instance, if a section with a grammar production is marked as normative, this production must hold for the implemented language.

```
[normative] structuredSectionElement:
      normative
```

Informative sections are supplementary and provide some useful information that can be omitted or overridden if deemed appropriately.

For instance, if all sections with code samples are marked as informative, we cannot rely on the set of examples extracted from the standard to be the test set for the language. Apparently, we can still run some analyses on the basis of this, but it is not legitimate to make any conclusions about standard inconsistencies based on the informative sections, nor can they be legitimately be used to resolve inner conflicts of the documentation.

```
[informative] structuredSectionElement:
      informative
```

Grammar productions can be easily extracted from the language document for the purpose of grammar recovery, grammar adaptation, grammar convergence, etc.

```
[production] structuredSectionElement:
      production
```

Even hardbound standards contain hypertext-like elements. One of them is attaching a list of links and backlinks to every section—that way, one can easily find any related language constructs when learning the language or debugging a particular feature.

```
[references] structuredSectionElement:
      simpleList
```

Any other type of normative or informative section.

```
[section] structuredSectionElement:
      simpleSection
```

In big documents it is not uncommon to find one topic divided into several subtopics, each one dedicated to a separate issue and each one structured in the same way its parent section is. DSLs and 4GLs specification authors often find it easier to lay out different clauses of one language construct in different sections. When the language has a lot of parametrised constructs, it makes sense to dedicate a special subsection for each field. Parameters, types, methods, operations, participants—whatever categories inspire these subtopics, each of them is a fully structured section in itself.

```
[subtopic] structuredSectionElement:
      structuredSection
```

A singled named value can be bound to a language document section: it can be a type of a parameter that is being described here, or an alternative name, or a superclass, or anything else that is atomic and non-structured.

```
[value] structuredSectionElement:
      key::STR data::STR
```

6.5.13 Normative sections

Normative sections form the core of the language standard: for each core section they belong to, they define the purpose of the language construct, provide a description, a syntax definition, list use constraints and other semantic details, etc.

6.5.13.1 Syntax

A separate subsection named “purpose” is only encountered in some 4GL language manuals (e.g., JCL). However, it is quite common for the first paragraph of any new section of any language document to briefly introduce the purpose of the language construct that is about to be described.

```
[synopsis] normative:
      simpleSection
```

Description is the core of the parent section, containing the main details about the defined topic, information about its usage, motivation behind its design.

```
[description] normative:
      simpleSection
```

Syntax sections consist of one or more BGF productions, possibly complemented by textual descriptions.

```
[syntax] normative:
      simpleSection
```

This section can list requirements needed for using a specific language construct, applicability constraints and other restrictions.

```
[constraints] normative:
      simpleSection
```

Related language constructs can be named and referenced here. It is not a simple list of references, but rather a comprehensive overview on the kind of consequences other parts of the language can bear if this one is used.

```
[relationship] normative:
      simpleSection
```

A section on semantics explains in plain English, if no other specific formalism is used, how exactly the language construct works, what happens inside the system when it is utilised. It also describes the context in which the introduced language construct can be encountered and in which it should or should not be used.

```
[semantics] normative:
      simpleSection
```

It is quite common in the sections that describe an optional parameter to tell the reader what will happen in the case nothing was specified. “There is no default” can be as valid a definition as a real default value.

```
[default] normative:
      simpleSection
```

6.5.14 Informative sections

Unlike normative sections that impose some conformance constraints that need to be satisfied by a language processor to claim compatibility with the standard, the informative sections are only presented to provide some additional information to the reader.

6.5.14.1 Syntax

A rationale or a note usually lists some narrow places of non-obvious usage, implementation details, incompatibility issues, coding standards, common programming practices, etc. It is a subsection of secondary importance, contributing some information on a minor point that can still be interesting and useful for some readers. Notes usually tell the readers how to use certain language constructs or tell compiler vendors how to implement them.

```
[rationale] informative:
      simpleSection
```

A subsection with an example can contain a code sample as well as some accompanying text.

```
[example] informative:
      simpleSection
```

It is quite uncommon practice, but some standardisation bodies really put the information about language evolution directly into each section that changed since the last public version of the language document.

If this is done consistently and carefully, it is possible to generate the global “What’s new” section automatically.

```
[update] informative:
      simpleSection
```

6.5.15 Formulæ

Formulae can be used in language documentation in many ways. The internal representation format is MathML which is W3C Recommendation, but the external representation can vary, in our case there are two: TEX and HTML.

6.5.15.1 Syntax

```
formula:
      ANY+
```

6.5.16 Keywords, links and plain text

Keywords are usually printed in bolder font weight. They need to be marked as such for two purposes: for presentation and for meta-information. The former goal serves as a basis for typesetting and hyperlinking, while the latter allows for correct indexing and searching facilities.

Inlined pieces of code are usually printed in a typewriter-like font. They are frequently incomplete, mostly nothing more than simple literals, and can only be checked to be correct tokens of the language in a lexical sense.

Internal links are pairs of text that will become clickable in hypertext presentation forms or precede the reference itself when this is the only option. The reference points to a section or a subsection of the same document that the link should refer to. If the explicit text is omitted, its default value is the name of the section being referenced.

Any unstructured element of the language document belongs to a so called mixed type: i.e., it is plain text with some keywords marked.

6.5.16.1 Syntax

```
keyword:
      STR
```

```
code:
      STR
```

```
link:
      text::STR? (reference::STR | external::STR)
```

```
mixedType:
      (ANY | STR)*
```

6.6 Transforming a language document

Unlike XBGF that has been developed in a consistent manner to cover all possible grammar transformations and that has been partially published as [166, 167, 261], the language for language document transformations, XLDF, was kept to a minimum set of commands necessary for a demonstration of our approach. A prototype has been designed and made publicly available at [263], but no claims have been made about the coverage of XLDF operator suite. The prototype has been tested by doing a range of case studies for this thesis: even this section was also generated from XLDF schema by using programmed XLDF transformation steps.

6.6.1 evolutionSequence

Sequential composition of multiple transformations.

6.6.1.1 Syntax

```
evolutionSequence:
      documentTransformation*
```

6.6.2 documentTransformation

XLDF transformation suite does not include any sophisticated grammar manipulations, in that sense it complements XBGF rather than extends it. All grammar transformations in XBGF that did not take place beforehand, can be executed as a part of language document transformations.

6.6.2.1 Syntax

```
documentTransformation:
        addFigure
        addSection
        addSubsection
        append
        changeRole
        combine
        drop
        extractSubsection
        hyperlinkify
        insert
        importGrammar
        importSample
        place
        retitle
        removeSection
        transformDocument
        transformGrammar
```

6.6.3 addFigure

Adds a figure to the designated section or subsection.

6.6.3.1 Syntax

```
addFigure:
       figure::simpleFigure to::ID
```

6.6.4 addSection

Completes the language document with a top-level section. The target is unspecified because it is possible for the transformation engine to decide it automatically (i.e., lexical sections go to the lexical part, core sections to the core, etc).

6.6.4.1 Syntax

```
addSection:
       frontList
       lexicalSection
       placeholder::generated
       core::structuredSection
       annex::structuredSection
```

6.6.5 addSubsection

Adds an inner section to the document. All front matter sections go automatically to the front matter, otherwise a target needs to be specified.

6.6.5.1 Syntax

```
addSubsection:
      topSection
      (normative | informative) to::ID
```

6.6.6 append

Adds more content to the existing section or subsection.

6.6.6.1 Syntax

```
append:
      where::STR content::simpleText
```

6.6.7 combine

Reorganises all the content from one section or subsection to be merged with another section or subsection.

6.6.7.1 Syntax

```
combine:
      section::ID with::ID
```

6.6.8 changeRole

Changes the role of a section (i.e., syntax section can become description). If there is no section of the same role in scope, the original section becomes one. If there was already a section with the same role in scope, the new section is appended to it instead.

6.6.8.1 Syntax

```
changeRole:
      scope::STR from::sectionRole to::sectionRole
```

```
sectionRole:
      frontMatter::ε
      backMatter::ε
      synopsis::ε
      description::ε
      syntax::ε
      constraints::ε
      relationship::ε
      semantics::ε
      default::ε
      rationale::ε
      example::ε
      update::ε
      section::ε
```

6.6.9 drop

Excludes one top level section from the language document.

6.6.9.1 Syntax

```
drop:
      section::ID
```

6.6.10 extractSubsection

Cuts out a part of an existing inner section and promotes it to a separate inner section. If the target context is not specified, the new section becomes a subsection of the place it has been extracted from.

6.6.10.1 Syntax

```
extractSubsection:
      from::ID content::simpleText title::STR id::ID to::ID?
            role::sectionRole?
```

6.6.11 hyperlinkify

Turns a document part referenced by an XPath expression into a hyperlink.

6.6.11.1 Syntax

hyperlinkify :
`goal::STR uri::STR`

6.6.12 importGrammar

Includes a piece of grammar taken from an external BGF file.

6.6.12.1 Syntax

importGrammar :
`target::ID file::STR`

6.6.13 importSample

Includes a sample taken from an external file. It is possible to specify a pretty-printer that will be fed that file and the output will be included.

For instance, in chapter 7 the samples were originally in XBGF, but were pretty-printed with an external universal pretty-printer to get the nice EBNF-ish look (otherwise the reader would be overwhelmed with XML listings). There were two different pretty-printers: one for the TEX version inlined in the thesis, and one for browsable HTML version available as [261].

6.6.13.1 Syntax

importSample :
`target::ID file::STR prettyprinter::STR?`

6.6.14 insert

This is one of the transformations that has no corresponding command in XBGF, since grammars do not have order of elements (in fact, any concrete grammar notation does, but it does not matter) and language documents do (and do care about it a lot). This transformation inserts new content right before or after another piece of content that is looked up by the XLDF engine at the run-time.

6.6.14.1 Syntax

```
insert:
      relative content::simpleText
```

```
relative:
      before::(id::ID | simpleTextElement)
      after::(id::ID | simpleTextElement)
      to::ID
```

6.6.15 place

Makes one section to become a subsection of another section. The target section must be of a type that supports subsections.

6.6.15.1 Syntax

```
place:
      section::ID inside::ID
```

6.6.16 retitle

Changes the title of a section or a subsection. It is still possible to designate the target for this transformation with an ID, but a more friendly and user-expected way of designating it with its current title is also possible: that way by looking at a transformation one can immediately tell what was the title before and what will it be (and possibly deduct the reason for this change).

6.6.16.1 Syntax

```
retitle:
      from::(id::ID | title::STR) to::STR
```

6.6.17 removeSection

Excludes one subsection from a specified section.

6.6.17.1 Syntax

```
removeSection:
      id::ID from::ID?
```

6.6.18 transformDocument

Modularisation support: we allow to put an evolution sequence in a separate file to be called from a place in another file.

6.6.18.1 Syntax

```
transformDocument:
      file::STR
```

6.6.19 transformGrammar

This is an interface between XLDF and XBGF: all the grammar productions are gathered from the target section and its subsections, to be fed into a transformation (or a chain of transformations). If the XBGF engine evaluates successfully, the result of that transformation is returned to the target section. It is also possible to specify a number of context section — all productions from them will be gathered, too, but since they are not subject to change, they will not be confused with the result.

6.6.19.1 Syntax

```
transformGrammar:
      target::ID transformation+ context::ID*
```

6.7 XBGF case study

As we can recollect from the previous chapters and papers, the megamodel of the grammar life cycle looks like as depicted on Figure 6.2: we extract grammar knowledge from various types of artefacts, then we transform it according to our goals (grammar evolution, restructuring, abstraction, correction, specialisation), and finally compute the result in a suitable form. Similarly, the language document life cycle in our infrastructure is schematically presented on Figure 6.3. The grammar (either the extracted one or the one obtained by transformations) is combined with additional information derived from the existing documentation, then transformed according to our goals (language evolution,

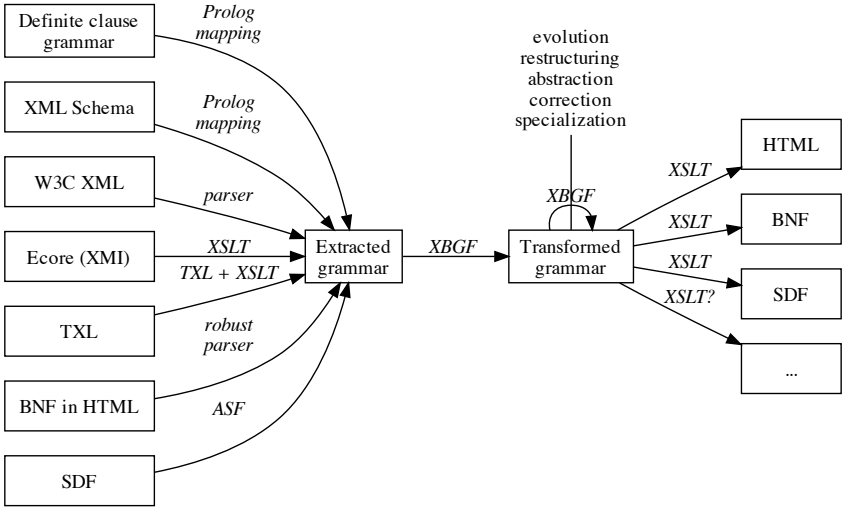


Figure 6.2: The life cycle of a language grammar in the transformational environment: from a grammar knowledge possessing software artefact on the left to the usable working grammar on the right.

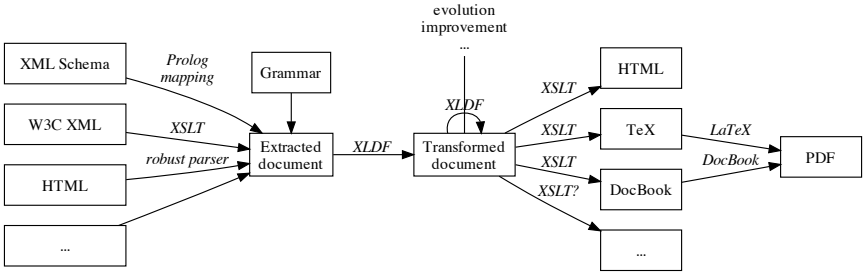


Figure 6.3: The life cycle of a language document in the prototype language document infrastructure: from the structure source on the left to the extracted document gradually transformed to its ultimate form and finally pretty-printed for presentation on the right.

language improvement, any grammar maintenance activities, etc) and then the final form is computed.

In the prototype of this chapter we started with an XML Schema definition. We have the tools to map definitions of XML elements, groups and other entities to grammar productions, for which the extractor from chapter 4 is reused. We also developed new tools to map XSD annotations to LDF textual paragraphs. Once an LDF document is ready, one can use XLDF commands to transform it. These commands can utilise secondary sources of information such as test suites to fill in the gaps in the language documentation. Transformations written in XLDF can take this LDF as an input and allow for adaptation, evolution, beautification, etc, as discussed earlier. Eventually the LDF document is considered ready for presentation, and a range of generator tools allow to make a PDF file out of it, a TEX source or an HTML web page.

6.7.1 Extraction

For us the central part of any language document is the grammar behind it. At the point when we started composing the XBGF manual, the grammar of XBGF has already been specified by an XML Schema definition `shared/xsd/xbgf.xdf` in SLPS [263]. This schema was not used directly in parsing by the Prolog program that handled the transformations, but validation checks were performed with it.

XSD to BGF mapping has also already been established as a part of FL case study—see section 4.3 and Listing 4.10. We needed only to extend it to design XSD to LDF mapping. It was decided that every XSD construct that defined a schema entity should be mapped to a separate top-level section of a language document. Those constructs were: XML elements, XML attributes, named content types, groups and attribute groups—each of them was mapped to a nonterminal symbol for BGF and to a section describing this nonterminal for LDF.

In XSD it is possible to annotate any construct with a piece of text, and that feature was extensively utilised during schema development phase to provide comments for XBGF operators. With `xsd:annotation` and `xsd:documentation` tags we basically inserted typical language documentation information right into the schema. The idea came naturally to map such annotations to the textual content of the corresponding sections of the language manual.

Two front matter sections were decided to be filled differently: foreword and normative references. The top level annotations (those assigned to the whole document and not to a specific definition) were mapped to foreword and the list of imported XML Schema definitions became normative references. After filling out details like the document title and author we were ready to produce a correct LDF document for any given XSD.

6.7.2 Transformation

Since the structure of the language document generated by XSD to LDF extractor was very simple and too straightforward, we needed document transformation steps to reorder the sections, to add lacking textual content, to connect and pretty-print samples, etc. The transformation suite explained in section 6.6 was used for that.

6.7.2.1 Adding sections

Grammars or schemata, even heavily annotated ones, contain only information specific to their level of abstraction. Unlike them, language documents tend to contain meta-level information that is not usually part of a schema or a grammar, and low-level details—in our scenario, summaries of experience gained from case studies. An example of meta-level information can be the "design goals" section—section 7.3. It has been added explicitly with an **addSubsection** call:

```
addSubsection (
  designGoals: (author: "Vadim Zaytsev",
             content: (text: ..., list: ... ) );
```

An example of less abstract information follows:

```
addSubsection (
  semantics: (content: (text: "The inline transformation
    is by far the most used in Java Language Specification
    case study..." ) ),
  to: "inline" );
```

6.7.2.2 Combining sections together

Composite sections like section 7.4 are not uncommon in language documentation. When the extracted sections are either too small or too insignificant, they are combined in a way similar to this:

```
addSubsection (
  notation: (author: "Vadim Zaytsev",
           content: (text: "BGF is a BNF-like Grammar Format..." ) );
combine (
  section: "grammar",
  with: "notation-section" );
combine (
  section: "production",
  with: "notation-section" );
combine (
  section: "expression",
  with: "notation-section" );
...
```

6.7.2.3 Beautifying the grammar

In cases when the extracted grammar is decided not to be readable enough, it can be transformed by calling XBGF commands from XLDF:

```
transformGrammar(
  target: "unfold",
  anonymize(
    unfold: ⟨nonterminal::nonterminal⟩ in::scope?)
);
```

6.7.2.4 Reorganising the sectioning

XML Schema definition has a flat structure, yet it often defines a tree-like hierarchy. In our case the transformation operators are grouped in several categories:

```
retitle(
  from: (title: "folding-unfolding-transformation" ),
  to: "Folding and unfolding transformations" );
place(
  section: "unfold",
  inside: "folding-unfolding-transformation" );
place(
  section: "fold",
  inside: "folding-unfolding-transformation" );
...
```

This cannot be executed automatically since one XSD element can belong to several XSD groups. In this case extraction process would include running graph analysis algorithms as applicability metrics, which will make the extractor's behaviour highly sophisticated. Since our main goal was to design and implement a simple prototype, we have chosen for less automation and more control here. When reorganisation is executed by a human expert, it is driven by domain concepts and their relations, therefore not involving any nontrivial decision making algorithms.

6.7.2.5 Adding code samples

As the source of code samples we used a test suite that was composed during the early development stages of XBGF and maintained ever since at `topics/transformation/xbgf/tests` in SLPS. Each test case consists of the grammar input, grammar transformation script and expected output. They are connected to the language document in a way similar to this:

```
addSubsection (
  example: (id: "unfold-example",
     content: (text: "Given the input:" ) ),
  to: "unfold" );
importGrammar (
  target: "unfold-example",
  file: "../../transformation/xbgf/tests/unfold.bgf" );
append (
  where: "unfold-example",
  content: (text: "After using this transformation:" ) );
importSample (
  target: "unfold-example",
  file: "../../transformation/xbgf/tests/unfold.xbgf",
  prettyprinter: "../../../shared/tools/xbgf2xbnf" );
append (
  where: "unfold-example",
  content: (text: "Will look like this:" ) );
importGrammar (
  target: "unfold-example",
  file: "../../transformation/xbgf/tests/unfold.baseline" );
```

More transformations can be run at this stage, we have only listed the most important classes of language document transformations that were encountered in the case study.

6.7.3 Presentation

Once the internal structure of the language document is satisfactory, it can be rendered for public, for third parties, for the web, etc. We have developed two mappings in particular: TEX and HTML. The first one was used for this thesis and for separated PDFs that are generated at `topics/documents/bgf`, `lcf`, `ldf`, `xldf` and `xbgf` in SLPS [263]. The second mapping helped to create an online browsable manual of XBGF [261].

Both mappings are trivial for anyone familiar with the target language, so we do not discuss them here. Perhaps the only detail worth attention was a secondary tool called a **re-pretty-printer**. As one can recall from the previous section, the language document may have built-in code samples. These samples are stored directly in the LDF document in a form ready to be used—i.e., they are pretty-printed. However, the link to the original file (if any) is stored as well, which lets us write an automated grammarware tool that can take all the samples in the given language document and re-apply another pretty-printer, replacing its result with the rendering statically saved in the LDF document. We wanted to retain the flexibility of having the code samples rendered and saved (because running a pretty-printer in not expressible in XSLT), but needed different ones for two of our presentation generators.

6.7.4 Conclusion

The XBGF chapter (chapter 7), including the notation subsection (section 7.4); the LCF/LCI section (section 4.10); the LDF section (section 6.4); the XLDF section (section 6.6) — they were all generated automatically from the corresponding operational schemata, with additional text added by transformations on the fly. All the samples from

the XBGF chapter were taken (included or pretty-printed) directly from the test suite we used to ensure the correctness of the XBGF engine. Both the engine and the scripts are available freely via the SVN repository of SLPS [263].

Shall our future research lead to extension of, say, XBGF, another XSD element will be added to the schema definition in the appropriate namespace. We will then annotate it there in a way common for XMLware. The same XLDF scripts that are used to produce the current manual will be used for the extended manual as well.

Existing technology can be integrated with other grammar engineering activities and generic language technology such that complete life cycle of a programming language can covered and a language definition finally receives its place as an important and valuable part of an IT portfolio. This generalised programming language life cycle technology can be embedded in existing programming and development environments, integrated into various third parties' products.

Chapter 7

XBGF language manual

The only thing that matters is the change, not the end result.

Linus Torvalds, 2010,
http://bit.ly/9WXmqr

The purpose and the nature of this chapter is twofold:

- ◇ First, it serves as an application for language documentation prototype infrastructure described in chapter 6. **This chapter, not including this inlay, is a complete language manual, and as such it has its content fully generated in a programmable fashion from the corresponding schema of XBGF.** There is also a hyperlinked browsable version of the same manual available as [261]. For implementation details please consult section 6.7.
- ◇ Second, it describes in much detail and many examples the transformational language used in chapter 4 and chapter 5 and was only briefly mentioned at section 4.8 and section 5.6. Having such a language is crucial for grammar recovery, for grammar convergence, as well as for grammar documentation.

7.1 Foreword

This chapter describes the transformational suite for BGF. Most of the information present here is located in the XML Schema definition of the language, part of the SLPS project. The rest was introduced by the language documentation transformation commands in the process of automated generation of the manual in its present form.

7.2 Normative references

- **bgf.xsd**

7.3 Design goals

XBGF operator suite was developed mainly for grammar convergence activities.

- **Grammar recovery.** In order to recover a working grammar from a real grammar artefact one needs to perform various activities such as grammar extraction, grammar beautification, deyaccification, error fixing, etc,—all of them are present in XBGF.

- **Grammar convergence.** An infrastructure for grammar convergence (see Chapter 4) contains three essential parts: grammar extractors, a grammar comparator and a grammar transformer. The latter component is needed for changing the source grammars so that they become structurally identical at the convergence point. XBGF suite can be used as a framework for programmable grammar transformations, it provides all the required expressive power.

- **Language documentation.** It is possible and quite expected that the grammar that is shown in the language standard, programmer's manual or any similar document is not exactly the same as the one used for creating a compiler. For instance, Java Language Specification includes two grammars, each one created and maintained manually and separately. A transformation suite can be useful in such a case since it helps to generate one of the versions (perhaps the more reader-friendly one) from the other automatically, both making their relationship more explicit and reducing maintenance effort and increasing reliability.

- **Language evolution.** A language rarely stays as it was developed on its first day. If any language is being used in practice for some time, new constructs are added to it to respond to the needs of the end users (programmers). XBGF suite has multiple operators for expressing language extensions and revisions, they can be used to document the changes between versions and dialects of the language.

- **Language design.** DSL design is not a rare activity nowadays, and it is quite common to develop the language gradually, regularly synchronising with the needs and knowledge of domain experts. XBGF suite allows to document these steps as transformation steps departing from the base-line grammar or even an empty grammar and finally reaching the language ready to be deployed.

- **Disciplined grammar adaptation.** In order to adapt the baseline grammar to the working circumstances one needs it to work in, it is useful to have a good support for grammar transformations [41, 43, 154, 219]. XBGF can be used to converge differently adapted grammars or as a standalone tool for applying grammar adaptation steps.

7.4 Notation

BGF is a BNF-like Grammar Format, an XML dialect of Extended Backus Naur Form that was used in the language convergence infrastructure. Its abstract syntax grammar is automatically extracted from the corresponding XML Schema and presented below:

```
grammar:
      root::nonterminal* production*
```

```
production:
      label::label? nonterminal::nonterminal expression
```

```
expression:
      epsilon::ε
      empty::ε
      value::value
      any::ε
      terminal::terminal
      nonterminal::nonterminal
      selectable::(selector::selector expression)
      sequence::(expression+)
      marked::expression
      choice::(expression+)
      optional::expression
      plus::expression
      star::expression
```

```
value:
      int::ε
      string::ε
```

```
label:
      STR
```

```
nonterminal:
      STR
```

```
selector:
      STR
```

```
terminal:
      STR
```

All BGF and XBGF listings are presented in a unified pretty-printed way. The concrete syntax for it is presented below:

```
grammar:
      production+
```

```
label:
      "[" STR "]"
```

```
production:
      label::label? nonterminal::STR ":" right-hand-side
```

```
right-hand-side:
      NEWLINE (INDENT symbol+ NEWLINE)+ NEWLINE
```

```
symbol:
      "ε"
      "EMPTY"
      "ANY"
      "STRING"
      "INT"
      terminal::(""" STR """)
      nonterminal::STR
      selectable::(selector::STR "::" symbol)
      sequence::("(" symbol+ ")")
      choice::("(" (symbol ("|" symbol)*) ")")
      optional::(symbol "?")
      plus::(symbol "+")
      star::(symbol "*")
      marked::("⟨" symbol "⟩")
```

Any XBGF command is pretty-printed as the name of the transformation and all the parameters (productions, expressions, etc) in brackets, followed by a semicolon.

7.5 List of definitions

Grammar A set of interdependent productions.

```
grammar:
      production*
```

Sequence Sequential composition of multiple transformations.

```
sequence:
      (transformation | atomic)*
```

Atomic Multiple transformations that must be for some reason perceived as one step.

```
atomic:
      transformation+
```

Transformation A list of all the XBGF transformations is grouped in seven categories: **folding-unfolding-transformation** collects those commands that perform the well-known folding/unfolding operations in slightly varied circumstances; **refactoring-transformation** contains transformations that perform factoring and reorganising procedures that do not alter the language generated by the grammar; **increasing-transformation** increase the semantics of the language by adding new options and alternatives to it; **decreasing-transformation** similarly decrease the semantics; **concrete-revising-transformation** are refactorings if we use term-oriented semantics (abstract syntax) but they are neither semantic preserving, increasing nor decreasing transformations if we use string-oriented semantics (concrete syntax); transformations from **abstract-revising-transformation** change the language generated by the grammar in a way that they cannot be a priori classified as any of the above; **decorative-transformation** are special refactorings that are used to make or destroy labels and selectors in BGF.

```
transformation:
      folding-unfolding-transformation
      refactoring-transformation
      increasing-transformation
      decreasing-transformation
      concrete-revising-transformation
      abstract-revising-transformation
      decorative-transformation
      rename
      reroot
      dump
```

Scope Several transformation operators are possibly restricted to a specific scope as opposed to their application to the full input grammar. Two major forms of scope can be identified. First, a production can be appointed by its label. Second, a definition (nonterminal) can be appointed by its defined nonterminal. Arguably, one may want to be able to appoint a production even when it is not labeled, but a prior designate transformation can then be used in order to attach a label to the production in question.

```
[label] scope:
      label
```

```
[nonterminal] scope:
      nonterminal
```

Marked Some of the grammar transformations, namely **addH**, **removeH**, **appear**, **disappear**, **upgrade**, **downgrade**, **abstractize**, **concretize**, **inject**, **project**, **anonymize**, **deanonymize**, accept only marked productions as their arguments.

```
marked-production:
      production
```

When transformations need to happen very locally, the level of nonterminal or production is insufficient and introduction of selectable sub-expressions is too much extra effort. For this cases, XBGF uses markers that show at which point exactly should the transformation take place. Markers are pretty-printed as angle brackets.

7.6 Folding and unfolding transformations

Folding and unfolding activities are the most basic ones in grammar transformation and the most used ones in grammar convergence. Since grammar comparison is done in such a way that only applies very basic algebraic laws in its endeavours to match the two grammars, many more sophisticated manipulations need to be executed semi-automatically in a programmable fashion. These manual steps help to establish a stronger link between the convergence point and the original grammar artefact since they aid to reveal some unapparent properties of those grammars.

All these transformations are provably correct in the sense that it is possible to prove that the languages generated by the grammars before the transformation and after it are indeed the same. All refactorings are easily reversible and introduced below in pairs.

```
folding-unfolding-transformation:
      unfold
      fold
      inline
      extract
      abridge
      detour
      unchain
      chain
```

7.6.1 unfold

The most basic unfolding transformation searches the scope for all the instances of the given nonterminal usage and replaces such occurrences with the defining expression of

that nonterminal. By default the scope of the transformation is the full grammar, but it can be limited to all the definitions of one nonterminal or to one labelled production. Regardless of the specified scope, unfolding is not applied to the definition of the argument nonterminal.

Almost all the **unfold** transformations used in Java Language Specification case study are restricted in scope by a nonterminal. The reason for such statistics is that when the language engineer wants to give up the nonterminal, he uses the **inline** transformations. However, **unfold** usually happens as a part of sequences with **fold**, **downgrade**, **disappear**, **deyaccify**, **distribute**, etc.—in which case it is only natural to limit the impact of every step.

The definition that is being unfolded is assumed to be horizontal, i.e. to consist of one single production. See the section on refactorings for more information about horizontal and vertical definitions.

7.6.1.1 Syntax

```
unfold:
      nonterminal in::scope?
```

7.6.1.2 Example

Given the input:

```
[l1] foo:
      bar
```

```
[l2] qux:
      bar
```

```
bar:
      wez*
```

After using this transformation:

```
unfold(bar);
```

Will look like this:

```
[l1] foo:
      wez*
```

```
[l2] qux:
       wez*
```

```
bar:
       wez*
```

7.6.2 fold

Folding replaces every expression that matches with the right hand side of the given nonterminal's definition with the nonterminal itself. As with **unfold**, **fold** works on the scope of the grammar, and its impact can be limited to one labelled production or to all the productions belonging to one nonterminal. Regardless of the specified scope, folding is not applied to the definition of the argument nonterminal.

Since this transformation strives to preserve the language, it needs a horizontal definition to work. When only one of several existing definitions is used for folding, it would actually increase the semantics of the language after transformation—the corresponding XBGF command is called **upgrade**.

7.6.2.1 Syntax

```
fold:
       nonterminal in::scope?
```

7.6.2.2 Example

Very much like unfolding, folding can take place locally. For instance,

```
[l1] foo:
       wez*
```

```
[l2] qux:
       wez*
```

```
bar:
       wez*
```

After using this transformation:

```
fold(bar in foo);
```

Will look like this:

```
[l1] foo:
       bar
```

```
[l2] qux:
       wez*
```

```
bar:
       wez*
```

7.6.3 inline

When this transformation is performed, an existing definition is eliminated by inlining. This means that the argument nonterminal identifies the (horizontal) definition that is to be unfolded and stripped away from the grammar.

The semantics of **inline** is that of a sequential composition of **unfold** and **eliminate**.

7.6.3.1 Syntax

```
inline :
       nonterminal
```

7.6.3.2 Semantics

The **inline** transformation is by far the most used in Java Language Specification case study. One of the reasons is what we call layering: in particular expressions and statements are introduced in the G_j^R with a set of related nonterminals: LabeledStatement, IfThenElseStatement, WhileStatement, ForStatement, etc, and CastExpression, PreIncrementExpression, PreDecrementExpression, PostfixExpression, etc. G_j^I takes another approach, just listing all the alternatives in the productions for Statement and Expression. In order to converge these two variants, a lot of inlining transformations are needed. This can also be apparent from the Table 5.6, that demonstrates that targets that converge only "readable" or only "implementable" grammars, require less than ten inline transformations each, while each target that takes both readable and implementable grammars in, contains 67–100 **inline** transformations in convergence path.

7.6.3.3 Example

An example follows. When we have:

```
foo:
       wez
```

```
bar:
      wez ".." wez
```

```
wez:
      qux*
```

After using this transformation:

```
inline(wez);
```

It will look like this:

```
foo:
      qux*
```

```
bar:
      qux* ".." qux*
```

7.6.4 extract

A new definition is introduced by extraction. The argument provided for this transformation is a production that identifies both the fresh nonterminal to be introduced and the expression that is used both as a pattern for folding and as a right hand side of the added definition. An optional scope can limit the application of the folding part of the extraction transformation to a specific production or a specific nonterminal.

If the nonterminal defined by the argument production is already mentioned (i.e., defined or referenced) in the current grammar, the transformation refuses to work and reports an error. This usually signals an error in the language engineer's logic because the existing traces of a possibly similar nonterminal conflict with the definition that is being introduced.

7.6.4.1 Syntax

```
extract:
      production in::scope?
```

7.6.4.2 Semantics

As seen from the experience gained from Java Language Specification case study, it is highly unusual for extract to have limited scope. However, sometimes a limited impact is desired in order to avoid excessive subsequent unfolding when the convergence requests for having several nonterminals with similar definitions.

7.6.4.3 Example

Extraction also works vertically. Given the input:

```
TypeDeclaration:
      ClassDeclaration
      InterfaceDeclaration
      ";"
```

After performing this transformation step:

```
extract (
 ClassOrInterfaceDeclaration:
      ClassDeclaration
      InterfaceDeclaration
);
```

The result will be:

```
TypeDeclaration:
      ClassOrInterfaceDeclaration
      ";"
```

```
ClassOrInterfaceDeclaration:
      ClassDeclaration
      InterfaceDeclaration
```

7.6.5 abridge

Given a reflexive chain production, i.e., a production whose defined nonterminal equals its body, this production is simply removed from the grammar, even if it contains some potentially valuable information (like labels and selectors).

7.6.5.1 Syntax

```
abridge :
      production
```

7.6.5.2 Semantics

Reflexive chain productions are rarely encountered explicitly in the base-line grammars, but sometimes series of transformations result in them, and usually they are not needed. An example of a transformation sequence that yields a reflexive chain production can be a step from concrete syntax definition to abstract syntax definition. Concrete syntax usually needs explicit bracketing constructions for recursive composition, and after stripping

away terminals and merging layers, these bracketing constructions become reflexive chain productions. The Factorial Language case study has shown the need for it.

7.6.5.3 Example

Consider this abstract syntax:

```
[constant] expr:
      int
```

```
[neg] expr:
      expr
```

```
[bracket] expr:
      expr
```

After performing this transformation step:

```
abridge (
 [bracket] expr:
      expr
);
```

The grammar will be the same, but without the reflexive chain production labelled as "bracket" previously:

```
[constant] expr:
      int
```

```
[neg] expr:
      expr
```

7.6.6 detour

A reverse of **abridge** that can introduce a reflexive chain production.

7.6.6.1 Syntax

```
detour:
      production
```

7.6.6.2 Example

In the same way it was removed in the previous example, the bracketing production can be added to the grammar. The transformation that reverts the impact of the previous **abridge**, looks like this:

```
detour (
 [bracket] expr:
       expr
);
```

7.6.7 unchain

Unchaining is a disciplined form of **inlining**. The argument production must occur in the input grammar, and it must be a chain production, i.e., a production with a nonterminal as its defining expression. The latter nonterminal is the one whose definition is to be inlined; it must not have any occurrences except in the chain production at hand.

The unchain transformation does not increase the expressivity of the transformational language: technically, it is nothing more than an inline with a precondition. However, this particular precondition seems useful and not uncommon when dealing with layered grammars.

7.6.7.1 Syntax

```
unchain :
       production
```

7.6.7.2 Semantics

Chain productions are not commonly encountered in grammars of mainstream programming languages. However, when converging grammars that hail from different kinds of sources (i.e., different extraction processes) it can be frequently needed to align grammars that use chain productions with grammars that use labelled ones.

7.6.7.3 Example

Consider this grammar:

```
[constant] expr:
       int
```

```
[binary] expr:
       binexpr
```

```
binexpr:
      expr op expr
```

After performing this transformation step:

```
unchain (
 [binary] expr:
      binexpr
);
```

The auxiliary nonterminal symbol is gone, as is the chain production:

```
[constant] expr:
      int
```

```
[binary] expr:
      expr op expr
```

7.6.8 chain

Just like **unchain** is a specific form of **inline**, chaining is a disciplined form of **extraction**. The argument production will be part of the resulting grammar; it is a chain production, i.e., a production with a nonterminal as its defining expression. That nonterminal is the one whose definition is to be extracted. That definition is the defining expression of the production (from the input grammar) whose defined nonterminal and label (if any) matches with the argument production.

7.6.8.1 Syntax

```
chain :
      production
```

7.6.8.2 Example

In the same way it was removed in the previous example, the chain production can be added to the grammar. The transformation that reverts the impact of the previous **unchain**, looks like this:

```
chain (
 [binary] expr:
      binexpr
);
```

7.7 Other refactoring transformations

Here is a list of the XBGF transformations that perform other provably semantic-preserving refactorings

```
refactoring-transformation:
        massage
        distribute
        factor
        deyaccify
        yaccify
        eliminate
        introduce
        import
        vertical
        horizontal
        equate
        rassoc
        lassoc
```

7.7.1 massage

The grammar is rewritten by local transformations such that the language generated by the grammar (or the denotation according to any other semantics for that matter) is preserved. The known rewriting rules affect the use of selectors and regular expression operators: e.g., any symbol will always generate the same set of strings that the same symbol wrapped in a selector.

There are two expression arguments: one to be matched, and another one that replaces the matched expression. One of them must be in a "massage relation" to the other.

The scope of the transformation can be limited.

7.7.1.1 Syntax

```
massage:
        expression expression in::scope?
```

If (x, y) represents sequential composition of symbols x and y, and $(x; y)$ represents a choice with x and y as alternatives, then the following algebraic laws define the massage-equality relation:

$$x? = (x;\varepsilon) \qquad (x?)? = x? \qquad (x, x^\star) = x^+$$
$$x? = (x?;\varepsilon) \qquad (x?)^+ = x^\star \qquad (x^\star, x) = x^+$$
$$x^\star = (x^+;\varepsilon) \qquad (x?)^\star = x^\star \qquad (x?, x^\star) = x^\star$$
$$x^\star = (x^\star;\varepsilon) \qquad (x^+)? = x^\star \qquad (x^\star, x?) = x^\star$$
$$x? = (x?;x) \qquad (x^+)^+ = x^+ \qquad (x^+, x^\star) = x^+$$
$$x^+ = (x^+;x) \qquad (x^+)^\star = x^\star \qquad (x^\star, x^+) = x^+$$
$$x^\star = (x^\star;x) \qquad (x^\star)? = x^\star \qquad (x^+, x?) = x^+$$
$$x^\star = (x?;x^+) \qquad (x^\star)^+ = x^\star \qquad (x?, x^+) = x^+$$
$$x^\star = (x?;x^\star) \qquad (x^\star)^\star = x^\star \qquad (x^\star, x^\star) = x^\star$$
$$x^\star = (x^+;x^\star)$$

These additional formulæ for associativity also hold:

$$(x,(y,x)?) = ((x,y)?,x)$$
$$(x,(y,x)^+) = ((x,y)^+,x)$$
$$(x,(y,x)^\star) = ((x,y)^\star,x)$$

If we introduce an infix operator "::" to denote selectors (named addressable subexpressions), then this additional rule is also needed:

$$x = (s_1 :: x; s_2 :: x)$$

The reason for having such an explicit rule is the normalisation. As explained in section 5.5, without selectors $(x;x)$ is always normalised back to x.

7.7.1.2 Example

Distributivity rules for optionality modifier such as these:

$$(x?y?)? = x?y?$$
$$(x^*y?)? = x^*y?$$
$$(x?y^*)? = x?y^*$$
$$(x^*y^*)? = x^*y^*$$
$$(x|y)? = x?|y?$$

are not explicitly covered by **massage** since it is possible to emulate them with a sequence of above mentioned patterns of **massage**, as well as with **factor** and similar transformations. Let us take the last formula as an example of a massaging that takes several steps to complete. The input BGF is:

```
foo:
      (bar | qux)?
```

After performing these transformation steps:

```
massage(
 (bar | qux)?,
 ((bar | qux) | ε));
massage(
 ε,
 (bar::ε | qux::ε));
factor(
 ((bar | qux) | (bar::ε | qux::ε)),
 ((bar | bar::ε) | (qux | qux::ε)));
anonymize(
 foo:
      (bar | ⟨bar::ε⟩)
      (qux | qux::ε)
);
massage(
 (bar | ε),
 bar?);
anonymize(
 foo:
      bar?
      (qux | ⟨qux::ε⟩)
);
massage(
 (qux | ε),
 qux?);
```

The result will be:

```
foo:
      bar?
      qux?
```

The selectors and **anonymize** commands are necessary because otherwise the choice of two epsilons would be removed automatically during the normalisation phase. The rest of distributivity laws are expressed quite similarly to this example.

7.7.2 distribute

Distribute sequential composition over choices so that choices are pulled out of sequences. The transformation is either attempted for all productions of a nonterminal or for a specific one appointed by its label.

7.7.2.1 Syntax

```
distribute:
      scope
```

7.7.2.2 Semantics

In fact, **distribute** is nothing more than an automated version of **factor** that aggressively pushes all the choices that can be found in a production outwards.

This transformation is apparently non-injective, hence, it is impossible to have a complete inverse of it. A more general **factor** transformation, however, is as capable of emulating **distribute**'s effect as it is capable of doing the reverse thing.

7.7.2.3 Example

For instance,

```
foo:
      bar (qux | wez)
```

After using this transformation:

```
distribute( in foo );
```

Will look like this:

```
foo:
      bar qux
      bar wez
```

7.7.3 factor

Massage modulo factorisation over choices. The transformation is either attempted for all productions of a nonterminal or for a specific one appointed by its label.

7.7.3.1 Syntax

```
factor:
      expression expression in::scope?
```

7.7.3.2 Semantics

Factor transformations tend to be quite frequently used in grammar convergence. They also have a tendency to be very long—since it is impossible to implement **factor** symmetrically to **distribute** (i.e., fully automated), the language engineer needs to supply two

complete expressions. The transformer then can easily assert that they are related by distribution: basically, it internally performs **distribute** on both of them and expects them to become identical. Hence, it is possible to do "incomplete" factoring by pushing choices inwards but not to the innermost position.

Two most commonly seen patterns of **factor** use are the following. First, it is applied when we have a choice of two long expressions that are almost identical except for some mismatching part. That part can be either extracted or massaged later with more transformations. Second, it is needed when we have a wide choice with the same leading (or trailing) symbol, and the goal is to let the common part stay and encapsulate the rest inside a different nonterminal (by following **extract**).

7.7.3.3 Example

For instance,

```
a:
      a
      b
      c d e
      c f g
      h
      i
```

After using this transformation (note the order of expressions):

```
factor (
 ((c d e) | (c f g)),
 c ((d e) | (f g)));
```

Will look like this:

```
a:
      a
      b
      c ((d e) | (f g))
      h
      i
```

7.7.4 deyaccify

Deyaccification is a widely used term that means converting recursive definitions to iterative ones where possible. The name comes from YACC, or Yet Another Compiler Compiler, a tool which underlying parsing technology limits were enforcing the usage of recursive definitions back in the 1970s. However, it somehow became common practice to remain within them even when grammar engineers do not use yacc as such at all.

The name of a nonterminal must be provided as an argument, then the transformation engine checks if the grammar productions for this nonterminal fit into one of the yaccified

patterns. If not, the error is reported, otherwise the definition is replaced by one that uses regular expression operators instead of epsilon, choice, and recursion.

Both left- and right-recursive forms can be factored with this transformation.

7.7.4.1 Syntax

```
deyaccify:
      nonterminal::nonterminal
```

Deyaccification uses several general patterns. Left recursion like this:

```
foo:
      bar
```

```
foo:
      foo wez
```

Becomes:

```
foo:
      bar wez*
```

Right recursion like this:

```
foo:
      bar
```

```
foo:
      wez foo
```

Becomes:

```
foo:
      wez* bar
```

In either case, it is checked if bar and wez are the same nonterminal. If they are, the result is more concise:

```
foo:
      bar+
```

7.7.5 yaccify

This transformation is the reverse of **deyaccify**. The productions provided as arguments must be yaccified with respect to the actual content of the grammar. If the deyaccification process on them is successful and yields the production that can be found in the grammar, it is removed and replaced by these simpler definitions of an optional or repeating nonterminal, given in BNF-only expressiveness.

Some complex **yaccify** cases require prior use of extract for introduction of an nonterminal for the optional or repeating phrase.

7.7.5.1 Syntax

```
yaccify :
      production+
```

7.7.5.2 Semantics

Yaccification is a typical example of grammar adaptation activity. However, it can be utilised in grammar convergence process as well: think of a situation when one of the sources is yaccified using left recursion while the other one—using right recursion. In such a case it would be better to deyaccify both of them. If this is due to some considerations impossible or undesirable, one can deyaccify, say, left recursion and then yaccify if back to right recursion.

Since it is not possible for the transformation engine to guess which kind of BNF recursion the suite user would need, **yaccify** takes two productions as parameters, unlike **deyaccify** which works perfectly just given the nonterminal name.

7.7.5.3 Example

For instance, this piece of grammar:

```
foo:
      bar+
```

can either be yaccified with left recursion:

```
yaccify (
 foo:
      bar
 foo:
      foo bar
);
```

to look like this:

```
foo:
      bar
```

```
foo:
      foo bar
```

or yaccified with right recursion:

```
yaccify (
 foo:
      bar
 foo:
      bar foo
);
```

to look like this:

```
foo:
      bar
```

```
foo:
      bar foo
```

7.7.6 eliminate

An unused definition (at most used within the definition itself) is removed. The **undefine** operator should be utilised instead when the definition must be removed despite remaining uses. The **remove** operator should be utilised instead when only part of the definition (i.e., a production of a vertical definition) is to be removed.

7.7.6.1 Syntax

```
eliminate:
      nonterminal::nonterminal
```

7.7.6.2 Example

For instance,

```
expr:
      int
```

```
intexpr:
      int
```

After using this transformation:

```
eliminate(intexpr);
```

Will look like this:

```
expr:
      int
```

7.7.7 introduce

A definition of a fresh nonterminal is added. The **add** operator should be used instead, if the nonterminal is already defined, is to be merely extended. The **define** operator should be used instead, if the nonterminal is readily in use, but merely lacks a definition.

7.7.7.1 Syntax

```
introduce:
      production+
```

7.7.7.2 Example

For instance,

```
a:
      b
```

```
b:
      ε
```

After using this transformation:

```
introduce(
 c:
      a
 c:
      b
);
```

Will look like this:

```
a:
      b
```

```
b:
      ε
```

```
c:
      a
```

```
c:
      b
```

7.7.8 import

Allows to import another grammar: the nonterminals within it can refer to one another, but none of the existing productions are allowed to refer to them before this transformation takes place.

7.7.8.1 Syntax

```
import:
      production+
```

7.7.8.2 Semantics

Consider a scenario where we want to introduce two productions, each defining a fresh nonterminal symbol, and each using the other. Without **import** the only way to do so would be to run one **introduce** and one **define**, which is semantically wrong since we are sure that before the first nonterminal is introduced, the second one was fresh. So, instead we take the interdependent productions together and introduce them in one step.

Technically, **import** can be used any time to substitute any number of **introduce** transformations. Whether this is a desired use pattern or not, is left at the discretion of the language engineer.

7.7.8.3 Example

For instance,

```
X:
      "a" "b"
```

After using this transformation:

```
import (
 A:
       B X
 B:
       A
       ε
);
```

It will look like this:

```
X:
       "a" "b"
```

```
A:
       B X
```

```
B:
       A
       ε
```

7.7.9 vertical

Turn top-level choices into multiple productions. The transformation is either attempted for all productions of a nonterminal or for a specific one appointed by its label. The action is a reverse of **horizontal**.

Occasionally we use terms "vertical" productions or nonterminals and "horizontal" ones. By vertical nonterminals we mean those that are defined by a list of productions, with every production lacking a top-level choice. A horizontal nonterminal, on the other hand, is defined by one production that is a top-level choice. Nonterminals that employ both top-level choices and splitting into multiple productions are neither horizontal nor vertical.

7.7.9.1 Syntax

```
vertical :
       scope
```

7.7.9.2 Example

If the original production contained selectors:

```
decs:
       onedec::dec
       moredecs::(dec decs)
```

then, after using this transformation:

```
vertical( in decs );
```

they are converted to labels:

```
[onedec] decs:
      dec
```

```
[moredecs] decs:
      dec decs
```

7.7.10 horizontal

Turn a definition based on multiple productions into a top choice-based one. The action is a reverse of **vertical**.

7.7.10.1 Syntax

```
horizontal:
      nonterminal::nonterminal
```

7.7.10.2 Example

If some or all of the original productions are labelled:

```
[onedec] decs:
      dec
```

```
[moredecs] decs:
      dec decs
```

the, after using this transformation:

```
horizontal(decs);
```

each label is converted to a selector in a corresponding place:

```
decs:
      onedec::dec
      moredecs::(dec decs)
```

7.7.11 rassoc

This transformation operator replaces an iterative production found in the grammar by the argument production, which is a right associative repeating equivalent of the former. Its defining expression involves a pattern of binary recursion with regard to the defined nonterminal. The “r” in “rassoc” refers to the intended effect at the level of derivation trees: the list of subtrees is to be converted into a nested binary tree in a right-associative manner.

7.7.11.1 Syntax

```
rassoc:
      production
```

7.7.11.2 Example

For instance,

```
[constant] expr:
      int
```

```
[binary] expr:
      expr (op expr)*
```

After using this transformation:

```
rassoc(
 [binary] expr:
      expr op expr
);
```

Will look like this:

```
[constant] expr:
      int
```

```
[binary] expr:
      expr op expr
```

7.7.12 lassoc

The same as **rassoc**, but replaces an iterative production found in the grammar by a left associative repeating equivalent. The “l” in “lassoc” refers to the intended effect at the

level of derivation trees: the list of subtrees is to be converted into a nested binary tree in a left-associative manner.

7.7.12.1 Syntax

```
lassoc:
      production
```

7.7.12.2 Example

For instance,

```
[terminal] expr:
      STR
```

```
[sequence] expr:
      expr+
```

After using this transformation:

```
lassoc(
 [sequence] expr:
      expr expr
);
```

Will look like this:

```
[terminal] expr:
      STR
```

```
[sequence] expr:
      expr expr
```

7.7.13 equate

Two nonterminals, say x and y, are merged, if their definitions are identical.

7.7.13.1 Syntax

```
equate:
      align::nonterminal with::nonterminal
```

7.8 Language increasing transformations

Here is a list of the XBGF transformations that lengthen the grammar (increase semantics).

```
increasing-transformation:
      add
      appear
      widen
      upgrade
      unite
```

7.8.1 add

Nonterminal definitions can be extended ("added to") vertically and horizontally. In the former case, a given production is added to an existing definition. In the latter case, a given branch is added to a given expression. The horizontal mode is there for convenience only because it could be simulated by a sequence of extraction, vertical addition, and inlining. There are two operators that are very similar to the (vertical) add operator: **define** and **introduce**. The define operator should be used when an the definition of an undefined nonterminal is added. The introduce operator should be used when a fresh nonterminal is to be defined.

7.8.1.1 addV

Syntax

```
[vertical] add:
      production
```

Vertical addition operates on the level of productions: it adds one more production for some nonterminal to any number of productions that are already present in the grammar.

Example Given the input:

```
expr:
      int
```

After using this transformation:

```
addV(
 expr:
      id
);
```

The result will look like this:

```
expr:
      int
```

```
expr:
      id
```

7.8.1.2 addH

Syntax

```
[horizontal] add:
      marked-production
```

Horizontal addition looks inside productions: it adds any marked part of an internal choice by either introducing one or enhancing the existing one. This allows to skip pre-transformational **vertical** and post-transformational **horizontal** steps for productions with a top-level choice, which is the most common use of this transformation. However, it is useful to have a command at hand that is capable of adding alternatives to any particular place of any grammar production.

Markers must denote the new part: i.e., the production without the marked part must be present in the grammar, and if it is, the result will contain a production with the marked part instead. Obviously, the markers itself do not end up in the grammar.

Example Given the input:

```
N:
      a
      b
```

After using this transformation:

```
addH(
 N:
      ⟨"x"⟩
      a
      b
);
```

The result will look like this:

```
N:
      "x"
      a
      b
```

Example Given the input:

```
expr:
      "-"? int
```

After using this transformation:

```
addH(
 expr:
      (⟨"+"⟩ | "-")? int
);
```

The result will look like this:

```
expr:
      ("+" | "-")? int
```

7.8.2 appear

The purpose of this transformation operator is to insert an nillable symbol (i.e., reducible to an empty sequence) at any place in any existing grammar production. It takes a production as an input. Inside that production, one nillable subexpression should be marked.

7.8.2.1 Syntax

```
appear:
      marked-production
```

7.8.2.2 Example

Given the input:

```
foo:
      bar
```

After using this transformation:

```
appear(
 foo:
      bar ⟨qux?⟩
);
```

The result will look like this:

```
foo:
      bar qux?
```

7.8.3 widen

The grammar is rewritten by local transformations such that the language generated by the grammar (or the denotation according to any other semantics for that matter) is increased. The known rewriting rules affect the use of epsilon and regular expression operators. There are two expression arguments: one to be matched, and another one that replaces the matched expression. The scope of the transformation can be limited.

7.8.3.1 Syntax

```
widen:
      expression expression in::scope?
```

The widening relation is defined as follows:

$$x \longrightarrow x^? \text{ or } x^+ \text{ or } x^*$$
$$x^? \longrightarrow x^*$$
$$x^+ \longrightarrow x^*$$

It is trivial to prove that for each case the expression on the left is included in the expression on the right, but not otherwise. For going the other way **narrow** transformation is used. For shaping an expression into a completely equivalent one, use **massage**.

7.8.3.2 Example

Given the input:

```
[main] program:
      fun::function
```

After using this transformation:

```
widen (
 function,
 function+
 in [main]);
```

The result will look like this:

```
[main] program:
      fun::(function+)
```

7.8.4 upgrade

Upgrading is a special variation of replacement and a slightly more powerful and liberal form of **folding**. This operator replaces an expression by a nonterminal that can be evaluated to it. The first parameter is the scope production with an expression marked. The second parameter is one of that nonterminal's definitions, which right hand side equals that expression.

7.8.4.1 Syntax

```
upgrade:
      marked-production production
```

7.8.4.2 Example

Given the input:

```
a:
      d e c
```

```
b:
      d e
```

```
b:
      f g
```

After using this transformation:

```
upgrade(
 a:
      ⟨b⟩ c
 b:
      d e
);
```

The result will look like this:

```
a:
      b c
```

```
b:
      d e
```

```
b:
      f g
```

7.8.5 unite

Two nonterminals, say x and y, are merged (possibly recursively). That is, the definitions of x and y (i.e., their productions) are merged in one definition while preserving the nonterminal y and replacing all occurrences of x (in the definition of x and anywhere else) by y.

7.8.5.1 Syntax

```
unite:
      add::nonterminal to::nonterminal
```

7.8.5.2 Example

Given the input:

```
foo:
      "a"
```

```
foo:
      "b"
```

```
bar:
      "d"
```

After using this transformation:

```
unite(bar, foo);
```

The result will look like this:

```
foo:
      "a"
```

```
foo:
      "b"
```

```
foo:
      "d"
```

7.9 Language decreasing transformations

Here is a list of the XBGF transformations that shorten the grammar (decrease semantics).

```
decreasing-transformation:
      remove
      disappear
      narrow
      downgrade
```

7.9.1 remove

Productions can be removed from existing, vertical definitions. The remaining definition must not become empty, i.e., undefined. There is the undefine operator that can be applied in that case. There is also a horizontal mode of removing branches from choices.

7.9.1.1 removeV

Syntax

```
[vertical] remove:
      production
```

Vertical removal operates on the level of productions: it takes away one production for some nonterminal leaving at least one more in the grammar.

Example Given the input:

```
expr:
      int
```

```
expr:
      id
```

After using this transformation:

```
removeV(
 expr:
      id
);
```

The result will look like this:

```
expr:
      int
```

7.9.1.2 removeH

Syntax

```
[horizontal] remove:
      marked-production
```

Horizontal removal looks inside productions: it removes any marked part of an internal choice, getting rid of the choice altogether if necessary (say, if the removed part was one of two alternatives). This allows to skip pre-transformational **vertical** and post-transformational **horizontal** steps for productions with a top-level choice, which is the most common use of this transformation. However, it is useful to have a command at hand that is capable of removing alternatives from any particular place of any grammar production.

Markers must denote the part to be removed: i.e., the production with the marked part must be present in the grammar, and if it is, the result will contain a production without the marked part instead. Obviously, the markers itself do not end up in the grammar.

Example Given the input:

```
foo:
      "x"
      bar
      wez
```

After using this transformation:

```
removeH (
 foo:
      ⟨"x"⟩
      bar
      wez
);
```

The result will look like this:

```
foo:
      bar
      wez
```

Example Given the input:

```
expr:
      ("+" | "-")? int
```

After using this transformation:

```
removeH(
 expr:
      (⟨"+"⟩ | "-")? int
);
```

The result will look like this:

```
expr:
      "-"? int
```

7.9.2 disappear

This operator behaves like **project**, but only allows for nillable elements (optional, star) to disappear.

7.9.2.1 Syntax

```
disappear:
      marked-production
```

7.9.2.2 Example

Given the input:

```
foo:
      bar wez* qux
```

After using this transformation:

```
disappear(
 foo:
      bar ⟨wez*⟩ qux
);
```

The result will look like this:

```
foo:
      bar qux
```

7.9.3 narrow

The grammar is rewritten by local transformations such that the language generated by the grammar (or the denotation according to any other semantics for that matter) is decreased. The known rewriting rules affect the use of epsilon and regular expression operators. There are two expression arguments: one to be matched, and another one that replaces the matched expression. The scope of the transformation can be limited.

7.9.3.1 Syntax

```
narrow:
      expression expression in::scope?
```

The narrowing relation is defined as follows:

$$x^? \longrightarrow x$$

$$x^+ \longrightarrow x$$

$$x^* \longrightarrow x \text{ or } x^? \text{ or } x^+$$

It is possible to prove that for each case the expression on the right is included in the expression on the right, but not otherwise. For going the other way **widen** transformation is used. For shaping an expression into a completely equivalent one, use **massage**.

7.9.3.2 Example

Given the input:

```
program:
      fun::(function*)
```

After using this transformation:

```
narrow(
 function*,
 function+);
```

The result will look like this:

```
program:
      fun::(function+)
```

7.9.4 downgrade

Replaces a nonterminal with one of its definitions. The first parameter is the scope production with one of the nonterminals marked. The second parameter is one of that nonterminal's definitions, which right hand side will be used for replacement.

The XBGF processor looks for the first production with the marked part (but without the markers). If it is found, the marked part is replaced with the right hand side of the second argument production.

7.9.4.1 Syntax

```
downgrade:
      marked-production production
```

7.9.4.2 Example

Given the input:

```
a:
      b c
```

```
b:
      d e
```

```
b:
      f g
```

After using this transformation:

```
downgrade(
 a:
      ⟨b⟩ c
 b:
      d e
);
```

The result will look like this:

```
a:
      d e c
```

```
b:
      d e
```

```
b:
      f g
```

7.10 Refactorings in term-oriented semantics

We may refer to the semantics of a grammar as the language (set of strings) generated by the grammar, as it is common for formal languages — for context-free grammars, in particular. With the string-oriented semantics in mind, all transformations mentioned above in folding and refactoring sections are semantics-preserving. To give an example where different semantics are needed consider the scenario of aligning a concrete and an abstract syntax.

When necessary, we may apply the algebraic interpretation of a grammar, where grammar productions constitute an algebraic signature subject to a term-algebraic model. In this case, the terminal occurrences in any given production do no longer carry semantic meaning; they are part of the function symbol. (Hence, abstract and concrete syntaxes can be aligned now.)

Some transformations that were effortlessly semantics-preserving w.r.t. the string-oriented semantics, require designated bijective mappings w.r.t. the term-oriented semantics, e.g., fold/unfold manipulations, but generally, the term-oriented semantics admits a larger class of semantics-preserving transformations than the string-oriented one.

The following section gathers those transformations that would have been considered refactorings if we only used term-oriented semantics. From the string-oriented point of view they revise semantics and can be deemed as neither grammar lengthening nor grammar shortening transformations.

```
concrete-revising-transformation:
      abstractize
      concretize
      permute
```

7.10.1 abstractize

As **project**, but with an additional precondition that the part to be removed should consist of terminals. This is checked automatically by the XBGF engine: if the precondition fails, the transformation is inapplicable.

7.10.1.1 Syntax

```
abstractize:
      marked-production
```

7.10.1.2 Example

Given the input:

```
A:
      b "x" c "y"
```

After using this transformation:

```
abstractize (
 A:
      b ⟨"x"⟩ c "y"
);
```

Will look like this:

```
A:
      b c "y"
```

7.10.2 concretize

Just as **abstractize** is a preconditioned version of **project**, this operator is a variation of **inject**. The XBGF engine checks if the marked part only consists of terminal symbols: if yes, injection happens; if not, the transformation is inapplicable.

7.10.2.1 Syntax

```
concretize :
      marked-production
```

7.10.2.2 Example

Given the input:

```
A:
      b c
```

After using this transformation:

```
concretize (
 A:
      b ⟨"x"⟩ c
);
```

Will look like this:

```
A:
      b "x" c
```

7.10.3 permute

The argument production defines the intended result of the transformation — a production that has the same components in the sequential composition, but in a different order, when compared to the corresponding production in the input grammar with the same defined nonterminal and the same label, if any.

7.10.3.1 Syntax

```
permute:
      production
```

7.10.3.2 Example

Given the input:

```
a:
      b d* c
```

After using this transformation:

```
permute(
 a:
      b c d*
);
```

Will look like this:

```
a:
      b c d*
```

7.11 Semantics revising transformations

Some grammar differences may require more arbitrary replacements, that cannot be expressed as semantics-preserving even in abstract syntax. These include projections or injections of non-optional nonterminals, adding definitions for bottom nonterminals, performing volatile replacements.

Whenever a transformation from this group is used in a convergence path, it signals either about a construction point where the grammar engineer is having a temporary

shortcut to be substituted later with a longer sequence of more accurate manipulations, or an inevitable error in the BGF that needs fixing (preferably in the original artefact—specification, compiler sources, etc).

```
abstract-revising-transformation:
        define
        undefine
        redefine
        inject
        project
        replace
```

7.11.1 define

An undefined nonterminal is resolved by this operator. The nonterminal must be in use. The introduce operator should be used when a fresh nonterminal is to be defined. The add operator should be used when an existing definition is to be extended.

7.11.1.1 Syntax

```
define:
        production+
```

7.11.2 undefine

This operator allows to undefine a nonterminal, i.e., remove all its productions. The nonterminal must have using occurrences elsewhere than just in its own definition. If there are no such using occurrences, then the less disruptive eliminate operator is to be used.

7.11.2.1 Syntax

```
undefine:
        nonterminal+
```

7.11.3 redefine

Redefine is a **replace** variant that works on production level. When this transformation is executed, all current productions for the nonterminal are dropped, and all the given ones are added to the grammar.

This transformation is nothing more than syntactic sugar for an **undefine** followed directly with **define**. Having it as a separate type of transformation allows to more clearly distinguish completing the grammar with absent definitions (usually as initial correction step) and brutal nonterminal-level replacements (semantic revising).

7.11.3.1 Syntax

```
redefine:
      production+
```

7.11.4 inject

The argument production defines the intended result of the transformation — a production that has additional components in the sequential composition, when compared to the corresponding production in the input grammar with the same defined nonterminal and the same label, if any.

7.11.4.1 Syntax

```
inject:
      marked-production
```

7.11.4.2 Example

Given the input:

```
a:
      b d
```

After using this transformation:

```
inject(
 a:
      b ⟨c⟩ d
);
```

Will look like this:

```
a:
      b c d
```

7.11.5 project

The argument production defines the intended result of the transformation — a production that has fewer components in the sequential composition, when compared to the corresponding production in the input grammar with the same defined nonterminal and the same label, if any.

7.11.5.1 Syntax

```
project:
      marked-production
```

7.11.5.2 Example

Given the input:

```
a:
      b c d
```

After using this transformation:

```
project(
 a:
      b ⟨c⟩ d
);
```

Will look like this:

```
a:
      b d
```

7.11.6 replace

This operator provides a last resort to grammar editing. It basically provides access to free editing without any semantically meaningful preconditions. There are two expression arguments: one to be matched, and another one that replaces the matched expression. The scope of the transformation can be limited.

7.11.6.1 Syntax

```
replace:
      expression expression in::scope?
```

7.11.6.2 Example

It is possible to use **replace** in sophisticated context, cutting out any pieces of the grammar to be replace with something different. For instance, given the input:

```
a:
      b c
      b d
      b e
```

After using this transformation (suppose we do not have **factor**):

```
replace (
 ((b c) | (b e)),
 b (c | e));
```

Will look like this:

```
a:
      b (c | e)
      b d
```

7.12 Decorative transformations

Last but not least, there are four refactorings that are very specific to the BGF itself. Not all grammar definition formalisms have labelled productions, but since we do, we also need two transformation steps made possible: to **designate** an already available production with a label, and to **unlabel** an existing labelled production. We strip selectors from subexpressions with **anonymize** and add them, naturally, with **deanonymize**.

```
decorative-transformation:
      designate
      unlabel
      deanonymize
      anonymize
```

7.12.1 designate

An unlabeled production is labeled. The argument production is the intended result, i.e., the labeled production—the transformation refuses to work if the argument production contains no label.

Labelling transformations serve two roles usually: they can be used directly to make the labels in both grammars agree so that they can converge; or they are used to mark the target for the transformations that follow them and perform local manipulations.

7.12.1.1 Syntax

```
designate :
      production
```

7.12.1.2 Example

Given the input:

```
expr:
      int
```

After using this transformation:

```
designate (
 [intexpr] expr:
      int
);
```

Will look like this:

```
[intexpr] expr:
      int
```

7.12.2 unlabel

This is a reverse of **designate** that strips an existing production from a label.

7.12.2.1 Syntax

```
unlabel:
      label
```

7.12.2.2 Example

Unlike the previous one, this transformation relies on the fact that all labels are unique within a grammar. This assumption allowed us to simplify the calling syntax. So, given the input:

```
[intexpr] expr:
      int
```

After using this transformation:

```
unlabel([intexpr]);
```

Will look like this:

```
expr:
      int
```

7.12.3 deanonymize

While labels are only used to name individual productions, selectors can name arbitrary parts of any BGF expression. This command allows to add a selector to the existing production. To avoid disambiguations, the whole production is required as an argument, with the newly introduced part being marked.

7.12.3.1 Syntax

```
deanonymize:
      marked-production
```

7.12.3.2 Example

Selectors can be introduced one at a time or in batch, but each one must be marked separately. For instance, given the input:

```
A:
      first::"a" "a" "a"
```

After using this transformation:

```
deanonymize(
 A:
      first::"a" ⟨second::"a"⟩ ⟨third::"a"⟩
);
```

Will look like this:

```
A:
      first::"a" second::"a" third::"a"
```

7.12.4 anonymize

This operator is the reverse of **deanonymize** that strips one (marked) selector from an existing definition.

7.12.4.1 Syntax

Given the input BGF and a clear goal to strip all selectors, it becomes trivial to generate a list of **anonymize** commands that, if executed on the same grammar, would produce a selector-free yet structurally equivalent grammar. We used such a generator called strips in the FL case study as the final stage to converge the abstract syntax (with selectors) with the concrete syntax (with terminals), see section 4.9.

```
anonymize:
      marked-production
```

7.12.4.2 Example

Given the input:

```
[binary] expr:
      "(" expr op::binary_op expr ")"
```

```
[unary] expr:
      op::unary_op expr
```

After using this transformation:

```
anonymize (
 [unary] expr:
      ⟨op::unary_op⟩ expr
);
```

Will look like this:

```
[binary] expr:
      "(" expr op::binary_op expr ")"
```

```
[unary] expr:
      unary_op expr
```

7.13 dump

This is a debugging tool for XBGF. When the **dump** command is encountered, the transformation sequence stops and the grammar in its current state is dumped to the file xbgf.log.

The contents of xbgf.log can be used as an input for grammar comparator or for copy-pasting productions and expressions from the grammar to the construction point of the XBGF sequence.

7.13.1 Syntax

```
dump:
      ε
```

7.14 rename

Labels, nonterminals, selectors and terminals can be renamed. Being in line with the fundamental notion of renaming, such renaming must be done consistently throughout the entire grammar, without introducing any clashes. There is one justifiable exception. That is, arguably, the scope of selectors is the level of production as opposed to necessarily the entire grammar. Hence, selectors can be renamed in such a scope, optionally.

7.14.1 renameL

7.14.1.1 Syntax

```
[label] rename:
      from::label to::label
```

Renaming labels is a semantic preserving grammar transformation pretty-printed as **renameL**. Given two label names, it simply searches the grammar for productions with the original label and re-designates them with the new one.

renameL is a simple syntactic sugar for the specific combination of **unlabel** and **designate**.

7.14.1.2 Example

Given the input:

```
[constant] expr:
      int
```

```
[binary] expr:
      expr op::binary_op expr
```

```
[unary] expr:
      op::unary_op expr
```

After using this transformation:

```
renameL([binary], [binary_expr]);
```

Will look like this:

```
[constant] expr:
      int
```

```
[binary_expr] expr:
      expr op::binary_op expr
```

```
[unary] expr:
      op::unary_op expr
```

7.14.2 renameN

7.14.2.1 Syntax

```
[nonterminal] rename:
      from::nonterminal to::nonterminal
```

Similarly, this transformation can be used to rename nonterminals. This variant is a syntactic sugar for the specific combination of **inline** and **extract**, it is a semantic preserving grammar transformation that is pretty-printed as **renameN**.

7.14.2.2 Example

Given the input:

```
[constant] expr:
      int
```

```
[binary] expr:
      expr op::binary_op expr
```

```
[unary] expr:
      op::unary_op expr
```

After using this transformation:

```
renameN(expr, exp);
```

Will look like this:

```
[constant] exp:
      int
```

```
[binary] exp:
      exp op::binary_op exp
```

```
[unary] exp:
      op::unary_op exp
```

7.14.3 renameS

7.14.3.1 Syntax

```
[selector] rename:
      in::label? from::selector to::selector
```

Selectors can also be renamed by a semantic preserving grammar transformation that is pretty-printed as **renameS**. This variant is a syntactic sugar for the specific combination of **anonymize** and **deanonymize**.

7.14.3.2 Example

Given the input:

```
[constant] expr:
      int
```

```
[binary] expr:
      expr op::binary_op expr
```

```
[unary] expr:
      op::unary_op expr
```

After using this transformation:

```
renameS(op, operator);
```

Will look like this:

```
[constant] expr:
      int
```

```
[binary] expr:
      expr operator::binary_op expr
```

```
[unary] expr:
      operator::unary_op expr
```

7.14.4 renameT

7.14.4.1 Syntax

```
[terminal] rename:
      from::terminal to::terminal
```

Renaming terminals breaks string-oriented (concrete) semantics, but is still possible. This variant is pretty-printed as **renameT**, its behaviour is essentially that of a sequential composition of **abstractize** and **concretize**, but its meaning is different: it changes an entity that is already present in the grammar, not removes or adds anything.

7.14.4.2 Example

Given the input:

```
x:
      "x"
```

After using this transformation:

```
renameT("x", "y");
```

Will look like this:

```
x:
      "y"
```

7.15 reroot

Redefine the roots (start symbols) of the grammar. The empty set of roots is interpreted to abbreviate the complete set of nonterminals used or defined by a grammar.

7.15.1 Syntax

```
reroot:
      root::nonterminal*
```

7.16 Compatibility

In order to establish the relation between BGF that is being used in the actual working system and BNF that is being reported here, we apply the whole process of grammar convergence to these two grammars. We presume to conclude that this BNF dialect plus empty grammars plus root elements is the same as BGF language plus indentation rules plus layout rules plus terminal symbols.

BNF was defined as a base-line grammar for a pretty-printer and therefor defines concrete syntax. BGF is extracted from the corresponding XML Schema and contains abstract syntax annotated with selectors. We choose to converge them closer to abstract syntax (BGF).

First, we resolve the name mismatch and compensate for the lack of notion of "root elements" in BNF:

```
renameN(expression, symbol);
reroot();
disappear(
 grammar:
      ⟨root::nonterminal*⟩ production*
);
```

The only remaining transformations applied to the BGF grammar are these:

```
narrow(
 production*,
 production+);
inline(terminal);
inline(nonterminal);
inline(selector);
vertical( in symbol );
vertical( in value );
```

As we can see, the first of these transformations, **narrow**, shortens the grammar, but its semantics only means that we do not want to have empty samples while an empty grammar is still acceptable in general. Since this is data refinement, a semantic decreasing transformation is used without hesitation. The rest of the transformational sequence are trivial refactorings (verticalisations are already performed in the example from the previous section).

The BNF source undergoes the following transformation for stripping it from lexical details:

```
project(
 right-hand-side:
      ⟨NEWLINE⟩ (⟨INDENT⟩ symbol+ ⟨NEWLINE⟩)+ ⟨NEWLINE⟩
);
```

Before the rest of the concrete syntax (i.e., the terminals) is stripped away, we need to add some labels that correspond to the selectors of its BGF counterpart:

```
extract (
 value:
      "STRING"
      "INT"
);
vertical ( in symbol );
vertical ( in value );
designate (
 [epsilon] symbol:
      "ε"
);
designate (
 [empty] symbol:
      "EMPTY"
);
designate (
 [any] symbol:
      "ANY"
);
designate (
 [string] value:
      "STRING"
);
designate (
 [int] value:
      "INT"
);
designate (
 [value] symbol:
      value
);
```

Now we can remove all terminals from the grammar without disrupting its structure (i.e., various alternatives in symbol will not collide and vanish during normalisation phase). Since we do not want to make a distinction and plan for all terminals to be removed, the following transformation script is generated by a special tool executed automatically from LCI (see section 4.9).

```
abstractize (
 label:
      ⟨"["⟩ STR ⟨"]"⟩
);
```

```
abstractize (
 production:
      label::label? nonterminal::STR ⟨":"⟩ right-hand-side
);
```

```
abstractize(
 [epsilon] symbol:
      ⟨"ε"⟩
);
```

```
abstractize(
 [empty] symbol:
      ⟨"EMPTY"⟩
);
```

```
abstractize(
 [any] symbol:
      ⟨"ANY"⟩
);
```

```
abstractize(
 [terminal] symbol:
      ⟨"""⟩ STR ⟨"""⟩
);
```

```
abstractize(
 [selectable] symbol:
      selector::STR ⟨"::"⟩ symbol
);
```

```
abstractize(
 [sequence] symbol:
      ⟨"("⟩ symbol+ ⟨")"⟩
);
```

```
abstractize(
 [choice] symbol:
      ⟨"("⟩ symbol (⟨"|"⟩ symbol)* ⟨")"⟩
);
```

```
abstractize(
 [optional] symbol:
      symbol ⟨"?"⟩
);
```

```
abstractize(
 [plus] symbol:
      symbol ⟨"+"⟩
);
```

```
abstractize(
 [star] symbol:
      symbol ⟨"*"⟩
);
```

```
abstractize(
 [marked] symbol:
      ⟨"⟨"⟩ symbol ⟨"⟩"⟩
);
```

```
abstractize(
 [string] value:
      ⟨"STRING"⟩
);
```

```
abstractize(
 [int] value:
      ⟨"INT"⟩
);
```

Finally we run a grammar comparator to see what is left and notice one mismatch that is easily fixed with **massage**, as well as the right hand side still having $(symbol^+)^+$ instead of just $symbol$. This corresponds to the design decision that treats top-level choices and top-level sequences differently in BNF to make them more appealing to the eye by avoiding unnecessary parenthesizing. The very specific **upgrade** command is run twice here to fold first the sequence and then the choice.

```
massage(
 symbol symbol*,
 symbol+);
upgrade(
 right-hand-side:
      ⟨symbol⟩+
 [sequence] symbol:
      symbol+
);
upgrade(
 right-hand-side:
      ⟨symbol⟩
 [choice] symbol:
      symbol+
);
inline(right-hand-side);
```

After that, the grammars fully converge. The conclusion is that BNF language plus empty grammars plus root elements is the same as BGF language plus indentation rules plus layout rules plus terminal symbols.

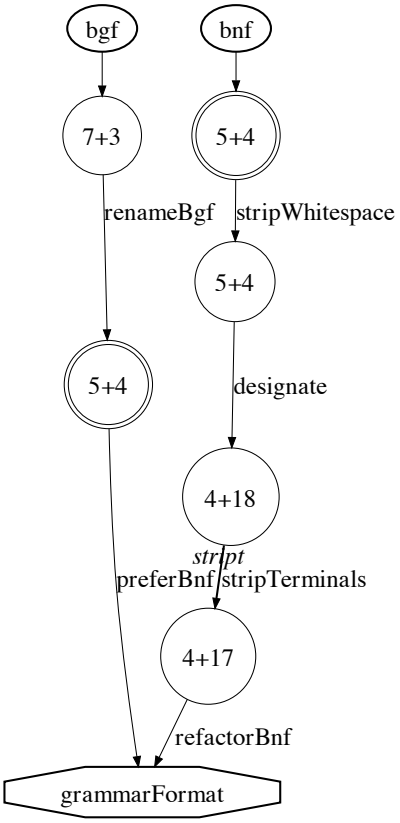


Figure 7.1: Full convergence diagram for BNF and BGF. The top nodes are sources, the bottom node is the target, the arc labels are separate XBGF scripts, the nodes contain numbers of name mismatches and structural mismatches between each step and the synch point (marked as a double circle).

Chapter 8

Conclusion

An ideal world is left as an exercise to the reader.

Paul Graham, 1993 [80]

8.1 Summary

The conceptual contributions of this thesis are listed by the fields of research.

Grammar recovery. A successful endeavour has been made to generalise the steps needed for recovering grammars from real software artefacts with embedded grammar knowledge.

Grammar extraction. The possibility has been shown to automate grammar extraction and to make those extractors so advanced that they operate on a set of rules specified by a language engineer beforehand. Based on such rules, the extractors detect and repair presentation inconsistencies in typical existing language artefacts such as standards that many assume are flawless.

Grammar convergence. We presented the methodology that allows a language engineer to take two or more grammars that are assumed to be related (equal, one covered by another, etc) and by applying a combination of described methods and tools to surface the relationships among them. Such relationships are formally represented by sequences of grammar transformation steps.

Grammar transformation. After careful examination of the existing achievements in this field, an operator suite called XBGF was developed. To the best of our knowledge and experience of working with different transformational frameworks, XBGF surpasses previously existing technology in automation, granularity, maintainability. The proposed set of operators fits the domain of grammar transformation closely, providing separate specialised commands for common use patterns.

Language documentation. We reverse engineered a large library of language specifications and designed a general document schema that describes a format of a typical language standard or manual. We presented a description of an infrastructure that needs to evolve around this schema to be integrated into the language evolution process. The transformational language and all the tools have been successfully prototyped, although the full scale application of this methodology to industrial standards is beyond the scope of the thesis and was left for future work.

8.2 Research contributions

Scientific contributions of the thesis are summarised in the following list:

Deeper insight into grammar recovery field. The case study we have done for C# has shown that the problems encountered in Cobol and C# manuals are quite similar since similar methods turned out to be applicable to such different cases. Published as [257].

Grammar convergence methodology. We presented a lightweight verification method for transforming grammars until they become identical. We proposed to treat the resulting transformation chains as relationships among the source grammars. A case study converging six related Java grammars was also performed successfully. Published as [166, 167].

Principles of automated grammar extraction. The rule-based structure of the extractor that we used in the convergence case study, proved to be a viable grammar recovery infrastructure. By generalising every problem and lifting it to the level of a pattern applicable to a class of similar problems, we automated a significant part of the error recovery process. The incremental development process of such an extractor is repeatable for other notations and sources.

Detailed analysis of existing language documents. More than 40 language specifications and reference manuals were analysed and compared for their notations and structure. Despite the diversity of notation for syntactic definitions, most EBNF dialects have the same basic principles and the same expressivity, so automated migration from one to another does not pose any technical challenge. However, the ways language documents are organised and structured, are sometimes fundamentally different.

Unified language documentation data model. Based on our analysis, we proposed a general schema that is useful for many language documents. Specific benefits of its application are straightforward grammar extraction, solid separation of concerns, advanced opportunities for testing and document querying.

Language documentation infrastructure prototype. We formulated the principles of an integrated approach to language documentation design, evolution and maintenance. We also proposed a supporting infrastructure to implement it. Published as [143, 258].

Operator suite for grammar transformation. After using grammar transformation in several convergence scenarios, we designed an advanced and detailed grammar transformation operator suite. Having a separate operator for each specific case fits the domain closer by allowing better applicability conditioning and property proving. Having a wide choice of well-advocated operators allows grammar engineers to limit themselves to refactoring steps whenever possible and not backslide to local replacements as is common in many frameworks. As a result, sequences of grammar transformation steps are easier to comprehend.

8.3 Engineering deliverables

We did not limit ourselves to exploring ideas about grammar recovery, transformation and convergence. All aspects were worked out up to the point of having a practical prototype, an reusable tool, a language definition or a useful documentation. This approach yielded several primarily engineering deliverables that are useful and reusable for a wide range of tasks.

The following list describes all types of engineering deliverables, and each item has a subsection associated with it.

Grammars. Designed or recovered, grammars are useful to parse code or for further investigations into their properties.

Languages. The domain-specific languages form the backbone of our proposed framework.

Documents. Each of the language documents both serves as a manual and exemplifies language documentation facilities.

Grammar relationships. Grammar convergence case studies directly result in established relationships among existing grammars.

Tools. We have developed mappers between existing notations and our DSLs.

8.3.1 Grammars

The thesis delivered the following grammars that are electronically accessible through the internet at the SLPS Grammar Zoo (`http://slps.sf.net/zoo`).

C#. We applied grammar recovery skills to the international standard of C# by ECMA [225] and ISO [114]. The recovering transformations were performed with the Grammar Deployment Kit [149], a previously available framework. The resulting grammar is available in Syntax Definition Formalism [86] as well as in a browsable form generated from it.

Java: JDK 1.0 “Oak”, J2SE 1.2 “Playground”, J2SE 5.0 “Tiger”. The case study that we completed with our own infrastructure delivered three Java grammars recovered from three editions of the Java Language Specification [77, 78, 79]. The

extractor was designed to automatically correct presentation inconsistencies found in the language documents.

Factorial Language. Nine small grammars in different forms were developed as examples when prototyping the method of grammar convergence. The author of this thesis designed four FL grammars in ANTLR [202], TXL [39], ASF+SDF [22] and Ecore [59]. Two other FL grammars were derived automatically by existing model transformation or grammar transformation tools.

C: ISO/IEC 9899:1999(E), 9899:TC2, 9899:TC3. Since grammar notations used in JLS and in [117, 118, 119] are very similar, it was possible to re-use the principles of the Java grammar extractor for two different ISO standards containing grammars of C.

C++: ISO/IEC 14882:1998(E), SC22/WG21 N2723. Grammar notations used in JLS and ISO C standards are similar also to [111] and [120]. We re-used exactly the same tool that was employed for extracting ISO C grammars, for the extraction of ISO C++ grammars. Both C and C++ grammars will be used by SLPS contributors in future work on language convergence, and are already being used by others as a stable extraction source[1].

These grammars differ at a level of possible practical use. Technically speaking, the C# grammar is a Level 3 grammar, i.e., it is complete enough for a parser to be generated from it and it was tested on a small codebase. The three Java grammars are Level 2 grammars, i.e., they only define language syntax and lack the lexical parts. All nine FL grammars are at Level 5, i.e., they have been directly derived from compiler sources, parser definitions, schemata and similar artefacts.

8.3.2 Languages

During this research, we designed, developed and implemented six new languages specific for grammarware domain. They are used for storing grammars, documents, transformations, configurations and parse trees, and are generally useful and applicable to many adjacent topics of grammar engineering, re-engineering and reverse engineering. We chose to express these DSLs in XML Schema.

BGF, BNF-like Grammar Format. This language emerged as a compromise for storing grammar knowledge that is traditionally expressed in BNF, EBNF and dialects thereof. BGF provides the usual functionality of specifying terminal and nonterminal symbols and combining them sequentially or alternatively, plus extra features like selectors, i.e., named sub-expressions for which we found use in many extraction and transformation activities.

[1] E.g., Lukas Renggli (University of Bern): http://twitter.com/renggli/status/17401122412.

XBGF, BGF Transformations. This is the basic transformation language used in grammar convergence, we utilise it to express the sequences of calls to grammar transformation operators. Whenever an operator requires a grammar production or a grammar expression as a parameter, BGF is re-used.

BTF, BGF Tree Format. We used this language for coupled transformations, when we needed to specify not only grammars as such, but also their instances, i.e., parse trees. The trees contain the original productions that were used for their generation.

LDF, Language Document Format. After analysing 40+ language documents of different nature, we inferred a general metamodel to cover all observed variations. This metamodel is expressed and exploited in the form of LDF. The language defines domain concepts like code samples of syntax description sections, as well as the rules for combining them.

XLDF, LDF Transformations. We extended XBGF to work on language documents: each XLDF command represents a language evolution step (such as adding a new language construct) or a language documentation evolution step (such as rearranging sections).

LCF, LCI Configuration Format. Language convergence infrastructure needed a separate DSL to represent the starting configuration and the desired outcome of a convergence scenario. LCF concepts include the notions of an extractor, a generator, a branch of convergence, a convergence phase and others.

8.3.3 Language documents

The thesis delivered the following language documents:

XBGF. This manual served as a showcase for our language documentation framework for language documentation life cycle, i.e., storage, evolution, transformation, testing and convergence. Its core was extracted in a completely automated manner from the XML Schema definition of the language to form the starting LDF document. Then it was transformed by a chain of XLDF commands that completed the text with side remarks and inserted pretty-printed test cases. The resulting LDF document was used directly to generate the grammar transformation chapter of the thesis (see chapter 7 that is completely automatically generated using our framework). The online hyperlinked version of the XBGF manual was generated from the same LDF document as well.

XLDF. The XML Schema definition of XLDF was used as an extraction source for the XLDF manual, and several XLDF scripts were used to refactor the resulting LDF document to its final form. The corresponding thesis section was generated automatically from this final LDF document using the general LDF to LaTeX pretty-printer.

LDF. The self-definition of LDF in LDF was generated from the XML Schema definition of LDF, and several XLDF scripts were used to refactor the resulting LDF document

to its final form. The corresponding thesis section was generated automatically from this final LDF document using the general LDF to LaTeX pretty-printer.

LCF. The LCF manual focuses on the domain of configuration of the overall infrastructure and mainly on the semantics of the language constructs. For instance, if a tool is defined with a particular tag, the manual documents, when it will be called and with which parameters. The LCF manual is extracted from a corresponding XML Schema definition, beautified by transformations and pretty-printed to LaTeX. It contains considerably less code fragments than the other manuals and more information about informally described semantics.

8.3.4 Grammar relationships

The following grammar relationships were recovered by performing the corresponding case studies:

Factorial Language. Nine different grammars of the same language were converged, extracted from ANTLR, DCG, SDF, TXL, Ecore, XSD and Java sources. The transformation steps representing the relationships uncovered all the peculiarities of each of the source formats.

Java Language Specification. Six different grammars of the same language were converged, extracted from three versions of Java Language Specification by pairs of a grammar intended for reading and a grammar intended for implementation. The recovered relationships were compared with the explicit and implicit claims such as backward compatibility.

BNF-like Grammar Format. The abstract definition in the schema of BGF was converged with its concrete representation used by a pretty-printer. This case study was so small that we could a priori predict exactly the differences that we encountered during convergence. In such a situation, our method was used as complementary framework testing.

8.3.5 Tools

We do not list general tools like the transformation engine here, but rather assume that the claims from other sections should be interpreted in a wide sense: thus, "we have a language XBGF" means that there is a document schema for it, as well as the appropriate language processor that can run XBGF commands, as well as libraries for parsing XBGF files. However, we do distinguish between extractors and pretty-printers. **Extractors** establish mappings between certain classes of grammar artefacts and the BGF-driven infrastructure, i.e., they transform other formats to BGF, BTF or LDF. **Pretty-printers** were developed to map the grammars, documents and scripts stored in XML in a disciplined fashion, to other formats, i.e., to generate alternative textual representations of them (mostly plain text, hypertext or LaTeX). Only those tools attributed to the author of the thesis are listed. All tools are easily executable via wrapper scripts with one or

two necessary parameters, although internally they can consist of multiple calls to various utilities from different frameworks.

Extractors:

HTML to BGF. This advanced extractor had to work on a manually and loosely hypertext source. It comprised a set of generalised rules in a pattern form that it tried to apply for automated recovery.

ANTLR to BGF. In order to be able to extract grammars from ANTL parser definitions, we re-used the standard ANTLR grammar for ANTLR grammars by attaching appropriate semantic actions to it.

SDF to BGF. We encoded the necessary traversal functions for crawling the parse trees of SDF grammars and producing BGF and reused the SDF module and the XML module from the standard package of the Meta-Environment.

TXL to BGF. We re-used the TXL grammar for TXL grammars; the mapping between TXL XML and our XML (i.e., BGF) was straightforwardly encoded in XSLT.

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

AsFix to BTF. We encoded the necessary traversal functions for crawling the parse trees of AsFix parse trees and producing BTF and reused the AsFix module and the XML module from the standard package of the Meta-Environment.

XML Schema to LDF. The XSD has proven to be a suitable format for prototyping language documentation extraction, since it is capable of expressing a grammar-like structure, as well as annotating functionality. This extractorintertwines the XSD annotations with the extracted grammar productions.

LDF to BGF. Since we assume that any LDF document does contain grammar knowledge explicitly, i.e., in BGF, we use a special extractor to take out the BGF bits and compose a grammar from them.

Pretty-printers:

BGF to text. The main pretty-printer is the one that takes any BGF file and generates a purely textual serialisation of it suitable to be read by a human. Strictly speaking, this pretty-printer re-invents the concrete syntax for BGF.

BGF to TEX. In order to use BGF more extensively and more efficiently for examples in this thesis and in all related articles, we developed a separate pretty-printer that wraps the textual syntax of BGF in a proper LATEX environment.

LDF to TEX. Given a proper LDF, this pretty-printer generates a compilable standalone LATEX document. All textual sections are named properly, and all BGF productions included in the LDF are pretty-printed by re-using the BGF to text mapper.

LDF to PDF. We have two competing implementations that map LDF to PDF: one generates a LaTeX file and runs `pdflatex`, while the other one generates XHTML, transforms it to XSL:FO by re-using an openly distributed XSLT sheet, and runs `fop`. Apparently, the LaTeX-based path produces more appealing PDFs.

LDF to HTML. It is easy to represent an LDF document in a hypertext form. Due to the nature of XSLT, the transformation always produces a well-formed XHTML document (claim that is hard to make for LDF to TeX mapper). The XBGF manual intended for online distribution was created using this pretty-printer.

XBGF to text. We developed a BNF-resembling concrete syntax that we use for representing the transformation scripts in a textual form. The productions and expressions of BGF occur as parameters to XBGF commands.

XBGF to TeX. The appropriately parametrised LaTeX package `listings` allows to present XBGF scripts in an appealing presentation format. The textual component is pretty-printed by re-using the XBGF to text mapping.

All pretty-printers were implemented in XSL: either in XSLT or in XSL:FO.

8.4 Future work

Automated grammar recovery. For this project we have designed, developed, implemented and presented a heuristic rules-based grammar extractor. It was used for the Java Language Specification case study, where it successfully extracted six grammars from differently formatted HTML documents. However, it must be possible to generalise the rules further and make them universally applicable to a broad class of hypertext grammar descriptions. The development of robust extractors capable of solving extraction problems in a generic way is one of the important directions of future research and future contributions to the Software Language Processing Suite.

Dialect-parametric parsing of EBNF. During the research we have seen dozens of different dialects of BNF and EBNF and completed their study with concise conclusions about how the variations relate to the ISO EBNF standard. In the future, we plan to design a DSL for defining EBNF dialects and implement a generic parser based on it. By combining disciplined dialect definition with automated grammar extraction we will reach the milestone of treating all (E)BNF-based language specifications the same for the purposes of language recovery.

Suggestive convergence metrics. At its current state the method of grammar convergence relies on a set of guidelines that the language engineer follows, and on a set of comparison-based metrics that measure the convergence progress. The grammar comparator at this stage displays only apparent differences between the two grammars under consideration. One can suggest experimenting with aggressive forms of normalisation, search automation for specific constructs that are known to

be refactoring material, name matching automation algorithms. The boundaries for effectiveness and automation in this area have not been reached yet.

Inference of converging transformations. Our method of grammar convergence is based on automation of small atomic grammar transformation operators. All operator calls and parameters are performed by a human expert with the aid of metrics. It is also known that the opposite extreme is possible, where all transformation steps are derived automatically give the source and the target grammar. However, the inference algorithms used in such a scenario use a limited set of possible transformations (cf. the Levenshtein distance: deletion, insertion, substitution). The question of inferring transformation chains for the purpose of grammar convergence based on an advanced operator suite such as XBGF is an interesting research topic.

Generality of XBGF. At this point we cannot make a useful generality statement about the XBGF grammar transformation operator suite. Proving that all possible grammar refactorings are expressible in XBGF, or that any transformation step except error correction is expressible without substitution, is an interesting future research topic.

Formal proofs for XBGF operator properties. The properties of grammar transformation operators such as preservation of semantics were explained in an "intuitive" way. For operators dealing with concrete syntax, we also used arguments based on string languages. However, it can also be useful to have formal proofs for them in term algebra. This is a highly nontrivial task: the underlying grammar sources are heterogeneous and are based on different semantics—the meaning of a selector in one formalism can overlap with the meaning of a nonterminal symbol in another. In order to approach a solution to this problem, we intend to enhance the BGF with annotations and develop bidirectional mapping tools based on our existing grammar extractors. With this, one will not only be able to derive a grammar from a data model or a parser definition, but to consistently transform a data model or a parser definition to a grammar and back without any loss of information. When such mapping is established, we will have the information about the necessary amount of extra annotations, and then we plan to utilise automated theorem proving techniques here.

Language documentation. On the pages of this thesis we have proposed an infrastructure for language documentation life cycle. Several case studies were performed to support the prototype. All of them used a "clean" source format for extraction (XML Schema) and the beautification transformations were dictated by trivial differences between LDF and XSD. In the future we will take an existing specification, extract the LDF document from it, perform improving transformations until the acceptable level of usability is reached, examine it with various automated or semi-automated analysis techniques, resolve the inconsistencies found in the process, and regenerate the specification and satellite artefacts such as a test set.

Language document transformation. At this stage XLDF, the language for language document transformation, is a minimal one needed to complete the case studies we

have scheduled for this project. Just like with XBGF and even more so, this set of commands will need streamlining in the future, with constructive claims about its completeness and expressive power and verification thereof. There are also some open research questions about the criteria for language document transformations: e.g., will it be truly viable to have one suite for language evolution and language adaptation?

Reversibility of grammar transformations. Currently most of XBGF operators are implemented and presented in pairs: **add** and **remove**, **upgrade** and **downgrade**, **project** and **inject**, etc—in each such pair one of the operators is the inverse of the other. In future work, reversibility will be addressed and examined more thoroughly, with guaranteed properties and development of the prototype higher-order tools to generate transformations that reverse the effect of a transformational script given as input.

EBNF extension for language documentation. Given the analysis of EBNF dialects that we performed and the general language documentation engineering effort, we can see that a practically aimed, technology independent, modern and powerful formalism for describing everything needed for a modern language definition is still needed. Backus normal form, Wirth syntax notation and their variations were mostly used for expressing grammar knowledge during the last five decades, together with attempts to improve it, successful [112, 252], unsuccessful [31, 123, 250] and domain-specifically successful [175, 201]. We did admit that a standardised grammar definition formalism is needed, and even listed the factors that should affect its design, such as tool support, freedom from a fixed parsing technology, modularity, unambiguity. A possible future work direction is to define such a language and perform case studies with it. The extension would perhaps go even beyond "syntactic sugar", they can bear something like static semantics to express more structured details than possible with EBNF now.

Synchronisation points. We introduced the notion of a synchronisation point for targets with two branches. It should be possible to define a synchronisation point for all targets, possibly by letting the comparator decide which grammar is closer to the one under measurement. By doing so, an important issue of choosing what grammars are to converge first, will be solved.

Coupled transformation. Convergence of parse trees is a topic that was not fully developed in this project. We did mention on several occasions that it is important to extend grammar transformation operators' behaviour to the level of instances. For example, if the research on coupled transformation is completed, it will allow us to converge test sets together with grammars.

Bridging grammarware and modelware. Currently there is no consensus about what should count as programming and what as modelling. Together with classic programming languages such as C and classic modelling ones like UML there exist methodologies like EMFText that have both characteristics. Similarly, almost the same problems often arise before grammar experts and metamodel experts. There

are domains in which one or the other of these approaches prevails, and many where their comparison was not performed or turned out indecisive. For convergence, we will need to examine similar methods that compare, converge and calculate the difference between two models. Also a deeper analysis and comparison of grammarware and modelware is needed in order to draw conclusions about overall compatibility, applicability and the advantages of our methodology when applied to this adjacent area.

Samenvatting

Het in dit proefschrift beschreven onderzoek is gericht op de ontwikkeling, het herstel en het onderhoud van programmeertalen. Een programmeertaal kan gezien worden als een formele grammatica. Er bestaan instrumenten die gebruikt kunnen worden om een dergelijke grammatica te verbeteren, te verifiëren, aan te passen of te herstructureren. Ook zijn er instrumenten die softwareartefacten, zoals taaldocumenten, automatisch uit een formele grammatica kunnen generen.

Deze instrumenten worden vaak "grammarware" genoemd. Dit proefschrift heeft de doelstelling om 1) de hedendaagse grammarware te verbeteren, uit te breiden en te verdiepen, en om 2) nieuwe benaderingen en methodologieën te ontwikkelen.

In het proefschrift is herwinning van een grammatica gedaan met behulp van zogenaamde "grammaticale onttrekkers". Dit zijn speciale programma's die formele grammatica's kunnen afleiden van bestaande softwareartefacten met ingebouwde taalkennis (zoals taaldocumentatie of de broncode van de compiler). Een typische herwinning van een grammatica en een generiek stappenplan hiervoor zijn in hoofdstuk 3 te lezen. Daarnaast vindt de lezer een beschrijving van een aantal complexe onttrekkers in de andere hoofdstukken. Het automatiseren van het onttrekking proces is cruciaal in deze beschrijving.

Grammaticale convergentie, besproken in hoofdstuk 4, is een nieuwe techniek voor het afleiden van de relatie tussen twee of meer grammatica's met behulp van grammaticale transformaties. Deze methode is één van de weinige die op natuurlijke wijze in staat zijn een aantal grammatica's tegelijkertijd te behandelen. Grammaticale convergentie wordt voor het eerst in dit proefschrift en de verwante publicaties gepresenteerd. Hoofdstuk 5 bevat een gevalsstudie die laat zien dat de methodologie goed toepasbaar is op grote industriële projecten.

Het succes en de kracht van grammaticale convergentie wordt bepaald door de keuze van de set operatoren die men gebruikt om de grammatica's te transformeren. In dit proefschrift wordt onze operatorenset XBGF zorgvuldig en nauwkeurig uitgelegd. We tonen aan dat XBGF krachtiger en verder ontwikkeld is dan de bestaande sets operatoren.

In ons onderzoek hebben we veel gebruik gemaakt van taalstandaarden, onder andere als een bron voor de onttrekkers of voor het documenteren van onze domein-gerichte talen. Om efficient gebruik te kunnen maken van deze taaldocumentatie, is onderzocht hoe deze in elkaar zit. Hiervoor werden tientallen taalstandaarden geanalyseerd, met het LDF datamodel als resultaat. LDF is een taal waarin men niet alleen grammatica's, maar ook andere delen van een typisch taaldocument (zoals codevoorbeelden of tekst in een natuur-

lijke taal) kan beschrijven. Zo wordt het mogelijk om een document op semi-automatische wijze te verbeteren, te verifiëren, aan te passen of te herstructureren. Ook wordt het mogelijk semi-automatisch een PDF- of HTML-versie van een document te genereren.

De voornaamste contributies van dit proefschrift zijn de volgende:

- ◇ Het stappenplan voor herwinning van een grammatica en andere inzichten op dat gebied — zie [257] en Hoofdstuk 3.
- ◇ De lichtgewicht verificatietechniek genaamd “grammaticale convergentie” — zie [166, 167, 168, 258, 259] en Hoofdstukken 4–5.
- ◇ De ontwikkeling van de grammaticale onttrekkers, met name de regel-gebaseerde — zie [168] en Hoofdstuk 5.
- ◇ De 18 verschillende grammatica’s geproduceerd door deze onttrekkers — zie [260].
- ◇ De gedetailleerde analyse van meer dan 40 huidige taalstandaarden en taalhandboeken — zie [262] en Hoofdstuk 6.
- ◇ Het datamodel voor het taalspecificatiedomein — zie [262] en Hoofdstuk 6.
- ◇ Het opstellen en het prototyperen van de taaldocumentatie infrastructuur — zie [143, 258, 259] en Hoofdstuk 6.
- ◇ De 6 domein-specifieke talen voor grammarware en de door onze infrastructuur geproduceerde taaldocumenten voor hen — zie [258, 259, 261] en Hoofdstukken 6–7.
- ◇ De krachtige set operatoren voor grammaticale transformaties — zie [168, 261] en Hoofdstuk 7.

Met uitzondering van online documenten, zijn er in totaal acht publicaties op basis van dit proefschrift, waarvan één journal paper [168], één ISO document [143], twéé extended abstracts [257, 258] en vier proceedings papers [166, 167, 259, 262].

Bibliography

[1] AcuCorp Inc. *AcuCobol-85 Reference Manual*, 1999.

[2] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1985.

[3] J. Albert, D. Giammarresi, and D. Wood. Normal Form Algorithms for Extended Context-free Grammars. *Theoretical Computer Science*, 267(1–2):35–47, 2001.

[4] E. Allen. *Bug Patterns in Java*. APress L. P., 2002.

[5] T. L. Alves, P. F. Silva, J. Visser, and J. N. Oliveira. Strategic Term Rewriting and Its Application to a VDMSL to SQL Conversion. In *FM 2005: Formal Methods, International Symposium of Formal Methods Europe, Newcastle, UK, July 18-22, 2005, Proceedings*, volume 3582 of *LNCS*, pages 399–414. Springer, 2005.

[6] T. L. 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.

[7] American National Standards Institute. www.ansi.org. Accessed in June 2009.

[8] J. Axelsson, M. Birbeck, M. Dubinko, B. Epperson, M. Ishikawa, S. McCarron, A. Navarro, and S. Pemberton. XHTML™2.0. *W3C Working Draft*, 26 July 2006. www.w3.org/TR/2006/WD-xhtml2-20060726.

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

[10] J. W. Backus, F. L. Bauer, J. Green, C. Katz, J. McCarthy, P. Naur, A. J. Perlis, H. Rutishauser, K. Samelson, B. Vauquois, J. H. Wegstein, A. van Wijngaarden, and M. Woodger. Revised Report on the Algorithmic Language ALGOL 60. *Numerische Mathematik*, 4:420–453, Springer-Verlag, Berlin, Heidelberg, New York, 1963. International Federation for Information Processing 1962. Edited by Peter Naur.

[11] J. W. Backus, R. J. Beeber, S. Best, R. Goldberg, H. L. Herrick, R. A. Hughes, L. B. Mitchell, R. A. Nelson, R. Nutt, D. Sayre, P. B. Sheridan, H. Stern, and I. Ziller. *The Fortran Automatic Coding System for the IBM 704 EDPM. Programmer's Reference Manual*. Applied Science Division and Programming Research Department, International Business Machines Corporation, 590 Madison Ave., New York 22, NY, October 15 1956.

[12] D. T. Barnard. Syntax Error Handling Techniques. Technical Report Technical Report 81-125, Queen's University, Department of Computing and Information Science, September 1981. 23 pages.

[13] D. T. Barnard and R. C. Holt. Hierarchic Syntax Error Repair for LR Grammars. *International Journal of Computer and Information Sciences*, 11(4):231–258, 1982.

[14] P. Berdaguer, A. Cunha, H. Pacheco, and J. Visser. Coupled Schema Transformation and Data Conversion for XML and SQL. In *Practical Aspects of Declarative Languages, 9th International Symposium, PADL 2007, Nice, France, January 14-15, 2007*, volume 4354 of *LNCS*, pages 290–304. Springer, 2007.

[15] A. Berglund, S. Boag, D. Chamberlin, M. Fernández, M. Kay, J. Robie, and J. Siméon. XML Path Language (XPath) 2.0. *W3C Recommendation*, 23 January 2007. www.w3.org/TR/2007/REC-xpath20-20070123.

[16] J. A. Bergstra, J. Heering, and P. Klint. The Algebraic Specification Formalism ASF. In J. Heering and P. Klint, editors, *Algebraic Specification*, ACM Press Frontier Series, pages 1–66. ACM Press, Addison-Wesley, 1989.

[17] D. Blasband. Parsing in a Hostile World. In R. Koschke, E. Burd, and P. Aiken, editors, *Proceedings of the 8th Working Conference on Reverse Engineering (WCRE'01)*, pages 291–300. IEEE Computer Society Press, 2001.

[18] G. Booch. *Object-Oriented Analysis and Design with Applications*. Addison-Wesley, 1994.

[19] B. Bos, T. Çelik, I. Hickson, and H. W. Lie. Cascading Style Sheets, Level 2 Revision 1. CSS 2.1 Specification. *W3C Working Draft*, 6 November 2006. www.w3.org/TR/2006/WD-CSS21-20061106.

[20] J. Bosak, T. Bray, D. Connolly, E. Maler, G. Nicol, C. Michael Sperberg-McQueen, L. Wood, and J. Clark. W3C XML Specification DTD ("XMLspec"). ArborText Inc., 1998. Available at www.w3.org/XML/1998/06/xmlspec-report-19980910.htm.

[21] E. Bouwers, M. Bravenboer, and E. Visser. Grammar Engineering Support for Precedence Rule Recovery and Compatibility Checking. *ENTCS*, 203(2):85–101, 2008.

[22] M. G. J. van den Brand, A. van Deursen, J. Heering, H. A. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P. A. Olivier, J. Scheerder, J. J. Vinju, E. Visser, and J. Visser. The ASF+SDF Meta-Environment: a Component-Based Language Development Environment. In *Proceedings of the 10th International Conference on Compiler Construction (CC'01)*, volume 2027 of *LNCS*, pages 365–370, London, UK, 2001. Springer-Verlag.

[23] M. G. J. van den Brand, J. Heering, P. Klint, and P. A. Olivier. Compiling Language Definitions: the ASF+SDF Compiler. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 24(4):334–368, 2002.

[24] M. G. J. van den Brand, A. T. Kooiker, N. P. Veerman, and J. J. Vinju. An Architecture for Context-sensitive Formatting. In *Proceedings of the International Conference on Software Maintenance (ICSM'05)*, 2005.

[25] M. G. J. van den Brand, P. E. Moreau, and J. J. Vinju. Environments for Term Rewriting Engines for Free! In *Proceedings of the 14th International Conference on Rewriting Techniques and Applications (RTA 2003)*, volume 2706 of *LNCS*, pages 424–435. Springer-Verlag, 2003.

[26] M. G. J. van den Brand, J. Scheerder, J. J. Vinju, and E. Visser. Disambiguation Filters for Scannerless Generalized LR Parsers. In N. Horspool, editor, *Compiler Construction 2002 (CC 2002)*, pages 143–158, 2002.

[27] M. G. J. van den Brand, A. Sellink, and C. Verhoef. Current Parsing Techniques in Software Renovation Considered Harmful. In *International Workshop on Program Comprehension*, 1998.

[28] M. G. J. van den Brand, M. P. A. Sellink, and C. Verhoef. Obtaining a COBOL Grammar from Legacy Code for Reengineering Purposes. In M. P. A. Sellink, editor, *Proceedings of the 2nd International Workshop on the Theory and Practice of Algebraic Specifications*, Berlin, 1997. Springer-Verlag.

[29] M. G. J. van den Brand and E. Visser. Generation of Formatters for Context-Free Languages. *ACM Transactions on Software Engineering Methodology (TOSEM)*, 5(1):1–41, 1996.

[30] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau. Extensible Markup Language (XML) 1.0 (Fourth Edition). *W3C Recommendation*, 16 August 2006, edited in place 29 September 2006. www.w3.org/TR/2006/REC-xml-20060816.

[31] W. H. Burkhardt. Metalanguage and Syntax Specification. *Communications of the ACM*, 8(5):304–305, 1965.

[32] N. Chomsky. Three Models for the Description of Language. *IRE Transactions on Information Theory*, 2(2):113–123, 1956.

[33] A. Cicchetti, D. Di Ruscio, R. Eramo, and A. Pierantonio. Automating Co-evolution in Model-Driven Engineering. In *12th International IEEE Enterprise Distributed Object Computing Conference (EDOC 2008)*, pages 222–231. IEEE Computer Society, 2008.

[34] J. Clark and M. Murata. RELAX NG Specification. *OASIS Committee Specification*, 3 December 2001. Available at relaxng.org/spec-20011203.html, also being standardised by ISO as multiple parts of ISO/IEC 19757.

[35] A. Cleve and J.-L. Hainaut. Co-transformations in Database Applications Evolution. In *Generative and Transformational Techniques in Software Engineering, International Summer School (GTTSE'05), Braga, Portugal, July 4-8, 2005. Revised Papers*, volume 4143 of *LNCS*, pages 409–421. Springer, 2006.

[36] M. Cohen, J. Reid, et al. *ISO/IEC 1539-1: Information Technology—Programming Languages—Fortran, WD 1539–1 J3/08–007, an Internal Working Document of J3*, 2008.

[37] COMPAQ Information Technology Group. *COBOL Reference Manual Version 2.5*, 2002.

[38] J. R. Cordy. Generalized Selective XML Markup of Source Code Using Agile Parsing. In *Proceedings of the 11th IEEE International Workshop on Program Comprehension (IWPC)*, pages 144–153, Portland, Oregon, May 2003.

[39] J. R. Cordy. The TXL Source Transformation Language. *Science of Computer Programming*, 61(3):190–210, 2006.

[40] J. Cunha, J. Saraiva, and J. Visser. From Spreadsheets to Relational Databases and Back. In *PEPM '09: Proceedings of the 2009 ACM SIGPLAN workshop on Partial evaluation and program manipulation*, pages 179–188, New York, NY, USA, 2008. ACM.

[41] T. Dean and M. Synytskyy. Agile Parsing Techniques for Web Applications. In *Proceedings of the International Summer School on Generative and Transformational Techniques in Software Engineering, Part II, Technology Presentations*, pages 29–38, Braga, Portugal, July 2005.

[42] T. R. Dean, J. R. Cordy, A. J. Malton, and K. A. Schneider. Grammar Programming in TXL. In *Proceedings, Source Code Analysis and Manipulation (SCAM'02)*. IEEE, 2002.

[43] T. R. Dean, J. R. Cordy, A. J. Malton, and K. A. Schneider. Agile Parsing in TXL. *Journal of Automated Software Engineering*, 10(4):311–336, 2003.

[44] F. L. Deremer. Practical Translators for LR(k) Languages. Technical report, Massachusetts Institute of Technology, Cambridge, MA, USA, 1969.

[45] A. van Deursen, J. Heering, and P. Klint, editors. *Language Prototyping: An Algebraic Specification Approach: Vol. V*. World Scientific Publishing Co., Inc., River Edge, NJ, USA, 1996.

[46] Digital Equipment Corporation. *COBOL Reference Manual Version 2.3*, 2002.

[47] E. W. Dijkstra. EWD 1009: On a Somewhat Disappointing Correspondence. Available at www.cs.utexas.edu/users/EWD/transcriptions/EWD10xx/EWD1009.html.

[48] E. W. Dijkstra. Go To Statement Considered Harmful. *Communications of the ACM*, 11:147–148, 1968. Available at www.acm.org/classics/oct95.

[49] H. H. Do and E. Rahm. Matching Large Schemas: Approaches and Evaluation. *Information Systems*, 32(6):857–885, 2007.

[50] J. J. Donovan and H. F. Ledgard. Formal Language Description Languages for Computer Programming. In T. B. Steel, Jr., editor, *IFIP Working Conference on Formal Language Description Languages*, Amsterdam, 1966. North-Holland Publishing Company.

[51] J. J. Donovan and H. F. Ledgard. A Formal System for the Specification of the Syntax and Translation of Computer Languages. In *AFIPS '67 (Fall): Proceedings of the November 14–16, 1967, Fall Joint Computer Conference*, pages 553–569, New York, NY, USA, 1967. ACM.

[52] M. Dowson. The Ariane 5 Software Failure. *ACM SIGSOFT Software Engineering Notes*, 22(2):84, 1997.

[53] 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.

[54] A. Dubey, S. K. Aggarwal, and P. Jalote. A Technique for Extracting Keyword Based Rules from a Set of Programs. In *9th European Conference on Software Maintenance and Reengineering (CSMR 2005), Proceedings*, pages 217–225. IEEE, 2005.

[55] A. Dubey, P. Jalote, and S. K. Aggarwal. A Deterministic Technique for Extracting Keyword Based Grammar Rules from Programs. In *SAC '06: Proceedings of the 2006 ACM symposium on Applied computing*, pages 1631–1632. ACM, 2006.

[56] A. Dubey, P. Jalote, and S. K. Aggarwal. Inferring Grammar Rules of Programming Language Dialects. In *Grammatical Inference: Algorithms and Applications, 8th International Colloquium, ICGI 2006, Tokyo, Japan, September 20-22, 2006, Proceedings*, volume 4201 of *Lecture Notes in Computer Science*, pages 201–213. Springer, 2006.

[57] A. Dubey, P. Jalote, and S. K. Aggarwal. Learning Context-Free Grammar Rules from a Set of Program. *IET Software*, 2(3):223–240, 2008.

[58] 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 Computer Society, 2007.

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

[60] Ecma International — Standards @ Internet Speed. www.ecma-international.org. Accessed in June 2009.

[61] C. Exton, D. Watkins, and D. Thompson. Comparisons between CORBA IDL & COM/DCOM MIDL: Interfaces for Distributed Computing. In *Proceedings, TOOLS 1997: 25th International Conference on Technology of Object-Oriented Languages and Systems*, pages 15–32. IEEE Computer Society, 1997.

[62] 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.

[63] J.-M. Favre. Preprocessors from an Abstract Point of View. In *Proceedings of the International Conference on Software Maintenance (ICSM'96)*, pages 329–338, Washington, DC, USA, 1996. IEEE Computer Society Press.

[64] J.-M. Favre and T. NGuyen. Towards a Megamodel to Model Software Evolution through Transformations. *Electronic Notes in Theoretical Computer Science, Proceedings of the SETra Workshop*, 127(3), 2004.

[65] L. M. G. Feijs, R. L. Krikhaar, and R. C. van Ommering. A Relational Approach to Support Software Architecture Analysis. *Software—Practice and Experience*, 28:371–400, April 1998.

[66] R. W. Floyd. A Note on Mathematical Induction on Phrase Structure Grammars. *Information and Control*, 4:353–358, 1961.

[67] R. W. Floyd. The Syntax of Programming Languages—A Survey. *IEEE Transactions on Electronic Computers*, EC-13(4):345, August 1964.

[68] B. Ford. Packrat Parsing: Simple, Powerful, Lazy, Linear Time, Functional Pearl. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming (ICFP'02)*, pages 36–47, New York, NY, USA, 2002. ACM Press.

[69] B. Ford. Parsing Expression Grammars: a Recognition-Based Syntactic Foundation. In *Proceedings of the Symposium on Principles of Programming Languages*, January 2004.

[70] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design or Existing Code*. Addison-Wesley Professional, 1999.

[71] D. J. Frailey. An Intermediate Language for Source and Target Independent Code Optimization. In *SIGPLAN '79: Proceedings of the 1979 SIGPLAN symposium on Compiler construction*, pages 188–200, New York, NY, USA, 1979. ACM Press.

[72] Fujitsu Limited. *NetCobol Language Reference*, ninth edition, 2005. Available at www.adtools.com/download/v8manuals/NetCOBLanguageRef.pdf.

[73] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[74] E. Gansner, E. Koutsofios, and S. North. *Drawing Graphs with dot*, 26 January 2006. Available at http://www.graphviz.org/pdf/dotguide.pdf.

[75] S. Gao, C. M. Sperberg-McQueen, H. S. Thompson, N. Mendelsohn, D. Beech, and M. Maloney. W3C XML Schema Definition Language (XSD) 1.1 Part 1: Structures. *W3C Candidate Recommendation*, 30 April 2009. www.w3.org/TR/2009/CR-xmlschema11-1-20090430.

[76] S. Gorn. Specification Languages for Mechanical Languages and Their Processors—A Baker's Dozen. *Communications of the ACM*, 4(12):532–542, 1961.

[77] J. Gosling, B. Joy, and G. L. Steele. *The Java Language Specification*. Addison-Wesley, 1996. Available at java.sun.com/docs/books/jls.

[78] 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.

[79] 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.

[80] P. Graham. *On Lisp*. Prentice Hall, 1993. Available at www.paulgraham.com/onlisptext.html.

[81] P. Graham. *ANSI Common Lisp*. Prentice Hall, 1995.

[82] R. Grimm. Practical Packrat Parsing. Technical Report TR2004-854, New York University, Department of Computer Science, March 2004. Available at cs.nyu.edu/rgrimm/papers/tr2004-854.pdf.

[83] D. Grune and C. J. H. Jacobs. *Parsing Techniques: A Practical Guide*. Monographs in Computer Science. Springer, 2 edition, 2008. http://www.cs.vu.nl/~dick/PT2Ed.html.

[84] I. Guilfanov. A Simple Type System for Program Reengineering. In *Proceedings of the Eighth Working Conference on Reverse Engineering*, page 357. IEEE Computer Society Press, 2001. Software available at www.datarescue.com/idabase.

[85] J.-L. Hainaut, C. Tonneau, M. Joris, and M. Chandelon. Schema Transformation Techniques for Database Reverse Engineering. In *Entity-Relationship Approach - ER'93, 12th International Conference on the Entity-Relationship Approach, Arlington, Texas, USA, December 15-17, 1993, Proceedings*, volume 823 of *LNCS*, pages 364–375. Springer, 1994.

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

[87] C. A. R. Hoare. Proof of Correctness of Data Representations. *Acta Informatica*, 1(4):271–281, 1972.

[88] W. E. Howden. Contemporary Software Development Environments. *Communications of the ACM*, 25(5):318–329, 1982.

[89] IBM. *Informix 4GL Reference Manual*, 7.32 edition, March 2003.

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

[91] IBM SA22–7597–10. *MVC JCL Reference*, eleventh edition, April 2006. publibz.boulder.ibm.com/epubs/pdf/iea2b661.pdf.

[92] International Electrotechnical Commission — International Standards and Conformity Assessment. www.iec.ch. Accessed in June 2009.

[93] Institute of Electrical and Electronics Engineers Standards Association. standards.ieee.org. Accessed in June 2009.

[94] IEEE. Standards Status. Technical report, IEEE-SA, May 2009. Available at standards.ieee.org/db/status/index.shtml.

[95] IEEE Std 1178–1990. *IEEE Standard for the Scheme Programming Language*, Approved December 10, 1990. Reaffirmed December 12, 1995.

[96] The Internet Engineering Task Force. www.ietf.org. Accessed in June 2009.

[97] International Organization for Standardization — Standards Development. www.iso.org/iso/standards_development.htm. Accessed in September 2010, updated in September 2010.

[98] ISO 10303-11:2004. *Industrial Automation Systems and Integration—Product Data Representation and Exchange—Part 11: Description Methods: The EXPRESS Language Reference Manual*, 2004.

[99] ISO 10303-21:2002. *Industrial Automation Systems and Integration—Product Data Representation and Exchange—Part 21: Implementation Methods: Clear Text Encoding of the Exchange Structure*, 2002. Also known as STEP—Standard for the Exchange of Product Model Data.

[100] ISO/IEC JTC1/SC22 — International Standardization Subcommittee for Programming Languages, Their Environments and System Software Interfaces. www.open-std.org/jtc1/sc22. Accessed in September 2010, updated in September 2010.

[101] ISO/IEC JTC1/SC22/WG4 — COBOL Standardization Working Group. www.cobolstandard.info/wg4/wg4.html. Accessed in September 2010, updated in June 2010.

[102] ISO/IEC JTC1/SC22/WG5 — Fortran Standards Working Group. www.nag.co.uk/sc22wg5. Accessed in September 2010, updated in June 2010.

[103] ISO/IEC JTC1/SC22/WG9 — Ada Standards Working Group. www.open-std.org/JTC1/SC22/WG9. Accessed in September 2010, updated in September 2010.

[104] ISO/IEC JTC1/SC22/WG14 — International Standardization Working Group for the Programming Language C. www.open-std.org/JTC1/SC22/WG14. Accessed in September 2010, updated in June 2010.

[105] ISO/IEC JTC1/SC22/WG16 — International Standardization Working Group for the Programming Language ISLISP. www.open-std.org/JTC1/SC22/WG16. Accessed in September 2010, updated in October 2007.

[106] ISO/IEC JTC1/SC22/WG17 — International Standardization Working Group for the Programming Language Prolog. www.sju.edu/~jhodgson/wg17/wg17web.html. Accessed in September 2010, updated in December 2005.

[107] ISO/IEC JTC1/SC22/WG19 — Formal Specification Languages Working Group. www.open-std.org/JTC1/SC22/WG19. Accessed in September 2010, updated in March 2001.

[108] ISO/IEC JTC1/SC22/WG21 — The C++ Standards Committee. www.open-std.org/JTC1/SC22/WG21. Accessed in September 2010, updated in August 2010.

[109] ISO/IEC 10279:1991(E). *Information Technology—Programming Languages—Full BASIC*, 1991.

[110] ISO/IEC 14882. *Information Technology—Programming Languages—C++, Committee Draft WG21/N2315*, 2007. Available at http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2007/n2315.pdf. Accessed in September 2010.

[111] ISO/IEC 14882:1998(E). *Information Technology—Programming Languages—C++, First Edition*, 1998. Available at http://www-d0.fnal.gov/~dladams/cxx_standard.pdf. Accessed in September 2010.

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

[113] ISO/IEC 1989:2002. *Information Technology—Programming Languages—COBOL*, 2002.

[114] ISO/IEC 23270:2003. *Information Technology—C# Language Specification*, 2003.

[115] ISO/IEC 23270:2006(E). *Information Technology—C# Language Specification*, 2006. Available at standards.iso.org/ittf/PubliclyAvailableStandards/index.html.

[116] ISO/IEC 9126. *Software Engineering—Product Quality*, 1991.

[117] ISO/IEC 9899:1999(E). *Information Technology—Programming Languages—C*, 1999.

[118] ISO/IEC 9899:TC2. *Information Technology—Programming Languages—C, Committee Draft WG14/N1124*, 2005.

[119] ISO/IEC 9899:TC3. *Information Technology—Programming Languages—C, Committee Draft WG14/N1256*, 2007.

[120] ISO/IEC SC22/WG21 N2723=08-0233 Working Draft. *Standard for Programming Language C++*, 2008. Available at http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2723.pdf. Accessed in September 2010.

[121] Information Technology Laboratory of National Institute of Standards and Technology. www.itl.nist.gov. Accessed in May 2007.

[122] International Telecommunication Union. www.itu.int. Accessed in June 2009.

[123] K. E. Iverson. A Method of Syntax Specification. *Communications of the ACM*, 7(10):588–589, 1964.

[124] I. Jacobs. World Wide Web Consortium Process Document. *W3C*, 14 October 2005. www.w3.org/Consortium/Process.

[125] J. Jagger. Hyperlinked ECMA-334 C# Grammar. Available at www.jaggersoft.com/csharp_grammar.html, 2003. Accessed in September 2010.

[126] JCP JSR 31. JAXB 2.0/2.1 — Java Architecture for XML Binding, 2008. http://jaxb.dev.java.net/.

[127] D. Jin, J. R. Cordy, and T. R. Dean. Where's the Schema? A Taxonomy of Patterns for Software Exchange. In *Proceedings of the 10th International Workshop on Program Comprehension (IWPC'02)*, pages 65–74, 2002.

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

[129] C. Jones. *Estimating Software Costs*. McGraw-Hill, New York, 1998.

[130] C. Jones. *The Year 2000 Software Problem: Quantifying the Costs and Assessing the Consequences*. Addison-Wesley, Reading, Massachusets, 1998.

[131] S. P. Jones, editor. *Haskell 98 Language and Libraries. The Revised Report*. Cambridge University Press, 2003.

[132] 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.

[133] D. Jonsson. Next: the Elimination of Goto-Patches? *ACM SIGPLAN Notices*, 24(3):85–92, 1989.

[134] 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.

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

[136] B. M. Kadhim and W. M. Waite. Maptool—Supporting Modular Syntax Development. In T. Gyimothy, editor, *Proceedings of the 6th International Conference on Compiler Construction (CC'96)*, volume 1060 of *LNCS*, pages 268–280. Springer, 1996.

[137] M. Kay. XSL Transformations (XSLT) Version 2.0. *W3C Recommendation*, 23 January 2007. www.w3.org/TR/2007/REC-xslt20-20070123.

[138] J. Kerievsky. *Refactoring to Patterns*. Addison-Wesley Professional, 2005.

[139] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Bell Laboratories, Murray Hill, New Jersey, 1978.

[140] P. Klint. A Meta-Environment for Generating Programming Environments. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 2(2):176–201, 1993.

[141] 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.

[142] S. Klusener and R. Lämmel. Deriving Tolerant Grammars from a Base-line Grammar. In *Proceedings of the International Conference on Software Maintenance (ICSM'03)*, pages 179–188, Los Alamitos, CA, USA, September 2003. IEEE Computer Society.

[143] S. Klusener and V. Zaytsev. ISO/IEC JTC1/SC22 Document N3977—Language Standardization Needs Grammarware. Available at http://www.open-std.org/jtc1/sc22/open/n3977.pdf, 2005.

[144] D. E. Knuth. Backus Normal Form vs. Backus Naur Form. *Communications of the ACM*, 7(12):735–736, 1964.

[145] D. E. Knuth. On the Translation of Languages from Left to Right. *Information and Control*, 8:607–639, 1965.

[146] D. E. Knuth. Structured Programming with Go To Statements. *ACM Computing Surveys*, 6(4):261–301, 1974.

[147] D. E. Knuth. Literate Programming. *The Computer Journal*, 27(2):97–111, 1984.

[148] R. Koppler. A Systematic Approach to Fuzzy Parsing. *Software Practice and Experience*, 27(6):637–649, 1997.

[149] J. Kort, R. Lämmel, and C. Verhoef. The Grammar Deployment Kit. In M. G. J. van den Brand and R. Lämmel, editors, *Electronic Notes in Theoretical Computer Science*, volume 65. Elsevier Science Publishers, 2002.

[150] N. A. Kraft, E. B. Duffy, and B. A. Malloy. Grammar Recovery from Parse Trees and Metrics-Guided Grammar Refactoring. *IEEE Trans. Software Eng.*, 35(6):780–794, 2009.

[151] N. A. Kraft, B. A. Malloy, and J. F. Power. An Infrastructure to Support Interoperability in Reverse Engineering. *Information & Software Technology*, 49(3):292–307, 2007.

[152] S.-Y. Kuroda. Classes of Languages and Linear-bounded Automata. *Information and Control*, 7(2):207–223, June 1964.

[153] I. Kurtev, J. Bézivin, F. Jouault, and P. Valduriez. Model-based DSL Frameworks. In *OOPSLA '06: Companion to the 21st ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages and Applications*, pages 602–616, New York, NY, USA, 2006. ACM Press.

[154] 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-Verlag, 2001.

[155] R. Lämmel. Grammar Testing. In H. Hussmann, editor, *Proceedings, Fundamental Approaches to Software Engineering (FASE'01)*, volume 2029 of *LNCS*, pages 201–216. Springer, 2001.

[156] R. Lämmel. Towards Generic Refactoring. In *Proceedings of Third ACM SIGPLAN Workshop on Rule-Based Programming (RULE'02)*, pages 15–28, Pittsburgh, Pennsylvania, USA, October5 2002. ACM Press.

[157] R. Lämmel. Coupled Software Transformations (Extended Abstract). In *First International Workshop on Software Evolution Transformations*, November 2004.

[158] R. Lämmel. The Amsterdam Toolkit for Language Archaeology. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 137(3):43–55, 8 September 2005. Proceedings of the Second International Workshop on Metamodels, Schemas and Grammars for Reverse Engineering (ATEM'04).

[159] R. Lämmel and W. Lohmann. Format Evolution. In *Proceedings of the 7th International Conference on Reverse Engineering for Information Systems (RETIS 2001)*, volume 155 of *books@ocg.at*, pages 113–134. OCG, 2001.

[160] R. Lämmel and E. Meijer. Mappings Make Data Processing Go 'Round. In R. Lämmel, J. Saraiva, and J. Visser, editors, *Generative and Transformational Techniques in Software Engineering, International Summer School (GTTSE'05), Braga, Portugal, July 4-8, 2005. Revised Papers*, volume 4143 of *LNCS*, pages 169–218. Springer, 2006.

[161] R. Lämmel and E. Meijer. Revealing the X/O Impedance Mismatch. In *Datatype-Generic Programming, International Spring School, SSDGP 2006, Nottingham, UK, Revised Lectures*, volume 4719 of *LNCS*, pages 285–368. Springer, 2006.

[162] R. Lämmel and W. Schulte. Controllable Combinatorial Coverage in Grammar-Based Testing. In *Proceedings of the 18th IFIP TC6/WG6.1 International Conference on Testing of Communicating Systems (TestCom'06)*, volume 3964 of *LNCS*, pages 19–38. Springer, 2006.

[163] R. Lämmel and C. Verhoef. Cracking the 500-Language Problem. *IEEE Software*, pages 78–88, November/December 2001.

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

[165] 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.

[166] R. Lämmel and V. Zaytsev. An Introduction to Grammar Convergence. In *Proceedings of 7th International Conference on Integrated Formal Methods (iFM'09)*, volume 5423 of *LNCS*, pages 246–260. Springer, 2009.

[167] R. Lämmel and V. Zaytsev. Recovering Grammar Relationships for the Java Language Specification. In *Ninth IEEE International Working Conference on Source Code Analysis and Manipulation*, pages 178–186. IEEE, September 2009.

[168] R. Lämmel and V. Zaytsev. Recovering Grammar Relationships for the Java Language Specification. *Software Quality Journal*, SCAM Special Issue, August 2010. In print.

[169] B. Lang. Deterministic Techniques for Efficient Non-deterministic Parsers. In J. Loeckx, editor, *Proceedings of the Second Colloquium on Automata, Languages and Programming, volume 14 of Lecture Notes in Computer Science*, pages 255–269. Springer-Verlag, 1974.

[170] M. Lange and H. Leiß. To CNF or not to CNF? *Informatica Didactica*, 8, 2009.

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

[172] W. Lohmann, G. Riedewald, and M. Stoy. Semantics-preserving Migration of Semantic Rules after Left Recursion Removal in Attribute Grammars. In *Proceedings of 4th Workshop on Language Descriptions, Tools and Applications (LDTA 2004)*, volume 110 of *ENTCS*, pages 133–148. Elsevier Science, 2004.

[173] F. Lundh. ElementTree XML Framework for Python. http://effbot.org/zone/element-index.htm.

[174] B. A. Malloy, J. F. Power, and J. T. Waldron. Applying Software Engineering Techniques to Parser Design: the Development of a C# Parser. In *Proceedings of the Conference of the South African Institute of Computer Scientists and Information Technologists*, pages 75–82, 2002. In cooperation with ACM Press.

[175] P. B. Mann. A Translational BNF Grammar Notation (TBNF). *ACM SIGPLAN Notices*, 41(4):16–23, 2006.

[176] M. Marcotty, H. Ledgard, and G. V. Bochmann. A Sampler of Formal Definitions. *ACM Computing Surveys (CSUR)*, 8(2):191–276, 1976.

[177] M. D. McIlroy. 'Mass Produced' Software Components. In *Software Engineering, report on a conference sponsored by the NATO Science Committee*, pages 138–155, Garmisch, Germany, October 1968. Available at homepages.cs.ncl.ac.uk/brian.randell/NATO/nato1968.PDF.

[178] W. McKeeman. Differential Testing for Software. *Digital Technical Journal of Digital Equipment Corporation*, 10(1):100–107, 1998.

[179] B. McLaughlin. *Java and XML Data Binding*. Nutshell handbook. O'Reilly & Associates, Inc., 2002.

[180] B. L. Meek. The Programming Language Standards Scene, Ten Years on. *Computer Standards & Interfaces*, 16(5–6):425–426, 1994.

[181] M. Mernik, G. Gerlic, V. Zumer, and B. R. Bryant. Can a Parser be Generated from Examples? In *Proceedings of the 2003 ACM Symposium on Applied Computing (SAC), March 9-12, 2003, Melbourne, FL, USA*, pages 1063–1067. ACM, 2003.

[182] Micro Focus Limited. *Micro Focus COBOL for UNIX Pocket Guide, Issue 5*, 1994.

[183] Micro Focus Limited. *Object COBOL Language Reference, Issue 16*, June 1998.

[184] Microsoft .NET Framework Developer Center. *ECMA and ISO/IEC C# and Common Language Infrastructure Standards*, 2003. Available at msdn.microsoft.com/netframework/ecma and mirror sites.

[185] MIL–STD–1589C. *Military Standard Jovial (J73)*, July 1984.

[186] L. Moonen. Generating Robust Parsers using Island Grammars. In *WCRE'01: Proceedings of the 8th Working Conference on Reverse Engineering*, pages 13–22. IEEE Computer Society Press, October 2001.

[187] L. Moonen. Lightweight Impact Analysis using Island Grammars. In *IWPC'02: Proceedings of the 10th International Workshop on Program Comprehension*. IEEE Computer Society Press, June 2002.

[188] D. Moore, C. Musciano, M. J. Liebhaber, S. F. Lott, and L. Starr. ""Go To Considered Harmful" Considered Harmful" Considered Harmful. *Communications of the ACM*, 30(5):351–355, 1987.

[189] Robert C. Moore. Removing Left Recursion from Context-Free Grammars. In *Proceedings of the 1st Conference on North American Chapter of the Association for Computational Linguistics*, pages 249–255. Morgan Kaufmann Publishers Inc., 2000.

[190] Carroll Morgan. *Programming from Specifications*. Prentice Hall International, 1990.

[191] M. J. Nederhof, C. H. A. Koster, C. Dekkers, and A. van Zwol. The Grammar Workbench: a First Step Towards Lingware Engineering. In W. ter Stal, A. Nijholt, and H. J. op den Akker, editors, *Proceedings of the Second Twente Workshop on Language Technology — Linguistic Engineering: Tools and Products,* volume 92-29 of *Memoranda Informatica*, pages 103–115, 1992.

[192] O. Nierstrasz, M. Kobel, T. Gîrba, M. Lanza, and H. Bunke. Example-Driven Reconstruction of Software Models. In R. Krikhaar, C. Verhoef, and G. Di Lucca, editors, *Proceedings of the 11th European Conference on Software Maintenance and Reengineering (CSMR'07)*, pages 275–284. IEEE Computer Society Conference Publishing Services, 2007.

[193] T. Nipkow and D. von Oheimb. $Java_{light}$ 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.

[194] OASIS: Organization for the Advancement of Structured Information Standards. www.oasis-open.org. Accessed in May 2007.

[195] OASIS. DITA Version 1.1 Language Specification Approved as an OASIS Standard. *Committee Specification 01*, 31 May 2007. docs.oasis-open.org/dita/v1.1/CS01/langspec.

[196] Object Management Group. *MOF 2.0/XMI Mapping*, 2.1.1 edition, December 2007. Available at http://www.omg.org/spec/XMI/2.1.1.

[197] Object Management Group. *Meta-Object Facility (MOF™) Core Specification*, 2.0 edition, January 2006. Available at http://www.omg.org/spec/MOF/2.0.

[198] M. Odersky. *The Scala Language Specification*, November 2008. Available at http://www.scala-lang.org/sites/default/files/linuxsoft_archives/docu/files/ScalaReference.pdf. Accessed in December 2008.

[199] J. N. Oliveira. Transforming Data By Calculation. In *Generative and Transformational Techniques in Software Engineering II, International Summer School (GTTSE'07), Braga, Portugal, July 2007, Revised Papers*, volume 5235 of *LNCS*, pages 134–195. Springer, 2008.

[200] W. F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[201] P. Overell. *RFC 4234. Augmented BNF for Syntax Specifications: ABNF*, October 2005. Available at tools.ietf.org/html/rfc4234. Accessed in June 2007.

[202] T. Parr. ANTLR—ANother Tool for Language Recognition, 2008. http://antlr.org.

[203] T. Parr. The Reuse of Grammars with Embedded Semantic Actions. In *The 16th IEEE International Conference on Program Comprehension (ICPC'08)*, pages 5–10. IEEE Computer Society, 2008.

[204] T. J. Pennello and F. DeRemer. Efficient Computation of LALR(1) Look-ahead Sets. *ACM SIGPLAN Notices*, 39(4):14–27, 2004.

[205] M. Di Penta, P. Lombardi, K. Taneja, and L. Troiano. Search-based inference of dialect grammars. *Soft Computing - A Fusion of Foundations, Methodologies and Applications*, 12(1):51–66, 2008.

[206] M. Di Penta and K. Taneja. Towards the automatic evolution of reengineering tools. In *Proceedings of the 9th European Conference on Software Maintenance and Reengineering (CSMR '05),*, pages 241–244. IEEE Computer Society, 2005.

[207] P. Pepper. LR Parsing = Grammar Transformation + LL Parsing. Technical Report CS-99-05, TU Berlin, 1999.

[208] D. Peterson, S. Gao, A. Malhotra, C. M. Sperberg-McQueen, H. S. Thompson, and P. V. Biron. W3C XML Schema Definition Language (XSD) 1.1 Part 2: Datatypes. *W3C Candidate Recommendation*, 30 April 2009. www.w3.org/TR/2009/CR-xmlschema11-2-20090430.

[209] C. Pressey. *Funge-98 Final Specification*, September 1998. Available at quadium.net/funge/spec98.html. Accessed in September 2010, updated in September 1998.

[210] G. K. Pullum. Learnability. In W. Frawley, editor, *Mathematical Linguistics, International Encyclopedia of Linguistics, Second Edition*. Department of Linguistics, University of Delaware, 2000.

[211] D. Raggett. HTML 3.2 Reference Specification. *W3C Recommendation*, 14 January 1997. www.w3.org/TR/REC-html32.

[212] E. Rahm and P. A. Bernstein. A Survey of Approaches to Automatic Schema Matching. *VLDB Journal*, 10(4):334–350, 2001.

[213] J. Rekers. *Parser Generation for Interactive Environments*. PhD thesis, University of Amsterdam, 1992. Available at http://homepages.cwi.nl/~paulk/dissertations/Rekers.pdf.

[214] T. Röetschke and R. Krikhaar. Architecture Analysis Tools to Support Evaluation of Large Industrial Systems. In *Proceedings of IEEE International Conference on Software Maintenance (ICSM'02)*, pages 182–191, Montreal, Canada, October 2002.

[215] G. van Rossum and F. L. Drake, Jr. *The Python Language Reference Manual.* Network Theory Ltd., 2003.

[216] F. Rubin. "Go To Considered Harmful" Considered Harmful. *Communications of the ACM*, 30(3):195–196, 1986.

[217] A. Sellink and C. Verhoef. Generation of Software Renovation Factories from Compilers. In *Proceedings of 15th International Conference on Software Maintenance (ICSM'99)*, pages 245–255, 1999. Available at www.cs.vu.nl/~x/com.

[218] M. P. A. Sellink, H. M. Sneed, and C. Verhoef. Restructuring of COBOL/CICS Legacy Systems. In P. Nesi and C. Verhoef, editors, *Proceedings of the Third European Conference on Maintenance and Reengineering (CSMR'99)*, pages 72–82. IEEE Computer Society Press, 1999. Available at www.cs.vu.nl/~x/cics.

[219] M. P. A. Sellink and C. Verhoef. Development, Assessment, and Reengineering of Language Descriptions. In J. Ebert and C. Verhoef, editors, *Proceedings of the Fourth European Conference on Software Maintenance and Reengineering (CSMR'00)*, pages 151–160. IEEE Computer Society Press, March 2000. Available at www.cs.vu.nl/~x/cale.

[220] Y.-P. Shan et al. *NCITS J20 DRAFT of ANSI Smalltalk Standard, Revision 1.9*, December 1997. Available at wiki.squeak.org/squeak/uploads/172/standard_v1_9-indexed.pdf. Accessed in June 2007.

[221] P. Shvaiko and J. Euzenat. Ten Challenges for Ontology Matching. In *OTM '08: Proceedings of the OTM 2008 Confederated International Conferences, CoopIS, DOA, GADA, IS, and ODBASE 2008. Part II on On the Move to Meaningful Internet Systems*, pages 1164–1182, Berlin, Heidelberg, 2008. Springer-Verlag.

[222] E. G. Sirer and B. N. Bershad. Using Production Grammars in Software Testing. In USENIX, editor, *Proceedings, Domain-Specific Languages (DSL'99)*, pages 1–13. USENIX, 1999.

[223] R. Stallman. *The C Preprocessor*. GNU Project, Free Software Foundation, July 1992.

[224] Standard ECMA-262. *ECMAScript Language Specification*, 3^{rd} edition, December 1999. Available at http://www.ecma-international.org/publications/standards/Ecma-262.htm.

[225] Standard ECMA-334. *C# Language Specification*, 4^{th} edition, June 2006. Available at http://www.ecma-international.org/publications/standards/Ecma-334.htm.

[226] P. Stevens. Bidirectional Model Transformations in QVT: Semantic Issues and Open Questions. In *Proceedings, Model Driven Engineering Languages and Systems, 10th International Conference, MoDELS 2007*, volume 4735 of *LNCS*, pages 1–15. Springer, 2007.

[227] N. Synytskyy, J.R. Cordy, and T.R. Dean. Robust Multilingual Parsing using Island Grammars. In *Proceedings CASCON'03, 13th IBM Centres for Advanced Studies on Collaborative Research*, pages 149–161. IBM Press, 2003.

[228] C. Szyperski, D. Gruntz, and S. Murer. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Professional, Second edition, 2003.

[229] B. A. Tate. *Bitter Java*. Manning Publications, 2002.

[230] D. A. Thomas. The Impedance Imperative — Tuples + Objects + Infosets = Too Much Stuff! *Journal of Object Technology*, 2(5):7–12, 2003.

[231] M. Tomita. LR Parsers for Natural Languages. In *Proceedings of the 10th International Conference on Computational Linguistics and 22nd annual meeting on Association for Computational Linguistics*, pages 354–357, Morristown, NJ, USA, 1984. Association for Computational Linguistics.

[232] *The Unlambda Programming Language*, August 2003. Available at www.madore.org/~david/programs/unlambda. Accessed in September 2010, updated in August 2003.

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

[234] N. Veerman. *Automated Mass Maintenance of Software Assets*. PhD thesis, Vrije Universiteit, January 2007. Available at www.cs.vu.nl/~nveerman/research/thesis/thesis.pdf.

[235] N. P. Veerman. Revitalizing Modifiability of Legacy Assets. *Software Maintenance and Evolution: Research and Practice, Special issue on CSMR 2003*, 16(4–5):219–254, 2004.

[236] E.-J. Verhoeven. COBOL Island Grammars in SDF. Master's thesis, Informatica Instituut, Universiteit van Amsterdam, 2000.

[237] S. Vermolen and E. Visser. Heterogeneous Coupled Evolution of Software Languages. In *Model Driven Engineering Languages and Systems, 11th International Conference, MoDELS 2008, Toulouse, France, September 28 - October 3, 2008. Proceedings*, volume 5301 of *LNCS*, pages 630–644. Springer, 2008.

[238] E. Visser. Scannerless Generalized-LR Parsing. Technical Report P9707, University of Amsterdam, July 1997. Available at www.wins.uva.nl/pub/programming-research/reports/1997/P9707.ps.Z.

[239] E. Visser. *Syntax Definition for Language Prototyping*. PhD thesis, University of Amsterdam, 1997.

[240] E. Visser. Program Transformation with Stratego/XT: Rules, Strategies, Tools, and Systems in Stratego/XT 0.9. In *Domain-Specific Program Generation, International Seminar, Dagstuhl Castle, Germany, March 23-28, 2003, Revised Papers*, volume 3016 of *LNCS*, pages 216–238. Springer, 2004.

[241] VU Browsable Grammars Website. www.cs.vu.nl/grammars/browsable.

[242] World Wide Web Consortium. www.w3.org. Accessed in September 2010.

[243] G. Wachsmuth. Metamodel Adaptation and Model Co-adaptation. In Erik Ernst, editor, *ECOOP'07*, volume 4609 of *LNCS*, pages 600–624. Springer, July 2007.

[244] M. Wallace and C. Runciman. Haskell and XML: Generic Combinators or Type-Based Translation? *ACM SIGPLAN Notices, Conference Proceedings of International Conference on Functional Programming (ICFP'99)*, 34(9):148–159, 1999.

[245] N. Walsh and L. Meullner. *DocBook: The Definitive Guide*. O'Reilly & Associates, 1999.

[246] G. M. Weinberg. *The Psychology of Computer Programming*. Van Nostrand Reinhold, 1971.

[247] M. Weiser. Program Slicing. In *Proceedings of the Fifth International Conference on Software Engineering*, pages 439–449, San Diego, California, United States, March 1981. IEEE Press.

[248] 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.

[249] D. S. Wile. Abstract Syntax from Concrete Syntax. In *Proceedings of the 19th International Conference on Software Engineering (ICSE'97)*, pages 472–480. ACM Press, 1997.

[250] M. H. Williams. A Flexible Notation for Syntactic Definitions. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(1):113–119, 1982.

[251] M. Wimmer and G. Kramler. Bridging Grammarware and Modelware. In *Proceedings of the 4th Workshop in Software Model Engineering (WiSME'05)*, October 2005. Available at www.big.tuwien.ac.at/research/publications/2005/1105.pdf.

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

[253] Website Standards Association Inc. websitestandards.org. Accessed in June 2009.

[254] 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.

[255] D. H. Younger. Recognition and Parsing of Context-free Languages in Time n^3. *Information and Control*, 10(2):189–208, 1967.

[256] D. Yu, A. Chander, N. Islam, and I. Serikov. JavaScript Instrumentation for Browser Security. In *POPL '07: Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 237–249, New York, NY, USA, 2007. ACM Press.

[257] V. Zaytsev. Correct C# Grammar too Sharp for ISO. In *Pre-proceedings of the International Summer School on Generative and Transformational Techniques in Software Engineering, Part II, Participants Workshop*, pages 154–155, Braga, Portugal, July 2005. Extended abstract.

[258] V. Zaytsev. Language Convergence Infrastructure. In *Pre-proceedings of the 3rd International Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE'09)*, pages 399–400, Braga, Portugal, July 2009. Extended abstract.

[259] V. Zaytsev. Language Convergence Infrastructure. In *Post-proceedings of the 3rd International Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE'09)*, August 2010. In print.

[260] V. Zaytsev. Software Language Processing Suite — Grammar Zoo, 2010. http://slps.sourceforge.net/zoo. As of September 2010, contains 3 C grammars, 2 C++ grammars, 10 C# grammars and 3 Java grammars, all of them are freely downloadable in a variety of formats.

[261] V. Zaytsev. *XBGF Manual: BGF Transformation Operator Suite v.1.0*, August 2009. Available at http://slps.sf.net/xbgf.

[262] V. Zaytsev and R. Lämmel. Language Documentation: Survey and Synthesis of a Unified Format. In *Proceedings of the Third International Conference on Software Language Engineering (SLE'10)*, October 2010. Accepted for publication.

[263] V. Zaytsev, R. Lämmel, and T. van der Storm. Software Language Processing Suite, 2008–2010. http://slps.sf.net.