Semi-automatic Grammar Recovery

R. Lämmel* and C. Verhoef**

*Centrum voor Wiskunde en Informatica,
Kruislaan 413, 1098 SJ Amsterdam, The Netherlands

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

ralf@cwi.nl, x@wins.uva.nl

Abstract

We proposed a new approach for the construction of grammars and parsers for existing languages. The approach is both very powerful and simple. We provided a structured process and explained our methods in detail so that others can apply our ideas for their own grammar construction activities. We illustrated the proposed approach with a nontrivial case study. Using our process, we constructed in a few weeks a complete and correct VS COBOL II grammar specification for IBM mainframes. We not only constructed a parser for it, but also published a web-enabled grammar specification so that others can use this result to conveniently construct their own grammar-based tools for VS COBOL II, or derivatives.

Categories and Subject Description: D.2.7 [**Software Engineering**]: Distribution and Maintenance—Restructuring.
Additional Keywords and Phrases: Reengineering, System renovation, Software renovation factories, Grammar engineering, Grammar Recovery, Grammar reverse engineering, VS COBOL II, COBOL.

1 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 [62, loc. cit. 321]. If we put the age of software engineering on 50 years, this implies that on the average, more than once a month a new language is born somewhere. To illustrate that this estimate is conservative, compare this to Weinberg who estimated already in 1971 that in 1972 programming languages will be invented at the rate of one week—or more, if we consider the ones which never make it to the literature, and enormously more if we consider dialects [118, p. 242].

It will not be a surprise that there are entire conferences devoted to languages, that there are several journals devoted to computer languages, some even to a particular computer language. There is an abundance of news groups

in the `comp.lang` hierarchy devoted to computer languages. Not only programming languages play a crucial role, but also languages dealing with other parts of the software life cycle, such as modeling languages of which OMT [99] and UML [100] are probably the most prevalently known visual languages. Seen from this perspective it seems that the area of language development is one of the most lively parts of software engineering. Indeed so-called language prototyping and construction tools are becoming a sub-industry of their own to supply the language needs [32, 35]. There is a wealth of tools to aid in the design and implementation of programming, specification and modeling languages. The tools Lex [77] and Yacc [59] are the most well-known ones that come to mind here. But there are many others including academic and commercial products: LDL [49], ASF+SDF Meta-Environment [66], Cocktail [46, 45], Software Refinery [94], TXL [32], TAMPR [14], Eli [44], Gandalf [47], Elegant [88], ToolMaker [108], RainCode [91], COSMOS [38], and so on. A crucial aspect of any language is its grammar. A grammar is the formal specification of the syntactic structure of a language. Such specifications are crucial input to parser generator tools, like Lex and Yacc tools, or other so-called generic tools, like programming environment generators.

Grammars are omnipresent in software engineering. Not solely in the language technology field, but also in other areas. For instance, the parsing of simple input is such a common problem that there are software patterns devoted to the issue [42]. Or in database design, flat-files are designed using grammars [10]. Also document type definitions (DTDs) in the sense of XML [24] can be regarded as grammars. DTDs/XML are widely used to define the abstract representation of structured documents. Note also that reading of grammars is often necessary. Any language reference manual contains grammar descriptions: from telecommunications language standards like Chill [58] to modern programming languages like Java [43] to the Object Constraint Language (OCL) [117] for UML.

"Okay, so many people use grammars, read grammars, there are industry proven tools like Lex and Yacc or design patterns, to deal with them. So, what is the problem? And why do we need to read this paper?" the reader might wonder at this point. Indeed, there is a good reason to continue reading. The last decade, the software engineering field made the transition from being a developing community, to a community where more than half of the professional software engineers are working on the enhancement of existing systems. It is estimated that this trend will continue in this decade [61, loc. cit. 319]. "But what has this to do with grammars?" Tools! Tools! Tools. There is so much code around (about 7 billion function points [111, 60]) that it is no longer possible to deal with the existing codebase by hand. We have to use computers to process programs, systems, or even entire software portfolios. Such tools report us about interesting things we wish to know about the software under investigation. Also there is a constant need for tools that automatically modify entire systems to diminish the workload. All those tools share that they heavily rely on the underlying grammars of the languages that are used in the software systems.

The premise of this article is to provide you with a semi-automated process to recover the grammars of computer languages. With recovery we mean:

extraction, assessment, correction, completion, testing, and so on. We have applied the process to a particularly hard, ancient, but pervasive language to show the power of the approach. We were able to entirely recover IBM's VS COBOL II grammar in two weeks, which includes the development of the necessary tools to accomplish this. We reused existing parser generators to test the recovered grammar. We tested the generated parsers on VS COBOL II code, and parsed about 2 million lines of code. For the uninitiated reader, let us compare this to other productivity measures. Vadim Maslov of Siber Systems is a professional grammar constructor. Based on his own extensive experience in COBOL parser construction (12 dialects of COBOL) he estimates [81] the effort for a single knowledgeable person as follows:

> My prediction is that a quality COBOL parser may as well take you 2–3 years to implement.

Another indication that constructing a COBOL parser is a very laborious task is that there never existed a publically available COBOL grammar, despite the fact that COBOL had its 40th anniversary party in January 2000. Such complex artifacts are worth money and considered trade secrets. Actually, we were the first to publish a freely available COBOL grammar [74]. The published grammar was recovered with the results presented in this paper.

Our approach in a nutshell Our approach is very simple. The grammars are already written, we only have to extract them, transform them into the correct form, and that's it. Let us briefly explain. Grammars reside in compiler source code, in language reference manuals, in language formatters, in complexity metric tools, in function point counting tools, and what have you. Such grammars can be extracted using simple tools. However, that is only part of the story, since those grammars all serve a certain purpose, all with their own design tradeoffs. As a consequence, they are perfect for one purpose but devastating when applied for another task. For instance, for an automated Euro conversion project it is not a good idea to reuse a parser from a compiler. Namely, if the converted system is parsed and converted using a parser that resides in a compiler, it means that files referred to include commands are indeed included, commands for macro expansion are indeed expanded, the syntax is minimalized, embedded languages are expanded by preprocessors, and all useful comments are removed. A system that is modified in such a manner, is unacceptable for its owners because it is turned into a completely unmaintainable system. So for this Euro conversion we need something different, like so-called syntax retention (a term coined by Reasoning Inc.) on locations that are not affected by the modifications, no expansion of macros and include file commands, no expansion of embedded languages (such as SQL or CICS), and all comments should remain in place (possibly even updated to other comments). As is clear from the given example, after we recovered a grammar, we also need to transform it to make the grammar fit for new tasks like software renovation.

Grammar life-cycle enabling It is our experience that often a grammar for a certain purpose can be obtained from another grammar by means of formal grammar transformations. One could call that the *grammar life-cycle*. Although

there can be quite some algorithmic complexity involved in such grammar transformations (viz. the creation of scaffolding grammars [106]), the truly hard part is to first obtain a correct and complete base-line grammar specification so it can be subject to further processing. This paper focuses on the hard part: recovering correct base-line grammars. One could call that grammar life-cycle *enabling*.

In the paper we discuss a case that has significant application potential in the area of automated modifications to existing systems. In our opinion, this is one of the most pressing grammar engineering issues at this moment. The grammars that we need are named *renovation grammars*. They are equipped to make automated modifications to software without all the unwanted side-effects that a typical compiler grammar has. "Is *most pressing* also urgent for others than grammar techies?" Good question. Ed Yourdon popularized the so-called *500 language problem*, a problem observed earlier by Capers Jones in his book [62]. The 500 language problem is the fact that one of the major impediments to solving the Y2K problem with tools, was the inability to parse the code to analyze it and make automated modifications. This was due to the fact that the Y2K problem resided in code that has been written in 700 different languages (500 commercially available plus 200 proprietary ones) for which no parser components were available. Of course, not only the Y2K problem demanded ready availability of renovation grammars, also current and future enhancement projects like Euro conversions, language migrations, redocumentation projects, system comprehension tasks, operating systems migrations, software merging projects due to company merges, web-enabling of mainframe systems, daily mass-maintenance projects, and so on are in need for such grammars. All such software enhancement problems are indeed rather urgent, given the fact that many organizations are spending most of their software dollar to activities after the first release of a software system.

"So there are 7 billion function points of software written in 700 different languages, in constant need for modification?" Yes, that is basically the issue at hand. Speaking of hands, it is not advisable to enhance any seriously sized software portfolio manually. In fact, already for the Y2K problem—which was technically perceived to be simple—Gartner Group [48, 63] advised that any software portfolio over 2 million lines of code should be repaired using a so-called software renovation factory.

Software renovation factories A *Software Renovation Factory* is a product-line architecture that enables rapid development of tools intended to perform mass-changes (like the ones we listed above) on entire software systems. We observed a trend among large software portfolio owners to experiment with or even install such facilities on-site in order to see how they can help enabling the rapid pace of change to their existing software assets. Renovation grammars are crucial assets in the construction and even generation of major components of a software renovation factory. Generative and other general construction issues with respect to software renovation factories are elaborately discussed elsewhere [16, 23, 17, 20, 22, 102, 67, 103, 18, 106, 113]. Here it suffices to realize that grammars are input to software renovation factory generators, in the same way as grammars have been input for parser generators to efficiently implement parsers.

Related work Perhaps due to the firm establishment of parser generation technology in the 1970s it has taken the software renovation field a lot of time to realize that mainstream compiler oriented parser generation technology is problematic. In the paper [21] we posed that mainstream parser generation technology is harmful and that we need alternatives. Therefore, not much related work other than ours can be found, although a renaissance of grammar oriented research is to be expected. A positive sign in that direction is an elaborate PhD Thesis [11], in which backtracking extensions of mainstream lexer and parser generation technology (and their integration) are proposed that are able to conveniently parse ancient languages, like COBOL, PL/I, and so on. Other technologies like Generalized LR parsing [76, 110, 95, 114, 116] are also used to parse ancient languages [20]. Work relating to the recovery of the grammars itself is still an exception in the software engineering community. Our first attempt in grammar recovery was published as an extended abstract in 1998 [101]. In that paper, the grammar of a proprietary language for programming switching systems of Ericsson was subject to recovery. The grammar was recovered from an on-line manual that contained the language definition. Although that paper failed to come up with a working parser (the on-line documentation contained much less than the actual language comprised) we see the paper as an important contribution. For, the very same technology was used fruitfully to entirely recover the same switching system language grammar from the source code of its compiler [104]. After this result was achieved, we realized that grammar recovery is an important subject, and we decided to publish the full version [105] of [101]. We base ourselves in this paper partly on this full version [105] and show that using its ideas it is possible to recover the grammar of a real language from real documentation other than compiler source code. In this paper we share our experiences with the recovery of an IBM COBOL dialect. Moreover, we generalize on our earlier efforts in grammar reconstruction and come up with a structured recovery process.

The linguistic connection In the realm of natural language processing the need for grammar oriented tools has been recognized in the early 1990s. In the paper [86] the term *lingware engineering* was coined. Grammars of natural languages tend to be large, incomplete and constructed by evolution. These qualifications also apply to the construction of grammars of many ancient programming languages. In linguistics [2, 84], also the related question of grammar *inference* is posed: given a source text, is it possible to infer a grammar from it? In the domain of software renovation this question is sometimes also posed by software engineers: given a legacy system, is there a method to recover the grammar from it? The grammar that parses only that system will be sufficient. In Usenet discussions [112] we have noticed that people are thinking in the same direction as the linguists: grammar inference by *solely* using the source texts. Although we acknowledge the importance of the linguistic approach, the programming language situation is different. While there is probably no existing artifact containing the grammar in some hidden form in the linguistic area, in the software engineering area there is almost always such an artifact: a compiler, a language reference manual, a standard, an interpreter, and so on. So for the recovery of grammars from source code we do not recommend to use the linguistic approach, but to use our proposed methods that make essential use

of the existing abovementioned artifacts. Of course, we use the source texts as debugging aids to incrementally improve the raw extracted grammar.

Grammar transformations We use special-purpose transformations to correct and complete grammars during recovery. Grammar transformations are used by other people for similar or other reasons. In [120], grammar transformations are used for the development of domain-specific languages starting from reusable syntax components. In [119], semantics-preserving grammar transformations are used to derive abstract syntaxes from concrete syntaxes. The other direction is also very common since it is often part of the implementation of a grammar specification with rather restricted parser generators such as Yacc. We comment on that approach in more detail in Section 4.5.

The future We estimate that software renovation factories are becoming the future "compilers" of our existing software assets, meant to "compile" them into improved systems better capable of meeting new business needs. We think that future programming environments will be designed from day one to also support maintenance, enhancement and ultimately renovation. The first signs of this trend are already visible, for instance Microsoft and IBM provide support in some of their contemporary compilers to tap the entire abstract syntax tree (AST) for program comprehension purposes. Also in [52] it is emphasized that language oriented tools ranging from scanners/parsers to software maintenance and renovation tools should be developed as soon as a language sees the light. At the moment we are not in such a situation, but every effort to ease the construction of software renovation factories is important for the IT industry. The enabling technology for software modification tools starts with correct and complete grammars. Since the majority of the existing software systems is written in ancient languages, cost-effective grammar recovery methodology for those languages is of significant importance to the entire IT industry. And how to obtain them is enabled by grammar (re)engineering, the subject of this paper.

Organization The rest of this paper is organized as follows. In Section 2, we discuss the grammar life-cycle that comprises the artifacts containing grammars and their connections. In Section 3, we elaborately treat the case study and show how to recover a complete and correct specification of IBM's VS COBOL II grammar. In Section 4, we explain a structured process to implement a grammar specification in order to derive a realistic parser. The interesting part of that phase is grammar disambiguation. We illustrate the approach by showing how this applies to the VS COBOL II grammar. In Section 5, we shortly discuss grammar engineering tools. Finally, in Section 6, we make some concluding remarks.

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

2 The Grammar Life-Cycle

As illustrated in Section 1, dealing with software implies dealing with grammars. The current state-of-the-art in software engineering is that grammars for different purposes are not related to each other. In an *ideal* situation, all the grammars can be inferred from some base-line grammar. We are not in this ideal situation. With grammar recovery we can enable the grammar life-cycle and deliver the missing grammars in a cost-effective manner so that urgent code modification tasks can rapidly be implemented with tools based on the recovered grammars.

Before discussing the grammar life-cycle we should make clear what its components are. The following artifacts have proved to be useful during the life-cycle of software:

- compilers,
- debuggers resp. animators,
- profilers,
- pretty printers,
- language reference manuals,
- language browsers,
- software analysis tools,
- code preprocessing tools,
- software modification tools,
- and so on.

See [52] for an even more elaborate list. In Section 2.1 we will elaborate on the various grammars that play a role for such tools, and in Section 2.2 we discuss the connections between the different grammars.

2.1 The Grammar Gamut

There are many grammars that we use day in and out, often without realizing it. We mention the most prominent grammars and the tools they reside in, and we discuss differences and similarities.

Compilers The grammar(s) residing in a compiler we call *compiler grammars*. We assume familiarity with the reference architecture of compilers [1]. What is maybe less familiar is that if we write C code, we are not at all using syntax that is in accordance with the grammar that is part of the C compiler. Let us explain what is the matter. The C compiler massages the code we write in many stages before it is actually parsed and further processed for generating code. All comments are removed, since they do not generate any code. The C preprocessor, `cpp`, expands macros and include file syntax. In-lined assembler is being dealt with so that it occurs at the correct location when the conversion to assembler of the rest of the code is performed. Conditional compilation issues are resolved, and so on. Obviously, the grammar of the code is minimalized to make the implementation of the compiler as simple as possible. In other grammars there are similar issues clearly showing that a compiler grammar is not in accordance with the grammar of the code that we actually write. Let

us give a striking example. Here is some COBOL code that is produced by nonnative English software developers.

```
5705.
    ADD 1 TO TELLER BLZTEL.
    IF BLZTEL GREATER THEN 195
        ADD 1 TO HELPTEL
        MOVE TITEL TO TITELREGEL
        CALL 'HELP3' USING HELP321 FUNKTIE1 STUURGEBIED
        MOVE ZERO TO BLZTEL
     ELSE CALL 'HELP3' USING HELP321 FUNKTIE2 STUURGEBIED.
    MOVE TELLER TO VULBLADNUMMER.
```

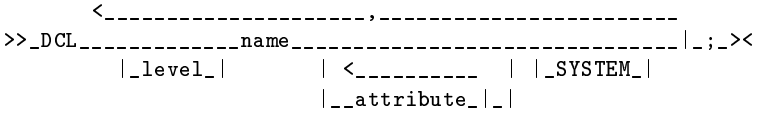
Did you find the error? The OS/VS COBOL compiler did not. But we did: when we parsed the code with our recovered grammar [74], it did not parse the `GREATER THEN` part. The reason is that in the manual it spells `THAN` instead of `THEN`. Now why does the OS/VS COBOL compiler still compile this code without warnings? The compiler first preprocesses the code and removes *all* `THEN` keywords. Since the use of `THAN` is optional, the compiler assumes that this keyword was not used. So due to implementation choices in the OV/VS compiler, it is possible to write this kind of code. This is not an isolated case. There are many more such examples, for instance, we detected the use of `NOT IS` in conditions in code by programmers who are not native English speakers. Also empty sentences (loose dots in the code) often occur. The OS/VS COBOL compiler removes all such things in the preprocessing phase and will accept entirely undocumented syntax without a warning. Due to copy/paste actions by developers, entire systems can be infected with this kind of totally ungrammatical code. In order to *renovate* such code, we must be able to parse this kind of code without problems, for instance, to correct the syntax so that platform migrations can commence. The most important message to remember here is that we need a significantly different grammar for renovation than the ones that exist in products like compilers or manuals.

Pretty printers They are tools that format code in a consistent style. It is not necessary for such a tool to do a deep syntactic analysis of all the aspects of the code in order to pretty print it. We call the grammar of a pretty printer a *formatter grammar*. A formatter grammar is different from a compiler grammar. For instance, `#include "foo.h"` cannot be parsed by a compiler if `foo.h` does not exist, but it should be possible to pretty print such source code. After all, the purpose of a formatter grammar is not to analyze the correctness of the code, but to format it according to some (company) standard. Therefore, a different analysis of the source text is necessary for optimal formatting purposes. More information on the generation of formatters from context-free grammars can be found in [85, 23, 80].

Language reference manuals Sometimes we need to study a manual to check a special language construct, or simply to learn the language. We call a grammar that is contained in a manual a *language reference grammar*. Usually it is not a good idea to look at the source code of a compiler grammar to understand the syntax of the language. Often, a compiler grammar is implemented using parser generation technology, so, its form is not optimal reading material

for humans, since the parser generation algorithms put restrictions on the form of the compiler grammar. Vice versa, a reference manual grammar is usually not suited to be used for parser generation: it is usually ambiguous.

Also visual languages such as syntax diagrams (often found in language reference manuals) are more appropriate for human understanding than other tasks. As an example, we depicted the syntax diagram of the `DCL` construct as we found it in IBM's PL/I language reference manual [53]:



This visual language is to be read as follows: with the `>>` symbol we start reading. The horizontal line is the main path. Required items are on the main path, so the `DCL` is mandatory syntax. And so is the sort `name`. Then we see on the main path a vertical bar with a long back arrow. This indicates a repeatable item. The comma in the back arrow indicates a field separator. Then we find the semi-colon on the main path, and the `><` sign indicates that we are ready. Items below the main path are optional, so `level` is optional; the list of `attribute`s is optional and the keyword `SYSTEM` is. Note that the comma-separated list ranges over the optional `level`, the mandatory `name`, a nested optional list (without separators) and the optional keyword `SYSTEM`. So the scope of the iteration construct is expressed in a two-dimensional visual way, rather than with explicit scope terminators like braces. It will be clear that this visual representation is less suited for parser generators but that it is optimized towards human comprehension of the language syntax [40].

Language browsers This is a variant of a language reference manual: it is a browsable version of it. A comprehensive archive of on-line language reference manuals is provided by IBM [54]. Fast and flexible access to a language reference manual is of course accomplished best nowadays by a hypertext version of the manual. Of course, such manuals should be cross-linked. It is useful to web-enable the grammar part of the manual as well, so that, e.g., by clicking on `attribute` in the above PL/I syntax diagram we will jump to the location where this sort is defined. We call a grammar that is web-enabled a *browsable grammar*. The IBM manuals are using hypertext, but the grammars are not browsable; we developed tools to generate browsable grammars from the recovered unlinked grammars. On the home page [39] a kind of browsable grammar is provided for certain languages.

Software analysis tools Many source code analysis tools exist, each with its own requirements for the grammars it contains. A very rudimentary grammar will do for a function point counting tool based on backfiring, that is, estimating the number of function points by counting logical source lines of code [61]. For a tool that provides rapid insight into the call structure of an entire system, it is only necessary to parse those parts of a system that contribute to the information that constitutes the call-structure. For the McCabe complexity metric [82], we need to parse a little bit more, but not as detailed as is necessary

for a compiler. We call grammars serving the purpose of analyzing software *analyzing grammars*.

Analysis does not need to determine a complete picture of a system. Therefore, also the parsers do not need to be complete and partial parsers suffice. Partial parsers are also called island parsers in computational linguistics [28, 109]. The accompanying partial grammars are therefore sometimes called *island grammars* [36]. With this partial parsing technology, we trade velocity for accuracy and completeness. By picking the relevant pieces of syntax that are analyzed in depth, we hope that the accuracy is not traded off. Before our work on grammar recovery, island parsing also speeded up the development of analysis tools: it was not necessary to implement an entire parser, which was known to be laborious. With the possibilities that our technology opens up we are not sure whether this advantage is still valid. Apart from that, we think that the idea of island parsing is useful. For instance, certain islands of entire systems can be analyzed to redocument call-structures between files and so on without the need for a complete grammar. Parsers for software analysis tools are often so idiosyncratic that even exchanging them among each other is impossible, as was observed in a paper attempting this [96]. This problem is seen as important in the software renovation field as can be seen by the many papers proposing different common exchange formats [121, 65, 70, 37]. Dagstuhl seminar nr 00461 will be devoted to the subject of interoperability between reengineering tools. For those who don't know, Schloß Dagstuhl is a German initiative to bring together leaders in computer science to discuss cutting-edge computer science research at the international forefront on a weekly basis. Moreover there is a Workshop on Standard Exchange Formats dealing with the issue. Although we recognize the importance for software engineering tools—they should be open, and exchange information freely between each other [16, 67, 7, 6, 8, 15]—we sensed in many discussions with this community that the problems as discussed in [96] are one of the major reasons to try to come to a common format is to cope with the difficulties of parser reuse. This problem is, in our opinion, caused by the fact that people try to reuse the parser *output*. Our method does not have the abovementioned problems, since we reuse the grammars underlying the parsers and not just their idiosyncratic output. In that way we can easily construct the output in any format necessary for a particular tool. This should solve in our opinion the interoperability problem for software reengineering tools.

Software modification tools Software modification tools are less wide-spread, but they do exist. Software modification tools contain what we call *renovation grammars*, that is, grammars that can handle the following issues in a satisfactory manner:

- different dialects
- embedded languages
- (home grown) preprocessors
- compiler directives
- unexpanded macros
- unexpanded include files resp. copy books
- debugging lines, continuation lines, etc.
- undocumented syntax (cf. the `GREATER THEN` issue)

- comments
- layout
- scaffolding constructs [106] or annotations

We can summarize the above list by stating that the source code, as is, can be parsed. Moreover, by unparsing the code as much as possible syntax retention should be achieved. Grammars that are capable of parsing the code as is are not widespread, and the purpose of this paper is to use as many as possible knowledge sources to construct (or generate) base-line grammars from which renovation grammars can be generated in a cost-effective manner. Note that there are several related aspects of software modification tools which are however not fully represented by the underlying grammars themselves, for example, certain ways of expanding and collapsing copy books to take them into consideration during modification.

2.2 Connections Between Grammars

We have seen a number of different grammars. In the ideal situation, those grammars are derivable from each other or from some base-line grammar. Moreover the derivations are automated via grammar transformations. The idea of a base-line grammar that is used as the basis for tools is not new. Already in the early eighties several research and commercial projects on the generation of interactive programming environments were started with the aim to generate structured editors, parsers, unparsers, etc. directly from the grammar of a language [51, 66, 107, 34, 64]. In this case, the grammar was in all cases the same artifact. So grammars play a central role in such programming environments. We add to this idea that we do not demand all artifacts to be generated (more or less) *directly* from the base-line grammar. Each artifact can be rather generated from a different grammar, which is, however, obtained from the base-line grammar by possibly elaborate grammar transformations. In that way, there are more possibilities to make use of generic language technology so that we can generate different tools, more efficient tools, or more convenient tools. Most importantly, we are able to use generic language technology to enable tool-supported maintenance and automated reengineering of software; the most urgent and expensive parts of the software life-cycle [79, 97, 12, 62, 90, 83].

In Figure 1, we present a number of language oriented tools: a compiler, a language reference manual, an on-line manual, a pretty printer, some analysis tools (data-flow analyzers, control-flow analyzers, complexity metric tools and so on), renovation tools (language converters, mass maintenance tools, code restructuring systems, software renovation factories, and such), island and full parsers (for extracting specific information such as function points, or call relations or entire ASTs), and other tools (syntax directed editors, interpreters, etc.). Those tools all have some grammar on top of which they are built. The abbreviations in Figure 1 are in one-to-one correspondence with the tool they point to, for instance, AG is short for Analysis Grammar.

The abbreviations above the figure are technologies rather than grammars.

GE is short for Grammar Engineering.

GLT stands for Generic Language Technology.

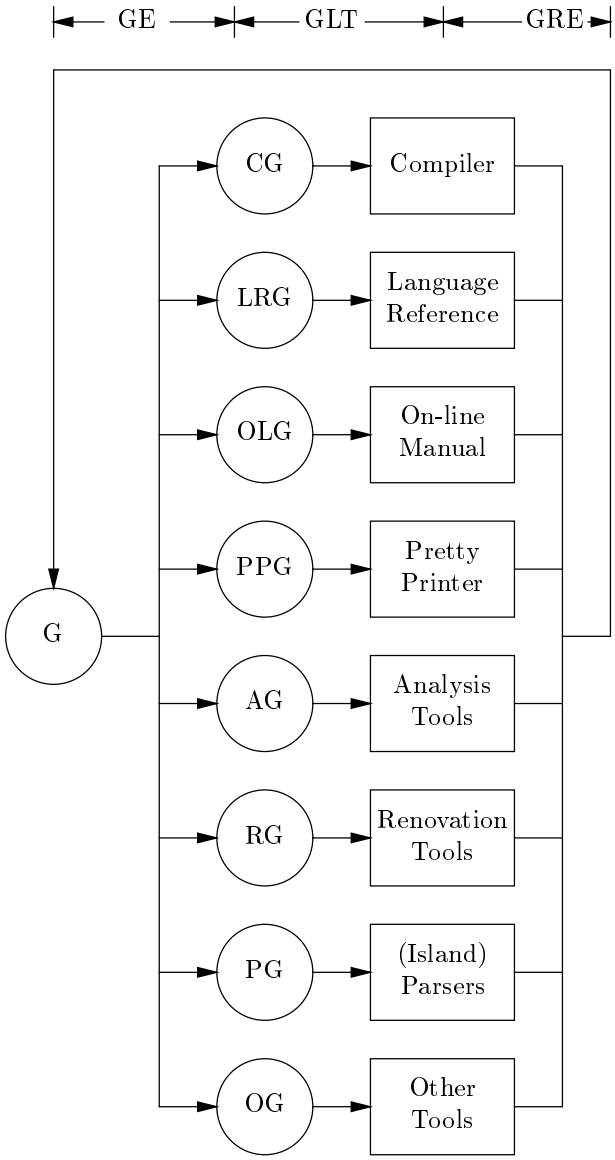


Figure 1: The Grammar Life-Cycle.

GRE is Grammar Reverse Engineering.

The arrows above the figure are indicating the range of the technologies. Of course, their borders are more diffuse than the picture suggests.

Figure 1 visualizes the grammar life-cycle. To illustrate the grammar life-cycle, we discuss a few examples.

A very important path in Figure 1 is one that helps in the the construction of software renovation factories. In the paper [104] we extracted 20 grammars from

the source code of proprietary compilers in order to generate hypertext manuals and large parts of a software renovation factory. This fits the grammar life-cycle as follows: starting in the compiler box, we recovered 20 compiler grammars, we turned them into one large grammar, and from that one we constructed an on-line manual so that we and others could understand the languages. Then we used the recovered grammar to generate a software renovation grammar from which in turn large parts of a software renovation factory were generated [104].

Let us consider another path in Figure 1. As we will see later on, in our recovery process the grammars can have various levels of completeness. Initially, a recovered grammar is not at all complete, but is already suited to be used in implementing island grammars suited for system understanding tasks. We have published a recovered but incomplete PL/I grammar [73] that has been used for implementing an island parser for PL/I. Also this fits in Figure 1: we extracted from an on-line manual a rough grammar of PL/I. Parts of it were used to implement an island parser for system understanding purposes. Also the VS COBOL II grammar that we published [74] has been used for the implementation of island parsers. Also this process can be visualized by walking along some of the arrows in Figure 1.

Yet another path in Figure 1 should be illustrated. In the paper [9] it is explained that from a certain kind of pretty printer a parser is derived. This fits the grammar life-cycle as well: from a pretty printer tool that accumulates grammar knowledge, it is possible to derive a grammar that is capable of parsing the code that could be pretty printed with the formatter. In Figure 1 this is represented by walking from the pretty printer box, via the grammar G to the parser grammar, and end in the (island) parser box.

We believe that Figure 1 illustrates how a rather optimal situation could look like when language oriented tools are integrated from early development to extensive renovation. Then the relations between the grammars and the tools that are built with them are indeed connected and the related grammars can be transformed to each other using grammar engineering tools. Integration of renovation tools with development tools is not present. In this paper we focus on how to move towards more integration. This is the subject of the next section.

3 Recovery of IBM's VS COBOL II Grammar

Anyone who is involved in software renovation knows that no matter how many languages you can handle with your tools, there is always another language or dialect that is not yet covered by the tool set you developed. This often occurs since people base their grammar development on the codebase they obtain from their customers. As soon as a new customer comes in, different constructs are being used. Of course, when we need to renovate IBM's VS COBOL II sources this would never be a reason for IBM to provide us with the source code of their compiler. There are obvious competitive reasons for that. Moreover, it is not at all clear whether the grammar of VS COBOL II is easily extractable from their compiler. So how to solve this issue? One solution is to write one by hand. We have done that [20], and others have, too. Manual approaches have several drawbacks. First of all, we recall that Vadim Maslov of Siber Systems estimates the effort for COBOL on 2–3 years. Second, it is very hard to reason about the correctness of the process and ensuing grammar when working entirely by

hand. We will follow a different approach than writing the grammar by hand. Using a semi-automated approach, we derive the VS COBOL II grammar in a traceable and cost-effective way. We extend on our earlier work published elsewhere [101, 104, 105].

3.1 Overview

We retrieved an on-line manual for VS COBOL II from the IBM BookManager BookServer Library [54]. Then with a simple lexical script, we extracted the syntax diagrams, also known as railroad diagrams. We wrote a parser that is able to parse the syntax diagrams [75]. We transformed the parsed diagrams into (extended) Backus Naur Forms (BNF) [4]. Then we assessed the resulting BNF code for connectedness (as proposed in [101, 104, 105]). We found more errors than one would expect from a language reference manual. However most of the errors were of a type that is fairly easily repairable. Namely, the majority of the problems did not reside in the concrete syntax of the VS COBOL II dialect, but more in how those constructs are combined into a grammar. So, for example, there was no error in what the concrete syntax of an `IF` statement looks like, but the errors resided in "Where is an `IF` statement allowed?", that is, the `IF` statement was not "connected" resp. used in the diagrams. The latter problems are solved much more easily than figuring out all the possible variants of the concrete syntax of `IF` statements.

We were able to write formal transformations that corrected such so-called connectivity errors fairly easily. The formal transformations are a means to record the modifications to the grammar and to reason about correctness of the ensuing grammar rules. This is convenient, since many errors and problems are reiterated in IBM manuals, because many syntax diagrams are copied from earlier versions and pasted in the newer manuals. In this way we could reuse the sometimes highly specialized transformations fruitfully.

Within a few days, the VS COBOL II grammar was entirely connected. This means, that all sort names that were used, were also defined and vice versa. With a few exceptions: the start symbol of the grammar is only defined and not used. The other exception being that the sorts that were used but not defined were all lexical entities. There were in the end 17 lexical entities, and by reading the manual we were quickly able to define them by hand. Of course, we had domain knowledge on COBOL grammars so uninitiated implementors may need more time for defining the lexicals and connecting the grammar. In the next phase we took actual VS COBOL II code, and used the extracted BNF to generate a parser solely intended for error detection. We started to parse code to see whether the grammar was correct. Neither an efficient parser nor a non-ambiguous grammar is needed in that phase. The first few days there were still a lot of problems. We could not parse more than 5 programs a day. This was due to the fact that a lot of the structural information was not in the syntax diagrams and had to be recovered from the text. In other cases the problems were that the structural information was present but erroneous. After about 2–3 days, it was possible to parse about 20 programs a day. The problems we obtained in that phase were due to typing errors in the syntax diagrams, and other small issues. After a few days of dealing with these issues, the amount of work to be done in order to parse programs decreased significantly, and we parsed entire systems in half a day. This half day was not due to the

repair of many errors, but more due to the fact that our Prolog-based parser used for error detection parsed just a few lines of code per second. After a week we parsed about 500.000 lines of VS COBOL II code. In this phase, we encountered language constructs that were accepted by the compiler but not documented (cf. the already mentioned `GREATER THEN` issue). But most of the time we encountered issues that were explained outside the scope of the syntax diagrams. Often, the syntax diagrams were too restrictive and were relaxed in the accompanying text. In one case we decided to extent the notation so that arbitrary ordering of choices (so-called permutation phrases [27]) could be expressed in both the BNF and the syntax diagrams, but in all other cases there was no real reason why the syntax diagrams were too restrictive. We located such problems by parsing code. We could have been able by inspection of the grammar plus our domain knowledge to find a number of errors, but the parsing approach is more systematic and self-checking.

Noteworthy, perhaps is that the more systems we parsed, the less work we had to do on correcting errors. To the uninitiated reader this might seem obvious. However, in the everyday practice of software renovation this is unusual. For instance, the phrase *grammar reengineering* was used in [21] to denote the fact that reengineering companies often have a maintenance problem with their grammars, and that this can become so severe that the grammars need actual reengineering themselves. As one CEO put it: "we are driving in a very fast sports car but we are about to hit the wall at any time now" to express his concerns about the maintainability of his parsers. Tom McCabe who chaired the Reengineering Week 2000 in Zurich clearly agreed with this CEO: he stated that the parser construction and maintenance problem was the number one technical problem for his company. The superiority of our approach largely arises from one particular property, that is, the completion and correction of the grammar specification is not tangled with the implementation of an efficient parser.

Knowing the "grammar molasses phenomenon", we were amazed that the amount of effort decreased so rapidly while the amount of code that we could parse increased. After a week, we obtained hardly any errors anymore. Then we decided to ask our colleagues to send us as much COBOL code as possible. The systems that we obtained were from various companies and various countries. They revealed additional errors in the manual. So in this case study we cannot claim formal correctness of the recovered grammar, but we can claim that the amount of work on the grammar is sharply decreasing whereas this normally drastically increases.

We proceed to present more details on our VS COBOL-II case study so that others can apply the proposed technology as well. Occasionally, we also refer to other grammars we recovered if that illustrates our points more clearly.

3.2 Parsing the Diagrams

Let us consider one of the diagrams of the VS COBOL II manual. It is the `SEARCH` statement. In Figure 2 we copied the diagram verbatim into the paper. It is an interesting exercise to write a parser for such diagrams. We will not explain how to do this in the deepest detail, but we highlight some issues to give the reader an idea. One peculiarity is that there is no layout, that is, even in the parser phase soft spaces and line breaks are significant. The token classes are uncommon, like all kinds of vertical lines and ASCII codes and connectors. The

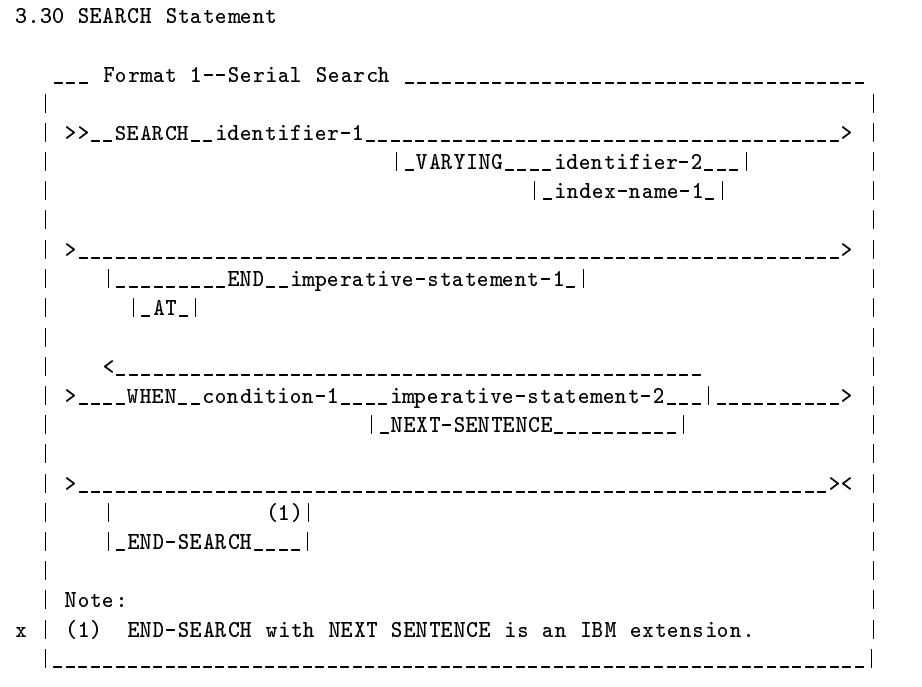


Figure 2: The original syntax diagram for the SEARCH statement.

position information of the tokens is significant, whereas for many languages the row and column information has no special semantics. Finally, we do not build an AST in the usual compositional manner. For example, look at the <_____ above the WHEN in Figure 2. Recall that this notation denotes an iterative construct. We push such a token on the stack and keep parsing more tokens and lines. We use the position information to derive whether the stacked iterative construct can be popped from the stack to complete a phrase in a subsequent line. For detailed information on how to parse visual languages such as these syntax diagrams we refer the reader to a paper focusing on that issue entirely [75].

Of course during the parsing phase several syntax errors in the syntax diagrams were revealed. For example, there was one loose colon, four cases of a wrong white space character, and in the ALTER statement the header line of the syntax diagram was missing. We repaired those trivial errors by hand until the entire set of syntax diagrams passed our diagram parser.

3.3 Deriving the Raw Grammar

Once we can parse the syntax diagrams, it is possible to derive a BNF. This is possible since syntax diagrams just provide a visual notation for BNF-like syntax definition constructs. For instance, the converted syntax diagram of Figure 2 looks like this in (extended) BNF:

```
search-statement =
```

```
"SEARCH" identifier ["VARYING" (identifier | index-name)]
[["AT"] "END" imperative-statement]
{"WHEN" condition (imperative-statement | "NEXT-SENTENCE")}+
["END-SEARCH"]
```

The resulting grammar is called a *level 1* grammar. It is nothing more than the raw extracted grammar but void of typo's and in textual BNF. We recall that we can already use level 1 grammars to implement system understanding tools. See [73, 72] for level 1 grammars for Pl/I and COBOL 2000. At the time of writing this paper, an employee of a software renovation company is implementing island parsers using our recovered grammars.

We should mention another problem related to grammar extraction as discussed above. In deriving BNF rules from the diagrams, each diagram has to be associated with the sort which is intended to be defined by the diagram. A heuristic that we used was the default rule to use the name of the section header as a sort name. In the the syntax diagram depicted in Figure 2, we recall that the header is: `3.30 SEARCH Statement`. We used this information and created the sort name `search-statement` in the above BNF version of the syntax diagram. Of course this heuristic was not always applicable due to sort confusions and irregularities in the section headers, so additionally we defined a mapping to override section headers which do not induce useful sort names according to the heuristic. We mention for instance that the syntax diagram presenting the `EXIT PROGRAM` syntax, does not start with a sort name, but with the keyword `EXIT PROGRAM` itself. Since a keyword is not a sort name, we constructed an overriding transformation solving this issue, and we introduced a sort name (in this case `exit-program-statement`). Overriding was necessary for just 15 sections, i.e., the original section headers were mostly useful. This is not always the case. In recovering the PL/I grammar [73], we found that the section headers in IBM's PL/I language reference manual [53] were mostly not useful.

3.4 Aiming at Connectivity

As soon as we have the level 1 grammar we start to assess the quality of the grammar rules. Two important quality indicators are: the sorts that are used but not defined—bottom sorts—and the sorts that are defined but never used—top sorts. These indicators have been identified in [101, 104, 105]. In the ideal situation, there are only a few top sorts, preferably one corresponding to the start symbol of the grammar, and the bottom sorts are exactly the sorts that need to be defined lexically. In the case of the VS COBOL II manual we found 168 sorts in total of which 53 bottom sorts and 73 top sorts (see Table 1). From the data, we could immediately conclude that the extracted grammar was rather unconnected and incomplete. This is not too much of a surprise, since the diagrams were not containing consistent sort names of the constructs that were being described. Several connections in the sense of syntactical chain productions were not made explicit in the diagrams. Also, certain constructs such as arithmetic expressions lacked a syntax diagram in the IBM manual. Using about 70 rather simple transformation steps to solve all these connectivity resp. obvious incompleteness problems, we could easily reduce the number of

Category	level 1	level 2	level 3	level 4
Rules	166	202	202	234
Sorts	168	176	176	205
Top sorts	73	2	2	2
Bottom sorts	53	17	-	-
Lexical sorts	-	-	17	17
Keywords	337	364	364	363

Table 1: Levels of grammar recovery for VS COBOL II.

top sorts to 2 and the bottoms sorts to 17. The remaining bottom sorts were identified as lexical sorts. We call a level 1 grammar that is maximally connected a *level 2* grammar. It does neither mean that a parser generated from the grammar recognizes real programs written in the language at hand, nor that a parser can be generated at all.

We summarized the differences between the various levels of our VS COBOL II grammar in Table 1. We note that in the raw extracted grammar we had 166 grammar rules and 168 sorts but after our special-purpose transformations to achieve connectivity we had 202 rules and 176 sorts. The increase in rules and sorts is mostly related to trivial chain productions and unfolding steps used to massage the grammar. As we mentioned already, some aspects of the grammar were not expressed in syntax diagrams for no good reason, e.g., arithmetic expressions or figurative constants. To complete the grammar, the corresponding informal explanations of the corresponding phrases had to be incorporated into the grammar via suitable transformations. That also explains the increased number of keywords in Table 1 (we come back on this table later on).

3.5 Building the Lexical Syntax

In this phase we have a complete BNF in the sense that used sorts are defined and vice versa. Except the sorts that are supposed to be top sorts, and sorts that are probably lexical. We illustrate the effort it takes to build the lexicals by providing an example.

The following chain production rule for the sort `index-name` in the `SEARCH` statement (see Figure 2) was added by us in a special-purpose transformation in order to improve connectivity. We expressed the natural language in the manual by the following chain production:

```
index-name = alphabetic-user-defined-word
```

The sort name `alphabetic-user-defined-word` has a lexical definition to be added by hand, since it was not defined in any syntax diagram in the manual (but it was described in natural language). It was not a problem to define all the 17 lexical sorts by hand. For example, our definition for `alphabetic-user-defined-word` is as follows:

```
alphabetic-user-defined-word =
  ([0-9]+ [\-]*)* [0-9]* [A-Za-z]
  [A-Za-z0-9]* ([\-]+ [A-Za-z0-9]+)*
```

The above notation is similar to the language of regular expressions as used for Lex [77]. We call a level 2 grammar that also has its lexicals recovered a *level 3* grammar. In other words a level 3 grammar has no bottom sorts and only valid top sorts (see Table 1).

3.6 Generating a Parser

A level 3 grammar can be used as input to generate a lexer and a parser. Since a language reference manual grammar is not geared towards any parser generation technology, the generated parser will in general be ambiguous. We deal with this problem in Section 4. But note that in some reference manuals *two* grammars reside: one for human understanding and one that can be fed to a parser generator (this is the case for Java [43]). For now, we are only interested in generating a parser to parse VS COBOL II code with the only goal to test and correct the level 3 grammar. The parser that we use, just makes an arbitrary choice when an ambiguity is found. So as soon as *some* parse is possible we consider the program parsed successfully for now. We used top-down parsing with backtracking as supported by, e.g., LDL [49] or PRECC [25].

By using this stepwise approach we are able to solve the problems one at a time: first getting the specification as correct as possible, and then we can deal with ambiguities. In principle, ambiguities can cause some errors to go unnoticed because parts of the input might be parsed in an unintended manner still leading to a complete parse tree. However, even with the ambiguous parser, we reveal so many problems that this minor issue is not relevant in this phase. For, we are going to disambiguate the grammar in a next phase (see Section 4) and then the remaining errors that we could have missed, will pop up anyhow.

Recall from Figure 1 that we are moving from the arrow of the on-line manuals, via the generic grammar G, to a grammar that is suited to be fed to a parser. At this point we are combining the area of grammar engineering with the generic language technology field, with as application parser generation. In order to parse real COBOL code, we reused a preprocessor that prepares the programs for a parser. This preprocessor has been discussed elsewhere [20]. After preprocessing, the code is amendable to parsing.

3.7 Detecting Errors

With the working parser, we were able to detect many semantic errors in the level 3 grammar. Let us give an example.

```
SEARCH WRONG-ELEM   END
        SET I1 TO 76
     WHEN ERROR-CODE (I1) = A-WRONG-VWR
        NEXT SENTENCE.
```

This code is obviously correct in the sense that the VS COBOL II compiler accepts the code. However from the manual we extracted something different. If you look closely to the syntax diagram that we displayed earlier in Figure 2, you will see that the exact keyword used is `NEXT-SENTENCE`. But in the footnote

of the syntax diagram, there is no a dash in the keyword (which is the correct usage). This error was not found in earlier phases. One reason is that it is legal to have dash symbols in COBOL keywords, viz. the `END-SEARCH` keyword. To solve this *semantic* error in the syntax diagram, we constructed a special-purpose transformation solving exactly this issue. Of course, this kind of error seems trivial, but it is only detected using a serious test: namely by actually parsing a lot of code. Note that the ambiguity of the generated parser is not at all an impediment in finding (most) errors, which is our present goal.

Let us have a look at another error. Here is a piece of code that is accepted by the VS COBOL II compiler but that did not parse with our parser generated from the recovered grammar:

```
SEARCH TRANS-MESSAGE-SWITCH-VALUES
          END
             CALL 'PROG343'   USING CLEANUP
             CALL 'PROG445' USING ABT-ERROR-ABEND-CODE
          WHEN TRANS-PGM-ID(TRNSIDX) = NEXT-PROGRAM-NAME-ID
             MOVE 'M' TO LINKAGE-WRITE-XFER-INDIC
             PERFORM C-400-TERMIO-XFER-MSG-SWITCH
             MOVE 'N' TO ABT-IN-PROGRESS
             GO MAIN-LOOP.
```

The parse problem was that the original syntax diagram (see Figure 2) stated that after the `END` the sort `imperative-statement-1` is expected. This sort allows only for a single statement. But in the code that is accepted by the VS COBOL II compiler there are two `CALL` statements. In the VS COBOL II Reference Summary [56] this confusion is not resolved. However in the application programming guide [55]—it is stated that

> A series of imperative statements can be specified whenever an imperative statement is allowed.

So the actual BNF rule for the `SEARCH` statement could be relaxed. We used a grammar transformation to do this. The result of the transformation is depicted below.

```
search-statement =
   "SEARCH" identifier ["VARYING" (identifier | index-name)]
   [["AT"] "END" statement-list]
   {"WHEN" condition (statement-list | "NEXT" "SENTENCE")}+
   ["END-SEARCH"]
```

Of course, a definition for `statement-list` had to be provided. We took care of that in another transformation. As can be imagined, the process of parsing a few million of lines of VS COBOL II code revealed much more semantical errors in the grammar description. We totaled a number of about 200 transformation steps that were necessary to correct the grammar. This means that we could parse all our VS COBOL II code without errors. After we successfully parsed about 2 million lines of code that originated from various companies, from various countries, and from various continents, we called the thusly ensued grammar

a *level 4* grammar. Recall, that we now have a recovered grammar specification, but that the specification itself still is ambiguous.

Also, note that we can never state the complete correctness of a grammar that is obtained in this way. The kind of restricted correctness we can claim is the following. The generated parser parses *all* the available code that is also parsed by the compiler. From a different perspective pursued for example in the logic programming community [41, 26], the statement that a parser parses a given test set successfully can be also regarded as a kind of completeness statement. Correctness might be then conceived as the property that the underlying grammar is fully covered by the test set according to a suitable coverage criterion such as rule coverage [89]. Thereby, correctness means that the parser does not accept more programs than intended based on a coverage argument.

Level 5 grammars We speak of a *level 5* grammar when we directly extract the grammar from the compiler source code itself. This usually implies that *all* the code that is accepted by the compiler is also parsed by other parsers derived from the extracted grammar. However, due to idiosyncrasies of the target platform or the derivation process, the derived parsers might deviate from the reference compiler.

We delivered a level 5 grammar to Ericsson [104]. From the millions of lines of code that we obtained from Ericsson we could parse everything except 3 files: two files were from the compiler test bank and not supposed to parse, but one file contained the single-character keyword `T` that we used in a transformed renovation grammar for another purpose.

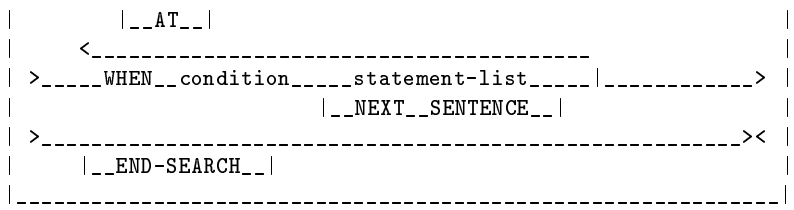
We believe that in future programming environment implementations it is possible to work with renovation grammars of level 5: then the entire development of such environments is designed from day one in such a way that renovation of the developed code is enabled. An early example of this trend is the possibility of several new development environments in which it is possible to tap the exact syntax tree from the compiler as we indicated in the introduction. The work that we present in this paper will then not be outdated: on the contrary, for, then we need grammar engineering and grammar transformations from day one, but since it is used during the development phase and not as an after thought we do not need to solve a number of the typical renovation problems that we now face.

3.8 Generating Correct Manuals

We regenerated improved syntax diagrams from the corrected BNF descriptions. They are improved in the sense that they better reflect what the actual language comprises. Here is the recovered syntax diagram for the `SEARCH` statement:

```
search-statement

 ------------------------------------------------------------
|                                                            |
| >>--SEARCH--identifier--------------------------------------> |
|                          |--VARYING-----identifier-----|     |
|                                      |--index-name--|        |
| >----------------------------------------------------------> |
|     |------------END--statement-list--|                     |
```



As can be observed, the footnote that was originally in the syntax diagram is no longer there (viz. Figure 2). Since our primary aim in this paper is to produce a correct and complete grammar specification, we did not bother to implement a diagram parser that handles the comments in the diagrams as well. This is however not at all a limitation of our approach, and is in fact easy to accomplish.

Standards Although in this paper the focus is on recovery and our intentions are to use the recovered grammars for software renovation factories, there are many more applications of our work possible. Several standardization people are interested in using our technology to produce *new* standards for languages. This effort fits the grammar life-cycle depicted in Figure 1 as well. Indeed, it is a very good idea to use techniques similar to ours to deal in an accurate manner with language standards. At this moment the state-of-the-art in language reference standards is to use just word processors like Microsoft Word and no more sophisticated tools. We have experience with the recovery of grammars from language standards as well, and these documents contained a lot of errors and confusing issues. Those problems are all due to the lack of grammar tool support.

Not only ANSI, ISO, or ITU standards could benefit from grammar engineering, but also company standards. At the time of writing this paper we obtained a request to cooperate with a company on the production of a company language reference manual for the IBM compiler product *COBOL for OS/390 & VM* [57]. The idea is to come up with a browsable grammar plus natural language where the programmers are informed about the subset of the language they should use and what to avoid. Many more features like clear separations of what is ANSI supported and what is an extension in which compiler product are on the list of this company.

IBM Manuals Of course, also IBM could benefit from our work, since obviously our published grammar [74] is at the moment the most authoritative source for what syntax is accepted by their VS COBOL II product. It will be clear that our work can be repeated for all their other language reference manuals.

3.9 Generating On-line Manuals

It is of course a simple task to also generate web-enabled versions of the syntax diagrams and BNF for the recovered VS COBOL II grammar. Web-enabled BNF was also created in the study [104] for a proprietary language of Ericsson. The hypertext version of our recovered VS COBOL II grammar is made available to the general public [74]. We recall that we were the first to make such

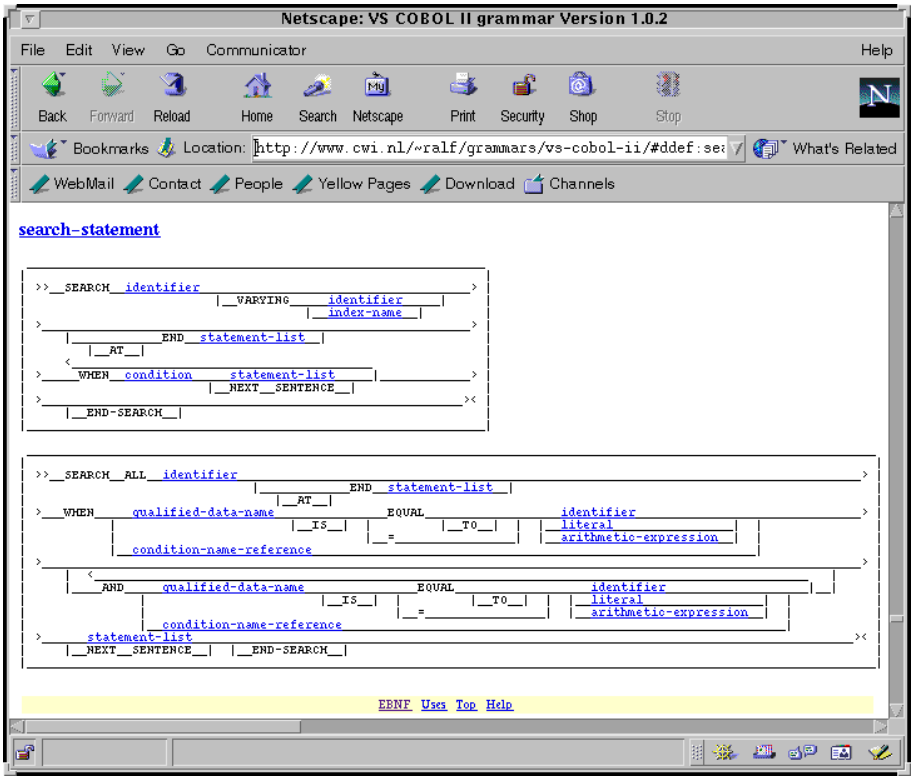


Figure 3: A Dump of the actual HTML page containing the recovered VS COBOL II grammar specification.

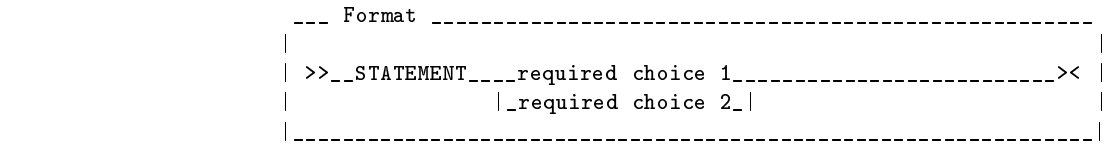
information publically available. Within an hour after its publication, the URL was in the frequently asked questions of the Usenet news group `comp.compilers` answering the frequent question where a COBOL grammar could be obtained. In fact, the recovery process is so easy that the effort to come up with a corrected version can hardly be interpreted as a significant investment in time and effort. We think that this is good news: our work discloses important information for the software renovation area, but also for both GNU COBOL compiler initiatives [92].

We provide a screen dump of the actual HTML page that we generated with our tools. See Figure 3 for an impression. We showed the `SEARCH` statement again. The page contains many more features than just the diagrams. The page starts with a summary comparable to Table 1. Furthermore, context-free syntax in BNF and lexical syntax is present. There are many search possibilities: we provide indexes and lists of all sorts, top sorts, bottom sorts, and all keywords. All definition-use relations are cross linked. We also provide the diagrams as is shown in the figure. Moreover, all cross-links that one could wish for are present so that navigating in the document is very convenient. Switching between the BNF and the diagrams is enabled. The generation tools are generic: any context-free grammar can be turned into a browsable version with the tools.

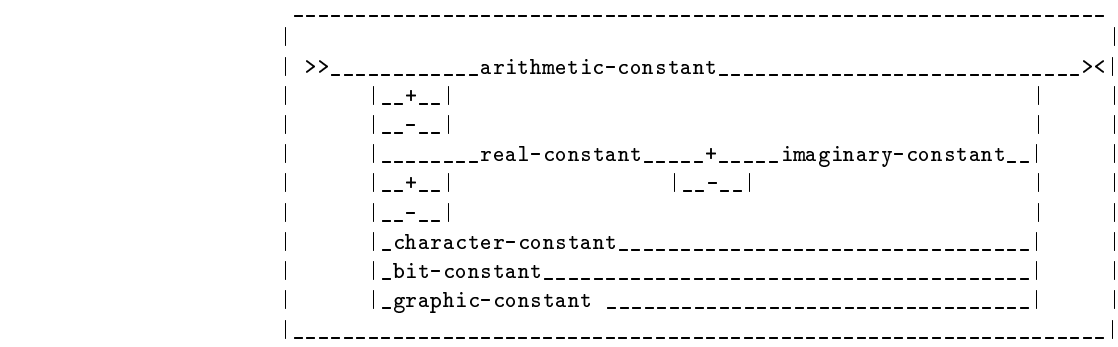
3.10 Summary of the Kinds of Errors

We collected examples of the most prominent types of errors. In this paragraph we summarize them for convenience's sake.

- Syntactic typographical errors in syntax diagrams. This kind of problem enforces parsing errors while processing the diagrams. Think of erroneous spaces, loose colons, and so on.
- Semantic typographical errors in syntax diagrams. Think of the `NEXT-SENTENCE` error. A diagram containing that kind of typo is still a syntactically correct syntax diagram. So the diagram parser does not report an error. Only when we parse real code with the parser generated from the recovered grammar, we will detect that the syntax diagrams parse another language than is intended, hence a semantic error.
- Another type of error is that the syntax diagrams have undocumented syntax themselves. This is quite common for syntax description languages. Let us mention an example of an undocumented feature of the syntax diagram language as found in the IBM's PL/I language reference manual [53]. The example is concerned with stacking alternatives. Stacked alternatives are explained in the manual to be aligned. The manual provides the following illustration.



Only aligned stacking was explained, whereas we found unaligned stacks in the manual that we could not parse at first. Here's an example:



Consequently, we had to relax the syntax diagram parser. As can be imagined, there are a few more of these undocumented uses of the syntax diagrams. A related problem is that the various language references use quite different dialects of the syntax diagram notation. Even worse, one document might resort to different contradictory styles, refer, for example, to §13.1.7 in [53] which—besides the fact that it contains several syntax errors—uses a style entirely different from the rest of the document.

- Incompleteness of the grammar specification as discussed already in Section 3.4. The fact that a grammar is not complete resp. connected can be assessed to a certain extent by measuring top and bottom sorts.
- Confusion about sort names. For example, the sort name `condition-name` was used as a section header in a syntax diagram in the VS COBOL II manual, but the diagram actually defined *referencing* condition names. Condition names themselves are just defined by lexical syntax, whereas referencing them might involve qualification. This kind of confusion of pure condition names and references happens more often in the VS COBOL II manual. Sometimes the enclosed text explains that a certain occurrence of `condition name` in a diagram can be qualified.
- Conflicting uses of sort names. The sort `operand` was used in the manual in two places, but for different purposes. It was used in the `COPY` Statement meaning `quoted-pseudo-text`, `identifier`, `literal` or `cobol-word`. The sort `operand` was also used in conditional expressions. There it meant an `arithmetic-expression`. We repaired this error using a grammar transformation by renaming `operand` so that different sort names are used in the different contexts. This solves the homonym problem.
- Connectivity mismatches like the definition of the content of the `PROCEDURE DIVISION`. In the original IBM manual, the content definition actually attempts to define the structure of the entire `PROCEDURE DIVISION`. In particularly, the definition is started with the keywords `PROCEDURE DIVISION`. The diagram for entire programs, however, really refers to just the *content* of the `PROCEDURE DIVISION`.

As can be seen, we started in Figure 1 with an on-line manual, we removed all kinds of errors as summarized above, we were able to generate a parser for it (albeit an ambiguous one), and additionally we were able to generate corrected manuals which closed the loop. As was stated earlier, in our opinion the generation of correct manuals is a by-product, although it is important to have access to correct and completely connected grammars at all times. We experienced that it is very useful during renovation tasks to have access to accurate web-enabled grammar knowledge. Now that we have solved myriads of plain errors in the specification, we are confident that we have a firmly established specification that we can further refine, which is the subject of the next section.

4 Implementation of the Grammar Specification

At this point we have recovered the grammar specification of a language from its manual. Of course, the grammar specification was not geared towards being used by some parser generator tool, but optimized towards human comprehension. In practice this implies that some production rules are ambiguous. Therefore we worked until now with a parser tool that handled ambiguities by making some arbitrary choice. This was not the ultimate goal of our effort. Therefore, we now take the work one step further: we start to turn the grammar specification into a realistic parser specification. The hard part of this process is to disambiguate the grammar. In this phase of the process we take the recovered level 4 grammar specification as input and we proceed with our systematic process.

We start with choosing another parser generator in Section 4.1 more suited for realistic parsing. After all, we are now no longer interested in finding errors, but in efficient parsing and generating ASTs. Of course, we also need to implement the lexicals as addressed in Section 4.2. Then we can start with the elimination of ambiguities. Ambiguities are illustrated and disambiguation by special-purpose transformations is discussed extensively in Section 4.3. Some ambiguities are better eliminated using priorities as briefly mentioned in Section 4.4. Finally, Section 4.5 gives pointers how to convert from the parser technology that we use to mainstream parser technology like Lex and Yacc.

4.1 From BNF to SDF

The first step is to convert the recovered BNF into the correct dialect so that a new parser generator accepts the input. We chose to use a sophisticated parser generator that can handle arbitrary context-free languages. It is a scannerless generalized LR (SGLR) parser generator [76, 110, 95, 114, 116]. The technology is suited for reporting ambiguities, which is what we need to solve in this phase. Also here, we take one step at a time: we do not convert the specification directly to more mainstream parser generators like Yacc. Because then we would have to solve more problems simultaneously: not only ambiguities but also the idiosyncratic problems dealing with shift/reduce and reduce/reduce conflicts (see [78] for details on these types of problems).

The chosen parser generator accepts grammar rules in so-called SDF format (Syntax Definition Formalism [50, 114]). So with some grammar transformation we turn the BNF specification first into SDF. Below we illustrate the transformation by giving the `SEARCH` statement in SDF format:

```
"SEARCH" Identifier ("VARYING" Identifier | Index-name)?
("AT"? "END" Statement-list)?
("WHEN" Condition Statement-list | ("NEXT" "SENTENCE"))+
"END-SEARCH"?
    -> Search-statement
```

Of course, this particular piece of code is nothing more than different syntax than we had before. This syntax swap is not the real problem. The hard problem is the disambiguation addressed in Section 4.3.

4.2 Implementing the Lexical definition

There is another very simple step involved in the implementation of the grammar specification, that is to say the implementation of the lexical part of the grammar. Recall that the lexicals were defined while completing the grammar in order to obtain a level 3 grammar without bottom sorts. We can reuse the lexical definition derived in the process of completing the grammar more or less directly for SDF. Since we are dealing with implementation and not specification, we have to deal with the idiosyncrasies of the used platform. Thus, some small additional effort is required.

Let us consider one of the 17 lexical definitions and its definition in SDF. We have the following definition for a `cobol-word` in the recovered grammar specification:

```
cobol-word = [A-Za-z0-9]+ ([\-]+ [A-Za-z0-9]+)*
```

In natural language this expresses that a `cobol-word` is an alphanumeric string that should neither start nor end with a dash (-). We can just reuse the above specification in the SDF implementation:

```
[A-Za-z0-9]+ ([\-]+ [A-Za-z0-9]+)* -> Cobol-word
```

Indeed, we need to deal with an idiosyncrasy of SDF, that is, there is no default longest-match heuristic. So we have to add a so-called follow-restriction simulating longest match of lexicals:

```
lexical restrictions
 Cobol-word                    -/- [A-Za-z0-9\-]
```

The listed characters at the right-hand side should not follow after a `Cobol-word`. Since the parser generator supporting SDF is scannerless [114], without this kind of restrictions there are at best a lot of local ambiguities and in the worst case some non-local ones.

Another lexical issue has to be taken care of as well. In a language usually certain tokens are reserved words. Of course, these things can be summarized in a simple one-liner when we are dealing with a language reference manual. Actually, we abstracted from that issue in the grammar specification. However in a realistic implementation of a parser, these things should be made explicit in code. In yet another grammar transformation we automatically generate the grammar rules that take care of reserved words. For instance, we cannot use the keyword `PROGRAM` in a COBOL program as a `Cobol-word` although it is not (yet) forbidden by the recovered grammar specification. Of course, it should be rejected by the parser. The same holds for all the 300+ keywords of the VS COBOL II grammar. We listed two such rules in SDF format to give the reader an idea of the complexity of the transformation (which is very low):

```
"PROGRAM" -> Alphabetic-user-defined-word {reject}
"PROGRAM" -> Cobol-word {reject}
```

These rules indicate that the keyword `PROGRAM` is rejected to occur as in case of the above two sorts.

4.3 Disambiguating the Grammar

Now that we have settled everything so that we can generate an SGLR parser for the recovered grammar, the disambiguation can start. In the first program we immediately found an ambiguity. We abstracted from the actual variables in the following code fragment:

```
MOVE A IN B TO C IN D.
```

The browsable grammar is very helpful to locate what is going on, namely the fact that with navigation we immediately reveal the problem. The syntax of the `MOVE` statement looks like this:

```
move-statement

 ------------------------------------------------
|                                                |
|                                <_____________  |
| >>__MOVE_____identifier_____TO____identifier__|__>< |
|           |__literal_____|                     |
|________________________________________________|
```

When we click on the `identifier` we see that this actually starts with a `qualified-data-name` as is indicated below:

```
identifier

 ------------------------------------------------------------------
|                                                                  |
| >>__qualified-data-name_____________________________________> |
|                          |  <_________________   |             |
|                          |____(__subscript__)__|__|             |
| >_____________________________________________________________><  |
|     |__(__leftmost-character-position__:_____________)__|      |
|                                        |__length__|              |
|__________________________________________________________________|
```

In the definition of the `qualified-data-name` which is below we can now see the ambiguity:

```
qualified-data-name

 ------------------------------------------------------------------
|                                                                  |
| >>__data-name_____________________________________________>    |
| >_____________________________________________________________><  |
|     |  <_____________________   |  |_____IN_____file-name__|     |
|     |_______IN_____data-name__|__|     |__OF__|                  |
|          |__OF__|                                                |
|__________________________________________________________________|
```

What happens is that when the keyword `IN` (appearing in the code fragment `A IN B`) is detected after the `data-name A`, the grammar rule can be read in two ways. Either `B` is also a `data-name` or `B` is a `file-name`. We then check what the definitions of those two are and they happen to coincide: both are an `alphabetic-user-defined-word`. As can be seen, the manual contains some precision that is geared towards human comprehension: either use the name of some piece of data or the name of a file. Of course, for a parser this makes no difference because they are both strings of the same type. We eliminate the ambiguity with a grammar transformation that unifies the `data-name` and

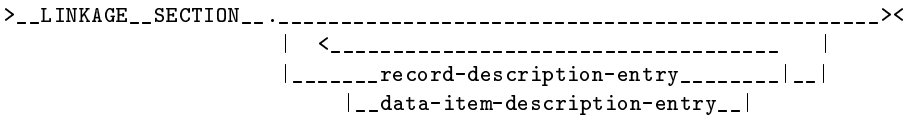
file-name in the definition of qualified-data-name as far as qualifiers are concerned. Since both sorts are just alphabetic-user-defined-words this transformation does not modify the semantics of the grammar specification, but it removes the ambiguity in the generated parser. As a result we modified the grammar and the following SDF fragment shows the result of the transformation:

```
Data-name ("IN" | "OF" Data-or-file-name)* -> Qualified-data-name
Alphabetic-user-defined-word -> Data-or-file-name
```

Let us give another example to show that ambiguities can be traced systematically and they can be resolved easily. In the DATA DIVISION we encountered an ambiguity related to the distinction of records and data items. The problem is that COBOL's records and data items cannot be separated in a truly context-free manner, but the grammar specification attempts it. The following piece of code from the LINKAGE SECTION of a program illustrates the problem:

```
01  BA-LH-006-IF.
    04  BA-LH-006-IF-FIELDS.
        06  BAC006-STATUS.
            08  COMPONENT PIC X(2).
            08  COP-COL-NUMBER PIC X(4).
            08  LOCAL-STATUS-CODE PIC 9(2).
```

In the example, records are started at levels 01, 04 and 06. The fields at level 08 are data items. To realize the scope of a record, level numbers need to be taken into consideration. This can hardly be done in BNF. As an aside, it is possible in principle because there only finitely many level numbers, but the resulting grammar would become extremely complex. In order to find the exact location of the ambiguity in the grammar we browsed through the grammar specification and we found in the data-division-content the following syntax diagram snippet:



By clicking on both sorts we inferred that they were mapped to the same sort data-description-entry. These chain productions were enforced by a special-purpose transformation in the phase of completing resp. connecting the grammar. It was suggested by a comment in the manual saying that data-description-entry is used to refer to both, record-description-entry and data-item-description-entry. The difference between these sorts was only explained informally for the abovementioned reasons. Consequently, our completion introduced an ambiguity. The solution is to eliminate both the sorts record-description-entry and data-item-description-entry and instead use the sort that they both happen to be: data-description-entry. After conversion we obtained the following SDF fragment:

```
("LINKAGE" "SECTION" "." Data-description-entry*)?
    -> Data-division-content
```

Again, this solves an ambiguity in the parser while the semantics of the grammar specification is invariant.

After having seen some of these ambiguities, we started to look in the grammar specification itself for similar overlapping definitions. Indeed the `CALL` statement has two locations that are overlapping. In one case it can end as follows. We elided the irrelevant parts for clarity:

```
call-statement
[..]
>_____________________________________________________________________
      |____________OVERFLOW__statement-list__|  |__END-CALL__|
         |__ON__|
```

Or it can end as follows:

```
call-statement
[..]
>_____________________________________________>
     |____________EXCEPTION__statement-list__|
        |__ON__|
[..]
>_____________________________________________________________><
  |__NOT________EXCEPTION__statement-list__|  |__END-CALL__|
        |__ON__|
```

Elisions are denoted by `[..]`. So when we parse a `CALL` statement and we have neither an `OVERFLOW` nor an `EXCEPTION` we can use both of the above production rules to parse this `CALL` statement which leads to an ambiguity. We solve the problem by factoring out the overflow and exception phrases which are only parts of the format to separate these formats. This leads to the following grammar transformation after conversion to SDF:

```
[..]
Call-rest-phrase "END-CALL"? -> Call-statement
Call-overflow-phrase -> Call-rest-phrase
Call-exception-phrase -> Call-rest-phrase
("ON"? "EXCEPTION" Statement-list)?
("NOT" "ON"? "EXCEPTION" Statement-list)?
    -> Call-exception-phrase
("ON"? "OVERFLOW" Statement-list)? -> Call-overflow-phrase
```

Both end parts of the `CALL` statement are now taken care of by one sort `Call-rest-phrase`. This sort is either a `Call-overflow-phrase` or a `Call-exception-phrase`. And they in turn define the concrete syntax of the `EXCEPTION` and `OVERFLOW` parts. We note that in this transformation process, we created two identical grammar rules for the `CALL` statement, and we used a grammar transformation

to remove the double rules. Otherwise we would have exchanged one ambiguity for another. The grammar transformations are preserving the semantics of the grammar specification while the ambiguity in the generated parser is gone.

We are not ready with the `CALL` statement. The following syntax diagram fragment (from the `CALL` statement) shows that there is another ambiguity:

```
      <-------------------------------------
-------------identifier---------------------|---
          |__ADDRESS__OF__identifier__|
          |__file-name_____________|
```

Since `identifier` and `file-name` can be derived to the same sort, the above stack of 3 possibilities leads to an ambiguity if the `ADDRESS OF` keyword is not present. We choose the pragmatic solution to reject the alternative for `file-name` with the intention that an `identifier` is interpreted as a `file-name` when necessary. We apply the following grammar transformation on the BNF for the `CALL` statement to solve this ambiguity and we turn the following grammar fragment:

```
(identifier | "ADDRESS" "OF" identifier | file-name)
```

into the grammar fragment below:

```
(identifier | "ADDRESS" "OF" identifier)
```

The `file-name` is gone but since this reduces to the same sort as `identifier` the grammar rule is equivalent to the original one. As an aside, we are aware of the fact that this solution is a bit, well, rude. There is another option. We could delay disambiguation and wait until we can use type checking to build the final parse tree, for example, in the style of so-called disambiguation filters [68]. This example illustrates that there is sometimes more than one option how to disambiguate. The advantage of separating grammar specification and grammar implementation is that we can maintain a clean specification which is not polluted by particular implementation choices such as various ways to resolve ambiguities.

We are not ready yet with the `CALL` statement. Lists of call-by-reference parameters for `CALL` statements can be obtained in two different ways. We depict a small BNF fragment of the `CALL` statement to illustrate the problem:

```
{([["BY"] "REFERENCE"] {(identifier | "ADDRESS" "OF" identifier )}+
  | ["BY"] "CONTENT" {(["LENGTH" "OF"] identifier |
  "ADDRESS" "OF" identifier | literal)}+ )}+
```

Either we can iterate over the parameter blocks (which is represented by the outer `{...}+`) or by iteration of parameters after the possibly empty `BY REFERENCE` phrase (which is the inner `{...}+`). We remove this second choice in order to disambiguate. As a consequence in our grammar, we enforce iteration over just parameters. Of course, we can still recognize the same set of programs with this change, only we remove yet another ambiguity. We now provide the final result of all the grammar transformations applied to the `CALL` statement and after conversion to SDF:

```
"CALL" Identifier | Literal ("USING"
      ((("BY"? "REFERENCE")? Identifier | ("ADDRESS" "OF" Identifier))
      | ("BY"? "CONTENT" ((("LENGTH" "OF")? Identifier)
      | ("ADDRESS" "OF" Identifier)
      | Literal)+))+)? Call-rest-phrase "END-CALL"? -> Call-statement
```

As a final example we take care of an ambiguity that is caused by some IBM extensions regarding optional section headers and paragraph names. We depict the grammar specification in BNF format below.

```
procedure-division =
   "PROCEDURE" "DIVISION" [ "USING" { data-name }+ ] "."
   [ "DECLARATIVES" "." { section-header "." use-statement
   "." paragraphs }+ "END" "DECLARATIVES" "." ] sections

procedure-division =
   "PROCEDURE" "DIVISION" [ "USING" { data-name }+ ] "."
   paragraphs
```

The first production rule fully covers the second one. This is due to the IBM extensions put on paragraphs and sections. No matter what, these two rules will cause ambiguities that we have to eliminate. We apply a few grammar transformations and then we end up with the following SDF grammar rules:

```
"PROCEDURE" "DIVISION" ("USING" Data-name+)? "." ("DECLARATIVES" "."
(Section-header "." Use-statement "." Paragraphs)+ "END"
"DECLARATIVES" ".")? Paragraphs (Section-header "." Paragraphs)*
        -> Procedure-division
```

The output of the disambiguation grammar transformations differs sometimes significantly from the original input. Nonetheless, we hope also to have shown with the examples that the process of disambiguation is quite straightforward and just a matter of work.

4.4 Enforcing Priorities

Some kind of ambiguities are more convenient to be addressed using a priority mechanism rather than special-purpose transformations on the actual grammar. Such mechanisms are usually supported by input languages of parser generators. It is, for example, common to enforce priorities of arithmetic operators and others in this way. Indeed, in SDF, priorities of context-free productions can be specified.

Let us consider an example concerned again with the `CALL` statement. There are two ways in which a `Call-rest-phrase` can be empty. The corresponding ambiguity can be resolved by the following priority:

```
   Call-overflow-phrase  -> Call-rest-phrase
         >
   Call-exception-phrase -> Call-rest-phrase
```

That means the first production rule has a higher priory than the second one. We solved in this way an ambiguity in the `CALL` statement since we give a higher priority to an `OVERFLOW` phrase than an `EXCEPTION` phrase.

The disambiguation is done back-to-back with extensive tests using the 2MLOC VS COBOL II code. By this, we end-up with a realistic parser that took us a few weeks to implement. We stress that similar technology and processes can be used for constructing parsers for all reasonably well documented language reference manuals.

4.5 From Context-free to LALR(1)

In this phase of the process we have a working parser that we are happy with. In our case, we want to use the parser for the renovation of legacy systems, so this puts special requirements on the grammar as was indicated earlier in the paper. For that purpose, it is convenient to use grammars with explicit list-constructs since then we can use those list constructs also in patterns [102] (think of `Statement*`) to identify certain pieces of code, and to update them to the new business needs. We will not discuss a grammar transformation that turns the VS COBOL II grammar into a renovation grammar. For details on such issues we refer to [103, 104, 106, 22]. Also from a parsing perspective, the paper [21] discusses why GLR is a more convenient to use for software renovation than is LR.

However, we recognize that others may have different goals with grammars, and presumably want to use different parser techniques. A natural goal would be then to further convert the grammar so that tools like Lex and Yacc can process the grammars without problems. Typically, the SDF grammars that we deliver contain native list constructs. The wide-spread class of LR parser generators does not allow for such constructs to be present. This implies that with yet another grammar transformation the grammar should be derecursified, and moreover the problem of shift/reduce and reduce/reduce conflicts needs to be addressed. We have personally not executed these steps, but others have. Both in academic and in industrial efforts, people have worked on exactly this part of the process, so we are pretty confident that for those who need to convert to the Lex/Yacc paradigm there are no unsurmountable problems. After all, the grammar specification or its derived SDF implementation comprises a very accurate and executable requirements specification.

In the RainCode system [11], in Software Refinery [93], and in a paper on optimizing a GLR algorithm [3], methods are proposed to turn grammar specifications that contain list constructions like `+` and `*` into recursive rules that can be handled by LR parser generators like Yacc.

Let us discuss the cases in a little more detail. In the RainCode approach they aim for approximations of LR languages. The lexer generator is called *lexyt* and the parser generator is called *Dura*. The lexyt/Dura pair uses integrated backtracking. During extensive use in industry, they claim that almost all languages are indeed rather close approximations of LR languages. The backtracking facility is just there to resolve the few remaining problems. This fall back mechanism also makes sure that no amount of work is necessary to resolve the classical shift/reduce and reduce/reduce conflicts. Also the RainCode approach uses internal grammar transformations to translate list constructions back to production rules that simulate recursion. In the paper [3] a similar

approach is taken, and constructions that are potentially time consuming are translated back to LR like constructions. In Software Refinery the language called DIALECT is used for defining syntax. Also here the DIALECT specifications are translated back to Lex/Yacc like structures. Since we have no insight in the sources of the DIALECT tool, it is hard to discuss the latter case in more detail.

5 Grammar Engineering Tools

The technology we proposed in this paper is simple. It is not at all the case that the implementation that we used to illustrate the approach is mandatory if others want to implement the approach as well. Tool support is feasible for the following aspects of grammar engineering with some emphasis on grammar recovery:

1. Raw grammar extraction
2. Generation of browsable grammars
3. Grammar manipulation
4. Error detection by parsing
5. Derivation of parser specification
6. Resolution of ambiguities
7. Efficient parsing

For the case study discussed in this paper we used the system LDL [49] for the first five aspects, whereas the ASF+SDF Meta-Environment [66] was used for the rest, i.e., realistic parsing including resolution of ambiguities. In the papers [101, 104, 105], we solely used the ASF+SDF Meta-Environmentas implementation platform for grammar recovery. It is not at all the case that grammar engineering tools can only be implemented using LDL or the ASF+SDF Meta-Environment. For convenience, we shortly characterize LDL and the ASF+SDF Meta-Environment.

The system LDL (LDL is short for Language Development Laboratory) is a specification framework for prototyping language tools. It has been derived in the LDL project [98, 71, 49] on provable correct prototype interpreters. It provides support for language design and test set generation. The actual specification formalisms supported by LDL cover lexical definition, attribute grammars, inference rules, module composition, text generation, meta-programming and others. LDL is based on an implementational model using Prolog [31, 87]. We refer to [69] for comparisons between various language development environments including LDL.

The ASF+SDF Meta-Environment is a generic interactive programming environment. We recall that SDF stands for syntax definition formalism and is used to describe arbitrary syntax (see [50] for a reference manual, and [114] for a recent version of SDF). ASF stands for algebraic specification formalism, and is used to describe semantics (see [5] for a textbook on this subject). For a comparison between the ASF+SDF Meta-Environment and related systems we refer to [22].

There are many other systems in which it is convenient to implement grammar engineering tools such as tools for extraction, generation of browsable grammars, grammar manipulation and parsing. We want to mention a few tools and criteria.

Tool support for grammar extraction is crucially dependent on the form of the available source. There is no single approach applying to extraction from parser generator inputs, manually written language processors, language manuals with textual syntax notation or visual notation or from other sources. In this paper, we discussed how to extract a raw grammar from IBM's references using simple lexical tools and a diagram parser relying on the ASCII encoding of the diagrams [75]. For more involved visual notation and/or proper graphical encoding, visual language tools might be needed [30, 29, 33]. Note that efficiency is not an issue in the extraction phase because extraction is done only once. Minimum development effort and adaptability are much more important.

Generation of browsable grammars is relatively easy to accomplish. It merely means to export grammars in HTML or some other suitable format. [39] presents browsable grammars for several languages with linked grammar rules and syntax diagrams based on HTML + Java for layouting the diagrams [13]. For the diagrams in our browsable grammars we used a pretty printer relying on ASCII-encoding. Syntax diagrams are very useful to deal with a language's syntax [40].

There are many systems facilitating transformational programming as required for grammar manipulation. We mention Software Refinery [94], Stratego [115], TXL [32], TAMPR [14], Eli [44], Gandalf [47], Elegant [88], RainCode [91], COSMOS [38], and so on.

Different kinds of parsers appear in our process. A diagram parser is used for grammar extraction. A simple parser is generated from the evolving grammar specification for error detection purposes based on a trusted test set. Ultimately, a realistic parser is derived. It has been argued throughout the paper that our process is more convenient if powerful parsing technology is deployed, but we also described how the more restrictive mainstream parsing technology can be integrated in the process. Support for general context-free grammars is particularly useful in the error detection phase because in this phase we would like to approve a proper grammar specification which is not tuned towards any grammar implementation criteria. Top-down parsing with backtracking as facilitated by LDL or PRECC [25], or generalized LR parsing [76, 110, 95, 114, 116] is convenient in this context.

Summarizing there are no real restrictions on what technology to use, so go ahead and use your favorite program transformation tool, lexer/parser generator etc. to setup your own grammar engineering tools so that you can quickly produce the parsers and other grammar-based tools that you need.

6 Concluding Remarks

In this paper we have shown that grammar transformations are a useful technique to recover large grammars and that most parts of the recovery process can be automated. We argued that the question of grammar recovery is urgent and seen as a large impediment for solving many and diverse pressing problems in software renovation, such as Euro conversions, language migrations, redocu-

mentation projects, system comprehension tasks, operating systems migrations, software merging projects due to company merges, web-enabling of mainframe systems, daily mass-maintenance projects, and so on. Although the grammar recovery problem has been recognized in the software engineering area it has been studied originally in linguistics. We argued that the solution domain in linguistics does not carry over to the software engineering field. The body of work that was presented here is the result of extensive experience in solving grammar recovery problems with numerous tools and technologies. The results that were achieved by these efforts are in our opinion very convincing and lead us to believe that the entire problem of grammar recovery for existing languages has been solved. We illustrated the approach that we proposed with the recovery and publication on the Internet of IBM's VS COBOL II grammar specification. We tested it with about 2 million lines of code. Furthermore, we were able to implement a realistic parser from the specification.

The application domain of our work extends to what is reported on in this paper. In the realm of computer documentation, in particular the production of language reference manuals and international standards for languages the proposed technology can be beneficial.

We estimate that in the techniques that we developed will find their way in future development environments where maintenance, enhancement and ultimately software renovation are supported by tools from the start. Our work will then not be outdated, for, then grammar engineering will be an integrated part of such development environments.

References

[1] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers. Principles, Techniques and Tools.* Addison-Wesley, 1986.

[2] H. Ahonen, H. Mannila, and E. Nikunen. Forming Grammars for Structured Documents: An Application of Grammatical Inference. In R. Carrasco and J. Oncina, editors, *Proceedings of the Second International Colloquium on Grammatical Inference and Applications (ICGI)*, volume 862 of *Lecture Notes in Artificial Intelligence*, pages 153–167. Springer-Verlag, 1994.

[3] J. Aycock and N. Horspool. Faster Generalized LR Parsing. In S. Jähnichen, editor, *Proceedings of the eight International Conference on Compiler Construction*, volume 1575 of *LNCS*, pages 32–46. Springer-Verlag, 1999.

[4] 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.

[5] J.A. Bergstra, J. Heering, and P. Klint. The algebraic specification formalism ASF. In J.A. Bergstra, J. Heering, and P. Klint, editors, *Algebraic Specification*, ACM Press Frontier Series, pages 1–66. The ACM Press in co-operation with Addison-Wesley, 1989.

[6] J.A. Bergstra and P. Klint. The ToolBus coordination architecture. In P. Ciancarini and C. Hankin, editors, *Coordination Languages and Models*, volume 1061 of *Lecture Notes in Computer Science*, pages 75–88, 1996.

[7] J.A. Bergstra and P. Klint. The discrete time ToolBus. In M. Wirsing and M. Nivat, editors, *Algebraic Methodology and Software Technology*, volume 1101 of *Lecture Notes in Computer Science*, pages 286–305. Springer-Verlag, 1996.

[8] J.A. Bergstra and P. Klint. The discrete time ToolBus—a software coordination architecture. *Science of Computer Programming*, 31:205–229, 1998.

[9] S. Björk. Parsers, Pretty Printers and PolyP. Master's thesis, Göteborg University, 1997.

[10] M.H. Blaha and W. Premerlani. *Object-Oriented Modeling and Design for Database Applications*. Prentice Hall, 1998.

[11] D. Blasband. *Automatic Analysis of Ancient Languages*. PhD thesis, Free University of Brussels, 2000. Available via http://www.phidani.be/homes/darius/thesis.html.

[12] B. Boehm. *Software Engineering Economics*. Prentice Hall, 1981.

[13] M. Bonjour, G. Falquet, J. Guyot, and A. Le Grand. *Java: de l'esprit á la méthode*. Editions Vuibert, 2nd edition, 1999. In French.

[14] J.M. Boyle. A transformational component for programming language grammar. Technical Report ANL-7690, Argonne National Laboratory, Argonne, Illinois, 1970.

[15] M.G.J. van den Brand, H.A. de Jong, P. Klint, and P.A. Olivier. Efficient annotated terms. *Software—Practice and Experience*, 30:259–291, 2000.

[16] M.G.J. van den Brand, P. Klint, and C. Verhoef. Core technologies for system renovation. In K.G. Jeffery, J. Král, and M. Bartošek, editors, *SOFSEM'96: Theory and Practice of Informatics*, volume 1175 of *LNCS*, pages 235–255. Springer-Verlag, 1996.

[17] M.G.J. van den Brand, P. Klint, and C. Verhoef. Re-engineering needs generic programming language technology. *ACM SIGPLAN Notices*, 32(2):54–61, 1997. Available at http://adam.wins.uva.nl/~x/sigplan/plan.html.

[18] M.G.J. van den Brand, P. Klint, and C. Verhoef. Term rewriting for sale. In C. Kirchner and H. Kirchner, editors, *Second International Workshop on Rewriting Logic and its Applications*, Electronic Notes in Theoretical Computer Science. Springer-Verlag, 1998. Available at: http://adam.wins.uva.nl/~x/sale/sale.html.

[19] M.G.J. van den Brand, M.P.A. Sellink, and C. Verhoef. Generation of components for software renovation factories from context-free grammars. In I.D. Baxter, A. Quilici, and C. Verhoef, editors, *Proceedings Fourth Working Conference on Reverse Engineering*, pages 144–153, 1997. Available at http://adam.wins.uva.nl/~x/trans/trans.html.

[20] 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*, electronic Workshops in Computing. Springer-Verlag, 1997. Available at http://adam.wins.uva.nl/~x/coboldef/coboldef.html.

[21] M.G.J. van den Brand, M.P.A. Sellink, and C. Verhoef. Current parsing techniques in software renovation considered harmful. In S. Tilley and G. Visaggio, editors, *Proceedings of the sixth International Workshop on Program Comprehension*, pages 108–117, 1998. Available at http://adam.wins.uva.nl/~x/ref/ref.html.

[22] M.G.J. van den Brand, M.P.A. Sellink, and C. Verhoef. Generation of components for software renovation factories from context-free grammars. *Science of Computer Programming*, 36(2–3):209–266, 2000. Available at http://adam.wins.uva.nl/~x/scp/scp.html. An extended abstract with the same title appeared earlier: [19].

[23] M.G.J. van den Brand and E. Visser. Generation of formatters for context-free languages. *ACM Transactions on Software Engineering and Methodology*, 5:1–41, 1996.

[24] T. Bray, J. Paoli, and C.x M. Sperberg-McQueen. Extensible Markup Language (XML) 1.0, February 1998. W3C Recommendation, WWW Consortium, http://www.w3.org/.

[25] P.T. Breuer and J.P. Bowen. A PREttier Compiler-Compiler: Generating Higher-order Parsers in C. *Software—Practice and Experience*, 25(11):1263—1297, November 1995.

[26] F. Bueno, P. Deransart, W. Drabent, G. Ferrand M. Hermenegildo, J. Małuszyński, and G. Puebla. On the Role of Semantic Approximations in Validation and Diagnosis of Contraint Logic Programs. In M. Kamkar, editor, *AADEBUG'97. Proceedings of the Third International Workshop on Automatic Debugging: Linköping, Sweden*, pages 155–170, 26–27 May 1997.

[27] R.D. Cameron. Extending context-free grammars with permutation phrases. *ACM Letters on Programming Languages and Systems*, 2(4):85–94, March 1993.

[28] J.A. Carroll. An island parsing interpreter for the full augmented transition network formalism. In *Proceedings of the first European Chapter of the Association for Computational Linguistics*, pages 101–105, Pisa, Italy, 1983.

[29] S.-K. Chang. A Visual Language Compiler for Information Retrieval by Visual Reasoning. *IEEE Transactions on Software Engineering*, 16(10):1136–1149, October 1990. Special Section on Visual Programming.

[30] S.-K. Chang, M.J. Tauber, B. Yu, and J.-S. Yu. A Visual Language Compiler. *IEEE Transactions on Software Engineering*, 15(5):506–525, May 1989.

[31] W.F. Clocksin and C.S. Mellish. *Programming in Prolog*. Springer-Verlag, Berlin, 3 edition, 1987.

[32] J.R. Cordy, C.D. Halpern-Hamu, and E. Promislow. TXL: A rapid prototyping system for programming language dialects. *Computer Languages*, 16(1):97–107, 1991.

[33] G. Costagliola, G. Tortora, S. Orefice, and A. De Lucia. Automatic Generation of Visual Programming Environments. *Computer*, 28(3):56–66, March 1995.

[34] T. Despeyroux. Typol: A formalism to implement Natural Semantics. Technical Report 94, INRIA Sophia-Antipolis, 1988.

[35] A. van Deursen, J. Heering, and P. Klint, editors. *Language Prototyping: An Algebraic Specification Approach*, volume 5 of *AMAST Series in Computing*. World Scientific Publishing Co., 1996.

[36] A. van Deursen and T. Kuipers. Building documentation generators. In H. Yang and L. White, editors, *Proceedings of the International Conference on Software Maintenance*, pages 40–49. IEEE Computer Society Press, 1999.

[37] J. Ebert, B. Kullbach, and A. Winter. GraX – An Interchange Format for Reengineering Tools. In F. Balmas, M. Blaha, and S. Rugaber, editors, *Proceedings of the Sixth Working Conference on Reverse Engineering*, pages 89–98, 1999.

[38] Emendo Software Group, The Netherlands. *Emendo Y2K White paper*, 1998. Available at http://www.emendo.com/.

[39] T. Estier and J. Guyot. The BNF Web Club, 1998. Centre Universitaire d'Informatique, Université de Genève, http://cui.unige.ch/db-research/Enseignement/analyseinfo/BNFweb.html.

[40] G. Falquet, J. Guyot, and L. Nerima. Simple tools to learn Ada. *ACM SIGADA Ada Letters*, 4(6):44–48, May/June 1985.

[41] G. Ferrand. The Notions of Symptom and Error in Declarative Diagnosis of Logic Programs. In P. Fritzon, editor, *Automated and Algorithmic Debugging, First International Workshop, AADEBUG'93*, volume 749 of *Lecture Notes in Computer Science*, pages 40–57. Springer-Verlag, 3–5 May 1993.

[42] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[43] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[44] R.W. Gray, V.P. Heuring, S.P. Levi, A.M. Sloane, and W.M. Waite. Eli: A complete, flexible compiler construction system. *Communications of the ACM*, 35(2):121–131, 1992.

[45] J. Grosch. Are attribute grammars used in industry? In D. Parigot and M. Mernik, editors, *Second Workshop on Attribute Grammars and their Applications, WAGA'99*, pages 1–16, Amsterdam, The Netherlands, March 1999. INRIA rocquencourt.

[46] J. Grosch and H. Emmelmann. A toolbox for compiler construction. In D. Hammer, editor, *Proceedings of the Third International Workshop on Compiler Compilers*, volume 477 of *Lecture Notes in Computer Science*, pages 106–116. Springer-Verlag, 1990.

[47] A.N. Habermann and D. Notkin. Gandalf: Software development environments. *IEEE Transactions on Software Engineering*, SE-12:1117–1127, 1986.

[48] B. Hall. Year 2000 tools and services. In *Symposium/ITxpo 96, The IT revolution continues: managing diversity in the 21st century*. GartnerGroup, 1996.

[49] J. Harm, R. Lämmel, and G. Riedewald. The Language Development Laboratory (LDL). In M. Haveraaen and O. Owe, editors, *Selected papers from the 8th Nordic Workshop on Programming Theory, December 4–6, Oslo, Norway, Research Report 248, ISBN 82-7368-163-7*, pages 77–86, May 1997.

[50] J. Heering, P.R.H. Hendriks, P. Klint, and J. Rekers. The syntax definition formalism SDF — Reference manual. *SIGPLAN Notices*, 24(11):43–75, 1989.

[51] J. Heering, G. Kahn, P. Klint, and B. Lang. Generation of interactive programming environments. In The Commission of the European Communities, editor, *Esprit '85 - Status Report of Continuing Work 1*, pages 467–477. North-Holland, 1986.

[52] J. Heering and P. Klint. Semantics of programming languages: A tool-oriented approach. *ACM SIGPLAN Notices*, March 2000.

[53] IBM Corporation. *IBM PL/I for VSE/ESA(TM) Programming Guide*, release 1 publication number sc26-8053-01 edition, 1971, 1998.

[54] IBM Corporation. *IBM BookManager BookServer Library*, 1989, 1997. In 1999 and 2000 accessible via http://www.s390.ibm.com:80/bookmgr-cgi/bookmgr.cmd/library.

[55] IBM Corporation. *VS COBOL II Application Programming Language Reference*, 4 Publication number GC26-4047-07 edition, 1993.

[56] IBM Corporation. *VS COBOL II Reference Summary*, 1.2. Publication number SX26-3721-05 edition, 1993.

[57] IBM Corporation. *COBOL for OS/390 & VM, COBOL Set for AIX, VisualAge COBOL – Language Reference*, fourth edition, 1998.

[58] International Telecommunication Union. *Recommendation Z.200 (11/93) - CCITT High Level Language (CHILL)*, 1993.

[59] S.C. Johnson. YACC - Yet Another Compiler-Compiler. Technical Report Computer Science No. 32, Bell Laboratories, Murray Hill, New Jersey, 1975.

[60] C. Jones. *Software Productivity and Quality – The World Wide Perspective*. IS Management Group, Carlsbad, CA, 1993.

[61] C. Jones. *Applied Software Measurement: Assuring Productivity and Quality*. McGraw-Hill, second edition, 1996.

[62] C. Jones. *Estimating Software Costs*. McGraw-Hill, 1998.

[63] N. Jones. Year 2000 market overview. Technical report, GartnerGroup, Stamford, CT, USA, 1998.

[64] G. Kahn, B. Lang, B. Mélèse, and E. Morcos. Metal: A formalism to specify formalisms. *Science of Computer Programming*, 3:151–188, 1983.

[65] R. Kazman, S.G. Woods, and J. Carrière. Requirements for Integrating Software Architecture and Reengineering Models: CORUM II. In M.H. Blaha, A. Quilici, and C. Verhoef, editors, *Proceedings of the Fifth Working Conference on Reverse Engineering*, pages 154–163, 1998.

[66] P. Klint. A meta-environment for generating programming environments. *ACM Transactions on Software Engineering and Methodology*, 2(2):176–201, 1993.

[67] P. Klint and C. Verhoef. Evolutionary software engineering: A component-based approach. In R.N. Horspool, editor, *IFIP WG 2.4 Working Conference: Systems Implementation 2000: Languages, Methods and Tools*, pages 1–18. Chapman & Hall, 1998. Available at: http://adam.wins.uva.nl/~x/evol-se/evol-se.html.

[68] P. Klint and E. Visser. Using filters for the disambiguation of context-free grammars. Technical Report P9426, Programming Research Group, University of Amsterdam, 1994.

[69] P. Knauber. *Ein System für die Konstruktion objektorienierter Übersetzer*. PhD thesis, Universität Kaiserslautern, 1997. In German.

[70] R. Koschke, J.-F. Girard, and M. Würthner. Intermediate Representation for Integrating Reverse Engineering Analyses. In M.H. Blaha, A. Quilici, and C. Verhoef, editors, *Proceedings of the Fifth Working Conference on Reverse Engineering*, pages 241–250, 1998.

[71] R. Lämmel and G. Riedewald. Provable Correctness of Prototype Interpreters in LDL. In P. A. Fritzson, editor, *Proceedings of Compiler Construction CC'94, 5th International Conference, CC'94, Edinburgh, U.K.*, volume 786 of *Lecture Notes in Computer Science*, pages 218–232. Springer-Verlag, 1994.

[72] R. Lämmel and C. Verhoef. *COBOL 2000 grammar Version 0.1*, 1999. Available at: http://adam.wins.uva.nl/~x/grammars/cobol-2000/.

[73] R. Lämmel and C. Verhoef. *OS PL/I V2R3 grammar Version 0.1*, 1999. Available at: http://adam.wins.uva.nl/~x/grammars/os-pli-v2r3/.

[74] R. Lämmel and C. Verhoef. *VS COBOL II grammar Version 1.0.2*, 1999. Available at: http://adam.wins.uva.nl/~x/grammars/vs-cobol-ii/.

[75] R. Lämmel and C. Verhoef. Rejuvenation of an Ancient Visual Language. Draft; available at http://www.cwi.nl/~ralf/; submitted for publication, April 2000.

[76] 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.

[77] M.E. Lesk and E. Schmidt. *LEX - A lexical analyzer generator*. Bell Laboratories, UNIX Programmer's Supplementary Documents, volume 1 (PS1) edition, 1986.

[78] J.R. Levine, T. Mason, and D. Brown. Yacc ambiguities and Conflicts. In *lex & yacc*, pages 217–241. O'Reilly & Associates, Inc., 2nd edition, 1992.

[79] B.P. Lientz and E.B. Swanson. *Software Maintenance Management—A Study of the Maintenance of Computer Application Software in 487 Data Processing Organizations*. Reading MA: Addison-Wesley, 1980.

[80] M. de Jonge. A Pretty-Printer for Every Occasion. to appear in the proceedings of The Second International Symposium on the Constructing Software Engineering Tools (CoSET 2000), 2000.

[81] V. Maslov. Re: An odd grammar question, 1998. Retrieved via: http://www.iecc.com/comparch/article/98-05-108.

[82] T.J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-12(3):308–320, 1976.

[83] S. McConnell. *Rapid Development*. Microsoft Press, 1996.

[84] L. Miclet and C. de la Higuera, editors. *Grammatical Inference: Learning Syntax from Sentences*, volume 1147 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, 1996.

[85] E. Morcos-Chounet and A. Conchon. PPML: a general formalism to specify pretty printing. In H.-J. Kugler, editor, *Information Processing 86*, pages 583–590. Elsevier, 1986.

[86] 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. University of Twente, 1992.

[87] U. Nilsson and J. Maluszynski. *Logic Programming and Prolog*. John Wiley, 2 edition, 1995.

[88] Philips Electronics B.V., The Netherlands. *The Elegant Home Page*, 1993. http://www.research.philips.com/generalinfo/special/elegant/elegant.html.

[89] P. Purdom. A sentence generator for testing parsers. *Behaviour and Information Technology*, 12(3):366–375, July 1972.

[90] L.H. Putnam and W. Myers. *Measures for Excellence – Reliable Software on Time, Within Budget*. Yourdon Press Computing Series, 1992.

[91] RainCode, Brussels, Belgium. *RainCode*, 1.07 edition, 1998. ftp://ftp.raincode.com/cobrc.ps.

[92] K. Rayhawk. Re: gnubol: How do we parse this language, anyway?, 1999. Retrieved via: http://www.lusars.net/maillists/gnu-cobol/msg00836.html.

[93] Reasoning Systems, Palo Alto, California. *DIALECT user's guide*, 1992.

[94] Reasoning Systems, Palo Alto, California. *Refine User's Guide*, 1992.

[95] J. Rekers. *Parser Generation for Interactive Environments*. PhD thesis, University of Amsterdam, 1992. ftp://ftp.cwi.nl/pub/gipe/reports/Rek92.ps.Z.

[96] H. Reubenstein, R. Piazza, and S. Roberts. Separating parsing and analysis in reverse engineering tools. In *Proceedings of the 1st Working Conference on Reverse Engineering*, pages 117–125, 1993.

[97] J. Reutter. Maintenance is a management problem and a programmer's opportunity. In A. Orden and M. Evens, editors, *1981 National Computer Conference*, volume 50 of *AFIPS Conference Proceedings*, pages 343–347. AFIPS Press, Arlington, VA, 1981.

[98] G. Riedewald. The LDL — Language Development Laboratory. In U. Kastens and P. Pfahler, editors, *Compiler Construction, 4th International Conference, CC'92, Paderborn, Germany*, number 641 in LNCS, pages 88–94. Springer-Verlag, October 1992.

[99] J. Rumbaugh, M.H. Blaha, W. Premerlani, F. Eddy, and W. Lorenson. *Object-Oriented Modeling and Design*. Prentice Hall, 1991.

[100] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Object Technology Series. Addison-Wesley, 1998.

[101] M.P.A. Sellink and C. Verhoef. Development, assessment, and reengineering of language descriptions – extended abstract. In B. Nuseibeh, D. Redmiles, and A. Quilici, editors, *Proceedings of the 13th International Automated Software Engineering Conference*, pages 314–317, 1998. For a full version see [105]. Available at: http://adam.wins.uva.nl/~x/ase/ase.html.

[102] M.P.A. Sellink and C. Verhoef. Native patterns. In M. Blaha, A. Quilici, and C. Verhoef, editors, *Proceedings of the Fifth Working Conference on Reverse Engineering*, pages 89–103. IEEE Computer Society, 1998. Available at http://adam.wins.uva.nl/~x/npl/npl.html.

[103] M.P.A. Sellink and C. Verhoef. An Architecture for Automated Software Maintenance. In D. Smith and S.G. Woods, editors, *Proceedings of the Seventh International Workshop on Program Comprehension*, pages 38–48. IEEE Computer Society Press, 1999. Available at http://adam.wins.uva.nl/~x/asm/asm.html.

[104] M.P.A. Sellink and C. Verhoef. Generation of software renovation factories from compilers. In H. Yang and L. White, editors, *Proceedings of the International Conference on Software Maintenance*, pages 245–255. IEEE Computer Society Press, 1999. Available via http://adam.wins.uva.nl/~x/com/com.html.

[105] 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*, pages 151–160. IEEE Computer Society, March 2000. Full version of [101]. Available at: http://adam.wins.uva.nl/~x/cale/cale.html.

[106] M.P.A. Sellink and C. Verhoef. Scaffolding for software renovation. In J. Ebert and C. Verhoef, editors, *Proceedings of the Fourth European Conference on Software Maintenance and Reengineering*, pages 161–172. IEEE Computer Society Press, March 2000. Available via http://adam.wins.uva.nl/~x/scaf/scaf.html.

[107] D.R. Smith, G.B. Kotik, and S.J. Westfold. Research on knowledge-based software environments at Kestrel institute. *IEEE Transactions on Software Engineering*, SE-11(11):1278–1295, 1985.

[108] SoftLab, Linköping, Sweden. *ToolMaker Reference Manual*, version 2.0 edition, 1994.

[109] O. Stock, R. Falcone, and P. Insinnamo. Island parsing and bidirectional charts. In *Proc. of the 12th COLING*, pages 636–641, Budapest, Hungary, 1988.

[110] M. Tomita. *Efficient Parsing for Natural Languages—A Fast Algorithm for Practical Systems*. Kluwer Academic Publishers, 1986.

[111] W.M. Ulrich. The evolutionary growth of software reengineering and the decade ahead. *American Programmer*, 3(11):14–20, 1990.

[112] C. Verhoef. Re: How to extract grammar from a program?, 1999. Retrieved via: http://www.iecc.com/comparch/article/99-12-042.

[113] C. Verhoef. Towards Automated Modification of Legacy Assets. *Annals of Software Engineering*, 9, March 2000. Available at http://adam.wins.uva.nl/~x/ase/ase.html.

[114] E. Visser. Scannerless Generalized-LR Parsing. Technical Report P9707, Programming Research Group, University of Amsterdam, July 1997. Available at http://www.wins.uva.nl/pub/programming-research/reports/1997/P9707.ps.

[115] E. Visser, Z.-A. Benaissa, and A. Tolmach. Building Program Optimizers with Rewriting Strategies. In *International Conference on Functional Programming (ICFP'98), Baltimore, Maryland. ACM SIGPLAN*, pages 13–26, September 1998.

[116] E. Visser, J. Scheerder, and M. van den Brand. Scannerless Generalized-LR Parsing, 2000. Work in Progress.

[117] J.B. Warmer and A.G. Kleppe. *The Object Constraint Language – Precise Modeling With UML*. Object Technology Series. Addison-Wesley, 1999.

[118] G.M. Weinberg. *The Psychology of Computer Programming*. Van Nostrand Reinhold, 1971.

[119] D.S. Wile. Abstract syntax from concrete syntax. In *Proceedings of the 1997 International Conference on Software Engineering*, pages 472–480. ACM Press, 1997.

[120] D.S. Wile. Integrating Syntaxes and their Associated Semantics. Draft, 1999.

[121] S.G. Woods, L. O'Brian T. Lin, , K. Gallagher, and A. Quilici. An architecture for interoperable program understanding tools. In *Proceedings of the sixth International Workshop on Program Comprehension*, pages 54–63, 1998.