Handbook for Language Engineers

Ali Farghaly (ed.)

August 4, 2003

**CENTER FOR THE STUDY
OF LANGUAGE
AND INFORMATION**

Center for the Study of Language and Information
Leland Stanford Junior University
Printed in the United States
07 06 05 04 03   1 2 3 4 5

Library of Congress Cataloging-in-Publication Data

Farghaly, Ali Ahmed Sabry, 1938–
Handbook for language engineeers / by Ali Farghaly.
p. cm. – (CSLI lecture notes ; no. 164)
Includes bibliographical references and index.

ISBN 1-57586-395-2 (alk. paper)
ISBN 1-57586-396-0 (pbk. : alk. paper)

1. Computational linguistics. 2. Applied linguistics. I. Title.
P98.F36 2003
410′.285–dc22 2003015094
CIP

∞ The acid-free paper used in this book meets the minimum requirements of the American National Standard for Information Sciences—Permanence of Paper for Printed Library Materials, ANSI Z39.48-1984.

CSLI was founded in 1983 by researchers from Stanford University, SRI International, and Xerox PARC to further the research and development of integrated theories of language, information, and computation. CSLI headquarters and CSLI Publications are located on the campus of Stanford University.

CSLI Publications reports new developments in the study of language, information, and computation. In addition to lecture notes, our publications include monographs, working papers, revised dissertations, and conference proceedings. Our aim is to make new results, ideas, and approaches available as quickly as possible. Please visit our web site at
http://cslipublications.stanford.edu/
for comments on this and other titles, as well as for changes and corrections by the author and publisher.

Contents

4

Grammar Writing, Testing, and Evaluation

MIRIAM BUTT AND TRACY HOLLOWAY KING

4.1 Introduction

Grammar writing is a difficult and often underappreciated task in natural language processing.[1] A good grammar writer will have solid linguistic instincts based on their linguistic knowledge and training. However, a good grammar writer also has to have the ability and willingness to cast aside lofty linguistic ideals when confronted with the harsh reality of needing to get something done today rather than in ten years. In addition, grammar development platforms can sometimes be limiting in that the implementational possibilities do not always mirror current linguistic theory. On the other hand, grammar development platforms also often provide tools and possible approaches to coding grammars which linguists have not considered, and which may thus open up entirely new avenues of approaching a given linguistic problem.

Therefore, a good grammar writer is somebody who combines linguistic knowledge with a solid practical ability and an understanding and interest in the technical and computational aspects of language processing. This combination is relatively rare, especially as grammar writing is often a slow, laborious process. The task of a grammar writer can be made easier by the existence of morphological analyzers (section 4.2.7), large lexicons (section 4.2.6), and statistical methods (see

[1]We would like to thank Emily Bender, Richard Crouch, Mary Dalrymple, Stefanie Dipper, Regine Eckardt, Ali Farghaly, Jonas Kuhn, and Heike Zinsmeister for their help with this chapter.

Handbook for Language Engineers.
Ali Farghaly (ed.).

Callison-Burch and Osborne, chapter 7, this volume) which make the grammar more robust (section 4.3).

Language is a system of complex interactions. As such, design decisions made in one part of the grammar often have unforeseen consequences in another part of the grammar. Unforeseen interactions between rules or modules make grammars difficult to expand systematically and to maintain (section 4.2.4), particularly if more than one grammar developer is involved. Systematic testing and evaluation of grammars is therefore also one of the main tasks of the grammar writer (section 4.5).

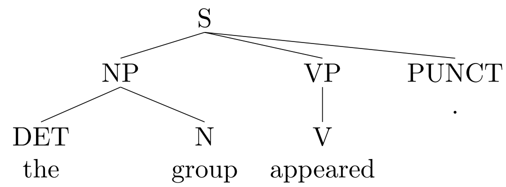
A grammar consists of rules which a parser can understand. A parser is generally a computational implementation of a particular linguistic theory or theoretical direction. For example, a GB parser (Retoré and Stabler, 1999) parses rules which presuppose the tree representations and movements assumed by Government-Binding theory or, more recently, Minimalism (Stabler, 2001). A Lexical-Functional Grammar (LFG) or Head-Driven Phrase Structure Grammar (HPSG) parser deals with rules which create the kinds of representations assumed by these theories (trees and attribute-value matrices for LFG; attribute-value matrices for HPSG). Various types of parsers currently exist, including ones which assume a mixture of theoretical approaches such as the *Trug* parser of Siemens, which models a type of tree-unification grammar (Block and Schachtl, 1992). Other well-known parsers assume Tree-Adjoining Grammar (TAG, see Schabes et al., 1997; Abeillé and Rambow, 2000) and Categorial Grammar (CG, see Steedman, 2001; Morrill, 1995); see also Carpenter's Type-Logical Grammar Theorem Prover (Carpenter, 1998) which is available on-line (`http://www.colloquial.com/tlg/index.html`). The task of writing a parser differs considerably from the task of writing a grammar which a parser can read. A grammar cannot be run without a parser to parse it, just as any computer program cannot be run without the underlying compiler for the language the computer program is written in. See Jurafsky and Martin (2000) and references therein for information about writing parsers.

In this chapter, we primarily use LFG as a basis for discussing the issues surrounding grammar development. However, regardless of theoretical persuasion, the job of a parser is to load a grammar and other tools in order to take a natural language string as input and produce a linguistic structure as output for that string. For a string like (1), for example, some possible outputs are shown in (2). The outputs in (2a,b) are instances of what is known as *shallow parsing*. The shallow parsing results in (2a,b) illustrate part of speech tagging and chunking, respectively (section 4.3). The LFG structures in (2c,d) are the result of a deeper level of analysis. The output of a grammar can be of more than one type; for example, LFG grammars output both trees (2c) and attribute-value matrices (2d).

(1) The group appeared.

(2) a. the/DET group/N appeared/V ./PUNCT

b. [the group]$_{NP}$ [appeared]$_{VP}$.

c. 

d. $$\begin{bmatrix} \text{PRED} & \text{appear} \\ \text{SUBJ} & \begin{bmatrix} \text{PRED} & \text{group} \\ \text{SPECIFIER} & \text{the} \end{bmatrix} \\ \text{TENSE} & \text{past} \end{bmatrix}$$

The choice of output depends on what the output will be used for. The shallow parse outputs are extremely useful for a variety of low-level tasks, but cannot replace the type of deep analyses represented by (2c,d) for other tasks.

Many researchers and grammar developers subscribe to the idea that an ideal grammar should be *reversible*. That is, the same grammar used in a parser to produce the linguistic structures in (2c,d) for the input string in (1) should also be able to take the linguistic structure (or a subset of it, such as just the attribute-value matrix in (2d)) as input and produce a natural language expression as output. This process is referred to as *generation* and represents the flip-side of parsing.

Grammars for natural language processing tend to be quite large, even for limited domain applications. As such, testing and evaluation of the grammar are a crucial part of grammar writing. These are discussed in section 4.5. One thing to keep in mind is that large-scale grammar writing shares many problems with all large-scale software projects. For some ideas about how to handle large-scale and multi-person coding, see Part II of Butt et al. (1999) or any of the many books written on software development, such as Maguire (1993) and McConnell (1996).

This chapter is organized as follows. First we discuss deep and shallow parsing. We next discuss auxiliary mechanisms which can be combined with deep parsing techniques and follow this by a section on machine learning, describing efforts to arrive at grammars by a process of induction. Finally, we discuss testing and evaluation, which apply to both shallow and deep grammar writing. The chapter ends with a summary and suggestions for further reading.

4.2 Deep Grammars

In this section we describe the ideas and mechanisms behind grammars which provide a deep analysis for natural language strings.

When computational linguistics first got underway in the 1950s, one of its main goals was to build machine translation systems. The computational efforts during World War II had included a number of successful code breaking episodes.[2] Then, during the Cold War, a crucial ingredient of counter-intelligence measures was to read and understand the enemies' communications. Given that breaking coded enemy communications had worked very well, it was thought that applying a version of that technology to develop machine translation systems should yield reasonable results within a reasonable amount of time. However, language proved to be a much harder problem than the code systems devised artificially by humans.

A prerequisite for building a good machine translation system is to have an in-depth understanding of how natural language works and to have formal tools which can model natural language adequately. In the last few decades, our understanding of natural languages and the formal tools used to describe them have grown considerably, but they are still far from complete, thus leaving machine translation as a basically unsolved problem (commercial machine translation systems are currently available, but none of them are ultimately satisfactory).

4.2.1 Context-Free Grammars

In his pioneering work on syntax, Chomsky (1957) argued that the *syntax* of a language was largely independent of semantics and world knowledge. So, for example, speakers of English can tell that (3) is a well-formed sentence of English even though they may never have heard it before and even though it represents semantic gibberish.[3]

(3) Colorless green ideas sleep furiously.

Chomsky therefore proposed the notion of *phrase structure* which consisted of rewrite rules of the type shown in (4).

[2] Alan Turing, who is responsible for some of the foundational concepts within computational linguistics and artificial intelligence, was involved in code breaking.

[3] Another famous example is Lewis Carroll's poem *Jabberwocky*.

(4) a. S $\longrightarrow$ NP VP

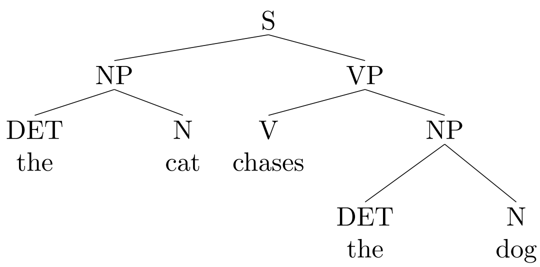
b. NP $\longrightarrow$ DET N

c. VP $\longrightarrow$ V NP

These rewrite rules or *phrase structure rules* parse an input string such as in (5). Their output can be rendered in terms of a tree, as shown in (6).[4]

(5) The cat chases the dog.

(6)



These phrase structure or rewrite rules are *context free*. A context-free grammar is a set of rules as in (7), which are the non-terminal symbols, and a lexicon as in (8), which encodes the terminal symbols.

The conclusion Chomsky (1957) comes to is that context-free rules are not sufficient to deal with natural language phenomena (for a very good discussion of this issue see Jurafsky and Martin, 2000). This conclusion continues to be widely accepted. However, context-free rules do describe some useful properties of natural languages. These include the fact that they are recursive and infinite. Consider recursiveness. English NPs can contain PPs which comprise a P followed by an NP, as in (7).

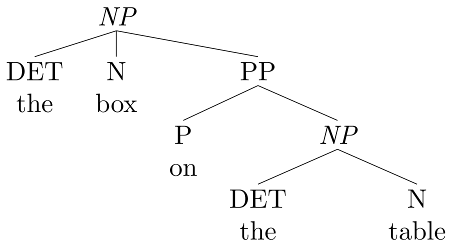
(7) a. NP $\longrightarrow$ DET N (PP)
b. PP $\longrightarrow$ P NP

(8) a. box N
b. on DET

These produce a structure in which an NP contains an NP further embedded in its structure, as shown in (9).

[4]The original motivation behind the tree structure was to keep a record of the steps in the parsing process. These steps were understood by Chomsky to be a series of "derivations."

(9)



Such structures are pervasive in natural language and all deep grammars need to encode recursiveness in order to obtain broad coverage.

Recursiveness is part of the infinite nature of natural language. The rule in (7), allows for strings like those in (10).

(10) a. [$_{\text{NP}}$ the box [$_{\text{PP}}$ on [$_{\text{NP}}$ the table]]
b. [$_{\text{NP}}$ the box [$_{\text{PP}}$ on [$_{\text{NP}}$ the table [$_{\text{PP}}$ in [$_{\text{NP}}$ the room]]]]
c. [$_{\text{NP}}$ the box [$_{\text{PP}}$ on [$_{\text{NP}}$ the table [$_{\text{PP}}$ in [$_{\text{NP}}$ the room [$_{\text{PP}}$ under [$_{\text{NP}}$ the roof]]]]]]]

Another simple example of the potentially infinite number of strings natural language can generate is provided by adjunct PPs. As shown in (11), English potentially allows for an infinite number of PPs.

(11) a. I saw it on the table.
b. I saw it on the table with a microscope.
c. I saw it on the table with a microscope on Thursday.
d. ...

This property of natural languages can be captured in phrase structure rules by the use of the Kleene star (*) which allows for zero or more, up to infinity, instances of a constituent. The PP examples in (11) are represented by the phrase structure rule in (12).

(12) VP $\longrightarrow$ V NP PP*

Not all theories allow context-free rules with regular expressions (see Megerdoomian, chapter 6, this volume) on the right hand side.[5] Such theories must find other ways to capture these generalizations.

Despite the recursive and infinite properties of context-free grammars, some context-sensitivity is necessary. A classic example of this is subject-verb agreement in English. The context-free rule in (14) allows for mismatched subject-verb agreement as in (13a), for example.

[5]Government-Binding or Minimalism, for example, do not rely on context-free rules, but instead work with predefined sets of possible tree structures. HPSG implementations have no provision for the Kleene Star on the right hand side of rules (Stefan Müller, p.c. Spring 2003)

(13) a. *The men arrives.
b. The man arrives.

(14) S ⟶ NP VP

To avoid this problem, context sensitive rules can be used. The equation under the phrase structure rule in (15) indicates that the agreement of node 1, the subject NP, must be the same as that of node 2, the VP.

$$\text{(15)}\quad \text{S} \longrightarrow \underset{\text{AGR1=AGR2}}{\text{NP1}\quad\text{VP2}}$$

Basic context-free rules can also be enriched to provide other information. In LFG, for example, one extension to the context-free grammar is to *annotate* the basic phrase structure rules, as shown in (16).

(16) a. $\text{S} \longrightarrow \underset{(\uparrow\text{SUBJ}) = \downarrow}{\text{NP}} \quad \text{VP}$

b. $\text{NP} \longrightarrow \underset{(\uparrow\text{SPECIFIER}) = \downarrow}{\text{DET}} \quad \text{N}$

c. $\text{VP} \longrightarrow \text{V} \quad \underset{(\uparrow\text{OBJ}) = \downarrow}{\text{NP}}$

In English phrase structure position reflects grammatical function, e.g., in English the subject is generally preverbal and the object immediately postverbal. The annotation shown in (16a) identifies the first NP within a sentence (S) as the subject of the sentence. Determiners, as dealt with by (16b), are identified as specifiers of the noun phrase, and finally, the immediately postverbal NP is marked as the object in (16c).

Most other languages do not pattern in this way and thus require different kinds of annotations. The ability to annotate phrase structure rules with differing information, depending on the needs and requirements of individual languages, provides a very powerful mechanism for the analysis of natural languages (Kaplan, 1987).

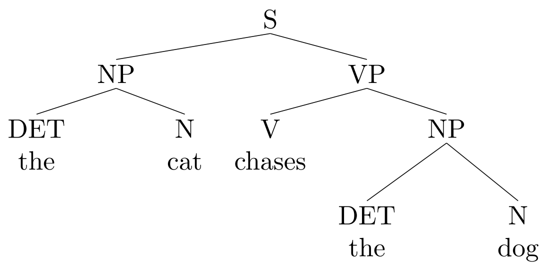
4.2.2 Feature Unification

Many of the more popular computational grammatical frameworks make extensive use of *unification*. Most frameworks use some kind of feature representation and provide for feature unification; HPSG and LFG make extensive use of unification. In this chapter, we demonstrate how unification is used in grammar writing within LFG.

LFG includes two core syntactic representations: c(onsituent)-structure (a tree) and f(unctional)-structure (an attribute-value matrix), illustrated in (18). The c-structure in (18a) is produced by the types of phrase structure rules discussed in the previous section in (16). As also mentioned in the previous section, the context-free phrase structure rules in LFG are generally annotated with functional information whereby the arrows encode mappings between nodes of the phrase structure tree and the functional-structure. The '↓' refers to the attribute-value matrix that the phrase structure node in question corresponds to. The '↑' refers to that of the node's mother. These phrase structure annotations together with information gathered from the individual lexical items yield the f-structure representation in (18b).

(17) The cat chases the dog.

(18) a.



b.
$$\begin{bmatrix} \text{PRED} & \text{'chase< SUBJ, OBJ >'} \\ \text{SUBJ} & \begin{bmatrix} \text{PRED} & \text{'cat'} \\ \text{SPECIFIER} & \text{the} \\ \text{NUM} & \text{sg} \\ \text{PERS} & 3 \end{bmatrix} \\ \text{OBJ} & \begin{bmatrix} \text{PRED} & \text{'dog'} \\ \text{SPECIFIER} & \text{the} \\ \text{NUM} & \text{sg} \\ \text{PERS} & 3 \end{bmatrix} \\ \text{TENSE} & \text{present} \end{bmatrix}$$

The f-structure takes the form of an attribute-value matrix. That is, a given feature or attribute such as SUBJ or NUM must have a certain value. This value could be another attribute-value matrix, as is the case for SUBJ, it could be an atomic value, such as 'sg' for NUM, or it could be a set, such as the values for CASE in (24) below.

Values of features may come from more than one source. For example, in order to check whether the verb displays the proper agreement morphology, the verbal agreement feature values must be checked against the feature values of the subject. In grammars such as LFG and HPSG, the agreement features provided by the verb (e.g., NUM sg, PERS 3) must be able to unify with the feature specifications of the subject.

For a sentence as in (19), for example, feature unification will fail because the person features of the subject and the verb do not match: the subject is third person whereas the verb cannot be third person.

(19) *The cat chase the dog.

(20)
$$\begin{bmatrix} \textsc{pred} & \text{'chase< } \textsc{subj}, \textsc{obj} \text{ >'} \\ \textsc{subj} & \begin{bmatrix} \textsc{pred} & \text{'cat'} \\ \textsc{specifier} & \text{the} \\ \textsc{num} & \text{sg} \\ \textsc{pers} & \mathbf{3/\neg 3} \end{bmatrix} \\ \textsc{obj} & \begin{bmatrix} \textsc{pred} & \text{'dog'} \\ \textsc{specifier} & \text{the} \\ \textsc{num} & \text{sg} \\ \textsc{pers} & 3 \end{bmatrix} \\ \textsc{tense} & \text{present} \end{bmatrix}$$

The feature clash in (20) illustrates the most basic kind of checking which can be done via feature unification. However, more complex feature logics can be employed as well. For example, LFG also contains the notion of an *instantiated feature*. Instantiated features cannot unify with any other feature, even if it is of the same type. This ensures that a feature can be supplied at exactly one place in the grammar and is useful for phenomena such as particle verbs. In English, particles can occur before or after the object, but not in both places at once, as seen in (21) for the particle verb *throw out*.

(21) a. I threw *out* the trash.
b. I threw the trash *out*.
c. *I threw *out* the trash *out*.

If the particle provides the feature PARTICLE with the value *out*, then the two *out* values in (21c) could unify to produce a (simplified) structure like in (22).

(22)
$$\begin{bmatrix} \textsc{pred} & \text{'throw< } \textsc{subj}, \textsc{obj} \text{ >'} \\ \textsc{subj} & \begin{bmatrix} \textsc{pred} & \text{'I'} \end{bmatrix} \\ \textsc{obj} & \begin{bmatrix} \textsc{pred} & \text{'trash'} \end{bmatrix} \\ \textsc{particle} & \text{out} \end{bmatrix}$$

This is undesirable and can be avoided by instantiating the value of the PARTICLE feature (represented by an underscore on the value: *out_*) so that it cannot unify with another instance of the feature.

Another method for the checking of feature values is *feature indeterminacy* (Dalrymple and Kaplan, 2000). Feature indeterminacy is useful when a given item is compatible with more than one feature specification. One example which came up during the development of the German grammar for the ParGram project (Butt et al., 1999) is the function of the relative pronoun *was* 'what' in a free relative clause such as (23). Here, the free relative *was* 'what' can function as the accusative object of *gegessen* 'eaten' while fulfilling the requirements of *lag* 'lay' for a nominative subject.

(23) Ich habe gegessen, was auf dem Tisch lag.
I have eaten what on the table lay
'I ate what was lying on the table.'

Dalrymple and Kaplan (2000) propose that *was* 'what' be annotated as being *indeterminate* between the features nominative and accusative. That is, the case values of *was* 'what' are given as a set designator, as shown in (24). This notation expresses the idea that the case values of *was* 'what' are not atomic, but rather are a set. The notation also provides an exhaustive enumeration of the possible members of the set.

(24) was: $(\uparrow\textsc{case}) = \{\text{nom, acc}\}$

The feature unification of the f-structural analysis checks whether the object of *gegessen* 'eat' has the case value 'acc' as required by the verb. A test as to whether the value 'acc' is a value for the case feature of *was* 'what' yields a positive result. Similarly, a test as to whether the value 'nom' is a value for the case feature of *was* 'what', as required by the second clause, also yields a positive result. The two requirements are thus mutually consistent, and the analysis is well-formed.

Other methods of dealing with feature unification are possible. For example, the default unification logic as presented by Lascarides and Copestake (1999) provides a different set of well-formedness possibilities. For an application of this approach to Finnish case licensing, see Asudeh (2000). The use of features in combination with different methods of unification thus allows for implementations which are both powerful in terms of their linguistic descriptive power and yet mathematically well-constrained. For further readings on features and unification in general, see Jurafsky and Martin (2000) and for HPSG in particular see Shieber (1986) and Copestake (2002).

4.2.3 Semantic Representations

So far, we have not said very much about semantic analyses. This is because the construction of a good semantic analysis is a much harder problem than the writing of annotated phrase structure rules.

However, there is significant work on computational semantics, including the definition of ontologies (Noy, chapter 5, this volume) and the building of knowledge representation systems (Stone, chapter 8, this volume). We cannot do justice to the field of computational semantics here and confine ourselves to noting that due to the modular nature in which computational grammars are usually written, various theories of semantics are often compatible with one and the same grammar.

In theories like HPSG, for example, which take a Saussurean *sign-based* approach to natural language, the syntax is considered to be intimately connected to semantic analysis. Despite this intimate connection between syntax and semantics, computational implementations of HPSG have been shown to be compatible with Discourse-Representation Theory (DRT) (Kamp and Reyle, 1993) and Minimal-Recursion Semantics (MRS) (Egg, 1998; Copestake et al., 1999, 2001; Copestake, 2002). The Linguistic Resources Online English Resource Grammar (LinGO ERG, see Copestake and Flickinger, 2000) provides a broad coverage syntactic and semantic grammar using MRS.

LFG has also been shown to be compatible with DRT (Reyle, 1988). In recent years the linear logic or *glue* approach to semantics has led to significant research (Dalrymple, 1999, 2001). This linear logic approach in turn can be used with a number of semantic frameworks including DRT (van Genabith and Crouch, 1999b), Dynamic Semantics (van Genabith and Crouch, 1999a), and Intensional Logic (Dalrymple et al., 1999). Within ParGram (Butt et al., 1999), for example, fglue semantic analyses are based on the f-structural analyses and are dealt with in a separate module from the syntax.

General practice is thus to treat computational syntax and computational semantics as different modules of the grammar which interact via an interface. Whether this is the most perspicuous method of dealing with the interaction between syntax and semantics remains to be seen. However, a realistic assessment of the current situation is that it is relatively easy (though labor intensive) to achieve a broad-coverage grammar which focuses on the morphological and syntactic properties of a language. A broad-coverage grammar which provides general semantic and discourse-based analyses of large corpora, on the other hand, so far remains out of practical reach, with the Core Language Engine (Alshawi, 1992) being one of the closest to this goal.

4.2.4 The Challenge of Deep Grammar Writing

Deep grammars are necessary for certain applications, particularly those that need information about argument structure (i.e, who did what to whom in a given event). However, they face a number of problems that need to be addressed before using them in real world applications. We discuss these challenges to deep grammars here.

One of the major problems facing deep grammars is robustness. For many sentences, the deep grammar will not produce a parse either because the syntactic or lexical structure involved has not yet been incorporated into the grammar or because the sentence is not, in fact, well formed. This is a problem even with closed domains in which the vocabulary is relatively fixed and many syntactic constructions are not used (e.g., interrogatives are absent in some domains). Although it is possible to partially combat this problem by extending the grammar and the lexicon (section 4.2.6 and 4.2.7), complete coverage is unlikely to be achieved. As such, to ensure some output for any input, other techniques are needed. One possibility is to use a shallow parser (section 4.3) when the deep parser fails. If the shallow parser is integrated into the deep parser as a way of producing well-formed chunks, it is possible to obtain complete parses for subparts of the input. For example, an ungrammatical sentence like (25), which could be the result of a cut and paste error, could be divided into two parts, each of which obtains a deep parse.

(25) a. The box the box arrived.

b. parse for: *the box*

$$\begin{bmatrix} \textsc{pred} & \text{'box'} \\ \textsc{specifier} & \text{the} \\ \textsc{num} & \text{sg} \\ \textsc{pers} & 3 \end{bmatrix}$$

c. parse for: *the box arrived.*

$$\begin{bmatrix} \textsc{pred} & \text{'arrive}< \textsc{subj} >\text{'} \\ \textsc{subj} & \begin{bmatrix} \textsc{pred} & \text{'box'} \\ \textsc{specifier} & \text{the} \\ \textsc{num} & \text{sg} \\ \textsc{pers} & 3 \end{bmatrix} \\ \textsc{tense} & \text{past} \end{bmatrix}$$

In extending coverage, grammar writers have to analyze structures that are not normally dealt with in theoretical syntax because they are not considered to be part of core grammar. This presents a linguistic challenge that can result in interesting innovations.

Some examples of phenomena which are not often dealt with as part of standard syntactic analyses are direct speech (quotations), headers, and date and time expressions. These phenomena are prevalent in certain applications, such as those involving newspaper texts. Examples are shown in (26a–c) respectively.

(26) a. “The board of directors met on Thursday,” said John Smith.
b. Disputed Elections
c. They were registered on Monday, December 6, at 3:30 p.m.

Consider how to deal with instances of direct speech. There is relatively little linguistic literature on this subject (but see Collins and Branigan, 1997). In this case, the grammar writer can use the existing theoretical literature to draft a computational treatment of direct speech. However, there will not be ready answers for all of the issues which arise with respect to direct speech, such as: Is the material in the quotes the head of the clause or is the verb of saying the head? How is the inversion of the verb of saying and the subject handled? What happens when the material between the quotes is not the entire utterance or when it is absent, as in (27)?

(27) a. The board of directors “met on Thursday,” said John Smith.
b. The board of directors met on Thursday, said John Smith.

In the English grammar for the ParGram project (Butt et al., 1999), these issues were dealt with as follows. The material in the quotes is treated as the head of the clause; the verb of saying and its subject are a type of parenthetical with both V-subject and subject-V orders possible. Quotes are allowed to surround any constituent and are permitted regardless of whether the special quote parenthetical is present; conversely, a quote parenthetical can appear even when there are no quotation marks present. This type of analysis may not always be aesthetically pleasing to the theoretical syntactician, but the realities of the data force such considerations and can lead to insights into how these constructions fit into the better understood parts of the syntax.

Conversely, a consideration of issues generally regarded as low-level and mostly of technical interest can engender a useful linguistic discussion. Within the ParGram project (Butt et al., 1999) an example of this is the proposal of *m(orphosyntactic)-structure*. This new level of representation was proposed as the result of a discussion of how to treat local well-formedness features such as agreement morphology on determiners and adjectives or dependencies across verbal forms. In German, for example, a determiner such as *der* ‘the’ in (28) which inflects according to a “strong” morphological paradigm must be followed by an adjective which is inflected according to a “weak” morphological paradigm. Conversely, a weak determiner must be followed by strong adjectives.

(28) der graue Hund (German)
the.M.Sg.Nom.**S** grey.M.Sg.Nom.**W** dog
'the grey dog'

These inflectional dependencies must be checked in order to ensure syntactic well-formedness within the noun phrase. However, the weak and strong inflectional features do not contribute any semantic information, nor do they reflect deep language universal principles.

Another example along these lines are verb form dependencies in a cascade of verbs and auxiliaries such as in the English (29).

(29) The dog will have been chasing the cat.

As is well known, the modal *will* selects a base form of the auxiliary *have*: the forms *had* or *having*, for example, would have been ungrammatical. The auxiliary *have*, in turn, selects the form of the auxiliary *be*: *being* or *be* would have been ungrammatical, etc. Again, the precise verbal form selected encodes a local dependency and does not contribute much useful semantic information.

In the interests of parallelism in analyses across languages (Butt et al., 1996), a level of representation called m-structure was proposed which would encode language dependent, idiosyncratic local well formedness information. This idea, which was originally motivated by the need to find a viable computational implementation, has since been taken up in the theoretical literature as part of discussions on the morphology-syntax interface (e.g., Spencer and Sadler, 2001; Frank and Zaenen, 2000).

Beyond the issue of the kind of phenomena grammar writers must consider, the development of a broad coverage deep grammar also raises several other issues which concern grammar writers but which do not always concern theoretical linguists. Grammar maintenance and consistency,[6] for example, become a real issue (see section 4.5 on grammar testing and evaluation). As such, implementational tools may be needed which may not be of theoretical significance. In the LFG system, *templates* are one such device (Butt et al., 1999). Templates are a short-hand for a larger set of information: they permit a name associated with a complex set of formulas to be used instead of that set. Templates can take arguments that substitute for variables in the definition of the template. As such, they express linguistic generalizations that would otherwise have to be repeated many times in the lexicon and grammar. One issue that arises with respect to templates is that one

[6]Consistency is a goal of syntactic theory, but it is difficult to test without an implemented grammar.

grammar's convenience may be another grammar's theoretical claim. The inheritance hierarchies of HPSG do much of the same work as the templates described here (Copestake, 2002). However, within HPSG, inheritance hierarchies are an important theoretical concept, whereas within LFG a discussion of templates is usually not found outside of the computational/implementational realm.

One example of a template is subcategorization frames for verbs. These have to be entered many times in the lexicon and may contain very complex information. A simplified template for a particle verb might look like (30).

(30) V-SUBJ-OBJ-PARTICLE(_PRED _PARTICLE) =
(↑PRED) = '_PRED<(↑SUBJ) (↑OBJ)>'
(↑PRT-FORM)=_PARTICLE

Different particle verbs call this template with the relevant values, as in (31).

(31) a. throw: @(V-SUBJ-OBJ-PARTICLE throw out_)
b. ring: @(V-SUBJ-OBJ-PARTICLE ring up_)

Templates can also call other templates so that deeply nested dependencies result. Having templates or a related implementational tool decreases the chance of making a mistake since there is less to type (just a template name, not all the contained information), increases consistency (when changes are made they are made in just one place), and shows generalizations (every place that a given template is called can be found). Without such tools, writing large-scale grammars becomes difficult as coverage increases.

Another interesting problem as grammar coverage increases is rule interaction. The addition of a rule may have unintended interactions with other rules; these interactions may be desirable or may not. The interactions can be extremely interesting from a theoretical linguistic perspective as well as from a grammar writing perspective; only by implementing a large grammar fragment can such interactions of analyses be seen. Consider the following situation. The grammar allows both transitive and ditransitive uses of verbs, as in (32).

(32) a. We sold boxes.
b. We sold the man boxes. (= We sold boxes to the man.)

A rule is added to allow noun-noun compounds for examples like (33).

(33) a. the tractor trailer
b. the door hinge
c. utility company stocks

The addition of this rule will result in (32b) having two parses: the intended one and one in which *sell* is transitive and what is sold are *the man boxes* (=boxes for men/of men). This is a syntactically viable parse, but one that was unlikely to be foreseen. Generally, adding rules, especially for less core constructions, results in more unexpected interactions and more ambiguities.

Thus, as grammar coverage increases, syntactic ambiguity also increases. It is not possible to eliminate this ambiguity in the syntax without loosing coverage or suppressing correct parses. In addition, although the range of possible parses for a sentence is interesting from a theoretical perspective, most applications only want to have one parse, the correct one in a given context. As such, this ambiguity is best dealt with by a filter on the output of the syntax. This filtering effect can be obtained from a number of sources. Semantics and world knowledge (Stone, chapter 8, this volume) can be used to process the output of the syntax and eliminate certain incongruous parses. Consider the sentence *I saw the bug with the microscope*. In the syntax, the PP *with the microscope* could either attach to the object *the bug* or the VP *saw the bug*; both are syntactically valid. However, world knowledge tells us that it is more likely that the seeing event involved the microscope than that the bug was carrying a microscope. A second approach to ambiguity is to use statistical methods to determine which of the parses is the most probable (see Callison-Burch and Osborne, chapter 7, this volume). By doing this, a ranked set of parses, which can be reduced to the one most probable parse, is produced and can then be used as input to applications.

4.2.5 Probabilistic Techniques and Grammar Writing

As mentioned in the previous section, probabilistic techniques can be used in conjunction with deep grammars. In general, they are used to manage the ambiguity that results from broad coverage. In this section we describe how probabilistic methods can be combined with grammar writing. Callison-Burch and Osborne (chapter 7, this volume) discuss probabilistic techniques in language engineering in detail; a general discussion can be found in Jurafsky and Martin (2000) and a more detailed one in Manning and Schütze (1999).

One way to use probabilistic techniques in conjunction with a deep grammar is to have a separate program rank the output of the deep grammar based on which of the outputs is most probable (Johnson et al., 1999; Riezler et al., 2002). Under this approach, first the deep grammar is used to parse a sentence. With a large scale grammar operating on real world data, this can result in thousands of parses. Then the probabilistic ranking program takes these parses as input and choses one as the most probable. This most probable parse can then be used as the input to applications. As deep grammars gain in coverage and hence in ambiguity, this technique is very promising for allowing deep grammars to be used on large scale real world data.[7]

Although not a statistical approach, a version of Optimality Theory (Kager, 1999), which involves ranked constraints, can be used with a grammar to specify the more probable parses and hence to solve some of the same problems that statistical approaches are used for (for a discussion of the relation between the parsing-based application of Optimality Theory marks and Optimality Theory in the theory of grammar, see Kuhn, 2001). Such an approach has been implemented in the XLE system and is used by the LFG grammars in the ParGram project (Frank et al., 2001; King et al., 2001). Optimality Theory allows the grammar writer to mark certain constructions or lexical items as dispreferred and others as preferred. As such, rare or ungrammatical constructions can be marked in such a way that they only surface when there is no other option. For example, in order to increase robustness, the grammar may need to parse mismatched subject-verb agreement since this is a common error found in natural language texts. An example is shown in (34a), which contrasts with the grammatical (34b).

(34) a. *The men arrives.
     b. The man arrives.

In order to allow the sentence in (34a) to parse, the subject agreement features on the verb must be weakened so that *arrives* does not require a third singular subject. However, it is undesirable to have them weakened all the time. So, a special mark is put in that allows the grammar to only use that part of the grammar if no other parses are found. The subject agreement for the verb *arrives* would now look as in (35).

(35) ($\uparrow$SUBJ NUM)=sg   `disjunct 1`
($\uparrow$SUBJ PERS)=3
or
OTmark: NoSubjVerbAgreement   `disjunct 2`

When parsing (34b) the first disjunct in (35) is chosen since the second disjunct is dispreferred. However, when parsing (34a) the first conjunct results in a conflict when the features of the subject and verb try to unify, as was seen in the discussion on feature unification (sec-

[7] Another possibility is to combine the probabilities more directly into the parser for the deep grammar. This approach is being taken by Manning (2000) with respect to Probabilistic Head-driven Parsing.

tion 4.2.1; e.g. (20)). So, although the second disjunct is dispreferred by the grammar specification due to the Optimality Theory mark, it is this disjunct which is chosen. The choice of this disjunct results in a well-formed structure; if needed for the application, the inherent ungrammaticality can also be marked in the structure by a feature such as STATUS *ungrammatical*. This is shown in (36).

(36) *The men arrives.

$$\begin{bmatrix} \textsc{pred} & \text{'arrive} < \textsc{subj} >\text{'} \\ \textsc{subj} & \begin{bmatrix} \textsc{pred} & \text{'man'} \\ \textsc{specifier} & \text{the} \\ \textsc{num} & \text{pl} \\ \textsc{pers} & 3 \end{bmatrix} \\ \textsc{tense} & \text{present} \\ \textsc{status} & \text{ungrammatical} \end{bmatrix}$$

The previous example showed how certain constructions can be dispreferred so that they only occur when there is no other option. The same technology can also be used to mark certain constructions as being more preferred than others. Consider the German sentence in (37).

(37) Hans sieht Maria. (German)
Hans sees Maria
'Hans sees Maria.' or 'Maria sees Hans.'

Due to the relatively free word order in German, this sentence can have two readings, one in which Hans is the subject and Maria the object and the other in which Maria is the subject and Hans the object. However, speakers generally interpret the noun phrase in initial position as the subject. So, we would like the grammar to prefer this reading. We can encode this preference by an Optimality Theory mark in the phrase structure rules. A vastly simplified rule for German is shown in (38).

(38)
$$\begin{array}{lcccl} \text{S} \longrightarrow & \text{NP} & \text{V} & \text{NP} & \\ & (\uparrow\textsc{subj}){=}\downarrow & \uparrow{=}\downarrow & (\uparrow\textsc{obj}){=}\downarrow & \texttt{disjunct 1} \\ & \text{OTmark: InitSubj} & & & \\ & \textbf{or} & & \textbf{or} & \\ & (\uparrow\textsc{obj}){=}\downarrow & & (\uparrow\textsc{subj}){=}\downarrow & \texttt{disjunct 2} \end{array}$$

When sentence (37) is parsed, the first NP *Hans* can be either a subject or an object according to the annotations on the rule in (38). However, the Optimality Theory preference mark InitSubj will guarantee that it surfaces as a subject. Although this blocks a syntactically legitimate reading of the sentence, this reading is rare in practice and thus the tradeoff in greatly reduced ambiguity is considered worthwhile.[8] The initial NP object option will only be chosen with nouns that cannot be subjects for some other reason, e.g., ones with overt accusative (objective) case marking. Thus, having a system like the Optimality Theory (dis)preference marks allows the grammar writer to increase coverage and robustness while controlling ambiguity and overgeneration.

4.2.6 Lexical Knowledge

Beyond the coding of phrase structure rules in conjunction with context sensitive annotations (and, perhaps, Optimality Theory marks), the development of broad scale computational grammars requires a number of resources without which the grammar could not exist. One such resource is the lexicon. In addition to thousands of nouns, verbs, and adjectives, every language contains a finite set of functional elements such as auxiliaries, negation markers, complementizers, and conjunctions. These lexical items must be hand-coded, as they contain idiosyncratic and unique information.

Another difficult part of creating a large-scale lexicon is the information needed to determine subcategorization frames and the lexical semantics which plays a role in the syntax. This is primarily true for verbs, but also applies to certain nouns and adjectives. For example, a verb like *eat* can be either intransitive or transitive, but not ditransitive, as seen in (39).

(39) a. I ate.
b. I ate the cake.
c. *I ate him the cake.

In languages like German, all of the verbs' complements have to be marked with a particular case. Furthermore, while verbs like *go* cannot be passivized, verbs like *eat* can be. Finally, in languages like German and the Romance languages, the lexical semantics of the verb determines which auxiliary is selected in periphrastic constructions.

This and other information is part of the lexical entry of the verb. Lexicons are generally stem or *lemma* based. The lemma is a canonical form of the word; in English this is usually the stem form. That is, only the base form of a given verb, noun or adjective is listed in the lexicon, not all of its morphological instantiations. For example, *push* will be listed, but not *pushes*, *pushed*, or *pushing*. The inflected forms of

[8] In spoken language, differing intonation patterns serve to disambiguate the string. However, a parser processing text does not generally have access to intonational information.

a lemma are derived via a *morphological analyzer*, discussed in section 4.2.7. The existence of a morphological analyzer reduces the potential size of lexicons by eliminating the coding of redundant information. In English, this reduces the size of the verb lexicon by a factor of four as most verbs have four surface forms; for other languages the difference is more dramatic.

Lexicons containing this type of information can be compiled in various ways. The most obvious way is to hand-code this kind of information. However, as verbs (and nouns and adjectives) do not constitute closed classes, the job is not finitely bounded and is furthermore tedious, error-prone, and time-consuming.

The past few years have seen the development of several tools which allow the semi-automatic generation of large lexicons. These tools generally use a combination of statistical methods, corpora searches, and existing (but incomplete) hand-coded resources. For example, the German lexicon within the ParGram effort (Butt et al., 1999) was produced semi-automatically (Eckle and Heid, 1996; Eckle, 1997; Eckle-Kohler, 1998) from a combination of data extracted from corpora, such as machine readable and tagged newspaper texts, and existing resources specifying German verb subcategorization frames, such as SADAW (Baur et al., 1994) and CELEX (Baayen et al., 1995). To give an indication of the size of these lexicons, the current ParGram German verb lexicon consists of more than 14,000 entries (i.e., verb stems; there are over 28,000 verb stem-subcategorization frame pairs).

The semi-automatic development of lexicons based on statistical data extraction methods continues to be a topic of intense research, with new tools being developed on the basis of different combinations of methodology (e.g., Schulte im Walde et al., 2001). This is in part because large scale lexicons are a valuable language resource, whose manual compilation is time-consuming. A large lexicon which can be developed and updated quickly via a combination of freely available basic resources, such as machine readable dictionaries or newspaper texts, with reliable statistical resources is a valuable industrial commodity.

Finally, as lexicons grow and can cover many different types of texts robustly, the ambiguity for each lexical item tends to grow. For example, verbs and nouns may express very specialized meanings in technical manuals or medical texts. These specialized meanings are often inappropriate for the parsing of a more general domain, such as newspaper text. However, as they are in the lexicon, the grammar will consider these readings to be viable possibilities, thus increasing the number of parses for a given sentence. To avoid a potentially exponential increase in ambiguities, many grammars divide up their lexicons according to the domain being treated. That is, there is a core lexicon which is used in all applications. This core lexicon is supplemented by a lexicon of technical or medical terms, for example, depending on the text to be parsed.

4.2.7 Morphological Analyzers

As mentioned in section 4.2.6, lexicons usually associate lexical information with the lemma of that lexical item. In order to identify *go* as the correct lemma for a form like *went*, grammars rely on morphological analysis. At present, morphological analysis has reached quite sophisticated levels, becoming both fast and broad-coverage, which benefits the grammars that interface with them.

Currently, most morphological analyzers or parsers use *finite-state technology* (see Jurafsky and Martin, 2000, chapter 3 for a very good, detailed discussion of finite-state transducers and Beesley and Karttunen, 2003). Earlier systems tended to rely on *stemmers* which attempted to identify the correct stem of a lexical item by applying basic heuristics as to the expected forms of morphemes (Porter, 1980). These stemmers are quick and easy to implement for languages like English which have relatively little morphology. As such they can still be used for applications such as information retrieval which do not need sophisticated morphological analysis. However, they cannot be used for more morphologically complex languages or when more detailed information is necessary.

Morphological analyzers can function autonomously from a grammar. The interface to a grammar is defined via a system of *tags*. Basically, morphological analyzers associate surface forms of words with a stem form of the word and its relevant morphological information (and vice versa). The morphological information is expressed in the form of abstract tags. The tags represent formalism-neutral information in that a given grammar can associate these tags with the syntactic information appropriate to that grammar. As such, morphological analyzers interact with both the deep grammars discussed in this section and the shallow parsers discussed in section 4.3.

Consider the surface form *warning* in (40) as an illustration of how a morphological analyzer can interface with a grammar. The surface form *warning* receives three possible analyses from the morphological analyzer: it can be a progressive verb whose lemma is *warn* or a singular noun or adjective whose lemma is *warning*.

(40) warning
1. warn +Verb +Prog
2. warning +Noun +Sg
3. warning +Adj

Within the ParGram project (Butt et al., 1999), for example, the tags provided by the morphological analyzer are used as the input to a series of (sublexical) rules which parse the output of the morphology. For instance, `warn +Verb +Prog` is a sublexical 'phrase' which consists of the stem and two tags.

(41) the stem form: *warn*
the tags: *+Verb, +Prog*

Each morphological tag is listed and identified in the lexicon. That is, tags are treated as lexical items, like the stems of canonical words. (42) shows entries for the stem form *warn* and the two tags *+Verb* and *+Prog*. The lemma entry for *warn* contains part of speech information, namely that it is a verbal stem (V-stem), and subcategorization information, namely that *warn* is a transitive verb. The subcategorization information is generally drawn from work on semi-automatic lexicon generation as discussed in section 4.2.6.

(42)	a.	warn	V-stem	@(TRANS-VERB warn)
	b.	+Verb	V-tag	
	c.	+Prog	TENSE-tag	(↑TENSE-ASPECT)=progressive

In this example, the grammar associates no information with the *+Verb* tag. This has the effect of ignoring the tag for the purposes of the functional structure, although the tag is still used to build the tree structure. The *+Prog* tag, on the other hand, contributes the information that the verb has progressive aspect; the tag is therefore annotated with this information.

The stem plus the tags are parsed by the grammar via sublexical rules as in (43). The annotations on the lemma and the tag now enter the analysis of the string, just as the annotations on phrase structure rules would.

(43) V $\longrightarrow$ V-stem V-tag TENSE-tag

Finite-state morphological analyzers are quite powerful and can also be written to include information about words that is not, strictly speaking, morphological in nature. For example, proper nouns will often be subdivided into names, locations, companies, etc. with appropriate tags.

It should be reiterated that the rather abstract system of tags allows different interfaces to the grammar. The one described above is tailored for interfacing with an LFG grammar, but other interfaces are possible.

Also note that while the development of a grammar is an unbounded task, the development of a morphological analyzer is a finite task: there are only a finite number of derivational and inflectional morphemes in a language, and the morphological formation rules do not tend to be recursive, unlike the phrasal syntax of a language (although highly agglutinative languages provide some interesting challenges to morphological analysis). Morphological analyzers rely heavily on lists of lexical items in order to perform the basic lookup procedures which result in a morphological analysis. Thus, while the list of lexical items is finite, new words are always being formed and foreign words are incorporated into the language. When one of these words is encountered, no part of speech information is available; so, no morphological analysis is possible and the parse for the sentence will fail.

New forms are particularly prevalant in information extraction tasks where proper names from all over the world are encountered, e.g., when parsing newspaper texts or scanning the internet (Baron, chapter 3, this volume). To avoid this problem, an additional morphological analyzer can be created which guesses the part of speech for otherwise unrecognized forms. For English, such a *guesser* would assume, for example, that a word comprised of a capital letter followed by a string of lower case letters is a proper noun and that a word ending in an *s* is either a verb with a singular third person subject or a plural noun.[9]

4.3 Shallow Grammars

As mentioned above, the ultimate aim of most grammar writing projects is to arrive at a large-scale grammar which can parse massive amounts of natural language (e.g., on-line newspaper texts or web pages) in a reasonably quick time and to do so *robustly*. As also mentioned above, deep grammars which perform analyses of the kind aimed for by linguists have so far had the practical disadvantage that they take too long to produce a parse and that they often fail when confronted with unexpected, unusual or ungrammatical instances of natural language. While some of these problems are being alleviated in more recent approaches to deep parsing (e.g., see the discussion in Butt et al., 1999; Copestake, 2002), a general dissatisfaction with the amount of work it takes to write a deep grammar as compared to its usefulness in certain applications has motivated researchers to look into alternative parsing methods.

These methods have come to be known as *shallow parsing*. Rather

[9] A further resource needed for parsing in general is a *tokenizer*, which takes an input string and pre-parses the punctuation, normalizes capitalization, and recognizes special symbols. We do not discuss tokenizers any further here, but see: (Megerdoomian, chapter 6, this volume) for some issues involving tokenization in corpora, Karttunen et al. (1996) on tokenization in general, and Asahara and Matsumoto (2000) on the ChaSen tokenizer for Japanese.

than supplanting deep analyses, current approaches can integrate shallow methods with the deep approach in order to produce grammars which are robust and fast as well as intelligent.

Many shallow parsers are essentially sophisticated versions of part of speech taggers (section 4.3.1): they use stochastic part-of-speech tagging tools to provide a first approximation at the analysis of the sentence (section 4.3.2). Another method that has gained popularity in the last few years is that of identifying useful *chunks* of a natural language string, even if a deep or complete analysis of the entire string cannot be provided (section 4.3.3). These kinds of shallow grammars are extremely useful for certain applications. For example, they are used extensively in information extraction and related applications such as question answering.

As already mentioned, one of the major advantages of shallow grammars is their robustness. For any input, no matter how "ungrammatical" or unusual, the grammar will produce an output. This is in stark contrast to most deep grammars; however, as discussed in section 4.2, deep grammars have begun to incorporate certain shallow parsing techniques to improve their robustness.

4.3.1 Part-of-Speech Tagging

Part-of-speech (POS) tagging involves taking a natural language string and determining the part of speech of each word (and each piece of punctuation) in the context of the string. Consider the sentences in (44) or the famous pair in (45), furnished by the Marx brothers.

(44) a. The ducks flew off.
     b. The boxer ducks to avoid a hit.

(45) a. Time flies like an arrow.
     b. Fruit flies like a banana.

Both sentences in (44) contain the word *duck*. However, in (44a) *ducks* is a noun referring to a type of bird, while in (44b) it is a verb referring to a type of motion. The same kind of ambiguity is found in (45), where the word *flies* either denotes a noun or a verb and *like* either a verb or a preposition.

A good POS tagger marks this difference by noticing the surrounding context of the words. So, in (44a) *ducks* is immediately preceded by a determiner *the* and followed by a verb *flew*; in (44b) it is preceded by a noun *boxer* and followed by a formative *to* which introduces an infinitival. The surrounding words thus provide clues as to the possible POS one could assign to *ducks*. One possible POS tagging of these sentences is shown in (46); note that *ducks*, *off*, *a*, and *hit* could all have other POS tags in different environments.

(46) a. The/DET ducks/N flew/V off/ADV
b. The/DET boxer/N ducks/V to/PREP avoid/V a/DET hit/N

Good POS taggers achieve an accuracy rate of over 90%, with the better taggers recording an accuracy rate of 96% to 97% for English; some taggers for Japanese are over 99% accurate (Asahara and Matsumoto, 2000). Because POS taggers work with stochastic methods, one important component of a POS tagger is the type of tag set which is used. Developing the right tag set for a language is something of an art. It involves a combination of linguistic knowledge and fine judgement of how many differentiations are useful for improving the stochastic accuracy of a POS tagger. The tag set can also be influenced by the application for which it is intended; some applications require much more detailed tags than others.

Tag sets are crucial to treebanking tasks since they are used as the bottom leaves of the phrase-structure trees to be annotated from the corpus. As such, specialized tag sets, based on POS taggers, have been created for treebanking tasks. The pioneering TAGGIT system used for the Brown corpus (Francis, 1964; Francis and Kučera, 1982) included 87 different labels or tags. The now widely used Penn treebank tagset (Marcus et al., 1993) uses only 36 different tags (excluding tags used for punctuation): this was found to increase accuracy significantly (the accuracy of the Brown corpus tagging via TAGGIT was around 77%). Some typical tags are illustrated in (47). As can be seen, the POS tags do not always correspond to the traditional parts of speech.

The tagset most commonly used for German is the STTS (Stuttgart-Tübingen Tagset). It includes 54 tags. The 18 extra tags (as compared with the Penn Treebank tagset) primarily deal with the more complex verbal morphology in German and with various pronominals and adverbials (Schiller et al., 1999). Well-known taggers and tag sets include the Xerox tagger (Cutting et al., 1992), which uses a finite-state lexicon and a morphological analyzer in conjunction with stochastic methods. For further discussion and readings see chapter 8 in Jurafsky and Martin (2000). While a number of excellent POS taggers are available for English and some of the European languages, the demand for POS taggers in languages outside of this set has been increasing steadily.[10]

[10]The methodology for building POS taggers continues to improve. One recent direction is the use of neural nets, e.g., Nakamura et al. (1990) and Schmid (1994).

(47)

Tag	**Use**	**Example**
CC	Coordinating Conjunction	*and*
CD	Cardinal Number	*2*
DT	Determiner	*the*
NN	Noun, singular or mass	*notebook*
NNS	Noun, plural	*notebooks*
RP	Particle	*up*
TO	*to*	*to*
VB	Verb, base form	*appear*
VBD	Verb, past tense	*appeared*

In addition to being used to determine POS, POS taggers often double as stemmers. Stemmers take the surface form of a word (i.e., the one found in the natural language string) and turn it into a canonical, stem form. In this way, POS taggers are an implementation of a morphological analyzer in that they attempt to recognize and to produce the correct forms of words. Although this is a relatively simple task in English, it is crucial in languages with substantial inflectional morphology in which each stem form has numerous surface forms. An English example is shown in (48); note that the stem form can also be a surface form (see also the discussion of stems and lemmas in section 4.2.7).

(48) a. Stem form: push
  b. Surface forms: push, pushes, pushed, pushing

Stemmers can simply canonicalize surface forms to the stem. As mentioned previously, this is all that is needed for certain applications, such as simple information extraction techniques. However, stemmers can also provide information as to the grammatical category of the stem when it has that surface form. This grammatical information can be presented in the form of tags that follow the stem. Some possible tagged forms for the surface forms in (48b) are shown in (49).

(49) a. push: push/N/SINGULAR, push/V/NON-3SINGULAR
  b. pushes: push/N/PLURAL, push/V/3SINGULAR
  c. pushed: push/V/PAST-PASSIVE
  d. pushing: push/V/PROGRESSIVE

In (49b) the form *pushes* can either be a plural noun (*They gave it six pushes.*) or a verb with a third singular subject (*He pushes it.*). Although many applications only use the POS information, the more detailed information can be used, for example, as input to deep grammars (section 4.2).

Since POS taggers are used in a variety of applications, such as spell checkers and information mining systems, we recommend that all grammar writers become familiar with them even if they will not be writing a POS tagger themselves.

4.3.2 Smart Annotation

A more sophisticated use for POS taggers is as the core engine for smart tree annotations. The Penn Treebank (`http://www.cis.upenn.edu/treebank/home.html`), for example, used the tagset and tagger discussed above to provide a "first pass" at parsing a natural language string. A human annotator then went over the output of the tagger, corrected the tags where necessary, and bundled the tagged elements into constituents (Marcus et al., 1993).

The structure arrived at by this combination of stochastic POS tagging and human analysis can be represented by means of a tree, especially when the POS tagging is combined with a parser. These tree analyses are stored in what is called a *treebank*. A treebank represents a potentially valuable language resource: annotated corpora of texts (see Megerdoomian, chapter 6, this volume). The annotated corpora most often represent a *shallow parse* of a given natural language string, though efforts to build LFG- and HPSG-based treebanks have taken shape over the last few years (e.g., the TIGER project at IMS Stuttgart (Brants et al., 2002) and the Redwoods (2002a; 2002b) project at Stanford University). These shallow parses can be used to investigate linguistic phenomena such as the distribution of subjects vs. objects or the occurrence of certain prepositional phrases.

Treebanks are useful resources for other types of research, such as grammar induction (section 4.4) and grammar testing (section 4.5). The Penn Treebank is the most well established treebank to date and thus provides a resource for researchers working on English. The production and storage of treebanks for other languages is one of the tasks which currently engages the computational linguistic community.

In recent years, the human labor involved in making treebanks has decreased markedly. This is due in part to more sophisticated user interfaces that are available with some of the taggers, in part to the more sophisticated morphological analyzers (section 4.2.7), and in part to the application of statistical methods for grammar induction.

Several smart annotators exist for German. One example is the *TreeTagger*, which has been used to tag German, French, Italian and Greek texts (Schmid, 1995). Another good system is *Annotate* (Brants and Plaehn, 2000). This tool allows for the interactive semi-automatic annotation of corpora. In addition to allowing for basic stochastic POS tagging, *Annotate* also provides some interactive parsing via cascaded Hidden Markov Models, thus building up constituency information as well, as shown in Figure 1.

This system also allows the user to define new phrase structure categories on-line, thus providing help in the manual annotation of analyses. Furthermore, the system "watches" the users as they treebank; through watching and keeping track of the decisions and choices made by the user statistically, the system "learns" the grammar rules the user is following. The practical effect is that the system provides the user with what it thinks is the right constituency analysis and guesses correctly about 70% of the time, again relieving the user of some of the manual labor involved in treebanking.[11]

A system such as *Annotate* thus provides a semi-automatic shallow parser which can then be used to create a more structured treebank. Other kinds of shallow parsers use mainly finite-state technologies (e.g., Chanod and Tapanainen, 1996). A further kind of shallow parser is based on the notion of *chunk parsing*.

4.3.3 Chunk Parsing

The notion of parsing only chunks of a sentence in order to achieve robustness was pioneered by Steve Abney (1991, 1996a, 1996b). The basic idea behind chunk parsing is to provide a robust parsing mechanism which concentrates on identifying the major "chunks" in a sentence, without having to worry about the attachment ambiguities which have plagued traditional deep analysis grammars. Note that these chunks are usually labeled with basic category information. A PP-attachment problem is illustrated in (50). A deep grammar will provide two analyses for this sentence, one in which the PP *with the telescope* is attached high, i.e., has scope over the verb, and one in which the PP modifies just the NP *the monkey*.

(50) Shankar saw the monkey with the telescope.

A chunk parser, in contrast, returns only one analysis, something of the sort shown in (51).

(51) $[_{\mathrm{DP}}$ Shankar$]$ $[_{\mathrm{VP}}$ $[_{\mathrm{V}}$ saw $[_{\mathrm{DP}}$ $[_{\mathrm{DET}}$ the$]$ $[_{\mathrm{NP}}$ monkey$]]]]$ $[_{\mathrm{PP}}$ $[_{\mathrm{P}}$ with$]$ $[_{\mathrm{DP}}$ $[_{\mathrm{DET}}$ the$]$ $[_{\mathrm{NP}}$ telescope$]]]$

As seen in (51) a basic chunk parser does not worry about what overarching categories to assign to relatively complex syntactic formations.

[11]Treebanking software has also been developed for the LKB (Linguistic Knowledge Building) system (Copestake and Flickinger, 2000) for HPSG as part of the Redwoods project. This system allows for dynamic treebanks of full linguistic analyses. These can be updated automatically as the grammar they are based on changes.

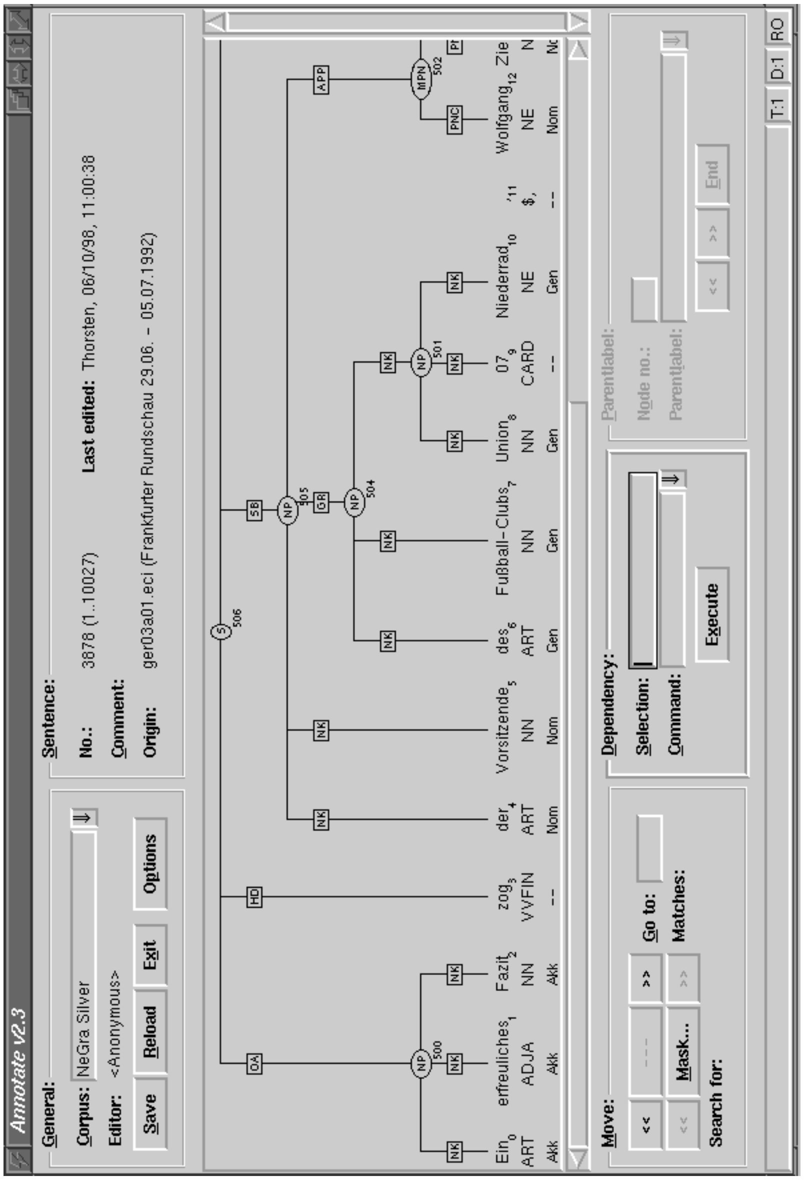


FIGURE 1 Screen shot of the *Annotate* system

Instead, it identifies basic chunks and returns these. Another example is shown in (52), taken from Abney (1991).

(52) [I begin] [with an intuition]: [when I read] [a sentence], [I read it] [a chunk] [at a time].

Abney cites psycholinguistic evidence from sentence processing which appears to confirm his intuition that a good way to deal with sentence analysis is to begin with the identification of chunks. Questions that arise immediately are how to define a chunk and how to tell where a chunk begins and ends. These questions are addressed by Abney and are the subject of continuing research.

Like most other shallow parsers, chunk parsers depend heavily on POS taggers to provide the initial information that helps the parser identify chunks. For example, if a POS tagger correctly identifies the *with* in (50) as a preposition (P), then the chunk parser can group the following DP together with the preposition in a prepositional phrase (PP).

This preprocessing via POS information and the successive regrouping of individual words into larger and larger chunks is generally accomplished by a cascade of finite-state transducers whereby the output of one transduction is used as the input for another (Abney, 1996a). The reliance on well-understood computational tools such as POS taggers and finite-state transducers creates a good development base for chunk parsers: they work with well-known techniques to identify basic chunks and can thus be relied upon to handle large amounts of data in a robust and useful manner. See Jurafsky and Martin (2000) for an explanation of the formal properties of finite-state transducers and how they work, and also section 4.2.7 for an application within morphology.

One particular industrial application of chunk parsers is that of entity finders. Entity finders are used to locate certain types of noun phrases and classify them. This is especially useful for the extraction of specialized terminology from technical texts such as manuals, patents, or medical texts, and for information extraction from a large corpus such as the web.

The identification of noun phrases is known as noun chunking (Schiller, 1996; Schmid and Schulte im Walde, 2000). Consider the sentence in (53) and some possible outputs in (54).

(53) President Clinton visited the Hermitage in St. Petersburg.

(54) a. [President Clinton]$_{\mathrm{NP}}$ visited [the Hermitage]$_{\mathrm{NP}}$ in [St. Petersburg]$_{\mathrm{NP}}$.

b. [President Clinton]$_{\mathrm{NP}}$ [visited the Hermitage in St. Petersburg]$_{\mathrm{VP}}$.

The parse in (54a) is an example of noun phrase chunking. It picks out only the noun phrases, ignoring the verb *visited* and the preposition *in*. In contrast, the parse in (54b) provides a basic structure for the sentence, marking the subject noun phrase and the verb phrase.

The very basic noun chunking seen in (54a) is usually augmented with more details to perform entity extraction, just the way that POS tagging can be augmented with further grammatical information. One way to augment an entity extractor is to provide information as to the type of entity found. These usually work with proper nouns, ignoring common nouns and non-nominals. So, the three entities found in (54a) might be further labeled as in (55); for domain specific uses even more finely grained distinctions can be made.

(55) a. President Clinton/NP/PERSON
b. the Hermitage/NP/ORGANIZATION
c. St. Petersburg/NP/LOCATION/CITY

Another way to use an entity extractor is to extract domain-specific terminology from documents. This is particularly useful in technical and scientific domains which often use very specific, multiword terms. An extreme example is provided by biology. The biochemical domain in particular employs specialized terminology which is further characterized by a specialized noun phrase syntax. Identifying these specialized terms is important both to obtain lists of the terms and, more frequently, to determine which terms appear in which contexts.

4.4 Machine Learning and Grammar Induction

Another effort which is connected with the general area of grammar writing is the idea of machine learning as applied to grammar induction. The idea behind grammar induction is that given a suitably annotated corpus, it should be possible to program a computer to *learn* or induce a grammar for that corpus. Given the amount of time and training required to write grammars manually, grammar induction via machine learning is a growing field which has led to workshops and special journal issues devoted to this topic (e.g., Cardie and Mooney, 1999).

The most popular resource for this kind of research is again the Penn treebank (Marcus et al., 1993) which is used for creating treebank-derived probabilistic parsers. Within LFG, for example, this treebank is being used as the core corpus for the induction of LFG grammars

(Frank, 2000; Sadler et al., 2000). One immediate result of this type of effort is the creation of further language resources, namely LFG analyses of sentences which have themselves been banked and are available for further study. For example, Cahill et al. (2002) have implemented an automatic annotation algorithm for the Penn treebank which associates 98% of the trees in the treebank with high quality f-structures.

These f-structures are not just of potential use for a specialized LFG community as they can be converted to Quasi-Logical Forms (QLF) (Cahill et al., 2003), a semantic formalism which allows a statement of semantic relations via logical forms without dealing with scope ambiguities (Alshawi, 1992). QLF representations have proven to be very useful in natural language processing in that they allow a relatively quick analysis of semantic relations of the nature of "who did what to whom and when."

The f-structures resulting from the automatic annotation algorithm can be further used to automatically extract subcategorization frames and in turn to construct a broad coverage LFG grammar of the Penn treebank. Cahill et al. (2002) have two architectures for inducing this broad coverage grammar from the f-structure dependency bank. The first is a pipeline architecture in which they extract a probablistic context-free grammar from the WSJ, parse new text with it, annotate the resulting parse trees using the automatic f-structure annotation algorithm, and finally generate the f-structures for the new text. The second is an integrated architecture in which they annotate the WSJ trees with the automatic f-structure annotation algorithm, extract an annotated probablistic context-free grammar from the annotated trees, parse the new text, and finally generate the f-structures for the text. These two basic architectures can be adapted to other types of grammars and treebanks.

Another type of effort (again within LFG), has been to use *data-oriented parsing* (DOP) methods (Bod and Kaplan, 1998) to induce LFG analyses from an annotated corpus like the Penn treebank. DOP models of natural language hypothesize that human language processing works with representations from past language experiences rather than with abstract linguistic rules. These models operate by decomposing the given representations into fragments and recomposing the fragments to analyze new utterances. A probability model selects the most likely analysis from all possible analyses of an utterance. DOP models can be based on simple tree representations or on more detailed representations, such as pairs of LFG trees and corresponding functional structures. There is also recent work on HPSG DOP (Neumann and Flickinger, 2002; Neumann, 2002). For a general introduction to data-oriented parsing see Bod et al. (2002).

Grammar induction can also be done based on plain text, perhaps in conjunction with a POS tagger or a similar shallow parser. This is grammar induction in the narrow sense of the term. The advantage of this approach is that it is possible to obtain large amounts of data and not be limited to existing annotated corpora such as the treebanks. As is the case with statistical approaches and other automatic learning, grammar induction works better the larger the amount of data which is available. See (Megerdoomian, chapter 6, this volume) for more on large-scale corpora construction and (Baron, chapter 3, this volume) on the language of the internet.

However, certain types of grammars are difficult to induce from texts; generally, the more detailed the output of the grammar is to be, the harder it will be to use unmarked/unprocessed texts to induce the grammar. As such, much grammar induction is done at least in part on marked up (i.e, tagged or annoted) text. Also note that the value of linguistic knowledge is often not highlighted in such approaches, but is extremely important since in most cases it is linguistic knowledge that underlies the machine learning. For more information on grammar induction, see Charniak (1993) and Manning and Schütze (1999).

4.5 Grammar Testing and Evaluation

Grammar writing is like any large, evolving major software project: it requires disciplined testing and evaluation tools to be successful. Good testsuites and testing practices and solid evaluation tools are invaluable in any large-scale grammar writing project. In this section we discuss the basics of testing, evaluating, and documenting grammars. The TSNLP (Testsuites for Natural Language Processing)[12] project, and its successor the DiET (Diagnostic and Evaluation Tools for Natural Language Applications) project, deal with issues of testing and testsuites in detail; see also Nerbonne (1998) which includes a paper on TSNLP and Oepen and Flickinger (1998).

4.5.1 Testsuites

Testing the grammar is usually done using testsuites. Testsuites are lists of grammatical and ungrammatical sentences and other constituents that the grammar should be able to parse (or, in the case of the ungrammatical sentences, should not receive an analysis). In this section we first discuss some general testing policies and then types of testsuites.

[12]Information about the TSNLP project can be found at the DFKI web site `http://cl-www.dfki.uni-sb.de/tsnlp/`.

4.5.2 Testing Policies

We suggest three basic policies for testing; the three are meant to be used in conjunction with one another. Having an agreed-upon testing policy is particularly important when there are multiple grammar writers whose changes to the grammar must be merged. Note that as processing speed and parsers improve, the grammar writer's ability to run detailed testsuites increases greatly.

Policy 1: The testsuites should be written first. When starting on a new phenomenon, the grammar writer should create a basic testsuite for that phenomenon before beginning any grammar writing. This has two advantages. The first is that the grammar writer will have a better idea of what the phenomenon entails. This is especially true if some of the data is extracted from corpora. The second is that there will already be a testsuite in existence when the new rules need to be tested. The pre-grammar writing testsuite can be augmented during development by any sentences that were tested during development. Such sentences tend to be simple, but crucial.

For example, if relative clauses are about to be added to the grammar, the first step would be to list as many relative clause types as possible (e.g., relative pronoun as subject, as object, reduced relatives, extraposed relatives). Then a number of relative clauses can be extracted from an existing document by searching for relative pronouns; reduced relatives, which do not contain relative pronouns, can be extracted from a treebank. Finally, all the phrases that are parsed during development can be added, such as the relative pronoun in isolation.

Policy 2: Records of past grammar performance should be kept. Keeping records of past performance is extremely useful for determining progress in grammar development and for determining how long a problem has existed. As grammar coverage increases, the testsuites will grow and the number of sentences receiving parses will increase; storing past test runs provides documentation of this. Another use for such records is to see whether a given sentence has ever parsed successfully. If it has, the records will show when it stopped being parsed and hence help indicate what grammar changes might have caused the problem or if it has gained spurious parses.

Policy 3: Testing should be required before releasing a grammar revision. After any change to the grammar, testsuites for both the new phenomena and for the entire grammar must be run. This detects any interaction of the new phenomena with the existing rules and any unintended interactions with other grammar writers changes. Any detected problems must be fixed and the entire testsuite rerun before releasing the changes. This policy can either be enforced by using programs which will not allow a grammar release without a test run or can simply be a policy observed by the grammar writers. Although this detailed testing can be tiresome, especially for minor bug fixes, it saves time in the long run since there will be fewer bug reports from the grammar users.

4.5.3 Types of Testsuites

There are three basic sources for testsuites. First, the grammar writer or someone else can make up the data for the testsuite. This is essential for the maintenance of the grammar since such testsuites are tuned to the intended coverage of the grammar. In addition, there are standard testsuites available, such as the HP or TSNLP testsuites, which cover a large number of linguistic phenomena. These testsuites are interesting in that they show what other grammar engineers feel is important in grammar coverage. Finally, real world data can be used in testsuites (see Megerdoomian, chapter 6, this volume). For example, if the grammar is meant to parse technical manuals, a typical manual can be broken into sentences or other constituents and parsed as a testfile. Although such testsuites are difficult to use for basic development, they are invaluable in hunting down unintended interactions between rules (and they can be very enlightening as to what is considered grammatical). In addition, they can be used to prioritize phenomena to cover and to discover relevant non-core constructions which do not appear in standard testsuites.

When writing a testsuite, the test items should be tested in isolation and in context, e.g., in a sentence. The isolated versions make sure that the basic construction is working, while the contexted versions detect interactions with other rules. For example, when testing adjectives, first the basic adjective and adjective rule should be tested and then the adjective in context, as in (56). Note that the items in (56) are not parses, but instructions to the parser on what it should parse: i.e., *flimsy* should be parsed as an adjective, *the very flimsy box* as an NP.

(56) a. ADJ: flimsy
b. ADJP: very flimsy
c. NP: the very flimsy box
d. S: It is very flimsy.

In addition to having testsuites which focus on a particular phenomena, larger testsuites that combine phenomena should be created. As a first step, a master testsuite can be created by combining all of the specialized testsuites (for maintainability, it may be easier to write a script that calls the individual testsuites instead of copying them into one large file). If such a testsuite is then combined with actual corpus examples, a good idea of the grammar's performance can be obtained.

Another way to create a testsuite that covers the grammar is to extract a testsuite from the comments in the grammar (Bröker, 2000). That is, for each (sub)rule in the grammar, a comment is created with an example of what that (sub)rule parses. These are then collected and parsed as a testsuite. Such testsuites give a good idea of current grammar coverage, especially when used with more specialized testsuites that show the intended coverage of the grammar.

In addition to formulating testsuites of items that are covered by the grammar, testsuites of items that are ungrammatical should also be created. These avoid problems with overgeneration in which the grammar parses strings which it should not. For example, in a testsuite on verb agreement, in addition to forms like those in (57), ungrammatical forms like those in (58) should also be included.

(57) They run. She runs.

(58) *They runs. *She run.

Note that part of the testsuite should include notation to mark the input as ungrammatical (if it is indeed ungrammatical). This mark should be one which the parser knows how to deal with or which does not interfere with the normal processing of the parser. Also note that although a sentence may be ungrammatical, it may in fact occur in input such as a newspaper text that the parser must deal with. Furthermore, a sentence may turn out to be well-formed under some unintended reading. As such, testsuites are often parametrized to account for these differing possibilities.

For example, when testing the English auxiliary system, a sentence like (59) should be blocked.

(59) *They have appearing.

This sentence is ill-formed under the reading where the *have* is an auxiliary and the *appearing* is a main verb (cf. *They have appeared.*). However, it is well-formed under the reading where *have* is a main verb and *appearing* is a gerund (cf. *His appearing bothered me.*). Ideally a testsuite should therefore record on what grounds a given sentence was blocked.

With all types of testsuites, it is important to determine not only that a test item gets a parse, but that that parse is the correct one. One way to do this is to randomly parse some of the test items and manually inspect them to determine whether the parse is correct. However, when possible this checking should be automated. A basic way to do this is to encode as part of the test item information that must also be checked (Kuhn, 1998). For example, a testsuite of verb subcategorization frames can include in it which noun is the subject, which the object, etc. Two possible encodings are shown in (60), one which has the information as part of the string and one which has it after the string.

(60) a. [SBJ They] pushed [OBJ it].
b. They pushed it. [they=SBJ; it=OBJ]

Once a sentence can be parsed by the grammar, another approach is to save the correct parse structure, or a stripped down version of it, and have it compared to the current parse structure. For example, a system which includes semantic analysis might compare just the semantics of the sentence, with the assumption that the correct semantics could only be achieved if the correct syntax was also present. Regardless of the approach, running testsuites, even extensive ones, without checking for correctness of the parses can lead to trouble.

4.5.4 Other Testing Mechanisms

There are two other testing mechanisms that we briefly mention here: checking which rules are used and running a generator.

Ideally, every (sub)rule in a grammar is used for some construction. To make sure that rules have not fallen out of use, a testsuite that is supposed to test the entire grammar can be run and a record kept of which rules in the grammar are used in the parsing. If rules are not being used, either the testsuite is incomplete or the rule is not needed and should be pruned to increase efficiency.

Most grammars are written in the context of parsing. That is, the grammar writer chooses a sentence, parses it, and checks if the output is correct. However, even when testsuites checking for ungrammatical parses exist, such grammars often overgenerate: they allow many ungrammatical strings that the grammar writer is unlikely to think of. For this reason, having a generator, as well as a parser, can be a useful tool for testing, even if the grammar is not being used for generation. The basic idea is to parse a sentence and run the output through the generator. If more than the input string is returned, then the grammar may need to be constrained, although if all of the output strings are grammatical, then no changes may be necessary. Note that in a free word order language like German, overgeneration is to be expected. For example, since German allows extraposed relative clauses, the generation of relative clauses will include the extraposed version in addition to the non-extraposed one.

4.5.5 Parse Failure and Problem Identification

Once a testsuite is run, the problem is then how to most efficiently locate and fix the problems that were found. How best to do this in part depends on the grammar being used and the preferences of the grammar writer. In general, as with other big software projects, the bigger the grammar and the less familiar the grammar writer is with the grammar, the harder it is to find the problem. Below we briefly outline a few techniques that are applicable to most grammars.

The basic idea behind locating the problem is to first determine what is working and thus, by process of elimination, determine what is not. This can be done in a variety of ways. First, the grammar writer should determine that the parser recognizes all the lexical items in the sentence correctly; this can be done by parsing them individually or by substituting a known word in for one that is suspected of causing the problem. For example, if the sentence is *I went to Booneville* and the grammar writer suspects the grammar does not recognize *Booneville*, then *I went to London* can be parsed. If the *London* sentence parses, then either *Booneville* is not recognized or it is not assigned the correct analysis. Second, the grammar writer can simplify the sentence until it parses. Some ways to do this include: removing modifiers like adverbs, putting pronouns in for nouns, parsing only one conjunct of a coordination. Once the sentence parses, the phrases can be added in until the one causing the problem is located. Similarly, the subconstituents of a sentence can be parsed individually. If they all parse, then the problem is in how they combine.

Debugging of this type cannot be eliminated (and, in fact, can be quite interesting). However, it is possible to avoid some of the more common problems. One way is to have a program to check that rules are properly formatted and that they are free of obvious typos before any sentences are parsed. In addition, the parser may check for inconsistencies when it loads the grammar. Use of templates also helps in avoiding typos and ensuring consistency. A second way is to document all rules; what is obvious now will not be tomorrow. Documentation is especially crucial if there is more than one grammar writer. Finally, keeping a list of grammar changes and known problems is useful; if a known bug in the grammar is encountered frequently, it should become a high priority for the grammar writer to fix.

4.5.6 Analysis of Text for Linguistic Patterns

Analyzing a text for linguistic patterns is difficult. However, if a grammar is being created from scratch, knowing what phenomena need to be included can save time. For general large-scale grammars, using a general purpose testsuite, like the HP or TSNLP testsuites, is useful.

If the text corresponds to a treebank as opposed to just strings, it may be possible to extract linguistic patterns from the treebank itself.[13] For example, knowing what constituents can be daughters of VP can guide the construction of the VP rule and of the verb subcategorization frames. Also, specialized testsuites can be derived in this way since all the instances of a given type of constituent can be formed into a testsuite, e.g., a testsuite of NPs or of PPs. Even without a treebank, a chunker can be run over the text to pick out NPs. Although there are likely to be mistakes in the resulting list, it can still provide a useful guide to grammar development.

For limited domain grammars, analysis of the corpus can be used to derive lexicons. By restricting the lexicon to only the relevant words or word senses, ambiguity is greatly reduced. One way to do this is to run a part of speech tagger over the corpus to obtain a list of nouns, verbs, adjectives, and adverbs. A list of closed class items can also be obtained this way, although the grammar writer might want to reclassify some of the POS tags. With a treebank of a corpus, subcategorization frames can be extracted as well. For example, nouns that subcategorize for *that* clauses (e.g., *the idea that the earth is round*) can be extracted. See section 4.2.6 for more details.

4.5.7 Evaluation

The idea behind *evaluation* is to determine how good the grammar is. There is no industry-wide standard for grammar evaluation, although there are efforts to arrive at such a standard. Some examples of such projects are EAGLES (Expert Advisory Group on Language Engineering Standards) and DiET (Diagnostic and Evaluation Tools for Natural Language Applications).[14] However, one as yet unresolved difficulty is that not all grammars share the same assumptions as to the type and level of detail of syntactic analyses. Furthermore, for special purpose grammars, such general purpose evaluation is not important. That is, a special purpose grammar only needs to parse a specific genre of language, e.g., airline reservation dialogs, and not all of English.

Most evaluation techniques involve treebanks of structures for large natural language corpora. The idea behind such treebanks (e.g., the Penn Treebank) is that a grammar should produce all (and only) the structures found in the treebank; not producing a structure or

[13] In fact, the grammar itself can be extracted from the treebank; see section 4.4 on grammar induction.

[14] More information on these two projects can be found at the EAGLES and DiET web pages: `http://www.ilc.pi.cnr.it/EAGLES/intro.html` and `http://lrc.csis.ul.ie/research/projects/DiET/index.html`.

producing additional structures counts as a failed parse (see Manning and Schütze, 1999 for discussion of using labeled bracketing as an evaluation measure for probabilistic parsers).

However, comparison against tree structures does not work for many types of grammars (and even the most carefully constructed treebanks contain errors and inconsistencies). First, even if the grammar in question only produces tree structures, there may be fundamental differences in analysis. Such differences are usually systematic, e.g., phrase structure nodes have different names or different attachment levels. Second, many grammars either do not produce tree structures or the tree structures are only part of the analysis (e.g., HPSG- and LFG-based grammars). For such grammars, a new type of evaluation is needed. Carroll et al. (1999) have suggested a schema in which what is evaluated is the number of dependencies (grammatical functions) that are correct (see Xia et al., 1998 for a similar approach). An example from the PARC-700 dependency bank (Crouch et al., 2002) is shown in (61b) for the sentence in (61a).

(61) a. The device was replaced.
b. mood(replace::0, indicative)
tense(replace::0, past)
passive(replace::0, +)
stmt_type(replace::0, declarative)
subj(replace::0, device::1)
vtype(replace::0, main)
det_form(device::1, the)
det_type(device::1, def)
num(device::1, sg)
pers(device::1, 3)

In the dependencies in (61b), the relevant syntactic information is represented as triples. For example, *subj(replace::0, device::1)* states that the subject of *replace* is *device*. The indices mark reentrancies (e.g., in *John wants to leave*, *John* is the subject of both *want* and *leave*) and disambiguate multiple instances of the same word (e.g., in *Bagels cost $1 and donuts cost $2*, one instance of *$* will have one index and one another index). Note that this representation, like attribute-value matrices, allows for very detailed syntactic information to be represented (e.g., tense, number, statement type).

Grammars which use tree structures can define certain dependencies via tree position, e.g., an object is an NP under VP, while other grammars may encode this directly, e.g., the subject attribute in an attribute-value matrix. This technique of using dependencies instead of trees is especially useful for evaluating languages with free word order where determining the grammatical function via the phrase structure tree is impossible; for such languages, representations like that in (61) are more suitable. This type of grammar evaluation is gaining credence and standards are being developed for dependency banks; see the proceedings of the Language Resources and Evaluation conferences (LREC) which began in 1998 (e.g., see Bangalore et al., 1998 for discussion). When evaluating against a dependency bank, the dependencies in the grammar output are compared against those in the dependency bank. If the grammar output has dependencies which are not in the dependency bank or vice versa, then the grammar evaluation is penalized. With this approach, evaluation can be customized: evaluation can be done against only certain dependencies, ignoring the others. For example, if the grammar writer is most interested in determining whether the grammar is assigning the correct predicate-argument structure, then the evaluation can be done against only the grammatical function information (e.g., subject, object). A comparison of evaluations using a Carroll et al. (1999) style dependency bank and more traditional labeled bracketing derived from the Penn Treebank are discussed in Crouch et al. (2002) for the output of the ParGram English deep grammar used in conjunction with a stochastic disambiguator.

Finally, note that a specialized testsuite in the form of a treebank or dependency bank can be created for evaluation purposes if the grammar is for a limited domain. The basic idea is to create the ideal parse for each sentence (i.e., a tree or a set of dependencies) and then compare the grammar output to that ideal. This is extremely time consuming, but may be necessary where high accuracy is essential.

4.5.8 Documentation

Documentation is a necessary part of grammar writing. This is especially true for large-scale grammars: the grammar rapidly becomes extremely complex and there is often more than one person working on it. It is useful to think of two types of documentation: one within the grammar itself and one as an overview of the grammar. These are necessarily related as they describe the same grammar.

When working on the grammar, it is invaluable to have documentation of each rule. This documentation should include what the rule does, along with some example sentences. In addition, the subparts of the rule should be marked as to their purpose, with examples. For example, the simplified partitive rule in (62) contains some basic in-grammar documentation in the form of comments; the comments are enclosed in double quotes.

(62) NPpart $\longrightarrow$ "This is the rule for partitive NPs."
"Ex: some of the apples"
"It is called by the main NP rule."

(DetP) "Optional determiner phrase."
Npart "The head of the partitive."
"These are a closed class."
"Numbers are a subset of Npart."
"Ex: five of the apples"

PPpart "The *of* phrase"
"The oblique argument of the Npart."

Finally, it is useful to include comments as to why a particular analysis was chosen. Although this seems immediately obvious when the rule is written, it may not be so transparent at a later date or to another grammar writer.

The documentation just described is useful for grammar writers who have some basic familiarity with the grammar. However, it does not give a good overall picture of what the grammar does. This overview is necessary for people who want to know about the grammar but have not worked on it; this includes grammar writers who join a project once the grammar is already in existence and people deciding whether the grammar provides the basic coverage they want for a particular application. This overview is best accomplished by top down documentation. The basic idea is to describe what phenomena the grammar covers and the types of analyses it provides. A good way to approach this is to start with the main category and describe what it does (e.g., declaratives, imperatives, interrogatives, headers) and then provide more detailed coverage of the individual phenomena (e.g., noun phrases, adjectives, coordination). As with the documentation in the grammar, the overview documentation should include examples; providing, possibly simplified, sample analyses of these examples is also helpful.

One of the major problems with documentation is keeping it up to date. Ideally, the documentation should be updated every time a change is made. Waiting to do major documentation changes all at one time runs the risk of having some changes forgotten. To alleviate these problems, techniques are being developed to help automatically update the grammar documentation. Dipper (2003) has developed a system to automatically extract rules from the grammar and to include them in the top down documentation; this way, the most up-to-date version of the rule is always included in the documentation.

4.6 Summary

Much more could be said about grammar writing; however, we hope to have addressed some of the main issues with respect to grammar development, testing and evaluation. In particular, we covered the following main topics in this chapter:

- There are two basic types of grammars: deep and shallow.
- Deep grammars:
  - Provide grammatical relation information
  - Provide parses spanning the entire sentence
  - Are not robust due to limited grammar coverage
  - Are often slow (but the situation is improving)
- Shallow grammars:
  - Provide limited (often no) grammatical relation information
  - Do not connect all subparts of a sentence
  - Are robust: every input gets an output
  - Are fast
- It is possible to combine deep and shallow parsing techniques to reap the benefits of both. For example, shallow grammars can preprocess a string to restrict the possible parses produced by a deep grammar.
- Testing is extremely important for the maintenance of large scale grammars. Testing is used to determine whether a grammar has the intended coverage and whether it overgenerates.
- Evaluation involves comparing the grammar to other grammars or to a specific task.
- Documentation of the grammar is essential for debugging, for use with multiple grammar writers, and for determining whether the grammar will be useful for a given application.

Finally, we would like to emphasize that in addition to its computational and practical aspects, grammar writing can provide useful input into theoretical linguistics by testing linguistic theories on a large scale.

4.7 Suggested Reading

In addition to the references cited in the body of this chapter, we recommend two general works to learn more about grammar writing.

The first is Jurafsky and Martin (2000). This is a text book on speech and language processing. It is divided into four parts: words, syntax, semantics, and pragmatics. The syntax part is of most immediate use to the grammar writer. However, the word part deals with topics that can immediately interact with the grammar and the semantics and pragmatics parts discuss areas which use the output of the grammar as input. This book is well written, containing an index, a detailed bibliography, and nice discussions of the history of the field.

The second is Butt et al. (1999). This discusses deep grammars, using the LFG ParGram project as an example. It covers the types of syntactic data a broad coverage grammar must consider. In addition, the second part of the book discusses grammar engineering and various tools that can help the grammar writer. It also contains an index and references.

Another recent book is Copestake (2002). This book describes working with the LKB grammar development platform, which assumes HPSG as the underlying formalism. The discussion provides detailed examples, actual grammar writing code and a user manual complete with screen shots.

Since grammar writing, testing, and evaluation is a rapidly advancing field, the web is a natural place to find up to date information. As a starting point we recommend looking at the following institutions' web pages:

Center for the Study of Language and Information (CSLI)
`http://www-csli.stanford.edu/`
DFKI (German Research Center for Artificial Intelligence)
`http://www.dfki.de/`
IMS (Institute for Natural Language Processing) Stuttgart
`http://www.ims.uni-stuttgart.de/`
Linguistic Data Consortium
`http://www.ldc.upenn.edu/`
Linguistic Grammars Online (LinGO)
`http://lingo.stanford.edu/`
Natural Language Processing at the University of Pennsylvania
`http://www.cis.upenn.edu/~linc/home.html`
Verbmobil (Machine Translation Project)
`http://verbmobil.dfki.de/`
Palo Alto Research Center's Natural Language Theory and Technology group
`http://www.parc.com/istl/groups/nltt/`
Institute for Communicating and Collaborative Systems, University of Edinburgh
`http://www.iccs.informatics.ed.ac.uk/`
WordNet
`http://www.cogsci.princeton.edu/~wn/`

References

Abeillé, Anne and Owen Rambow, eds. 2000. *Tree Adjoining Grammars: Formalisms, Linguistic Analysis, and Processing*. Stanford, California: CSLI Publications.

Abney, Steve. 1991. Parsing by chunks. In R. Berwick, S. Abney, and C. Tenny, eds., *Principle Based Parsing*, pages 257–278. Dordrecht: Kluwer Academic Publishers.

Abney, Steve. 1996a. Partial parsing via finite-state cascades. *Journal of Natural Language Engineering* 2(4):337–344.

Abney, Steve. 1996b. Tagging and partial parsing. In K. Church, S. Young, and G. Bloothooft, eds., *Corpus-Based Methods in Language and Speech*. Dordrecht: Kluwer Academic Publishers.

Alshawi, Hiyan, ed. 1992. *The Core Language Engine*. Cambridge, Massachusetts: The MIT Press.

Asahara, Masayuki and Yuji Matsumoto. 2000. Extended models and tools for high-performance part-of-speech tagger. In *Proceedings of International Conference on Computational Linguistics (COLING)*, pages 21–27.

Asudeh, Ash. 2000. A licensing theory for finnish. Unpublished Manuscript, `http://www.stanford.edu/~asudeh/`.

Baayen, R.H., R. Piepenbrock, and L. Gulikers. 1995. The CELEX Lexical Database (CD-ROM). Linguistic Data Consortium, University of Pennsylvania.

Bangalore, Srinivas, Anoop Sarkar, Christine Doran, and Beth-Ann Hockey. 1998. Grammar & parser evaluation in the xtag project. In *Proceedings of the Workshop on Evaluation of Parsing Systems*. Granada, Spain.

Baur, Judith, Fred Oberhauser, and Klaus Netter. 1994. SADAW Abschlußbericht. Tech. rep., Universität des Saarlandes and SIEMENS AG.

Beesley, Kenneth and Lauri Karttunen. 2003. *Finite-State Morphology: Xerox Tools and Techniques*. Stanford, CA: CSLI Publications.

Block, Hans Ulrich and Stefanie Schachtl. 1992. Trace and unification grammar. In *Proceedings of the 14th International Conference on Computational Linguistics (COLING)*, vol. 2, pages 658–664. Nantes, July.

Bod, Rens and Ronald Kaplan. 1998. A probablistic corpus-driven model for Lexical-Functional analysis. In *The Joint Meeting of the 36th Annual Meeting of the Association of Computational Linguistics (ACL) and the 17th International Conference on Computational Linguistics (COLING)*, pages 145–151. Montréal.

Bod, Rens, Remko Scha, and Khahil Sima'an, eds. 2002. *Introduction to Data-oriented Parsing*. Stanford, California: CSLI Publications.

Brants, Sabine, Stefanie Dipper, Silvia Hansen, Wolfgang Lezius, and George Smith. 2002. The TIGER treebank. In *Proceedings of the Workshop on Treebanks and Linguistic Theories*. Sozopol, Bulgaria. To Appear.

Brants, Thorsten and Oliver Plaehn. 2000. Interactive corpus annotation. In *Second International Conference on Language Resources and Evaluation (LREC00)*, pages 453–459. Athens.

Bröker, Norbert. 2000. The use of instrumentation in grammar engineering. In *Proceedings of the 18th International Conference on Computational Linguistics (COLING)*, pages 118–124. Saarbrücken, August.

Butt, Miriam, Tracy Holloway King, María-Eugenia Niño, and Frédérique Segond. 1999. *A Grammar Writer's Cookbook*. Stanford, California: CSLI Publications.

Butt, Miriam, María-Eugenia Niño, and Frédérique Segond. 1996. Multilingual processing of auxiliaries in LFG. In D. Gibbon, ed., *Natural Language Processing and Speech Technology: Results of the 3rd KONVENS Conference*, pages 111–122. Bielefeld.

Cahill, Aoife, Mairead McCarthy, Josef van Genabith, and Andy Way. 2002. Parsing with PCFGs and automatic f-structure annotation. In M. Butt and T. H. King, eds., *Proceedings of the LFG02 Conference*. Stanford, California: CSLI On-line Publications. `http://csli-publications.stanford.edu/`.

Cahill, Aoife, Mairead McCarthy, Josef van Genabith, and Andy Way. 2003. Quasi-logical forms from f-structures for the penn treebank. In H. Bunt, I. van der Sluis, and R. Morante, eds., *Proceedings of the Fifth International Workshop on Computationsl Semantics, IWCS-5*, pages 55–71. January, Tilburg, The Netherlands.

Cardie, Claire and Raymond J. Mooney, eds. 1999. *Machine Learning: Special Issue on Natural Language Learning*, vol. 34. 1–3, February 1999.

Carpenter, Bob. 1998. *Type-Logical Semantics*. Cambridge, Massachusetts: The MIT Press.

Carroll, John, Guido Minnen, and Ted Briscoe. 1999. Corpus annotation for parser evaluation. In *Proceedings of the European Association for Computational Linguistics (EACL) Workshop on Linguistically Interpreted Corpora (LINC)*, pages 35–41. Bergen.

Chanod, Jean-Pierre and Pasi Tapanainen. 1996. A robust finite-state parser for french. In *Workshop on Robust Parsing, European Summer School in Language, Logic, and Information (ESSLLI)*, pages 12–16. Prague.

Charniak, Eugene. 1993. *Statistical Language Learning*. Cambridge, Massachusetts: The MIT Press.

Chomsky, Noam. 1957. *Syntactic Structures*. The Hague: Mouton de Gruyter.

Collins, Chris and Phil Branigan. 1997. Quotative inversion. *Natural Language and Linguistic Theory* 15:1–41.

Copestake, Ann. 2002. *Implementing Typed Feature Structure Grammars*. Stanford, California: CSLI Publications.

Copestake, Ann and Dan Flickinger. 2000. An open-source grammar development environment and broad-coverage english grammar using hpsg. In *Proceedings of the Second Conference on Language Resources and Evaluation (LREC00)*, pages 591–600.

Copestake, Ann, Dan Flickinger, Ivan Sag, and Carl Pollard. 1999. Minimal recursion semantics: An introduction. Unpublished manuscript, Stanford University, `http://www-csli.stanford.edu/~aac/papers`.

Copestake, Ann, Alex Lascarides, and Dan Flickinger. 2001. An algebra for semantic construction in constraint-based grammars. In *Proceedings of the 39th Annual Meeting of the Association for Computational Linguistics (ACL)*. Toulouse.

Crouch, Richard, Ronald M. Kaplan, Tracy Holloway King, and Stefan Riezler. 2002. A comparison of evaluation metrics for a broad coverage stochastic parser. In *Proceedings of the Workshop on Parseval and beyond and the 3rd International Conference on Language Resources and Evaluation (LREC02)*. Las Palmas, Spain.

Cutting, Doug, Julian Kupiec, Jan Pedersen, and Penelope Sibun. 1992. A practical part-of-speech tagger. In *Proceedings of the Third Conference on Applied Natural Language Processing*, pages 133–140. Trento.

Dalrymple, Mary, ed. 1999. *Semantics and Syntax in Lexical Functional Grammar: The Resource Logic Approach*. Cambridge, Massachusetts: The MIT Press.

Dalrymple, Mary. 2001. *Lexical Functional Grammar*. New York, New York: Academic Press.

Dalrymple, Mary and Ronald M. Kaplan. 2000. Feature indeterminacy and feature resolution. *Language* 76(4):759–798.

Dalrymple, Mary, John Lamping, Fernando Pereira, and Vijay Saraswat. 1999. Quantificaton, anaphora and intensionality. In *Semantics and Syntax in Lexical Functional Grammar: The Resource Logic Approach*, pages 39–90. Cambridge, Massachusetts: The MIT Press.

Dipper, Stefanie. 2003. *Implementing and Documenting Large-Scale Grammars: German LFG*. Ph.D. thesis, IMS Stuttgart. To Appear.

Eckle, Judith. 1997. *Entwicklung eines Simulationsmodells für Software-Projekte*. Berlin: Logos Verlag.

Eckle, Judith and Ulrich Heid. 1996. Extracting raw material for a German subcategorization lexicon from newspaper text. In *Proceedings of the 4th International Conference on Computational Lexicography (COMPLEX '96)*. Budapest.

Eckle-Kohler, Judith. 1998. Methods for quality assurance in semi-automatic lexicon acquisition from corpora. In *Proceedings of EURALEX '98*. Liège.

Egg, Markus. 1998. Wh-questions in underspecified minimal recursion semantics. *Journal of Semantics* 15:37–82.

Francis, W. Nelson. 1964. A standard sample of present-day english for use with digital computers. Tech. rep., Brown University, Providence, Rhode Island. Report to the U.S. Office of Education on Cooperative Research Project No. E-007.

Francis, W. Nelson and Henry Kučera. 1982. *Frequency Analysis of English Usage: Lexicon and Grammar*. Houghton Mifflin.

Frank, Anette. 2000. Automatic f-structure annotation of treebank trees. In M. Butt and T. H. King, eds., *Proceedings of the LFG00 Conference*. Stanford, California: CSLI On-Line Publications. http://csli-publications.stanford.edu/.

Frank, Anette, Tracy Holloway King, Jonas Kuhn, and John T. Maxwell III. 2001. Optimality theory style constraint ranking in large-scale lfg grammars. In P. Sells, ed., *Formal and Empirical Issues in Optimality Theoretic Syntax*, pages 367–397. Stanford, California: CSLI Publications.

Frank, Anette and Annie Zaenen. 2000. Tense in lfg: Syntax and morphology. In H. Kamp and U. Reyle, eds., *Tense and Aspect Now*. Tübingen: Niemeyer. To appear.

Johnson, Mark, Stuart Geman, Stephen Canon, Zhiyi Chi, and Stefan Riezler. 1999. Estimators for stochastic "unification-based" grammars. In *Proceedings of the 37th Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 535–541. College Park, Maryland.

Jurafsky, Daniel and James H. Martin. 2000. *Speech and Language Processing*. Upper Saddle River, New Jersey: Prentice Hall.

Kager, René. 1999. *Optimality Theory*. Cambridge: Cambridge University Press.

Kamp, Hans and Uwe Reyle. 1993. *From Discourse to Logic: An Introduction to Model-theoretic Semantics of Natural Language, Formal Logic and Discourse Representation*. Dordrecht: Kluwer Academic Publishers.

Kaplan, Ronald M. 1987. Three seductions of computational linguistics. In P. Whitelock, M. Wood, H. Somers, R. Johnson, and P. Bennett, eds., *Linguistic Theory and Computer Applications*, pages 149–188. London: Academic Press. Republished in *Formal Issues in Lexical-Functional Grammar*, Mary Dalrymple, Ronald M. Kaplan, John T. Maxwell III and Annie Zaenen (eds.) 1995. CSLI Publications.

Karttunen, Lauri, Jean-Pierre Chanod, Gregory Grefenstette, and Anne Schiller. 1996. Regular expressions for language engineering. *Natural Language Engineering* 2:305–338.

King, Tracy Holloway, Stefanie Dipper, Anette Frank, Jonas Kuhn, and John T. Maxwell III. 2001. Ambiguity management in grammar writing. *Journal of Language and Computation* In Press.

Kuhn, Jonas. 1998. Towards data-intensive testing of a broad-coverage grammar. In B. Schröder, W. Lenders, W. Hess, and T. Portele, eds., *Computers, Linguistics, and Phonetics between Language and Speech, Proceedings of the 4th Conference on Natural Language Processing*, pages 43–56. Bonn: Peter Lang.

Kuhn, Jonas. 2001. *Formal and Computational Aspects of Optimality-theoretic Syntax*. Ph.D. thesis, Universität Stuttgart. Revised version to be published by CSLI Publications.

Lascarides, Alex and Anne Copestake. 1999. Default unification in constraint-based frameworks. *Computational Linguistics* 25:55–105.

Maguire, Steve. 1993. *Writing Solid Code: Microsoft's Techniques for Developing Bug-free C Programs*. Redmond, Washington: Microsoft Press.

Manning, Christopher. 2000. Probabilistic head-driven parsing. Presented at Carnegie Mellon University; slides available, Stanford University.

Manning, Christopher and Hinrich Schütze. 1999. *Foundations of Statistical Natural Language Processing*. Cambridge, Massachusetts: The MIT Press.

Marcus, Mitchell P., Beatrice Santorini, and Mary Ann Marcinkiewicz. 1993. Building a large annotated corpus of english: The penn treebank. *Computational Linguistics* 19(3):313–330.

McConnell, Steve C. 1996. *Rapid Development: Taming Wild Software Schedules*. Redmond, Washington: Microsoft Press.

Morrill, Glyn. 1995. CATLOG: A compiler and parser for type-logical grammar. Dyana-2 Prototype P3.5.

Nakamura, Masami, Katsuteru Maruyama, Takeshi Kawabata, and Kiyohiro Shikano. 1990. Neural network approach to word category prediction for English texts. In H. Karlgren, ed., *Proceedings of the International Conference on Computational Linguistics*, pages 213–218. Helsinki University.

Nerbonne, John. 1998. *Linguistic Databases*. Stanford, California: CSLI Publications.

Neumann, Guenter. 2002. Data-driven approaches to head-driven phrase structure grammar. In R. Bod, R. Scha, and K. Sima'an, eds., *Introduction to Data-oriented Parsing*. Stanford, California: CSLI Publications.

Neumann, Guenter and Dan Flickinger. 2002. Hpsg-dop: Data-oriented parsing with hpsg. Unpublished manuscript, presented at HPSG-2002, Seoul.

Oepen, Stephan, Ezra Callahan, Dan Flickinger, and Christoper D. Manning. 2002a. LinGO Redwoods. A rich and dynamic treebank for HPSG. In *Beyond PARSEVAL. Workshop of the Third International Conference on Language Resources and Evaluation (LREC02) Conference*. Las Palmas, Spain.

Oepen, Stephan and Daniel Flickinger. 1998. Towards systematic grammar profiling: Testsuite technology ten years after. *Journal of Computer Speech and Language* 12:411–436. Special issue on evaluation.

Oepen, Stephan, Kristina Toutanova, Stuart Shieber, Chris Manning, Dan Flickinger, and Thorsten Brants. 2002b. The LinGO Redwoods treebank. Motivation and preliminary applications. In *Proceedings of International Conference on Computational Linguistics (COLING)*. Taipei, Taiwan.

Porter, M.F. 1980. An algorithm for suffix stripping. *Program* 14(3):127–130.

Retoré, Christian and Edward Stabler. 1999. Resource logics and minimalist grammars. Tech. Rep. 3780, Institute national de Recherche en Informatique et Automatique (INRIA).

Reyle, Uwe. 1988. Compositional semantics for lfg. In U. Reyle and C. Rohrer, eds., *Natural Language Parsing and Linguistic Theories*, pages 448–474. Dordrecht: Reidel.

Riezler, Stefan, Tracy Holloway King, Richard Crouch, John T. Maxwell III, Ronald M. Kaplan, and Mark Johnson. 2002. Parsing the wall street journal using a lexical-functional grammar and discriminative estimation techniques. In *Proceedings of 40th Annual Meeting of the Association for Computational Linguistics (ACL)*. University of Pennsylvania.

Sadler, Louisa, Josef van Genabith, and Andy Way. 2000. Automatic f-structure annotation from the ap treebank. In M. Butt and T. H. King, eds., *Proceedings of the LFG00 Conference*. Stanford, California: CSLI On-Line Publications. `http://csli-publications.stanford.edu/`.

Schabes, Yves, Patrick Paroubek, and the XTAG Research Group. 1997. XTAG user manual: An x window graphical interface tool for manipulation of tree-adjoining grammars. Department of Computer and Information Science, University of Pennsylvania; available on-line.

Schiller, Anne. 1996. Multilingual finite-state noun phrase extraction. In A. Kornai, ed., *Extended Finite State Models of Language*, pages 65–69. Proceedings of the ECAI 96 Workshop.

Schiller, Anne, Simone Teufel, Christine Stöckert, and Christine Thielen. 1999. Guidelines für das tagging deutscher textkorpora mit stts (kleines und großes tagset). Tech. rep., IMS, University of Stuttgart.

Schmid, Helmut. 1994. Part-of-speech tagging with neural networks. In *Proceedings of the 15th International Conference on Computational Linguistics (COLING)*, pages 172–176. August.

Schmid, Helmut. 1995. Improvements in part-of-speech tagging with an application to german. In *Proceedings of the Association of Computational Linguistics (ACL) SIGDAT-Workshop*. March.

Schmid, Helmut and Sabine Schulte im Walde. 2000. Robust german noun chunking with a probabilistic context-free grammar. In *Proceedings of the 18th International Conference on Computational Linguistics (COLING)*, pages 726–732. Saarbrücken, August.

Schulte im Walde, Sabine, Helmut Schmid, Mats Rooth, Stefan Riezler, and Detlef Prescher. 2001. Statistical grammar models and lexicon acquisition. In C. Rohrer, A. Rossdeutscher, and H. Kamp, eds., *Linguistic Form and its Computation*. Stanford, California: CSLI Publications.

Shieber, Stuart. 1986. *An Introduction to Unification-based Approaches to Grammar*. Stanford, California: CSLI Publications.

Spencer, Andrew and Louisa Sadler. 2001. Syntax as an exponent of morphological features. In G. Booij, ed., *Yearbook of Morphology*, pages 71–96. Dordrecht: Kluwer Academic Publishers.

Stabler, Edward P. 2001. Minimalist grammars and recognition. In C. Rohrer, A. Roßdeutscher, and H. Kamp, eds., *Linguistic Form and its Computation*, pages 327–352. Stanford, California: CSLI Publications.

Steedman, Mark. 2001. *The Syntactic Process*. Cambridge, Massachusetts: The MIT Press.

van Genabith, Josef and Richard Crouch. 1999a. Dynamic and undersepcified semantics for LFG. In M. Dalrymple, ed., *Semantics and Syntax in Lexical Functional Grammar: The Resource Logic Approach*, pages 209–260. Cambridge, Massachusetts: The MIT Press.

van Genabith, Josef and Richard Crouch. 1999b. How to glue a donkey to an f-structure: Porting a dynamic meaning representation language into LFG's Linear Logic Glue-language semantics. In H. Bunt and R. Muskens, eds., *Computing Meaning*, vol. I, pages 129–148. Dordrecht: Kluwer. Studies in Linguistics and Philosophy 73.

Xia, Fei, Martha Palmer, K. Vijay-Shanker, and Joseph Rosenzweig. 1998. Consistent grammar development using partial-tree descriptions for lexicalized tree-adjoining grammar. In *TAG+ 4 Workshop*. Philadelphia, Pennsylvania, August 1–3, 1998.