



SWAT

Notation-Parametric Grammar Recovery

Twelfth International Workshop on Language
Descriptions, Tools and Applications (LDTA 2012)

Vadim Zaytsev, SWAT, CWI

2012

Notation-Parametric Grammar Recovery

- Existing software artefacts with grammar knowledge
- Grammar recovery
- Problem: how to reuse grammar artefacts that are written in different notations
- EDD = EBNF Dialect Definition
- edd2rsc and Grammar Hunter
- Software Language Processing Suite

Grammar
recovery
progress and
timeline

Message Sequence Charts

- ITU Z.120, 1996
- Microsoft Word document
→ PostScript document
- PostScript document
→ ASCII file
- ASCII + extract.perl
→ BNF rules
- ...14 manual changes...
- Corrected BNF rules + script
→ HTML

Development, Assessment, and Reengineering of Language Descriptions

Alex Sellink and Chris Verhoef

University of Amsterdam, Programming Research Group
Kruislaan 403, 1098 SJ Amsterdam, The Netherlands

alex@wins.uva.nl, x@wins.uva.nl

Abstract

We discuss tools that aid in the development, the assessment and the reengineering of language descriptions. The assessment tools give an indication as to what is wrong with an existing language description, and give hints towards correction. From a correct and complete language description, it is possible to generate a parser, a manual, and on-line documentation. The parser is geared towards reengineering purposes, but is also used to parse the examples that are contained in the documentation. The reengineered language description is a basic ingredient for a reengineering factory that can manipulate this language. We demonstrate our approach with a proprietary language for real time embedded software systems that is used in telecommunications industry. The described tool support can also be used to develop a language standard without syntax errors in the language description and its code examples.

Categories and Subject Description: D.2.6 [Software Engineering]: Programming Environments—Interactive; D.2.7 [Software Engineering]: Distribution and Maintenance—Restructuring; D.3.4. [Processors]: Parsing.

Additional Key Words and Phrases: Reengineering, System renovation, Language description development, Grammar reengineering, Document generation, Computer aided language engineering (CALE), Message Sequence Charts.

1 Introduction

Since the emerge of computer languages, the need to describe languages in a precise way became an indispensable part of computer science. In his paper on the syntax and semantics of the proposed international algebraic language, Backus [2] writes: ‘we shall need some metalinguistic conventions for characterizing various strings of symbols. To begin, we shall need *metalinguistic formulae*.’ Then he introduced using an example what is now widely known as the Backus-Naur Formalism. In virtually all documents that give a precise language description the method of Backus is used: first the syntax description notation is explained using an example accompanied with some conventions, and then the language description itself follows. In this way myriads of dialects of the Backus-Naur Formalism emerged. They are referred to as BNF, or EBNF, for extended BNF, or metasyntax, metalanguage.

Language descriptions serve more than one purpose: they are used as a guide to implement tools such as compilers or they serve as a reference manual for users. We use language

descriptions to implement tools that serve the reengineering of those languages. Such grammar descriptions form the basis of our approach towards reengineering. Let us give an idea to make this more concrete. It is possible to generate all kinds of prefab components that are useful in an environment for reengineering. We can generate a native pattern language from a context-free grammar that can be used to recognize code fragments [30]. It is possible to generate full documentation for such a language [9]. In [8] we generate components for software renovation factories. A sophisticated parser can be generated from this grammar [17]. A structured editor can be generated from the grammar [22]. It is also possible to generate complete programming environments from context-free grammars. In order to generate such environments, one needs an environment as well. The ASF+SDF Meta-Environment [19] is such an environment. We use it for the generation of tool factories [8]. SDF stands for Syntax Definition Formalism [16], it can be used to define the syntax of a language. ASF stands for Algebraic Specification Formalism [3], it can be used to describe the semantics of a language. The combination is thus adequate for defining syntax and semantics of languages and the ASF+SDF Meta-Environment is the supporting environment for both formalisms.

It is not a trivial task to construct a grammar for reengineering purposes. First of all, such a grammar should have certain properties that make reengineering easy. Secondly, since reengineering problems do not have the habit to reside in small languages, the development process is time consuming. For instance, many academics and companies have struggled with a language definition for COBOL in order to create a decent parser for reengineering targets. Due to the myriad of COBOL dialects, it can be the case that such a grammar itself needs reengineering. Such grammars can suffer from large maintenance problems. In [7] this was called the Year 1999 problem: before that date the grammars had to be ready so that the generated parsers can be used to analyze Year 2000 problems. We refer to [7] for an overview of current parser technology that is used in reengineering and problems that induce maintenance problems on grammars. Since in reengineering, the grammar seems to be the variable and the problem the constant, grammars should be modifiable and tool support should be insensitive to such modifications. Therefore, generating everything from a grammar is in our opinion a solution. According to [28] there are two problems with parser-based technology: first the stringent constraints on the input, and second it is problematic to extend existing parsers. We solve this by

Message Sequence Charts

- ITU Z.120, 1996
- Microsoft Word document
→ PostScript document
- PostScript document
→ ASCII file
- ASCII + extract.perl
→ BNF rules
- ...14 manual changes...
- Corrected BNF rules + script
→ HTML
- Browse all the productions!

Sellink, Verhoef, *Development, Assessment, and Reengineering of Language Descriptions*, CSMR 2000.

Development, Assessment, and Reengineering of Language Descriptions

Alex Sellink and Chris Verhoef

University of Amsterdam, Programming Research Group
Kruislaan 403, 1098 SJ Amsterdam, The Netherlands

alex@wins.uva.nl, x@wins.uva.nl

Abstract

We discuss tools that aid in the development, the assessment and the reengineering of language descriptions. The assessment tools give an indication as to what is wrong with an existing language description, and give hints towards correction. From a correct and complete language description, it is possible to generate a parser, a manual, and on-line documentation. The parser is geared towards reengineering purposes, but is also used to parse the examples that are contained in the documentation. The reengineered language description is a basic ingredient for a reengineering factory that can manipulate this language. We demonstrate our approach with a proprietary language for real time embedded software systems that is used in telecommunications industry. The described tool support can also be used to develop a language standard without syntax errors in the language description and its code examples.

Categories and Subject Description: D.2.6 [Software Engineering]: Programming Environments—Interactive; D.2.7 [Software Engineering]: Distribution and Maintenance—Restructuring; D.3.4. [Processors]: Parsing.

Additional Key Words and Phrases: Reengineering, System renovation, Language description development, Grammar reengineering, Document generation, Computer aided language engineering (CALE), Message Sequence Charts.

1 Introduction

Since the emerge of computers, we describe languages in a precise part of computer science. semantics of a language proposed in Backus [2] was the first. We shall not discuss the details of this process. We shall not discuss the details of this process. We shall not discuss the details of this process.

descriptions to implement tools that serve the reengineering of those languages. Such grammar descriptions form the basis of our approach towards reengineering. Let us give an idea to make this more concrete. It is possible to generate all kinds of prefab components that are useful in an environment for reengineering. We can generate a native pattern language from a context-free grammar that can be used to recognize code fragments [30]. It is possible to generate full documentation for such a language [9]. In [8] we generate components for software renovation factories. A sophisticated parser can be generated from this grammar [17]. A structured editor can be generated from the grammar [22]. It is also possible to generate complete programming environments from context-free grammars. In order to generate such environments, one needs an environment as well. The ASF+SDF Meta-Environment [19] is such an environment. We use it for the generation of tool factories [8]. SDF stands for Syntax Definition Formalism [16], it can be used to define the syntax of a language. ASF stands for Algebraic Specification Formalism [3], it can be used to describe the semantics of a language. The combination is thus adequate for defining syntax and semantics of languages and the ASF+SDF Meta-Environment is the supporting environment for both formalisms.

It is not a trivial task to construct a grammar for reengineering purposes. First of all, the grammar should have certain properties that make it easy to use. Secondly, since reengineering is a complex task, we have the habit to reside in the development process is time consuming. In academia and companies the definition for COBOL in order for reengineering targets. Due to the complexity of the targets, it can be the case that reengineering. Such grammars are used to solve maintenance problems. In [7] this was the problem: before that date the grammar was used to generate parsers. We refer to this as an environment for an environment. In reengineering, the environment is a complex problem. The grammar seems to be a constant, grammars are used to solve problems. The environment should be insensitive to the environment. The environment should be insensitive to the environment. The environment should be insensitive to the environment.



COBOL (dialects)

Obtaining a COBOL Grammar from Legacy Code for Reengineering Purposes

Mark van den Brand, Alex Sellink, Chris Verhoef *

*University of Amsterdam, Programming Research Group
Kruislaan 403, NL-1098 SJ Amsterdam, The Netherlands
markvdb@wins.uva.nl, alex@wins.uva.nl, x@wins.uva.nl*

Abstract

We argue that maintenance and reengineering tools need to have a thorough knowledge of the language that the code is written in. More specifically, for the family of COBOL languages we present a general method to define COBOL dialects that are based on the actual code that has to be reengineered or maintained. Subsequently, we give some typical examples of maintenance and reengineering tools that have been specified on top of such a COBOL grammar in order to show that our approach is useful and leads to accurate and relatively simple maintenance and reengineering tools.

Categories and Subject Description: D.2.6 [Software Engineering]: Programming Environments—Interactive; D.2.7 [Software Engineering]: Distribution and Maintenance—Restructuring; D.2.m [Software Engineering]: Miscellaneous—Rapid prototyping

Additional Key Words and Phrases: Reengineering, System renovation, COBOL.

1 Introduction

There is a constant need for updating and renovating business-critical software systems for many and diverse reasons: business requirements change, technological infrastructure is modernized, the government changes laws, or the third millennium approaches, to mention a few. So, in the area of software engineering the subject of reengineering becomes more and more important. Reengineering is the analysis and renovation of software, see, for instance, [11] for an introductory paper and [7] for an annotated bibliography on this subject. To aid in the analysis and renovation of software it is crucial to have tool support. We strongly believe that such tools largely benefit from having knowledge of the language they have to analyze or renovate. In other words, tools to aid in reengineering should be based on the grammar of the language they intend to process.

As most of the business-critical software is written in COBOL, we will present a general method describing how to obtain a COBOL grammar, given a number of

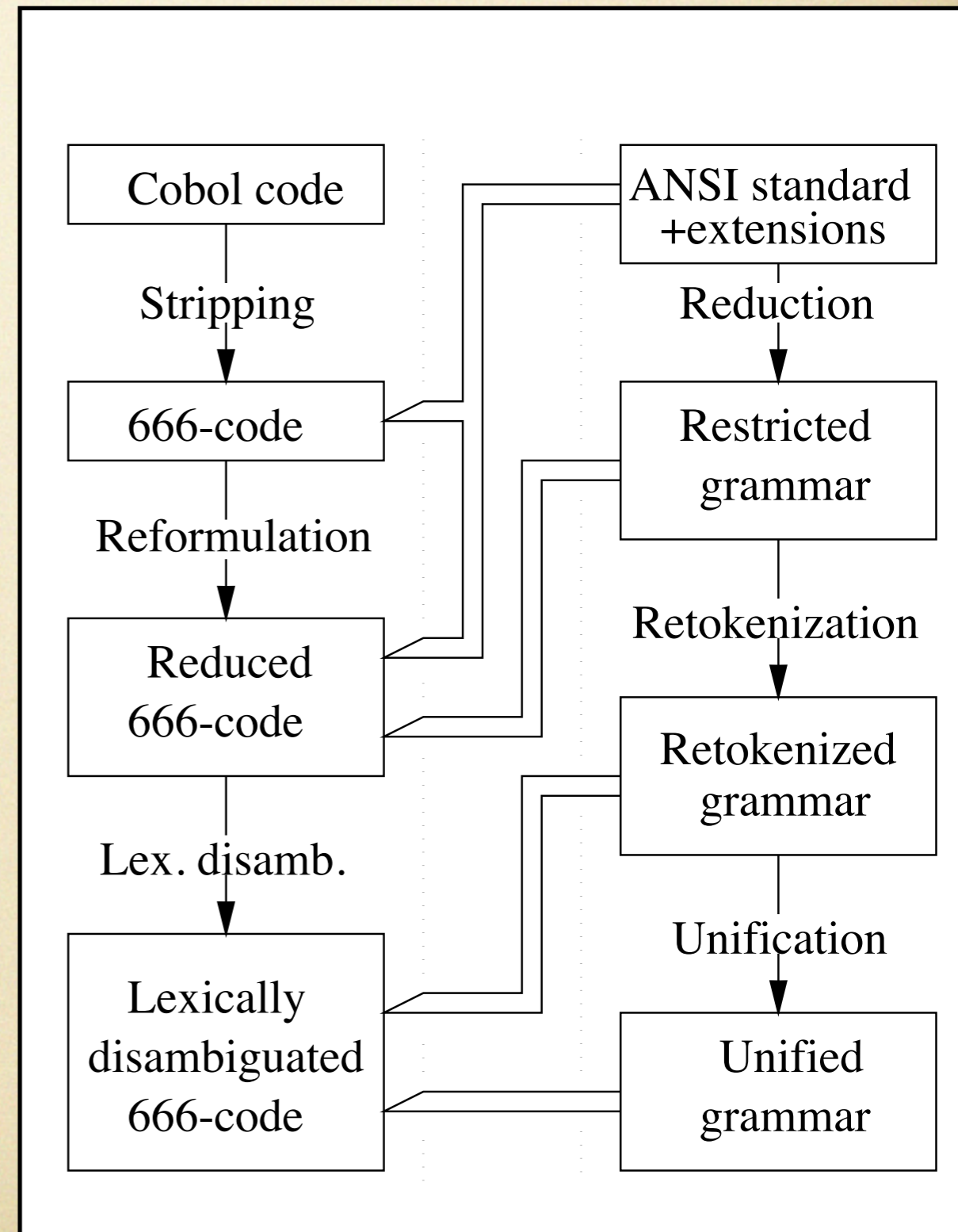
COBOL programs. Moreover, we explain how to reduce the size of this grammar significantly by means of, as we call it, *unification of production rules*. Finally, we show that tools that are based on such a compact language definition are useful in reengineering the code written in that language. Thus, the challenge of specifying reengineering tools lies in the (compact) specification of the underlying language. When it comes to defining syntax is is obviously better to use a parser generator, like Lex+Yacc than to write a parser by hand. Therefore, it is not surprising that, e.g., Software Refinery [24], which is based on generic programming language technology incorporates a number of reengineering tools (for more connections between reengineering and generic language technology we refer to [6]).

To emphasize the importance of tools based on language definitions we give a small example. Suppose that we need a tool that analyzes COBOL programs by looking for date-related variables and that this tool is solely based on lexical scanning. This implies that the tool will list comments containing the patterns it is looking for, as well. To reduce the number of so-called false-positives it is sensible to modify this tool so that listing of comments will be suppressed. In fact, the tool now contains knowledge of the language it is analyzing. Still many false-positives will be found for various reasons. This can mostly be solved in an ad-hoc manner by adding extra knowledge of the language. This procedure has to be reiterated for a tool that needs to perform another task. In fact, most of the time a tool developer is busy with adding knowledge of the language to the tool instead of implementing the actual reengineering or maintenance task of the tool. So in our opinion it is a better idea to approach this in a more structured manner, by specifying COBOL syntax and subsequently specifying tools on top of that syntax. Which brings us to the matter of defining COBOL syntax. There is a myriad of COBOL dialects. We will discuss in this paper a general method describing how to obtain a practical COBOL grammar that recognizes a particular dialect. Then we discuss how tools can be specified that aid in reengineering and maintenance tasks. At this point the reader may observe that although putting more knowledge in a tool implies less false-positives, the processing overhead will increase. However, since legacy systems are notoriously large, the reduction of false-positives is a serious matter. In the end

*Chris Verhoef was supported by the Netherlands Computer Science Research Foundation (SION) with financial support from the Netherlands Organization for Scientific Research (NWO), project *Interactive tools for program understanding*, 612-33-002.

COBOL (dialects)

- Code → strip → reformulate
→ disambiguate → ...
- ANSI COBOL 85 standard
- 1100 production rules + MSc student → SDF
- Grammar
→ restricted grammar
- Restricted grammar
→ retokenised grammar
- Cleaned up grammar can parse cleaned up code



Switching System Language

- Ericsson Reengineering Center: HTML files
- HTML files → syntax rules
- SBNF parser + syntax rules → grammar in SDF?
- ...naming convention violations...
- ...non-matching brackets...
- ...interactive grammar hacking in MetaEnv...
- SBNF' parser + syntax rules' → grammar in SDF!

Development, Assessment, and Reengineering of Language Descriptions

Alex Sellink and Chris Verhoef

University of Amsterdam, Programming Research Group
Kruislaan 403, 1098 SJ Amsterdam, The Netherlands

alex@wins.uva.nl, x@wins.uva.nl

Abstract

We discuss tools that aid in the development, the assessment and the reengineering of language descriptions. The assessment tools give an indication as to what is wrong with an existing language description, and give hints towards correction. From a correct and complete language description, it is possible to generate a parser, a manual, and on-line documentation. The parser is geared towards reengineering purposes, but is also used to parse the examples that are contained in the documentation. The reengineered language description is a basic ingredient for a reengineering factory that can manipulate this language. We demonstrate our approach with a proprietary language for real time embedded software systems that is used in telecommunications industry. The described tool support can also be used to develop a language standard without syntax errors in the language description and its code examples.

Categories and Subject Description: D.2.6 [Software Engineering]: Programming Environments—Interactive; D.2.7 [Software Engineering]: Distribution and Maintenance—Restructuring; D.3.4. [Processors]: Parsing.

Additional Key Words and Phrases: Reengineering, System renovation, Language description development, Grammar reengineering, Document generation, Computer aided language engineering (CALE), Message Sequence Charts.

1 Introduction

Since the emerge of computer languages, the need to describe languages in a precise way became an indispensable part of computer science. In his paper on the syntax and semantics of the proposed international algebraic language, Backus [2] writes: 'we shall need some metalinguistic conventions for characterizing various strings of symbols. To begin, we shall need *metalinguistic formulae*.' Then he introduced using an example what is now widely known as the Backus-Naur Formalism. In virtually all documents that give a precise language description the method of Backus is used: first the syntax description notation is explained using an example accompanied with some conventions, and then the language description itself follows. In this way myriads of dialects of the Backus-Naur Formalism emerged. They are referred to as BNF, or EBNF, for extended BNF, or metasyntax, metalanguage.

Language descriptions serve more than one purpose: they are used as a guide to implement tools such as compilers or they serve as a reference manual for users. We use language

descriptions to implement tools that serve the reengineering of those languages. Such grammar descriptions form the basis of our approach towards reengineering. Let us give an idea to make this more concrete. It is possible to generate all kinds of prefab components that are useful in an environment for reengineering. We can generate a native pattern language from a context-free grammar that can be used to recognize code fragments [30]. It is possible to generate full documentation for such a language [9]. In [8] we generate components for software renovation factories. A sophisticated parser can be generated from this grammar [17]. A structured editor can be generated from the grammar [22]. It is also possible to generate complete programming environments from context-free grammars. In order to generate such environments, one needs an environment as well. The ASF+SDF Meta-Environment [19] is such an environment. We use it for the generation of tool factories [8]. SDF stands for Syntax Definition Formalism [16], it can be used to define the syntax of a language. ASF stands for Algebraic Specification Formalism [3], it can be used to describe the semantics of a language. The combination is thus adequate for defining syntax and semantics of languages and the ASF+SDF Meta-Environment is the supporting environment for both formalisms.

It is not a trivial task to construct a grammar for reengineering purposes. First of all, such a grammar should have certain properties that make reengineering easy. Secondly, since reengineering problems do not have the habit to reside in small languages, the development process is time consuming. For instance, many academics and companies have struggled with a language definition for COBOL in order to create a decent parser for reengineering targets. Due to the myriad of COBOL dialects, it can be the case that such a grammar itself needs reengineering. Such grammars can suffer from large maintenance problems. In [7] this was called the Year 1999 problem: before that date the grammars had to be ready so that the generated parsers can be used to analyze Year 2000 problems. We refer to [7] for an overview of current parser technology that is used in reengineering and problems that induce maintenance problems on grammars. Since in reengineering, the grammar seems to be the variable and the problem the constant, grammars should be modifiable and tool support should be insensitive to such modifications. Therefore, generating everything from a grammar is in our opinion a solution. According to [28] there are two problems with parser-based technology: first the stringent constraints on the input, and second it is problematic to extend existing parsers. We solve this by

Programming Language for EXchanges

feature
programming languages

Cracking the 500-Language Problem

Ralf Lämmel and Chris Verhoef, Free University of Amsterdam

At least 500 programming languages and dialects are available in commercial form or in the public domain, according to Capers Jones.¹ He also estimates that corporations have developed some 200 proprietary languages for their own use. In his 1998 book on estimating Year 2000 costs, he indicated that systems written in all 700 languages would be affected.² His findings inspired many Y2K whistle-blowers to characterize this situation as a major impediment to solving the Y2K

problem; this impediment became known as the 500-Language Problem.

In 1998, we realized that we had discovered a breakthrough in solving the 500LP—so we had something to offer regarding the Y2K problem. We immediately informed all the relevant Y2K solution providers and people concerned with the Y2K awareness campaign. In answer to our emails, we received a boilerplate email from Ed Yourdon explaining that the 500LP was a major impediment to solving the Y2K problem (which we knew, of course). Ed was apparently so good at creating awareness that this had backfired on him: he got 200 to 300 messages a day with Y2K questions and was no longer able to read, interpret, and answer his email other than in “write-only” mode. Although he presumably missed our input, his response regarding the 500LP is worth quoting:

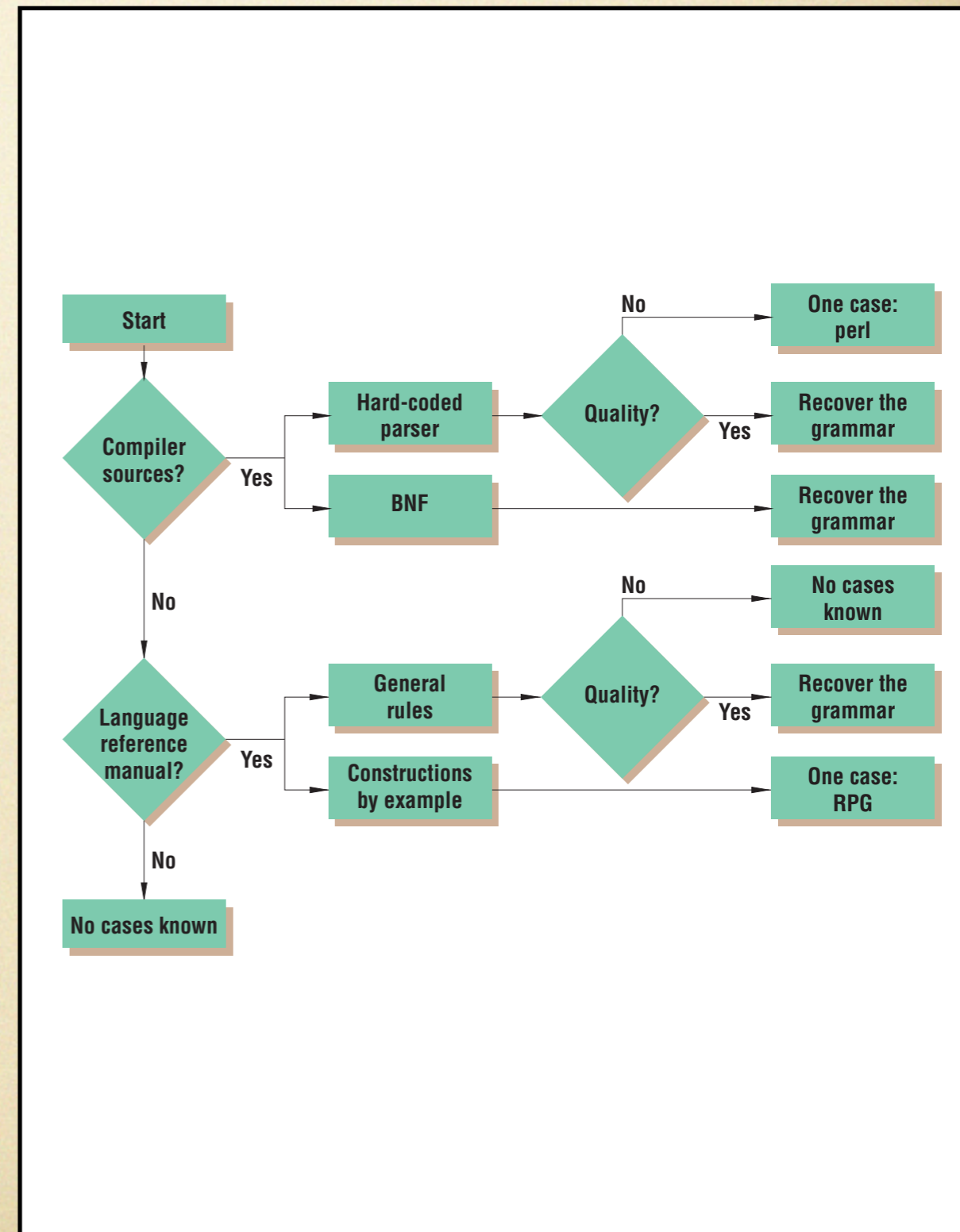
I recognize that there is always a chance that someone will come up with a brilliant solution that everyone else has overlooked, but at this late date, I think it's highly unlikely. In particular, I think the chances of a “silver bullet” solution that will solve ALL y2k problems is virtually zero. If you think you have such a solution, I have two words for you: embedded systems. If that's not enough, I have three words for you: 500 programming languages. The immense variety of programming languages (yes, there really are 500!), hardware platforms, operating systems, and environmental conditions virtually eliminates any chance of a single tool, method, or technique being universally applicable.

The number 500 should be taken poetically, like the 1,000 in the preserving process for so-called 1,000-year-old eggs, which last only 100 days. For a start, we

Parser implementation effort dominates the construction of software renovation tools for any of the 500+ languages in use today. The authors propose a way to rapidly develop suitable parsers: by stealing the grammars. They apply this approach to two nontrivial, representative languages, PLEX and VS Cobol II.

Programming Language for EXchanges

- 20 sublanguages (sectors)
- 63 Mb of source code
→ search for BNF
- BNF in comments + 6 parsers
→ BNF
- BNF → SDF
- Lexer → SDF
- ...combine...
- Parse 8 MLOC in 2 weeks



IBM VS COBOL II

- Language reference
→ raw grammar
- Non-executable source
⇒ static errors
- ??? → lexical syntax
- Test-driven correction & completion
- ... → beautification → modularisation → ...
- Disambiguation
- Adaptation

SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2001; 12:1–6

Prepared using *speauth.cls* [Version: 2000/03/16 v2.12]

Semi-automatic Grammar Recovery



R. Lämmel^{1,2,†}, C. Verhoef^{*2,‡}

¹ *CWI, Kruislaan 413, 1098 SJ Amsterdam, The Netherlands*

² *Division of Mathematics and Computer Science, Vrije University Amsterdam, De Boelelaan 1081a, 1081 HV Amsterdam, The Netherlands*

SUMMARY

We propose an approach to the construction of grammars for *existing* languages. The main characteristic of the approach is that the grammars are not constructed from scratch but they are rather recovered by extracting them from language references, compilers, and other artifacts. We provide a structured process to recover grammars including the adaptation of raw extracted grammars and the derivation of parsers. The process is applicable to possibly all existing languages for which business critical applications exist. We illustrate the approach with a non-trivial case study. Using our process and some basic tools, we constructed in a few weeks a complete and correct VS COBOL II grammar specification for IBM mainframes. In addition, we constructed a parser for VS COBOL II, and were the first to publish a (web-enabled) grammar specification so that others can use this result to construct their own grammar-based tools for VS COBOL II or derivatives.

KEY WORDS: Reengineering, System renovation, Software renovation factories, Grammar engineering, Grammar recovery, Grammar reverse engineering, VS COBOL II, COBOL

INTRODUCTION

Languages play a crucial role in software engineering. Conservative estimates indicate that there are at least 500 languages and dialects available in commercial form or in the public domain. On top of that, Jones estimates that some 200 proprietary languages have been developed by corporations for their own use [1, p. 321]. If we put the age of software engineering at 50 years, this implies that on the average, more than once a month a new language is born somewhere (14 times per year). To illustrate that this estimate is conservative, compare this to Weinberg who estimated as early as in 1971 that in

*Correspondence to: Division of Mathematics and Computer Science, Vrije University Amsterdam, De Boelelaan 1081a, 1081 HV Amsterdam, The Netherlands

†E-mail: ralf@cs.vu.nl

‡E-mail: x@cs.vu.nl

Contract/grant sponsor: The first author received partial support from the Netherlands Organization for Scientific Research (NWO) under the *Generation of Program Transformation Systems* project.

Copyright © 2001 John Wiley & Sons, Ltd.

Received 1 December 2000

Revised 30 July 2001

Accepted 7 August 2001

ECMA/ISO C#

- PDF → text
- Text → LLL
- LLL + FST + GDK → LLL
 - %redefine ... %to ...
- LLL + GDK → SDF
- SDF + tool? → HTML

Correct C# Grammar too Sharp for ISO

Vadim Zaytsev

Vrije Universiteit Amsterdam, The Netherlands,
vadim@cs.vu.nl

Introduction. The most used programming language nowadays is COBOL. At the Free University in Amsterdam we have done numerous transformations on COBOL, parsed and transformed millions of lines of code. COBOL is standardised, but vendors usually deviate from the standard, making their own dialects. In order to parse code, we need a working grammar, which should be derived from the compiler documentation. However, documentation is never complete nor error-free, and special techniques are needed to obtain correct grammars: grammar recovery and grammar (re)engineering. One can argue whether this happens because of COBOL decades-long evolution and legacy.

Recently we started thinking about transforming C# code, too. C# is quite different from COBOL, it is a very sharp modern language, the latest big accomplishment in programming languages design. C# was produced by a big corporation and submitted as a specification to both ECMA International¹ and ISO². C# compiler provided by Microsoft claims to fully implement the standard. Thus, one might think that this standard is of much better quality than COBOL's, making it easier to use it in parser construction. This research piece shows that it is not.

Specification quality: being sharp upon C#. The C# specification is almost 500 pages long, it is written in English, explains all language features in detail, and has an appendix with the formal language definition in a BNF-like form (the same formulae are used throughout the text). One might suppose it would be very easy to take that grammar and transform it into a working parser (which is needed for our re-engineering purposes). Unfortunately, it did not work out that easy: the C# specification's formal contents turned out to be unusable "as is". This means: no compiler. Actually, not even one line of code could have been parsed with that specification—so inconsistent was it.

In order to get to the parser, we took the BNF grammar apart from the text and put it into GDK³, which is expected to generate SDF formulae from it (for use in the ASF+SDF Meta-Environment). This process showed that some BNF formulae are **informally** described ("separated by", "one of the following"), some are **redundant** (occur more than once, some sorts have identical definitions), some **incorrect** (e.g., forgotten "optional" marks), some **inconsistent** (formulae given in the text and in the appendix differ), some **non-intuitive** (e.g., expressions unintelligibly presented without priorities made implicit), some **idiosyncratic** (omnipresent YACCified constructions), some **ambiguous** (the

¹ *European Computer Manufacturers Association*. Here the ECMA-334 is meant.

² *International Organization for Standardization*. C# is ISO/IEC 23270:2003.

³ *Grammar Deployment Kit* by C. Verhoef, R. Lämmel and J. Kort.

Fortran, Modula, BNF, EBNF, YACC

- SDF → BGF
- ANTLR → BGF
- DCG → BGF
- TXL → BGF
- LLL → BGF
- → BGF

An Introduction to Grammar Convergence

Ralf Lämmel and Vadim Zaytsev

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

Abstract. Grammar convergence is a lightweight verification method for establishing and maintaining the correspondence between grammar knowledge ingrained in all kinds of software artifacts, e.g., object models, XML schemas, parser descriptions, or language documents. The central idea is to extract grammars from diverse software artifacts, and to transform the grammars until they become syntactically identical. The present paper introduces and illustrates the basics of grammar convergence.

1 Introduction

Grammar convergence is a lightweight verification method for establishing and maintaining the correspondence between grammar knowledge ingrained in all kinds of software artifacts. In fact, it is an integrated method that works purposely across different programming and specification languages as well as different approaches to software development. Here are few use cases for grammar convergence:

- Given are Java classes for a specific domain, say financial exchange. There is also an independently designed XML schema that is meant to standardize that domain. One needs to establish the agreement between the object model and the schema.
- Given is a compiler for a programming language, say gcc for C++. There is also a reverse/re-engineering tool for the same language based on a different parsing infrastructure. One needs to establish that both tools agree on the language at hand.
- Given is an XML-data binding technology. One needs to test the (customizable) mapping from XML schemas to object models. The oracle for testing relies on establishing an agreement between XML schemas and object models.
- Given are 3 versions of the Java language specification, with 2 grammars per version. One needs to align grammars per version and express the evolution from version to version. (We have done such a case study; see the authors' website.)

The central idea of grammar convergence is to extract grammars from diverse software artifacts, and to transform the grammars until they become syntactically identical. In more detail, the method entails the following core ingredients:

1. A unified *grammar format* that effectively supports abstraction from specialities or idiosyncrasies of the grammars as they occur in software artifacts in practice.
2. A *grammar extractor* for each kind of artifact – e.g., a Java extractor maps Java classes to the unified grammar format.
3. A *grammar comparer* that determines and reports grammar differences in the sense of deviations from syntactical equality (if any).

Java 1.0, 1.2, 5.0

- HTML → BNF
- HTML + very robust scanner
→ something
- Something + heuristics
→ something better
- Something better + initial
correction → grammar
- Grammar + transformations
→ anything

*Recovering grammar relationships for the
Java Language Specification*

Software Quality Journal

ISSN 0963-9314
Volume 19
Number 2

Software Qual J (2011)
19:333-378
DOI 10.1007/
s11219-010-9116-5

**Software
Quality
Journal**

VOLUME 19 NUMBER 2 JUNE 2011

Editor-in-Chief
Rachel Harrison

 Springer

Available
online
www.springerlink.com

 Springer

ISO: C, C++, C#

- PDF → TXT
- Assume the formalism
→ preliminary grammar
- Apply heuristics
→ automated corrections
- Manual analysis → post-extraction transformations
- Automated analyses → ...
- ... → SLPS Zoo



Ada, C++, Eiffel, Modula, MediaWiki, LLL, ISO EBNF

- Grammar Hunter
- Grammar text + EBNF dialect definition → BGF
- Works in steps
- Needs XBGF to make a complete framework.
- Solves two problems:
 - deal with large range of metasyntax dialects
 - disregard typographic (& other) errors

arXiv:1107.4661v1 [cs.MM] 23 Jul 2011

MediaWiki Grammar Recovery

Vadim Zaytsev, vadim@grammarware.net
SWAT, CWI, NL

July 26, 2011

1 Introduction

Wiki is the simplest online database that could possibly work [41]. It usually takes a form of a website or a webpage where the presentation is predefined to some extent, but the content can be edited by a subset of users. The editing ideally does not require any additional software nor extra knowledge, takes place in a browser and utilises a simple notation for markup. Currently there are more than a hundred of such notations, varying slightly in concrete syntax but mostly providing the same set of features for emphasizing fragments of text, making tables, inserting images, etc [10]. The most popular notation of all is the one of MediaWiki engine, it is used on Wikipedia, Wikia and numerous Wikimedia Foundation projects.

In order to facilitate development of new wikiware and to simplify maintenance of existing wikiware, one can rely on methods and tools from software language engineering. It is a field that emerged in recent years, generalising theoretical and practical aspects of programming languages, markup languages, modelling languages, data definition languages, transformation languages, query languages, application programming interfaces, software libraries, etc [15, 23, 25, 70] and believed to be the successor for the object-oriented paradigm [14]. The main instrument of software language engineering is on disciplined creation of new domain specific languages with emphasis on extensive automation. Practice shows that automated software maintenance, analysis, migration and renovation deliver considerable benefits in terms of costs and human effort compared to alternatives (manual changes, legacy rebuild, etc), especially on large scale [11, 61, 65]. However, automated methods do require special foundation for their successful usage.

Wikiware (wiki engines, parsers, bots, etc) is a specific case of grammarware (parsers, compilers, browsers, pretty-printers, analysis and manipulation tools, etc) [25, 75]. The most straightforward definition of grammarware can be of software which input and/or output must belong to a certain language (i.e., can be specified implicitly or explicitly by a formal grammar). An operational 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

EBNF
Dialect
Definition

EBNF Dialect Definition

`program ::=`

`function+;`

`function ::=`

`name argument* "=" expr?;`

- List of indicators
- Together form a notation specification

LLL in LLL

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

LLL in EDD

defining metasymbol	:	definition separator metasymbol	
terminator metasymbol	;	postfix optionality metasymbol	?
postfix star metasymbol	*	postfix plus metasymbol	+
start terminal metasymbol	"	end terminal metasymbol	"
start group metasymbol	(end group metasymbol)
start separator list star metasymbol	{	end separator list star metasymbol	}*
start separator list plus metasymbol	{	end separator list plus metasymbol	}+

Semi-automatic
recovery

Semi-automatic recovery

- Assume the absence of (notational) errors
- Obtain a notation specification
- Generate a parser specification (“grammar for grammars”)
- Fix errors interactively as parsing errors
- Effectiveness depends on IDE support

LLL in Rascal

```
module LLL
import util::IDE; // needed only for advanced IDE support (see last two lines)
start syntax LLLGrammar = LLLLayoutList LLLProduction* LLLLayoutList;
syntax LLLProduction = LLLNonterminal ":" {LLLDefinition "|" }+ ";";
syntax LLLDefinition = LLLSymbol+;
syntax LLLSymbol
= @category="Identifier" nonterminal: LLLNonterminal
| @category="Constant" terminal: LLLTerminal
| group: "(" LLLDefinition ")"
| optional: LLLSymbol "?"
| star: LLLSymbol "*"
| plus: LLLSymbol "+"
| sepliststar: "{" LLLSymbol LLLSymbol "}" "*"
| seplistplus: "{" LLLSymbol LLLSymbol "}" "+";
lexical LLLTerminal = "\"" LLLTerminalSymbol* "\"";
lexical LLLTerminalSymbol = ![""];
lexical LLLNonterminal = [A-Za-z_01-9\-\-]+ !>> [A-Za-z_01-9\-\-];
layout LLLLayoutList = LLLLayout* !>> [\t-\n \r \ ] !>> "#";
lexical LLLLayout = [\t-\n \r \ ] | LLLComment ;
lexical LLLComment = @category="Comment" "#" ![\n]* [\n];
Tree getLLL(str s,loc z) = parse(#LLLGrammar,z);
public void registerLLL() = registerLanguage("LLL","lll",getLLL);
```



```
1 # Idents:      153
2 # - top:      1
3 # - used:     152
4 # - defined:  145
5 # - undefined: 8
6 # Literals:   121
7
8 ref-or-out
9   : "ref"
10  | "out"
11  ;
12
13 expression-unary-operator
14   : lex-csharp-extra/plus
15   | lex-csharp-extra/minus
16   | increment-decrement
17   | "!"
18   | "~"
19   | "*"
20   ;
21
22 increment-decrement
23   : "++"
24   | "--"
25   ;
26
27 expression-shift-operator
28   : "<<"
29   | ">>"
30   ;
31
32 expression-relational-operator
33   : lex-csharp-extra/less-than
34   | lex-csharp-extra/greater-than
35   | "<="
36   | ">="
37   | "is"
38   | "as"
39   ;
40
41 expression-equality-operator
42   : "=="
43   | "!="
44   ;
45
46 conversion-kind
```


Automatic recovery

Automatic recovery

- Assume the notational errors will happen
- Obtain a specification of the correct notation
- Perform robust parsing with it
- Infer heuristics and encode them in a tool
- Recover from all errors automatically
- Once finished, the grammar can be analysed, corrected etc

Grammar Hunter

Block 1: Selective line reading.

- Reads the file, fetches grammar fragments, applies line continuation rules to relevant lines, filters out comments, delivers the list of characters.

Block 2: Composition of tokens from characters.

- Transforms the list of characters into the list of tokens, while taking quoting rules into account.

Block 3: Tokens classification.

- Classifies each token as a terminal, nonterminal or a metasymbol.

Block 4: Token groups normalisation.

- Converts postfix/prefix to confix, delivers the list of grammar rules.

Block 5: Context-dependent reconsideration.

- Performs correction heuristics: decomposes and assembles symbols, rebalances symmetric metasymbols, ignores negligible leftovers.

Block 1:

Selective line reading

Formal Parameters

Every function declaration includes a formal parameter list, which consists ...

The following can be simplified to:

```
formalParameterList
```

```
: '(' normalFormalParameters (',' optionalFormalParameters)? ')'
```

```
;
```

```
optionalFormalParameters
```

```
: restFormalParameter |
```

```
namedFormalParameters
```

```
;
```

```
normalFormalParameters:
```

```
normalFormalParameter (',' normalFormalParameter)*
```

```
;
```

Positional Formals

A positional formal parameter is a simple variable declaration.

Block 2:

Composition of tokens from chars

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

SwitchBlockStatementGroups

`<page-first-char> ::= <ucase-letter> | <digit> | <uscore> | ...?`

`Primary.new Identifier (ArgumentListopt) ClassBodyopt`

Block 3:

Tokens classification

```
Line = PlainText { PlainText } { " " { " " } PlainText { PlainText } } ;
```

nonterminal
symbol

terminal
symbol

metasymbol

Block 4:

Token groups normalisation

- Only terminator metasyMBOL is known
 - the less reliable is the notation, the more errors we get

```
foo ::= bar iwx fwc ysk ;  
uwr ::= wzx abq iin ync  
hnl ::= pjx hwz gwo pai ;;  
djr ::= bcx opv nfx rcj  
bwf ::= tbv kle gbx xik ;
```

Block 4:

Token groups normalisation

- Only defining metasymbol is known
 - better because left hand side is a nonterminal (CFG)

foo = bar iwx fwc ysk
uwr = wzx abq "iin" = ync
hnl = = pjx "hwz" gwo pai
djr = bcx opv nfx rcj
bwf = tbv kle gbx "xik"

Block 4:

Token groups normalisation

- Both terminator and defining metasymbols are known
 - additional validation leads to stability

```
foo = bar iwx fwc ysk ;  
uwr ::= wzx abq "iin" = ync ;  
hnl = = pjx "hwz" gwo pai ;;  
djr = bcx opv nfx rcj  
bwf = tbv kle gbx "xik" ;
```

Block 4:

Token groups normalisation


- Neither terminator nor defining metasymbols are known
 - infer by frequency analysis of tokens

```
foo ::= bar iwx fwc ysk ;  
uwr ::= wzx abq iin ync ;  
hnl ::= pjx hwz gwo pai ;  
djr ::= bcx opv nfx rcj ;  
bwf ::= tbv kle gbx xik ;
```

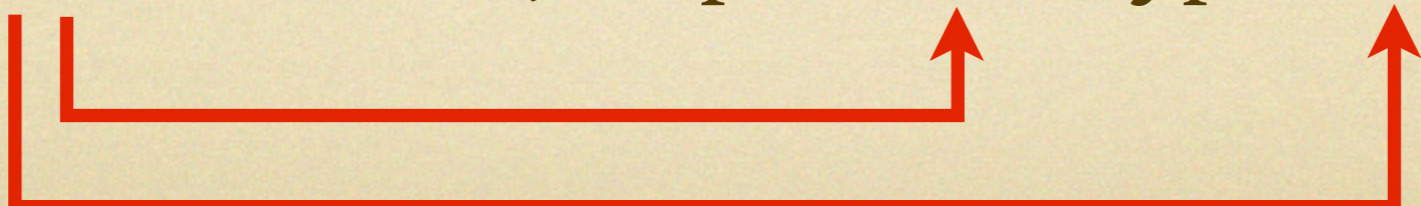
Block 5:

Context-driven reconsideration

VariableDeclaratorId:
Identifier
VariableDeclaratorId []



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



Conclusion

- A victory for grammar recovery
- Syntactic notation specification
- Semi-automatic:
 - generate a parser spec from a notation spec
 - work interactively
- Automatic:
 - encode heuristics and let them loose
- Beyond (E)BNF?

Thank you!

grammarware.net

slps.sf.net