Electronic Notes in Theoretical Computer Science 65 No. 3 (2002)
URL: http://www.elsevier.nl/locate/entcs/volume65.html 7 pages

The *G*rammar *D*eployment *K*it — System Demonstration —

Jan Kort [1]

Universiteit van Amsterdam

Ralf Lämmel [2]

CWI and Vrije Universiteit

Chris Verhoef [3]

Vrije Universiteit

Abstract

Grammar deployment is the process of turning a given grammar specification into a working parser. The *G*rammar *D*eployment *K*it (for short, GDK) provides tool support in this process based on grammar engineering methods. We are mainly interested in the deployment of grammars for software renovation tools, that is, tools for software re- and reverse engineering. The current version of GDK is optimized for Cobol. We assume that grammar deployment starts from an initial grammar specification which is maybe still ambiguous or even incomplete. In practice, grammar deployment binds unaffordable human resources because of the unavailability of suitable grammar specifications, the diversity of parsing technology as well as the limitations of the technology, integration problems regarding the development of software renovation functionality, and the lack of tools and adherence to firm methods for grammar engineering. GDK helps to largely automate grammar deployment because tool support for grammar adaptation and parser generation is provided. We support different parsing technologies, among them `btyacc`, that is, `yacc` *with backtracking*. GDK is free software.

Further information:

- Grammar Engineering Page: `http://www.cs.vu.nl/grammars/`
- GDK Page: `http://www.cs.vu.nl/grammars/gdk/`

[1] Email: `kort@science.uva.nl`
[2] Email: `ralf@cwi.nl`
[3] Email: `x@cs.vu.nl`

1 Structure of GDK

The Grammar Deployment Kit (GDK) is a lightweight ANSI-C-based kit for getting from a grammar specification to a parser suitable for automated software renovation. Developing renovation parsers for Cobol or any other complex real-world language, is not trivial because of complicated syntax rules, the variety of existing dialects and embedded languages, or pre- and post-processing issues [1]. GDK is based on a few principles and assumptions: (i) recovery of base-line grammars from resources like language references or compilers [3], (ii) grammar adaptation by automated transformations to enable grammar manipulation in a traceable manner [2], (iii) parser generation for various parsing technologies, (iv) selection of a suitable grammar format for grammar engineers, (v) focus on the grammar part of a renovation parser as opposed to idiosyncratic scanners, pre- and post-processors. The current version of GDK is optimized for the deployment of Cobol grammars.

GDK *components*

- LLL: a simple EBNF-based grammar format.
- FST: a tool to transform grammars.
- GENPARSER: a tool for generating parsers from grammars.
- GDKLIB: a library for parsing, transformation, and unparsing.
- VS COBOL II: deployment of a Cobol grammar for software renovation.

We will discuss all these components accordingly. In the VS COBOL II section, we will also sketch an illustrative renovation tool EXPAND dealing with data expansion. This sort of software renovation task is similar to problems a la Y2K or Euro conversion.

2 The LLL grammar format

LLL should be read as "L-Cube". The name hints on *L*ightweight LL parsing which is the built-in parsing technology of GDK. The LLL grammar format cross-cuts all components in GDK in that it is used as (i) the primary format to import recovered grammar specifications for deployment, (ii) to develop new grammars, (iii) to transform grammars in the course of deployment, and (iv) to generate parsers for various technologies.

EBNF of LLL *grammar format in* LLL *notation*

```
grammar    : rule+;
rule       : sort ":" alts ";";
alts       : alt alts-tail*;
alts-tail  : "|" alt;
alt        : term*;
term       : basis repetition?;
basis      : literal | sort;
repetition : "*" | "+" | "?";
```

A VS COBOL II *sample: the* `MOVE`*-statement*

```
move-statement   : move-statement-1 | move-statement-2;
move-statement-1 : "MOVE" cobword-or-literal "TO" identifier+;
move-statement-2 : "MOVE" corresponding identifier "TO" identifier+;
corresponding    : "CORRESPONDING" | "CORR";
```

A suitable grammar format is crucial for grammar deployment. There are three major problems with commonly used grammar formats. (i) A too restrictive format such as the simple BNF format underlying `yacc` requires encoding of lists and optionals. (ii) On the other hand, use of a too liberal format, e.g., non-trivially nested phrases makes the grammar less suitable for matters of abstract syntax, debugging, and adaptation. (iii) Using the input language of a specific parser generator for grammar recovery, development, maintenance, and others enforces one to deal with the idiosyncrasies of the format all the time. LLL supports EBNF to enable *regular expression operators* in the view of (i), but it *restricts* them to rule out (ii). Furthermore, LLL is a *pure* EBNF notation to remedy (iii), that is, LLL does not directly deal with conflict resolution, disambiguation, lexical syntax, and others. Our experience with many grammar recovery, development, and deployment projects resulted in a simple and sound grammar notation that is particularly suited for developing renovation parsers.

3 Automated grammar adaptation with FST

FST stands for *F*ramework for *S*yntax *T*ransformation. The FST tool supports the adaptation of grammars by means of step-wise transformation. Adaptation of grammars is needed (i) to correct and complete a grammar in the process of grammar recovery as discussed in [3], and (ii) to refactor and to disambiguate a grammar specification in the process of grammar deployment. The input language of the command-line oriented tool FST offers 11 transformation operators. Sequences of such transformation steps can be recorded in transformation scripts. The FST tool is invoked as follows:

`fst` *`trafo-script`* `<` *`grammar-in`* `>` *`grammar-out`*

An illustrative selection of FST *operators*

- `%rename sort %to sort` — rename a nonterminal
- `%resolve rule` — provide a missing definition
- `%redefine rule` — replace a definition of a nonterminal
- `%include rule` — add alternatives to a definition
- `%exclude rule` — remove alternatives from a definition

FST enforces pre- and post-conditions for all these operators to see whether a transformation is sound to some extent, at the very least it will be checked whether the grammar has changed after every transformation step. The foundations of grammar adaptation are described in [2]. The design of the FST operator suite has been optimized for ease of use, a good fit with the LLL grammar format, and simple im-

plementation of FST. A more ambitious predecessor of GDK's FST is discussed in [4] where FST expands to *F*ramework for *S*DF *T*ransformation.

4 Parser generation with GENPARSER

The GENPARSER tool of GDK generates parsers from LLL grammars. Various parsing technologies are supported, among them `btyacc`. One might say that GENPARSER is a "parser generator input generator" but some of the supported parsing technologies do not employ a proper parser generator but rather a combinator approach. Hence, we say that GENPARSER "generates parsers" which maybe still need to be processed by a "so-called" parser generator. Note that GENPARSER does not just export the grammar for use with some parsing technology, but it also generates code for parse-tree construction, scanner templates, and, for the C-based parsers, one-level term builds and matches supporting a simplified form of rewriting over the parse trees. Cobol scanners are rather idiosyncratic, and the manual effort for their implementation is affordable. Hence, we only generate scanner templates. For several technologies we also consider scannerless parsing, and then empty function definitions for parsing the lexical sorts are provided. The GENPARSER tool is invoked as follows:

```
genparser -f format options < grammar
```

The *`format`* selects the parsing technology, e.g., `-f btyacc` could be used. Further *`options`* depend on the parsing technology. One can, for example, issue backtracking cuts for backtracking parsers such as `btyacc`. In general, we assume that little tweaking of this kind is needed because the grammar is prepared accordingly by transformations.

Summary of supported `formats`

- `btyacc`: `btyacc` parser and a `flex` scanner template.
- `lll`: GDKLIB-based combinator parser and a `flex` scanner template.
- `slp`: GDKLIB-based scannerless combinator parser.
- `precc`: Scannerless `precc` parser without parse-tree construction.
- `antlr`: `antlr` parser without parse-tree construction.
- `haskell`: Haskell-based scannerless combinator parser.
- `sdf`: SDF/`pgen`/`sglr`-based parser.
- `accent`: Accent parser and a `flex` scanner template.

The use of too basic parsing technology such as LALR(1) is considered harmful [1]. In fact, not even our final VS COBOL II grammar gets close to LALR(1): there are 542 shift/reduce conflicts, and 62 reduce/reduce conflicts, not all of them being serious. It would take continuous effort to make a grammar work with plain `yacc`, and to keep it conflict-free when the grammar needs to be adapted. Hence, we focused on more powerful parsing technologies in the above list. We have benchmarked all the technologies with a 2 MLOC code base for VS COBOL II. As for preprocessing, we used a home-grown tool not distributed with GDK. Let us sum-

marize the most important results of the deployment case. (i) The `btyacc` parser requires the least tweaking, and it is the most efficient parser. Note that `btyacc` requires a commercial license. (ii) The GDKLIB-based parsers require only slightly more tweaking, and they are still faster than all the other technologies we looked at (and about factor 2 slower than `btyacc`). The GDKLIB-based parsers rely on combinators for top-down parsers with local backtracking for the alternatives of a rule. Additional means of guarding an alternative are supported to enable some form of parser tweaking. (iii) Most other technologies—and, in fact, we looked at more than those from the above list—expose problems with either grammar-class restrictions, the model of disambiguation, parse-tree construction, installation, usability, performance, scalability, robustness, integration, and others.

5 The GDK library

GDK is a self-contained kit for grammar deployment. This means one is not required to employ any third-party components to develop simple software renovation tools. In fact, the C-based GDKLIB offers lightweight functionality for parsing, parse-tree construction, rewriting, traversal, unparsing, and pretty printing. Hence, there is no need to install additional software to assess the suitability of the other components of GDK, especially the VS COBOL II grammar which was deployed with GDK. As an aside, GDKLIB is also used for the implementation of FST and GENPARSER.

Modules in GDK *V1.0 (1689 LOC in total)*

Module	**Intention**	**LOC**
`mt.c`	Term construction and inspection	270
`mtutils.c`	traversal, container, unparsing functionality	622
`parselib.c`	LLL combinator parsing	245
`slp.c`	Scannerless LLL combinator parsing	284
`pretty.c`	Pretty printing	122
`token.c`	Tokenization	92
`stack.c`	Generic stacks	54

The GDKLIB functionality can be used for the implementation of simple renovation tools. That is, GDKLIB is complementary to more sophisticated technologies such as ASF+SDF, DMS, or REFINE. To give examples of current limitations of GDKLIB, the term format `MTerm` provided by the GDKLIB does not support garbage collection, and the C-encodings of rewrite rules and traversals are not type-checked whereas this is suported, e.g., by ASF+SDF. On the other hand, for GDKLIB it is not necessary to learn a new language, which is the case for more sophisticated tools. The performance of GDKLIB-based combinator parsers is only outperformed by `btyacc`. Here, we take advantage of grammar class restrictions for combinator parsing. These restrictions are not necessarily appropriate for all languages.

6 VS COBOL II

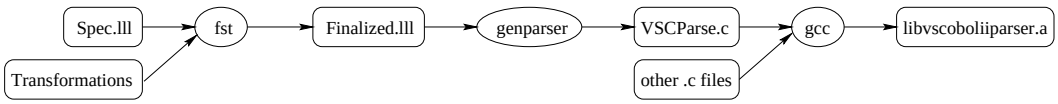
The distribution of GDK comes with the initial VS COBOL II grammar specification as delivered in [3], with all the transformation scripts to further disambiguate and refactor the grammar, and to prepare it for use with different parsing technologies. To illustrate the usefulness of the deployed grammar, a software renovation tool EXPAND is included in the distribution of GDK. In the present section, we briefly indicate the renovation task addressed by EXPAND, we describe the process to derive the underlying renovation parser, and finally we sketch the internal structure of EXPAND. This tool demonstrates how to use a generated renovation parser for a re-engineering task.

A VS COBOL II *program before and after adaptation by* EXPAND

```
IDENTIFICATION DIVISION.                IDENTIFICATION DIVISION.
 PROGRAM-ID. LITTLE-Y2K-TEST.            PROGRAM-ID. LITTLE-Y2K-TEST.
DATA DIVISION.                          DATA DIVISION.
 WORKING-STORAGE SECTION.                WORKING-STORAGE SECTION.
  01  SEEK-NAME       PIC 99.             01  SEEK-NAME       PIC 999.
  01  OTHER-NAME-1    PIC 99.             01  OTHER-NAME-1    PIC 999.
  01  OTHER-NAME-2    PIC 99999.          01  OTHER-NAME-2    PIC 99999.
  01  OTHER-NAME-3    PIC 99.             01  OTHER-NAME-3    PIC 99.
PROCEDURE DIVISION.                     PROCEDURE DIVISION.
 ...                                     ...
 MOVE SEEK-NAME TO OTHER-NAME-1.         MOVE SEEK-NAME TO OTHER-NAME-1.
 MOVE SEEK-NAME TO OTHER-NAME-2.         MOVE SEEK-NAME TO OTHER-NAME-2.
 IF OTHER-NAME-2 > 99                    IF OTHER-NAME-2 > 299
  ...                                     ...
```

The sample illustrates that we want to expand certain two-digits fields to three digits, that is, `PIC 99` becomes `PIC 999`. This range expansion also triggers the adaptation of literals, namely the replacement of a maximum value `99` by the new value `299`. The affected fields are determined by a seed-and-propagate algorithm. The identification of seed-set elements is based on name heuristics. The propagation relies on a type-of-usage analysis.

Derivation of a Cobol parser



The initial grammar specification `Spec.lll` is transformed by a number of FST scripts resulting in a grammar `Finalized.lll`. This grammar is passed to GENPARSER to issue parser generation. The rest of the figure is specific to C-based parsers. The output of GENPARSER (maybe after processing by a "parser generator") is compiled by `gcc`. This process is captured in a `Makefile`. To make it straightforward to switch between technologies, all parsers are provided as a library with one top-level function `MTerm parseCbl(FILE *f)` for reading in a file and producing a parse tree.

The *`main` function of the* EXPAND *tool*

```
#include "parsecbl.h" // btyacc or gdklib-based Cobol parser
#include "cobpp.h"    // Cobol pretty printer
#include "expand.h"   // Prototypes of transformation function

int main(int argc, char **argv)
{
    MTerm pt;                       // parse tree
    pt = parseCbl(stdin);           // parse program
    COBPPdump(stdout, expand(pt)); // expand and pretty-print
    return 0;
}
```

That is, the library function `parseCbl` is invoked resulting in a parse tree `pt`. The renovation task is encoded in the function `expand` which is structured as follows. (i) The seed set is determined. (ii) The propagation is performed. (iii) The picture mask of affected fields is expanded. (iv) Affected literals are adapted. These steps basically amount to traversals which are specific only for a few Cobol patterns. We use generic traversal functionality supplied by GDKLIB. The transformed program is finally pretty-printed with `COBPPdump`. The complete C code for the EXPAND tool amounts to 241 LOC.

7 Conclusion

GDK is a natural step from semi-automated grammar recovery [3] to actual grammar deployment in software renovation. GDK illustrates that tool-supported grammar recovery and deployment is a realistic and useful means to carry out actual renovation tasks, without the burden of first constructing a parser by hand. Because GDK is lightweight and self-contained, you can easily review the underlying engineering ideas for software renovation, without having to install additional tools.

References

[1] 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 IWPC'98*, pages 108–117, 1998.

[2] R. Lämmel. Grammar Adaptation. In *Proceedings of FME'01*, volume 2021 of *LNCS*, pages 550–570. Springer-Verlag, 2001.

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

[4] R. Lämmel and G. Wachsmuth. Transformation of SDF syntax definitions in the ASF+SDF Meta-Environment. In M.G.J. van den Brand and D. Parigot, editors, *Proceedings of LDTA'01*, volume 44 of *ENTCS*. Elsevier Science, April 2001.