A PRINCIPLE-BASED SYSTEM FOR NATURAL LANGUAGE ANALYSIS AND TRANSLATION

by

Matthew Walter Crocker[1]

Technical Report 88-18

August 1988

[1] A thesis was submitted in partial fulfillment of the requirements for the degree of Master of Science.

Abstract

Traditional views of grammatical theory hold that languages are characterised by sets of constructions. This approach entails the enumeration of all possible constructions for each language being described. Current theories of transformational generative grammar have established an alternative position. Specifically, Chomsky's Government-Binding theory proposes a system of principles which are common to human language. Such a theory is referred to as a "Universal Grammar"(UG). Associated with the principles of grammar are parameters of variation which account for the diversity of human languages. The grammar for a particular language is known as a "Core Grammar", and is characterised by an appropriately parametrised instance of UG. Despite these advances in linguistic theory, construction-based approaches have remained the status quo within the field of natural language processing. This thesis investigates the possibility of developing a principle-based system which reflects the modular nature of the linguistic theory. That is, rather than stipulating the possible constructions of a language, a system is developed which uses the principles of grammar and language specific parameters to parse language. Specifically, a system is presented which performs syntactic analysis and translation for a subset of English and German. The cross-linguistic nature of the theory is reflected by the system which can be considered a procedural model of UG.

Contents

List of Figures

Acknowledgements

First and foremost I would like to express my gratitude to my supervisors: Dr. Michael Rochemont, for his invaluable assistance with the linguistic theory and guidance throughout the course of this thesis, and Dr. Harvey Abramson, for originally pointing me in the right direction and providing me with a background in logic programming.

I would especially like to thank Randy Sharp for his comments and suggestions during the final stages of the thesis, Brian Ross for his friendship and numerous helpful discussions, Barry Brachman for technical assistance, and the members of the Department of Computer Science who made the whole process enjoyable: Dave, Brent, Rick, and Heidi to name just a few. In addition, special thanks must go to my parents and Myron for their constant support and encouragement of my academic pursuits.

Finally, I would like to thank the Natural Sciences and Engineering Research Council for its support through two postgraduate scholarships, Dr. Abramson for research assistantships under an IBM SUR Grant, and the Department of Computer Science for additional funding.

Chapter 1

Introduction

Linguistic theory as it has developed within the transformational generative enterprise is fundamentally concerned with the nature of the human language faculty. The underlying hypothesis is that humans are innately endowed with some knowledge of language. Those principles of grammar which are innate are said to constitute a theory of *Universal Grammar* (UG). Such a theory could, for example, shed light on the process of child language acquisition. As Chomsky observes, some fundamental questions which arise in this pursuit are [Chomsky 86b]:

(1) (i) What constitutes knowledge of language?
(ii) How is knowledge of language acquired?
(iii) How is knowledge of language put to use?

The questions of (1i) and (1iii) raise the distinction between a theory of linguistic competence, represented by a particular grammar, and a theory of linguistic performance. Specifically, a theory of grammar must provide an account of the fundamental principles of UG, as well as those elements of grammar which must be learned. The latter must also be consistent with some psychologically plausible theory of language acquisition (i.e. the question of (1ii)). A theory of performance will indicate how knowledge of language is put to use in tasks of recognising and producing utterances.

Work within transformational generative grammar focuses primarily on the questions (1i&ii). The prevalent theory is Chomsky's *Government-Binding Theory* which posits a set of fundamental, language independent subtheories. Each subtheory consists of principles of grammar, which are taken to have parameters of variation which account for the diversity in phenomena across human languages. The set of principles constitutes an instance of UG, where the grammar for an individual language, or *Core Grammar*, is represented by a specific set of parameter settings.

It is important to note that the program of research sketched above represents a significant shift in focus from systems of *rules* to systems of *principles*. That is, while some linguistic theories, including early transformational grammar, attempt to characterise languages via descriptive sets of rules, the current approach is oriented toward the determination of a set of underlying principles

which can account for all languages. The advantages of such an approach are clear; in addition to accommodating theories of acquisition, the resulting theory will possess explanatory abilities which permit us to derive the properties of various languages instead of merely stipulating them.

Despite these advances in linguistic theory, rule-based approaches have remained the *status quo* in natural language processing systems. These systems are typically based upon some large context free grammar which is either used to compute parsing tables, or used directly in an Augmented Transition Network (ATN) or Logic Grammar. As a result, they inherit many of the problems of the rule-based grammar upon which they are founded. As Barton points out, rule systems are both unconstrained and stipulative in nature [Barton 84]. Not only are large numbers of language dependent rules required, but their correctness is difficult to ensure given the lack of underlying principles.

An alternative approach is to construct systems which employ the principles of grammar directly. This thesis is concerned with the implementation of a natural language analysis system which is based upon a subset of the principles of Government-Binding theory. We propose a particular procedural interpretation of the principles of GB theory, which are realised as a Prolog logic program[1]. In this way, the parser can be considered to model human linguistic competence. An effort is made to reflect the modularity present within the theory, and preserve its cross-linguistic capacity.

The parser attempts to maintain the distinction between the language independent principles of UG, and the specific rules and parameters relevant to an individual language. That is, a language independent parser is constructed which accesses the language specific information to parse a particular language. To illustrate this, a syntactic translation[2] system is developed, which permits bi-directional translation between English and German. The system consists of a principle-based parser which performs syntactic analysis, and a translator which translates lexical items and generates sentences in the target language. The basic approach was initially developed by Sharp, and later adopted by Dorr (see [Sharp 85], [Dorr 87]). Both describe principle-based parsing/translation systems for English and Spanish.

The basic goal of this research is to provide further insight into possible design strategies for principle-based systems. Emphasis is placed particularly on the design of an efficient system which retains the cross-linguistic capacity of the linguistic theory. The application to English and German brings to light certain language variations which are not clearly handled by the previous systems, and as such provides further evidence that a "universal parser" is indeed possible.

In Chapter 2 we present the Government-Binding theory as an instance of UG. Specifically, we motivate the theory with respect to language acquisition and the desire for an explanatory theory of grammar. We then present the model of grammar and each of the relevant subsystems

---

[1] The discussion of implementation in Chapters 4 and 5 assumes a knowledge of Prolog on the part of the reader. For an introduction to the language see [Clocksin *et al.* 81], [Hogger 84], and [Sterling *et al.* 86]. For a more theoretical discussion see [Lloyd 87].

[2] By syntactic translation we mean to imply that no semantics or pragmatics are involved. Rather, a simple system of lexical equivalence is assumed.

of principles and parameters. In Chapter 3, we describe the system of representations adopted for the lexicon and phrase structure. In addition, we present a language specific analysis of the transformations which are accounted for by the present system. Chapter 4 presents the implementation of the syntactic analysis component while Chapter 5 describes the translation component. Chapter 6 is devoted to a discussion and evaluation of the overall system, and examines related work in proposing possible improvements and extensions. Chapter 7 presents a general summary of the work and results, and suggests directions for future research. Finally, Appendix A illustrates the performance of the system with a number of example translations, while the remaining appendices contain the Prolog source code.

Chapter 2

Government-Binding Theory

Current efforts in transformational generative grammar (TGG) have led to the approach we know as *Government-Binding Theory* (henceforth, GB theory). The aim of the research has been to move away from systems of rules, favoring systems of principles. GB theory posits a set of independent subtheories of principles which are fundamental to all human languages. When combined, these subtheories yield a coherent theory of grammar, known as *Universal Grammar*. Each subsystem, or module, is intended to apply cross-linguistically, with some degree of parametrization of the principles for individual languages. Such a theory, with parameters instantiated appropriately, is said to constitute the *Core Grammar* of a specific language.

In this chapter we will first discuss language acquisition and explanation as motivating forces in theoretical linguistics. We then outline the overall model of grammar and present each of the modules of GB theory relevant to this work.

2.1 Acquisition and Explanation

The relevance of language acquisition to linguistic theory was recognised by transformational grammarians as early as *Aspects of the Theory of Syntax* [Chomsky 65]. This early work, however, was oriented more towards achieving descriptive adequacy for natural language grammars, than towards producing a grammatical theory. The more recent aim of Chomskian linguistics (see [Chomsky 73]) has been to provide a theory of grammar with certain explanatory abilities. Specifically, linguistic research has been guided more explicitly by the problem of language acquisition. This entails that grammatical theory account for the ability of children to master a rich and highly structured knowledge of grammar despite the fact that they are presented with data which is both degenerate and deficient. Hornstein and Lightfoot outline these deficiencies on three levels [Hornstein *et al.* 81]:

(2) (i) Children hear speech which does not consist uniformly of complete grammatical sentences, but also utterances with pauses, incomplete statements, slips of the

tongue, etc.

(ii) Despite being presented with finite data, children become able to deal with an infinite range of utterances.

(iii) People attain knowledge of the structure of language, despite the absence of such data. That is, people are able to make judgements concerning complex/rare sentences, ambiguity relations, and grammaticality using knowledge which is not available as primary linguistic data (PLD) to the child.

The problem of language acquisition then is that children acquire an extremely sophisticated knowledge of language despite it being underdetermined through the *poverty of stimuli* of (2). Furthermore, this occurs rather uniformly despite variation in intelligence and experience.

In attempting to account for this problem, the theory of grammar presupposes an *a priori* knowledge of language [1]. The assumption is that humans are endowed with a language faculty which consists of a set of "genetically encoded" principles [Lightfoot 82]. These principles are then activated appropriately during the acquisition process. A crucial observation at this point is that a child learns any language to which he is exposed. This entails that the innate principles be language independent. A fundamental goal of linguistics research is to determine the content of these principles such that they are abstract enough to account for all attainable languages, while still being rich enough to explain language acquisition under seemingly impoverished conditions. It is worth noting that these criteria of abstractness and richness are in conflict. However, this seems desirable as it provides rather strict guidelines for research.

The basic approach taken in the pursuit of such a theory has been to attribute parameters of variation to each of the principles. The value of a given parameter is drawn from a finite (presumably small) set of possible values. The theory of grammar defined by the set of principles is known as *Universal Grammar* (UG). When the parameters are appropriately instantiated for a specific language, UG is said to constitute a *core grammar* for that language.

The process of language acquisition can now be considered a parameter setting operation. The child begins at the initial state S_i with an uninstantiated set of principles. The parameters then become set via some *Language Acquisition Device* (LAD) which interprets the data presented to the child (for further discussion see [Chomsky 81b]). After sufficient information and experience, all the parameters are set and the child has a knowledge of the core grammar (i.e. the final state, S_f). If we take the data presented to the child to be random and unstructured, we can make a further simplification by suggesting that acquisition is effectively instantaneous [2]. This essentially assumes that the relevant data is presented to the child all at once. This hypothetical formulation of the acquisition process is known as the *logical problem of language acquisition.*

---

[1] This is opposed to inductive theories of learning which appear insufficient to account for the language acquisition problem given the poverty of stimuli assumption.

[2] While this may indeed be too strong a simplification, it should be noted that it does not conflict with theories which assume a more incremental learning process involving maturation of the child and language faculty. Rather, it abstracts away from these issues, so as to simplify the task at hand.

By adopting such a solution to the problem of language acquisition, we now have a metric for evaluating the explanatory adequacy of a theory of grammar. The criteria for explanation have been outlined as follows [Hornstein *et al.* 81]:

(3) (i) Coverage of empirical data: show that facts follow from the principles.
(ii) Simplicity and elegance of the principles.
(iii) The principles should contribute insight to the problem of language acquisition.

The requirement of (3iii) entails that the parameters of variation meet certain learnability criteria. Specifically, they must be determinable on the basis of positive evidence only. The motivation for this is that children by assumption are not exposed to the ungrammatical data, in any systematic way, which might be necessary to fix parameters (for further discussion see [Wexler *et al.* 80]).

2.2 A Model of Grammar

Transformational generative grammar is considered to have its origins in *Syntactic Structures* [Chomsky 57]. The original model proposed two levels of syntactic representation; *deep-structure* (or, D-structure) and *surface-structure* (or, S-structure) [3]. D-structures constituted a representation of the semantically relevant grammatical functions, and were considered to be generated by a set of phrase-structure rules. D-structures were then mapped to their "surface form" via a set of transformational operations. These transformational operations included rules for passivisation, wh-movement, subject raising, etc. and typically specified the *structural description* (SD) and *structural change* (SC). Consider for example the following:

(4) (a) The poem was written by Coleridge.
(b) Coleridge [TNS_{past}] write the poem.
(c)

	X	NP	AUX	V	NP	Y	by	Z
SD:	1	2	3	4	5	6	7	8
SC:	1	5	3+be	4+en	ϕ	6	7+2	8

If we take the sentence of (4a) to have the D-structure representation of (4b), where *the poem* is the object of *write*, and *Coleridge* is the subject, then we can see that a rule such as that of (4c) is perfectly adequate to describe the passive transformation of (4b) to (4a).

If we consider however the criteria and motivations for an explanatory theory of grammar as presented in Section 2.1, we see that the model presented above is inadequate. Not only are the rules language dependent in nature, but they also are highly specialised and rather vast in

[3] In fact, the notions of "deep" and "D" structure are not coextensive, and the same can be said for "surface" and "S" structure. Indeed, the abbreviated notation was adopted by Chomky explicitly to avoid confusion in the literature. For our purposes however, we may take them to be similar.

number. The former almost certainly eliminates the plausability of such rules being innate, and the latter presents a problem for learning under the poverty of stimuli assumption of (2).

The pursuit of an explanatory theory of grammar has led to the reduction of the rule component in favor of universal principles. The current model has a similar format to that of early TGG, that is the generation of D-structures which are mapped to S-structures via a transformational component. In addition, two interpretive components have been added; the LF (Logical Form) component, and the PF (Phonetic Form) component. Both are derived from S-structure, resulting in the model of grammar illustrated in Figure 2.1.

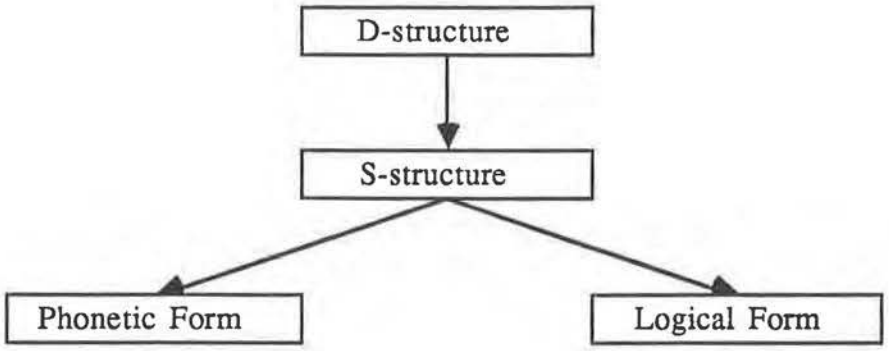

Figure 2.1: Model of Grammar

In the remainder of this section, we will discuss the system of rules and principles which are relevant to the system presented here. Specifically, we adopt the linguistic framework of Government-Binding theory as presented in [Chomsky 81a], and recent developments within that theory. A fairly detailed account of the evolution of generative grammar is presented in [vRiemsdijk *et al.* 86].

2.2.1 A System of Rules

The three basic parts of the rule system of GB are outlined in [Chomsky 82] as follows:

(5) (A) Lexicon
(B) Syntax: (i) Base component
(ii) Transformational component
(C) Interpretive: (i) PF component
(ii) LF component

The *Lexicon* constitutes the vocabulary of a language. That is, it consists of a set of entries for each lexical item, with information concerning its syntactic features, morphology, phonology,

and selectional requirements. The *Base Component*, combined with information projected from the lexicon, generates D-structures.

The *Transformational Component* maps a D-structure representation to a corresponding S-structure via application of the Move-α rule. This rule simply provides for the movement of elements from D-structure to S-structure positions. The resulting S-structure is then subject to the well-formedness conditions imposed by the principles of grammar. Viewed slightly differently, we can consider the principles to act as constraints on the application of Move-α. Additionally, the interpreted components map S-structures to their PF (phonetic form) and LF (logical form) representations respectively. As with the generation of S-structure, Move-α also plays a role in the generation of these representations. We restrict the discussion here however to the D-structure and S-structure levels of representation. The Base Component and Transformational Component together generate the structural representations, or *Syntax* of sentences.

2.2.2 A System of Principles

As we have seen, the rule systems are stated very generally. That is, subcategorization information projected from the lexicon, combined with $\overline{X}$-theory [4] and lexical insertion yields D-structures. The basic transformation operation Move-α then maps D-structures to S-structures. This simplicity in the systems of rules is made possible by the shift in emphasis to the system of principles.

The principles of GB theory interact so as to impose well-formedness conditions at the various syntactic levels of representation (i.e. D-structure, S-structure, and LF). These subsystems of principles, as outlined in [Chomsky 82] are:

(6) (a) $\overline{X}$-theory
(b) θ-theory
(c) Case theory
(d) Binding theory
(e) Bounding theory
(f) Control theory
(g) Government theory

As we have mentioned earlier, the generation of D-structures is determined by information projected from the lexicon in conjunction with some version of the $\overline{X}$-theory of phrase structure. Specifically, lexical properties may include certain selectional requirements. Consider for example that some prepositions, such as *for*, subcategorize for an NP object, while others, such as *away* do not. Additionally, verbs may subcategorize for a variety of phrasal constituents. In general, these subcategorized positions must be assigned a thematic role (henceforth, θ-role). It is also

[4] Additionally, $\overline{X}$-theory creates positions for adjuncts. Adjuncts may appear, regardless of subcategorization, for the purposes of modification and further description. We will return to the discussion of adjuncts in Section 2.3.

possible for a predicate VP to assign a θ-role to the subject position, even though that position is not subcategorized.

The generation of S-structures via the rule Move-α is constrained by the interaction of several principles. The *Projection Principle* requires that selectional properties of lexical items be represented at each syntactic level. This entails that moved constituents leave a trace behind in their D-structure positions. Bounding theory imposes locality conditions on the application of Move-α via the principle of *Subjacency*. Note however that under the right conditions elements may still be moved an arbitrary distance. This is achieved by successive "hopping" from one subjacent position to another. The abstract notion of *chains* is used to represent a constituent in terms of its "history of movement", in which the links of the chain record each application of Move-α. As we will see, chains provide an extremely convenient and elegant means of stating certain principles.

Case theory concerns the assignment of Case to noun phrases (NP's). The so-called *Case Filter* is instrumental in determining the distribution of NP's by requiring that each chain receive Case exactly once. Binding theory outlines possible coreference relations holding between NP's, while Control theory determines the reference of PRO, a phonetically null pronominal anaphor. Finally, the theory of Government defines the structural domain of lexical items. This notion plays an integral part in several of the principles.

The remainder of this chapter will examine each of the relevant principles and their interaction in some detail. The theories of Binding and Control have been omitted from the discussion as they are not present in the system developed here.

2.3 $\overline{X}$-Theory

$\overline{X}$-theory provides a generalised schema for the representation of phrase structure. The theory capitalises on the observation that all phrasal categories bear a certain structural resemblance. Specifically, $\overline{X}$-theory captures the endocentric nature of phrases with respect to a *(lexical) head*. The head of a phrase is taken to be that lexical item of which the phrase can be considered a "projection". Consider for example the following set of traditional phrase structure rules [5] for a subset of English:

(7) (a) S → NP Aux VP
(b) NP → Det N (PP)
(b) VP → V (NP) (PP)
(b) PP → P (NP)

A cursory inspection of (7b-d) makes apparent the systematic redundancy of such phrase structure rules. That is, NP is a projection of N, VP of V etc. for each lexical category. Fur-

[5] Using standard notation and terminology, we take those symbols on the left of the → to be non-terminal, phrasal nodes, and the rest, by default, to be terminal symbols (i.e. representing a class of lexical items). Symbols appearing in ()'s are considered optional.

thermore, the choice of constituents which may follow the head (known as the *complements*), is already specified in the lexicon as selectional requirements. These two observations make possible the following general rule:

(8) $\overline{X} \rightarrow X\ Complements$

Where X may be any lexical category[6], N, V, A or P. $\overline{X}$ represents a non-terminal node which inherits its grammatical properties from X. The choice of *Complements* is projected from the subcategorization information in the lexical entry for X.

The next level of phrase structure includes the rule for *specifiers*. The possible specifiers of a phrase are determined by the lexical properties of a given head. For instance, the possible specifiers for noun phrases are taken to be determiners (eg. articles *the,a* and quantifiers *each,all*) and possessive NP's, as in *the man's coat*. For adjective and prepositional phrases, the specifiers may be degree modifiers. The phrase structure rule for specifiers is the following:

(9) $\overline{\overline{X}} \rightarrow Specifier\ \overline{X}$

Finally, the theory must account in some way for *adjuncts* to phrases. Adjuncts are those phrases which are in no way lexically selected by the head, but rather appear optionally to further modify or describe the phrase in question. Typical examples are prepositional phrases (PP's) which can modify NP's, VP' or AP's as in the following:

(10)	(a)	the man on the hill	(= NP)
	(b)	watched with the telescope	(= VP)
	(c)	drunk at the party	(= AP)

Indeed, such PP adjuncts can also be a source of syntactic ambiguity as in the now famous example: *I saw the man on the hill with the telescope.* in which the two PP's can be interpreted as modifying either *saw* or *man* in a variety of ways. Additional adjuncts include adjectives and relative clauses for noun phrases and adverbs to verb phrases. The addition of appropriate rules for adjuncts yields the following set of rules for a theory of $\overline{X}$[7]:

(11)	(a)	$\overline{\overline{X}} \rightarrow Specifier\ \overline{X}$
	(b)	$\overline{X} \rightarrow Adjuncts\ \overline{X}$
	(c)	$\overline{X} \rightarrow \overline{X}\ Adjuncts$
	(d)	$\overline{X} \rightarrow X\ Complements$

[6] The lexical categories are not "arbitrary". Rather, they can be considered abbreviations for the more fundamental *substantive* [± N] and *predicative* [± V] features. The correspondence is as follows: N = [+N -V], V = [-N +V], A = [+N +V], and P = [-N -V].

[7] In the two-level $\overline{X}$ system, $\overline{\overline{X}} \equiv$ XP. That is, NP, VP, etc. refer to the maximal projections of their respective phrasal categories. We use both notations interchangeably throughout the thesis.

We have so far considered phrase structure only with respect to the so-called lexical categories. We have yet to provide an analysis of the the structure of S (that is, the rule of (7a)) in terms of $\overline{X}$-theory. Additionally, we must account for the structure of embedded and relative clauses. Earlier formulations of $\overline{X}$ phrase structure took these as exceptional constituents, and the following rules were stipulated for them:

(12) (a) $S \rightarrow NP\ Infl\ VP$
(b) $\overline{S} \rightarrow Comp\ S$

The rule of (12a) is virtually identical to that of (7a), with the exception that the *Aux* (auxilliary) element has been replaced by *Infl* (for, inflection). Infl is presumed to contain information about the tense of the clause, typically represented by the [±TNS] feature. Just in the case Infl is [+TNS], it also contains the agreement element (abbreviated AGR) consisting of the usual person, number, and gender attributes. The second rule, (12b), states that embedded or relative clauses consist of a (possibly optional) element *Comp*, followed by an S. Comp is a complementizer element such as *that* or *for*. The following example uses the traditional *labeled bracketing* representation to illustrate these constructions:

(13) [$_S$ I think [$_{\overline{S}}$ that [$_S$ the man bought [$_{NP}$ the book [$_{\overline{S}}$ that [$_S$ I wanted]]]]]]

A more recent analysis however has been to consider Infl and Comp as *non-lexical* categories, which are subject to the rules of $\overline{X}$-theory in the same way as lexical categories [Chomsky 86a]. That is, take Infl (= I) to be the head of IP (=S) and Comp (= C) to be the head of CP (=$\overline{S}$). Under this analysis, it seems natural to let the subject NP be a specifier to IP, and assume that VP is a complement which is inherently selected for by I. Additionally, C selects for an IP complement. The specifier position of CP, as we shall see in Chapter 3, is useful as a landing site for moved constituents (such as wh-phrases), but is typically not filled at D-structure. Under this analysis the rules of (12) are replaced by the following instantiations of the $\overline{X}$-theory:

(14) (a) $\overline{\overline{C}} \rightarrow Wh-phrase\ \overline{C}$
(b) $\overline{C} \rightarrow C\ \overline{\overline{I}}$
(c) $\overline{\overline{I}} \rightarrow \overline{\overline{N}}\ \overline{I}$
(d) $\overline{I} \rightarrow I\ \overline{\overline{V}}$

The $\overline{X}$ rules outlined in (11) and (14) are indeed sufficient for describing the phrase structure of languages such as English. Problems arise however when attempting to account for other languages. Specifically, these problems concern the order, or *precedence* relations. That is, while specifiers are *initial* in many Indo-European languages, this generalisation does not hold for many other languages [8]. In addition, languages vary with respect to the position of heads

[8] See [Lightfoot 82] for a discussion of constituent order for a variety of languages.

relative to their complements. For English, the heads of all categories are initial with respect to their complements. In German however, both V and I are considered to be final [Thiersch 78]. In general then, the dominance relationships of the rule in (11) hold, but the precedence relationships must be taken to vary parametrically for individual languages.

It is possible then to express $\overline{X}$-theory in even more general terms. The rules of (15a-b) represent similar dominance relations, but permit variations in precedence to be determined by parameters (or, as we shall see, other principles) [9]. If we let the superscripts represent the bar-level used above, then we take *arguments* (i.e. subcategorized constituents) to be *sister-to* X^i, $i < 2$, and *adjuncts* are sister-to X^2 [10].

(15) (a) $X^i \rightarrow Y, X^j$
(b) $X^i \rightarrow X^j, Y$
(c) where: $i \leq 2, j \leq i, j \geq 0$.

We may take Y to be either an $\overline{\overline{X}}$ or a simple lexical specifier.

2.4 θ-Theory and Lexical Selection

We remarked above that lexical items may have certain selectional requirements. Specifically, they may select, or subcategorize, for certain phrasal complements. Additionally, lexical items may require the presence of constituents considered necessary thematic participants of the sentence. In such cases, the required constituent is assigned a θ-role by the lexical item which selects for it. Subcategorization properties of a given head will require that constituents of a specific category appear adjacent[11] to it, and dominated by some projection of the head. This is primarily a syntactic requirement which determines the *complements* of the head for the purposes of $\overline{X}$-theory.

θ-theory concerns the thematic relations which are assigned by a lexical item. This is related to subcategorization, in that subcategorized constituents are generally assigned θ-roles (we will strengthen this notion below). Furthermore, a lexical item may assign an *external* θ-role to its Subject[12]. Consider for example the sentence *John put the book on the refrigerator*. Here, the head of VP, *put*, subcategorizes for NP and PP complements. Additionally, the verb assigns

[9] This version of $\overline{X}$-theory is adapted from Chomsky's class notes, 1986.

[10] This notion of $\overline{X}$ phrase structure differs with that of (11) in two respects: firstly, adjuncts are now sister to X^2 instead of X^1, and secondly, the Specifier position is now a "special instance" of an adjunct. This is somewhat inconsistent with the literature, but is convenient for our purposes and nothing critical hinges on it.

[11] In fact, the adjacency requirement may be taken to vary parametrically, as suggested by languages such as Japanese.

[12] In fact, the external θ-role is taken to be assigned by the VP, as determined compositionally by the verb and its complements [Marantz 81].

three θ-roles; *Agent* is assigned to the subject, *Theme* to the direct object, and *Location* to the prepositional phrase.

The assignment of a θ-role to a constituent is known as *θ-marking*. More precisely, a constituent receives a θ-role by virtue of occupying a position which is θ-marked (henceforth, θ-position). It is a fundamental requirement that all θ-roles be assigned at D-structure, which is considered to constitute a "pure" representation of the thematic interpretation. The proper assignment of θ-roles is determined by the *θ-Criterion*, stated as follows:

(16) **θ-Criterion:** each argument bears one and only one θ-role, and each θ-role is assigned to one and only one argument.

Subcategorized elements are considered to be *directly* θ-marked. In addition, we observed above that the subject position may be *indirectly* θ-marked. This *indirect* θ-marking is generally considered to be mediated through I. If we further assume that the θ-criterion applies at each syntactic level of representation, then movement may never be to a θ-position, since this would result in a θ-role being assigned to two arguments (i.e. the D-structure argument which must have vacated the position, and the element which is moved to the position).

The *Projection Principle* in effect generalizes the notion of selectional requirements across each level of syntactic representation. Specifically, it is stated as follows[13]:

(17) **The Projection Principle:**
(i) Subcategorizable positions are θ-marked by the governing head, at each syntactic level of representation (i.e. D-structure, S-structure, and LF).
(ii) θ-marked positions must be represented categorially at each syntactic level of representation.

The Projection Principle is fundamental to GB theory, as it constrains the mapping between each level of syntactic representation. As a result of imposing this general well-formedness condition, the Projection Principle has as one of its consequences the theory of empty categories. That is, when an element is moved from its D-structure position, that position must remain in some sense even though it is not filled by any lexical constituent. To account for this, a phrasal constituent of the same category as the moved element remains in the vacated position, dominating a *trace* which is *co-indexed* with the moved element. Note that movement or *raising* to object position is now prohibited by the Projection Principle and θ-criterion, since the raised constituent would receive two θ-roles[14].

While the Projection Principle stipulates that direct θ-marking is entailed by subcategorization, indirect θ-marking is clearly not. The assignment of a θ-role to subject is in no predetermined sense obligatory but rather is determined purely by the θ-marking properties of the

[13] This statement of the Projection Principle differs slightly from the formulation in [Chomsky 81a], but is essentially the same in both spirit and function.

[14] There have been recent claims that raising to object is indeed possible, and that the θ-criterion should be restated. For discussion of this see [Pullum *et al.* 88] and [Massam 85].

head. So, while raising to object is not possible, raising to subject sometimes is, as the following example illustrates:

(18) (a) It seems $[_{CP}$ that John likes Mary].
(b) John$_i$ seems $[_{IP}\ e_i$ to like Mary].

While *seems* does select for a propositional complement, it does not θ-mark the subject position. In (18a) the non-referential, *pleonastic* element "it" is inserted to fill the position. This permits (18b) to be generated, where *John* moves from the subject position of the embedded clause (where it receives its θ-role), to the non-θ-marked subject position in the matrix clause (known as a $\overline{\theta}$*-position*).

The *Extended Projection Principle* introduces the additional stipulation that every clause must have a subject. This is necessary just in the case where the verb does not select for a subject, as was observed in (18). That is, the sentence *Seems that John likes Mary* is ruled ungrammatical by the Extended Projection Principle.

2.5 Movement

The shift from rules to principles has permitted the generalisation, indeed trivialisation of the transformational component. While past approaches posited numerous transformational rules, such as that for passive illustrated in (4), the more recent tact has been to propose the singular operation *Move-*α. Move-α essentially says *move anything, anywhere*. The possibility of over or incorrect application of this operation is ruled out by the principles of GB. In short, these principles dictate when movement is possible, necessary, or prohibited.

We have introduced the notion of *chains* as a representation of arguments in terms of their "history of movement". More precisely, a chain consists of a *tail* (its D-structure position) and a *head* (its S-structure position). By virtue of the θ-criterion, each chain is assigned exactly one θ-role, and this must be to the tail of the chain. The chain may also contain intermediate positions, represented by traces, through which the argument moved on the way to its destination, S-structure position.

Despite the general nature of the Move-α operation, the types of movement possible are rather severely constrained by the principles. As Chomsky observes, there are essentially two types of movement: *substitution* and *adjunction*. We are concerned here primarily with substitution which has the following general properties [Chomsky 86a]:

(19) (a) There is no movement to a complement position.
(b) Only X^0 can move to a head position.
(c) Only $\overline{\overline{X}}$ can move to a specifier position.
(d) Only X^0 and $\overline{\overline{X}}$ are visible to Move-α.

The θ-criterion clearly accounts for (19a), while (19b-c) would appear to follow from the *Structure Preserving Hypothesis* [Emonds 76]. The constraint of (19d) is simply stipulated, but seems intuitively desirable.

In English and German, the *head-to-head* movement of (19b) typically involves the categories V and I. As we shall see in Chapter 3, this type of movement is responsible for *have-be* raising and *Subject-Aux Inversion* (SAI) in English, as well as for the so-called *verb-second* phenomena in German.

We have observed that movement must always be to a $\overline{\theta}$-position. In addition, we can make the general distinction between argument positions (A-positions) which *may* receive a θ-role, and non-argument positions ($\overline{A}$-positions) which may not. The subject is therefore always an A-position, but not necessarily a θ-position. In general, any individual application of Move-α may be from an A-position to an A-position (henceforth, A-movement) or from an A or $\overline{A}$ position to an $\overline{A}$-position ($\overline{A}$-movement). A-movement is to a subject position ($SPEC, IP$) and is generally determined by Case theory, as we shall see in Section 2.7. An instance of $\overline{A}$-movement would be movement of a wh-phrase to the $SPEC, CP$ position.

2.6 Government

The phrase structure of $\overline{X}$-theory permits us to describe certain structural relationships between constituents. Central to GB theory is the notion that many of its principles prescribe certain *locality* constraints on the relations between items in the tree. That is, a given principle defines a *domain* which is local with respect to the constituent that principle concerns.

The most prominent, and indeed fundamental, of these structural domains is that of *government*. Government is itself defined in terms of the less restrictive notion of *m-command*. We take these to be defined as follows[15]:

(20) **m-command:** α *m-commands* β iff every maximal projection dominating α dominates β.

We may now define government as follows:

(21) **government:** α *governs* β iff
(i) α m-commands β, and
(ii) α is a head (i.e. X^0), and
(iii) β m-commands α

The theory of Government plays a central role in determining the distribution of empty categories. Specifically there are two types of empty categories: *traces*, which occupy the positions which have been vacated by movement, and *PRO*, which is a phonetically null pronominal

[15] These definitions have been adapted from a variety of sources in the current literature.

anaphor. The basic well-formedness condition for empty categories is known as the *Empty Category Principle* (ECP), which states that all traces must be *properly governed.* This might be extended, to include the observation that PRO must be ungoverned. This *Extended ECP* [Chomsky 82] can be stated simply as follows:

(22) **Extended ECP:** If α is an empty category, then
(i) α is trace iff it is *properly governed*
(ii) α is PRO iff it is ungoverned.

where,

(23) **Proper Government:** α *properly governs* β iff
α governs β, (α a lexical X^0) or
α locally $\overline{A}$-binds β

Where the notion of *locally* $\overline{A}$*-bound*, entails that the empty category (henceforth, *ec*) be bound by an $\overline{A}$ position which is "not too far away".

In general then, if an empty category is governed, it is a *trace*, and if not it is *PRO*. We may also assume that *PRO* may only appear in subject position, since the object position is necessarily governed by its subcategorizing head. To see how the ECP applies, consider the following sentences:

(24) (a) Why$_i$ does Barry want [PRO to be a rocket scientist] t_i ?
(b) Who$_i$ does Barry want [t'_i [t_i to be a rocket scientist]] ?

In (24a) we see that the embedded subject *ec* is ungoverned, and therefore must be PRO. In (24b) however, we see that the subject *ec* is locally bound by *t'*, which is in the SPEC,CP position (an $\overline{A}$-position). The subject *ec* of (24b) is therefore properly governed, and must be a trace.

2.7 Case Theory

Case theory concerns the assignment of Case to noun phrases, and plays an important role in determining their distribution. While Case may be realized morphologically, NP's are considered to receive Case, regardless of their morphological status. In English, morphological case is rather impoverished, generally distinguishing only the nominative, accusative and genitive form of pronouns such as *he/him/his*, *we/us/our* and the wh-pronouns *who/whom/ whose* (although this distinction between *who* and *whom* is disappearing). German, on the other hand, maintains a fairly rich case system, in which the nominative, accusative, dative, and genitive forms of most NP's are distinguished (although not always uniquely). Case theory, however, is not concerned with such phenomena, which appear to vary widely among languages, but rather with the more general assignment of *abstract* Case.

The fundamental requirement of Case theory is that every NP receive (abstract) Case. An NP receives Case by occupying a position to which Case is assigned. Specifically, a position is assigned Case if it is governed by a *Case-assigner*. For English the Case assigning categories are generally taken to be V, P, and I. The ability for V and P to assign Case must additionally be specified in the lexicon. If a lexical item can assign Case, it is said to be *transitive*, otherwise *intransitive*. In addition, I is a Case-assigner just in case it has the feature [+TNS][16]. The fundamental principle of Case theory, the Case Filter, is stated as follows:

(25) **Case Filter:** *NP, where NP is phonetically realized and has no Case.

Chomsky has suggested that this requirement can be re-formulated in terms of chains (see [Chomsky 81a], Chapter6). That is, Case theory requires that every chain be assigned Case exactly once. Consider for example the sentences below:

(26) (a) * It appears $[_{IP}$ John to like Mary].
(b) John$_i$ appears $[_{IP}$ e_i to like Mary].
(c) It appears $[_{CP}$ that John likes Mary].
(d) * John appears $[_{CP}$ (that) e_i likes Mary].

The Case Filter rules (26a) as ungrammatical, since John is not assigned Case (I is untensed). This can however be rendered grammatical by moving *John* to the subject position of the tensed matrix clause, as in (26b). This results in the formation of the chain (John, e_i). This is grammatical since the chain receives a θ-role at e_i, and Case is assigned to *John*. If the embedded clause is tensed, as in (26c), then its subject need not move. Indeed, movement of the matrix subject is prohibited since the resulting chain would be assigned Case twice, as shown in (26d)[17].

There are however instances when the subject of an untensed clause does not need to move. Consider,

(27) (a) I believe $[_{IP}$ John to like Mary].
(b) I believe $[_{CP}$ that John likes Mary].

As we would expect, (27b) is grammatical much as (26c) is. How then can we explain the grammaticality of (27a)? The solution adopted here is to assume that government may take place across the maximal projection IP (=S). The recent position on this is to assume that if $\overline{\overline{X}}$ is governed, so is its specifier, $SPEC, XP$ and its head, X^0 [Chomsky 86a]. Therefore, since *believe* governs the IP , the subject, $SPEC, IP$, is governed and thus can be Case-marked. This permits the matrix verb to Case-mark the subject of the embedded clause, rendering (27a) grammatical. The difference then between the verbs *believe* and *appear*, is that *believe* is transitive, permitting

[16] Another point of view is to consider AGR a Case-assigner, not I[+TNS], since AGR is present (or, *rich*) only for I[+TNS].

[17] For other accounts of the ungrammaticality of (26d), see [Chomsky 81a]

it to assign Case, while *appear* is not. This process is generally referred to as *Exceptional Case Marking* (ECM)[18].

As a final example, consider the following set of sentences:

(28) (a) I want [$_{IP}$ John to like Mary].
(b) I want very much [$_{CP}$ for John to like Mary].
(c) * I want [$_{CP}$ that John like(s) Mary].

At first glance, (28a) would suggest that *want* behaves similarly to *believe*. Examination of (28b) however has led to the analysis that *want* selects for CP with a *for* complementizer. The *for* acts as a preposition, able to assign Case to the embedded subject. While *for* may/must delete when adjacent to the verb, it must be present when there are intervening elements (behaving similarly to the *that* complementizer). An additional observation is that sentences with *believe* can be passivised, while those with *want* cannot.

It has been proposed that Case theory may also play a significant role in determining the so-called *free* and *fixed* word order properties of languages. The *Case Adjacency Principle* (CAP) [Stowell 81] requires that NP's be "string adjacent" to their Case-assigners. The invocation of this principle is taken to be determined by a parameter of variation for individual languages. Consider the following English and German examples:

(29) (a) The boy will put the book on the table.
(b) * The boy will put on the table the book.
(c) Der Junge wird das Buch auf den Tisch legen.
(d) Der Junge wird auf den Tisch das Buch legen.

We may explain the ungrammaticality of (29b), by assuming that the CAP is in effect for English. Additionally, we may take the relative free word (or constituent) order of German to indicate that the principle does not apply in this language. We will not pursue this matter here, but simply note it as a principled solution to determining such properties of language.

2.8 Bounding Theory

The previous sections have outlined how principles constrain possible landing sites for movement. These constraints are imposed primarily by the θ-criterion, the Projection Principle, and Case theory. Bounding theory constrains movement directly by prohibiting a constituent from being moved "too far" in a single hop. Additionally, the theory imposes certain *island* constraints, where islands are taken to be domains out of which no constituent can be moved.

An early approach was to specify these constraints individually. Consider for example the following island conditions proposed in [Ross 67]:

[18] Previous analysis suggested that *believe*-type verbs could somehow delete the $\overline{S}$ node, leaving only an S, which was a non-barrier to government [Chomsky 81a].

(30) **Complex NP Constraint (CNPC):** No element in an S dominated by an NP, may be extracted from that NP:

* Who$_i$ do [$_S$ you like [$_{NP}$ the book that John gave to t$_i$]]

(31) **Wh-Island Constraint (WhIC):** No element contained in an indirect question, $\overline{S}$, may be moved out of that $\overline{S}$:

* What$_j$ do [$_S$ you wonder [$_{\overline{S}}$ who$_i$ [$_S$ t$_i$ bought t$_j$]]]

(32) **Sentential Subject Condition (SSC):** No element may be extracted from an S, if that S is a (sentential) subject:

* Who$_i$ did [$_S$ [$_{NP}$ [$_S$ that she dated t$_i$]] bother you]

We can summarize these constraints by requiring that ϕ not be related [19] to ψ in the following contexts:

(33) CNPC: ... ϕ ... [$_S$ ... [$_{NP}$ ... ψ ...
WhIC: ... ϕ ... [$_S$ ... [$_S$ ... ψ ...
SSC: ... ϕ ... [$_S$ ... [$_{NP}$ ... [$_S$ ... ψ ...

In an attempt to capture these constraints in a principled fashion, Chomsky introduced the Subjacency Condition [Chomsky 73], a formulation of which is stated below:

(34) **Subjacency:** A singular application of Move-α may not cross more than one *bounding node*. That is, ϕ may not be related to ψ in the following context:

... ϕ ... [$_\alpha$ ... [$_\beta$ ... ψ ...
where α and β are bounding nodes.

Additionally, it is necessary to impose the condition of *Strict Cyclicity* on the operation of Move-α [Chomsky 73]. This condition may be stated as follows:

(35) **Strict Cycle Condition:** Once Move-α applies across a cyclic node β, no future application of Move-α may be applied so as to solely affect a subdomain of β.

This condition essentially requires that Move-α be applied in a strictly "bottom-up" manner. While the notion of *cyclic node* has been defined in a variety of ways, we take it to be any maximal projection, following [Williams 74].

Under this notion of Subjacency, the bounding nodes for English were generally taken to be NP and S. As we can see, these choices for bounding nodes account for all the island constraints in (33). The choice of bounding nodes was however shown to be subject to parametric variation

[19] We take "related" to include the antecedent-trace relationship of a moved constituent and its trace.

across languages, such as Italian where evidence suggested the bounding nodes were NP and $\overline{S}$ [Rizzi 82].

The Subjacency condition is not sufficient however, to account for all possible island effects. Consider for example the following sentences:

(36) (a) Who$_i$ did you read [a book about t_i] ?
(b) * What$_i$ did you read [a book under t_i] ?

While (36a) is perfectly grammatical, (36b) is not, with the intended reading: *What was the book that you read resting under ?*. To account for this phenomena, Huang proposed the *Condition on Extraction Domain* [Huang 82], which can be stated as follows:

(37) **Condition on Extraction Domain (CED):** A phrase α may be extracted out of a domain β only if β is properly governed.

The formulation of proper government adopted was assumed to exclude adjuncts. If we take *book* to optionally select for an "about" PP, then extraction out of the PP is possible, as in (36a) (since the PP will be properly governed). Extraction of the locative PP adjunct in (36a) is ruled ungrammatical by the CED principle, since the PP is not properly governed[20].

In an attempt to account for both the CED and Subjacency with one principle, Chomsky has proposed the concept of *barriers* [Chomsky 86a]. The work has been an attempt to recast the principles of government and bounding in terms of this more general notion. We restrict our discussion here to the account of Subjacency developed by Lasnik and Saito, which is a revision of that suggested by Chomsky. We take the definition of barrier to be as follows [Lasnik *et al.* 88a]:

(38) **barrier:** α is a *barrier* for β iff:
(i) α is a maximal projection,
(ii) α is not *L-marked*, and
(iii) α dominates β.

where,

(39) **L-marking:** α is *L-marked* by β iff β is directly θ-marked by α.

In addition, we follow Lasnik and Saito in assuming that VP can be L-marked by I, but that IP cannot be L-marked by C. With these notions defined, we can now restate Subjacency as follows:

(40) **Subjacency:** β is *Subjacent* to α if for every γ a barrier for β, the maximal projection immediately dominating γ dominates α.

---

[20] Note, this requires that the government domain be determined by the first (i.e. lowest) X^2 node. In the present system however, we assume m-command to be determined by the "collective" set of X^2 nodes (i.e. the highest node of the phrase).

To see how Subjacency now applies consider the following sentences taken from the examples above[21]:

(41) (a) * Who$_i$ do [$_\gamma$ you like the book [$_\gamma$ OP$_j$ that [$_\gamma$ John gave t$_i$ t$_j$]]] ?
(b) * What$_i$ do [$_\gamma$ you wonder who$_j$ [$_\gamma$ t$_j$ bought t$_i$]] ?
(c) * Who$_i$ did [$_\gamma$ [$_\gamma$ that she dated t$_i$] bother you] ?
(d) Who$_i$ did [$_\gamma$ you read a book about t$_i$]] ?
(e) * What$_i$ did [$_\gamma$ you read a book [$_\gamma$ under t$_i$]] ?

The γ symbol has been used to indicate those maximal projections which are barriers. That is, those phrases which are not L-marked. In each case, our new formulation of Subjacency accounts for the phenomena of each of the other approaches outlined above. Notice that in (41d), *book* is considered to subcategorize (optionally) for an *about* PP. The PP is therefore L-marked, and not a barrier. In (41e) however, the *under* PP is simply an adjunct, and therefore not L-marked, making a barrier. In this case, the antecedent for t appears outside the first $\overline{\overline{X}}$ dominating γ, thus violating the Subjacency condition.

[21] We follow linguistic convention in using *OP* to represent an *empty operator* – a phonetically null wh-pronoun.

Chapter 3

Representations and Analysis

In this chapter we will outline the system of representations adopted for the lexical and syntactic components of the system. The lexicon specifies the properties of words known to the system. The section discussing syntax will outline how the $\overline{X}$ phrase structures are represented, and we also present the syntactic analysis of English and German which has been adopted in this work.

3.1 The Lexicon

The lexicon constitutes the vocabulary of the system. It is comprised primarily of two components: 1) a database of lexical entries, known as the dictionary, and 2) morphological information for constructing the *regular* inflected forms of the dictionary entries. A lexical entry can contain basically three types of information:

(42) *Morphological:* Agreement features such as person, number, gender, and the tense and participle forms for verbs.

Syntactic: Primarily subcategorization information, properties of transitivity, and any other features affecting syntax.

Semantic: We restrict this to θ-marking properties of lexical items.

For our purposes, the syntactic and semantic information is effectively merged. That is, subcategorization information is considered to indicate the direct θ-marking properties of lexical items, and indirect θ-marking properties (i.e. of the subject) are specified by a simple binary feature. No attempt is made here to actually assign thematic roles to constituents, rather we observe only the syntactic requirements that this would impose.

3.1.1 The Dictionary

The format of the dictionary is similar to that of [Sharp 85], where entries are represented as Prolog *facts*, or unit clauses, of the following form:

(43) dict(Language, Category, Word, LexicalInfo).

Language simply specifies the language of the entry, and for our purposes may be instantiated as follows:

(44) **Language =** eng English
ger German

Category indicates the lexical category for the entry. Additionally, a category may be parametrized, to permit subcategorization of a specific form or feature. The parameters may also assist in specifying and restricting possible phrase structure configurations. *Category* may take on the following values:

(45) **Category =**

n(Case)	Noun, with a Case parameter (for German) Case = "nom" (nominative), "acc" (accusative), "dat" (dative).
v(TNS/Form)	Verb, with TNS (tense) and Form (participle): TNS = "tns(+)" (tensed) or "tns(-)" (untensed) Form = "part(pres)" (present participle) or "part(past)" (past participle).
p(Type)	Preposition, where Type = "loc" (locative), "dir" (directional), "tem" (temporal), "des" (descriptive).
i(TNS)	Infl, with TNS parameter, similar to verbs.
c(Lev/Type)	Complementizer, where Lev = "mat" (root clause), or "emb" (embedded clause), and Type = "sent" (sentential complement) or "rel" (relative clause).
d	Determiners, specifiers to noun phrases.

Word is the lexical item itself.

LexicalInfo is actually a *list* structure, containing all the morphological and syntactic information which is specific to the lexical entry. The list may contain the follow information structures:

(46)	**LexicalInfo =**	ftr(FtrList)	Morphological and inflectional features, contained in FtrList.
		subcat(Frame)	Subcategorization frame, where Frame is a list of subcategorized categories.
		theta(±)	Indicates whether the verb θ-marks the subject, theta(+) (yes), or theta(-) (no).
		irr(List)	A list of *irregular* forms for the entry.
		root(Word)	The root, if the entry is irregular.
		english(Eng)	Specifies the English lexical equivalent.
		german(Ger)	Specifies the German lexical equivalent.
		pl(Plural)	Specifies an irregular plural form.
		aux(±)	Marks an auxilliary verb, aux(+) (auxilliary), aux(-) (main).

The set of features contained in the *ftr* construct are drawn from the following:

(47)	**FtrList =**	per(Per)	Person, per(1), per(2), per(3).
		num(±)	Number, num(+) (plural), num(-) (singular).
		gen(Gen)	Gender, gen(m) (masc.) gen(f) (fem.) or gen(n) (neut.).
		wh(±)	Wh-item, wh(+) (e.g. who) or wh(-).
		case(Case)	Case, case(X), X = nom, acc, or dat.
		proper(±)	Indicate if a noun is proper, proper(+) (proper), proper(-) (common).
		num(N,G,C)	Represent the triple of number, gender, and Case, used for determiner forms (German only).

To reduce the size of entries, and the redundancy of information, default feature sets can be specified for the individual categories of a language. When the entry for a word is retrieved, the default *FtrList* of the category is consulted for any information not present in the entry itself. Defaults may be specified as follows:

(48) defaults(Language, Category, DefFtrList).

The defaults for the present system appear at the beginning of the lexicon for each language (Cf. Appendix G).

3.1.2 The Morphology

Any given lexical item may have a variety of "surface" forms. That is, nouns have a plural form, determiners may vary according to Case, number, and gender (for German, specifically), and verbs may be tensed or appear in participle forms. Often, these forms can be constructed from

the root word, by applying specific morphological rules[1]. An obvious example is the addition of "s" for the plural form of a noun, or the addition of "ed" for the simple past of a verb. If the various forms of a lexical entry can be constructed through the application of some predetermined operations for that language, the item is said to be *regular*. If these rules fail to apply to a given item, then it is *irregular*. Consider the plural form of "woman", which is not "womans", as our pluralisation rule would suggest, but rather "women".

While the present system performs only limited morphological analysis, it is capable of analyzing word suffixes. This permits the system to construct dictionary entries for the inflected forms or regular words, thus greatly reducing the size of the lexicon. The suffix rules are specified in the lexicon for each language, and have the following form:

(49) suff_table(Language, Category, InflEnd, RootEnd, Features).

where,

(50)	Language =	eng/ger	The language for which the rule applies.
	Category =	n/v/i	The category for which the rule applies.
	InflEnd =	'...'	The ending of the inflected form.
	RootEnd =	'...'	The ending of the root form.
	Features =	FtrList	The features associated with the inflected form.

The dictionary now consists of primarily two types of entries: 1) the root entries, containing the syntactic features, and 2) entries for the irregular forms, which contain their inflectional features, and a pointer to the root entry.

In fact, German presents a problem for the rather simple morphological analysis performed by this system. Specifically, regular lexical items may not just vary the suffix, but also the prefix, and possibly inner vowels. Although there may be formal schemes for deriving these variations, they are too complex for this system and are treated as irregular occurrences.

3.2 Phrase Structure

We adopt here the version of $\overline{X}$-theory presented at the end of Section 2.3. The rules are repeated here for convenience:

(51) (a) $X^i \rightarrow Y, X^j$
(b) $X^i \rightarrow X^j, Y$
(c) where: $i \leq 2, j \leq i, j \geq 0$.

[1] We are concerned here with generative morphology. Specifically, we account for the orthographic conventions for representing variations in the morphology.

We assume the *Specifier* position for all phrases appears as the first (highest and leftmost) *pre-adjunct*, and that all adjuncts are sister to an X^2 node. The complements are those arguments which are subcategorized by the head, and appear as sisters to the X^0 node or its X^1 projections. The result is a strictly binary branching representation of $\overline{X}$ phrase structure.

To represent this structure, we introduce the "/" operator to represent *dominance*, and use Prolog's list notation "[...]", to indicate *sisterhood* and *precedence*. Since the phrase structure trees are strictly binary, a list must contain exactly two nodes. As a trivial example, the representation of "γ dominates α and β, where α and β are sisters and α precedes β" is as follows:

(52) γ / [α , β]

With the representation of these structural relations defined, we will now present the choice of representations for the *nodes* of the tree. The classes of nonterminal and terminal nodes correspond to the $\overline{X}$ projections and the "lexical" nodes respectively.

There are three possible nonterminal nodes. The first is the *maximal projection*. This is the highest X^2 projection for the phrase (i.e. its *root*)[2]. We will call this the *phrasal node*, which has the following form:

(53) xmax(Category, ID, Features, Constraints)

where *Category* is the phrasal category, determined by the head. *ID* is a unique identifier for the phrase, used to distinguish it from other phrases. This is used by procedures which search the tree for a specific phrase. *Features* is a list of terms which indicate the features (such as agreement) and properties (such as θ-roles, and L-marking) assigned to the phrase. *Constraints* is used in syntactic analysis, and will be discussed in Section 4.2.

The remaining non-terminals represent the intermediate X^2 and X^1 projections. These are simply represented as follows:

(54) (a) xmax(Category)
(b) xbar(Category)

where *Category* is the same as that for the maximal node. Note that the X^2 maximal projection, and the X^2 intermediate projection are distinguished by their arity.

The terminal nodes are used to represent the actual lexical items, or leaves, of the parse tree. Specifically, they may be simple lexical specifiers or adjuncts to the head of a phrase (i.e. X^0). For these, we adopt the follow representation:

---

[2] We introduce the notion of the *maximal projection* as the "highest" X^2 projection of a phrase. This is not to suggest that there is a third $\overline{X}$ level. The distinction being made here is a purely categorial one, not typological. In general, the maximal projection refers collectively to all the X^2 projections. For convenience however, the features of the phrase are attached to the highest node, and it is also used as the *maximal projection* for purposes of m-command and government. This differs from [Chomsky 86a], which takes the first X^2 projection as determining the m-command domain.

(55) (a) spec(Category, Word, Features) *Specifier*
(b) adj(Category, Word, Features) *Adjunct*
(c) head(Category, Word, Features) *Head*

where,

(56) Category = the category of the lexical item.
Word = the lexical item itself.
Features = the list of morphological/syntactic features of the word.

Additionally, punctuation symbols have the following representation:

(57) punc(punc,Punctuation,Mode)

where,

(58) Punctuation = "?" or "."
Mode = "ques" or "decl"

Finally, we must be able to represent empty categories (*ec*'s), which are phonetically null, but nonetheless present in the structure. These have the general form:

(59) e_cat(Category, Type)

Where *Category* represents the category of the position occupied by the ec, and *Type* indicates whether the ec is an *np-trace*, *wh-trace* or *PRO*, represented as "trace(np)", "trace(wh)" and "pro", respectively.

3.3 Transformations

In this section, we will discuss the various types of movement transformations modeled by this system. Specifically, we will examine the basic movement phenomena as presented in Section 2.5, and the conditions under which they may occur in English and German. In as much as we are accounting for only a subset of each of these languages, we also only account for a subset of possible transformations. Specifically, these are head-to-head movement involving V, I, and C, wh-movement to the front of a matrix sentence (i.e. indirect questions are not handled), and NP-movement to a Case-marked position. These are fundamental substitution transformations, and no attempt is made here to account for the various types of adjunction transformations possible.

3.3.1 X^0-Substitution

We take X^0-Substitution to be movement of an X^0 to another head position. Within this system we account for only two such transformations: *verb-raising* and *inversion*.

Verb-raising is the movement of V to I, when I is not lexically filled. The result of the transformation is that the verbal element is appropriately inflected by the AGR features present in I. Consider for example the following two sentences:

(60) (a) Mozart [TNS_{past}] compose the sonata.
(b) Mozart composed the sonata.

In this example, the inflection of *compose* in (60a) is the result of moving it to the head of the inflected I phrase. As we might expect, the same holds for German as illustrated by the following embedded sentence constructions (recall that VP and IP are head final):

(61) (a) dass Brigitte nach Berlin fahren [TNS_{pres}]
(b) dass Brigitte nach Berlin fährt

Inversion is the movement of I to C, typically in the matrix clause (at least for our purposes). This transformation accounts for the so-called *Subject-Aux Inversion* (SAI) phenomena which appears in English and German question sentences. Consider the following:

(62) (a) The girl [TNS_{pres}] have read the book.
(b) Has the girl read the book?

The generation of (62b) is the result of *have* moving first to I, where it receives it inflection as above, and then to C, resulting in its pre-subject position. The same occurs in German, in the analogous sentence: *Hat das Mädchen das Buch gelesen*, where *haben* raises to the final I position, and then moves to the head of CP, at the front of the sentence. A difference arises between the two languages however, in that all German verbs may move to C, while in English only auxilliaries (e.g. *have* and *be*) are permitted to invert. The result is the insertion of the semantically null "do" element in the I position, in cases where no auxilliary is present and inversion is required[3]. This is illustrated by the following grammatical/ungrammatical examples:

(63) (a) The woman [TNS_{past}] write the letter.
(b) * Wrote the woman the letter?
(c) Did the woman write the letter?

In fact, inversion is an obligatory transformation in all German matrix clauses, resulting in the so-called verb-second phenomena, when combined with topicalisation. Additionally, it occurs in English wh-questions, both of these constructions are discussed below.

[3] In the present system, this is accomplished by simply checking to see if inversion is to take place after raising. If so, then *do* is inserted if no auxilliary can be raised. For further discussion, see Section 5.3 and the *raise_verb* predicate in Appendix E.

3.3.2 $\overline{\overline{X}}$-Substitution

We take $\overline{\overline{X}}$-Substitution to include any substitution transformation involving a maximal projection. We observed in Section 2.5 that such movement may only be to a specifier position. This may be derived from the following: a) maximal projections may only move to positions where maximal projection can appear (i.e. not to an X^0 position) by the Structure Preserving Hypothesis, and b) movement to a object position would violate the Projection Principle.

In the present treatment we are concerned primarily with two possible types of $\overline{\overline{X}}$-Substitution. The first is movement to the SPEC,IP position, known as *raising to subject*, and the second is movement to the SPEC,CP position, known as *movement to COMP*.

Raising to subject occurs in situations where an NP must move in order to receive Case. The two possibilities are raising of an object to subject, as in passive constructions, and raising of subject to subject. We are concerned here only with the latter. Consider the following examples:

(64) (a) e [TNS_{pres}] seem John to like Mary.
(b) John_i seems t_i to like Mary.

In this situation, the embedded subject must move to receive Case. An additional requirement is that the SPEC,IP position be a $\overline{\theta}$-position so as not to violate the θ-criterion. Finally, the trace, t_i, of the embedded subject must be properly governed so as to satisfy the ECP. Movement to Comp accounts for the movement of a wh-phrase to the front of a clause. In the case of the matrix clause, wh-movement is usually accompanied by inversion[4]. Consider the following:

(65) (a) The boy [TNS_{pres}] have put what on the table.
(b) What_i has the boy put t_i on the table?

In this example, the auxilliary verb inverts to the pre-subject position, and the wh-phrase moves to the front of the sentence in the SPEC,CP position. The same phenomena occurs in German, as in the sentence: *Was hat der Junge auf den Tisch gelegt.* In fact, the verb-second phenomena of German is the result of inversion, and movement of some *topic* to the SPEC,CP position. That is, if no wh-phrase has moved to the SPEC,CP position, then some other constituent is topicalised, for non question sentences. This is exemplified by the following:

(66) (a) Der Junge das Buch auf den Tisch legen haben [TNS_{pres}].
(b) Das Buch hat der Junge auf den Tisch gelegt.

In this example, *haben* raises to I and then inverts to C. In addition, some constituent is topicalised, in this case *das Buch*, resulting in the verb-second configuration of the matrix clause.

[4] Except in those cases where the matrix subject is the wh-phrase.

Chapter 4

Parsing with Principles

In constructing a parser based on the principles of GB theory, a variety of approaches may be taken. In Sharp's system [Sharp 85], possible S-structures are generated according to $\overline{X}$ phrase structure, and then well-formedness is determined by the principles. The principles are specified as *filters* on possible S-structure configurations. If a constraint of some principle is violated, the parser backtracks, generating a new S-structure. This process continues until the parser generates a grammatical S-structure.

The efficiency problems of the above strategy are obvious. That is, by postponing the application of well-formedness conditions until after the entire S-structure has been generated the effects of backtracking are magnified. An alternative approach is to apply relevant constraints *during* the parsing stage. In this way it becomes possible to block erroneous parses as early as possible. The model adopted here is roughly illustrated in Figure 4.1.

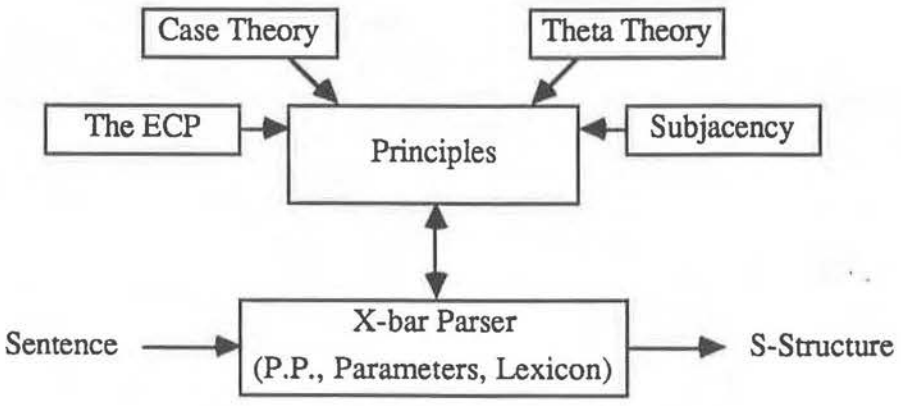

Figure 4.1: The Parsing Model

For our purposes, we take S-structure to include a representation of the sentence's phrase structure along with antecedent-trace relations of moved constituents (i.e. chains). In the present system, we exclude the recovery of NP coreference (Binding theory) and the reference of PRO (Control theory).

The syntactic analysis component of the system consists primarily of two modules. Specifically these are the *Parsing Module* and the *Constraint Module*. The parsing module generates candidate S-structures. As each phrase is completed, the constraint module is called to apply relevant principles and constraints.

The parsing module employs an $\overline{X}$-parser resembling that developed by Sharp. That is, the "language independent" rules of $\overline{X}$-theory are represented in *Definite Clause Grammar* (DCG) notation. This includes the basic $\overline{X}$ rules, as well as those for parsing arguments, adjuncts, and specifiers. The actual phrase structure rules used here are an instantiation of the $\overline{X}$-rules presented in Chapter 2, tailored for parsing by DCG's. In addition to $\overline{X}$-theory, the parser uses subcategorization information projected from the lexicon, combined with the Projection Principle, and language specific information about possible choices for adjuncts and specifiers.

The $\overline{X}$-parser constructs phrase structure trees during the derivation. The constraint module is applied to these trees as each maximal projection is parsed, in a bottom-up fashion. The result is a "co-routining" of the parser and the principles, which is not unlike the approach taken by Dorr [Dorr 87]. Furthermore, a facility is provided for passing information which is relevant to constraints, up the tree. The result is that violation of a given constraint may be detected as soon as possible, forcing the parser to backtrack at that point where the problem occurred.

In this chapter we will present the design, implementation, and operation of both the $\overline{X}$-parser and the constraint module. These two modules constitute the *syntactic analysis* component of the system.

4.1 The Parsing Module

The purpose of the $\overline{X}$-parser is to generate possible phrase structure representations for a given sentence. Since we are recovering S-structure, not D-structure, $\overline{X}$-theory alone is insufficient. The parser must also be able to account for moved constituents and hypothesize empty categories. To this end, several elements of θ-theory are incorporated directly in the parser. Specifically, the parser accesses subcategorization information of lexical items to drive the parsing of arguments. If a subcategorized phrase is not present, then it is assumed to have moved, and an empty category is inserted. Furthermore, the parser assumes that all IP's have a Subject specifier. These two features essentially capture the Extended Projection Principle. θ-marking and L-marking of subcategorized constituents is also performed by the parser, but θ-marking of the subject and enforcement of the θ-criterion are handled by the constraint module.

The parser is also provided with language specific knowledge about where possible adjuncts and specifiers may appear. Information concerning head position (w.r.t. arguments) is specified for the categories of each language. Additionally, information about where moved elements may

appear is included. These phrase structure parameters are accessed by the parser *during* the parsing stage. In this way, a distinction is maintained between the "universal" $\overline{X}$-parser and the language specific parameters.

In the parsing module, a *logic grammar* is used to specify the $\overline{X}$ phrase structure rules. Logic grammars, originally developed by Colmerauer, permit the specification of typical rewrite rules using logical terms as grammar symbols [Colmerauer 78]. That is, grammar symbols may be functors containing arguments. The arguments, or *informants* [Dahl *et al.* 86], may be used to pass information via Prolog's inherent unification mechanism.

The specification of logic grammar rules is itself a specification of a parser. That is, Prolog's underlying theorem prover can be used to derive or parse strings for the language specified. As Prolog uses a top-down, left-to-right, backtracking theorem prover, the result is essentially a recursive descent parser for the logic grammar [1]. In the present system we use the *Definite Clause Grammar* (DCG) formalism [Pereira *et al.* 80], which is a restricted form of Colmerauer's *Metamorphosis Grammars* (MG) [Colmerauer 78].

4.1.1 Parsing $\overline{X}$ Phrase Structure

The $\overline{X}$ phrase structure rules outlined in Sections 2.3, and 3.2, present a problem for the DCG formalism. Specifically, they contain left recursive rules. The phrase structure template adopted for parsing purposes is roughly the following:

(67) (a) $\overline{\overline{X}} \rightarrow \mathit{Spec}\ X^2$
(b) $X^2 \rightarrow \mathit{Adjuncts}\ X^1\ \mathit{Adjuncts}$
(c) $X^1 \rightarrow \mathit{Arguments}\ X^0$
(d) $X^1 \rightarrow X^0\ \mathit{Arguments}$

It is important to note that this is not assuming a three level $\overline{X}$ system. The purpose of proposing distinct rules for (67a&b) is to permit the parsing of only one Specifier, and enforce its phrase initial position (i.e. the highest, leftmost position, at least for English and German). The DCG rules corresponding to (67a&b) are specified as follows:

(68)

```
xmax(L,C,Tree,S,NS)     -->   spec(L,C,TXmax,STree,S,S1),
                              xmax2(L,C,TXmax,S1,NS),
                              {agree(L,STree,BTree),gen_ID(BTree),
                               constraints(L,C,BTree,Tree)}.

xmax2(L,C,Tree,S,NS)    -->   adjunct(L,pre,C,TApost,Tree,S,S1),
                              xbar(L,C,TXbar,S1,S2),
                              adjunct(L,post,C,TXbar,TApost,S2,NS).
```

[1] In fact, other parsing strategies may be used in parsing logic grammars. See for example [Abramson *et al.* 88] which employs the LL(k), LALR(k), and Earley parsing algorithms, among others.

These define the rules for parsing the maximal and intermediate X^2 projections for a language L, of a category C. The third argument, $TREE$, contains the parse tree representation of the current phrase. It is important to note that the parse tree does not exactly reflect the derivation of the DCG rules used, but rather is constructed in accordance with the representations presented in Section 3.2. The final two arguments represent the old and new lookahead lists, S and NS, respectively.

The three Prolog calls at the end of the first rule invoke the *agree*, *gen_ID* and *constraint* routines. The former ensures that the agreement relations between various elements are maintained (such as Spec/Noun and Subject/Verb agreement). The *gen_ID* predicate simply assigns a unique identifier to the maximal projection. The latter calls the constraint module, which applies each of the relevant GB principles to the current subtree.

The following rules correspond to those of (67c&d), used for parsing arguments:

(69)
```
xbar(L,C,Tree,S,NS)      -->   {head_position(L,C,initial),!},
                               xmin(L,C,Args,HD,S,S1),
                               {stack(L,initial,HD,S1,S2)},
                               arg(L,post,C,Args,HD,[],Tree,S2,NS).

xbar(L,C,Tree,S,NS)      -->   {head_position(L,C,final),
                               HD=X/head(C,_,_),
                               stack(_,final,head(C,W,_,F),S,R),!,
                               getargs(L,head(C,W,As,F))},
                               arg(L,pre,C,As,HD,[],Tree,R,S1),
                               xmin(L,C,As,HD,S1,NS).

xbar(L,C,Tree,S,NS)      -->   {head_position(L,C,final),!,
                               poss_empty(L,C,As),
                               HD=X/head(C,empty,F)},
                               arg(L,pre,C,As,HD,[],Tree,S,NS1),
                               xmin(L,C,As,HD,NS1,NS).
```

Each rule begins by checking the *head_position* parameter, which determines whether the head of a phrase is initial or final with respect to its arguments. The parameter has the following form:

(70) head_position(L,C,Position).

where for a language L, and a phrase of category C, *Position* specifies the head as being "initial" or "final" with respect to its arguments. This head position parameter is language specific, and must be set for each category of a language. We take all categories to be initial, except for V and I in German.

The first rule, for head initial phrases, is straight forward. The head is parsed by ***xmin***, which returns the subcategorization frame in *As*. Each argument is then parsed by ***arg***, which is discussed in more detail below. The second two rules handle head final cases. Since the head's subcategorization frame is necessary to parse the initial arguments, the routine gets the head by inspecting the lookahead stack (using the *stack* predicate). If the head is found, then *getargs* is used to retrieve the frame. The final rule handles the situation where the final head is not present, either due to absence, or movement. The *poss_empty* parameter, shown in (71) specifies that a category C for language L may be empty for the above reasons. The default subcategorization frame, for categories such as C and I which are predictable, is stated in *As*.

(71) poss_empty(L,C,As).

The final two rules are used to parse the X^0 level of a phrase:

(72)

```
xmin(eng,C,Args,Tree,[HD|S],S)     -->   [],
                                         {HD = head(C,W,Args,F),
                                         poss_move(eng,c(mat/sent),C,_),
                                         head_anal(eng,C,W,HD,initial),
                                         getargs(eng,head(C,W,Args,F)),
                                         Tree = xbar(C)/head(C,empty,_),!}.
xmin(L,C,Args,xbar(C)/Tree,S,S)    -->   head(L,C,Args,Tree).
```

The first rule handles the case where a V or I element has moved to the matrix C position (i.e. inversion), and is not present at I. The purpose of the clause is to determine what the moved element was, so that the appropriate subcategorization information can be retrieved. The second rule simply parses the head of the phrase as follows:

(73)

```
head(L,C,As,head(C,W,F))           -->   [Word],
                                         {head_anal(L,C,Word,HD,initial),!,
                                         HD = head(C,W,_,F),
                                         getargs(L,head(C,W,As,F))}.
head(L,C,As,head(Cat,W,F))         -->   [Word],
                                         {head_anal(L,Cat,Word,HD,initial),
                                         HD = head(Cat,W,Args,F),
                                         poss_move(L,C,Cat,As),!}.
head(L,C,As,head(C,W,F))           -->   [head(C,W,_,F)],
                                         {!,getargs(L,head(C,W,As,F))}.
head(L,C,As,head(Cat,W,F))         -->   [head(Cat,W,Args,F)],
                                         {poss_move(L,C,Cat,As),!}.
head(L,C,As,head(C,empty,F))       -->   [],
                                         {poss_empty(L,C,As),!}.
```

The first two rules parse initial heads. The first rule will parse a head, and retrieve its subcategorization frame, used by the *xbar* rules. The second rule handles the case where a head of category *Cat*, has moved to head of a phrase of category C. This possibility is verified by the *poss_move* parameter which has the following form:

(74) poss_move(L,C1,C2,Args).

This states that for language L, a head of category $C2$ may move to the head of a phrase of category $C1$. The default subcategorization frame for $C1$ is specified in *Args*. The various possibilities for head movement must be stated for each language, as they vary from one to the other, and are not handled in a principled fashion by the present system. The second pair of rules are similar to the first two, with the exception that they parse final heads which have been inserted on the lookahead stack. The final rule parses those heads which are not present, and calls the *poss_empty* parameter mentioned above.

4.1.2 Parsing Specifiers, Adjuncts, and Arguments

In addition to the $\overline{X}$ rules outlined above, rules are necessary to parse the specifiers, adjuncts and arguments referred to by the above rules. That is, the DCG rules for the *spec*, *adjunct* and *arg* non-terminals must be specified.

The rules for parsing specifiers are relatively straightforward. The first clause parses the specifier, while the second clause permits the specifier to be absent for certain categories.

(75)
```
spec(L,C,TX,xmax(C,ID,[],Cons)/[TSpec,TX],S,NS)   -->   spec(L,C,TSpec,S,NS).
spec(L,C,X/R,xmax(C,ID,[],Cons)/R,S,S)            -->   [],{no_spec(L,CList),
                                                          member(C,CList)}.

no_spec(L,[n(_),v(_),p(_),c(mat/sent)]) :- !.
```

The choice of specifier is set for each category of a language. In this system, we take wh-phrases to be the specifiers for CP, and determiners for NP, in English. German is similar, with the addition that it may take a *topic* as a specifier to the matrix CP. We return to the parsing of wh-phrases and topics below.

The rules for parsing adjuncts are very similar to those for specifiers. The first rule below handles a special case for German, where the highest verb dominated by an infinitival IP must adjoin to the right of I. This can be illustrated by the sentence *Ich versuchte das Buch zu sehen* ("I tried to see the book"), where *sehen* has moved to an adjoined position to the right of *zu*[2]. Note that the remaining two rules are ordered such that the parser will first assume no adjuncts.

[2] In fact, this is a case where head movement is an adjunction transformation, in contrast with head-to-head movement which is substitution. This is the only example of adjunction dealt with by the present system, and it is incorporated only because of its obligatory application in German infinitivals.

This generally improves parse times, and also tends to yield preferred attachment preferences for PP's in accordance with the *Right Association* principle [Kimball 73] [3].

(76)
```
adjunct(ger,post,C,TX,xmax(C)/[TX,Tadj],S,S)      -->  {C = i(tns(-)),!},
                                                       [head(v(T),W,F)],
                                                       {Tadj = head(v(T),W,F)}.
adjunct(L,Pos,C,X/Tree,xmax(C)/Tree,S,S)          -->  [].
adjunct(L,pre,C,Tree,xmax(C)/[Tadj,Tree],S,NS)    -->  adj(L,pre,C,Tadj,S,NS).
adjunct(L,post,C,Tree,xmax(C)/[Tree,Tadj],S,NS)   -->  adj(L,post,C,Tadj,S,NS).
```

The possible choices of adjuncts in the present system are PP and CP (relative clause) adjuncts to NP's, and PP adjuncts to VP's. The choices of specifier and adjunct possibilities appear in Appendix C.

The rules for arguments attempt to parse each constituent in the subcategorization frame for the head of the phrase. The rules are as follows:

(77)
```
arg(L,Pos,C,[],HD,ALst,T,S,S)                  -->
                    [],{build_tree(Pos,C,HD,ALst,T)}.
arg(L,Pos,C,As,HD,ALst,T,S,NS)                 -->
                    {select(L,C,As,A,NewAs)},
                    a_xmax(L,C,A,Tx,S,S1),
                    arg(L,Pos,C,NewAs,HD,[Tx|ALst],T,S1,NS).
arg(L,Pos,C,[A|Args],HD,ALst,T,S,NS)           -->
                    a_xmax_e(L,C,A,Tx),
                    arg(L,Pos,C,Args,HD,[Tx|ALst],T,S,NS).
```

The first three arguments indicate language, head position, and category. The fourth informant is the list of categories of the subcategorized constituents. The fifth and sixth arguments represent the head of the phrase and the list of parsed arguments. The first rule is selected when all the arguments have been parsed. It calls *build_tree* with the head and list of parsed arguments, to construct the phrase structure representation which is returned in the seventh informant T. The second rule attempts to parse an argument by calling *a_xmax*, and then calls *arg* recursively. The *select* predicate is used to retrieve an argument category from the subcategorization frame. This routine can be used to enforce the constituent order requirements for a specific language (i.e. fixed or free). The third rule is applied if the argument is not present, and parses an empty category using *a_xmax_e*. The *a_xmax* and *a_xmax_e* predicates are identical to *xmax* and *xmax_e* except that the former perform the θ and L-marking of the parsed phrases, since they are subcategorized.

The following special rules are introduced to parse wh-phrases:

[3] For further discussion of the RA principle and parsing, see [Pereira 85].

(78)

```
wh_phrase(L,c(mat/_),Tree,S,NS)    -->    {empty_cat(L,C)},
                                          wh_xmax(L,Cat,C,Tree,S,NS).
wh_phrase(L,c(emb/T),Tree,S,S)     -->    [],{gensym(e,ID)},
                                          {set_ec(L,T,ID,Tree)}.
```

The first rule attempts to parse a lexical wh-phrase in the specifier position of the matrix CP. Specifically, it will try to parse any phrase that can have *wh* status, which is specified by the *empty_cat* parameter for each language (taken to be NP and PP here). The *wh_xmax* rule is similar to that for *xmax* with the addition of a check for wh-morphology. The second clause will parse an empty operator as the specifier of a relative clause or a comp-trace as the specifier of an embedded sentence. The comp-trace may be used in the construction of chains, as an intermediate position.

Topics with verb-second are a German phenomenon, and are treated here roughly as wh-phrases. Specifically, the topic is a constituent that has moved to the matrix SPEC,CP position, but is not a wh-phrase (i.e. it does not have *wh* morphology) [4]. The rule for parsing the topic is as follows:

(79)

```
topic(L,c(mat/sent),Tree,S,NS)    -->    {empty_cat(L,C)},
                                         spec(L,C,TXmax,XTree,S,S1),
                                         xmax2(L,C,TXmax,S1,NS),
                                         {add_feature([ant(+),case(_)],XTree,Tr),
                                          agree(L,Tr,BTree),gen_ID(BTree),
                                          constraints(L,C,BTree,Tree)}.
```

Again, this is similar to the *xmax* rule, with the exception that the topic is marked as an antecedent, much as a wh-phrase. This is accomplished by adding the *ant(+)* term to the phrase's feature list, using the *add_feature* predicate.

4.2 Principles as Constraints

The $\overline{X}$-parser uses $\overline{X}$-theory and elements of θ-theory (notably, the Projection Principle and subcategorization) to generate candidate S-structures. During the course of parsing, the constraint module is consulted for the purpose of constructing chains and verifying well-formedness. Specifically, the constraint module is invoked as each maximal projection is completed, as shown in the first clause of (68). The effect is that the constraints are applied in a bottom-up, left-to-right manner.

[4] Topicalisation phenomena in German is much more complex than the treatment here reflects. In reality, a variety of phrasal categories may be topicalised, and the transformation is not restricted to matrix clauses. For more discussion see [Haider *et al.* 85].

Each principle comprises a component of the module. Here, these are taken to be the ECP, Case theory, θ-theory and Bounding theory. Each component is constructed such that only the subtree of the current maximal projection is accessible. That is, no information about higher, previously parsed constituents is available[5]. In this way, the constraints can be considered "local" in a very strict sense. Specifically, a constraint may affect only the m-command domain for the head of the current projection – a natural restriction as many of the principles are stated in terms of government.

While the constraints may be formulated so as to require only information that is contained in the current subtree, this can be very inefficient. The ECP for example will spend time looking for *ec*'s in every subtree, even when there are none present. Additionally, some mechanism is necessary to keep track of partially constructed structures, notably chains. To deal with both of these problems we introduce the notion of a *constraint list*. Specifically, these lists maintain that information about the current subtree which is of direct relevance to the principles. That is, once the constraint module has been applied to a phrase, it may wish to pass information up the tree, for later consideration. The constraint lists are stored as the fourth argument of the phrasal node, shown in (53) and repeated here for convenience:

(80) xmax(Category, ID, Features, Constraints)

The *Constraints* variable is left uninstantiated by the parser, for use by the constraint module. The constraint module first merges the constraint lists of the immediately dominated phrases. As each principle is applied, the list may be modified and information no longer relevant is removed. When all the constraints have been applied, any new information may be added to the list, and *Constraints* becomes so instantiated. The constraint list affiliated with a phrasal node therefore represents the "state of affairs" for that node's subtree at the conclusion of the constraint module's application. We refer to information added to the constraint list as *constraint requests*, since their purpose is to request the attention of the individual constraints. The range of possible requests is as follows:

(81)	(a)	case(ID,Case)	A request for Case by an NP.
	(b)	theta(ID)	A request for an external θ-role.
	(c)	ec(Cat,ID,Type)	An *ec* requests an antecedent.
	(d)	ant(Type,Cat,ID)	An antecedent requests a trace.
	(e)	chain(Type,ID,Cat,Case,Theta,Chain)	A partially constructed chain.
	(f)	c_chain(Type,ID,Cat,Case,Theta,Chain)	A completely constructed chain.

In each case, the *ID* argument is used to indicate the position of the phrase which issued the request. That is, *ID* identifies the location of the NP requesting Case, the *ec* requesting an antecedent, and so on. In (81c-f), *Cat* simply indicates the category of the individual trace,

[5] Recall that the parser does employ a top-down strategy, making access to the left context theoretically accessible. In the present system however, only the current subtree is available at each node.

antecedent, or chain (i.e. PP or NP). In the case of empty categories, *Type* is used to indicated if the *ec* is np-trace, wh-trace, or PRO. For antecedents however, *Type* indicates whether the moved phrase is in an A or $\overline{A}$ position (i.e. as "a" or "abar"). For chains, *Type* is similar to that for antecedents, with the additional possibility of "pro" chains. This will be discussed in more detail below in the Section 4.2.5. In (81e&f), *Case* and *Theta* represent the Case and θ marked position of the chain, while *Chain* is a list of each of the positions which make up the chain. Each of the constraint requests will be discussed in more detail with respect to their function in the Sections below.

4.2.1 Applying Constraints

The highest level of the constraint module involves three stages, as shown in the following Prolog segment:

(82)
```
constraints(L,C,Tree1,Tree2) :-
        get_constraints(Tree1,Cons),
        satisfy_constraints(L,C,Tree1,Cons,Tree2,Constraints),
        add_constraints(Tree2,Constraints).
```

The *get_constraints* predicate merges the constraint lists of each maximal projection which is immediately dominated by the current phrase. In doing so the constraint module becomes aware of what chains have been partially constructed, and what requests remain to be satisfied. The *satisfy_constraints* predicate applies each of the constraints to the phrase, passing the revised tree and constraint list as arguments. The Prolog code is stated simply as follows:

(83)
```
satisfy_constraints(L,C,Tree1,Constraints,Tree3,NewConstraints) :-
        ecp(L,C,Tree1,Constraints),
        case_theory(L,C,Tree1,Tree2,Constraints,Constraints2),
        theta_theory(L,C,Tree2,Constraints2,Tree3,Constraints3),
        subjacency(L,C,Tree3,Constraints3,NewConstraints).
```

We discuss each of these constraint routines in the remaining subsections of this chapter.

The *add_constraints* routine generates any new constraint requests appropriate for the phrase itself. Specifically, "case" requests are issued for each lexical NP, "ant" requests are issued for each non θ-marked argument, and "theta" requests are issued by VP's which assign an external θ-role. The "ec" requests are not added by this predicate, but are inserted into the constraint list during parsing by the *xmax_e* and *a_xmax_e* rules. The "chain" requests are created and updated by the *subjacency* routine, based on *ec*'s.

4.2.2 The ECP

The Empty Category Principle (ECP) is instrumental in licensing empty categories. From a parsing point of view, it determines whether an empty category is a *trace* or PRO. We adopt the ECP basically as stated in (22), and repeat it here for convenience:

(84) **Extended ECP:** If α is an empty category, then
(i) α is trace iff it is *properly governed*
(ii) α is PRO iff it is ungoverned.

In addition, we stated that trace was *properly governed* if and only if it was governed by lexical head, or locally $\overline{A}$-bound (see (23)). The ECP constraint deals only with those cases where proper government is government by a lexical head, and defers those cases where a trace may be $\overline{A}$-bound until later. That is, it determines if an *ec* is a trace, by checking to see if it is governed by a lexical item. The following specifies the appropriate Prolog code for this:

(85)
```
ecp(L,i(_),Tree,Constraints) :- !.
ecp(L,C,Tree,Constraints) :-
        exists_ec(Constraints,Tree,ID,Type) ->
                ((properly_governs(L,C,Tree,xmax(Cat,ID,_,_)) ->
                        Type = trace(WH) ; Type = pro),
                ecp(L,C,Tree,Constraints)),!.
ecp(L,C,Tree,Constraints) :- !.

exists_ec(Constraints,Tree,ID,Type) :-
        member(ec(_,ID,Type),Constraints),
        var(Type).
```

The first clause causes the constraint to ignore a subject *ec*, since it won't be governed by a lexical category within the IP projection. Determination of the subject as either trace or PRO, is decided by virtue of it being locally $\overline{A}$-bound or not. This is not performed by the *ecp* routine, but rather by the Subjacency constraint or the Case constraint, both of which are discussed below.

The second clause captures the rest of the ECP definition, which states that if an *ec* is governed by a lexical category it must be trace, and is otherwise PRO. For our purposes, the *proper_government* predicate is defined strictly as government by a lexical head (i.e. N, V, A, or P)[6]. In this clause, the first goal checks if there exists an empty category in the constraint list.

[6] In the present system, adjectival phrases are not considered. So in fact, only N, V, and P are taken to be lexical heads here.

Then, if the *ec* is properly governed[7] it is marked as a trace, otherwise as PRO. Note, since PRO can only appear in subject position, this predicate will never actually mark an *ec* as PRO in a grammatical situation.

4.2.3 Case Theory

In the present system, Case theory has two effects. Firstly, it assigns Case to phonetically realised noun phrases. That is, if a noun phrase has requested Case, it must be assigned Case by the governing lexical item. Secondly, once an empty category has been identified as a trace (by the ECP), Case theory is used to determine if it is an np-trace (i.e. it is not assigned Case) or a wh-trace (i.e. it is assigned Case) [8]. Note, that a subject *ec* of a tensed clause will not be dealt with by the ECP constraint, but will be marked as wh-trace by the Case routine. The high-level predicate for the Case principle is as follows:

(86)
```
case_theory(L,C,Tree1,Tree3,Cons1,Cons2) :-
        case_assigner(L,C,Tree1,Trans),!,
        case_mark_lexical(Tree1,Tree2,Trans,Cons1,Cons2),
        case_mark_traces(Tree2,Tree3,Trans,Cons2),!.
case_theory(L,C,Tree,Tree,Cons,Cons) :- !.
```

The first goal, *case_assigner*, determines whether or not the head of the current maximal projection is a Case assigner. If not, then the predicate trivially succeeds, and application of the constraint is abandoned. If so, then any Case requests must be satisfied. The most crucial of the requests is that of a lexical NP, which if not satisfied will cause the parser to backtrack. The predicate for Case marking the lexical NP's is as follows:

---

[7] In the present system, we take the m-command domain of a given head to be everything dominated by its *highest* maximal projection. This is contrary to Chomsky's *Barriers* formulation [Chomsky 86a], in which the m-command domain is determined by the *first* maximal projection dominating the head. In this way we predict that an *ec* adjunct of a lexical category is trace, and not PRO.

[8] This is included in Chomsky's statement of the *Generalised Empty Category Principle* [Chomsky 81a] which is stated as follows:

Generalised ECP: If α is an empty category, then
(i) α is PRO if and only if it is ungoverned
(ii) α is trace if and only if it is properly governed
(iii) α is a variable only if it is Case-marked

(87)

```
case_mark_lexical(Tree1,Tree3,Trans,Cons1,Cons3) :-
        remove(case(ID,Case),Cons1,Cons2),!,
                governed(xmax(n(_),ID,_,_),Tree1,Trans),
                mark_node(ID,case(ID),Tree1,Tree2),
                case_mark_lexical(Tree2,Tree3,Trans,Cons2,Cons3),!.
case_mark_lexical(Tree,Tree,Trans,C,C) :- !.
```

If an NP dominated by the current maximal projection requires Case, then the *case(ID, Case)* request will appear in the constraint list. This request indicates the location of the NP (i.e. the node *ID*), and the *Case* requested (e.g. nominative, accusative, etc.). If the NP is governed, then it is Case-marked. This is accomplished by adding the *case(ID)* feature to the NP's feature list, using the *mark_node* predicate. If the NP is not governed by the Case assigner, then the routine fails. Note that wh-phrases and topics do not issue a Case request, since their traces must by Case-marked.

If there exists a trace in the current set of constraint requests, Case theory is used to distinguish np-trace from wh-trace, as mentioned above. The predicate for Case marking traces is specified as follows:

(88)

```
case_mark_traces(Tree1,Tree3,Trans,Constraints) :-
        exists_trace(Constraints,Tree1,ID,WH) ->
                (governed(xmax(Cat,ID,_,_),Tree1,Trans) ->
                        (mark_node(ID,case(ID),Tree1,Tree2), WH = wh) ;
                        ( Tree2 = Tree1 , WH = np)),
                case_mark_traces(Tree2,Tree3,Trans,Constraints),!.
case_mark_traces(Tree,Tree,Trans,Constraints) :- !.

exists_trace(Constraints,Tree,ID,WH) :-
        member(ec(Cat,ID,trace(WH)),Constraints),var(WH).
```

The first goal checks to see if there are any traces. If there is a trace, the predicate determines if it is governed, and if so the node is Case marked (as for lexical NP's), and the trace is marked as a wh-trace. If the trace is not Case marked, then it is marked as an np-trace.

4.2.4 θ-Theory

While θ-marking of arguments is performed by the parser, θ-marking of the subject is controlled by the constraint module. The θ constraint only applies at IP, and succeeds trivially for other maximal projections. Specifically it handles two cases: that where the subject is assigned a θ-role, and that where it is not.

The first case is implemented by the following clause:

(89)

```
theta_theory(L,i(_),Tree1,OldCons,Tree2,NewCons) :-
        remove(theta(+),OldCons,NewCons),!,
                Tree1 = X/[Subj1,Rest],
                Subj1 = xmax(_,ID,_,_)/_,
                add_feature(theta(ID),Subj1,Subj2),
                Tree2 = X/[Subj2,Rest].
```

If the subject is assigned a θ-role, then the *theta(+)* term will be present in the constraint list. If so, the subject position is marked as receiving a θ-role by adding the *theta(ID)* feature to the subject node, where *ID* indicates the subject's position.

The second clause handles the situation where the subject is not assigned a θ-role, indicated by the lack of the *theta(+)* term in the constraints list. This possibility is treated by the following clause:

(90)

```
theta_theory(L,i(_),Tree,OldCons,Tree,NewCons) :-
        \+ member(theta(+),OldCons),
        Tree = X/[Subj,_],
        \+ pleonastic(L,Subj),
        \+ Subj = _/e_cat(_,_),!,
                Subj = xmax(_,ID,_,_)/_,
                append([ant(a,n(_),ID)],OldCons,NewCons).
theta_theory(L,C,Tree,Constraints,Tree,Constraints) :- !.

pleonastic(eng,NP) :- get_head(NP,head(_,it,_)).
pleonastic(ger,NP) :- get_head(NP,head(_,es,_)).
```

In this case there are two possibilities: 1) the subject is a pleonastic element (e.g. *it* or *es*)[9], or 2) the subject is occupied by a referential NP which has received its θ-role elsewhere. In the case of the former, the predicate simply succeeds. In the latter instance, the requirement that the subject is an antecedent to an np-trace is added to the constraint list. This is done by adding the term *ant(a,n(_),ID)* to the list, where "a" indicates the antecedent is in an A-position, "n(_)" indicates it's an NP, and "ID" specifies its location. This constraint term is described in the following section, which discusses the implementation of Bounding theory.

4.2.5 Subjacency and Movement

In this section, we discuss the treatment of moved constituents. Specifically, this section of the constraint module is concerned with the recovery of *chains*. This involves initialising chains when

[9] English in fact has another pleonastic element, "there".

θ-marked traces are encountered, and prepending each Subjacent intermediate position to the chain until the antecedent is found. In addition to the Subjacency principle, this routine incorporates elements of the ECP, Case and θ-theory. More precisely, the Case filter is verified by ensuring that each NP chain receives Case exactly once. The θ-criterion is enforced by requiring that each chain receive exactly one θ-role. Finally, a mechanism is introduced whereby a subject *ec* of an infinitival can be determined as either np-trace or PRO, thus completing the implementation of the ECP. This is accomplished by introducing "PRO" chains, which have the effect of determining whether or not the *ec* is $\overline{A}$-bound. In the discussion that follows, we will present in more detail how the chains are constructed, and how the above principles are incorporated.

The highest level predicate of the *subjacency* routine is as follows:

(91)
```
subjacency(L,C,Tree,OldCons,NewCons) :-
        bind_traces(L,C,Tree,OldCons,Cons1),
        check_pro_chains(L,C,Tree,Cons1,Cons2),
        \+ member(barrier(ID),Cons2),
        barriers(L,C,Tree,Cons2,NewCons).
```

The first goal, *bind_traces*, controls the construction of chains. The second goal, as we shall see, is involved in implementing the remaining portion of the ECP. The third goal ensures that Subjacency is not violated, while the final goal determines if the present node is a barrier or not, for "higher" applications of the constraint.

The *bind_traces* predicate is by far the most involved. Its function is to initialise, construct, and close chains. The first two clauses, shown in (92) handle the cases where a chain of unknown type is present in the constraint list. These chains are created when an *ec* is found in a subject position of an infinitival IP (and may be either PRO or trace). We take advantage of the idea that if the IP is dominated by a CP, the *ec* may be locally $\overline{A}$-bound, but if the IP is dominated by a VP, the *ec* must be properly governed (by the V) and hence be an np-trace. As a result, if the first node dominating the chain is a CP, then we suggest that the chain is of type "pro", meaning it may or may not be locally $\overline{A}$-bound. We will return to this possibility below in the discussion of the *check_pro_chains* predicate. If the first node dominating the chain is a VP, then we know the subject of the embedded IP is an np-trace, since an intervening CP node would have caused the previous clause to be applied. In this case the chain must be an A-chain (i.e. type is "a"). This occurs in subject raising contexts, with verbs like *seems*, and is handled by the second clause.

(92)
```
bind_traces(L,c(_),Tree,OldCons,NewCons) :-
        member(chain(Type,ID,CC,Case,Theta,List),OldCons),
        var(Type),!,
        get_head(Tree,head(_,empty,_)),
        Type=pro,
        bind_traces(L,c(_),Tree,OldCons,NewCons),!.
bind_traces(L,v(_),Tree,OldCons,NewCons) :-
        member(chain(Type,ID,CC,Case,Theta,List),OldCons),
        var(Type),!,Type=a,set_last(trace(np),List),
        bind_traces(L,v(_),Tree,OldCons,NewCons),!.
```

The third clause shown in (93), handles the closure of chains. This occurs when the antecedent has been reached, and added to the chain. The first goal indicates the possible relationships which may hold between an antecedent and its chain. That is, if the antecedent is in an A-position, so must the current head of the chain (thus forbidding movement from an $\overline{A}$ to an A position). If the antecedent is in an $\overline{A}$-position, then any type of chain is possible (i.e. "a", "abar", or "pro"). The second and third goals simply remove the antecedent and chain from the constraint list, while the fourth sets the tail *ec* of the chain to wh-trace, if it was a pro-chain. Finally we insert the completed chain, or *c_chain*, into the constraint list.

(93)
```
bind_traces(L,C,Tree,OldCons,NewCns) :-
        member((At/Ct),[(a/a),(abar/a),(abar/abar),(abar/pro)]),
        remove(ant(At,CC,ID),OldCons,Cns1),
        remove(chain(Ct,IDC,CC,Case,Th,List),Cns1,Cns2),
        (Ct=pro,set_last(trace(wh),List) ; true),
        subjacent(ID,IDC,Tree,Cns2,Cns3),
        agree_case(ID,Tree,Case,Th),
        append([c_chain(At,ID,CC,Case,Th,[ant(At,CC,ID)|List])],Cns3,NewCns),!.
```

The following clauses handle the creation of new chains. The first creates a chain of unknown type for the subject of an infinitival IP, as mentioned above. The second may bind an intermediate trace to an existing, Subjacent chain via the *bind_to_chain* predicate. Alternatively, it will initiate a chain for a trace in a θ-marked position. The *make_chain* routine is relatively straightforward, and the reader is referred to Appendix D, for a listing of the Prolog code.

(94)
```
bind_traces(L,i(tns(-)),Tree,OldCons,NewCons) :-
        exists_ec(OldCons,Tree,ID,Type) ->
                make_chain(Tree,ec(Cat,ID,Type),Unknown,OldCons,Cons1),
        bind_traces(L,i(tns(-)),Tree,Cons1,NewCons).
```

```
bind_traces(L,C,Tree,OldCons,NewCons) :-
        member(ec(Cat,ID,trace(T)),OldCons),!,
                (bind_to_chain(ec(Cat,ID,trace(T)),Tree,OldCons,Cons1);
                make_chain(Tree,ec(Cat,ID,trace(T)),a,OldCons,Cons1)),
        bind_traces(L,C,Tree,Cons1,NewCons).
bind_traces(L,C,Tree,Cons,Cons) :- !.
```

The *bind_to_chain* predicate, shown in (95), appends intermediate traces (i.e. those traces in SPEC,CP or SPEC,IP, $\overline{\theta}$-positions). The routine simply removes the old chain, verifies that the intermediate trace is Subjacent, and inserts the new chain, with the new trace, into the constraint list. The code is as follows:

(95)
```
bind_to_chain(ec(C,ID,trace(T)),Tree,OldCons,NewCns) :-
        remove(ec(C,ID,Type),OldCons,Cns1),
        member(ChT,[a,abar,pro]),
        remove(chain(ChT,IDC,C,Cse,Th,List),Cns1,Cns2),
        subjacent(ID,IDC,Tree,Cns2,Cns3),
        agree_case(ID,Tree,Cse,Th),
        append([chain(ChT,ID,C,Cse,Th,[ec(C,ID,trace(T))|List])],Cns3,NewCns),!.
```

The interesting point here is the *agree_case* call, which ensures that any chain is assigned Case and a θ-role exactly once. This captures half of the θ-criterion, and part of the Case filter. The Prolog to enforce these constraints is as follows:

(96)
```
agree_case(ID,Tree,Case,Theta) :-
        get_subtree(ID,Tree,xmax(C,ID,F,Cons)/R),!,
        (member(case(IDC),F),Case1=case(IDC);true),
        (member(theta(IDT),F),Theta1=theta(IDT);true),!,
        Case=Case1,Theta=Theta1,!.
```

In the above discussion, we have mentioned *pro-chains*. These arise from the ECP condition which states that an *ec* is a trace if it is "locally $\overline{A}$-bound". This notion is not locally determinable, since we don't know if the *ec* is locally bound until an antecedent has been discovered. This possibility arises in the case of the subject position of an infinitival IP, where an *ec* may either be a wh-trace, or PRO. In these cases, a chain of type "pro" is created, indicating that if an antecedent is found the *ec* is a trace for it, otherwise it is PRO. These chains have the lowest priority. That is, if two chains are competing for a single antecedent, the non-pro-chain wins[10].

[10] In fact while this strategy works reasonably well, there do exist certain constructions which pose potential problems. Specifically, these are known as *parasitic gaps* (for discussion see [Taraldsen 81], [Engdahl 83], [Chomsky 82]).

The following code simply removes any pro-chain, if no Subjacent antecedent could be found, and sets the tail (i.e. the base *ec*) to "pro":

(97)
```
check_pro_chains(L,C,Tree,Cons1,Cons4) :-
        remove(chain(pro,ID,CC,Case,Theta,List),Cons1,Cons2),
        remove(barrier(ID),Cons2,Cons3),
        set_last(pro,List),
        check_pro_chains(L,C,Tree,Cons3,Cons4),!.
check_pro_chains(L,C,Tree,Cons,Cons) :- !.
```

The barriers predicate determines if the current maximal projection is a barrier. Recall, that Chomsky's notion of barrier is a relative one, in which γ is a barrier for α under certain conditions [Chomsky 86a]. In the present system however, we have adopted the definition proposed by Lasnik and Saito, in which all non-L-marked maximal projections are taken to be barriers for movement [Lasnik *et al.* 88b]. The code is represented as follows:

(98)
```
barriers(L,C,Tree,OldCons,NewCons) :-
        is_barrier(Tree),
        member(chain(Type,ID,CC,Case,Theta,List),OldCons),
        \+ member(barrier(ID),OldCons),!,
        append([barrier(ID)],OldCons,Cons1),
        barriers(L,C,Tree,Cons1,NewCons).
barriers(L,C,Tree,Cons,Cons) :- !.

is_barrier(xmax(C,ID,Features,Cons)/R) :-
        \+ member(l_marked,Features),!.
```

If the node is a barrier, that is it is not l-marked, then a *barrier(ID)* constraint is added to the constraint list for each chain (where ID identifies the specific chain).

Chapter 5

Principle-Based Translation

In this chapter we present the implementation of the translation component of the system. Specifically, this component translates S-structures of a "source" language into S-structures of a "target" language. To accomplish this, we capitalise on the notion of D-structure as a language independent syntactic representation, as it is intended by the linguistic theory. In so doing, the translation component avoids the problems which arise in dealing with seemingly idiosyncratic surface phenomena which may exist between the source and target languages.

The translation component consists of three modules. The first recovers D-structure from the S-structure representation produced by the syntactic analysis component. The second translates the D-structure of the source language into that of the target language, and the third generates the S-structure representation for the target language. This model of translation is basically identical to that employed by [Sharp 85], and [Dorr 87] for translation between English and Spanish, and is illustrated in Figure 5.1.

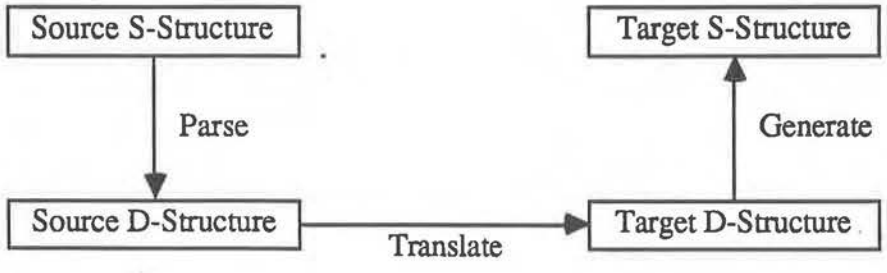

Figure 5.1: The Translation Model

It is important to note that the present system performs only *syntactic* translation. That is, it assumes lexical items in each language have essentially an injective mapping. The system does

not attempt to achieve the breadth of coverage of existing machine translation (MT) systems (see [Slocum 85] for a survey of existing systems). Rather, its purpose is to show how a principle-based approach can greatly simplify the task, by performing translation at D-structure.

Each of the modules is called in sequence by the *translate* predicate specified as follows:

(99)
```
translate(SourceTree,TargetTree) :-
        write('Source Surface Structure:'),nl,
        pretty(SourceTree),nl,
        gen_deep_structure(SourceTree,SourceDeep),
        trans_lexical(SourceDeep,TargetDeep,Mode),
        gen_surf_structure(Mode,TargetDeep,TargetTree).
```

The remainder of this chapter will discuss the implementation of each module.

5.1 Recovering D-structure

The recovery of D-structure from S-structure is a relatively straightforward task. Essentially it involves "undoing", or reversing the move-α transformations which are reflected by S-structure. Note, there is no need to employ any constraints at this stage, since the chains have already been verified as well-formed during syntactic analysis (e.g. the Case filter and θ-criterion have been satisfied). The predicate *gen_deep_structure* is therefore rather trivial, calling only one important goal, *rev_move_alpha*. Both are specified below:

(100)
```
gen_deep_structure(SourceTree,TargetTree) :-
        rev_move_alpha(SourceTree,TargetTree),!,
        write('Source Deep Structure:'),nl,
        pretty(TargetTree),nl.
rev_move_alpha(S_structure,D_structure) :-
        S_structure = xmax(C,ID,F,Cons)/_,
        make_trace_lists(1,Cons,Lists),
        rev_move_np(S_structure,D_structure1,Lists),
        rev_move_head(D_structure1,D_structure),!.
```

The *rev_move_alpha* predicate divides the reverse transformations into two categories; the first are moved $\overline{\overline{X}}$ constituents, namely NP's and wh-phrases, and the second are moved heads, such as verbs raising to I, and inversion to C. These reverse transformations are performed by *rev_move_np* and *rev_move_head* respectively (these are not order dependent).

The process of moving $\overline{\overline{X}}$'s to their D-structure positions is relatively straightforward. It essentially involves "collapsing" the chains which are recovered during syntactic analysis. Consider the specification of *rev_move_np* shown below:

(101)
```
rev_move_np(D_str,D_str,[]) :- !.
rev_move_np(S_str,D_str,[Chain|Rest]) :-
        extract_from_chain(Chain,SurfID,BaseID),
        collapse_chain(S_str,SurfID,D_str1,BaseID),
        rev_move_np(D_str1,D_str,Rest),!.
```

The first clause halts when the list of chains, the third argument, has been exhausted. The main clause calls *extract_from_chain* with the first chain in the list. This predicate simply returns the surface position of the moved constituent (i.e. the head of the chain) in *SurfID*, and the base position, or tail, in *BaseID*. These two positions, along with the S-structure representation, are passed to *collapse_chain* which returns a new representation, *D_str1*, in which the constituent has been returned to its base position. The predicate is then called recursively until all chains have been collapsed.

In the present system, head-movement is not recovered through the use of chains, since movement is always to the next highest X^0 position (possibly successively). As a result, undoing head movement is also relatively straightforward. The task here is to first find a moved head, and then return it to its base position. This is performed by *rev_move_head* which is written as follows:

(102)
```
rev_move_head(OldDstr,NewDstr) :-
        find_moved_head(OldDstr,HD,OldDstr1),!,
        move_hd_base(HD,OldDstr1,NewDstr).
rev_move_head(Dstr,Dstr).
```

The first goal, *find_moved_head*, takes an existing tree representation for the clause (*OldDstr*), and looks for situations in which a head does not match the category of the phrase. If it finds such an occurrence, it returns the head as *HD*, and sets the head of the phrase to *empty*. The second goal then searches for the D-structure position of the head and sets it to be *HD*. In fact, the only case this must deal with is inversion of V or I to C, since the parser accounts for raising of V to I automatically[1].

5.2 Translation

The translation module, as stated earlier, is highly simplified and performs only syntactic translation. The highest level, shown in (103), simply calls *trans_lex* to translate the lexical items of the source language into those of the target language.

[1] That is, the $\overline{X}$-parser will parse a V which has raised to I in its D-structure, V, position. This is relatively straightforward since these positions are generally string adjacent in both English and German, however a more principled approach using chains would doubtlessly be preferable.

(103)

```
trans_lexical(SourceDeep,TargetDeep,Mode) :-
        trans_lex(SourceDeep,TargetDeep,Mode),
        write('Target Deep Structure:'),nl,
        pretty(TargetDeep),nl.

trans_lex(SourceDeep,TargetDeep,Mode) :-
        SourceDeep = xmax(C,ID,F,Cons)/R,!,
        SourceDeep2 = xmax(C,ID,F,_)/R,
        trans_lex1(SourceDeep2,TargetDeep,Mode),
        target(L),!,
        rev_constraints(L,C,TargetDeep).
trans_lex(SourceDeep,TargetDeep,Mode) :-
        trans_lex1(SourceDeep,TargetDeep,Mode).
```

The *trans_lex* predicate simply recurses through the D-structure of the source language, calling *trans_lex1* at each maximal projection. The *trans_lex1* predicate (see Appendix E) translates each lexical item of the phrase, and in turn calls *trans_lex* for each "embedded" maximal projection. The result is essentially a top-down, left-to-right translation of the sentence. During translation, the *Mode* variable also becomes instantiated. This indicates if the sentence is interrogative ("ques") or declarative ("decl"), and is used for S-structure generation, discussed in the next section.

In addition to translation, the *trans_lex* predicate also calls *rev_constraints* at each maximal projection. The purpose of this is to determine which constituents of the target D-structure are going to have to move during the generation of S-structure. The result is the construction of chains which will be "unfolded" during generation[2]. The predicate is shown below:

(104)

```
rev_constraints(L,C,Tree) :-
        get_constraints(Tree,Constraints),
        subjacency(L,C,Tree,Constraints,NewConstraints),
        rev_add_constraints(Tree,NewConstraints).
```

This predicate essentially applies "in reverse" the constraint module used for parsing[3]. Note that while the translation is performed top-down, the application of constraints is done bottom-up, as the call appears at the end of the clause. In a complete system, the application of

[2] This should probably be performed in the generation component, but was included here since the D-structure is being traversed anyway, for translation, and the application of the constraint module does not interfere.

[3] In fact, if we want to remain true to the model of transformational grammar, the syntactic analysis applies the principles in reverse, while generation applies them in the intended manner.

principles would begin from scratch, applying Case theory to determine those constituents which must move, and ensuring that none of the principles are violated during the transformation phase. Here however, we take advantage of the similar Case marking properties of English and German. That is, if a constituent is returned to a non Case marked position in the recovery of English D-structure, then that position is not Case marked in the German D-structure (and *vice versa*). In general, structural similarities between English and German make verification of the many principles non-critical. Therefore, for our purposes we need only apply the Subjacency module. This has been implemented such that it can be applied in reverse. That is, by instantiating the D-structure argument, it will construct the chains which are to be reflected in the corresponding S-structure. The *get_constraints* predicate is as described in Chapter 4, and simply merges the constraint lists of immediately dominated maximal projections. The *rev_add_constraints* is the reverse analog of the *add_constraints* predicate used in parsing. That is, it adds requests to the constraint list of each $\overline{\overline{X}}$ for consideration by the constraint module. The relevant code is shown below:

(105)
```
rev_add_constraints(xmax(C,ID,F,NewCons)/R,Cons) :-
        rev_new_constraints(C,xmax(C,ID,F,Cons)/R,Cons1),
        append(Cons,Cons1,NewCons).
rev_new_constraints(_,xmax(C,ID,_,_)/e_cat(Type),[Cons]) :- !,
        ( (Type = ant , Cons = ant(abar,_,ID)) ;
          (Type = comp , Cons = ec(_,ID,trace(comp))) ;
          (Type = np , Cons = ant(a,_,ID)) ).
rev_new_constraints(_,xmax(C,ID,_,_)/e_cat(_,Type),Cons) :- !,
        ( (Type = trace(comp) , Cons = [ec(_,ID,trace(comp))]);
          Cons = [] ).
rev_new_constraints(_,xmax(C,ID,Ftr,_)/R,[ec(C,ID,trace(wh))]) :-
        member(wh(+),Ftr),!.
rev_new_constraints(n(_),xmax(C,ID,Ftr,_)/R,[ec(C,ID,trace(np))]) :-
        \+ member(case(_),Ftr),!.
rev_new_constraints(_,Tree,[]).
```

The purpose of this predicate, is basically to determine possible intermediate and destination positions for moved constituents. When chains are collapsed in recovering D-structure, the vacated S-structure positions (i.e. SPEC,CP and SPEC,IP positions) are filled by appropriate empty phrase markers. These are interpreted by the second clause of (105). Specifically, an empty subject position may be a destination for the head of an A-Chain, a matrix SPEC,CP may be a landing site for a wh-phrase, and an embedded SPEC,CP may be an intermediate position (since indirect questions are not handled). The fourth and fifth clauses find non Case marked NP's and wh-phrases, which add the *ec* request to the constraint list. This has the effect of requesting that these positions be vacated.

5.3 Generating S-structures

In this section, we present the model for generating S-structures from D-structures in the target language. This involves doing all the necessary transformations, and ensuring that agreement is reflected by the morphology of relevant elements. The generation module is controlled by the following predicate, *gen_surf_structure*:

(106)
```
gen_surf_structure(Mode,TargetDeep,TargetSurface) :-
        target(L),set_inversion(L,Mode,Inv),
        move_alpha(TargetDeep,TargetSurface1),
        raising(L,Inv,TargetSurface1,TargetSurface2),
        gen_pf(TargetSurface2,TargetSurface3),
        inversion(Inv,TargetSurface3,TargetSurface4),
        topicalize(L,Mode,TargetSurface4,TargetSurface),
        write('Target Surface Structure:'),nl,
        pretty(TargetSurface).
```

The clause is passed the D-structure representation, *TargetDeep*, and the mode of the sentence, *Mode*. When completed, the variable *TargetSurface* will be instantiated with the S-structure representation of the sentence in the target language. The first call simply instantiates L as the target language (either "english" or "german"), while the second sets the inversion flag, *Inv*, based on L and *Mode* using the parameters[4] below:

(107)
```
set_inversion(ger,_,yes).
set_inversion(eng,ques,yes).
set_inversion(eng,decl,no).
```

These parameters indicate that inversion always takes place in German (thus accounting for "verb-second" phenomena), and inversion is performed in English questions.

The next operation is to apply the *move_alpha* predicate which uses the chains constructed by *translate* to move the various constituents. The Prolog is shown below:

(108)
```
move_alpha(Deep,Surface) :-
        Deep = xmax(C,ID,F,Cons)/_,
        make_trace_lists(1,Cons,Lists),
        move_np(Surface,Deep,Lists).
```

[4] In fact, there exist more principled accounts of the parametric variation which determines inversion. For further discussion see [Davis 87].

This first two goals examine the constraint list and extract relevant information from the chains. The third goal, *move_np*, is responsible for actually moving the constituents, and is specified as follows:

(109)
```
move_np(D_str,D_str,[]) :- !.
move_np(S_str,D_str,[Chain|Rest]) :-
        extract_from_chain(Chain,SurfID,BaseID),
        collapse_chain(D_str1,SurfID,D_str,BaseID),
        move_np(S_str,D_str1,Rest),!.
```

This predicate works much as the *rev_move_np* predicate described in the previous section. That is, it determines the surface and base position for the constituent of each chain and then calls *collapse_chain*. Here however, the D-structure argument of *collapse_chain* is instantiated, and the S-structure representation is generated.

The next task is to perform verb raising. As noted in Section 5.1, the analysis performed here is somewhat tailored to the English and German. We take raising to be an obligatory transformation where the "highest" verb of a clause moves to the head of the dominating IP, just incase I is empty. In this way, the first verbal element is inflected by the AGR features present in the I node. A difference does exist between English and German however, in that only English auxilliaries (e.g. *have* and *be*) may raise, if inversion to C is to take place, while all German verbs raise, regardless of inversion[5]. As a result, this routine must be aware of whether or not inversion is to take place, since for English, "do-support" may be required in cases where no auxilliary is present. The main predicate for verb raising is as follows:

(110)
```
raising(Lang,Inv,DeepTree,SurfTree) :-
        DeepTree = xmax(i(Tns),ID,Ftr,Cons)/[Left,Right],!,
        raising(Lang,no,Left,NewL),raising(Lang,no,Right,NewR),
        DeepTree1 = xmax(i(Tns),ID,Ftr,Cons)/[NewL,NewR],
        check_subject(Left,Inv,NewInv),
        raise1(Lang,NewInv,ID,DeepTree1,SurfTree).
raising(Lang,Inv,X/[L,R],X/[NewL,NewR]) :- !,
        raising(Lang,Inv,L,NewL),raising(Lang,Inv,R,NewR).
raising(Lang,Inv,X/R,X/NewR) :- !,raising(Lang,Inv,R,NewR).
raising(Lang,Inv,X,X) :- !.
```

Basically, this traverses the tree, until the matrix IP clause is found. Then *raising* is applied to the subtrees of the clause with the inversion argument set to "no", since inversion may only

[5] For a more principled account of raising, see [Koopman 84] in which the ECP is used to account for these phenomena.

occur in the matrix clause. The subject of the IP is then checked to make sure it's not empty. This may revise the inversion flag, since an empty subject blocks the inversion transformation. Then, the *raise1* predicate is called, and is shown below:

(111)
```
raise1(Lang,Inv,ID,DeepInfl,SurfInfl) :-
        get_head(DeepInfl,head(i(_),empty,_)),!,
        raise_verb(Lang,Inv,ID,DeepInfl,DeepInfl1,Verb),
        set_head(Verb,ID,DeepInfl1,SurfInfl).
raise1(ger,Inv,ID,DeepInfl,SurfInfl) :-
        get_head(DeepInfl,head(i(tns(-)),zu,_)),!,
        raise_verb(ger,Inv,ID,DeepInfl,DeepInfl1,Verb),
        DeepInfl1 = Xmax/R,
        SurfInfl = Xmax/[xmax(i(tns(-)))/R,Verb].
raise1(L,Inv,ID,Infl,Infl) :- !.
```

The first clause checks to see if I is empty. If so, it calls *raise_verb* which retrieves the head of the subcategorized VP (and sets its head to empty). The *set_head* routine is then called to instantiate the head of the IP with the verb. The second clause is not really a raising case, but rather handles the adjunction of the highest V to the right of I in German infinitival clauses. Again, *raise_verb* is called to find the verb, and then the adjunction is performed. The final case will trivially succeed if I is lexical.

At this point, it is convenient to apply the agreement routines to ensure that person, number, gender, Case and tense are realised appropriately for the various phrases and lexical items. This is performed by the *gen_pf* routine, specified as follows:

(112)
```
gen_pf(SourceDeep,TargetDeep) :-
        SourceDeep = xmax(C,ID,F,Cons)/R,
        member(C,[n(_),c(emb/rel)]),!,
        target(L),
        rev_agree(L,SourceDeep,TargetDeep1),
        gen_pf1(TargetDeep1,TargetDeep).
gen_pf(SourceDeep,TargetDeep) :-
        SourceDeep = xmax(C,ID,F,Cons)/R,!,
        gen_pf1(SourceDeep,TargetDeep1),
        target(L),!,
        rev_agree(L,TargetDeep1,TargetDeep).
gen_pf(SourceDeep,TargetDeep) :-
        gen_pf1(SourceDeep,TargetDeep).
```

This predicate traverses the tree, applying *rev_agree* at each maximal projection. This task is basically performed in a bottom-up fashion by the second clause. The first clause however applies *rev_agree* first in the case of NP's and relative clause CP's since the operator of the CP must agree with the dominating NP, and this may further be used to inflect the embedded clause. The *rev_agree* predicate simply calls the *agree* routine which is used during syntactic analysis. In this case however, the D-structure, uninflected representation is instantiated, and the inflected S-structure is generated.

Once the various agreement inflections have been performed, the inversion transformation is performed, using the *inversion* predicate specified below:

(113)
```
inversion(yes,DeepTree,SurfTree) :- !,
        find_infl(DeepTree,SurfTree1,Infl),
        SurfTree1 = xmax(_,ID,_,_)/_,
        set_head(Infl,ID,SurfTree1,SurfTree).
inversion(no,Tree,Tree) :- !.
```

The two clauses handle the case where the inversion flag is set to "yes" or "no" respectively. In the latter case, the predicate trivially succeeds. In the case where inversion is to take place, the *find_infl* routine first locates the matrix IP, and retrieves its head (setting it to empty). Then the head of the matrix CP is instantiated, by the I (or possibly raised V) element.

The final transformation is that for topicalisation in German. This is not performed by *move_alpha*, since it is done independently of the standard wh or Case motivated transformations. Specifically, if the SPEC,CP of the matrix CP is empty, then some constituent must move to that position. The *topicalize* routine is shown below:

(114)
```
topicalize(ger,decl,DeepTree,SurfTree) :- !,
        get_topic(DeepTree,Tree,Topic),
        attach_topic(Topic,Tree,SurfTree).
topicalize(_,_,Tree,Tree) :- !.
```

The first goal examines the current tree and returns an appropriate phrase to topicalise. In this system, the highest phrase dominated by the matrix IP is selected (i.e. the subject), although this routine could be altered to use a more sophisticated selection strategy depending on emphasis and possibly syntactic restrictions. The final goal simply attaches the phrase to the topic, SPEC,CP, position.

Chapter 6

Evaluation and Discussion

The preceding chapters have presented a system for natural language analysis and translation which is based upon the principles of transformational generative grammar. Specifically, we have presented the linguistic framework of Government-Binding theory, a system of representations, and an implementation based upon the linguistic principles. The discussion thus far has been primarily descriptive, focusing on the exposition of the theory, representations, and implementation.

This chapter is devoted to an evaluation and discussion of the present system. We begin with a discussion of some general issues central to the construction of principle-based systems. We will then turn to an evaluation of each component of the system with respect to our general design criteria and compare ours with other principle-based systems. Finally, we discuss certain related issues which may prove relevant in future systems.

6.1 Principle-Based Systems

Thus far we have appealed largely to intuition in use of the term "principle-based" system. We have presented a linguistic theory which is founded upon the notion of a set of language independent principles, in which the grammar for a specific language is characterised by instantiating parameters of variation. This model has been proposed as a theory of linguistic competence. That is, it attempts to capture human's innate knowledge of language.

In designing a principle-based parser, the basic goal is to construct a system which determines syntactic well-formedness through the application of the principles of grammar. The linguistic theory is largely declarative in nature. That is, it consists of principles which act as conditions on representations. From a parsing perspective this presents a problem: how are these "representations" to be constructed in the first place? In solving this problem it is necessary to assign some procedural interpretation to the declarative principles of the theory. That is, the principles must not act only as conditions on representation, but must also contribute to the construction of those representations.

The degree to which principles of grammar are used procedurally or declaratively can be a significant factor in determining the organisation and efficiency of the system. The present system uses the principles of $\overline{X}$-theory, and θ-theory combined with lexical selection to construct phrase structure representations. Principles are applied to partially constructed structures to determine their "local" well-formedness. This organisation dictates the existence of certain extra machinery, namely the constraint lists, to allow for the incremental application of the principles. This contrasts with Sharp's system, in which the principles of grammar are stated purely as conditions on possible S-structure representations. The advantages of the more "procedural" approach taken here, and in Dorr's system, are reflected by the improved efficiency.

An interesting alternative to using $\overline{X}$-theory to drive parsing, is the *licensing* approach taken by Abney and Cole [Abney *et al.* 86]. Their parser, implemented within the *Actors* computational paradigm, capitalises on the licensing relationships which exist between lexical elements and constituents. Specifically, Case and θ theory are used directly to determine well-formedness and construct representations.

In the present work we are not concerned simply with the development of a principle-based parser, but rather the development of a parser with cross-linguistic application. In designing such a system, the necessity of maintaining the modularity and autonomy of the various subtheories becomes readily apparent. That is, parsers where the principles of grammar are somehow embedded directly in the architecture, such as the Marcus parser [Marcus 80], seem unlikely candidates for cross-linguistic systems. Rather, it seems that a modular approach, reflecting the linguistic model, should be employed.

The ultimate metric with which to evaluate the system is that of linguistic fidelity. That is, how faithful is the system to the linguistic theory. The ideal system should reproduce precisely the same grammaticality judgements as the linguistic theory. The closer the parser design is to the linguistic model, the more likely it is to be correct.

6.2 The Lexicon

The shift within linguistic theory from systems of rules to systems of principles has led to increased emphasis on the lexicon. That is, properties of lexical items are projected from the lexicon and interact with the principles of grammar in determining well-formed utterances. The most notable examples of this are the subcategorization and θ-marking properties of lexical items. As we have seen, lexical items may select for certain constituents so as to ensure that semantic roles are appropriately filled. Furthermore, the possible categories for a selected constituent are generally constrained in some way.

In the present system, θ-marking properties are not expressed in detail. That is, constituents are not assigned explicit θ-roles such as *Agent*, *Patient* or *Theme*. Rather, they are simply marked as receiving "some" θ-role. Subcategorization frames are specified, indicating the categories of the selected complements, and by the Projection Principle they are marked as θ-positions. Additionally, a lexical item (typically a verb) may indicate that it assigns an external θ-role to

the subject position[1].

While this approach to θ-marking is sufficient for enforcing the θ-criterion and the Projection Principle, there are clearly some problems which arise. In the first place, it may be necessary to specify a number of subcategorization frames, indicating different phrasal categories for the same selected θ-role. An example taken from [Chomsky 86b] illustrates how the verb *asks* semantically selects (or, *s-selects*) for *Proposition* complement, but categorially selects (or, *c-selects*) for either an NP or clause, as follows:

(115) (a) I asked [$_{NP}$ the time] .
(b) I asked [$_{CP}$ what time it was] .

In an attempt to eliminate redundancy in specifying s-selection and c-selection, Chomsky proposes that s-selection of some "semantic category" C entails c-selection of a syntactic category which is the "canonical structural representation" of C (i.e. CSR(C)). That is, if a verb s-selects for a Proposition, it c-selects CSR(Proposition), which may be either NP or clause (i.e. CP or IP) [2].

For purposes of translation it seems almost certain that a richer θ-marking system will prove necessary. Especially in more diverse languages where similar θ-roles assigned by corresponding lexical items may be assigned to different structural positions. While the present system simply matches structural positions, a more complete system should match θ-roles. Consider for example the following:

(116) (a) Er gefällt mir.
(b) I like him.

In this example, the Agent-role is assigned to the object position by *gefallen* and the subject by *like*, while the Patient-role is assigned to the subject and object position of these two verbs respectively[3].

6.3 Syntactic Analysis

The syntactic analysis component accepts an input sentence and recovers an annotated S-structure representation. That is, it recovers both phrase structure and chains. In the present system, this

[1] In the present system we assume all subjects are NP's, thus excluding the possibility of sentential subjects. Extending the system to include sentential subject would not however present any problem.

[2] There are instances where a verb s-selects for a Proposition which may only be a clause, such as *wonder*. That is, *I wondered what time it was* is grammatical, but *I wondered the time* is not. Case theory provides a reasonable explanation by suggesting that *wonder* but not *ask* is intransitive, resulting in a violation of the Case Filter in the latter sentence [Pesetsky 83].

[3] A literal translation might equate *gefallen* with *please*, in which case *He pleased me* would have similar structural θ-marking properties and thus present no problem.

component is comprised of two central modules: the *Parsing Module* and the *Constraint Module*. The Parsing Module uses $\overline{X}$-theory and the Extended Projection Principle to generate possible phrase structures and predict empty categories. During parsing it projects subcategorization information from the lexicon, and accesses language specific information such as head position for various categories, adjunct possibilities, and possible destinations for moved constituents. The Constraint Module is invoked as each maximal projection is completed. It incorporates Case theory, the ECP, θ theory, and Subjacency and performs the dual task of constructing chains and verifying the well-formedness of S-structures.

6.3.1 The Parsing Module

The Parsing Module employs a DCG specification of the $\overline{X}$ metarules to drive syntactic analysis. The result is a top-down, left-to-right, backtracking parser, based on Prolog's underlying theorem prover. The decision to use logic grammars was based primarily on their convenience and perspicuity. That is, $\overline{X}$ rules are specified directly, with logical variables being instantiated with categorial and other phrase specific information during the parsing. In addition, the top-down parsing strategy facilitates the relatively straightforward "prediction" of empty elements[4].

This parsing strategy does however have certain drawbacks. Specifically, a top-down approach leads to certain fundamental problems in a model where information relevant to parsing is projected from lexical items. The most obvious example of this is subcategorization information. In the present system, when a head is encountered its subcategorization frame is accessed and used to parse its arguments. The advantage is the prediction of traces in argument positions (i.e. if the argument isn't present, it must have moved and left a trace). A disadvantage is that a lookahead mechanism is required in the case of head final phrases, such that the head's subcategorization frame can be accessed to parse the pre-arguments.

The obvious solution to this is to use a bottom-up strategy where phrases are projected only as evidence for them appears in the input. Indeed, we have already observed that the constraints are intended to apply in a bottom-up manner. In such a system, some reduction procedure would be called upon to attach any arguments once the verb is encountered. Note, this doesn't entail the exclusion of logic grammars, but simply requires that some alternate parsing strategy be employed[5].

A number of existing systems employ bottom-up parsing strategies. Berwick and Weinberg present a modified version of Marcus' deterministic LR(k) parser [Marcus 80], [Berwick *et al.* 84]. These parsers use finitely bounded lookahead to compute their derivations without backtracking. The fundamental disadvantage of this approach is that the control table is language specific, in effect pre-computing the possible surface phenomena instead of directly consulting the principles of grammar as structures are computed. The cross-linguistic capability of their approach seems

[4] This is to say that no extra machinery is required, as is typically the case with bottom-up approaches.

[5] For further discussion of alternative parsing strategies for logic grammars see [Pereira *et al.* 87], [Abramson *et al.* 88].

questionable, since the tables must be completely re-calculated for a different language[6]. As Barton suggests, one approach might be to introduce principles and parameters to the Marcus framework, thus reducing the tables of rules [Barton 84]. In fact, Kuhns' system aims at doing precisely this in his Prolog implementation of a principle-based parser which augments a Marcus parser with Binding and Control theory, and chains which are used to enforce the θ-criterion [Kuhns 86].

Another interesting approach is the *assertion set* parser presented in [Barton *et al.* 85] and [Berwick *et al.* 85]. This system uses phrase markers to parse input with "information monotonicity", and without lookahead. Unfortunately, not enough is known about this technique to determine its cross-linguistic abilities, especially in head final languages.

Finally, we might pursue the use of more traditional, non-deterministic, parsing strategies. Dorr for example uses an Earley algorithm to parse a slight expanded $\overline{X}$ rule template [Dorr 87]. In her system, the parser is co-routined with the principles so as to block the derivation of ungrammatical structures, in a manner roughly similar to the system developed here. Pereira gives an interesting account of how a shift-reduce parser may combined with an oracle for the purpose of resolving attachment preferences when shift/reduce conflicts occur [Pereira 85]. This could provide an interesting method of incorporating various performance/processing principles directly into the parser[7].

6.3.2 The Constraint Module

The Constraint Module operates in tandem with the Parsing Module. Its purpose is to apply the principles of grammar to the partially constructed S-structures generated by the parser. These principles of grammar are used to construct chains of movement and verify the well-formedness of (sub-)structures with respect to their phrase structure and chains.

An attempt has been made in the current system to maintain the modularity of the subtheories of grammar. To some extent this has been accomplished, however the Subjacency constraint has amalgamated the tasks of constructing chains and verifying their well-formedness by incorporating elements of Case and θ theory. The design of future systems may benefit from a more modular approach which could prove more efficient, perspicuous, and easily modifiable.

While a relatively high degree of language independence has been achieved, some "short cuts" have been taken where English and German show similar characteristics. The most notable example of this is the treatment of head movement which is largely stipulated in the present system. More principled account of this phenomena in terms of chains do exist within the theory and should be reflected by future systems (see [Koopman 84]).

The present system applies constraints to completed phrases in a bottom-up fashion. This

---

[6] Indeed, it is not entirely clear that such a parser can handle a head final language such as German, using reasonably bounded lookahead.

[7] Pereira demonstrates his approach using a traditional phrase structure grammar. The suitability of his approach to an $\overline{X}$ system remains to be determined.

approach is especially convenient when combined with a bottom-up parsing strategy as in Dorr's system [Dorr 87]. It is conceivable, however, that the constraints could be formulated as conditions on left contexts. This is especially relevant to Prolog-based systems in which Horn clause theorem provers may be used to parser a logic grammar (as in the present system). As Stabler shows, it is possible to begin with a first-order logic specification of the linguistic constraints and rewrite them as Horn clauses, assuming negation as failure [Stabler 87]. Specifically, Stabler introduces a simplified set of linguistic constraints, intended to constrain the possible derivations of an underspecified DCG grammar. While the original formulation of the constraints assumes the existence of an entire proof/parse tree, Stabler shows that a series of program transformations can be invoked to produce constraints that can be applied at any point during the derivation. The constraints are formulated so as to use a "specialised" left context derivation tree, resulting in a much more efficient parser.

Admittedly, the feasibility Stabler's approach remains to be determined. Currently, it has only been applied to extremely simplified constraints, with much of the burden still being placed on a phrase structure grammar. Furthermore, the process of transforming the original linguistic constraints into their specialised left context form has only been partially automated. The goal, however, of creating a system which can transform some first-order specification of the linguistic constraints into and efficient Horn clause theorem prover for parsing is an attractive one.

6.4 Translation and Generation

In the present system, the translation and generation components have been somewhat simplified. The translation component maps D-structures from the source language to the target language by directly translating lexical items. Additionally, some structural changes are made to account for the head final position of V and I in German. Indeed, these configurational differences combined the possible thematic/structural divergences observed in Section 6.2 might lead us to question the suitability of D-structure as in "interlingual" representation for purposes of translation. As a solution to this, Dorr has suggested the use of a *lexical conceptual structure* (LCS) which represents sentence meaning through "predicate decomposition" [Dorr 88]. This would entail the development of a system to map the D-structures of a given language to and from an LCS representation. Such an approach certainly seems necessary in the development of translation systems for more diverse languages.

The generation component constructs S-structures from D-structures. That is, it applies the rule Move-α, subject to the constraints imposed by the principles of grammar, and their parameters. The present system takes advantage of certain structural similarities between English and German in performing the generation. Specifically, it assumes similar Case-marking properties, and landing sites for movement. The result is that only a single S-structure is generated, and it corresponds quite closely to that of the original source language sentence. An alternative, and likely preferable, approach would be to generate possible S-structures "from scratch". That is, begin with a bare D-structure and apply each of the principles, so as to force and constrain

possible movement. This strategy was employed in Sharp's system, which will generate a set of possible surface structures, which vary with respect to moved constituents and inflection of embedded clauses [Sharp 85].

As mentioned earlier, this approach to translation does not displace a knowledge-based approach, but rather compliments it. By performing translation at D-structure, or some other interlingual form, the task of dealing with idiosyncratic surface phenomena is side-stepped. This constrasts with the surface-to-surface form approaches, such as that employed by McCord [McCord 86]. The use of a surface form as the basis for translation entails that the transfer component perform complex restructuring of the surface structure in addition to translation. The result of McCord's approach is a relatively large set of language dependent rules, which only apply uni-directionally.

6.5 Related Issues

In addition to the points made above, there remain a numbers of areas open for possible improvement and extension of the system. In this section we will discuss two of these. The first concerns the possibility of improving the performance of the system through compilation techniques, while the second discusses the possibility of incorporating principles of human language performance.

6.5.1 Partial Evaluation

The system as presented here uses the principles of grammar in an on-line fashion. That is, they are consulted during the parsing process. Furthermore, the principles of grammar access their respective parameter values in a similar on-line manner.

This approach has a number of advantages in terms of maintaining the modular, autonomous nature of the principles and their cross-linguistic applicability. The efficiency, however, is hindered by the amount of computation which must be performed at parse time. An obvious solution to this is to investigate possible techniques for "pre-computing" or compiling certain aspects or components of the parser. Within a logic programming framework, techniques of *partial evaluation* are of particular interest.

Partial evaluation is the automatic derivation of a specialised instance of a program[8]. Consider, for example, a function which computes x^y, with x, y as parameters. This function could be partially evaluated with respect to $y = 3$ to produce the specialised cubic function x^3 [Ershov 82]. The partial evaluation of logic programs is facilitated by the ease with which meta-interpreters can be developed to evaluate, and possibly resolve, Prolog clauses with respect given input values. Additionally, a user must specify control information to limit the extent to which the evaluation

[8] For a detailed discussion of partial evaluation and Prolog see [Pereira *et al.* 87]. For a more theoretical discussion of the soundness and completeness of partial evaluation in Prolog see [Lloyd *et al.* 87].

is performed[9].

The possibility of partial evaluating the principles of the constraints module with respect to language specific parameters is both computationally and theoretically attractive. That is, it could be viewed as a core grammar generator, computing a specialised, efficient, language specific constraint module. If the control information can be specified independent of a set of language parameters, then an entirely automatic partial evaluator can be constructed for this task. The only major disadvantage is that a constraint module is necessary for each language in the system (although certain non-evaluated components might still be shared).

A similar partial evaluation technique could also be used with the parsing module to construct a (limited) set of phrase structure rules, based on the $\overline{X}$ metarules, language specific information about possible adjuncts and their positions, and the constraints. This process basically derives possible surface configurations ahead of time, to expedite the parsing of common structures using pre-compiled phrase structure rules. This would in some sense constitute return to the traditional phrase structure approach. Note, however, that the rules are constructed automatically, based on the principles of grammar, and the number of rules can be limited as much as desired – the constraint module will still be used to verify well-formedness at parse time.

6.5.2 Modeling Linguistic Performance

The present system consists of a procedural model of UG applied to syntactic analysis, translation, and generation. That is, the system models the principles of grammar, a theory of linguistic competence. A further area of interest is the construction of systems which model human linguistic performance. Specifically, these systems must reflect the basic principles of human language processing, and ideally possess similar organisation and employ similar algorithms.

Frazier, for example, draws on some intuitions supported by evidence from Dutch (a head final language similar to German) to suggest that principle-based systems should pre-compile the principles of $\overline{X}$, Case and θ theory to produce a limited set of phrase structure rules [Frazier 86]. The partial evaluation process discussed above would seem particularly relevant to such a theory of performance.

Alternatively, Pritchett suggests that the principles of grammatical theory be employed directly in a theory of language processing [Pritchett 88]. Specifically, he proposes a θ-Attachment principle and a θ-Reanalysis constraint, based on the θ-criterion, which accurately predict human processing of garden path sentences. Ultimately, he extends his theory to incorporate the entire theory of grammar by formulating the following principle:

(117) **Σ-Attachment:** Every principle of the syntax attempts to be satisfied at every point during processing.

---

[9] That is, if uncontrolled, a partial evaluator may attempt to expand or "compile-out" the original program beyond the point of any benefit.

Such a principle is compatible with the online, incremental application of principles performed by the present system and that of Dorr's.

In fact, the above two theories may not be entirely incompatible since some degree of pre-compilation would not necessarily conflict with Pritchett's Σ-Attachment principle. A more sophisticated discussion of these and other theories of performance is beyond the scope of this theory. We simply wish to make the point here that reconciling models of competence with those of performance appears to be both an interesting and promising area for future research[10].

[10] For further discussion see [Berwick 87].

Chapter 7

Conclusions

This thesis has presented a system for syntactic analysis and translation which is based upon the current theories of transformational generative grammar. Specifically, we have developed a system which employs the principles of Chomsky's Government-Binding theory in parsing and determining the well-formedness of sentences. As such, the system can be considered a procedural model of Universal Grammar, which accesses language specific information in analysing sentences of a particular language. Furthermore, we have shown that the cross-linguistic nature of the system lends itself particularly well to the task of language translation.

The principle-based approach to natural language analysis represents a significant departure from the traditional, construction-based systems. Specifically, the embodiment of universal principles of grammar drastically reduces the necessity of specifying language specific rules. Rather, the grammar for an individual language can be stated as a compact set of parameters and language specific information.

The current system accounts for a significant subset of German and English grammars. Specifically, a large set of substitution transformations are handled, including subject raising and $\overline{A}$-movement. The system will handle a variety of constructions, including embedded sentences, relative clauses, and adjunct PP's. Notable omissions of the system include passive and subjunctive forms, adjectival phrases, and adjunction transformations. Finally, the system has excluded the theories of Binding and Control and the PF and LF levels of representation which would be necessary for a complete implementation of the linguistic model.

The translation and generation components of the present system have been simplified in favour of the syntactic analysis component. The incremental application of the principles during parsing represents a significant improvement in overall efficiency relative to Sharp's system. Specifically, the use of constraint lists provides a convenient method for constructing chains and enforcing constraints. The modular nature of the system suggests that extending the coverage of the system, and introducing new languages should be relatively straightforward.

Principle-based approaches to the design of natural language systems are becoming increasingly popular. Most of these systems, however, are suited to the analysis of an individual language.

Specifically, only two multi-lingual systems have been previously constructed, namely those of Sharp and Dorr [Sharp 85], [Dorr 87]. Both of which handle English and Spanish. The present work has presented a system for the analysis of English and German which constitutes further evidence that the development of a parser with cross-linguistic application is indeed feasible. In addition we have shown that modularity and incremental constraint application are fundamental to the efficient design of such systems.

The construction of principle-based systems is relevant to a number of disciplines. As a model of the human language faculty, such systems are of direct interest in linguistics and cognitive science. Indeed, sophisticated systems may prove to be valuable testbeds for revisions or extensions of the theory. Additionally, principle-based approaches provide efficient, elegant, and robust techniques for natural language analysis, translation and generation within the fields of computational linguistics and artificial intelligence.

Bibliography

[Abney *et al.* 86] Steven Abney and Jennifer Cole. A Government-Binding Parser. unpublished manuscript, MIT, 1986.

[Abramson *et al.* 88] Harvey Abramson, Matthew Crocker, Brian Ross, and Doug Westcott. Towards a Logic Based Expert System for Compiler Development. In *PLILP'88 Proceedings*, Institut National de Recherche en Informatique et en Automatique, Orleans, France, 1988.

[Barton 84] G. Edward Barton. *Toward a Principle-Based Parser.* AI Memo 788, MIT AI Laboratory, Cambridge, Massachusetts, 1984.

[Barton *et al.* 85] G. Edward Barton and Robert C. Berwick. Parsing with Assertion Sets and Information Monotonicity. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 769–771, IJCAI, Los Angeles, 1985.

[Berwick 87] Robert C. Berwick. *Principle-Based Parsing.* Technical Report 972, MIT AI Laboratory, Cambridge, Massachusetts, June 1987.

[Berwick *et al.* 84] Robert C. Berwick and Amy S. Weinberg. *The Grammatical Basis of Linguistic Performance. Current Studies in Linguistics*, The MIT Press, Cambridge, Massachusetts, 1984.

[Berwick *et al.* 85] Robert C. Berwick and Amy S. Weinberg. *Deterministic Parsing and Linguistic Explanation.* AI Memo 836, MIT AI Laboratory, Cambridge, Massachusetts, June 1985.

[Chomsky 57] Noam Chomsky. *Syntactic Structures.* Mouton, The Hague, 1957.

[Chomsky 65] Noam Chomsky. *Aspects of the Theory of Syntax.* MIT Press, Cambridge, Massachusetts, 1965.

[Chomsky 73] Noam Chomsky. Conditions on Transformations. In S. R. Anderson and P. Kiparsky, editors, *A Festschrift for Morris Halle*, Holt, Rinehart and Winston, New York, 1973.

[Chomsky 81a] Noam Chomsky. *Lectures on Government and Binding.* Foris Publications, Dordecht, 1981.

[Chomsky 81b] Noam Chomsky. Principles and Parameters in Syntactic Theory. In Norbert Hornstein and David Lightfoot, editors, *Explanation in Linguistics*, chapter 2, pages 32–75, Longman, London, 1981.

[Chomsky 82] Noam Chomsky. *Some Concepts and Consequences of the Theory of Government and Binding. Linguistic Inquiry Monograph Six*, The MIT Press, Cambridge, Massachusetts, 1982.

[Chomsky 86a] Noam Chomsky. *Barriers. Linguistic Inquiry Monograph Thirteen*, The MIT Press, Cambridge, Massachusetts, 1986.

[Chomsky 86b] Noam Chomsky. *Knowledge of Language: Its Nature, Origin and Use. Convergence Series*, Praeger, New York, 1986.

[Clocksin *et al.* 81] W.F. Clocksin and C.S. Mellish. *Programming in Prolog.* Springer Verlag, 2nd edition, 1981.

[Colmerauer 78] Alain Colmerauer. Metamorphosis Grammars. In L. Bolc, editor, *Lecture Notes in Computer Science*, Springer Verlag, 1978.

[Dahl *et al.* 86] Veronica Dahl, Charles Brown, Michel Boyer, T. Pattabhiraman, and Diane Massam. *Mechanizing Expertise in GB Theory.* LCCR TR 86-10, LCCR, Simon Fraser University, Burnaby, B.C., Canada, 1986.

[Davis 87] Henry Davis. *The Acquisition of the English Auxilliary System and its Relation to Linguistic Theory.* PhD thesis, University of British Columbia, Vancouver, Canada, 1987.

[Dorr 87] Bonnie Dorr. *UNITRAN: A Principle-Based Approach to Machine Translation.* Master's thesis, MIT, Cambridge, Massachusetts, 1987.

[Dorr 88] Bonnie Dorr. *A Lexical Conceptual Approach to Generation for Machine Translation.* AI Memo 1015, MIT, Cambridge, Massachusetts, December 1988.

[Emonds 76] Joseph Emonds. *A Transformational Approach to English Syntax.* Academic Press, New York, 1976.

[Engdahl 83] Elisabet Engdahl. Parasitic Gaps. *Linguistics and Philosophy*, 6(1):5–34, 1983.

[Ershov 82] A. P. Ershov. Mixed Computation: Potential Applications and Problems for Study. *Theoretical Computer Science*, 18(1):41–67, 1982.

[Frazier 86] Lyn Frazier. Natural Classes in Language Processing. unpublished manuscript, MIT, 1986.

[Haider *et al.* 85] Hubert Haider and Martin Prinzhorn, editors. *Verb Second Phenomena in Germanic Languages. Publications in Language Sciences*, Foris, Dordecht, 1985.

[Hogger 84] Christopher J. Hogger. *Introduction to Logic Programming.* Volume 21 of *APIC Studies in Data Processing*, Academic Press, London, 1984.

[Hornstein *et al.* 81] Norbert Hornstein and David Lightfoot. Introduction. In Norbert Hornstein and David Lightfoot, editors, *Explanation in Linguistics*, chapter 1, pages 9–31, Longman, London, 1981.

[Huang 82] C.-T. James Huang. *Logical Relations on Chinese and the Theory of Grammar.* PhD thesis, MIT, Cambridge, Massachusetts, 1982.

[Kimball 73] John Kimball. Seven Principles of Surface Structure Parsing in Natural Language. *Cognition*, 2(1):15–47, 1973.

[Koopman 84] Hilda Koopman. *The Syntax of Verbs.* Foris, Dordecht, 1984.

[Kuhns 86] Robert J. Kuhns. A PROLOG Implementation of Government-Binding Theory. In *11th International Conference on Computational Linguistics*, pages 546–550, The International Committee on Computational Linguistics, Bonn, West Germany, August 1986.

[Lasnik *et al.* 88a] Howard Lasnik and Mamoru Saito. forthcoming, 1988.

[Lasnik *et al.* 88b] Howard Lasnik and Juan Uriagereka. *A Course in GB Syntax: Lectures on Binding and Empty Categories. Current Studies in Linguistics*, MIT Press, Cambridge, Massachusetts, 1988.

[Lightfoot 82] David Lightfoot. *The Language Lottery: Towards a Biology of Grammars.* MIT Press, Cambridge, Massachusetts, 1982.

[Lloyd 87] John W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, second edition, 1987.

[Lloyd *et al.* 87] John W. Lloyd and Joseph C. Shepherdson. *Partial Evaluation in Logic Programming.* Technical Report CS-87-09, University of Bristol, December 1987.

[Marantz 81] A. Marantz. *On the Nature of Grammatical Relations.* PhD thesis, MIT, Cambridge, Massachusetts, 1981.

[Marcus 80] Mitchell P. Marcus. *A Theory of Syntactic Recognition for Natural Language. The MIT Press Series in Artificial Intelligence*, The MIT Press, Cambridge, Massachusetts, 1980.

[Massam 85] Dianne Massam. *Case Theory and the Projection Principle.* PhD thesis, MIT, Cambridge, Massachusetts, 1985.

[McCord 86] Michael C. McCord. Design of a Prolog-Based Machine Translation System. In Ehud Shapiro, editor, *Third International Conference on Logic Programming*, pages 350–374, London, U.K., July 1986.

[Pereira 85] Fernando C. N. Pereira. A New Characterization of Attachment Preferences. In David R. Dowty, Lauri Karttunen, and Arnold M. Zwicky, editors, *Natural Language Parsing*, chapter 9, pages 307–319, Cambridge University Press, Cambridge, England, 1985.

[Pereira *et al.* 80] Fernando C.N. Pereira and D.H.D. Warren. Definite Clause Grammars for Language Analysis. *Artificial Intelligence*, 13:231–278, 1980.

[Pereira *et al.* 87] Fernando C.N. Pereira and Stuart M. Shieber. *Prolog and Natural-Language Analysis. CSLI Lecture Notes*, Center for the Study of Language and Information, Stanford, California, 1987.

[Pesetsky 83] D. Pesetsky. *Paths and Categories.* PhD thesis, MIT, Cambridge, Massachusetts, 1983.

[Pritchett 88] Brad Pritchett. Garden Path Phenomena and the Grammatical Basis of Language Processing. *Language*, September 1988.

[Pullum *et al.* 88] Geoffrey K. Pullum and Paul M. Postal. Expletive noun phrases in subcategorized positions. *Linguistic Inquiry*, 19(4), 1988.

[Rizzi 82] Luigi Rizzi. *Issues in Italian Syntax.* Foris, Dordrecht, 1982.

[Ross 67] John R. Ross. *Constraints on Variables in Syntax.* PhD thesis, MIT, Cambridge, Massachusetts, 1967.

[Sharp 85] Randall M. Sharp. *A Model of Grammar Based on Principles of Government and Binding.* Master's thesis, University of British Columbia, Vancouver, Canada, October 1985.

[Slocum 85] Jonathan Slocum. A Survey of Machine Translation: Its History, Current Status, and Future Prospects. *Computational Linguistics*, 11(1):1–17, 1985.

[Stabler 87] Edward P. Stabler. *Logic Formulations of Government-Binding Principles for Automatic Theorem Provers.* Cognitive Science Memo 30, University of Western Ontario, London, Ontario, June 1987.

[Sterling *et al.* 86] Leon Sterling and Ehud Shapiro. *The Art of Prolog. The MIT Press Series in Logic Programming*, The MIT Press, Cambridge, Massachusetts, 1986.

[Stowell 81] Timothy Stowell. *Origins of Phrase Structure.* PhD thesis, MIT, Cambridge, Massachusetts, 1981.

[Taraldsen 81] K.T. Taraldsen. The Theoretical Interpretation of a Class of Marked Extractions. In A. Belletti, L. Brandi, and L. Rizzi, editors, *Theory of Markedness in Generative Grammar, Proceedings of the 1979 Glow Conference*, pages 475–516, Scuola Normale Superiore, Pisa, 1981.

[Thiersch 78] Craig Thiersch. *Topics in German Syntax.* PhD thesis, MIT, Cambridge, Massachusetts, 1978.

[vRiemsdijk *et al.* 86] Henk van Riemsdijk and Edwin Williams. *Introduction to the Theory of Grammar. Current Studies in Linguistics*, The MIT Press, Cambridge, Massachusetts, 1986.

[Wexler *et al.* 80] Ken Wexler and Peter Culicover. *Formal Principles of Language Acquisition.* MIT Press, Cambridge, Massachusetts, 1980.

[Williams 74] Edwin Williams. *Rule Ordering and Syntax.* PhD thesis, MIT, Cambridge, Massachusetts, 1974.

Appendix A

Example Translations

This appendix illustrates the execution of the system on a number of example sentences from both English and German. The system was written in Quintus Prolog V2.2, under BSD UNIX 4.2 on a Sun 3 at the University of British Columbia. A complete listing of the Prolog source is presentented in the following appendices.

The source language is specified using the "language" command, followed by the language abbreviation (e.g. "language ger"). The target language is set automatically. Once the sentence is parsed, the S-structure form is displayed, with the various traces (i.e. np-#, wh-#, op-# and comp-#) shown coindexed with their antecedent. Additionally, PRO is shown. The D-structure of the source sentence is then displayed with the chains "collapsed" and head-movement undone. Finally, once translation has been performed, the D-structure and S-structure representations in the target language are printed. Note, capitalisation and accents are not handled by the present system.

```
(1) >>>language eng.
    >>>I have seen the book.
    Source Surface Structure: i have see the book.
    Source Deep Structure: i have see the book.
    Target Deep Structure: ich das buch sehen haben.
    Target Surface Structure: ich habe das buch gesehen.

(2) >>>Have the boys seen the girl?
    Source Surface Structure: have the boy see the girl?
    Source Deep Structure: the boy have see the girl?
    Target Deep Structure: das junge das madchen sehen haben?
    Target Surface Structure: haben die jungen das madchen gesehen?

(3) >>>What did the girl see on the table?
    Source Surface Structure: what-1 do the girl see wh-1 on the table?
```

```
Source Deep Structure: the girl see what-1 on the table?
Target Deep Structure: das madchen was-1 sehen auf das tisch?
Target Surface Structure: was-1 sah das madchen wh-1 auf den tisch?

(4) >>>What will the woman put on the table?
Source Surface Structure: what-1 will the woman put wh-1 on the table?
Source Deep Structure: the woman will put what-1 on the table?
Target Deep Structure: das frau was-1 auf das tisch legen werden?
Target Surface Structure: was-1 wird die frau wh-1 auf den tisch legen?

(5) >>>Have you seen the books that the girls put on the table?
Source Surface Structure: have you see the book op-1 that the girl put wh-1 on
the table?
Source Deep Structure: you have see the book that the girl put op-1 on the table?

Target Deep Structure: sie das buch das madchen das-1 auf das tisch legen sehen
haben?
Target Surface Structure: haben sie die buchen die-1 das madchen wh-1 auf den
tisch legte gesehen?

(6) >>>I believe that the girl tried to put the book on the table.
Source Surface Structure: i believe that the girl try PRO to put the book on
the table.
Source Deep Structure: i believe that the girl try PRO to put the book on the
table.
Target Deep Structure: ich dass das madchen dass PRO das buch auf das tisch legen
zu versuchen glauben.
Target Surface Structure: ich glaube dass das madchen PRO das buch auf den tisch
zu legen versuchte.

(7) >>>What do you believe the boy has put on the table?
Source Surface Structure: what-1 do you believe comp-1 the boy have put wh-1
on the table?
Source Deep Structure: you believe comp-1 the boy have put what-1 on the table?

Target Deep Structure: sie comp-1 dass das junge was-1 auf das tisch legen haben
glauben?
Target Surface Structure: was-1 glauben sie comp-1 dass der junge wh-1 auf den
tisch gelegt hat?

(8) >>>language ger.
```

```
>>>Ich habe das Buch gesehen.
Source Surface Structure: ich-1 haben wh-1 das buch sehen.
Source Deep Structure: ich-1 das buch sehen haben.
Target Deep Structure: i have see the book.
Target Surface Structure: i have seen the book.
```

(9)
```
>>>Haben Sie die Frau gesehen?
Source Surface Structure: haben sie das frau sehen?
Source Deep Structure: sie das frau sehen haben?
Target Deep Structure: you have see the woman?
Target Surface Structure: have you seen the woman?
```

(10)
```
>>>Was hat der Junge auf den Tisch gelegt?
Source Surface Structure: was-1 haben das junge wh-1 auf das tisch legen?
Source Deep Structure: das junge was-1 auf das tisch legen haben?
Target Deep Structure: the boy have put what-1 on the table?
Target Surface Structure: what-1 has the boy put wh-1 on the table?
```

(11)
```
>>>Ich glaube dass der Junge die Frau gesehen hat.
Source Surface Structure: ich-1 glauben wh-1 dass das junge das frau sehen haben.

Source Deep Structure: ich-1 dass das junge das frau sehen haben glauben.
Target Deep Structure: i believe that the boy have see the woman.
Target Surface Structure: i believe that the boy has seen the woman.
```

(12)
```
>>>Was glauben Sie dass der Junge auf den Tisch gelegt hat?
Source Surface Structure: was-1 glauben sie comp-1 dass das junge wh-1 auf das
tisch legen haben?
Source Deep Structure: sie comp-1 dass das junge was-1 auf das tisch legen haben
glauben?
Target Deep Structure: you believe comp-1 that the boy have put what-1 on the
table?
Target Surface Structure: what-1 do you believe comp-1 that the boy has put wh-1
on the table?
```

Appendix B

Parsing Module

```
%_____________________________________________________________
% Section: Parser
%
% Paradigm: parse(Language,Source,Tree).
%       Language:       Source language {english,german}.
%       Source:         Input sentence to be parse.
%       Tree:           The parse derivation tree.
%
% Description: This predicate controls parsing of the input
%       string. It first computes the lookahead stack of final
%       heads (bupdcg), and then calls the DCG rules which parse
%       the input according to the X-bar phrase structure rules.
%_____________________________________________________________

parse(L,String,Tree) :-
        bupdcg(L,String,XbarList,S),
        xmax(L,c(mat/sent),Tree,S,[],XbarList,[]).

%_____________________________________________________________
% bupdcg/4: The predicate constructs a lookahead stack of all final heads.
%       This stack is used during parsing to recover the subcategorization
%       frames of final heads. The second clause also "undoes" the adjunction
%       of a verb to the right of "zu".

bupdcg(L,[],[],[]) :- !.
bupdcg(ger,[zu|Rest],[HD|XRest],[Top,HD|Stack]) :-
```

```
        head_anal(ger,Cat,zu,HD,final),
        bupdcg(ger,Rest,XRest,[Top|Stack]).
bupdcg(L,[Word|Rest],[HD|XRest],[HD|Stack]) :-
        head_anal(L,Cat,Word,HD,final),
        bupdcg(L,Rest,XRest,Stack).
bupdcg(L,[Word|Rest],[Word|XRest],NS) :-
        head_anal(L,C,Word,HD,initial),
        bupdcg(L,Rest,XRest,NS),
        poss_move(L,Cat,C,_).
bupdcg(L,[LexItem|Rest],[LexItem|XRest],Stack) :-
        bupdcg(L,Rest,XRest,Stack).

%_____________________________________________________________
% head_anal/5: Call the morpher to determine if the word is in the lexicon.
%       Determine if the word is a phrasal head, what its category is,
%       it subcategorization frame, and its position (final/initial).

head_anal(L,Cat,Word,head(Cat,Word,As,Features),Pos) :-
        morph(L,Word,Cat,Root,Features),
        is_head(Cat),
        getargs(L,head(Cat,Word,As,Features)),
        head_position(L,Cat,Pos).

is_head(C) :- member(C,[n(_),v(_),p(_),c(_),i(_)]).

%_____________________________________________________________
% The DCG rules to parse according to X-bar theory.

%_____________________________________________________________
% xmax(L,C,Tree,S,NS):  parses the Specifier and Adjuncts of a phrase, and
%       also verifies the features of Tree match those subcategorized for.

xmax(L,C,XTree,S,NS)    -->   spec(L,C,TXmax,STree,S,S1),
                              xmax2(L,C,TXmax,S1,NS),
                              {agree(L,STree,BTree),gen_ID(BTree),
                               constraints(L,C,BTree,XTree)}.

xmax_e(L,C,Tree)        -->   [],
                              {empty_cat(L,C),
                              Tree1 = xmax(C,ID,[],[ec(C,ID,T)])/e_cat(C,T),
```

```
                                agree(L,Tree1,Tree),gen_ID(Tree)}.

xmax2(L,C,Tree,S,NS)    -->     adjunct(L,pre,C,TApost,Tree,S,S1),
                                xbar(L,C,TXbar,S1,S2),
                                adjunct(L,post,C,TXbar,TApost,S2,NS).

%_____________________________________________________________
% xbar(L,C,Tree,S,NS): parses the Arguments and Head of a phrase, depending
%       on the order (head initial/final parameter). In the case of a final
%       head, 'stack' is called to examine the lookahead list and retrieve
%       the subctegorization information.

xbar(L,C,Tree,S,NS)     -->     {head_position(L,C,initial),!},
                                xmin(L,C,Args,HD,S,S1),
                                {stack(L,initial,HD,S1,S2)},
                                arg(L,post,C,Args,HD,[],Tree,S2,NS).

xbar(L,C,Tree,S,NS)     -->     {head_position(L,C,final),
                                HD=X/head(C,_,_),
                                stack(_,final,head(C,W,_,F),S,R),!,
                                getargs(L,head(C,W,As,F))},
                                arg(L,pre,C,As,HD,[],Tree,R,S1),
                                xmin(L,C,As,HD,S1,NS).

xbar(L,C,Tree,S,NS)     -->     {head_position(L,C,final),!,
                                poss_empty(L,C,As),
                                HD=X/head(C,empty,F)},
                                arg(L,pre,C,As,HD,[],Tree,S,NS1),
                                xmin(L,C,As,HD,NS1,NS).

%_____________________________________________________________
% stack(L,Pos,Head,OldStack,NewStack): Maintains the lookahead of final
%       heads. This is used to retrieve their subcategorization frames.

stack(_,final,head(C,W,As,F),S,R) :-
        reverse(S,[head(C,W,As,F)|Rest]),
        reverse(Rest,R),!.
stack(eng,initial,xbar(c(_))/head(C,W,F),S,[head(C,W,As,F)|S]) :-
        \+ C = c(_),!.
stack(ger,initial,X/head(C,W,F),[head(C,W,As,F)|R],New) :-
```

```
        reverse(R,R1),reverse([head(C,W,As,F)|R1],New),!.
stack(_,initial,_,S,S) :- !.

%_____________________________________________________________
% xmin(L,C,Args,Tree,S,NS): parses the head of a phrase either as a lexical
%       head (which has possibly moved from its base generated position) or
%       as an empty head (if the category permits the head to be absent/moved).

xmin(eng,C,Args,Tree,[HD|S],S)    -->   [],
                                        {HD = head(C,W,Args,F),
                                        poss_move(eng,c(mat/sent),C,_),
                                        head_anal(eng,C,W,HD,initial),
                                        getargs(eng,head(C,W,Args,F)),
                                        Tree = xbar(C)/head(C,empty,_),!}.
xmin(L,C,Args,xbar(C)/Tree,S,S)   -->   head(L,C,Args,Tree).

head(L,C,As,head(C,W,F))          -->   [Word],
                                        {head_anal(L,C,Word,HD,initial),!,
                                        HD = head(C,W,_,F),
                                        getargs(L,head(C,W,As,F))}.
head(L,C,As,head(Cat,W,F))        -->   [Word],
                                        {head_anal(L,Cat,Word,HD,initial),
                                        HD = head(Cat,W,Args,F),
                                        poss_move(L,C,Cat,As),!}.
head(L,C,As,head(C,W,F))          -->   [head(C,W,_,F)],
                                        {!,getargs(L,head(C,W,As,F))}.
head(L,C,As,head(Cat,W,F))        -->   [head(Cat,W,Args,F)],
                                        {poss_move(L,C,Cat,As),!}.
head(L,C,As,head(C,empty,F))      -->   [],
                                        {poss_empty(L,C,As),!}.

%_____________________________________________________________
% The following are the general rule for Adjunct and Specifier productions.
%       These refer to the language specific choices for possible adjuncts
%       and specifiers.

spec(L,C,TX,xmax(C,ID,[],Cons)/[TSpec,TX],S,NS) -->   spec(L,C,TSpec,S,NS).
spec(L,C,X/R,xmax(C,ID,[],Cons)/R,S,S)          -->   [],{no_spec(L,CList),
                                                        member(C,CList)}.

no_spec(L,[n(_),v(_),p(_),c(mat/sent)]) :- !.
```

```
adjunct(ger,post,C,TX,xmax(C)/[TX,Tadj],S,S)      -->   {C = i(tns(-)),!},
                                                        [head(v(T),W,F)],
                                                        {Tadj = head(v(T),W,F)}.
adjunct(L,Pos,C,X/Tree,xmax(C)/Tree,S,S)          -->   [].
adjunct(L,pre,C,Tree,xmax(C)/[Tadj,Tree],S,NS)    -->   adj(L,pre,C,Tadj,S,NS).
adjunct(L,post,C,Tree,xmax(C)/[Tree,Tadj],S,NS)   -->   adj(L,post,C,Tadj,S,NS).

%_____________________________________________________________
% topic: parses a topic in the SPEC,CP position of a German root clause.

topic(L,c(mat/sent),Tree,S,NS)    -->   {empty_cat(L,C)},
                                        spec(L,C,TXmax,XTree,S,S1),
                                        xmax2(L,C,TXmax,S1,NS),
                                        {add_feature([ant(+),case(_)],XTree,Tr),
                                         agree(L,Tr,BTree),gen_ID(BTree),
                                         constraints(L,C,BTree,Tree)}.

%_____________________________________________________________
% wh_phrase: parses wh-phrases in the SPEC,CP position of root and relative
%       clauses.

wh_phrase(L,c(mat/_),Tree,S,NS)   -->   {empty_cat(L,C)},
                                        wh_xmax(L,Cat,C,Tree,S,NS).
wh_phrase(L,c(emb/T),Tree,S,S)    -->   [],{gensym(e,ID)},
                                        {set_ec(L,T,ID,Tree)}.

wh_xmax(L,Cat,C,XTree,S,NS)       -->   spec(L,C,TXmax,STree,S,S1),
                                        xbar(L,C,TXmax,S1,NS),
                                        {wh_agree(L,Cat,STree,BTree),
                                         gen_ID(BTree),
                                         constraints(L,C,BTree,XTree)}.

set_ec(_,sent,ID,
 xmax(C,ID,[agree(_,_,_,_)],[ec(C,ID,trace(comp))])/e_cat(C,trace(comp))).
set_ec(eng,rel,ID,
 xmax(n(_),ID,[agree(_,_,_,_),wh(+)],[ant(abar,n(_),ID)])/e_cat(n(_),operator)).

%_____________________________________________________________
% arg: uses the subcategorization frame of a head to parse its arguments. If
```

```
%       the argument is not present, a trace is inserted. 'a_xmax' parses
%       a lexical argument, 'a_xmax_e' parses a trace.

arg(L,Pos,C,[],HD,ALst,T,S,S)                   -->
                [],{build_tree(Pos,C,HD,ALst,T)}.
arg(L,Pos,C,As,HD,ALst,T,S,NS)                  -->
                {select(L,C,As,A,NewAs)},
                a_xmax(L,C,A,Tx,S,S1),
                arg(L,Pos,C,NewAs,HD,[Tx|ALst],T,S1,NS).
arg(L,Pos,C,[A|Args],HD,ALst,T,S,NS)            -->
                a_xmax_e(L,C,A,Tx),
                arg(L,Pos,C,Args,HD,[Tx|ALst],T,S,NS).
arg(L,Pos,C,[{A}|Args],HD,ALst,T,S,NS)          -->
                arg(L,Pos,C,Args,HD,ALst,T,S,NS).

a_xmax(L,Cat,C,XTree,S,NS)      -->     spec(L,C,TXmax,STree1,S,S1),
                                        xmax2(L,C,TXmax,S1,NS),
                                        {agree(L,STree1,BTree1),gen_ID(BTree1),
                                        BTree1 = xmax(_,ID,_,_)/Rest,
                                        l_mark(Cat,BTree1,BTree2),
                                        t_mark(Cat,ID,BTree2,BTree),
                                        constraints(L,C,BTree,XTree)}.

a_xmax_e(L,Cat,C,Tree)          -->     xmax_e(L,C,Tree1),
                                        {l_mark(Cat,Tree1,Tree2),
                                        t_mark(Cat,ID,Tree2,Tree)}.

%_____________________________________________________________
% poss_move(Language,Cat1,Cat2,Args): A head of Cat2 can move from its base
%       generated position to the head of Cat1. The default arguments of
%       Cat1 are Args.

poss_move(L,C,C,_).
poss_move(L,c(mat/sent),i(tns(+)),[i(tns(+))]).
poss_move(L,c(mat/sent),v(nil/tns(+)),[i(tns(+))]).

%_____________________________________________________________
% poss_empty(Language,Cat,Args): The head of Cat can be empty due to
%       absence/movement. The default arguments of Cat are Args.
```

```
poss_empty(L,i(tns(+)),[v(nil/tns(+))]).
poss_empty(L,c(emb/rel),[i(tns(+))]).
poss_empty(eng,c(emb/sent),[i(_)]).
poss_empty(ger,c(emb/sent),[i(tns(-))]).
poss_empty(eng,c(mat/sent),[i(tns(+))]).
poss_empty(ger,v(T),_).

%_____________________________________________________________
% empty_cat(Language,Category):  Category is a legal empty category for Language.

empty_cat(L,n(_)).
empty_cat(L,p(_)).

%_____________________________________________________________
% build_tree/5: constructs and appropriate binary tree for the X-bar level
%       of a phrase (e.g. the head and its arguments.)

build_tree(post,C,HD,ArgTrees,xmax(C)/Pair) :-
        reverse(ArgTrees,ATs),make_pairs(post,C,HD,ATs,Pair),!.
build_tree(pre,C,HD,ArgTrees,xmax(C)/Pair) :-
        make_pairs(pre,C,HD,ArgTrees,Pair),!.
make_pairs(Pos,C,xbar(C)/HD,[],HD) :- !.
make_pairs(Pos,C,HD,[A|As],Pair) :-
        pair(Pos,C,HD,A,P),make_pairs(Pos,C,xbar(C)/P,As,Pair),!.
pair(post,C,HD,A,[HD,A]) :- !.
pair(pre, C,HD,A,[A,HD]) :- !.

%_____________________________________________________________
% select(L,C,As,A,NewAs): determine if C has fixed/free order arguments for
%       language L, and selects arguments from As appropriately. For
%       the present system, we assume both English and German to be fixed
%       order, for efficiency purposes.

select(L,C,As,A,NewAs) :-
        order(L,C,Order),!,
        select(Order,As,A,NewAs).
select(free,As,A,NewAs) :-
        append(L,[A|R],As),
        append(L,R,NewAs).
select(fixed,[A|NewAs],A,NewAs) :- !.
```

```
order(eng,_,fixed).
order(ger,_,fixed).

%_____________________________________________________________
% l_mark: L-marks all subcategorized constituents (except IP by CP).

l_mark(c(_),Tree,Tree) :- !.
l_mark(_,Tree1,Tree2) :- !, add_feature(l_marked,Tree1,Tree2).

%_____________________________________________________________
% t_mark: Theta marks all arguments, except CP & IP do not theta-mark
%       their arguments.

t_mark(c(_),_,Tree,Tree) :- !.
t_mark(i(_),_,Tree,Tree) :- !.
t_mark(_,ID,Tree1,Tree2) :- !, add_feature(theta(ID),Tree1,Tree2).

%_____________________________________________________________
% add_feature: adds a given feature, or list of features to a phrasal node.

add_feature(Feature,Tree,Tree) :- var(Feature),!.
add_feature([F|Flist],xmax(C,ID,F1,Cons)/R,xmax(C,ID,F2,Cons)/R) :- !,
        append([F|Flist],F1,F2).
add_feature(Feature,xmax(C,ID,F1,Cons)/R,xmax(C,ID,F2,Cons)/R) :- !,
        append([Feature],F1,F2).

%_____________________________________________________________
% agree(L,SurfTree,DeepTree): verify that the agreement relations hold for
%       SurfTree, and construct a "canonical" (uninflected) DeepTree for
%       purposes of translation. The feature list of DeepTree receives the
%       'agree(Tns,Per,Num/Gen,Case)' feature to keep track of inflection.

agree(Language,SurfTree,BaseTree) :-
        case_agree(SurfTree,Case),
        SurfTree = xmax(C,_,_,_)/_,
        agree1(Language,C,SurfTree,BaseTree1,Tns,Per,NumGen,Case,wh(-)/Proper),
        add_feature([Proper,agree(Tns,Per,NumGen,Case)],BaseTree1,BaseTree),!.

%_____________________________________________________________
```

```
% wh_agree: verify agreement and wh-status of wh-phrases.

wh_agree(Language,Cat,SurfTree,BaseTree) :-
        SurfTree = xmax(C,_,_,_)/_,
        (Cat = c(emb/rel),Proper = proper(+),!,C=n(_) ; true),
        agree1(Language,C,SurfTree,BaseTree1,Tns,Per,NumGen,Case,wh(+)/Proper),
        add_feature([wh(+),ant(+),Proper],BaseTree1,BaseTree2),
        add_feature(agree(Tns,Per,NumGen,Case),BaseTree2,BaseTree),!.

%_____________________________________________________________
% rev_agree: similar to 'agree', but the DeepTree is instantiated and the
%       inflected surface tree is constructed (subject to agreement). Used
%       for generation purposes.

rev_agree(L,Target,NewTarget) :-
        Target = xmax(C,ID,F,Cons)/_,
        case_agree(Target,Case),
        (member(agree(Tns,Per,NumGen,Case),F);true),!,
        agree1(L,C,NewTarget,Target,Tns,Per,NumGen,Case,Wh).

case_agree(xmax(n(Case),_,_,_)/R,NCase) :- !,Case=NCase.
case_agree(xmax(_,_,_,_)/R,Case) :- !.

%_____________________________________________________________
% The following predicates are used by the three agree predicates above.
%       agree1: Traverses the tree, calling agree2.
%       agree2: Enforces agreement relations between constituents, and
%               calls 'agree_terminal' for lexical items.
%       agree_terminal: Accesses morphological information to determine the
%               agreement features of lexical items.

agree1(L,C,X/[LS,RS],X/[LB,RB],Tns,Per,NumGen,Case,Wh) :-
        agree2(L,C,LS,LB,Tns,Per,NumGen,Case,Wh),
        agree2(L,C,RS,RB,Tns,Per,NumGen,Case,Wh).
agree1(L,C,X/RS,X/RB,Tns,Per,NumGen,Case,Wh) :-
        agree2(L,C,RS,RB,Tns,Per,NumGen,Case,Wh).

agree2(L,i(tns(+)),Xmax,Xmax,Tns,Per,NumGen,Case,Wh) :-
        Xmax = xmax(v(nil/tns(+)),_,F,_)/R,!,
        member(agree(Tns1,_,_,_),F),
```

```
        (Tns1 = Tns ; Tns1 = -),!.
agree2(L,i(_),Xmax,Xmax,Tns,Per,NumGen,Case,Wh) :-
        Xmax = xmax(n(nom),_,F,_)/R,!,
        member(agree(_,Per,NumGen,_),F).
agree2(L,c(emb/rel),Xmax,Xmax,Tns,Per,NumGen,Case,Wh) :-
        Xmax = xmax(n(_),_,F,_)/R,!,
        member(agree(_,Per,NumGen,_),F).
agree2(L,c(mat/sent),Xmax,Xmax,Tns,Per,NumGen,Case,Wh) :-
        Xmax = xmax(i(_),_,F,_)/R,!,
        member(agree(ITns,Per,NumGen,_),F),
        (ITns = Tns ; true),!.
agree2(L,n(_),Xmax,Xmax,Tns,Per,NumGen,Case,Wh) :-
        Xmax = xmax(c(emb/rel),_,F,_)/R,!,
        Wh = WH/proper(-),
        member(agree(_,_,NumGen,_),F).
agree2(L,C,Xmax,Xmax,Tns,Per,NumGen,Case,Wh) :-
        Xmax = xmax(_,_,_,_)/R.
agree2(L,C,X/[LS,RS],X/[LB,RB],Tns,Per,NumGen,Case,Wh) :-
        agree2(L,C,LS,LB,Tns,Per,NumGen,Case,Wh),
        agree2(L,C,RS,RB,Tns,Per,NumGen,Case,Wh).
agree2(L,C,X/RS,X/RB,Tns,Per,NumGen,Case,Wh) :-
        agree2(L,C,RS,RB,Tns,Per,NumGen,Case,Wh).
agree2(L,C,XS,XB,Tns,Per,NumGen,Case,Wh)  :-
        agree_terminal(L,XS,XB,Cat,Features),
        extract_features(Cat,Features,Tns,Per,NumGen,Case,Wh).

agree_terminal(L,SurfTerm,BaseTerm,Cat,Features) :-
        member(Node,[spec,head,adj]),
        SurfTerm =.. [Node,Cat,empty,Features],
        BaseTerm =.. [Node,Cat,empty,Features],!.
agree_terminal(L,SurfTerm,BaseTerm,C,F) :-
        member(Node,[spec,head,adj]),
        (var(BaseTerm),SurfTerm=..[Node,C,Word,_],BaseTerm=..[Node,C,Root,F];
        var(SurfTerm),BaseTerm=..[Node,C,Root,_],SurfTerm=..[Node,C,Word,F]),!,
        morph(L,Word,C,Root,F),
        get_cat_features(L,C,F).
agree_terminal(L,SurfTerm,BaseTerm,nil,Features) :-
        (var(SurfTerm),terminal(BaseTerm);
        var(BaseTerm),terminal(SurfTerm)),!,
        SurfTerm = BaseTerm.
```

```
get_cat_features(L,v(Form),Features) :- !,
        v_features(L,Features,Form).
get_cat_features(L,_,_) :- !.

%_____________________________________________________
% extract_features: retrieve relevant features from the dictionary entry.

extract_features(d,Features,Tns,Per,Num/Gen,Case,wh(W)/proper(-)) :- !,
        ext_ftr(wh(W),Features),
        ext_ftr(num(Num,Gen,Case),Features).
extract_features(p(_),Features,Tns,Per,Num/Gen,Case,wh(W)/Proper) :- !,
        ext_ftr(wh(W),Features).
extract_features(n(_),Features,Tns,Per,Num/Gen,Case,wh(W)/proper(P)) :- !,
        ext_ftr(wh(N),Features), (wh(N)=wh(W);(wh(N)=wh(-),\+P=(-))),!,
        ext_ftr(per(Per),Features),
        ext_ftr(num(Num),Features),
        ext_ftr(gen(Gen),Features),
        ext_ftr(case(Case),Features),
        ext_ftr(proper(P),Features).
extract_features(v(_),Features,Tns,Per,Num/Gen,Case,Wh) :- !,
        ext_ftr(tns(Tns),Features),
        ext_ftr(per(Per),Features),
        ext_ftr(num(Num),Features).
extract_features(i(_),Features,Tns,Per,Num/Gen,Case,Wh) :- !,
        ext_ftr(tns(Tns),Features),
        ext_ftr(per(Per),Features),
        ext_ftr(num(Num),Features).
extract_features(_,_,Tns,Per,Num/Gen,Case,Wh) :- !.

ext_ftr(Ftr,Features) :-
        member(ftr(Flist),Features),!,
        member(Ftr,Flist).
```

Appendix C

Language Parameters

```
%_____________________________________________________________
% Section: Phrase Structure Parameters
%
% Description: This section contains the language specific choices for
%       possible specifiers and adjuncts. In addition the head_position
%       parameter is specified for the various categories of each
%       language.
%_____________________________________________________________

%_____________________________________________________________
% General rule to parse simple lexical items as specifiers.

spec(L,C,spec(Cat,W,F),S,S)        -->    [W],
                                          {specifier(L,C,Cat),
                                          morph(L,W,Cat,R,F)}.

%_____________________________________________________________
% Parse punctuation as an adjunct to root CP.

adj(L,post,c(mat/sent),Tree,S,S)-->       ['.'],{Tree = punc(punc,'.',decl)}.
adj(L,post,c(mat/sent),Tree,S,S)-->       ['?'],{Tree = punc(punc,'?',ques)}.

%_____________________________________________________________
% Section: English Configuration
%
% The following define the specific phrase structure rules for: English
```

```
spec(eng,c(T),Tree,S,NS)        -->   wh_phrase(eng,c(T),Tree,S,NS).
spec(eng,i(_),Tree,S,S)         -->   xmax(eng,n(nom),Tree,[],[]).
spec(eng,i(_),Tree,S,S)         -->   xmax_e(eng,n(nom),Tree).

adj(eng,post,n(_),Tree,S,NS)    -->   xmax(eng,p(_),Tree,S,NS).
adj(eng,post,n(_),Tree,S,NS)    -->   xmax(eng,c(emb/rel),Tree,S,NS).
adj(eng,post,v(T),Tree,S,NS)    -->   xmax(eng,p(_),Tree,S,NS).

specifier(eng,n(_),d).

head_position(eng,_,initial).          % All heads are initial.

%_____________________________________________________
% Section: German Configuration
%
% The following define the specific phrase structure rules for: German

spec(ger,c(mat/sent),Tree,S,NS) -->   topic(ger,c(T),Tree,S,NS).
spec(ger,c(T),Tree,S,NS)        -->   wh_phrase(ger,c(T),Tree,S,NS).
spec(ger,i(_),Tree,S,NS)        -->   xmax(ger,n(nom),Tree,S,NS).
spec(ger,i(_),Tree,S,S)         -->   xmax_e(ger,n(nom),Tree).

adj(ger,pre,v(_),Tree,S,NS)     -->   xmax(ger,p(_),Tree,S,NS).
adj(ger,post,n(_),Tree,S,NS)    -->   xmax(ger,p(_),Tree,S,NS).
adj(ger,post,n(_),Tree,S,NS)    -->   xmax(ger,c(emb/rel),Tree,S,NS).

specifier(ger,n(_),d).

head_position(ger,n(_),initial).
head_position(ger,v(_),final).
head_position(ger,a,initial).
head_position(ger,p(_),initial).
head_position(ger,i(_),final).
head_position(ger,c(_),initial).
```

Appendix D

Constraint Module

```
%_____________________________________________________
% Section: Constraints
%
% Paradigm: constraints(Language,Category,Tree).
%
% Description: This module evaluates the constraints, as appropriate for
%       the Category. The action is as follows:
%       1. Determine the constraint list, based on the immediately
%          dominated X-max nodes, and the properties of the current node.
%       2. Satisfy all constraints possible; if there is a definite
%          violation then fail (force a re-parse).
%       3. Return the new list of constraints, to be passed up the tree.
%_____________________________________________________

constraints(L,C,Tree1,Tree2) :-
        get_constraints(Tree1,Cons),
        satisfy_constraints(L,C,Tree1,Cons,Tree2,Constraints),
        add_constraints(Tree2,Constraints).

%_____________________________________________________
% get_constraints(Tree,Constraints): search the tree for all the maximal
%       projections dominated by Tree, gets their constraint lists and
%       merges them.

get_constraints(xmax(C,ID,F,Cons)/R,Cons) :- \+ var(Cons),!.
get_constraints(X/[L,R],Cons) :- !,
```

```
        non_terminal(X),
        get_constraints(L,CL),
        get_constraints(R,CR),
        append(CL,CR,Cons),!.
get_constraints(X/Y,Cons) :- !,
        non_terminal(X),
        get_constraints(Y,Cons),!.
get_constraints(X,[]) :- terminal(X).

terminal(Term) :-
        Term =.. [Pred|_],
        member(Pred,[e_cat,head,spec,adj,punc]),!.
non_terminal(Node) :-
        Node =.. [XBAR|_],
        member(XBAR,[xmax,xbar]),!.

gen_ID(xmax(C,ID,F,Cons)/R) :-
        gensym(C,ID),!.

%_____________________________________________________________
% add_constraints(Tree,Constraints): determines if any new constraints should
%       be added for the current tree (note: some are also added during the
%       satisfy constraint phase).

add_constraints(xmax(C,ID,F,Cons)/R,Constraints) :-
        new_constraints(C,xmax(C,ID,F,Cons)/R,NewCons),
        append(Constraints,NewCons,Cons),!.

new_constraints(n(Case),xmax(n(_),ID,F,Cons)/R,[case(ID,Case)]) :-
        \+ member(ant(+),F),!.
new_constraints(C,xmax(Cat,ID,F,Cons)/R,[ant(abar,Cat,ID)]) :-
        member(ant(+),F),!.
new_constraints(v(_),Tree,[theta(X)]) :-
        get_theta(Tree,X),!.
new_constraints(Cat,Tree,[]) :- !.

get_theta(Tree,X) :-
        get_head(Tree,head(C,W,F)),
        member(theta(X),F),!.
get_theta(_,+) :- !.
```

```
%_____________________________________________________________
% satisfy_constraints(Language,Cat,Tree,Constraints,NewConstraints): applies
%       the principles of Government-Binding theory to the current set of
%       constraints for Tree. Each principle determines its own applicability
%       (if it doesn't apply, it will trivially succeed).

satisfy_constraints(L,C,Tree1,Constraints,Tree3,NewConstraints) :-
        ecp(L,C,Tree1,Constraints),
        case_theory(L,C,Tree1,Tree2,Constraints,Constraints2),
        theta_theory(L,C,Tree2,Constraints2,Tree3,Constraints3),
        subjacency(L,C,Tree3,Constraints3,NewConstraints).

%_____________________________________________________________
% ecp(L,C,Tree,Constraints): if there exists an empty category in Constraints,
%       then if it is properly governed its 'trace', otherwise its 'PRO'.
%       (in fact, this avoids subjects, so never determines PRO).

ecp(L,i(_),Tree,Constraints) :- !.
ecp(L,C,Tree,Constraints) :-
        exists_ec(Constraints,Tree,ID,Type) ->
                ((properly_governs(L,C,Tree,xmax(Cat,ID,_,_)) ->
                        Type = trace(WH) ; Type = pro),
                ecp(L,C,Tree,Constraints)),!.
ecp(L,C,Tree,Constraints) :- !.

exists_ec(Constraints,Tree,ID,Type) :-
        member(ec(_,ID,Type),Constraints),
        var(Type).

%_____________________________________________________________
% case_theory(Language,Cat,Cons1,Cons2): case theory has two applications,
%       1. To mark lexical (non-wh) NP's with case, and fail if it
%          cannot (Case Filter).
%       2. To determine if a given trace is an np-trace or wh-trace.
%       First we determine if Cat is a case assigner for Language, and then
%       do (1) case_mark_lexical, and (2) case_mark_traces above.

case_theory(L,C,Tree1,Tree3,Cons1,Cons2) :-
        case_assigner(L,C,Tree1,Trans),!,
```

```
        case_mark_lexical(Tree1,Tree2,Trans,Cons1,Cons2),
        case_mark_traces(Tree2,Tree3,Trans,Cons2),!.
case_theory(L,C,Tree,Tree,Cons,Cons) :- !.

%_____________________________________________________________
% case_mark_lexical(Tree,Cons1,Cons2): every case request in Cons1, must
%       be governed in Tree, else Case Filter applies (ie. Fail).

case_mark_lexical(Tree1,Tree3,Trans,Cons1,Cons3) :-
        remove(case(ID,Case),Cons1,Cons2),!,
                governed(xmax(n(_),ID,_,_),Tree1,Trans),
                mark_node(ID,case(ID),Tree1,Tree2),
                case_mark_lexical(Tree2,Tree3,Trans,Cons2,Cons3),!.
case_mark_lexical(Tree,Tree,Trans,C,C) :- !.

%_____________________________________________________________
% case_mark_traces(Tree,Constraints): if there exists a trace in Constraints,
%       of unknown type (wh/np), then if it is governed in Tree its a
%       wh-trace, otherwise its an np-trace.

case_mark_traces(Tree1,Tree3,Trans,Constraints) :-
        exists_trace(Constraints,Tree1,ID,WH) ->
              . (governed(xmax(Cat,ID,_,_),Tree1,Trans)  ->
                        (mark_node(ID,case(ID),Tree1,Tree2), WH = wh) ;
                        ( Tree2 = Tree1 , WH = np)),
                case_mark_traces(Tree2,Tree3,Trans,Constraints),!.
case_mark_traces(Tree,Tree,Trans,Constraints) :- !.

exists_trace(Constraints,Tree,ID,WH) :-
        member(ec(Cat,ID,trace(WH)),Constraints),var(WH).

%_____________________________________________________________
% theta_theory: if a verb theta marks it's subject, the theta(+) constraint
%       is removed, and the new Tree, with theta-marked subject, is returned.
%       If at IP there is a theta(-) constraint, then the subject is not
%       theta-marked, but it must be a antecedent for a a-chain (or 'it').

theta_theory(L,i(_),Tree1,OldCons,Tree2,NewCons) :-
        remove(theta(+),OldCons,NewCons),!,
                Tree1 = X/[Subj1,Rest],
```

```
                Subj1 = xmax(_,ID,_,_)/_,
                add_feature(theta(ID),Subj1,Subj2),
                Tree2 = X/[Subj2,Rest].
theta_theory(L,i(_),Tree,OldCons,Tree,NewCons) :-
        \+ member(theta(+),OldCons),
        Tree = X/[Subj,_],
        \+ pleonastic(L,Subj),
        \+ Subj = _/e_cat(_,_),!,
                Subj = xmax(_,ID,_,_)/_,
                append([ant(a,n(_),ID)],OldCons,NewCons).
theta_theory(L,C,Tree,Constraints,Tree,Constraints) :- !.

pleonastic(eng,NP) :- get_head(NP,head(_,it,_)).
pleonastic(ger,NP) :- get_head(NP,head(_,es,_)).

%_____________________________________________________________
% subjacency(Language,Cat,Tree,OldConstraints,NewConstraints): This
%       predicate constrols the construct of chains, and applies the
%       Subjacency principle to determine their well-formedness.

subjacency(L,C,Tree,OldCons,NewCons) :-
        bind_traces(L,C,Tree,OldCons,Cons1),        % bind all traces.
        check_pro_chains(L,C,Tree,Cons1,Cons2),
        \+ member(barrier(ID),Cons2),               % no barriers remain.
        barriers(L,C,Tree,Cons2,NewCons).           % add new barriers if any.

%_____________________________________________________________
% check_pro_chains(L,Cat,Tree,OldConstraints,NewConstraints): if a pro-chain
%       is present, and there is no antecedent available, the the original
%       empty category must have been PRO, and is set accordingly.

check_pro_chains(L,C,Tree,Cons1,Cons4) :-
        remove(chain(pro,ID,CC,Case,Theta,List),Cons1,Cons2),
        remove(barrier(ID),Cons2,Cons3),
        set_last(pro,List),
        check_pro_chains(L,C,Tree,Cons3,Cons4),!.
check_pro_chains(L,C,Tree,Cons,Cons) :- !.

%_____________________________________________________________
% barriers(L,Cat,Tree,OldConstraints,NewConstraints): if the current node is
```

```
%       a barrier (ie, not l-marked), then it becomes a barrier to any
%       chains which it dominates.

barriers(L,C,Tree,OldCons,NewCons) :-
        is_barrier(Tree),
        member(chain(Type,ID,CC,Case,Theta,List),OldCons),
        \+ member(barrier(ID),OldCons),!,          % not already a barrier.
        append([barrier(ID)],OldCons,Cons1),
        barriers(L,C,Tree,Cons1,NewCons).
barriers(L,C,Tree,Cons,Cons) :- !.

is_barrier(xmax(C,ID,Features,Cons)/R) :-          % current node is not l-marked.
        \+ member(l_marked,Features),!.

set_last(Type,[ec(C,ID,Type)|[]]) :- !.
set_last(Type,[A|Y]) :- !,set_last(Type,Y).

%_____________________________________________________________
% bind_traces(Language,Cat,Tree,OldConstraints,NewConstraints): this
%       attempts to either close a chain, by prepending an antecedent to
%       it (clause 1) or extend a chain, by prepending a trace to it
%       (clause 2).

bind_traces(L,c(_),Tree,OldCons,NewCons) :-
        member(chain(Type,ID,CC,Case,Theta,List),OldCons),
        var(Type),!,
        get_head(Tree,head(_,empty,_)),
        Type=pro,
        bind_traces(L,c(_),Tree,OldCons,NewCons),!.
bind_traces(L,v(_),Tree,OldCons,NewCons) :-
        member(chain(Type,ID,CC,Case,Theta,List),OldCons),
        var(Type),!,Type=a,set_last(trace(np),List),
        bind_traces(L,v(_),Tree,OldCons,NewCons),!.
bind_traces(L,C,Tree,OldCons,NewCns) :-
        member((At/Ct),[(a/a),(abar/a),(abar/abar),(abar/pro)]),
        remove(ant(At,CC,ID),OldCons,Cns1),
        remove(chain(Ct,IDC,CC,Case,Th,List),Cns1,Cns2),
        (Ct=pro,set_last(trace(wh),List) ; true),
        subjacent(ID,IDC,Tree,Cns2,Cns3),
        agree_case(ID,Tree,Case,Th),
```

```
        append([c_chain(At,ID,CC,Case,Th,[ant(At,CC,ID)|List])],Cns3,NewCns),!.
bind_traces(L,i(tns(-)),Tree,OldCons,NewCons) :-
        exists_ec(OldCons,Tree,ID,Type) ->
                make_chain(Tree,ec(Cat,ID,Type),Unknown,OldCons,Cons1),
        bind_traces(L,i(tns(-)),Tree,Cons1,NewCons).
bind_traces(L,C,Tree,OldCons,NewCons) :-
        member(ec(Cat,ID,trace(T)),OldCons),!,
                (bind_to_chain(ec(Cat,ID,trace(T)),Tree,OldCons,Cons1);
                make_chain(Tree,ec(Cat,ID,trace(T)),a,OldCons,Cons1)),
        bind_traces(L,C,Tree,Cons1,NewCons).
bind_traces(L,C,Tree,Cons,Cons) :- !.

%_____________________________________________________________
% bind_to_chain(Trace,Tree,OldConstraints,NewConstraints): finds a trace,
%       and a chain, makes sure they are subjacent, and prepends the
%       trace to the chain, and puts the new chain in NewConstraints.

bind_to_chain(ec(C,ID,trace(T)),Tree,OldCons,NewCns) :-
        remove(ec(C,ID,Type),OldCons,Cns1),             % get the empty cat.
        member(ChT,[a,abar,pro]),
        remove(chain(ChT,IDC,C,Cse,Th,List),Cns1,Cns2), % get the chain.
        subjacent(ID,IDC,Tree,Cns2,Cns3),               % are they subjacent?
        agree_case(ID,Tree,Cse,Th),                     % only one theta role.
        append([chain(ChT,ID,C,Cse,Th,[ec(C,ID,trace(T))|List])],Cns3,NewCns),!.

%_____________________________________________________________
% make_chain(Trace,OldConstraints,NewConstraints): if the trace couldn't be
%       attached to a chain, then it must start a new chain.

make_chain(Tree,ec(C,ID,Type),ChType,OldCons,NewCons) :-
        \+ var(Type), Type = trace(T), \+ var(T), T = comp,
        remove(ec(C,ID,Type),OldCons,NewCons),!.
make_chain(Tree,ec(C,ID,Type),ChType,OldCons,NewCons) :-
        remove(ec(C,ID,Type),OldCons,Cons1),      % remove the ec
        agree_case(ID,Tree,Case,Theta),
        append([chain(ChType,ID,C,Case,Theta,[ec(C,ID,Type)])],Cons1,NewCons),!.

subjacent(IDtrace,IDchain,Tree,OldCons,NewCons) :-
        remove(barrier(IDchain),OldCons,NewCons),!.
subjacent(IDtrace,IDchain,Tree,Cons,Cons).
```

```
%_____________________________________________________
% agree_case: ensure that a chain receives Case and a Theta role exactly once.

agree_case(ID,Tree,Case,Theta) :-
        get_subtree(ID,Tree,xmax(C,ID,F,Cons)/R),!,
        (member(case(IDC),F),Case1=case(IDC);true),
        (member(theta(IDT),F),Theta1=theta(IDT);true),!,
        Case=Case1,Theta=Theta1,!.

%_____________________________________________________
% governs(Language,Cat,Tree,Node): first determines is Cat is a governor
%       for Language, and then calls 'governed' to see if Node is governed
%       by the head of the current maximal projection.

governs(L,C,Tree,Node) :-
        governor(L,C),!,
        governed(Node,Tree,trans(-)).

governor(L,C) :- governor_list(L,List),!,member(C,List).
governor_list(_,[n(_),v(_),a,p(_),i(tns(+))]).

%_____________________________________________________
% properly_governs(Language,Cat,Tree,Node): first determines is Cat is a
%       proper governor for Languages, and then calls 'governed' to see
%       if Node is governed by the head of the current maximal projection.

properly_governs(L,C,Tree,Node) :-
        proper_governor(L,C),!,
        pgoverned(Node,Tree).

proper_governor(L,C) :- pg_list(L,List),!,member(C,List).
pg_list(_,[n(_),v(_),p(_)]).

pgoverned(xmax(_,ID,_),xmax(C,IDG,F,Cons)/R) :-
        pgoverned1(ID,IDG,xmax(C,IDG,F,Cons)/R),!.
pgoverned1(ID,IGD,X/[L,R]) :-
        ( L = xmax(C)/T ; R = xmax(C)/T),!,
        pgoverned1(ID,IDG,xmax(C)/T).
pgoverned1(ID,IGD,X/[L,R]) :-
```

```
        ( L = xbar(C)/T ; R = xbar(C)/T),!,
        governed1(ID,IDG,X/[L,R],trans(+)).

%_____________________________________________________________
% governed(Node,Tree): determines if Node is governed by the head of
%       the current maximal projection, which is the root of Tree.

governed(xmax(_,ID,_,_),xmax(C,IDG,F,Cons)/R,Trans) :-
        governed1(ID,IDG,xmax(C,IDG,F,Cons)/R,Trans),!.
governed1(ID,IDG,xmax(_,ID,_,_)/_,Trans) :- !.
governed1(ID,IDG,xmax(C,IDX,_,_)/_,trans(+)) :- \+IDX=IDG,\+C=i(tns(-)),!,fail.
governed1(ID,IDG,X/[L,R],Trans) :- !,
        (governed1(ID,IDG,L,Trans) ; governed1(ID,IDG,R,Trans)).
governed1(ID,IDG,X/Y,Trans) :-
        governed1(ID,IDG,Y,Trans).

%_____________________________________________________________
% case_assigner: determines if the head of the phrase is a case-assigner.

case_assigner(L,v(_),Tree,trans(+)) :-
        get_head(Tree,head(_,_,Features)),
        member(trans(+),Features),!.
case_assigner(L,C,Tree,trans(-)) :-
        case_assigner(L,C),!.
case_assigner(L,c(emb/sent),Tree,trans(+)) :-
        get_head(Tree,head(_,Word,_)),
        member(Word,[for,empty]),!.

case_assigner(_,v(_)).
case_assigner(_,p(_)).
case_assigner(_,i(tns(+))).

%_____________________________________________________________
% mark_node: find the phrase labeled ID and mark its feature list with
%       the term Mark.

mark_node(ID,Mark,xmax(C,ID,F1,Cons)/R,xmax(C,ID,F2,Cons)/R) :- !,
        append([Mark],F1,F2).
mark_node(ID,Mark,X/[L,R],X/[L1,R1]) :- !,
        ( (mark_node(ID,Mark,L,L1),R=R1) ;
```

```
            (mark_node(ID,Mark,R,R1),L=L1) ).
mark_node(ID,Mark,X/R,X/R1) :- !,
        mark_node(ID,Mark,R,R1).
mark_node(ID,Mark,X) :- !,fail.

%_____________________________________________________
% get_subtree: find (and return) the phrase labeled ID.

get_subtree(ID,xmax(C,ID,F,Cons)/R,xmax(C,ID,F,Cons)/R) :- !.
get_subtree(ID,X/[L,R],Subtree) :- !,
        (get_subtree(ID,L,Subtree) ; get_subtree(ID,R,Subtree)).
get_subtree(ID,X/R,Subtree) :- !,
        get_subtree(ID,R,Subtree).
get_subtree(ID,R,Subtree) :- !,fail.
```

Appendix E

Translation Module

```
%_____________________________________________________________
% Section: Translate
%
% Paradigm: translate(SourceTree,TargetTree).
%
% Description: This section is responsible for translating the parse
%       tree for the source input, into the target language. The high
%       level predicates are called as follows:
%               1. gen_deep_structures: this produces the source
%                  D-structure, applying move-alpha in reverse.
%               2. trans_lexical: converts source lexical items into
%                  their inflected, target equivalents. The result is
%                  the D-structure for the target language.
%               3. gen_surf_structures: generates surface structures
%                  for the D-structures of the target language.
%_____________________________________________________________

translate(SourceTree,TargetTree) :-
        write('Source Surface Structure:'),nl,
        pretty(SourceTree),
        gen_deep_structure(SourceTree,SourceDeep),
        trans_lexical(SourceDeep,TargetDeep,Mode),
        gen_surf_structure(Mode,TargetDeep,TargetTree).

%_____________________________________________________________
% gen_deep_structure(SurfaceTree,DeepTree): collapses all chains, returning
```

```
%       moved constituents to their base generated positions. Heads which
%       have moved to Comp (through inversion), are also returned to their
%       base positions.

gen_deep_structure(SourceTree,TargetTree) :-
        rev_move_alpha(SourceTree,TargetTree),!,
        write('Source Deep Structure:'),nl,
        pretty(TargetTree).

%_____________________________________________________________
% rev_move_alpha: is essentially applying move-alpha in reverse.
%       It first calls rev_move_np, and then rev_move_head.

rev_move_alpha(S_structure,D_structure) :-
        S_structure = xmax(C,ID,F,Cons)/_,
        make_trace_lists(1,Cons,Lists),
        rev_move_np(S_structure,D_structure1,Lists),
        rev_move_head(D_structure1,D_structure),!.

%_____________________________________________________________
% rev_move_np: each NP/PP which is the head of a chain is moved back to its
%       base-generated position.

rev_move_np(D_str,D_str,[]) :- !.
rev_move_np(S_str,D_str,[Chain|Rest]) :-
        extract_from_chain(Chain,SurfID,BaseID),
        collapse_chain(S_str,SurfID,D_str1,BaseID),
        rev_move_np(D_str1,D_str,Rest),!.

extract_from_chain(ecl(_,[SurfID|Rest]),SurfID,BaseID) :-
        reverse(Rest,[BaseID|_]),!.

%_____________________________________________________________
% collapse_chain(S-str,SID,D-str,BID): move an element whose S-Str positition
%       is SID to its D-str position specified by BID. This predicate is
%       bi-directional (either S-str/SID or D-str/BID may be specified).

collapse_chain(X/[L,R],SurfID,X/[NewL,NewR],BaseID) :-
        R=xmax(C,SurfID,_,_)/_,leave_ec(X,R,NewR),
        move_to_base(R,BaseID,L,NewL),!.
```

```
collapse_chain(X/[L,R],SurfID,X/[NewL,NewR],BaseID) :-
        L=xmax(C,SurfID,_,_)/_,leave_ec(X,L,NewL),
        move_to_base(L,BaseID,R,NewR),!.
collapse_chain(X/[L,R],SurfID,X/[NewL,NewR],BaseID) :- !,
        collapse_chain(L,SurfID,NewL,BaseID),
        collapse_chain(R,SurfID,NewR,BaseID),!.
collapse_chain(X/R,SurfID,X/NewR,BaseID) :-
        collapse_chain(R,SurfID,NewR,BaseID),!.
collapse_chain(X,_,X,_) :- !.

%_____________________________________________________________
% move_to_base: returns a phrase to its Base position (at BaseID).

move_to_base(Ant,BaseID,X/[L,R],X/[L,NewAnt]) :-
        R=xmax(C,BaseID,_,_)/_,set_features(R,Ant,NewAnt),!.
move_to_base(Ant,BaseID,X/[L,R],X/[NewAnt,R]) :-
        L=xmax(C,BaseID,_,_)/_,set_features(L,Ant,NewAnt),!.
move_to_base(Ant,BaseID,X/[L,R],X/[NewL,NewR]) :- !,
        move_to_base(Ant,BaseID,L,NewL),
        move_to_base(Ant,BaseID,R,NewR),!.
move_to_base(Ant,_,X,X) :- !.

leave_ec(X,xmax(Cat,ID,F,Cons)/_,xmax(Cat,ID,_,_)/e_cat(Type)) :-
        x_node(X,C,_) ->
        ( (C = c(emb/rel), Type = ant) ;
          (C = c(mat/sent),Type = ant) ;
          (C = c(emb/sent),Type = comp) ;
          (C = i(_),       Type = np) ).

set_features(xmax(C,BID,Bftrs,BCons)/X,xmax(C,AID,Aftrs,ACons)/R,
             xmax(C,BID,NewFtrs,[])/R) :-
        \+ var(X),                          % Surface to Base.
        (\+ member(case(_),Aftrs),append([wh(+)],Aftrs,NewFtrs) ;
         Aftrs = NewFtrs),!.
set_features(xmax(C,BID,Ftrs,BCons)/X,xmax(C,AID,Ftrs,_)/R,
             xmax(C,BID,Ftrs,_)/R) :-
        var(X),                             % Base -> Surface.
        X = e_cat(C,trace(wh)).
```

```
%_____________________________________________________
% rev_move_head(OldDstr,NewDstr): heads which have been moved by have/be
%       raising or inversion are also returned to their base positions.

rev_move_head(OldDstr,NewDstr) :-
        find_moved_head(OldDstr,HD,OldDstr1),!,
        move_hd_base(HD,OldDstr1,NewDstr).
rev_move_head(Dstr,Dstr).

find_moved_head(Dstr,head(C2,W,Ftrs),NewDstr) :-
        Dstr = xmax(C1,ID,F,Cons)/R,
        get_head(Dstr,head(C2,W,Ftrs)),
        \+ W = empty,
        \+ C1 = C2,!,
        set_head(head(c(mat/sent),empty,[]),ID,Dstr,NewDstr).

move_hd_base(head(i(_),do,_),OldDstr,NewDstr) :-
        OldDstr = xmax(i(_),ID,F,_)/_,
        get_head(OldDstr,head(C,empty,_)),!,
        member(agree(Tns,_,_,_),F),
        adjust_vp(tns(Tns),OldDstr,NewDstr).
move_hd_base(head(C,W,F),OldDstr,NewDstr) :-
        OldDstr = xmax(C,ID,_,_)/_,
        member(C,[i(_),v(_)]),
        get_head(OldDstr,head(C,empty,_)),!,
        set_head(head(C,W,F),ID,OldDstr,NewDstr).
move_hd_base(HD,X/[L,R],X/[NewL,NewR]) :- !,
        move_hd_base(HD,L,NewL),
        move_hd_base(HD,R,NewR).
move_hd_base(HD,X/R,X/NewR) :- !,
        move_hd_base(HD,R,NewR).
move_hd_base(HD,X,X) :- !.

adjust_vp(Tns,VP,VP) :-
        VP = xmax(C,_,_,_)/_,
        \+ member(C,[i(_),v(_)]),!.
adjust_vp(Tns,OldVP,NewVP) :-
        OldVP = xmax(v(nil/tns(-)),ID,F,_)/_,!,
        get_head(OldVP,head(C,W,Ftr)),
        set_head(head(v(nil/tns(+)),W,Ftr),ID,OldVP,VP1),
```

```
        adjust_tns(Tns,VP1,NewVP).
adjust_vp(Tns,X/[L,R],X/[NewL,NewR]) :- !,
        adjust_vp(Tns,L,NewL),
        adjust_vp(Tns,R,NewR).
adjust_vp(Tns,X/R,X/NewR) :- !,
        adjust_vp(Tns,R,NewR).
adjust_vp(Tns,X,X) :- !.

%_____________________________________________________________
% trans_lexical(SourceDeep,TargetDeep,Mode): translates each lexical item from
%       the source language to that of the target language. At phrasal nodes
%       feature aggreement is forced, producing the inflected (surface) form.
%       Constituents are also re-ordered for the target language.

trans_lexical(SourceDeep,TargetDeep,Mode) :-
        trans_lex(SourceDeep,TargetDeep,Mode),
        write('Target Deep Structure:'),nl,
        pretty(TargetDeep).

trans_lex(SourceDeep,TargetDeep,Mode) :-
        SourceDeep = xmax(C,ID,F,Cons)/R,!,
        SourceDeep2 = xmax(C,ID,F,_)/R,
        trans_lex1(SourceDeep2,TargetDeep,Mode),
        target(L),!,
        rev_constraints(L,C,TargetDeep).
trans_lex(SourceDeep,TargetDeep,Mode) :-
        trans_lex1(SourceDeep,TargetDeep,Mode).

trans_lex1(X/[L,R],Tree,Mode) :-
        X =.. [xmax,Cat|_],
        (x_bar(Cat,L) ; x_bar(Cat,R)),!,
        list_args(X/[L,R],Arglist,Head,Mode),
        order_args(Cat,Arglist,Ordered),
        x_node(X,Cat,Pos),
        (Pos=initial,Position=post ; Pos=final,Position=pre),
        build_tree(Position,Cat,Head,Ordered,_/As),
        Tree = X/As.
trans_lex1(X/[L,R],X/[NL,NR],Mode) :- !,
        trans_lex(L,NL,Mode),
        trans_lex(R,NR,Mode).
```

```
trans_lex1(X/R,X/NewR,Mode) :- !,
        trans_lex(R,NewR,Mode).
trans_lex1(e_cat(C,operator),head(C,das,_),Mode) :- !.
trans_lex1(head(c(emb/rel),_,_),head(c(emb/rel),empty,_),Mode) :- target(ger).
trans_lex1(head(c(emb/sent),_,_),head(c(emb/T),dass,_),Mode) :- target(ger).
trans_lex1(head(c(emb/T),_,_),head(c(emb/T),that,_),Mode) :- target(eng).
trans_lex1(HD,NewHD,Mode) :-
        HD =.. [Pred,C,W,F],
        member(Pred,[head,spec,adj]),!,
        trans_word(C,W,F,NewW,NewF),
        NewHD =.. [Pred,C,NewW,NewF],!.
trans_lex1(Punc,Punc,Mode) :-
        Punc = punc(punc,_,Mode),!.
trans_lex1(X,X,Mode) :- !.

trans_word(Cat,empty,SFtr,empty,_) :- !.
trans_word(c(emb/sent),_,F,NewW,_) :-
        (target(ger),NewW=dass) ; (target(eng),NewW=that),!.
trans_word(c(emb/rel),_,F,NewW,_) :-
        (target(ger),NewW=empty) ; (target(eng),NewW=that),!.
trans_word(Cat,SWord,SFtr,TWord,_) :-
        (member(english(TWord),SFtr);member(german(TWord),SFtr)),!.

%_____________________________________________________________
% list_args(Tree,ArgList,Head,Mode): the arguments of a head (ie those sister
%       to x-bar) are returned as a list, along with the Head & Mode (if
%       applicable).

list_args(X/[L,R],[NewL|Rest],Head,Mode) :-
        x_node(X,C,_),x_bar(C,R),!,
        trans_lex(L,NewL,Mode),
        list_args(R,Rest,Head,Mode).
list_args(X/[L,R],[NewR|Rest],Head,Mode) :-
        x_node(X,C,_),x_bar(C,L),!,
        trans_lex(R,NewR,Mode),
        list_args(L,Rest,Head,Mode).
list_args(Head,[],HD,Mode) :-
        trans_lex(Head,HD,Mode).

%_____________________________________________________________
```

```
% order_args(Category,ArgList,OrderedArgs): accepts a list of arguments and
%       reorders them as appropriate for a given category. Currently this
%       simply moves accusative NP's next to the head.

order_args(Cat,Args,OrdArgs) :-
        remove(xmax(n(acc),ID,F,C)/R,Args,NewArgs),!,
        append(NewArgs,[xmax(n(acc),ID,F,C)/R],OrdArgs).
order_args(Cat,Args,Args) :- !.

order_cons(X,NL,NR,NR,NL) :-
        x_node(X,Cat,Pos),
        ((x_bar(Cat,NL),Pos=final);
        (x_bar(Cat,NR),Pos=initial)),!.
order_cons(X,NL,NR,NL,NR) :- !.

x_node(Node,C,Pos) :-
        Node =.. [Xbar,C|_],member(Xbar,[xmax,xbar]),
        target(L),head_position(L,C,Pos).
x_bar(C,xbar(C)/_) :- !.

%_____________________________________________________________
% rev_constrainst(Language,Category,Tree): apply the appropriate constraints
%       in reverse. Essentially, this routine uses 'subjacency' to determine
%       where Wh-phrases and caseless NP's should be moved.

rev_constraints(L,C,Tree) :-
        get_constraints(Tree,Constraints),
        subjacency(L,C,Tree,Constraints,NewConstraints),
        rev_add_constraints(Tree,NewConstraints).

rev_add_constraints(xmax(C,ID,F,NewCons)/R,Cons) :-
        rev_new_constraints(C,xmax(C,ID,F,Cons)/R,Cons1),
        append(Cons,Cons1,NewCons).

rev_new_constraints(_,xmax(C,ID,_,_)/e_cat(Type),[Cons]) :- !,
        ( (Type = ant   , Cons = ant(abar,_,ID)) ;
          (Type = comp , Cons = ec(_,ID,trace(comp))) ;
          (Type = np    , Cons = ant(a,_,ID)) ).
rev_new_constraints(_,xmax(C,ID,_,_)/e_cat(_,Type),Cons) :- !,
        ( (Type = trace(comp) , Cons = [ec(_,ID,trace(comp))]);
```

```
            Cons = [] ).
rev_new_constraints(_,xmax(C,ID,Ftr,_)/R,[ec(C,ID,trace(wh))]) :-
        member(wh(+),Ftr),!.
rev_new_constraints(n(_),xmax(C,ID,Ftr,_)/R,[ec(C,ID,trace(np))]) :-
        \+ member(case(_),Ftr),!.
rev_new_constraints(_,Tree,[]).

%_____________________________________________________________
% gen_surf_structure(Mode,DeepStructure,SurfaceStructure): generates a
%       surface structure for the target language by applying Move-alpha
%       and certain language specific transformations.


gen_surf_structure(Mode,TargetDeep,TargetSurface) :-
        target(L),set_inversion(L,Mode,Inv),
        move_alpha(TargetDeep,TargetSurface1),
        raising(L,Inv,TargetSurface1,TargetSurface2),
        gen_pf(TargetSurface2,TargetSurface3),
        inversion(Inv,TargetSurface3,TargetSurface4),
        topicalize(L,Mode,TargetSurface4,TargetSurface),
        write('Target Surface Structure:'),nl,
        pretty(TargetSurface).

%_____________________________________________________________
% move_alpha: moves non case-marked NP's and wh-phrase, according to the
%       chains constructed during translation.

move_alpha(Deep,Surface) :-
        Deep = xmax(C,ID,F,Cons)/_,
        make_trace_lists(1,Cons,Lists),
        move_np(Surface,Deep,Lists).

move_np(D_str,D_str,[]) :- !.
move_np(S_str,D_str,[Chain|Rest]) :-
        extract_from_chain(Chain,SurfID,BaseID),
        collapse_chain(D_str1,SurfID,D_str,BaseID),
        move_np(S_str,D_str1,Rest),!.

%_____________________________________________________________
% raising(Lang,Inv,DeepTree,SurfTree): if head of INFL is empty, then the
```

```
%       next highest verb is raised to that position (where is receives its
%       TNS,PER,NUM features). In English, if inversion is to take place,
%       then the verb raised must aux(+), otherwise do-support is required.

raising(Lang,Inv,DeepTree,SurfTree) :-
        DeepTree = xmax(i(Tns),ID,Ftr,Cons)/[Left,Right],!,
        raising(Lang,no,Left,NewL),raising(Lang,no,Right,NewR),
        DeepTree1 = xmax(i(Tns),ID,Ftr,Cons)/[NewL,NewR],
        check_subject(Left,Inv,NewInv),
        raise1(Lang,NewInv,ID,DeepTree1,SurfTree).
raising(Lang,Inv,X/[L,R],X/[NewL,NewR]) :- !,
        raising(Lang,Inv,L,NewL),raising(Lang,Inv,R,NewR).
raising(Lang,Inv,X/R,X/NewR) :- !,raising(Lang,Inv,R,NewR).
raising(Lang,Inv,X,X) :- !.

raise1(Lang,Inv,ID,DeepInfl,SurfInfl) :-
        get_head(DeepInfl,head(i(_),empty,_)),!,
        raise_verb(Lang,Inv,ID,DeepInfl,DeepInfl1,Verb),
        set_head(Verb,ID,DeepInfl1,SurfInfl).
raise1(ger,Inv,ID,DeepInfl,SurfInfl) :-
        get_head(DeepInfl,head(i(tns(-)),zu,_)),!,
        raise_verb(ger,Inv,ID,DeepInfl,DeepInfl1,Verb),
        DeepInfl1 = Xmax/R,
        SurfInfl = Xmax/[xmax(i(tns(-)))/R,Verb].
raise1(L,Inv,ID,Infl,Infl) :- !.

%_____________________________________________________________
% raise_verb: retrieve the head of the VP, and set the head of the phrase
%       to empty. For English, do support is performed if inversion is
%       to take place.

raise_verb(Lang,Inv,IDinfl,Tree,Tree,Verb) :-
        Tree = xmax(i(_),ID,_,_)/_,
        \+ ID = IDinfl,!.
raise_verb(Lang,Inv,IDinfl,DeepVP,SurfVP,Verb) :-
        DeepVP = xmax(v(_),ID,_,_)/_,!,
        get_head(DeepVP,head(v(F),V,Ftr)),
        ((Lang = ger ; Inv = no ; aux(Lang,V)) ->
                (set_head(head(v(F),empty,_),ID,DeepVP,SurfVP),
                 Verb = head(v(F),V,_)) ;
```

```
        (Lang = eng , Inv=yes) ->
                (set_head(head(v(nil/tns(-)),V,_),ID,DeepVP,SurfVP1),
                 adjust_tns(tns(-),SurfVP1,SurfVP),
                 F = Form/Tns,Verb = head(i(Tns),do,_))  ).
raise_verb(Lang,Inv,IDinfl,X/[L,R],X/[NewL,NewR],Verb) :- !,
        raise_verb(Lang,Inv,IDinfl,L,NewL,Verb),
        raise_verb(Lang,Inv,IDinfl,R,NewR,Verb).
raise_verb(Lang,Inv,IDinfl,X/R,X/NewR,Verb) :- !,
        raise_verb(Lang,Inv,IDinfl,R,NewR,Verb).
raise_verb(Lang,Inv,IDinfl,X,X,Verb) :- !.

aux(L,V) :- morph(L,V,v(_),R,Ftr),!,
            member(aux(+),Ftr).

check_subject(xmax(C,ID,F,Cns)/e_cat(_,_),_,no) :- !.
check_subject(_,Inv,Inv).

adjust_tns(tns(T),xmax(C,ID,F,Cns)/R,xmax(C,ID,NewF,Cns)/R) :-
        remove(agree(Tns,Per,NumGen,Case),F,F1),
        append([agree(T,Per,NumGen,Case)],F1,NewF),!.

set_head(HD,ID,xmax(C,IDX,F,Cons)/R,xmax(C,IDX,F,Cons)/R) :-
        \+ ID = IDX,!.
set_head(HD,ID,X/[L,R],X/[NewL,NewR]) :- !,
        set_head(HD,ID,L,NewL),
        set_head(HD,ID,R,NewR),!.
set_head(HD,ID,X/R,X/NewR) :- !,
        set_head(HD,ID,R,NewR),!.
set_head(HD,ID,head(_,_,_),HD) :- !.
set_head(HD,ID,X,X) :- !.

%_____________________________________________________________
% inversion(Language,Mode,DeepTree,SurfTree): simply moves the head of the
%       matrix INFL to head of COMP. This must apply after raising.

inversion(yes,DeepTree,SurfTree) :- !,
        find_infl(DeepTree,SurfTree1,Infl),
        SurfTree1 = xmax(_,ID,_,_)/_,
        set_head(Infl,ID,SurfTree1,SurfTree).
inversion(no,Tree,Tree) :- !.
```

```
find_infl(Xmax,InflMax,Infl) :-
        Xmax = xmax(i(tns(+)),ID,_,_)/_,!,
        get_head(Xmax,Infl),
        set_head(head(i(tns(+)),empty,_),ID,Xmax,InflMax).
find_infl(X/[L,R],X/[NewL,NewR],Infl) :- !,
        ((find_infl(R,NewR,Infl),L=NewL);
        (find_infl(L,NewL,Infl),R=NewR)).
find_infl(X/R,X/NewR,Infl) :- !,
        find_infl(R,NewR,Infl).

set_inversion(ger,_,yes).
set_inversion(eng,ques,yes).
set_inversion(eng,decl,no).

%_____________________________________________________________
% gen_pf(SourceTree,TargetTree): after move-alpha has applied to wh-phrase
%       and verbal heads, traverse the tree and generate the appropriate
%       surface forms for agreement.

gen_pf(SourceDeep,TargetDeep) :-
        SourceDeep = xmax(C,ID,F,Cons)/R,
        member(C,[n(_),c(emb/rel)]),!,
        target(L),
        rev_agree(L,SourceDeep,TargetDeep1),
        gen_pf1(TargetDeep1,TargetDeep).
gen_pf(SourceDeep,TargetDeep) :-
        SourceDeep = xmax(C,ID,F,Cons)/R,!,
        gen_pf1(SourceDeep,TargetDeep1),
        target(L),!,
        rev_agree(L,TargetDeep1,TargetDeep).
gen_pf(SourceDeep,TargetDeep) :-
        gen_pf1(SourceDeep,TargetDeep).

gen_pf1(X/[L,R],X/[NewL,NewR]) :- !,
        gen_pf(L,NewL),
        gen_pf(R,NewR).
gen_pf1(X/R,X/NewR) :- !,
        gen_pf(R,NewR).
gen_pf1(X,X) :- !.
```

```
%_____________________________________________________
% topicalize(Language,Mode,DeepTree,SurfTree): if Mode is Declarative then
%       find a Topic in DeepTree, and attach it to Comp. (German only).

topicalize(ger,decl,DeepTree,SurfTree) :- !,
        get_topic(DeepTree,Tree,Topic),
        attach_topic(Topic,Tree,SurfTree).
topicalize(_,_,Tree,Tree) :- !.

%_____________________________________________________
% get_topic(DeepTree,SurfTree,Topic): finds the first constituent under
%       Infl in DeepTree, then returns it as Topic, and SurfTree has an
%       ec where the topic was.

get_topic(Tree1,Tree2,Topic) :-
        Tree1 = X/[Topic,R],
        x_node(X,i(tns(+)),_),!,
        leave_ec(X,Topic,EC),
        Tree2 = X/[EC,R].
get_topic(X/[L,R],X/[NewL,NewR],Topic) :- !,
        ((get_topic(L,NewL,Topic),R=NewR) ;
        (get_topic(R,NewR,Topic),L=NewL)).
get_topic(X/R,X/NewR,Topic) :- !,
        get_topic(R,NewR,Topic).

attach_topic(Topic,X/Rest,Tree) :-
        Tree = X/[Topic,xmax(c(mat/sent))/Rest].
```

Appendix F

Morphological Analyser

```
%_____________________________________________________________
% Section: Morphological Analyzer
%
% Paradigm: morph(Language,Word,Category,Root,Features).
%       Language:       Source language {english,german}.
%       Word:           The source word to be morphed.
%       Category:       Lexical category {n,v,a,p,i,c,d}.
%       Root:           The Word's root, and
%       Features:       The syntactic features of the word.
%
% Description:  This section is responsible for morphological analyses
%       and dictionary lookup of lexical items. Language and either Word
%       or Root must be specified, Category is optional.
%       The <morph> predicate first checks if the entry is already in
%       the lexicon, if not it performs a suffix analysis based on the
%       assumption that the item is not irregular. The predicate has
%       been written so that if Word is instantiated, Root+Features is
%       returned, and vice versa.
%_____________________________________________________________

morph(L,Word,punc,punc(Word),[]) :- member(Word,['.','?','!']),!.
morph(L,Word,Cat,Root,Data) :-
        \+ atom(Word),var(Root), !, fail.
morph(L,Word,Cat,Root,Data) :-
        direction(Word,Root,forward),
        dict(L,Cat,Word,F1),
```

```
        remove(root(Root),F1,F2),
        get_root_features(L,Cat,Root,RF),
        remove_ftr(RF,NewRF),
        append(F2,NewRF,Data1),
        set_defaults(L,Cat,Data1,Data).
morph(L,Root,Cat,Root,Data) :-
        direction(Root,Root,forward),
        dict(L,Cat,Root,Data1),
        \+ member(root(_),Data1),
        set_defaults(L,Cat,Data1,Data).
morph(L,Word,Cat,Root,Data) :-
        direction(Word,Root,forward),
        suff_table(L,Cat,Suff,End,Sfeat),
        suffanal(Word,Cat,Suff,End,Root),
        get_root_features(L,Cat,Root,RF),
        adjust_ftr(RF,Sfeat,Data1),
        set_defaults(L,Cat,Data1,Data).
morph(L,Word,Cat,Root,Data) :-
        direction(Word,Root,backward),
        dict(L,Cat,Root,Data1),
        \+ member(root(_),Data1),
        member(irr(List),Data1),!,
        member(IrrWord,List),
        dict(L,Cat,IrrWord,Data2),
        set_defaults(L,Cat,Data2,Data).

direction(Word,Root,forward).
direction(Word,Root,backward) :- var(Word),atom(Root).

%_____________________________________________________________
% The following predicates are used to extract features, and construct new
%       feature lists for "created" lexical items.

remove_ftr(Ftr,NewFtr) :-
        (remove(ftr(_),Ftr,NewFtr);Ftr=NewFtr),!.

adjust_ftr(RF,Sftr,Data) :-
        (remove(ftr(Rftr),RF,NewRF);Rftr=[],NewRF=RF),!,
        def_merge(Rftr,Sftr,Newftr),
        append([ftr(Newftr)],NewRF,Data),!.
```

```
get_root_features(L,Cat,Root,Features) :-
        dict(L,Cat,Root,Features),
        \+ member(root(_),Features).

set_defaults(L,Cat,OldFeatures,NewFeatures) :-
        (remove(ftr(OldFlist),OldFeatures,Ftrs);
         Ftrs = OldFeatures,OldFlist = []),
        defaults(L,Cat,Defaults),!,
        def_merge(Defaults,OldFlist,NewFlist),
        append([ftr(NewFlist)],Ftrs,NewFeatures),!.

def_merge([],Features,Features) :- !.
def_merge([DF|R],Features,[DF|NewFeatures]) :-
        nul_feature(DF,NF),
        \+ member(NF,Features),
        def_merge(R,Features,NewFeatures).
def_merge([_|R],Features,NewFeatures) :-
        def_merge(R,Features,NewFeatures).

nul_feature(F,NulF) :-
        F =.. [P|Args],
        nul_args(Args,NulArgs),
        NulF =.. [P|NulArgs].

nul_args([],[]) :- !.
nul_args([W|X],[Y|Z]) :-
        nul_args(X,Z).

%_____________________________________________________________
% suffanal: Finds possible suffix/root combinations. (Either Word or Root
%       may be instantiated.)

suffanal(Word,Cat,Suff,End,Root) :-
        atom(Word),var(Root),
        process(Word,Wlist),
        process(Suff,Slist),
        match(Wlist,Slist,PRroot),
        reverse(PRroot,Proot),
        name(End,Endlist),
```

```
        append(Proot,Endlist,Rootlist),
        name(Root,Rootlist),!.

suffanal(Word,Cat,Suff,End,Root) :-
        var(Word),atom(Root),
        process(Root,Rlist),
        process(End,Endlist),
        match(Rlist,Endlist,BRlist),
        reverse(BRlist,Blist),
        name(Suff,Slist),
        append(Blist,Slist,Wordlist),
        name(Word,Wordlist),!.

%_____________________________________________________________
% Determine the subcategorization frame (arguments) of a given head.

getargs(L,head(C,W,As,RF)) :-
        member(subcat(_),RF),!,
        member(subcat(Args),RF),
        convert_to_list(Args,As).
getargs(L,head(C,W,[],F)) :- !.

convert_to_list([A|As],[A|As]) :- !.
convert_to_list(A,[A]) :- !.

%_____________________________________________________________
% get_head: retrieves the head of the current subtree.

get_head(xmax(C,ID,_,_)/R,Head) :-
        get_head1(ID,xmax(C,ID,_,_)/R,Head).
get_head1(ID,head(C,W,F),head(C,W,F)) :- !.
get_head1(ID,xmax(_,IDX,_,_)/R,_) :- \+ ID = IDX,!,fail.
get_head1(ID,X/[L,R],Head) :-
        (get_head1(ID,L,Head);get_head1(ID,R,Head)).
get_head1(ID,X/L,Head) :-
        get_head1(ID,L,Head).

%_____________________________________________________________
% Determine the participle form and tense of the verb, for purposes
% of subcategorization.
```

```
v_features(L,F,Form/Tns) :- !,
        get_ftr(F,Ftr),
        get_form(Ftr,Form),
        get_tns(Ftr,Tns).

get_ftr(F,Ftr) :- member(ftr(Ftr),F),!.
get_ftr(F,[]) :- \+ member(ftr(Ftr),F),!.

get_form(F,part(Form)) :- member(part(Form),F),!.
get_form(F,nil) :- \+ member(part(Form),F),!.

get_tns(F,tns(-)) :- member(tns(-),F),!.
get_tns(F,tns(-)) :- \+ member(tns(X),F),!.
get_tns(F,tns(+)) :- member(tns(X),F),\+ X = '-',!.

%_____________________________________________________________
% Utility routines used by morpher only.

match(Wlist,[],Wlist).
match([L|Wlist],[L|Slist],Root) :- match(Wlist,Slist,Root).

process('',[]).
process(Term,List) :-
        name(Term,List1),
        reverse(List1,List),!.

get_ftr(OldFeat,Feat,Rest) :-
        append(ftr(Feat),Rest,OldFeat),!.
get_ftr(R,[],R) :- !.
```

Appendix G

The Lexicon

```
%_____________________________________________________________
% Section: Suffix Table (English)
%
% Paradigm: suff_table(Language,Category,InflEnd,RootEnd,Features).
%       Language:       Language of source {eng,ger}.
%       Category:       Lexical category {n,v}.
%       InflEnd:        The inflected ending.
%       RootEnd:        The ending of the root form.
%       Features:       Features to be added by the inflection.
%_____________________________________________________________

suff_table(eng,n(_),ies,y,[num(+)]).
suff_table(eng,n(_),ves,f,[num(+)]).
suff_table(eng,n(_),ves,fe,[num(+)]).
suff_table(eng,n(_),s,'',[num(+)]).
suff_table(eng,n(_),es,'',[num(+)]).
suff_table(eng,v(_),'','',[tns(pres),per(1)]).
suff_table(eng,v(_),'','',[tns(pres),per(2)]).
suff_table(eng,v(_),'','',[tns(pres),per(3),num(+)]).
suff_table(eng,v(_),ies,y,[num(-),per(3),tns(pres)]).
suff_table(eng,v(_),s,'',[num(-),per(3),tns(pres)]).
suff_table(eng,v(_),ied,y,[tns(past)]).
suff_table(eng,v(_),ed,e,[tns(past)]).
suff_table(eng,v(_),ed,'',[tns(past)]).
suff_table(eng,v(_),ed,e,[part(past),tns(-)]).
suff_table(eng,v(_),ed,'',[part(past),tns(-)]).
```

```
suff_table(eng,v(_),en,e,[part(past),tns(-)]).
suff_table(eng,v(_),en,'',[part(past),tns(-)]).
suff_table(eng,v(_),ing,e,[part(pres)]).
suff_table(eng,v(_),ing,'',[part(pres)]).

defaults(eng,n(_),[per(3),num(-),gen(_),wh(-),case(_),proper(-)]).
defaults(eng,d,[num(_,_,_),wh(-)]).
defaults(eng,p(_),[wh(-)]).
defaults(eng,v(_),[tns(-),per(_),num(_),aux(-),trans(+)]).
defaults(eng,i(_),[tns(+),per(_),num(_)]).
defaults(eng,Cat,[]).

%_____________________________________________________
%
% Section: Lexicon (English)
%
% Paradigm: dict(L,Cat,Word,[ftr([tns(T),gen(G),num(P),per(N),part(F),wh(M)]),
%                   irr([F1,F2,...]),pastpart(F),prespart(F),
%                   pl(PF),proper(M),pro(M),root(RF),subcat(Frame)]).
%_____________________________________________________

%% _____________
%% Nouns
%%
dict(eng,n(_),book,[german(buch)]).
dict(eng,n(_),boy,[pl(boys),german(junge)]).
dict(eng,n(_),boys,[ftr([num(+)]),root(boy)]).
dict(eng,n(_),girl,[german(madchen)]).
dict(eng,n(_),table,[german(tisch)]).
dict(eng,n(_),woman,[pl(women),german(frau)]).
dict(eng,n(_),women,[ftr([num(+)]),root(woman)]).
dict(eng,n(_),i,[ftr([case(nom),per(1)]),proper(+),german(ich)]).
dict(eng,n(_),me,[ftr([case(acc),per(1)]),root(i)]).
dict(eng,n(_),we,[ftr([case(nom),num(+),per(1)]),proper(+),german(wir)]).
dict(eng,n(_),us,[ftr([case(acc),num(+),per(1)]),root(we)]).
dict(eng,n(_),you,[ftr([per(2)]),proper(+),german(sie)]).
dict(eng,n(_),he,[ftr([case(nom)]),proper(+),german(er)]).
dict(eng,n(_),him,[ftr([case(acc)]),root(he)]).
dict(eng,n(_),she,[ftr([case(nom)]),proper(+),german(sie)]).
dict(eng,n(_),her,[ftr([case(acc)]),root(she)]).
```

```
dict(eng,n(_),it,[german(es)]).
dict(eng,n(_),what,[ftr([wh(+),proper(+)]),german(was)]).
dict(eng,n(_),which,[ftr([wh(+),proper(+)]),german(das)]).
dict(eng,n(_),who,[ftr([wh(+),proper(+)]),german(wer)]).
%% _____
%% Verbs
%%
dict(eng,v(_),put,[subcat([n(acc),p(loc)]),irr([putting]),
        german(legen)]).
dict(eng,v(_),put,[ftr([part(past),tns(-)]),root(put)]).
dict(eng,v(_),put,[ftr([tns(past)]),root(put)]).
dict(eng,v(_),putting,[ftr([part(pres),tns(pres)]),root(put)]).

dict(eng,v(_),see,[subcat([n(acc)]),irr([seen,saw]),pastpart(seen),
        german(sehen)]).
dict(eng,v(_),seen,[ftr([part(past),tns(-)]),root(see)]).
dict(eng,v(_),saw,[ftr([tns(past)]),root(see)]).
dict(eng,v(_),try,[subcat(c(emb/sent)),irr([tried]),
        german(versuchen)]).
dict(eng,v(_),give,[subcat([n(_),p(dir)]),irr([gave])]).
dict(eng,v(_),gave,[ftr([tns(past)]),root(give)]).
dict(eng,v(_),believe,[subcat(i(tns(-))),subcat(c(emb/sent)),
                        irr([believed]),german(glauben)]).
dict(eng,v(_),believed,[ftr([part(past)]),root(believe)]).
dict(eng,v(_),seem,[ftr([trans(-)]),subcat(i(tns(-))),subcat(c(emb/sent)),
        theta(-),german(sheinen)]).
dict(eng,v(_),want,[subcat(c(emb/sent)),pastpart(wanted),
                        irr([wanted]),german(mogen)]).
dict(eng,v(_),have,[subcat(v(part(past)/tns(-))),aux(+),irr([has]),
        german(haben)]).
dict(eng,v(_),has,[ftr([tns(pres),per(3),num(-)]),root(have)]).
dict(eng,v(_),be,[subcat(v(part(pres)/tns(-))),aux(+),german(sein)]).
dict(eng,v(_),been,[ftr([part(past)]),subcat(v(part(pres)/tns(-))),
                        german(sein)]).
dict(eng,v(_),being,[ftr([part(pres)]),subcat(v(part(pres)/tns(-))),
                        german(sein)]).
dict(eng,v(_),is,[ftr([tns(pres),per(3),num(-)]),root(be)]).
dict(eng,v(_),am,[ftr([tns(pres),per(1),num(-)]),root(be)]).
dict(eng,v(_),are,[ftr([tns(pres),num(+)]),root(be)]).
dict(eng,v(_),are,[ftr([tns(pres),per(2)]),root(be)]).
```

```
dict(eng,v(_),was,[ftr([tns(past),per(3),num(-)]),root(be)]).
dict(eng,v(_),was,[ftr([tns(past),per(1),num(-)]),root(be)]).
dict(eng,v(_),were,[ftr([tns(past),num(+)]),root(be)]).
dict(eng,v(_),were,[ftr([tns(past),per(2)]),root(be)]).
%% ____________________
%% Modals
%%
dict(eng,i(tns(-)),to,[ftr([tns(-)]),subcat(v(nil/tns(-))),german(zu)]).
dict(eng,i(tns(+)),will,[ftr([tns(fut)]),subcat(v(nil/tns(-))),german(werden)]).
dict(eng,i(tns(+)),do,[subcat(v(nil/tns(-))),aux(+),german(nil)]).
dict(eng,i(tns(+)),do,[ftr([tns(pres),per(1),per(2)]),root(do)]).
dict(eng,i(tns(+)),do,[ftr([tns(pres),per(3),num(+)]),root(do)]).
dict(eng,i(tns(+)),does,[ftr([tns(pres),per(3),num(-)]),root(do)]).
dict(eng,i(tns(+)),did,[ftr([tns(past),per(_),num(_)]),root(do)]).
%% ____________________
%% Prepositions
%%
dict(eng,p(dir),to,[subcat(n(acc)),german([nach])]).
dict(eng,p(loc),on,[subcat(n(acc)),german(auf)]).
dict(eng,p(_),where,[ftr([wh(+)]),german(wo)]).
%% ____________________
%% Complementizers
%%
dict(eng,c(emb/_),that,[subcat(i(tns(+))),german(dass)]).
dict(eng,c(emb/_),for,[ftr([tns(-)]),subcat(i(tns(-))),german(denn)]).
%% ____________________
%% Determiners
%%
dict(eng,d,the,[german(das)]).
dict(eng,d,a,[ftr([num(-)]),german(ein)]).
dict(eng,d,what,[ftr([wh(+)]),german(welches)]).
dict(eng,d,which,[ftr([wh(+)]),german(welches)]).
dict(eng,d,every,[ftr([num(-)]),german(jede)]).

%_____________________________________________________
% Section: Suffix Table (German)
%
% Paradigm: suff_table(Language,Category,InflEnd,RootEnd,Features).
%       Language:       Language of source {ger,ger}.
%       Category:       Lexical category {n,v}.
```

```
%       InflEnd:        The inflected ending.
%       RootEnd:        The ending of the root form.
%       Features:       Features to be added by the inflection.
%_____________________________________________________________

suff_table(ger,n(_),en,'e',[num(+)]).
suff_table(ger,n(_),en,'',[num(+)]).
suff_table(ger,v(_),en,'',[part(past)]).
suff_table(ger,v(_),'','n',[per(1),num(-),tns(pres)]).
suff_table(ger,v(_),'','',[per(2),tns(pres)]).
suff_table(ger,v(_),'','',[num(+),tns(pres)]).

defaults(ger,n(_),[per(3),num(-),wh(-),gen(_),case(_),proper(-)]).
defaults(ger,d,[wh(-)]).
defaults(ger,p(_),[wh(-)]).
defaults(ger,v(_),[tns(-),per(_),num(_),trans(+)]).
defaults(ger,i(_),[tns(+),per(_),num(_)]).
defaults(ger,Cat,[]).

%_____________________________________________________________
% Section: Lexicon (German)
%
% Paradigm: dict(L,Cat,Word,[ftr([tns(T),num(PG,C),per(N),part(F),wh(M)]),
%                   irr([F1,F2,...]),pastpart(F),prespart(F),
%                   pl(PF),proper(M),pro(M),root(RF),subcat(Frame)]).
%_____________________________________________________________

%% _____________________
%% Nouns
%%
dict(ger,n(_),ich,[ftr([case(nom),per(1)]),proper(+),irr([mich,mir]),
        english(i)]).
dict(ger,n(_),mich,[ftr([case(acc),per(1)]),root(ich)]).
dict(ger,n(_),mir,[ftr([case(dat),per(1)]),root(ich)]).
dict(ger,n(_),sie,[ftr([case(nom),case(acc),per(2)]),proper(+),irr([ihnen]),
        english(you)]).
dict(ger,n(_),ihnen,[ftr([case(dat),per(2)]),root(sie)]).
dict(ger,n(_),sie,[ftr([case(nom),case(acc)]),proper(+),irr([ihnen]),
        english(she)]).
dict(ger,n(_),ihr,[ftr([case(dat)]),root(sie)]).
```

```
dict(ger,n(_),er,[ftr([case(nom)]),proper(+),irr([ihn,ihm]),
        english(he)]).
dict(ger,n(_),ihn,[ftr([case(acc)]),root(er)]).
dict(ger,n(_),ihm,[ftr([case(dat)]),root(er)]).
dict(ger,n(_),wir,[ftr([case(nom),per(1),num(+)]),proper(+),irr([uns]),
        english(we)]).
dict(ger,n(_),uns,[ftr([per(1),num(+),case(acc),case(dat)]),root(wir)]).
dict(ger,n(_),buch,[ftr([gen(n)]),english(book)]).
dict(ger,n(_),junge,[ftr([gen(m)]),english(boy)]).
dict(ger,n(_),madchen,[ftr([gen(n)]),english(girl)]).
dict(ger,n(_),tisch,[ftr([gen(m)]),english(table)]).
dict(ger,n(_),frau,[ftr([gen(f)]),english(woman)]).
dict(ger,n(_),es,[english(es)]).
dict(ger,n(_),was,[ftr([case(nom),case(acc),proper(+),wh(+)]),irr([wem]),english(what)]).
dict(ger,n(_),wem,[ftr([case(dat),proper(+),wh(+)]),root(was)]).
dict(ger,n(_),wer,[ftr([case(nom),proper(+),wh(+)]),irr([wen,wem]),english(who)]).
dict(ger,n(_),wen,[ftr([case(acc),proper(+),wh(+)]),root(wer)]).
dict(ger,n(_),wem,[ftr([case(dat),proper(+),wh(+)]),root(wer)]).
dict(ger,n(_),das,[ftr([wh(+),proper(+),num(-),gen(n),case(nom),case(acc)]),
        irr([das,der,die,den,dem]),english(which)]).
dict(ger,n(_),der,[ftr([wh(+),proper(+),num(-),gen(m),case(nom)]),root(das)]).
dict(ger,n(_),der,[ftr([wh(+),proper(+),num(-),gen(f),case(dat)]),root(das)]).
dict(ger,n(_),die,[ftr([wh(+),proper(+),num(-),gen(f),case(nom),case(acc)]),root(das)]).
dict(ger,n(_),die,[ftr([wh(+),proper(+),num(+),gen(_),case(nom),case(acc)]),root(das)]).
dict(ger,n(_),den,[ftr([wh(+),proper(+),num(-),gen(m),case(acc)]),root(das)]).
dict(ger,n(_),den,[ftr([wh(+),proper(+),num(+),gen(_),case(acc)]),root(das)]).
dict(ger,n(_),dem,[ftr([wh(+),proper(+),num(-),case(dat),gen(m),gen(n)]),root(das)]).
%% ____________________
%% Verbs
%%
dict(ger,v(_),legen,[subcat([n(acc),p(loc)]),
        irr([lege,legen,legt,lag,lagen,gelegt]),english(put)]).
dict(ger,v(_),legt,[ftr([per(3),tns(pres)]),root(legen)]).
dict(ger,v(_),legte,[ftr([num(-),tns(past)]),root(legen)]).
dict(ger,v(_),legten,[ftr([num(+),tns(past)]),root(legen)]).
dict(ger,v(_),gelegt,[ftr([part(past)]),root(legen)]).
dict(ger,v(_),sehen,[subcat([n(acc)]),english(see)]).
dict(ger,v(_),sieht,[ftr([per(3),tns(pres)]),root(sehen)]).
dict(ger,v(_),sah,[ftr([num(-),tns(past)]),root(sehen)]).
dict(ger,v(_),sahen,[ftr([num(+),tns(past)]),root(sehen)]).
```

```
dict(ger,v(_),gesehen,[ftr([part(past)]),root(sehen)]).
dict(ger,v(_),geben,[subcat([n(acc),n(dat)]),english(give)]).
dict(ger,v(_),gibt,[ftr([per(3),tns(past)]),root(geben)]).
dict(ger,v(_),gab,[ftr([num(−),tns(past)]),root(geben)]).
dict(ger,v(_),gaben,[ftr([num(+),tns(past)]),root(geben)]).
dict(ger,v(_),gegeben,[ftr([part(past)]),root(geben)]).
dict(ger,v(_),glauben,[subcat(i(tns(−))),subcat(c(emb/sent)),english(believe)]).
dict(ger,v(_),glaubt,[ftr([per(3),tns(pres)]),root(glauben)]).
dict(ger,v(_),glaubte,[ftr([num(−),tns(past)]),root(glauben)]).
dict(ger,v(_),glaubten,[ftr([num(+),tns(past)]),root(glauben)]).
dict(ger,v(_),geglaubt,[ftr([part(past)]),root(glauben)]).
dict(ger,v(_),versuchen,[subcat(c(emb/sent)),english(try)]).
dict(ger,v(_),versucht,[ftr([per(3),tns(pres)]),root(versuchen)]).
dict(ger,v(_),versuchte,[ftr([num(−),tns(past)]),root(versuchen)]).
dict(ger,v(_),versuchten,[ftr([num(+),tns(past)]),root(versuchen)]).
dict(ger,v(_),versucht,[ftr([part(past)]),root(versuchen)]).
dict(ger,v(_),sheinen,[subcat(i(tns(−))),subcat(c(emb/sent)),english(seem)]).
dict(ger,v(_),sheint,[ftr([per(3),tns(pres)]),root(sheinen)]).
dict(ger,v(_),gesheint,[ftr([part(past)]),root(sheinen)]).
dict(ger,v(_),sein,[subcat(v(_)),aux(+),english(be)]).
dict(ger,v(_),ist,[ftr([tns(pres),per(3),num(−)]),root(sein)]).
dict(ger,v(_),bin,[ftr([tns(pres),per(1),num(−)]),root(sein)]).
dict(ger,v(_),sind,[ftr([tns(pres),num(+)]),root(sein)]).
dict(ger,v(_),sind,[ftr([tns(pres),per(2)]),root(sein)]).
dict(ger,v(_),war,[ftr([tns(past),per(3),num(−)]),root(sein)]).
dict(ger,v(_),gewesen,[ftr([tns(past),per(1),num(−)]),root(sein)]).
dict(ger,v(_),haben,[subcat(v(part(past)/tns(−))),aux(+),english(have)]).
dict(ger,v(_),haben,[ftr([tns(pres),per(2),num(_)]),root(haben)]).
dict(ger,v(_),haben,[ftr([tns(pres),per(_),num(+)]),root(haben)]).
dict(ger,v(_),habe,[ftr([tns(pres),per(1),num(−)]),root(haben)]).
dict(ger,v(_),hat,[ftr([per(3),tns(pres),num(−)]),root(haben)]).
%% ____________________
%% Modals
%%
dict(ger,i(tns(−)),zu,[ftr([tns(−)]),subcat(v(nil/tns(−))),english(to)]).
dict(ger,i(tns(+)),werden,[subcat(v(nil/tns(−))),english(will)]).
dict(ger,i(tns(+)),werden,[ftr([tns(fut),per(2),num(_)]),root(werden)]).
dict(ger,i(tns(+)),werden,[ftr([tns(fut),per(_),num(+)]),root(werden)]).
dict(ger,i(tns(+)),werde,[ftr([tns(fut),per(1),num(−)]),root(werden)]).
dict(ger,i(tns(+)),wird,[ftr([tns(fut),per(3),num(−)]),root(werden)]).
```

```
%% ____________________
%% Prepositions
%%
dict(ger,p(dir),zu,[subcat(n(_)),english(to)]).
dict(ger,p(loc),auf,[subcat(n(_)),english(on)]).
dict(ger,p(loc),in,[subcat(n(_)),english(in)]).
dict(ger,p(_),wo,[ftr([wh(+)]),english(where)]).
%% ____________________
%% Complementizers
%%
dict(ger,c(emb/_),dass,[subcat(i(_)),english(that)]).
dict(ger,c(emb/_),denn,[ftr([tns(-)]),subcat(i(T)),english(for)]).
%% Determiners
%%
dict(ger,d,das,[irr([der,das,die,den,dem]),english(the)]).
dict(ger,d,das,[ftr([num(-,n,nom),num(-,n,acc)]),root(das)]).
dict(ger,d,der,[ftr([num(-,m,nom),num(-,f,dat)]),root(das)]).
dict(ger,d,die,[ftr([num(-,f,nom),num(-,f,acc),num(+,_,nom),num(+,_,acc)]),
        root(das)]).
dict(ger,d,den,[ftr([num(-,m,acc),num(+,_,acc)]),root(das)]).
dict(ger,d,dem,[ftr([num(-,m,dat),num(-,n,dat)]),root(das)]).
dict(ger,d,ein,[irr([ein,einer,eine,einem,einen]),english(a)]).
dict(ger,d,ein,[ftr([num(-,n,nom),num(-,n,acc),num(-,m,nom)]),root(ein)]).
dict(ger,d,einer,[ftr([num(-,f,dat)]),root(ein)]).
dict(ger,d,eine,[ftr([num(-,f,nom),num(-,f,acc)]),root(ein)]).
dict(ger,d,einen,[ftr([num(-,m,acc),num(+,_,acc)]),root(ein)]).
dict(ger,d,einem,[ftr([num(-,m,dat),num(-,n,dat)]),root(ein)]).
dict(eng,d,welches,[ftr([wh(+),num(_,_,_)]),english(which)]).
```