The Amsterdam toolkit for language archaeology

Ralf Lämmel [1,2]

[1] *Vrije Universiteit, Amsterdam*
[2] *Centrum voor Wiskunde en Informatica, Amsterdam*

Abstract

GRK — the Grammar Recovery Kit — illustrates options for automation and corresponding tool support in the context of developing quality language references that readily cater for the derivation of parsers.

GRK provides the proof-of-concept for two notions: (i) semi-automatic grammar recovery; (ii) language-reference re-engineering. GRK's support for semi-automatic grammar recovery means that GRK can be used to obtain a relatively correct and complete as well as implementable grammar from a language reference. GRK's support for language-reference re-engineering means that GRK can be used to update the original language reference such that it reflects the completed and corrected grammar knowledge.

As of today, GRK is particularly fit for Cobol archaeology, more specifically for IBM's VS Cobol II. That is, GRK offers a fully mechanised process, where IBM's reference is used as an input, and the output is a transformed language reference whose grammar portions are correct and complete. (The recovery required several hundreds of simple transformation steps in order to deliver a grammar that is fit for parser derivation.) As a byproduct, GRK also generates a slow, Prolog-based parser. Via export to GRK's sibling, GDK (the Grammar Deployment Kit), a reasonably fast, `btyacc`-based parser can be generated as well. Both parsers accept all of the VS Cobol II code that is at our avail (several millions of lines of code).

Keywords: Grammar Recovery, Cobol, Grammar Transformations, Visual Language Parsing, Grammarware Engineering, The Grammar Recovery Kit, The Grammar Deployment Kit

1 Getting into grammar recovery mood

(Let's assume that ...) you face the following assignment:

> *Your boss wants you to analyse and transform Cobol programs. You have got a few million lines of Cobol code in front of you, for which you are supposed to demonstrate the feasibility of some fairly simple transformations. The clock is ticking. You are an experienced software engineer who is fluent in language processing matters. So you are not scared off, and you plan to work on a Cobol parser. The requested transformations are trivially implemented — if you have only got a front-end working. Been there, done that. Not for Cobol! You try googling first.* [1] *There isn't much! You can't find a* `yacc` *specification for Cobol. Why is that? Cobol has been around since the early 1960's! Someone must have thought of this! You download some packages that seem to deal with Cobol somehow. You spend one week on one such package only to find out that the Cobol grammar has never been finished. Also, where is the pre-processor? Where is the embedded SQL support? How to do develop software transformations with this package? You spend one month on another package — this time a research prototype. You can't get it fully working because it makes some mind-boggling assumptions. The disclaimers should have put you off earlier. You start panicking. You call your friend who works for* << a vendor of a Cobol compiler >>. *He says that you must refuse the assignment because you only have five years left until retirement, and there is no way that anyone gets this damn thing done in this short time. Your boss refuses the refusal, and the deadline is fixed: you got three more weeks. Mission impossible?* ◇

This scenario instantiates an important and difficult problem: *the provision of tool support for automated software analysis and transformation normally requires quality grammar knowledge that caters for parser development for the language(s) at hand.* We have coined this prevalent problem as the "500-language problem" elsewhere [12]. GRK — the Grammar Recovery Kit is a *software toolkit* that helps in this context. GRK solves the specific Cobol assignment, and hints at a general solution to the 500-language problem.

2 So what's GRK?

GRK is a software toolkit for *semi-automatic grammar recovery* [13] and *language-reference re-engineering*. The basic idea of semi-automatic grammar recovery is to support the derivation of a relatively correct and complete as well as implementable grammar by *extracting* raw pieces of syntax description from a language reference, and *improving* these pieces as necessary. The basic idea of language-reference re-engineering is to support the evolution of language references in terms of revisions and extensions that concern the embedded grammar knowledge, but also other aspects of the references.

[1] We are back in history, before 3 December 1999.
(This is when we published the first quality Cobol grammar [11] on the web.)

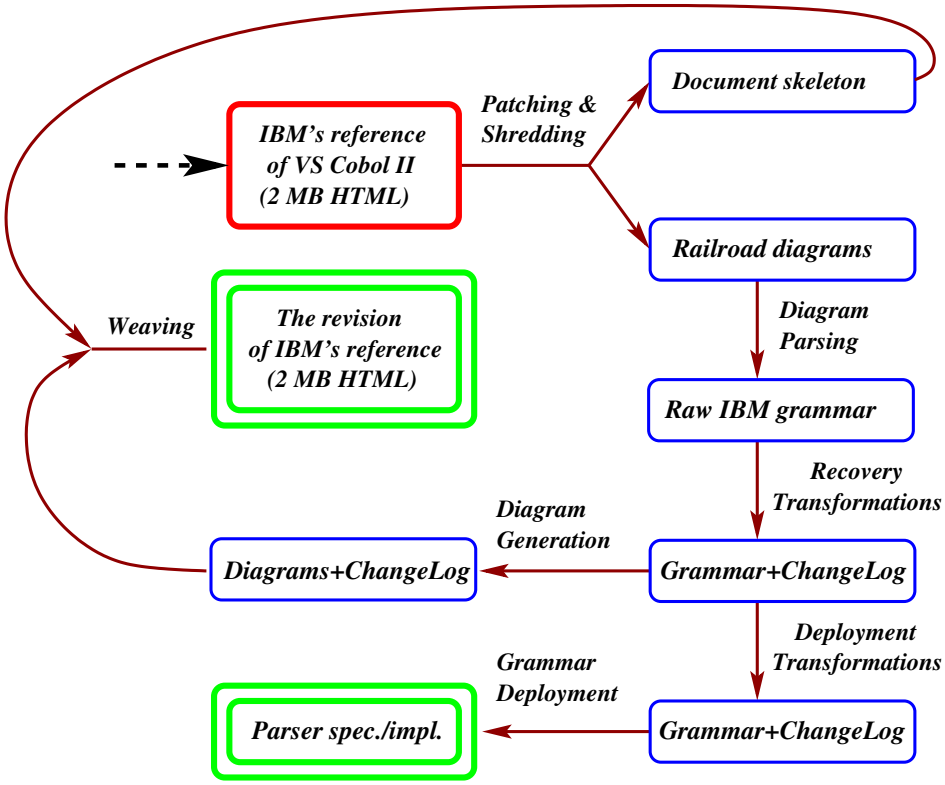

Fig. 1. The recovery case for IBM's VS Cobol II

We note that the overall process for grammar recovery is motivated and described in detail in [13]. In the present publication, we focus on GRK's mechanised process. As of today, GRK has been driven far enough to provide the publicly accessible, constructive proof of the feasibility of language archaeology. Our previous publications [13,12] relied on ad-hoc tools that we didn't dare to distribute. In particular, GRK has been worked out to the extent needed for a case study for Cobol, more specifically for VS Cobol II. The process for the recovery of the VS Cobol II grammar [11] and for the revision of IBM's application programming reference [5] is summarised in Fig. 1.

To give the reader an idea, let us re-execute this process. To this end, we assume that GRK is readily downloaded and installed. We issue "`make test`", which leads to the following console output with elisions in italics:

```
grk/grammars/vscobolii/> make test
Patching section "COVER" ...
Patching another 548 sections ...
Shredding section "COVER" ...
Shredding another 548 sections and 174 diagrams ...
Parsing scratch/abbreviated-combined-relation-condition.dia ...
Parsing another 173 diagrams ...
Parsing script recovery/leafs.fst ...
Applying transformations ...
Parse and apply another 5 scripts for recovery transformations ...
```

```
Dumping file scratch/abbreviated-combined-relation-condition.out ...
Dumping another 298 diagrams ...
Pretty-printing file scratch/abbreviated-combined-relation-condition.out ...
Pretty-printing another 298 diagrams ...
Inlining section "COVER" ...
Inlining another 548 sections and 299 diagrams ...
Parsing script deployment/liberal.fst ...
Applying transformations ...
Parse and apply another 5 scripts for deployment transformations ...
Dumping DCG in file ibm-transformed.pl ...
Parsed 42 lines of Cobol code.
```

That is, at the end, we parse a small Cobol program. We have tested the same grammar with several millions of lines of VS Cobol II code, but we can not distribute the underlying code bases for obvious reasons.

3 Access to the language reference

We will now discuss important details of the VS Cobol II case, as mechanised with the GRK. The first problem is to retrieve the raw grammar knowledge that is contained in IBM's reference.

The starting point: HTML

One line of attack is to download IBM's application programming reference for VS Cobol II from IBM's BookManager BookServer Library [4]. GRK distributes this download: an HTML file of 1763656 Bytes. A fragment is shown in Fig. 2, namely the subscripting syntax for Cobol data names and condition names. Diagrammatic and textual content is mixed. The left margin marks IBM-specific elements — as opposed to ANSI Cobol (cf. "|").

One might wonder whether starting from HTML is an optimal choice. Yes, it is. Grammar recovery is not different from other reverse and re-engineering efforts: one has to use the 'code' that is available. IBM does not publish the standard in any form that is more explicitly structured. IBM publishes the standard in printed form, but OCR (Optical Character Recognition) is more involved than HTML/text processing. In fact, OCR is insufficient because the printed diagrams would require non-trivial visual language parsing.

Patching and shredding

The HTML markup exposes notational anomalies, which resemble some low-level, perhaps even manual details in its production process. Also, the HTML markup refers to other documents, which IBM might eventually move or take offline. Hence, as a first step, *patching* is needed to make sure that:

- the sectioning structure is well formed,
- HTML links only refer to local anchors, and
- all syntax diagrams are in full compliance with the intended visual syntax.

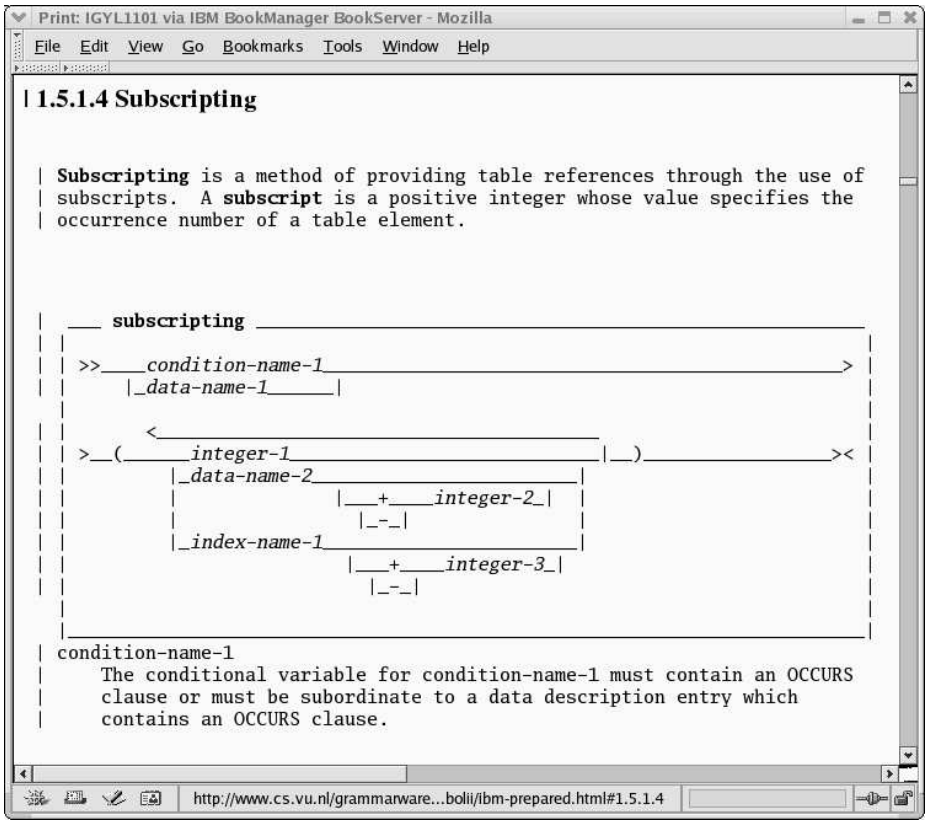

Fig. 2. The original IBM reference — a snapshot

These patches are recorded in scripts. We end up with a patched version of the original document, which we pass on to a shredding phase. Shredding produces a 'document skeleton' where all syntax diagram portions were singled out into separate files. The representation of the document skeleton allows for easy access to the sectioning structure and for re-insertion of the revised diagrams, when we re-generate the language reference later. We represent the shredded document as a plain Prolog term. We could have used XML as well, but interoperability was a non-issue for the early phases of grammar recovery.

Parsing syntax diagrams

To obtain proper access to the raw syntax of VS Cobol II, we need to parse the syntax diagrams to some representation format for grammars. We use (a form of) EBNF in GRK. In fact, there is support for EBNF in *concrete syntax*, and there is a *term-based* representation format.

IBM's HTML uses ASCII art for syntax diagrams. We basically need to parse bars, underscores, smaller-than signs, and all that. When we first tried parsing the diagrams in 1999, then we used a normal context-free grammar (i.e., a string grammar) to define the visual syntax. This approach was inappropriate because complex operations had to be performed when synthesising portions of the diagrams. The two-dimensional syntax of syntax diagrams is

much more easily parsed with an attributed multi-set grammar [3]. This is the approach taken in GRK.

To give an example, the following production describes the visual syntax of vertical layers in a stack of alternatives. (Please refer to Fig. 2 for stacked alternatives. For instance, see the layer with `data-name-1`.) The production is given in Prolog's DCG [2] notation for grammars, which is used in GRK:

```
push(Xmin,Xmax,Ymin,Ymax,Ycenter,Yglue,Ast)
 -->
  select((vline,Xmin,X1,Ycenter,Yglue,_)),
  list(X1,X2,Ymin,Ymax,Ycenter,Ast),
  select((vline,X2,Xmax,Ycenter,Yglue,_)),
  { Yglue >= Ycenter }.
```

The DCG production states that a layer consists of two vertical line segments (cf. `vline`) and a `list` in between. We use a `select` predicate for picking tokens from the remaining token set. The list and the `vline` tokens carry geometric attributes for their two-dimensional positions. We use unification and conditions, e.g., `Yglue >= Ycenter`, to constrain the visual alignment. There is another attribute (cf. `Ast`) for the Prolog-based term representation of the syntax diagram. In fact, the DCG maps the diagram in visual syntax to an EBNF production that is represented as Prolog term. The syntax diagram from Fig. 2 corresponds to the following Prolog term:

```
dia( "subscripting",concat(or(n("condition-name-1"),n("data-name-1")),
concat(nl ,concat(t("("),concat(plus(or(n("integer-1"),or(concat(n(
"data-name-2"),or(epsilon, concat(or(t("+"),t("-")),n("integer-2")))),
concat(n("index-name-2"),or(epsilon, concat(or(t("+"),t("-")),
n("integer-3")))))))), t(")"))))))
```

We note the occurrence of `nl`, which encodes a line-break. The EBNF representation transports such details to the later regeneration so that layout details of the diagrams can be preserved. The current GRK needs more work in a related context; it does not preserve some details such as the mark-up for IBM-specific elements.

4 Recovery transformations

The extracted raw grammar is by no means ready for parser generation. IBM has never intended that its reference is readily useful like that. We need to transform the raw syntax. To this end, GRK offers *fst* — a *f*ramework for *s*yntax *t*ransformation. GRK reuses the name *fst* from [14,8]. GRK allows us

[2] DCG — Definite Clause Grammar – is a grammar formalism combined with logic programming. Prolog systems provide syntactical sugar for DCGs, which is compiled away to plain Prolog. This allows for simple attributed top-down parsers.

to record *fst* transformations in scripts.

The recovery of the VS Cobol II requires 346 simple *fst* transformations to be applied to the raw syntax. The generously commented recovery scripts for refactoring, completion and correction count 2399 lines. Once, we have executed this pile of scripts, we have a grammar that actually accepts VS Cobol II code. We note that we will need further transformations, namely *deployment transformations* that appeal to specific parsing technologies.

Here is an example of a recovery transformation. We *refactor* the subscripting diagram. We extract the actual `subscript` part of the diagram, thereby enabling reuse of that part:

```
% Identified phrase as referred to elsewhere.
Extract subscript = integer-1
                  | data-name-2  (("+" | "-") integer-2)?
                  | index-name-1 (("+" | "-") integer-3)?
From subscripting
```

The existence of a `subscript` phrase is actually suggested in the textual explanations in IBM's reference, but it was not made explicit in the syntax diagrams for whatever reason. Extraction was necessary in order to avoid grammar-code duplication in subsequent steps. We note that each and every transformation includes a line comment (cf. "`%`" above), which will be added to the re-engineered language reference for documentary purposes. The line comments are given in past tense (cf. "`%` Identified ...") since these comments should appeal as change log entries in the reference.

Here is another example of a transformation. This time, we *complete* the subscripting syntax. While the original diagram described subscripts in terms of plain data names (and index names), it is evident from actual Cobol sources that *qualified* data names or even *identifiers* can be used as well. Hence, the raw syntax is incomplete. The following transformation generalises the diagram accordingly:

```
% Enabled identifiers instead of plain data names.
Replace data-name By identifier In subscript
```

We note that the generalisation is encoded as a plain *replacement*. The *fst* language of the current GRK does not provide any checks to distinguish language restrictions and generalisations from more arbitrary replacements. We plan to (re-) implement stronger concepts from [9,14].

Many small steps like those shown above (346 to be precise) complete the recovery phase for the VS Cobol II grammar. The modified subscripting syntax is shown in Fig. 3.

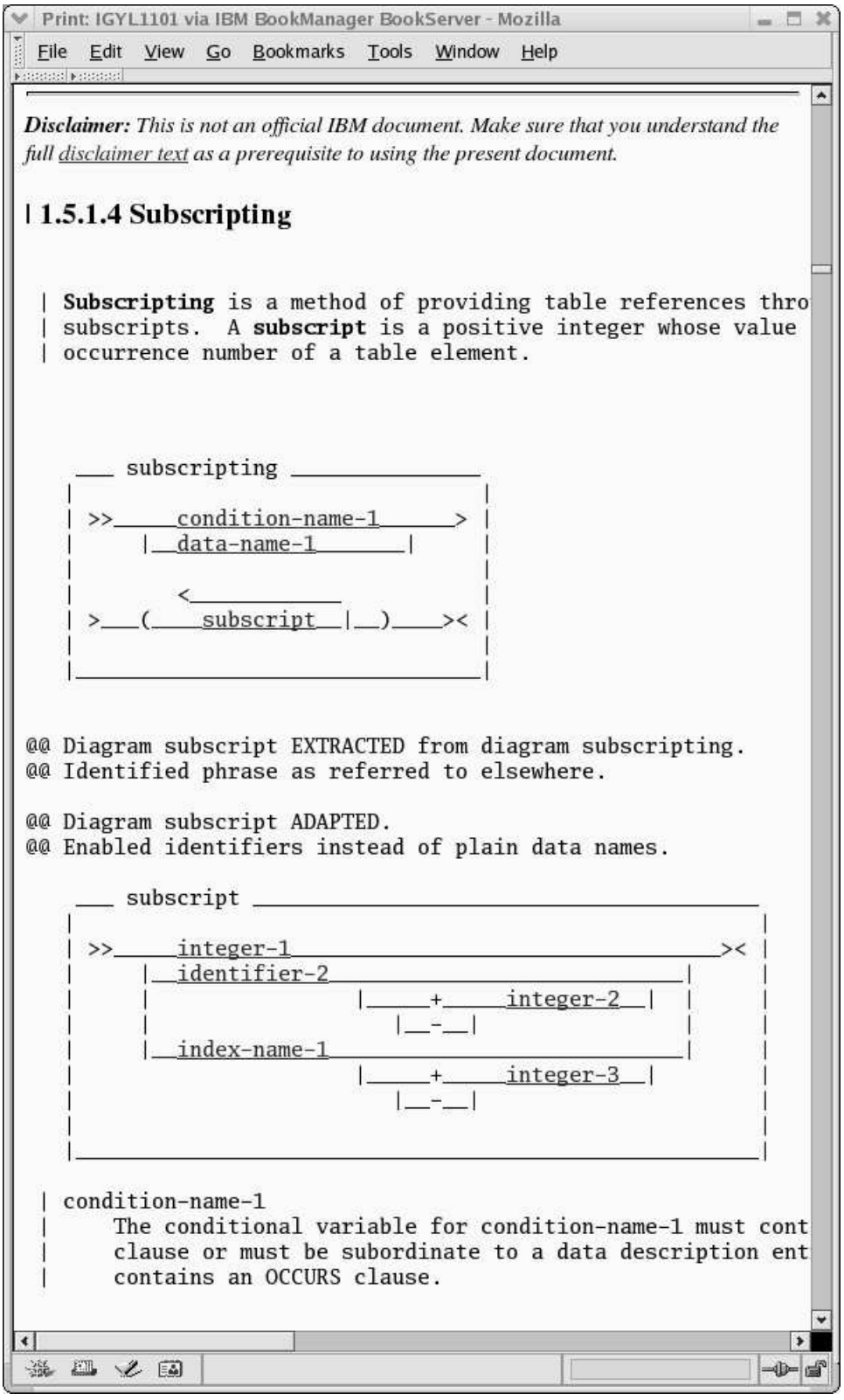

Fig. 3. The re-engineered IBM reference

5 Regeneration of the language reference

The generation of the HTML content in Fig. 3 is simple. We reproduce the original sectioning structure while pretty-printing unchanged, changed, and new syntax diagrams in the right spots. To each and every paragraph, we add a disclaimer (see at the top of Fig. 3) because the re-engineered document must not be confused with a document authorised by IBM.

GRK's incarnation of *fst* takes extra measures in the view of document re-engineering. Transformations that add new productions can explicitly specify the receiving section. Here is a corresponding example, where we add a missing piece of syntax to the section that informally describes this syntax:

```
% Implemented definition as given in the text.
Add
  simple-condition =
       class-condition
     | condition-name-condition
     | relation-condition
     | sign-condition
     | switch-status-condition
     | "(" condition ")"
To 2.8.5.1
```

6 Grammar deployment

GRK has a friend: GDK — the Grammar Deployment Kit [8] who helps out with versatile parser generation. GRK is capable of exporting grammars to GDK in GDK's preferred grammar format, LLL. GDK can then generate parsers covering a range of parsing technologies. GRK, by itself, only generates a Prolog-based prototype parser, which is not very fast, but it is a valid oracle — good enough for grammar debugging during recovery and evolution.

In fact, the Prolog-based prototype parser again uses DCG notation for the execution of grammars in Prolog. The following DCG production has been generated by GRK for the `subscript` diagram from Fig. 3:

```
subscript --> (
 integer;
 identifier, ((@("+") ; @("-")), integer ; true);
 index_name, ((@("+") ; @("-")), integer ; true)
).
```

We see that GRK opts for Prolog's disjunctions (cf. "`;`") to represent EBNF's alternatives and optionals. Tokens are scanned by the predicate `@/1`. Epsilon alternatives boil down to `true/0`.

The grammar that is contained in the re-engineered IBM reference is not

immediately used for parser generation and grammar export to GDK. We have accumulated another 158 *fst* transformations that specifically aim at the preparation of grammar deployment. Several of these transformations resolve ambiguities, or simplify some productions that look fine as syntax diagrams, but too complex in EBNF. Some other transformations eliminate permutation phrases that are not readily supported by various parsing technologies. Yet some other transformations appeal to the limited backtracking model that we applied for the execution of the generated DCG. The final grammar, without DCG-specific tweaks, is exported to GDK, which readily generates an operational `btyacc` [15] parser.

7 Questions & Answers

Question: There are already over a dozen, rock-solid Cobol parsers in the market. So what's the contribution of GRK?
Answer: (We do not label the GRK output as 'rock-solid'.) GRK supports an important insight: we need to improve automation and to adopt engineering techniques in order to recover grammars and to deploy them as parsers. The normal, manual process is too time-consuming and too error-prone [12]. There are thousands of languages in this world. Some counter questions: Why is it that rock-solid Cobol parsers are very expensive to build? Wouldn't grammar transformation help those who produce these parsers?

Question: There tend to be many releases of a language manual. How would this approach reconcile differences between versions of published manuals?
Answer: Road 1: the need for grammar *recovery* will vanish once language references *readily* provide correct and complete and implementable grammars. Grammar tools will then "merely" support the evolution of grammars (within language references), and the deployment of grammars. Road 2: The recovery and deployment transformations are *easily* adopted for a variation on a given language reference. One can maintain these differences in appropriately modularised transformation scripts.

Question: In practise, we need parsers that cope with vendor extensions or obsolete features. How are we supposed to handle such issues?
Answer: Such customisation is accommodated by suitable grammar transformations. In the Cobol case, we have used *deployment scripts* that enable those Cobol extensions that were found in the code bases used for validation. The ability to customise grammars is at the heart of grammar engineering.

Question: GRK promotes recovery in terms of user-provided transformations. What about research on grammar inference ... is it applicable?
Answer: Grammar inference is successful in a number of application domains. However, known efforts to infer grammars for use in programming-language parsers are quite limited in scale; see, e.g., [16,6,1]. Furthermore, we believe that grammar transformations specifically cater for (i) reusing raw grammar

knowledge; (ii) expressing firm but informal grammar knowledge; and (iii) recording precisely the evolution of a grammar. We are interested in a refinement of our approach such that inference concepts are incorporated.

Question: Should I use GRK? (... GDK?) Does it scale?
Answer: As of today, GRK is a prototype that illustrates the notions of semi-automatic grammar recovery and language-reference engineering. (Likewise for GDK.) The generated Cobol parser is readily useful. However, using GRK for other recovery projects or customising the VS Cobol II case requires investment. We hope that other parties get intrigued by grammar engineering. GRK and GDK should be replaced by production-quality tools.

8 Concluding remarks

GRK works towards a solution for the 500-language problem [12], which is the major obstacle to providing tool support for automated software analysis and modification. We need to be able to recover grammars and implement them in front-ends at reasonable speed, with reasonable effort, with predictable quality, all based on a repeatable and transparent process.

The concepts underlying GRK are fairly general. In particular, GRK's approach to the transformation of grammars is of general use. Several specific GRK tools are biased towards IBM standards. (Most notably, GRK's tool support for parsing syntax diagrams is biased in that sense.) It is hoped that GRK triggers work on more general grammar(ware) engineering kits.

GRK is free software. Version 1.0 was released on June 4, 2003. GRK is implemented in SWI-Prolog using Prological language processing [10]. GRK is available on-line at `http://www.cs.vu.nl/grammarware/grk/`. GRK is part of a larger effort at VUA & CWI in Amsterdam on what we call grammarware engineering [7]; refer to `http://www.cs.vu.nl/grammarware/`.

Software archaeologist Jean Marie Favre has compiled a profound and accessible presentation on language reverse engineering [2], where grammar recovery is placed in a historical context. Favre's presentation is warmheartedly recommended to everyone who suspects a link between Cobol and Egyptian history or a link between grammar engineering and model-driven development.

Acknowledgement

I am grateful for the collaboration with Jan Kort on the subject of providing tooling for treating grammars as engineering artifacts. GRK contributes to an overall effort on engineering of grammarware [7]. In this context, I am grateful for collaboration with Paul Klint, Steven Klusener, and Chris Verhoef. I am also very grateful for interaction with other grammar aficionados, and I apologise for any omission in the following list: Mark van den Brand, Jim Cordy, Kris De Schutter, Jean-Marie Favre, Jan Heering, Niels Veerman, Ernst-Jan Verhoeven, Joost Visser.

References

[1] A. Dubey, S. Aggarwal, and P. Jalote. A Technique for Extracting Keyword Based Rules from a Set of Programs. In *Proc. of Conference on Software Maintenance and Reengineering (CSMR 2005)*. IEEE Computer Society Press, 2005. To appear.

[2] J.M. Favre. Metamodel (Driven) (Reverse) Engineering – Stories of the Dagktis Stone and of the Rosetta Stone, March 2004. Presentation at Dagstuhl Seminar 04101 on "*Language Engineering for Model-Driven Software Development*"; Accompanying reading: the series "*From Ancient Egypt to Model Driven Engineering*".

[3] E.J. Golin. *A Method for the Specification and Parsing of Visual Languages*. PhD thesis, Brown University, 1991.

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

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

[6] F. Javed, B.R. Bryant, M. Crepinek, M. Mernik, and A. Sprague. Context-free grammar induction using genetic programming. In *ACM-SE 42: Proc. of the 42nd annual Southeast regional conference*, pages 404–405. ACM Press, 2004.

[7] P. Klint, R. Lämmel, and C. Verhoef. Towards an engineering discipline for grammarware. Draft; submitted for journal publication, 17 August 2003.

[8] J. Kort, R. Lämmel, and C. Verhoef. The Grammar Deployment Kit. In M.G.J van den Brand and R. Lämmel, editors, *Proc. of the 2nd Workshop on Language Descriptions, Tools and Applications (LDTA 2002)*, volume 65 of *ENTCS*. Elsevier Science, April 2002. 7 pages.

[9] R. Lämmel. Grammar Adaptation. In J.N. Oliveira and P. Zave, editors, *Proc. of Formal Methods Europe (FME 2001)*, volume 2021 of *LNCS*, pages 550–570. Springer-Verlag, 2001.

[10] R. Lämmel and G. Riedewald. Prological Language Processing. In M.G.J. van den Brand and D. Parigot, editors, *Proc. of the 1st Workshop on Language Descriptions, Tools and Applications (LDTA 2001)*, volume 44 of *ENTCS*. Elsevier Science, April 2001.

[11] 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/.

[12] R. Lämmel and C. Verhoef. Cracking the 500-Language Problem. *IEEE Software*, pages 78–88, November/December 2001.

[13] R. Lämmel and C. Verhoef. Semi-automatic Grammar Recovery. *Software—Practice & Experience*, 31(15):1395–1438, December 2001.

[14] R. Lämmel and G. Wachsmuth. Transformation of SDF syntax definitions in the ASF+SDF Meta-Environment. In M.G.J van den Brand and Didier Parigot, editors, *Proc. of the 1st Workshop on Language Descriptions, Tools and Applications (LDTA 2001)*, volume 44 of *ENTCS*. Elsevier Science, April 2001.

[15] V. Maslov and C. Dodd. Btyacc—backtracking yacc, 1995-2001. http://www.siber.org/btyacc/.

[16] M. Mernik, G. Gerlic, V. Zumer, and B.R. Bryant. Can a parser be generated from examples? In *SAC 2003: Proc. of the 2003 ACM Symposium on Applied Computing*, pages 1063–1067. ACM Press, 2003.