SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2001; **12**:1–6 *Prepared using* speauth.cls *[Version: 2000/03/16 v2.12]*

Semi-automatic Grammar Recovery

R. Lämmel[1,2,†], C. Verhoef[*,2,‡]

[1] *CWI, Kruislaan 413, 1098 SJ Amsterdam, The Netherlands*
[2] *Division of Mathematics and Computer Science, Vrije University Amsterdam, De Boelelaan 1081a, 1081 HV Amsterdam, The Netherlands*

SUMMARY

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

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

INTRODUCTION

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

*Correspondence to: Division of Mathematics and Computer Science, Vrije University Amsterdam, De Boelelaan 1081a, 1081 HV Amsterdam, The Netherlands
†E-mail: ralf@cs.vu.nl
‡E-mail: x@cs.vu.nl
Contract/grant sponsor: The first author received partial support from the Netherlands Organization for Scientific Research (NWO) under the *Generation of Program Transformation Systems* project.

Copyright © 2001 John Wiley & Sons, Ltd.
Received 1 December 2000
Revised 30 July 2001
Accepted 7 August 2001

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 [2, 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 [3] and UML [4] are probably the most prevalently known visual ones. Seen from this perspective the area of language development is a lively part of software engineering. Indeed so-called language prototyping and construction tools are becoming a sub-industry of their own to supply the language needs [5, 6]. There is a wealth of tools to aid in the design and implementation of programming, specification and modeling languages. The tools Lex [7] and Yacc [8] are the best known ones that come to mind here. But there are many others (we come back to this later).

An important aspect of any language is its grammar. A grammar is the formal specification of the syntactic structure of a language. Such specifications are indispensable inputs to parser generators, like Lex and Yacc, or other generic tools, such as programming environment generators. Grammars are omnipresent in software engineering. Not solely in the language technology field, but also in other areas. For instance, the processing of simple input is such a common problem that there are software patterns devoted to the issue [9]. Or in database design, flat-files are designed using grammars [10]. Also document type definitions (DTDs) or XML schemata in the sense of XML [11, 12, 13, 14] can be regarded as grammars. DTDs/XML are widely used to define the abstract representation of structured documents. Also studying grammars is often necessary. Any language reference manual contains grammar descriptions: from telecommunications language standards like Chill [15] to modern programming languages like Java [16] to the Object Constraint Language (OCL) [17] for UML. In order to learn such a language, studying its grammar is necessary.

Many people use grammars, read grammars, and there are industry proven tools like Lex and Yacc or design patterns, to deal with them. So, at first thought there is no problem, and thus no reason for this paper. At second thought, there are good reasons for our research. In 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 [18, p. 319]. The connection with grammars is that tools to enhance software assets heavily rely on those grammars. There is so much code around (about 7 billion function points [19, 20]) that it is becoming prohibitively expensive in time and effort to deal with the existing codebase by hand. For instance, in 1993 some of the Fortune 500 companies found themselves already in a situation where the entire maintenance budget is consumed with updating, repairing and enhancing aging (legacy) systems [21], so no green field development can be commenced. This post-release work is mainly done by hand, but some of it is amenable to be done using extensive tool support. Just to give an example, in a company someone spent a year on adding `END-IF`s to COBOL programs. Using a software renovation factory, we do this at a speed of 150 `END-IF`s per minute. Apart from that, if a software portfolio exceeds two million lines of code, the Gartner Group advices to use a software renovation factory to convert its systems to Y2K compliant ones [22, 23]. Of course when dealing with other changes amenable to automation there is no reason to assume that this advice is no longer valid. In other words, for many daily (mass) maintenance projects, we should better use computer-based tools to process programs, systems, or even entire

Copyright © 2001 John Wiley & Sons, Ltd.

software portfolios. Such tools give us interesting information about the software under investigation. In addition to *analysis* of existing systems, there is a constant need for tools that automatically *modify* entire systems to diminish the workload. Many such tools have in common their heavy reliance on the underlying grammars.

The premise of this article is to provide 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 several languages (more on that later). To illustrate our approach and to show the power of the approach, as a running example, we recover the grammar of a particularly hard, ancient, but pervasive language, namely IBM's VS COBOL II. We were able to entirely recover IBM's VS COBOL II grammar in two weeks, which includes the development of all the necessary tools to accomplish this. We used existing parser generators to test the recovered grammar. We tested the extracted grammars on VS COBOL II code, and parsed about two 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 [24] the effort for a single knowledgeable person as follows:

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

Another indication that constructing a COBOL parser is a 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 [25]. The published grammar was recovered with the techniques presented in this paper.

Generality of the approach Although the case study in this paper is focused on IBM's VS COBOL II for mainframes, we like to point out that the results described in this paper are not at all restricted to mainframes, or COBOL, or our implementation vehicles. Our methods are restricted to languages for which no grammar information whatsoever exists. Often this implies that there are no business critical applications for it either, for, if business critical applications exist for a language, the chances that there is neither a language reference nor compiler sources are virtually zero. But we have seen such exceptions. One is IBM's RPG that is running on the AS/400 midrange computer. For this language there is no grammar description: the language is explained by example. There is a freeware RPG compiler on the Internet, but it is not a complete compiler. So in the case of RPG it is not possible to use our methods directly; not because our approach is limited, but since there is no grammar information to our avail. We have experimented with extracting an RPG grammar, and our approach so far is to extract grammar productions from the examples given in the IBM manual. We think that this approach plus a preprocessing phase where RPG is made a bit more context-free leads to a cost-effective RPG grammar. Since we do not have sufficient RPG code to test an extracted RPG grammar, we have not (yet) done the RPG case. We think that RPG will give rise to a variant of our approach to recover its grammar from the code examples manual. Overall, our results are applicable to most existing languages for which business critical applications exist. We give an indication of this by listing a few languages for which we, and others, have performed grammar recovery projects. This list is not meant to be

Copyright © 2001 John Wiley & Sons, Ltd.

Prepared using speauth.cls

complete, but meant to be diverse. Our methods are applicable for modern languages such as Java and C++ or old ones such as COBOL, PL/I, and CICS, and even proprietary languages such as PLEX, or languages for which a standard exists, such as CHILL.

- The official C++ grammar was retrieved from the home page of AT&T, converted and adapted so that it was fit to be used in a tool to optimize C++ code meant for numerical software [26].
- We constructed a Java grammar from an online Java language specification†. Also others who reacted on this paper have constructed Java parsers using our process.
- For the CHILL language, we constructed a grammar from the ITU standard [15], a document written in MS Word.
- For the PLEX language, a proprietary telecommunications language from Ericsson, we constructed a parser using the source code of the compiler. Note that this is not a trivial project: the PLEX language consists of about 20 sublanguages, that needed to be combined in one large grammar for renovation purposes [27].
- Most of the languages for which IBM published an online manual can be dealt with using the tools we have developed for the VS COBOL II case study. They are: PL/I, SQL, CICS, and so on. See [28, 29] for initial versions of PL/I and COBOL 2000 that we extracted using our approach.

The paper focuses on the recovery of grammars to serve the software renovation field. But the applicability of our work is not at all restricted to that field. In addition to the construction of software analysis and modification tools, our work can be used to deliver correct language standards and to generate correct language manuals. Moreover, in the free software community at least two efforts are on their way to implement COBOL compilers, and the output of our research is used to understand how to implement a parser for those compiler projects [30].

Executive summary We briefly introduce our approach. We want to stress right now that the approach is simple, and that no magic is involved. For, the grammars are already written, we only have to extract them and transform them into the correct form. More precisely, the following steps are taken:

- raw grammar extraction from a language reference, a compiler or another artifact;
- resolution of static errors such as unconnected nonterminals, also called sort names, if the grammar is extracted from a nonexecutable source;
- extraction or definition of lexical syntax;
- test-driven correction and completion of the raw grammar if necessary;
- beautification;
- modularization;
- disambiguation if necessary;
- generation of a browsable version of the grammar if needed;
- adaptation of the grammar for the intended purpose (e.g., renovation).

The necessity of grammar transformations For the VS COBOL II case study, we developed tools to extract the syntax diagrams provided in IBM's language reference to obtain a raw grammar. The IBM

†see `java.sun.com/docs/books/jls/`

Copyright © 2001 John Wiley & Sons, Ltd.

Prepared using speauth.cls

reference is a nonexecutable resource, and we applied grammar transformations to derive a complete and correct grammar for the intended COBOL dialect. In other projects, we extracted the raw grammar from other types of manuals, or from compiler source code. Grammars reside in compiler source code, in language reference manuals, in language formatters, in complexity metric tools, in function point counting tools, etc. 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 tradeoffs. As a consequence, they are perfect for one purpose but useless when applied to another task. For instance, for an automated Euro conversion project it is not a good idea to reuse parser output from a compiler. Namely, if the converted system is parsed with such a parser, include files are indeed included, macro expansions are indeed expanded, the syntax is minimalized, embedded languages are expanded by preprocessors, and all 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, e.g., 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 another application. The idea of grammar transformations is not new in the broader context of grammar and language implementation. Let us mention two examples. In [31], grammar transformations are used for the development of domain-specific languages starting from reusable syntax components. In [32], semantics-preserving grammar transformations are used to derive abstract syntaxes from concrete syntaxes. The other direction is also common since it is often part of the implementation of a grammar specification with rather restricted parser generators such as Yacc (more on that later). The original contribution of the present article is that we show the essential role of grammar transformations in grammar recovery.

Grammar life-cycle enabling It is our experience that a grammar for a certain purpose can often be obtained from another grammar by means of formal grammar transformations. One could call that the *grammar life-cycle*. Although there can be significant algorithmic complexity involved in such grammar transformations (viz. the creation of scaffolding grammars [33]), 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*.

Application potential 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 we need are named *renovation grammars*. They are equipped to make automated modifications to software without all the just mentioned unwanted side-effects that a typical compiler grammar has. So you cannot use compiler grammars as is, if they are available to you at all. This grammar problem is also urgent for others than grammar specialists. For instance, Ed Yourdon popularized the *500 language problem*, a problem observed earlier by Capers Jones in his book [1]. The 500 language problem was one of the major impediments to solving the Y2K problem with tools: it 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 myriads of different languages (500 commercially available plus 200 proprietary

Copyright © 2001 John Wiley & Sons, Ltd.

Prepared using speauth.cls

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 large scale maintenance projects, and so on are in need of such grammars. All such software enhancement problems are indeed urgent, given the fact that many organizations are spending most of their software dollar on activities after the first release of a software system.

In 1998 it was estimated that there are seven billion function points of software written in 700 different languages, in constant need for modification. Obtaining grammars rapidly is a crucial first step in the construction of software renovation factories. This in turn is an important step towards automating software renovation problems.

Software renovation factories This is a product-line architecture that enables rapid development of tools intended to perform large scale changes on entire software systems. We give the reader an idea of such a large scale modification project. To convert COBOL 85 systems back to COBOL 74, we developed a set of tools that runs on more than 10 SUN Workstations, processing 500,000 LOC/hour so that a 10 MLOC conversion job is processed in 24 hours. This factory is a highly automated tool performing a problem-specific task for which no shrink-wrapped tool support exists, but that is easily programmed using the product-line architecture. The conversion from COBOL 85 back to COBOL 74 was necessary for a set of new applications that were supposed to be connected with COBOL 74 programs that should have been upgraded to COBOL 85 by the time they were ready, but since the COBOL 74 to COBOL 85 project took too long, a temporary conversion back was necessary [34].

We observed a trend among large software portfolio owners to experiment with or even install such facilities on-site to investigate how they can help enable the rapid pace of change to their existing software assets. Renovation grammars are crucial assets in the construction or even generation of major components of a software renovation factory. Generative and other construction issues for software renovation factories are elaborately discussed elsewhere [35, 36, 37, 38, 39, 40, 41, 42, 43, 33, 44]. 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.

Our earlier work Perhaps due to the firm establishment of parser generation technology in the 1970s, it has taken the software renovation field a long time to realize that mainstream compiler-oriented parser generation technology is problematic. In the paper [45] we suggested that mainstream parser generation technology is harmful and that we need alternatives. Therefore, not much work on grammar engineering other than ours can be found, although a renaissance of grammar-oriented research is to be expected. A positive sign in that direction is a recent PhD Thesis [46] by Blasband, 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. For a summary see [47]. Also others started to work on grammar engineering recently: the recovery of a proprietary AT & T SDL dialect using grammar engineering will be published in the International Conference on Software Maintenance [48]. Another recent paper on a program transformation framework involves grammar engineering support as a consequence of the necessity of dealing with (large) grammars [49].

Copyright © 2001 John Wiley & Sons, Ltd.

So, work by others on 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 [50]. 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 attempt failed to come up with a working parser (because the on-line documentation contained much less than the actual language comprised) the paper was an important contribution. For, the technology proposed in [50] was used fruitfully to entirely recover the same switching system language grammar from the source code of its compiler [27]. After this result was achieved, we realized that grammar recovery is an important subject, and we decided to publish the full version [51] of its extended abstract [50]. The current paper's roots are based on the full version [51]. Building on its ideas, we show that 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 work in grammar reconstruction, and we 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 reference [52] 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 [53, 54], 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 [55] 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 (not even 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, etc. 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 above mentioned artifacts. Of course, we use the source texts as debugging aids to incrementally improve the raw extracted grammar.

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 an abstract syntax tree (AST) for program comprehension purposes. Also in [56] 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-

Copyright © 2001 John Wiley & Sons, Ltd.

Prepared using speauth.cls

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. We start with the grammar life-cycle that comprises the artifacts containing grammars and their connections, followed by the case study. This study shows how to recover a complete and correct specification of IBM's VS COBOL II grammar. Based on our experience with this and other cases studies, we suggest a structured process to obtain a grammar specification and to derive a realistic parser. We briefly discuss possible platforms for implementing grammar engineering tools, to emphasize that with virtually all compiler compilers, or language design and processing environments you can implement grammar engineering tools.

THE GRAMMAR LIFE-CYCLE

As illustrated in the introductory section, 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 [56]:

- compilers,
- debuggers,
- animators,
- profilers,
- pretty printers,
- language reference manuals,
- language browsers,
- software analysis tools,
- code preprocessing tools,
- software modification tools,
- test-set generation tools,
- software testing tools, etc.

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 similarities and differences.

Copyright © 2001 John Wiley & Sons, Ltd.

Prepared using speauth.cls

Compilers The grammar(s) residing in a compiler we call *compiler grammars*. We assume familiarity with the reference architecture of compilers [57]. 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. The 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. 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 a COBOL fragment 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 our error correction parser did: when we parsed the code with our recovered grammar [25], it did not parse the `GREATER THEN` part. The reason is that in the manual it spells `THAN` instead of `THEN`, which is reflected in our parser. 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 OS/VS COBOL 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 erroneous usage of `NOT IS` in conditions in code by programmers who are nonnative 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 significantly different grammars for renovation than the ones that exist in products like compilers or manuals.

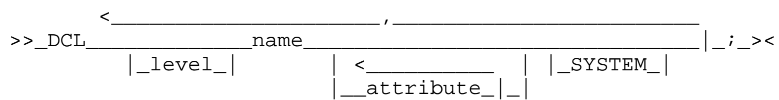
Pretty printers They are tools that format code in a consistent style. It is not necessary for such a tool to do a deep analysis of all the aspects of the code in order to pretty print it. We call the grammar of a pretty printer a *formatter grammar*. A formatter grammar is different from a compiler 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. Although

Copyright © 2001 John Wiley & Sons, Ltd.

it is not possible to predefine all heuristics of pretty printing, it is possible to generate large parts of pretty printers from the definition of a language, so also here the grammar is a crucial asset to have access to. More information on the generation of formatters from context-free grammars can be found in [58, 36, 59].

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 normally not suited to be used for parser generation: it is usually ambiguous.

Visual languages such as syntax diagrams (often found in language reference manuals) are more appropriate for human understanding than for other purposes. As an example, we depicted the syntax diagram of the `DCL` construct as we found it in IBM's PL/I language reference manual [60]:



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 `attributes` is optional and the keyword `SYSTEM` is. 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 language syntax [61].

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 [62]. Fast and flexible access to a language reference manual is nowadays best accomplished by a hypertext version of the manual. Ideally, 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 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. Also in the BNF Web Club project [63], a 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

Copyright © 2001 John Wiley & Sons, Ltd.

Prepared using speauth.cls

on backfiring, that is, estimating the number of function points by counting logical source lines of code [18]. For a tool that provides rapid insight into the call structure of an entire system, it is only necessary to parse those pieces of a system contributing to the information that constitutes the call-structure. For the McCabe complexity metric [64], 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 *analyzer 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 [65, 66]. The accompanying partial grammars are therefore sometimes called *island grammars* [67]. With this partial parsing technology, one trades velocity (of implementation) for accuracy and completeness. By picking the relevant pieces of syntax that are analyzed in depth, it is hoped that the accuracy is not traded off. Before our work on grammar recovery, island parsing speeded up the development of analysis tools: it was not necessary to implement an entire parser, which is 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.

Exchange formats and parsers Parsers for software analysis tools are often so idiosyncratic that even exchanging output with other tools is impossible, as was observed in a paper attempting this [68]. This problem is seen as important in the software renovation field as can be seen by the many papers proposing different common exchange formats [69, 70, 71, 72]. Dagstuhl[‡] seminar No. 01041 was devoted to the subject of interoperability between reengineering tools. Moreover there was a Workshop on Standard Exchange Formats dealing with the issue. Although we recognize the importance for software reengineering tools—they should be open, and exchange information freely between each other [35, 41, 73, 74, 75, 76]—we sensed in many discussions with this community that the problems as discussed in [68] 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 above mentioned problems, since we reuse the grammars underlying the parsers and not just their idiosyncratic output. In that way we can easily massage the output in any format necessary for a particular tool. If reengineering tools are parameterized by a language grammar, then exchanging high-quality grammars instead of parsed input solves in our opinion the interoperability problem for software reengineering tools.

Software modification tools Software modification tools are less widespread, 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;

[‡]Schloß Dagstuhl is a German initiative to bring together, every week, leaders in computer science to discuss cutting-edge computer science research at the international forefront.

Copyright © 2001 John Wiley & Sons, Ltd.

- (home grown) preprocessors;
- compiler directives;
- unexpanded macros;
- unexpanded include files;
- debugging lines;
- continuation lines;
- undocumented syntax (cf. the `GREATER THEN` issue);
- comments;
- layout;
- annotations in source code, also called scaffolding constructs [33].

We can characterize the above list by stating that the source code, as the software engineer writes and reads it, should be accepted by a parser as is. Moreover, by unparsing the code as much as possible syntax retention should be achieved. Grammars 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 derived in a cost-effective manner.

Connections Between Grammars

The base-line grammar 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 [77, 78, 79, 80, 81]. In this case, the grammar was in all cases the same artifact. So grammars play a central role in such programming environments.

Specific grammars by transformation We extend the idea of a base-line grammar by turning down the assumption that all artifacts have to be generated (more or less) *directly* from the base-line grammar. By contrast, we assume that each artifact can also be 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 [82, 83, 84, 1, 85, 86].

The grammar net In Figure 1, we present a number of language-oriented tools: a compiler, a language reference manual, an on-line manual, a pretty printer, 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

Copyright © 2001 John Wiley & Sons, Ltd.

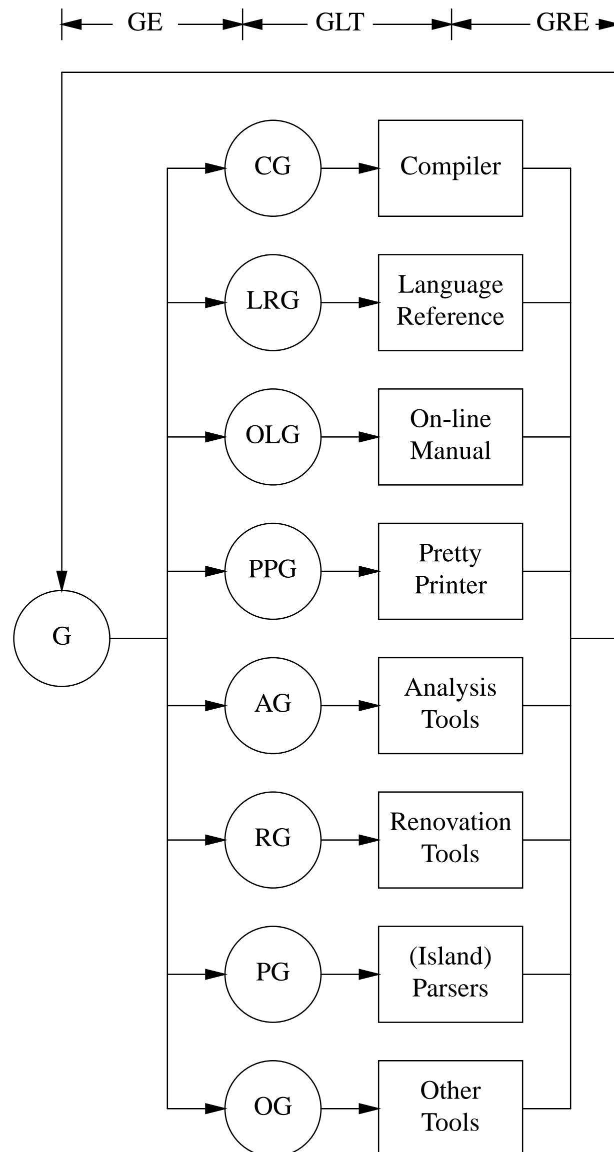


Figure 1. The Grammar Life-Cycle.

Copyright © 2001 John Wiley & Sons, Ltd.
Prepared using speauth.cls

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.
- GRE is Grammar Reverse Engineering.

The arrows above the figure are indicating the range of the technologies. Their borders are more diffuse than the picture suggests.

Examples Figure 1 visualizes the grammar life-cycle. To illustrate the connections and paths covered by the grammar life-cycle, we discuss a few examples.

A very important path in Figure 1 is one that helps in the construction of software renovation factories. In [27] we extracted 20 grammars from the source code of proprietary compilers 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 renovation grammar from which in turn large parts of a software renovation factory were generated [27].

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 useful in implementing island grammars suited for system understanding tasks. We have published a recovered but incomplete PL/I grammar [28] that has been used for implementing an island parser for PL/I. 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. The VS COBOL II grammar that we published [25] has been used for the implementation of island and full parsers. 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 [87] 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. 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.

Towards integration 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.

Copyright © 2001 John Wiley & Sons, Ltd.

Prepared using speauth.cls

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. When we need to renovate VS COBOL II code this is no reason for compiler vendor IBM to provide us with the source code of their compiler. There are obvious competitive reasons for that. So how to solve this issue? One solution is to write one by hand. We, and others, have done that [38]. 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 the resulting grammar when working entirely by hand. We will follow a different approach than writing the grammar by hand. Using a semi-automatic approach, we derive the VS COBOL II grammar in a traceable and cost-effective way.

We recall that our approach is not at all specific to mainframes, COBOL, and/or our choice of implementation. For other languages and dialects, certain details of the steps in the process may vary, e.g., the kind of artifact used for extraction, or the amount of different kinds of errors to be addressed. Yet, there are very good reasons that made us decide to address the IBM Mainframe VS COBOL II case study in this paper. These are our reasons:

- 70% of all the business critical software runs on mainframes [88, p. 13].
- 60% of all mainframe applications makes use of COBOL [89].
- 75% of all production transactions on mainframes is done using COBOL [90, p. 70].
- Over 60% of all Web-access data resides on a mainframe [90, p. 70].
- COBOL mainframes process more than 83% of all transactions worldwide [90, p. 70].
- Over 95% of finance-insurance data is processed with COBOL [90, p. 70]
- Certain versions of the VS COBOL II compiler are no longer supported; the remaining newer versions of the compiler have been supported until March 2001 [91]. After that, there is no compiler support for VS COBOL II anymore. Runtime support for the applications is much longer guaranteed. Therefore, over a longer period of time many VS COBOL II conversions to other COBOL dialects are necessary.
- We need real source code for our approach to test the extracted grammar. Since VS COBOL II code is around in abundancy, this was not a problem. So we could validate our approach by testing the recovered grammar.
- There is an electronic version of the VS COBOL II manuals [92, 93]. So in some way we have access to grammar knowledge.
- The syntax diagrams in the VS COBOL II manual are among the hardest encodings of grammar rules that we have come across.
- Finally, for this case study we could publish the resulting grammar giving ample evidence of our approach. In other case studies this was not possible due to competitive reasons.

This unique blend of reasons makes the VS COBOL II example the ideal case study. First of all we contributed significantly to the urgent problems of software renovation, and secondly it shows that other (easier) grammar recovery problems can surely be solved in an effective manner.

Copyright © 2001 John Wiley & Sons, Ltd.

Prepared using speauth.cls

The most prominent parts

The case study contains a lot of details so that others can use our work fruitfully, and to provide evidence that the actual solution is not too hard. Moreover the problems appear to be rule-based, so that automation of the approach is feasible. Before we dive into the details of the case study we discuss the prominent parts of the entire process.

Extraction from IBM's reference manual We retrieved an on-line manual for VS COBOL II from the IBM BookManager BookServer Library [62]. 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 [94]. We transformed the parsed diagrams into (extended) Backus Naur Forms (BNF) [95].

Connecting the raw grammar We assessed the resulting BNF code for connectivity (as proposed in [50, 27, 51]). We found more errors than one would expect from a language reference manual. However most of the errors were of a type that is repairable. Namely, the majority of the problems did not reside in the concrete syntax of VS COBOL II, 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 connectivity errors. The formal transformations are a means to record the modifications to the grammar and to reason about correctness of the ensuing grammar rules. Some of the transformation operators which we use in grammar recovery have been formalized [96]. It is convenient to record all the transformation steps. Many errors and problems are reiterated in IBM manuals, because many syntax diagrams are copied from earlier versions and pasted into the newer manuals. In this way we could reuse the sometimes highly specialized transformations fruitfully. Also, we sometimes needed to cancel some earlier transformation steps, and then we could simply replay all the remaining transformations.

Within two days, the VS COBOL II grammar was entirely connected. This means, that all nonterminals 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 nonterminals 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. We had domain knowledge on COBOL grammars so uninitiated implementors may need more time for defining the lexicals and connecting the grammar.

Test-driven correction 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. In the first two days there were still a lot of problems. We could not parse more than five 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 textual part of the manual. In other cases the problems were that the structural information was present but erroneous. From the third day on we could parse about 20 programs a day. The problems we obtained in that phase were due to typing errors in the

Copyright © 2001 John Wiley & Sons, Ltd.

Prepared using speauth.cls

syntax diagrams, and other small issues. After another day of dealing with these issues, the amount of work to be done 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 [97]) could be expressed in both the BNF and the syntax diagrams, but in all other cases there was no intrinsic reason why the syntax diagrams were too restrictive. We located such problems by parsing code. We could have been able to inspect the grammar ourselves using our domain knowledge to find a number of errors, but the parsing approach is more systematic, self-checking, and faster.

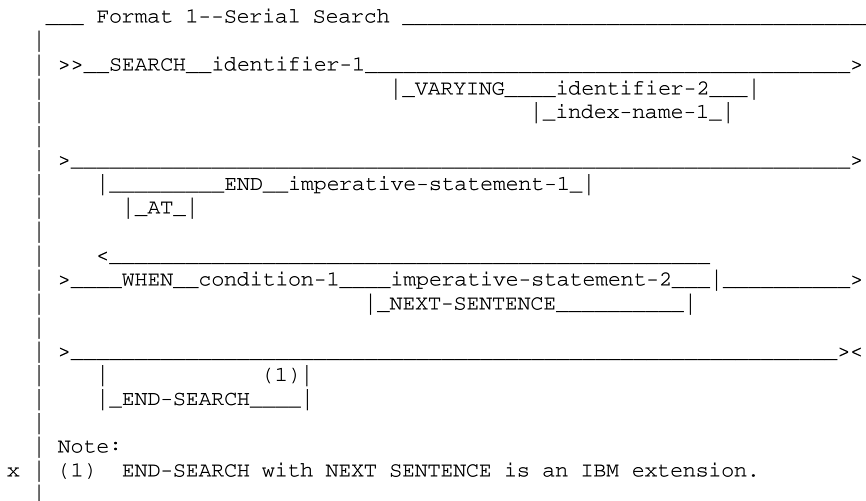
Convergence Noteworthy, perhaps, is that the more systems we parsed, the less work we had to do on correcting errors. To the uninitiated reader this might seem obvious. However, in the everyday practice of software renovation this is unusual. For instance, the phrase *grammar reengineering* was used in [45] 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 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. The usual way to implement parsers is to use mainstream technology like Lex and Yacc and make sure that a certain project parses. For the next project this has to be done again, and after a few projects the grammar suffers from patchwork incrementalism. That is where the difficulties start: solving shift/reduce and reduce/reduce conflicts then takes about 70% of the time of grammar developers. More information on the problems with mainstream parser technology as applied to software renovation can be found in references [45, 47].

Knowing this *grammar molasses phenomenon*, we were surprised that the amount of effort decreased so rapidly while the amount of code that we could parse increased. After a week, we encountered hardly any errors anymore. 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 up to an unmaintainable situation.

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

Copyright © 2001 John Wiley & Sons, Ltd.

```
3.30 SEARCH Statement
```



```
 ___ Format 1--Serial Search ____________________________________
|                                                                  |
| >>__SEARCH__identifier-1______________________________________> |
|                           |_VARYING____identifier-2___|          |
|                                      |_index-name-1_|            |
|                                                                  |
| >_____________________________________________________________> |
|    |_________END__imperative-statement-1_|                       |
|      |_AT_|                                                      |
|                                                                  |
|    <___________________________________________                 |
| >____WHEN__condition-1____imperative-statement-2___|__________> |
|                         |_NEXT-SENTENCE__________|               |
|                                                                  |
| >____________________________________________________________>< |
|    |            (1)|                                             |
|    |_END-SEARCH____|                                             |
|                                                                  |
| Note:                                                            |
x | (1)  END-SEARCH with NEXT SENTENCE is an IBM extension.         |
|__________________________________________________________________|
```

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

Parsing the Diagrams

Let us consider one of the diagrams of the VS COBOL II manual, 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 highlight some issues to give the reader an idea. One peculiarity is that there is no layout, that is, even in the parsing-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 position information of the tokens is significant for parsing, whereas for many languages the row and column information is not at all significant. Finally, the grammar underlying our diagram parser is not a context-free one. Instead, we use an *attributed multiset grammar* [98]. The diagrams are conceived as *sets* of tokens (as opposed to *sequences*) such as <_______ and `WHEN` in Figure 2. The tokens are associated with geometric attributes so that tokens can be composed to phrases like iterative constructs or stacks of alternatives while depending on spatial information. Since the syntax diagrams are made by hand, there is a lot of variation: each manual has its own dialect. It takes about a day of effort to write a syntax diagram parser from scratch, and for each dialect add a few hours extra. In a separate paper we provide details on parsing of visual languages such as these diagrams, since for this paper a full treatise of this subject would be out of scope [94]. For now it is important to know that the total effort for the entire project, including writing several diagram parsers took us two weeks.

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

Copyright © 2001 John Wiley & Sons, Ltd.

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.

Deriving the Raw Grammar

Once the syntax diagrams parse, we can derive BNF rules. 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 typographical errors and in textual BNF. We recall that we can already use level 1 grammars to implement system understanding tools and island parsers. See [28, 29] for level 1 grammars for Pl/I and COBOL 2000.

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 nonterminal which is intended to be defined by the diagram. Our heuristic was the rule to use the name of the section header as a nonterminal. In the syntax diagram depicted in Figure 2, we recall that the header is:

```
3.30 SEARCH Statement
```

We used this information and created the nonterminal `search-statement` in the above BNF version of the syntax diagram. 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 nonterminals. For instance, the syntax diagram presenting the `EXIT PROGRAM` syntax, does not start with a nonterminal, but with the keyword `EXIT PROGRAM` itself. Since a keyword is not a nonterminal, we constructed an overriding transformation solving this, and introduced a nonterminal (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 [28], we found that the section headers in IBM's PL/I language reference manual [60] were mostly not useful. So more overriding transformations were necessary to deal with this problem. Actually, we were able to define a few heuristics to derive many useful section headers by schemata. Recovery did not become harder.

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 nonterminals that are used but not defined—bottom sorts—and the nonterminals that are defined but never used—top sorts. These quality indicators have been originally identified in [50, 27, 51]. A formal discussion of the interaction between top sorts and grammar

Copyright © 2001 John Wiley & Sons, Ltd.

Table I. Levels of grammar recovery for VS COBOL II.

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

transformations has recently been worked out in [96]. 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 I). From the data, we could immediately conclude that the extracted grammar was rather unconnected. This is not too much of a surprise, since the diagrams were not containing consistent sort names of the constructs that were being described. Also certain haplographies and dittographies were hidden in the grammar rules. A haplography is an accidental omission of letters, words or lines. A dittography is the opposite: an accidental repetition of letters, words or lines. For example, the phrase PIN Number expands to using the word *number* twice: a dittography, which is incorrect, but a common error. Both haplographies and dittographies are common in hand-written grammar rules and have to be weeded out to obtain maximal connectivity.

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 or obvious incompleteness problems, we could easily reduce the number of top sorts to 2 and the bottoms sorts to 17. The remaining bottom sorts were identified as lexical sorts. The two top sorts were the start symbols of the grammar: one to parse COBOL constructs and one for the compiler directives. 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. There are just no loose ends.

We summarized the differences between the various levels of our VS COBOL II grammar in Table I. We note that in the raw extracted grammar we had 166 grammar rules and 168 sorts but after our grammar 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. 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 I (we come back to this table later on).

Copyright © 2001 John Wiley & Sons, Ltd.

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 with a grammar 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 corresponds to the language of regular expressions as used for Lex [7]. 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 sensible top sorts (see Table I).

The Error-correction Parser

A level 3 grammar can be used as input to generate a lexer and a parser. Not to generate an efficient production-oriented parser, but a parser meant to detect errors in the grammar specification. Although our error-correction parser's performance was slow as a parser, it was highly effective in detecting errors in the grammar. For, suppose we would try to use an efficient parser at this point, say a Yacc-like tool. Then we would have to wade through a truckload of shift/reduce, reduce/reduce conflicts at compile time, repair them one by one, only to find out when we parse source code that a grammar production is incorrect. Therefore, it is efficient to use a parser that accepts general context-free grammars, so that errors in the productions can be traced immediately.

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 the next section. 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 [16]). For now, we are only interested in an error correction 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 [99] (we used this one) or PRECC [100].

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 only then we deal with ambiguities. In principle, ambiguities

Copyright © 2001 John Wiley & Sons, Ltd.

can cause some errors to go unnoticed because parts of the input might be parsed in an unintended manner. However, even with the ambiguous parser, we reveal so many problems that this minor issue is not relevant in this phase. We are going to disambiguate the grammar in the next section and then the remaining errors that we 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. To parse real COBOL code, we reused a simple preprocessor that prepares the programs for a parser (a small program making the programs more context-free). This preprocessor has been discussed elsewhere [38]. After preprocessing, the code is amenable to parsing.

Detecting Errors

With the working parser, we were able to detect many semantic errors in the level 3 grammar. With semantic errors we mean that the parser recognizes a different language than intended. As an aside, it is common to say that the semantics of a grammar is the language (say, the set of terminal strings) generated by the grammar. Let us give an example of a semantic error.

```
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` (note the dash). But in the footnote of the syntax diagram, there is no 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 grammar transformation solving exactly this issue. This type of error seems trivial, but it is only systematically detectable using a serious test: namely by actually parsing a lot of code. The ambiguity of the generated parser is not at all an impediment in finding (most) errors, which is our present goal. Incidentally, this clarifies the decrease by one keyword at level 4 in Table I: `NEXT-SENTENCE` is gone now.

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.
```

Copyright © 2001 John Wiley & Sons, Ltd.

Prepared using speauth.cls

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 [92] this confusion is not resolved. However in the application programming guide [93]—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 can 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"]
```

A definition for `statement-list` had to be provided, clarifying an increase in rules and sorts in Table I at level 4. We took care of that using 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, after which we could parse all our VS COBOL II code without errors. After we successfully parsed about two million lines of code that originated from various companies, from various countries, and from various continents, we called this grammar a *level 4* grammar. Recall, that we now have a recovered grammar specification, but that the specification itself still is ambiguous. But it is already very valuable: many people use this specification to write COBOL parsers, some for compilers and some for renovation tools.

A note on correctness 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. Another indication for correctness is that the amount of effort decreased sharply while the amount of code that we could parse increased drastically. From a different perspective pursued for example in the logic programming community [101, 102], 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 [103]. Thereby, correctness means that the parser does not accept more programs than intended based on a coverage argument. We worked out these intuitions in [104] in a formal manner. We currently work on the employment of formal correctness and completeness criteria for the assessment of realistic grammars, and the scalable implementation of the required testing technology.

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.

Copyright © 2001 John Wiley & Sons, Ltd.

Prepared using speauth.cls

We delivered a level 5 grammar to Ericsson [27]. 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 a feature 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.

For convenience's sake we enumerate the definitions of the level 1–5 grammars.

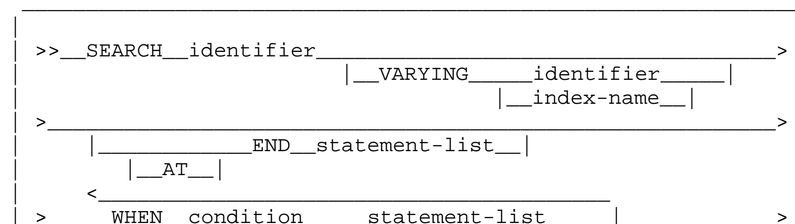
- A level 1 grammar is a raw extracted grammar in electronic format that is void of typographical errors represented in textual BNF. The BNF itself is parsed by a BNF-parser.
- A level 2 grammar is a level 1 grammar that is maximally connected, i.e., that does not contain unwanted bottom and top sorts.
- A level 3 grammar is a level 2 grammar that also has its lexicals recovered.
- A level 4 grammar is a tested level 3 grammar. With tested we mean the range of millions lines of code originating from various companies, various countries, and from various continents.
- A level 5 grammar is a grammar that has been directly extracted from the compiler source code.

Level 1–3 grammars are useful to derive system understanding tools and island parsers. From level 4 and level 5 grammars it is possible to derive realistic parsers for system modification tools. We separate grammars or grammar *specifications* from parsers or grammar *implementations*. In the above list of levels, we focus on grammar specifications. From grammar specifications of various levels it is feasible to derive grammar implementations of various use.

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:

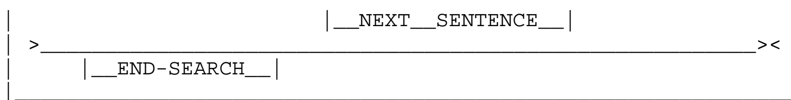
`search-statement`



```
 ____________________________________________________________
|                                                            |
| >>__SEARCH__identifier____________________________________> |
|                        |__VARYING_____identifier_____|      |
|                                    |__index-name__|         |
| >_________________________________________________________> |
|     |_____________END__statement-list__|                    |
|        |__AT__|                                             |
|     <________________________________________               |
| >_____WHEN__condition_____statement-list_____|____________> |
```

Copyright © 2001 John Wiley & Sons, Ltd.

Prepared using speauth.cls



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 possible applications of our work. People from the standards community 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 an excellent idea to use techniques similar to ours to deal in an accurate manner with language standards. At this moment language reference standards are written using word processors like Microsoft Word as opposed to more sophisticated tools to generate them from tested executable formal specifications. 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 mostly due to the lack of grammar engineering 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* [105]. 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 [25] is at the moment the most authoritative source for what syntax is accepted by their VS COBOL II product. It is clear that our work is direclty applicable to almost all their other language reference manuals.

Generating On-line Manuals

It is not hard to 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 [27] for a proprietary language of Ericsson. The hypertext version of our recovered VS COBOL II grammar is made available to the general public [25]. We recall that we were the first to make such 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

Copyright © 2001 John Wiley & Sons, Ltd.

Prepared using speauth.cls

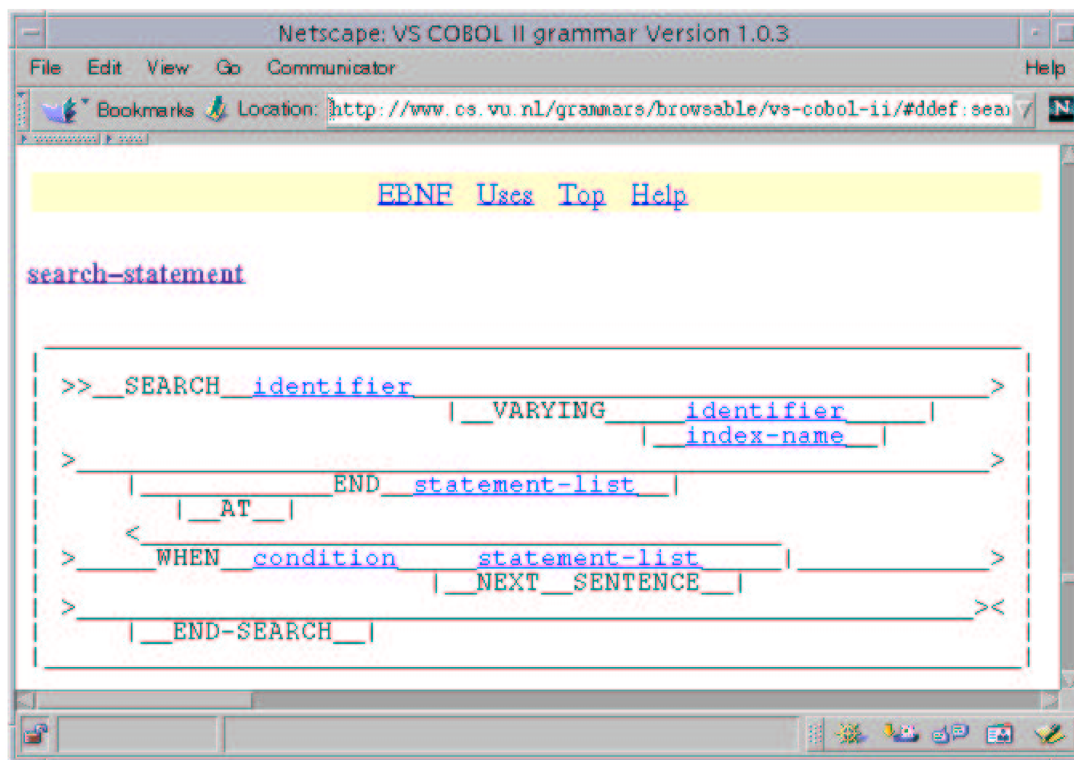


Figure 3. A dump of the actual HTML page containing the recovered VS COBOL II grammar specification.

good news: our work discloses important information for the software renovation area, but also in the compiler construction area. For instance, both GNU COBOL compiler initiatives [30] use our grammar specification to derive realistic COBOL parsers. Those users also debug our specifications, we receive suggestions for corrections, or requests for improving its presentation [106].

We provide a screen dump of the actual HTML page that we generated with our tools. In Figure 3 we show the `SEARCH` statement again. This page contains many more features than just the diagrams. The page starts with a summary comparable to Table I. 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 in the figure. Moreover, all cross-links that one could wish for are present so that navigating in the document is 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. At the time of writing this document, our VS COBOL II grammar is the most authoritative source on the Internet: it is often accessed, it is often linked or even mirrored, etc.

Copyright © 2001 John Wiley & Sons, Ltd.
Prepared using speauth.cls

Summary of the Kinds of Errors

We collected examples of the most prominent types of errors. 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, etc.
- Semantic typographical errors in syntax diagrams. Think of the `NEXT-SENTENCE` error. A diagram containing that kind of error 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 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 [60]. The example is concerned with stacking alternatives. Stacked alternatives are explained in the manual to be aligned. The manual contains the following illustration (explaining stacked alternatives).

```
 ___ Format ____________________________________________________
|                                                               |
| >>__STATEMENT____required choice 1__________________________>< |
|                |_required choice 2_|                          |
|_______________________________________________________________|
```

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

```
 _________________________________________________________________
|                                                                 |
| >>____________arithmetic-constant____________________________>< |
|      |__+__|                                              |     |
|      |__-__|                                              |     |
|      |________real-constant_____+_____imaginary-constant__|     |
|      |__+__|                   |__-__|                    |     |
|      |__-__|                                              |     |
|      |_character-constant_________________________________|     |
|      |_bit-constant_______________________________________|     |
|      |_graphic-constant __________________________________|     |
|_________________________________________________________________|
```

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 [60] which—besides the fact that it contains several syntax errors—uses a style entirely different from the rest of the document.

Copyright © 2001 John Wiley & Sons, Ltd.

- Incompleteness of the grammar specification as discussed earlier. The fact that a grammar is not complete or 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 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.
- 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`, resulting in a dittography.

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, and additionally we were able to generate corrected manuals which closed the loop. As was stated earlier, in software renovation 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. Others have communicated this to us as well. Now that we have solved a myriad of plain errors in the specification, we are confident that we have a firmly established specification that we can further process, which is the subject of the next section.

IMPLEMENTATION OF THE GRAMMAR SPECIFICATION

At this point we have recovered the grammar specification of a language from its manual. 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. We now take the work one step further: we transform 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 towards a realistic parser.

We start with choosing another parser generator more suited for realistic parsing. After all, we are now no longer interested in finding errors, but in efficient parsing and generating ASTs. We also need to implement the lexicals. Then we start with the elimination of ambiguities. Finally, we give pointers

Copyright © 2001 John Wiley & Sons, Ltd.

Prepared using speauth.cls

how to convert from the parser technology that we use to mainstream parser technology like Lex and Yacc.

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 [107, 108, 109, 110, 111]. The technology is suited for reporting ambiguities, which is what we want to work on in this phase. Also here, we proceed 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 like dealing with shift/reduce conflicts (see [112] for details).

The SGLR parser generator accepts grammar rules in SDF format (Syntax Definition Formalism [113, 110]). So with a 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
```

This particular piece of code is just different syntax than we had before. This syntax swap is not at all a problem. The real work is the disambiguation.

Implementing the Lexical definition

There is another 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 *follow-restriction* simulating longest match of lexicals:

Copyright © 2001 John Wiley & Sons, Ltd.

```
   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 [110], 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. Most of the times, such constraints are not formally expressed in a reference manual but only in informal text or in a table. For instance, *some* keywords are reserved, and others not. There might be even keywords which are reserved but which do not occur in the grammar at all. In PL/I, certain keywords can also be used as identifiers. Further problems arise from the use of embedded languages (such as SQL or CICS). Then, one might need to consider certain tokens as reserved in only part of the merged grammar. In the case of VS COBOL II, there are words which where reserved by CODASYL for future development. In the COBOL grammar specification we abstracted from these issues. 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` 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 other reserved 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:

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

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

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 this section we like to point out that the disambiguation process is not hard. But there are a large number of disambiguation issues, so we deal with a few different categories to prevent adopters of our approach from surprises.

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 helpful to locate what is going on: navigation support immediately reveals the problems. 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__|                                               |
|_________________________________________________________________|
```

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`. The definitions of those two 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. For a parser this makes no difference because a string is a string. 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 affect the language generated by 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
```

We give another example to show that ambiguities can be traced systematically and can be resolved easily. In the `DATA DIVISION` we encountered an ambiguity related to the distinction of records

Copyright © 2001 John Wiley & Sons, Ltd.

Prepared using speauth.cls

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 this 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 are only finitely many level numbers, but the resulting grammar would become extremely complex. To find the exact location of the ambiguity in the grammar we browsed through the grammar specification and found in the `data-division-content` the following syntax diagram snippet:

```
>__LINKAGE__SECTION__.________________________________________________><
                      |   <____________________________________    |
                      |_______record-description-entry________|__|
                           |__data-item-description-entry__|
```

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 grammar 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 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 (notation `[..]`) the irrelevant parts for clarity:

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

Copyright © 2001 John Wiley & Sons, Ltd.

Or it can end as follows:

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

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. 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 called 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 resolve to the same sort, the above stack of 3 possibilities leads to an ambiguity if the ADDRESS OF keywords are 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 transform the grammar fragment:

Copyright © 2001 John Wiley & Sons, Ltd.

Prepared using speauth.cls

```
[..] (identifier | "ADDRESS" "OF" identifier | file-name) [..]
```

into this fragment:

```
[..] (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. This solution is somewhat clumsy, and one might indeed argue that there are other possible approaches. It is for example very common to tweak scanners and parser via semantic information [114, 100]. In the above example, we might employ a symbol table to separate file names and identifiers. Instead of tweaking scanners and parser we could also delay disambiguation and wait until we can use type checking to build the final parse tree, for example, in the style of *disambiguation filters* [115]. This example illustrates that there is sometimes more than one option how to disambiguate. The advantage of separating grammar specification from 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 (represented by the outer `{...}+`) or by iteration of parameters after the possibly empty `BY REFERENCE` phrase (the inner `{...}+`). We remove this second choice in order to disambiguate. As a consequence, we enforce iteration over just parameters. 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
```

Copyright © 2001 John Wiley & Sons, Ltd.
Prepared using speauth.cls

```
   "." 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 for 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 rule:

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

Enforcing Priorities

Some ambiguities are more conveniently addressed using a priority mechanism rather than grammar 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
```

This 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 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. The disambiguation is done back-to-back with extensive tests using the two million lines of 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 written language reference manuals containing grammar information in some format.

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

Copyright © 2001 John Wiley & Sons, Ltd.
Prepared using speauth.cls

was indicated earlier in the paper. For that purpose, it is convenient to use grammars with explicit list-constructs so that we can use those list constructs in patterns [40] (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 [42, 27, 33, 39]. Also from a parsing perspective, the paper [45] discusses why GLR is 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 (denoted `*` and `+`) and optionals (notation `?`). The wide-spread class of LR parser generators does not allow for such constructs to be present. This implies that with grammar transformations the list-constructs and optionals have to be modeled via auxiliary nonterminals. Moreover the problem of shift/reduce conflicts needs to be addressed. One can use the term *Yaccification* to denote the corresponding kind of grammar adaptation. In the VS COBOL II case study, we did not aim at an operational LALR(1) parser. 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 an accurate and executable requirements specification.

In the RainCode system [46], in Software Refinery [116], and in a paper on optimizing a GLR algorithm [117], methods are proposed to turn grammar specifications that contain the list constructions `+` and `*` into recursive rules that can be handled by LR parser generators like Yacc.

Let us discuss the cases in a little more detail. The RainCode approach aims 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. The paper [117] presents a similar approach, and constructions that are potentially performance bottlenecks 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.

POSSIBLE IMPLEMENTATION PLATFORMS

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. We used tools to support the following aspects of grammar engineering with emphasis on grammar recovery:

- raw grammar extraction;

Copyright © 2001 John Wiley & Sons, Ltd.
Prepared using speauth.cls

- generation of browsable grammars;
- grammar transformation;
- error detection by parsing;
- derivation of parser specification;
- resolution of ambiguities;
- efficient parsing.

For the case study discussed in this paper we used the system LDL [99] for the first five aspects, whereas the ASF+SDF Meta-Environment [78] was used for the rest, i.e., realistic parsing including resolution of ambiguities. In the papers [50, 27, 51], we solely used the ASF+SDF Meta-Environment as implementation platform for grammar recovery. Recently we also employed the ASF+SDF Meta-Environment to create a general framework for grammar transformations [118]. This framework can be used for grammar recovery, grammar re-engineering, and also for Yaccification. For convenience, we briefly characterize LDL and the ASF+SDF Meta-Environment.

LDL is short for Language Development Laboratory. It is a specification framework for prototyping language tools. It has been derived in the LDL project [119, 120, 99] 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 [121, 122]. We refer to [123] 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 [113] for a reference manual, and [110] for a recent version of SDF). ASF stands for algebraic specification formalism, and is used to describe semantics (see [124] for a textbook on this subject). For a comparison between the ASF+SDF Meta-Environment and related systems we refer to [39].

Other systems There are many other systems capable of conveniently implementing 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 tool applicable to extract grammars 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 [94]. For more involved visual notation and/or proper graphical encoding, visual language tools might be needed [125, 126, 127]. 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. The BNF Web Club [63] presents browsable grammars for several languages with linked grammar rules and syntax diagrams based on HTML +

Copyright © 2001 John Wiley & Sons, Ltd.
Prepared using speauth.cls

Java for layouting the diagrams [128]. For the diagrams in our browsable grammars we used a pretty printer relying on ASCII-encoding.

There are many systems facilitating transformational programming as required for grammar manipulation. We mention Software Refinery [129], Cocktail [130, 131], ToolMaker [132], Stratego [133], TXL [5], TAMPR [134], Eli [135], Gandalf [136], Elegant [137], RainCode [138], COSMOS [139], and so on. Note that these systems do not routinely contain or employ grammar engineering tools, but are platforms easily adaptable to support grammar engineering with tools.

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. Then a disambiguation parser, and 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 and disambiguation phases because we would like to obtain a proper grammar specification which is not tuned towards any grammar implementation criterion. Top-down parsing with backtracking as facilitated by LDL or PRECC [100], or generalized LR parsing [107, 108, 109, 110, 111] is convenient in this context.

To summarize, there are no restrictions on what technology to use, so it is not necessary to have knowledge of LDL or the ASF+SDF Meta-Environment to solve the problems of grammar recovery. In one project we used the ASF+SDF Meta-Environment, and in this project we started off with LDL. Other tools could have been used instead. In fact, it is possible to use any kind of program transformation tool, lexer/parser generator for a grammar engineering project to quickly produce parsers and other grammar-based tools.

ACKNOWLEDGEMENTS

We thank Peter Aarnoutse, Prem Devanbu, Cordell Green, Capers Jones, Jan Kort, Tom McCabe, Harry Sneed, and the anonymous reviewers for their encouragement, code samples, and significant contributions. We thank David Essex, Rasmus Faust, William Klein and Bob Rayhawk for analyzing and using our VS COBOL II grammar, and pointing out occasional errors.

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, redocumentation projects, system comprehension tasks, operating systems migrations, software merging projects due to company merges, web-enabling of mainframe systems, daily large scale 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 solutions in linguistics do 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.

Copyright © 2001 John Wiley & Sons, Ltd.

The results that were achieved by these efforts are in our opinion convincing and lead us to believe that in principle the 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 two million lines of code. Furthermore, we were able to implement a realistic parser from the specification. In addition we applied this approach to a host of other languages with the same results. The tools necessary for our approach are simple and can be implemented using any kind of compiler kit, program transformation system, or language workbench.

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 expect that the techniques we developed will find their way into 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. C. Jones. *Estimating Software Costs*. McGraw-Hill, 1998.
2. G.M. Weinberg. *The Psychology of Computer Programming*. Van Nostrand Reinhold, 1971.
3. J. Rumbaugh, M.H. Blaha, W. Premerlani, F. Eddy, and W. Lorenson. *Object-Oriented Modeling and Design*. Prentice Hall, 1991.
4. J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Object Technology Series. Addison-Wesley, 1998.
5. 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.
6. 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.
7. M.E. Lesk and E. Schmidt. *LEX - A lexical analyzer generator*. Bell Laboratories, UNIX Programmer's Supplementary Documents, volume 1 (PS1) edition, 1986.
8. S.C. Johnson. YACC - Yet Another Compiler-Compiler. Technical Report Computer Science No. 32, Bell Laboratories, Murray Hill, New Jersey, 1975.
9. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
10. M.H. Blaha and W. Premerlani. *Object-Oriented Modeling and Design for Database Applications*. Prentice Hall, 1998.
11. T. Bray, J. Paoli, and C.M. Sperberg-McQueen. Extensible Markup Language (XML) 1.0, February 1998. W3C Recommendation, WWW Consortium, http://www.w3.org/.
12. XML Schema Part 0: Primer, October 2000. `http://www.w3.org/TR/xmlschema-0/`.
13. XML Schema Part 1: Structures, May 2001. `http://www.w3.org/TR/xmlschema-1/`.
14. XML Schema Part 2: Datatypes, May 2001. `http://www.w3.org/TR/xmlschema-2/`.
15. International Telecommunication Union. *Recommendation Z.200 (11/93) - CCITT High Level Language (CHILL)*, 1993.
16. J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1996.
17. J.B. Warmer and A.G. Kleppe. *The Object Constraint Language – Precise Modeling With UML*. Object Technology Series. Addison-Wesley, 1999.
18. C. Jones. *Applied Software Measurement: Assuring Productivity and Quality*. McGraw-Hill, second edition, 1996.
19. W.M. Ulrich. The evolutionary growth of software reengineering and the decade ahead. *American Programmer*, 3(11):14–20, 1990.
20. C. Jones. *Software Productivity and Quality – The World Wide Perspective*. IS Management Group, Carlsbad, CA, 1993.
21. C. Jones. *Assessment and Control of Software Risks*. Prentice-Hall, 1994.
22. B. Hall. Year 2000 tools and services. In *Symposium/ITxpo 96, The IT revolution continues: managing diversity in the 21st century*. GartnerGroup, 1996.
23. N. Jones. Year 2000 market overview. Technical report, GartnerGroup, Stamford, CT, USA, 1998.
24. V. Maslov. Re: An odd grammar question, 1998. Retrieved via: http://www.iecc.com/comparch/article/98-05-108.

Copyright © 2001 John Wiley & Sons, Ltd.

Prepared using speauth.cls

25. R. Lämmel and C. Verhoef. *VS COBOL II grammar Version 1.0.3*, 1999. Available at: http://www.cs.vu.nl/grammars/browsable/vs-cobol-ii/.
26. T.B. Dinesh, M. Haveraaen, and J. Heering. An algebraic programming style for numerical software and its optimization. Technical Report SEN-R9844, CWI, Amsterdam, 1998. CoRR Preprint Server xxx.lanl.gov/abs/cs.SE/9903002; to appear in *Scientific Programming*.
27. 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://www.cs.vu.nl/˜x/com/com.html.
28. R. Lämmel and C. Verhoef. *OS PL/I V2R3 grammar Version 0.1*, 1999. Available at: http://www.cs.vu.nl/grammars/browsable/os-pli-v2r3/.
29. R. Lämmel and C. Verhoef. *COBOL 2000 grammar Version 0.1.1*, 1999. Available at: http://www.cs.vu.nl/grammars/browsable/cobol/.
30. K. Rayhawk. Re: gnubol: How do we parse this language, anyway?, 1999. Retrieved via: http://www.lusars.net/maillists/gnu-cobol/msg00836.html.
31. D.S. Wile. Integrating Syntaxes and their Associated Semantics. Draft, 1999.
32. D.S. Wile. Abstract syntax from concrete syntax. In *Proceedings of the 1997 International Conference on Software Engineering*, pages 472–480. ACM Press, 1997.
33. 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://www.cs.vu.nl/˜x/scaf/scaf.html.
34. J. Brunekreef and B. Diertens. Towards a user-controlled software renovation factory. In P. Nesi and C. Verhoef, editors, *Proceedings of the Third European Conference on Maintenance and Reengineering*, pages 83–90. IEEE Computer Society Press, 1999.
35. 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.
36. 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.
37. 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://www.cs.vu.nl/˜x/sigplan/plan.html.
38. 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://www.cs.vu.nl/˜x/coboldef/coboldef.html.
39. 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://www.cs.vu.nl/˜x/scp/scp.html. An extended abstract with the same title appeared earlier: [140].
40. 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://www.cs.vu.nl/˜x/npl/npl.html.
41. 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://www.cs.vu.nl/˜x/evol-se/evol-se.html.
42. 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://www.cs.vu.nl/˜x/asm/asm.html.
43. 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://www.cs.vu.nl/˜x/sale/sale.html.
44. C. Verhoef. Towards Automated Modification of Legacy Assets. *Annals of Software Engineering*, 9:315–336, March 2000. Available at http://www.cs.vu.nl/˜x/ase/ase.html.
45. 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://www.cs.vu.nl/˜x/ref/ref.html.
46. D. Blasband. *Automatic Analysis of Ancient Languages*. PhD thesis, Free University of Brussels, 2000. Available via http://www.phidani.be/homes/darius/thesis.html.
47. D. Blasband. Parsing in a hostile world. In P. Aiken, E. Burd, and R. Koschke, editors, *Proceedings Working Conference on Reverse Engineering; WCRE'01*. IEEE Computer Society, 2001. To appear.

Copyright © 2001 John Wiley & Sons, Ltd.
Prepared using speauth.cls

48. M. de Jonge and R. Monajemi. Cost-effective maintenance tools for proprietary languages. In G. Canfora and A.A. Andrews, editors, *International Conference on Software Maintenance, ICSM2001*. IEEE Computer Society, 2001. To appear.
49. Merijn de Jonge, Eelco Visser, and Joost Visser. Xt: a bundle of program transformation tools. In Mark van den Brand and Didier Parigot, editors, *Electronic Notes in Theoretical Computer Science*, volume 44. Elsevier Science Publishers, 2001.
50. 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 [51]. Available at: http://www.cs.vu.nl/~x/ase/ase.html.
51. 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 [50]. Available at: http://www.cs.vu.nl/~x/cale/cale.html.
52. 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.
53. 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.
54. 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.
55. C. Verhoef. Re: How to extract grammar from a program?, 1999. Retrieved via: http://www.iecc.com/comparch/article/99-12-042.
56. J. Heering and P. Klint. Semantics of programming languages: A tool-oriented approach. *ACM SIGPLAN Notices*, March 2000.
57. A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers. Principles, Techniques and Tools*. Addison-Wesley, 1986.
58. 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.
59. M. de Jonge. A Pretty-Printer for Every Occasion. In *Proceedings of the 2nd International Symposium on Constructing Software Engineering Tools (CoSET2000)*, pages 68–77. University of Wollongong, Australia, 2000.
60. IBM Corporation. *IBM PL/I for VSE/ESA(TM) Programming Guide*, 1st edition, 1971, 1998. Publication number SC26-8053-01.
61. G. Falquet, J. Guyot, and L. Nerima. Simple tools to learn Ada. *ACM SIGADA Ada Letters*, 4(6):44–48, May/June 1985.
62. 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.
63. T. Estier and J. Guyot. The BNF Web Club, 1998. Centre Universitaire d'Informatique, Université de Genève, Available at http://cui.unige.ch/db-research/Enseignement/analyseinfo/BNFweb.html.
64. T.J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-12(3):308–320, 1976.
65. 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.
66. O. Stock, R. Falcone, and P. Insinnamo. Island parsing and bidirectional charts. In *Proc. of the 12th COLING*, pages 636–641, Budapest, Hungary, 1988.
67. 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.
68. 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.
69. S.G. Woods, L. O'Brian, T. Lin, K. Gallagher, and A. Quilici. An architecture for interoperable program understanding tools. In S. Tilley and G. Visaggio, editors, *Proceedings of the Sixth International Workshop on Program Comprehension*, pages 54–63, 1998.
70. 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.
71. 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.

Copyright © 2001 John Wiley & Sons, Ltd.

Prepared using speauth.cls

72. 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.
73. 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.
74. 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.
75. J.A. Bergstra and P. Klint. The discrete time TOOLBUS—a software coordination architecture. *Science of Computer Programming*, 31:205–229, 1998.
76. 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.
77. 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.
78. P. Klint. A meta-environment for generating programming environments. *ACM Transactions on Software Engineering and Methodology*, 2(2):176–201, 1993.
79. 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.
80. T. Despeyroux. Typol: A formalism to implement Natural Semantics. Technical Report 94, INRIA Sophia-Antipolis, 1988.
81. G. Kahn, B. Lang, B. Mélèse, and E. Morcos. Metal: A formalism to specify formalisms. *Science of Computer Programming*, 3:151–188, 1983.
82. 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.
83. 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.
84. B. Boehm. *Software Engineering Economics*. Prentice Hall, 1981.
85. L.H. Putnam and W. Myers. *Measures for Excellence – Reliable Software on Time, Within Budget*. Yourdon Press Computing Series, 1992.
86. S. McConnell. *Rapid Development*. Microsoft Press, 1996.
87. S. Björk. Parsers, Pretty Printers and PolyP. Master's thesis, Göteborg University, 1997.
88. H.M. Sneed. *Objektorientierte Softwaremigration*. Addison-Wesley, 1998. In German.
89. Ovum. Report on the status of programming languages in europe. Technical report, Ovum Ltd., 1997.
90. E. Arranga, I. Archbell, J. Bradley, P. Coker, R. Langer, C. Townsend, and M. Weathley. In Cobol's Defense. *IEEE Software*, 17(2):70–72, 75, 2000.
91. Santa Teresa Laboratory. Name That IBM COBOL Product! *IBM COBOL Newsletter*, 8, 2000.
92. IBM Corporation. *VS COBOL II Reference Summary*, 1.2. Publication number SX26-3721-05 edition, 1993.
93. IBM Corporation. *VS COBOL II Application Programming Language Reference*, 4th edition, 1993. Publication number GC26-4047-07.
94. R. Lämmel and C. Verhoef. Rejuvenation of an Ancient Visual Language. Draft; available at http://www.cwi.nl/˜ralf/; submitted for publication, April 2000.
95. 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.
96. Ralf Lämmel. Grammar Adaptation. In *Proceedings of Formal Methods Europe (FME) 2001*, volume 2021 of *LNCS*, pages 550–570. Springer-Verlag, 2001.
97. R.D. Cameron. Extending context-free grammars with permutation phrases. *ACM Letters on Programming Languages and Systems*, 2(4):85–94, March 1993.
98. S.-K. Chang. Picture Processing Grammar and its Applications. *Information Sciences*, 3:121–148, 1971.
99. 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.
100. 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.
101. 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.

Copyright © 2001 John Wiley & Sons, Ltd.

Prepared using speauth.cls

102. 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.
103. P. Purdom. A sentence generator for testing parsers. *Behaviour and Information Technology*, 12(3):366–375, July 1972.
104. R. Lämmel. Grammar Testing. In *Proceedings of Fundamental Approaches to Software Engineering (FASE) 2001*, number 2029 in LNCS, pages 201–216. Springer-Verlag, 2001.
105. IBM Corporation. *COBOL for OS/390 & VM, COBOL Set for AIX, VisualAge COBOL – Language Reference*, fourth edition, 1998. Publication number: SC26-9046-03.
106. R. Lämmel and C. Verhoef. *ChangeLog for VS COBOL II grammar*, 1999. Available at: http://www.cs.vu.nl/grammars/browsable/vs-cobol-ii/ChangeLog.html.
107. 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.
108. M. Tomita. *Efficient Parsing for Natural Languages—A Fast Algorithm for Practical Systems*. Kluwer Academic Publishers, 1986.
109. J. Rekers. *Parser Generation for Interactive Environments*. PhD thesis, University of Amsterdam, 1992. ftp://ftp.cwi.nl/pub/gipe/reports/Rek92.ps.Z.
110. 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.
111. E. Visser, J. Scheerder, and M. van den Brand. Scannerless Generalized-LR Parsing, 2000. Work in Progress.
112. 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.
113. 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.
114. T.J. Parr and R.W. Quong. ANTLR: A predicated-LL(k) parser generator. *Software—Practice and Experience*, 25(7):789–810, 1995.
115. P. Klint and E. Visser. Using filters for the disambiguation of context-free grammars. Technical Report P9426, Programming Research Group, University of Amsterdam, 1994.
116. Reasoning Systems, Palo Alto, California. *DIALECT user's guide*, 1992.
117. 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.
118. Ralf Lämmel and Guido Wachsmuth. Transformation of SDF syntax definitions in the ASF+SDF Meta-Environment. In Mark van den Brand and Didier Parigot, editors, *Proc. LDTA'01*, volume 44 of *ENTCS*, pages 1–25. Elsevier Science, April 2001.
119. 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.
120. 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.
121. W.F. Clocksin and C.S. Mellish. *Programming in Prolog*. Springer-Verlag, Berlin, 3rd edition, 1987.
122. U. Nilsson and J. Maluszynski. *Logic Programming and Prolog*. John Wiley, 2 edition, 1995.
123. P. Knauber. *Ein System für die Konstruktion objektorienierter Übersetzer*. PhD thesis, Universität Kaiserslautern, 1997. In German.
124. 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.
125. 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.
126. 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.
127. G. Costagliola, G. Tortora, S. Orefice, and A. De Lucia. Automatic Generation of Visual Programming Environments. *Computer*, 28(3):56–66, March 1995.
128. M. Bonjour, G. Falquet, J. Guyot, and A. Le Grand. *Java: de l'esprit á la méthode*. Editions Vuibert, 2nd edition, 1999. In French.
129. Reasoning Systems, Palo Alto, California. *Refine User's Guide*, 1992.

Copyright © 2001 John Wiley & Sons, Ltd.
Prepared using speauth.cls

130. 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.
131. 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.
132. SoftLab, Linköping, Sweden. *ToolMaker Reference Manual*, version 2.0 edition, 1994.
133. 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.
134. J.M. Boyle. A transformational component for programming language grammar. Technical Report ANL-7690, Argonne National Laboratory, Argonne, Illinois, 1970.
135. 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.
136. A.N. Habermann and D. Notkin. Gandalf: Software development environments. *IEEE Transactions on Software Engineering*, SE-12:1117–1127, 1986.
137. Philips Electronics B.V., The Netherlands. *The Elegant Home Page*, 1993. http://www.research.philips.com/generalinfo/special/elegant/elegant.html.
138. RainCode, Brussels, Belgium. *RainCode*, 1.07 edition, 1998. ftp://ftp.raincode.com/cobrc.ps.
139. Emendo Software Group, The Netherlands. *Emendo Y2K White paper*, 1998. Available at http://www.emendo.com/.
140. 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://www.cs.vu.nl/~x/trans/trans.html.

Copyright © 2001 John Wiley & Sons, Ltd.

Prepared using speauth.cls