Grammar Modularity and its Impact on Grammar Documentation

Stefanie Dipper

Universität Potsdam
Institut für Linguistik
D-14415 Potsdam
Germany
dipper@ling.uni-potsdam.de

Humboldt-Universität zu Berlin
Inst. für deutsche Sprache und Linguistik
D-10099 Berlin
Germany
stefanie.dipper@rz.hu-berlin.de

Abstract

This paper addresses the documentation of large-scale grammars.[1] We argue that grammar implementation differs from ordinary software programs: the concept of modules, as known from software engineering, cannot be transferred directly to grammar implementations, due to grammar-specific properties. These properties also put special constraints on the form of grammar documentation. To fulfill these constraints, we propose an XML-based, grammar-specific documentation technique.

1 Introduction

Research in the field of grammar development focuses on grammar modularization, ambiguity management, robustness, testing and evaluation, maintainability and reusability. A point which has often been neglected is the detailed documentation of large-scale grammars—despite the fact that thorough documentation of the grammar code is a prerequisite for code maintainability and reusability.

In this paper, we argue that documenting large-scale grammars is a complex task that requires special, grammar-specific documentation techniques. The line of reasoning goes as follows. We show that maintainability (and, hence, reusability) of a grammar depends to a large extent on the modularization of the grammar rules: a large-scale grammar remains maintainable only if linguistic generalizations are encoded explicitly, i.e., by modules (sec. 3.1). However, in contrast to modules in ordinary software programs, (certain) grammar modules cannot be black boxes (sec. 3.2). This property puts special constraints on the form of grammar documentation (sec. 4). Finally, we present an XML-based documentation technique that allows us to accomodate these constraints (sec. 5).

[1]The paper is based on my doctoral dissertation (Dipper, 2003), which I wrote at the IMS Stuttgart. I am very grateful to Anette Frank for invaluable discussions of the dissertation. Many thanks go to Bryan Jurish and the anonymous reviewers for helpful comments on the paper.

To illustrate the needs of documentation, we refer to a German LFG toy grammar (Lexical-Functional Grammar, cf. sec. 2). Our argumentation, however, applies not only to grammars in the LFG formalism but to any grammar that is modularized to a certain extent.

2 Lexical-Functional Grammar

LFG is a constraint-based linguistic theory (Bresnan (2001), Dalrymple (2001)). It defines different levels of representation to encode syntactic, semantic and other information.

The levels that are relevant here are constituent structure (c-structure) and functional structure (f-structure). The level of c-structure represents the constituents of a sentence and the order of the terminals. The level of f-structure encodes the functions of the constituents (e.g. subject, adjunct) and morpho-syntactic information, such as case, number, and tense.

The c-structure of a sentence is determined by a context-free phrase structure grammar and is represented by a tree. In contrast, the f-structure is represented by a matrix of attribute-value pairs. The structures are linked by a correspondence function (or mapping relation), called "ϕ-projection".

The (simplified) analysis of the sentence in (1) illustrates both representation levels, see fig. 1.

(1)	*Maria*	*liest*	*oft*	*Bücher*
	M.	reads	often	books
	'Maria often reads books'			

As an example, we display the CP rule in (2) (which gives rise to the top-most subtree in fig. 1).

$$(2)\quad \texttt{CP} \rightarrow \begin{array}{cc} \texttt{NP} & \texttt{C}' \\ (\uparrow\texttt{SUBJ})\texttt{=}\downarrow & \uparrow\texttt{=}\downarrow \end{array}$$

The arrows $\uparrow$ and $\downarrow$ refer to f-structures; they define the ϕ-projection from c-structure nodes to f-structures. The $\uparrow$-arrow refers to the f-structure of

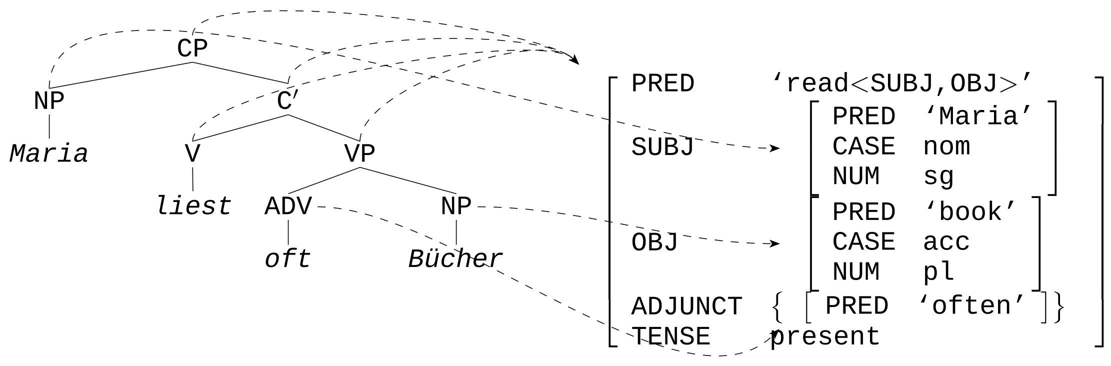

Figure 1: LFG c-structure and f-structure analysis of *Maria liest oft Bücher*

the mother node (= the CP), the ↓-arrow to the f-structure of the node itself (= NP, C′).[2]

That is, the above rule states that CP dominates an NP and a C′ node; the NP functions as the subject (SUBJ) of CP, and C′ is the head of CP (sharing all features, by unification of their respective f-structures).

However, the NP preceding C′ may as well function as the direct (OBJ) or indirect object (OBJ2), depending on case marking. We therefore refine the CP rule by making use of disjunctive annotations, marked by curly brackets, cf. (3).

```
(3)  CP →
                 NP                      C′
     { (↑SUBJ)=↓ (↓CASE)=nom        ↑=↓
     | (↑OBJ)=↓   (↓CASE)=acc
     | (↑OBJ2)=↓ (↓CASE)=dat }
```

3 Grammar Modularity

Large grammars are similar to other types of large software projects in that modularity plays an important role in the maintainability and, hence, reusability of the code. Modularity implies that the software code consists of different modules, which in ordinary sofware engineering are characterized by two prominent properties: (P1) they are "black boxes", and (P2) they are functional units.

Black boxes Modules serve to encapsulate data and are "black boxes" to each other. That is, the input and output of each module (i.e. the interfaces between the modules) are clearly defined, while the module-internal routines, which map the input to the output, are invisible to other modules.

Functional units Usually, a module consists of pieces of code that belong together in some way (e.g. they perform similar actions on the input). That is, the code is structured according to functional considerations.

Modular code design supports transparency, consistency, and maintainability of the code. (i) Transparency: irrelevant details of the implementation can be hidden in a module, i.e. the code is not obscured by too many details. (ii) Consistency is furthered by applying once-defined modules to many problem instances. (iii) Maintainability: if a certain functionality of the software is to be modified, the software developer ideally only has to modify the code within the module encoding that functionality. In this way, all modifications are local in the sense that they do not require subsequent adjustments to other modules.

Turning now to modules in grammar implementations, we see that similar to modules in ordinary software projects, grammar modules encode generalizations (functional units, property P2). However, we argue below that (certain) grammar modules are not black boxes (whose internal structure is irrelevant, property P1), because these generalizations encode important linguistic insights.

3.1 Grammar Modules

Similarly to modules in ordinary software projects, modules in grammar implementations assemble pieces of code that are functionally related: they do this by encoding linguistic generalizations. A linguistic generalization is a statement about properties that are common to/shared by different constructions. A grammar module consists of a coherent piece of code that encodes such common properties and in this sense represents a functional unit.

In a modularized grammar, all constructions that share a certain property should make use of the same grammar module to encode this property. Generalizations that remain implicit (i.e. generalizations that are not encoded by modules) are error-prone. If the analysis of a certain phenomenon is modified, all constructions that adhere

[2]Whenever an arrow is followed by a feature, e.g. SUBJ, they are enclosed in parentheses, (↑SUBJ).

to the same principles should be affected as well, automatically—which is not the case with implicit generalizations.

Which sorts of modules can be distinghuished in a grammar implementation? In this paper, we limit ourselves to two candidate modules: (i) syntactic rules and (ii) macros.

Syntactic rules Each syntactic rule, such as the CP rule in (3), can be viewed as a module. A syntactic category occurring on the right-hand side of a rule (e.g. NP in (3)) then corresponds to a module call (routine call); the f-structure annotations of such a category ((↑SUBJ)=↓) can be seen as the instantiated (actual) parameters that are passed to the routine. Groups of rules (e.g. CP, C′, and C) form higher-level modules: X′-projections.

To sum up, syntactic rules can sensibly be viewed as modules (cf. also Wintner (1999), Zajac and Amtrup (2000)). Their internal expansion is irrelevant for the calling rule (property P1), and they form a linguistically motivated unit (property P2).[3]

Macros Grammar development environments (such as XLE, Xerox Linguistic Environment, described in Butt et al. (1999, ch. 11)) provide further means of abstraction to modularize the grammar code, e.g. (parametrized) macros and templates. Each macro/template can be viewed as a module, encoding common properties.[4]

An example macro is NPfunc in (4), which may be used by the closely related annotations of NPs in different positions in German, e.g. the annotations of NPs dominated by CP and by VP, cf. (5). (Macro calls are indicated by '@'.)

```
(4)  NPfunc =
          { (↑SUBJ)=↓ (↓CASE)=nom
          | (↑OBJ)=↓  (↓CASE)=acc
          | (↑OBJ2)=↓ (↓CASE)=dat }

(5)  CP →          NP           C′
                @NPfunc        ↑=↓

     VP →          ADV          NP
              ↓∈(↑ADJUNCT)   @NPfunc
```

[3]With regard to f-structure, however, these modules are not canonical black boxes. LFG provides powerful referencing means within global f-structures, i.e. f-structure restrictions are not (and can in general not be) limited to local subtrees. In a way, f-structure information represents what is called "global data" in software engineering: all rules and macros are essentially operating on the same "global" data structures.

[4]XLE macros/templates can be used to encapsulate c-structure and f-structure code. Moreover, macros/templates can be nested, and can thus be used to model constraints similar to type hierarchies (Dalrymple et al., To Appear).

That is, NPfunc is used to encapsulate the alternative NP functions in German. This encoding technique has the advantage that the code is easier to maintain. For instance, the grammar writer might decide to rename the function OBJ2 by IOBJ. Then she/he simply has to modify the definition of the macro NPfunc rather than the annotations of all NPs in the code. Clearly, NPfunc represents a functional unit; the question of whether NPfunc is a black box to other modules, such as the syntactic rule CP, is addressed in the next section.

3.2 Code Transparency and Black Boxes

The above example shows how macros can be used to encode common properties. In this way, the intentions of the grammar writer are encoded explicitly: it is not by accident that the NPs within the CP and VP are annotated by identical annotations. In this sense, the use of macros improves code transparency. Further, macros help guarantee code maintainability: if the analysis of the NP functions is modified, only one macro (NPfunc) has to be adjusted.

In another sense, however, the grammar code is now obscured: the functionality of the CP and VP rules cannot be understood properly without the definition of the macro NPfunc. Macro definitions may even be stacked, and thus need to be traced back to understand the rule encodings. In this sense, one might say that the use of macros hinders code transparency.[5]

In order to distinguish these opposing views more precisely we introduce two notions of transparency, which we call *intensional* and *extensional*.

Intensional transparency of grammar code means that the characteristic defining properties of a construction are encoded by means of suitable macros, i.e in terms of generalizing definitions. Hence, all constructions that share certain defining properties make use of the same macros to encode these properties (e.g. the CP and VP rules in (5)).

Conversely, distinguishing properties of different constructions are encoded by different macros—even if the content of the macros is identical.

Extensional transparency means that linguistic generalizations are stated "extensionally", i.e. macros are replaced by their content/definition (similar to a compiled version of the code). The grammar rules thus introduce the constraints directly rather than by calling a macro that would introduce them (similar to the CP rule in (3)).

[5]The same argumentation applies to type hierarchies: to understand the functionality of a certain type, constraints that are inherited from less specific, related types must be traced back.

Comparing both versions, the extensional version (3) may seem easier to grasp and, hence, more transparent. To understand the generalized version in (5), it is necessary to follow the macro calls and look up the respective definitions. Obviously, one needs to read more lines of code in this version, and often these lines of code are spread over different places and files. For instance, the CP rule may be part of a file covering the CP internal rules, while the macro NPfunc figures in some other file.

Especially for people who are not well acquainted with the grammar, the intensional version thus requires more effort for understanding. In contrast, people who work regularly on the grammar code know the definitions/functionalities of macros more or less by heart. They certainly grasp the grammar and its generalizations more easily in the intensional version.

One might argue that to know the name of a macro, such as NPfunc, often suffices to "understand" or "know" (or to correctly guess) the functionality of the macro. Hence, a macro would be a black box (whose definition/internal structure is irrelevant), similar to modules in ordinary software programs.

However, there is an important difference between grammar implementations and canonical software programs: grammars encode linguistic insights. The grammar code by itself represents important information in that it encodes formalizations of linguistic phenomena (in a particular linguistic framework). As a consequence, users of the grammar are not only interested in the pure functionality (the input-output behaviour) of a grammar module. Instead, the concrete definition of the module is relevant, since it represents the formalization of a linguistic generalization.

We therefore conclude that macro modules, such as NPfunc, are only defined by property P2 (functional unit), not by property P1 (black box).

The criteria of maintainability and consistency clearly favour intensional over extensional transparency. We argue that the shortcomings of intensional transparency—namely, poorer readability for casual users of the grammar—can be compensated for by a special documentation structure.

4 Grammar Documentation

In large software projects, code documentation consists of high-level and low-level documentation. The high-level documentation comprises information about the function and requirements of (high-level) modules and keeps track of higher-level design decisions (e.g. which modules are distinguished). More detailed documentation includes lower-level design decisions, such as the reasons for the chosen algorithms or data structures.

The lowest level is that of code-level documentation. It reports about the code's intent rather than implementation details however, i.e. it focuses on "why" rather than "how". For instance, it summarizes relevant features of functions and routines. A large part of the code-level documentation is taken over by "good programming style", e.g. "use of straightforward and easily understandable approaches, good variable names, good routine names" (McConnell (1993, p. 454)).

The level that is of interest to us is that of code-level documentation. In contrast to documentation of other types of software, grammar documentation has to focus both on "why" and "how", due to the fact that in a grammar implementation the code in and of itself represents important information, as argued above. That is, the details of the input–output mapping represent the actual linguistic analysis. As a consequence, large parts of grammar documentation consist of highly detailed code-level documentation.

Moreover, the content/definition of certain dependent modules (such as macros) is relevant to the understanding of the functionality of the mother rule. Hence, the content of dependent modules must be accessible in some way within the documentation of the mother rule.

One way of encoding such dependencies is by means of links. Within the documentation of the mother rule, a pointer would point to the documentation of the macros that are called by this rule. The reader of the documentation would simply follow these links (which might be realized by hyperlinks).[6] However, a typical grammar rule calls many macros, and macros often call other macros. This hierarchical structure makes the reading of link-based documentation troublesome, since the reader has to follow all the links to understand the functionality of the top-most module.[7]

We therefore conclude that the structure of the

[6]Certain programming languages provide tools for the automatic generation of documentation, based on comments within the program code (e.g. Java provides the documentation tool Javadoc, URL: `http://java.sun.com/javadoc/`). The generated documentation makes use of hyperlinks as described above, which point to the documentation of all routines and functions that are used by the documented module.

[7]Routines and functions in ordinary software may be hierarchically organized as well. In contrast to grammar modules, however, these modules are (usually) black boxes. That is, a reader of the documentation is not forced to follow all the links

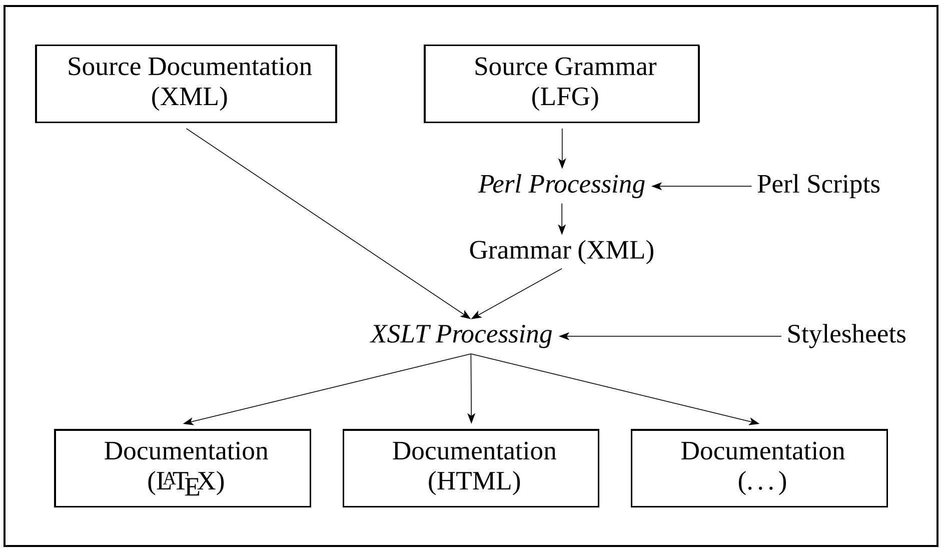

Figure 2: XML-based grammar documentation

documentation should be independent of the structure of the grammar code. We suggest a documentation method that permits copying of relevant grammar parts (such as macros) and results in a user-friendly presentation of the documentation.

5 An XML-based Grammar Documentation Technique

In our approach, grammar code and documentation are represented by separate documents. The documentation of a rule comprises (automatically generated) copies of the relevant macros rather than simple links to these macros. In a way, our documentation tool mirrors a compiler, which replaces each macro call by the content/defi nition of the respective macro. In constrast to a (simple) compiler, however, our documentation keeps a record of the macro calls (i.e. the original macro calls are still apparent). In the terminology introduced above, our documentation thus combines extensional transparency (by copying the content of the macros) with intensional transparency (by keeping a record of the macro calls).

The copy-based method has the advantage that the structure of the documentation is totally independent of the structure of the code which is being documented.

We propose an XML-based documentation method, i.e. the source documentation and the grammar code are enriched by XML markup. XSLT stylesheets operate on this markup to generate the actual documentation (e.g. an HTML document or a LaTeX document, which is further processed to result in a postscript or PDF fi le). The XML tags are used to link and join the documentation text and the grammar code. In this way, the documentation is independent of the structure of the code.

to understand the functionality of the top-most module.

Fig. 2 presents the generation of the output documentation. The source documentation is created manually in XML format (e.g. by means of an XML editor); the source grammar is written manually in LFG/XLE format. Next, XML markup is added to the source LFG grammar via Perl processing. Specifi c XML tags within the documentation refer to tags within the grammar code. The XSLT processing copies the referenced parts of the code to the output documentation.

This approach guarantees that the code fragments that are displayed in the documentation are always up-to-date: whenever the source documentation or grammar have been modifi ed, the output documentation is newly created by XSLT processing, which newly copies the code parts from the most recent version of the grammar.

5.1 Further Features of the Approach

The described documentation method is a powerful tool. Besides the copying task, it can be exploited in various other ways, both to further the readibility of the documentation and to support the task of grammar writing (see also the suggestions by Erbach (1992)).[8]

Snapshots Grammar documentation is much easier to read if pictures of c- and f-structures illustrate the analyses of example sentences. XLE supports

[8]Except for the different output formats, all of the features mentioned in this paper have been implemented.

the generation of snapshot postscript fi les, displaying trees and f-structures, which can be included in a LaTeX document. Note, however, that after any grammar modifi cation, such snapshots have to be updated, since the modifi ed grammar may now yield different c- and f-structure analyses.

In our approach, snapshots are updated automatically: All example sentences in the source documentation are marked by a special XML tag. XLE snapshots are triggered by this markup and automatically generated and updated for the entire documentation, by running the XSLT stylesheet.

Indices In our approach, the documentation does not follow the grammar structure but assembles grammar code from different modules. Moreover, documentation may refer to partial rules only (or macros). That is, the complete documentation of an entire rule can be spread over different sections of the documentation.

User-friendly documentation therefore has to include an index that associates a grammar rule (or macro) with the documentation sections that comment on this rule. That is, besides referencing from the documentation to the grammar (by copying), the documentation must also support referencing (indexing) from various parts of the grammar to the relevant parts of the documentation.

Again, such indices are generated automatically based on XML tags in our approach.

Test-Suites Example sentences in the documentation can be used to automatically generate a test-suite. In this way, the grammar writer can easily check whether the supposed coverage—as reported by the documentation—and the actual coverage of the grammar are identical.

It is also possible to create specialized test-suites. For instance, one can create a test-suite of interrogative NPs, by extracting all examples occurring within the section documenting interrogative NPs.

Up to now, we have seen how to create and exploit XML-based grammar documentation. The next section addresses the question of how to maintain such a type of documentation.

5.2 Maintainability

A grammar implementation is a complex software project and, hence, often needs to be modifi ed, e.g. to fi x bugs, to widen coverage, to reduce overgeneration, to improve performance, or to adapt the grammar to specifi c applications. Obviously, the documentation sections that document the modifi ed grammar parts need to be modifi ed as well.[9]

[9]As mentioned above, in some respects, the (output) documentation is updated automatically by our XML/XSLT-based approach. XSLT operates on the most recent version of the grammar, therefore all grammar-related elements within the output documentation that are generated via XSLT are automatically synchronized to the current grammar (e.g. snapshots).

In our approach, grammar code and documentation are represented by separate documents. Compared to code-internal comments, such code-external documentation is less likey to remain up-to-date, because it is not as closely associated with the code. This section discusses techniques that could be applied to support maintenance of our XML-based documentation.

We distinguish three types of grammar modifi cations. (i) An existing rule (or macro) is deleted. (ii) An existing rule is modifi ed. (iii) A new rule is added to the code.

In case (i), the XSLT processing indicates whether a documentation update is necessary: Any rule that is documented in the documentation is referenced by its ‘id’ attribute. If such a rule is deleted from the code, the referenced ‘id’ attribute does not exist any more. In this case, the XSLT processing prints out a warning that the referenced element could not be found.

If, instead, rules are modifi ed or added (cases (ii) and (iii)), utilities such as the UNIX command ‘diff’ can be applied to the output text fi les: Suppose that the grammar has been modifi ed while leaving the documentation text untouched. Now, if the LaTeX fi les are newly generated, the only parts that may possibly have changed are the parts citing grammar code. These parts can be located by means of the ‘diff’ command. If such changes between the last and the current LaTeX fi les have occurred, these changes indicate that the surrounding documentation sections may need to be updated. If no changes have occurred, despite the grammar modifi cations, this implies that the modifi ed parts are not documented in the (external) documentation and, hence, no update is necessary. By this technique, the grammar writer gets precise hints as to where to search for documentation parts that may need to be adjusted.

To sum up, maintenance of the documentation text can be supported by techniques that give hints as to where the text needs to be adjusted. In the scenarios sketched above, the grammar writer would fi rst modify the grammar only and generate some new, temporary output documentation. Comparing the current with the last version of the output documentation would yield the desired hints. After an update of the documentation text, a second run of the XSLT processing would generate the fi nal output documentation.

6 Conclusion and Outlook

In this paper, we discussed the importance of maintainability and documentation in grammar development. A modular and transparent design of the grammar and detailed documentation are prerequisites for reusability of the grammar code in general.

A modularized grammar is "intensionally transparent", as we put it, and thus favours maintainability. However, for casual users of the grammar, modularity may result in decreased readability. This is related to the fact that grammar modules are not black boxes, since they encode linguistic generalizations. We argued that this can be compensated for by a special documentation technique, which allows for user-friendly documentation that is independent of the structure of the grammar code.

Similar to common grammar-specific tools that are provided by grammar development environments, we propose a grammar-specific documentation technique (which ought to be integrated into the grammar development environments, as also suggested by Erbach and Uszkoreit (1990), Erbach (1992)).

Our XML-based documentation technique is a very powerful means that can be exploited to support the difficult task of grammar (and documentation) development in various further ways. For instance, the grammar code can be "translated" to a pure XML document, i.e. each atomic element of the code (syntactic categories such as NP; f-structure elements, e.g. ↑, SUBJ, =) is marked by a tag. This markup can be used in various ways, for instance:

– The grammar code can be displayed with refined highlighting, e.g. c-structure and f-structure elements can be printed in different colours. This improves the transparency and readability of the code.

– The grammar code can be mapped to a representation that uses annotated trees instead of rules. This may result in a better understanding of the code. (However, the mapping to the annotated-tree representation is not trivial, since c-structure rules make use of regular expressions.)

References

Joan Bresnan. 2001. *Lexical-Functional Syntax*, volume 16 of *Textbooks in Linguistics*. Oxford, UK: Blackwell.

Miriam Butt, Tracy Holloway King, María-Eugenia Niño, and Frédérique Segond. 1999. *A Grammar Writer's Cookbook*. Number 95 in CSLI Lecture Notes. Stanford, CA: CSLI.

Mary Dalrymple, Ron Kaplan, and Tracy H. King. To Appear. Lexical structure as generalizations over descriptions. In Miriam Butt and Tracy H. King, editors, *Proceedings of the LFG04 Conference*. CSLI Online Proceedings.

Mary Dalrymple. 2001. *Lexical Functional Grammar*, volume 34 of *Syntax and Semantics*. New York et al.: Academic Press.

Stefanie Dipper. 2003. *Implementing and Documenting Large-Scale Grammars—German LFG*, volume 9(1) of *AIMS (Arbeitspapiere des Instituts für Maschinelle Sprachverarbeitung)*. University of Stuttgart.

Gregor Erbach and Hans Uszkoreit. 1990. Grammar engineering: Problems and prospects. CLAUS Report No. 1. Report on the Saarbr ̈ucken Grammar Engineering Workshop, University of the Saarland, Germany.

Gregor Erbach. 1992. Tools for grammar engineering. In *Proceedings of ANLP-92*, pages 243–244, Trento, Italy.

Steve McConnell. 1993. *Code Complete. A Practical Handbook of Software Construction*. Redmond, WA: Microsoft Press.

Shuly Wintner. 1999. Modularized context-free grammars. In *Proceedings of MOL6—Sixth Meeting on Mathematics of Language*, pages 61–72, Orlando, Florida.

Rémi Zajac and Jan W. Amtrup. 2000. Modular unification-based parsers. In *Proceedings of the Sixth International Workshop on Parsing Technologies*, Trento, Italy.