Guarded Constraints in Natural Language Processing

Kathryn L. Baker

July 27, 2002

CMU-LTI-02-XXX

Language Technologies Institute
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Thesis Committee:
Bob Carpenter, Chair
Teruko Mitamura, Co-Chair
Chris Manning
Carl Pollard
Rich Thomason

Acknowledgements

I would like to thank the members of my committee: Bob Carpenter, Teruko Mitamura, Chris Manning, Carl Pollard, and Rich Thomason. All have ties to Pittsburgh, but as we have become separated by time and distance, I appreciate their seeing this project through. Thanks to Rich for bringing me to Pittsburgh in the first place.

I am struck by Bob's sharp memory and attention to detail as he has provided numerous comments, even while working at Lucent and SpeechWorks International. He kept his promise to advise me. Thanks also to Bob for providing me with funding to visit Stanford in the summer of 1995, which led to my thesis proposal.

Special thanks to Teruko for becoming my advisor locally. Not only did she coordinate my efforts, but she also read many drafts, and advised me especially on the overall organization of the thesis. She was quite generous with her time.

I had many discussions about guards, and ALE in general, with Gerald Penn. He has a great deal of insight on this topic. Frank Pfenning provided me with initial advice as to how to implement the algorithm. The advice must have been sound, as the implementation changed little over time. I also had conversations about implementing grammars with Detmar Meurers and Dan Flickinger. Alon Lavie lead parsing discussions at CMU, where I presented my work.

Thanks to Lori Levin and Robert Levine for advising and editing, respectively, my paper on PVP fronting. This became the basis for one of the grammars in the thesis.

Thanks to the Catalyst project at CMU, including Eric Nyberg and Jaime Carbonell, for allowing me a great deal of freedom in my work arrangements.

For moral support, I have many people to thank. I offer my very best wishes to my primary support group of Pam Jordan, Sue Holm, and Carolyn Rosé. Krzysztof

Czuba and Enrique Torrejón have been among my most supportive colleagues at the LTI. Thanks to Krzysztof for being a great office mate. So many past and present Catalyst project members have been my good friends.

I also thank the LTI ABD group including (past and present members): Jade Goldstein, Rosie Jones, Paul Placeway, Yan Qu, Laura Tomokiyo, and Klaus Zechner. This group was enormously helpful to one another, and I hope it continues.

Many thanks to Marnie Tynen, Mary Alice Drusbasky and Linda Stagon for help with child care, especially during the summers. Also, many members of Shadyside Presbyterian Church have offered spiritual support, including Anne Camp, Sunday school teachers, and members of the women's Bible study. I looked forward to Sundays throughout this research.

Thanks to my boys, David, Daniel, and Luke, for just being themselves.

I reserve my final thanks for my husband Eric, for his graciousness while I have pursued my degree. His time, energy and absolute support have enabled me to spend numerous evenings and weekends focused on this work.

Abstract

In a linguistic theory, features constrain a well-formed constituent, such as a phrase or a sentence. Identifying such a constituent from an input can be cast as a constraint-solving problem. Delaying the resolution of certain constraints is an established method for solving goals in logic programming languages. This method can be adapted not only for linguistic problem solving in general, but specifically for constraints in the form of complex, typed feature structures. Constraints which wait upon more information are called guarded constraints.

This thesis generalizes delays for processing a linguistic theory by providing a specification for guarded constraints as descriptions of typed feature structures. I extend the description language, a shorthand for typed feature structures, described fully in Carpenter (1992). My work enables the successful evaluation of modern grammars as written, even with highly lexicalized constraints. The work includes a full description of the implementation of three relevant linguistic examples. These are the Japanese causative, German topicalization, and quantifier scoping in English.

By generalizing delays over feature structures, I make three primary contributions. First, I give the operational semantics of guarding on typed feature structures. I show how either rules can be guarded, or descriptions of feature structures themselves can be guarded. Descriptions constrain the choice of available goals during resolution. Second, I place the work in a linguistic context, presenting a variety of analyses for which delaying is a good approach. The work enables a uniform treatment of many phenomena, all of which are analyzed linguistically via argument sharing. Third, with guards, I extend ALE (Carpenter & Penn 1994), a logic programming system that can be used for parsing and generation of large-scale grammars. The work is language independent and independent of the underlying parsing or generation algorithm.

Contents

Chapter 1

Introduction

Feature based theories of grammar provide a concise formalism in which to express linguistic concepts and relationships among concepts. Bundles of features are especially useful during automatic processing of written text as a way of specifying both syntactic relations and semantic concepts. A specific set of feature values can pick out a particular semantic category, while syntactic relations are realized in the graph or tree structure that feature-value pairs describe. For example, the feature structure in example 1 commonly represents a third person singular verb. In this feature structure, the boldfaced words are *types*. Each feature structure in this notation has a type, and **sg** (singular) and **3rd** are *atomic types*, which are unique semantic concepts that have no further features. The words in small capitals such as NUMBER and PERSON are the *features*.

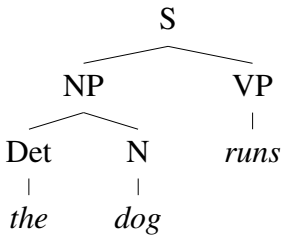
(1)
$$\begin{bmatrix} \textbf{verbform} \\ \text{NUMBER}:\textbf{sg} \\ \text{PERSON}:\textbf{3rd} \end{bmatrix}$$

The dog runs is a sentence whose grammatical categories such as subject and head verb are expressed in the feature structure in example 2. The variable index $\boxed{1}$ points to the part of the feature structure that contains referential index features for the subject *Dog*, including number and person. By appearing both under a grammatical category (SUBJECT) and a semantic category (RUNNER), the use of $\boxed{1}$ indicates that information is *shared* in two places the feature structure.

(2)
$$\begin{bmatrix} \textbf{sentence} & \\ \text{CATEGORY:} & \begin{bmatrix} \textbf{category} & \\ \text{SUBJECT:} & Dog\boxed{1}_{[3rd,sg]} \\ \text{HEAD:} & \textbf{verb} \end{bmatrix} \\ \text{CONTENT:} & \begin{bmatrix} \textbf{content} & \\ \text{RELATION:} & \textbf{run} \\ \text{RUNNER:} & \boxed{1} \end{bmatrix} \end{bmatrix}$$

Paths of features and values can be mapped to branches and nodes in a graph or a tree structure. If the value of the subject feature is mapped to NP and the head feature to VP, then the feature structure in example 2 corresponds to this more familiar tree diagram:

(3)



In order to use a feature-based formalism to full advantage, one can leave information *underspecified*. Feature values may be underspecified. This means that there is more than one possible value for a feature to take on and still be grammatical. In a typed system, types may be underspecified, which means that the subtype is unresolved. A typical example of underspecification is the ambiguous number value for English nouns which are alike in their singular and plural forms. Words of this kind include *fish* and *sheep*.

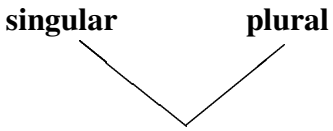
A simple feature structure showing the semantic features for the noun *sheep* might look as in example 4. The type **nom-object** refers to the fact that a sheep belongs to the class of things that are nominal objects.

(4)
$$\begin{bmatrix} \textbf{nom-object} & \\ \text{CONTENT:} & \begin{bmatrix} \textbf{content} & \\ \text{PERSON:} & \textbf{3rd} \\ \text{RELATION:} & \textbf{sheep} \end{bmatrix} \end{bmatrix}$$

The feature NUMBER is simply left unstated. We understand that the underspecified value of the number feature is simply the type **number**, which has subtypes

of either **singular** or **plural**. The subtypes inherit from the supertype in a simple type hierarchy, which shows the supertype at the bottom:

(5)



Re-stating the feature structure more explicitly, we have:

(6)
$$\begin{bmatrix} \textbf{nom-object} \\ \text{CONTENT}: \begin{bmatrix} \textbf{content} \\ \text{PERSON}:\textbf{3rd} \\ \text{NUMBER}:\textbf{number} \\ \text{RELATION}:\textbf{sheep} \end{bmatrix} \end{bmatrix}$$

Clues from other parts of the grammar could help to make the feature or type more specific when the feature structure gains information during natural language processing. We refer to these clues as constraints. In the sentence *The sheep graze*, the "filled-in" version of the feature structure has the subtype of **number** resolved to **plural** in 7:

(7)
$$\begin{bmatrix} \textbf{nom-object} \\ \text{CONTENT}: \begin{bmatrix} \textbf{content} \\ \text{PERSON}:\textbf{3rd} \\ \text{NUMBER}:\textbf{plural} \\ \text{RELATION}:\textbf{sheep} \end{bmatrix} \end{bmatrix}$$

The sentence *The sheep bleats* would likewise resolve to the singular form. The number for the noun is obtained via unification with the number form on the inflected verb. Typically, the number for the verb form is related to a basic, also underspecified feature structure for the verb's base form, by way of some morphological analysis. The benefit of such a representation is generality; the number of lexical entries is kept to a minimum, limiting lexical ambiguity. In this case ambiguity resolution may therefore be understood as further feature value refinement.

If the noun phrase (NP) *the sheep* is the subject of a sentence, information from the verb is the clue to further resolving the number of the subject. If the NP were the object of the sentence, as in *We saw the sheep*, the single lexical entry in 6 would still suffice. In fact, without other contextual knowledge, such as a prior sentence, the noun in object position would remain as either singular or plural. The strategy used to "fill in" the information in 7 is to wait until enough new information is gained from the verb or elsewhere to further instantiate the entry.

Technically speaking, waiting is *delaying* the resolution of the constraint on the number of the noun phrase. Delaying is a proven method for solving goals in programming languages such as Prolog, and can be adapted not only for linguistic problem solving in general, but specifically for constraints in the form of complex, typed feature structures. In the programming language community, constraints which wait upon more information are called *guarded constraints*.

This thesis generalizes delays for processing a linguistic theory by providing a specification for guarded constraints as descriptions of typed feature structures. I extend the description language, a shorthand for typed feature structures, described fully in Carpenter (1992). My work enables the successful evaluation of modern grammars as written, even with highly lexicalized constraints. The work includes a full description of the implementation of three relevant linguistic examples.

Delaying was introduced to the feature-based grammar community by Bouma and van Noord as a methodology for implementing linguistic constraints (1994). Johnson & Dörre (1995) also presented an approach with delaying for van Noord & Bouma's (1994) work, in the framework of Categorial Grammar (Ajdukiewicz 1935; Bar-Hillel 1953; Lambek 1958). While delays have been part of large-scale feature grammar processing systems such as CUF (Dörre & Dorna 1993), FUF (Elhadad & Robin 1992), and ConTroll (Götz *et al.* 1997; Götz & Meurers 1997; Götz & Meurers 1998), there has been no complete account of the link between constraint based linguistic theory and delays in practice. By providing this account, I show that delaying is a practical way for a parsing or generation system to successfully process highly constrained lexical entries, instances of which depend upon gaining information from various parts of the overall grammar.

1.1 Linguistic Background

A number of feature-based grammar formalisms have been used for computation, including Functional Unification Grammar (FUG) (Kay 1983; Kay 1985), PATR-

II (Shieber *et al.* 1983), Lexical-Functional Grammar (LFG) (Bresnan 1982b), Generalized Phrase Structure Grammar (GPSG) (Gazdar *et al.* 1985), and Head-Driven Phrase Structure Grammar (HPSG) (Pollard & Sag 1987; Pollard & Sag 1994). All of these formalisms have *unification* in common as a way to combine information and obtain a result. The focus here is not on unification per se but rather on various approaches to expressing linguistic knowledge in feature structures and, in turn, a methodology for assimilating that information during natural language processing.

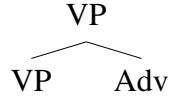
In a linguistic theory, constraints on feature structures restrict the possible values that a feature may take with respect to a well-formed constituent, such as a phrase or a sentence. Complex constraints may appear throughout the grammar, and have been adopted with greater frequency in the lexicon, hence the term "lexicalized" grammars. The term grammar is used here in a general sense to refer to the components of a theory, such as the lexicon, phrase and sentence rules, phonology, and context.

HPSG is a constraint-based formalism expressed fully in feature structure descriptions and relations over feature structures (Pollard & Sag 1987; Pollard & Sag 1994). Feature structures are assigned a type, and types are declared in an inheritance hierarchy. There are also conditions for determining which features are appropriate for a given type. One advantage to this theory's concise and uniform representation scheme is that there are no absolute boundaries between lexical, syntactic and semantic levels of representation. The caveat here is that the grammar writer is free to push the limits of the representation scheme, which traditionally includes a separate lexicon and constituent grammar. In much work of the past decade, operations on the arguments of syntactic heads, such as adjunction, fronting, extraposition, etc., have been expressed directly in the lexicon. These are part of the syntax in more traditional grammars.

An example of the blending of lexicon and syntax is the use of a lexical rule to add adverbial adjuncts onto a verb, first proposed for HPSG by Bouma and van Noord 1994, with reference to similar work by Miller (1992). The work has its roots in Categorial Grammar. Traditionally, adverbs such as *vandaag* in 8b are added onto a verb phrase (VP) node by way of a syntactic rule of adjunction (9).

(8) a. dat Arie Bob wil kussen
that Arie Bob wants to-kiss
Arie wants to kiss Bob.

b. dat Arie Bob vandaag wil kussen
that Arie Bob today wants to-kiss
Arie wants to kiss Bob today. (van Noord & Bouma 1994:12)

(9)


Lexical rules are used to build up lexicons from a set of basic entries. The rule in example 10 applies to the basic entry for a Dutch verbal complex such as *wil kussen*. The feature structure to the left of the arrow is the lexical entry for the verb complex, and the feature structure to the right is a lexical entry for a verb complex that has an adverbial adjunct. The notation $A \ominus B$ denotes the list A minus the elements in B, while $A \oplus B$ concatenates the lists A and B.

(10)
$$\begin{bmatrix} \textbf{verbal} \\ \text{SUBCAT}: \boxed{1} \oplus \boxed{2} \\ \text{SEM}: \boxed{4} \end{bmatrix} \longrightarrow \begin{bmatrix} \textbf{verbal} \\ \text{SUBCAT}: \boxed{1} \oplus \left\langle \begin{bmatrix} \textbf{adverbial} \\ \text{MOD}: \begin{bmatrix} \textbf{mod} \\ \text{ARG}: \boxed{4} \\ \text{VAL}: \boxed{3} \end{bmatrix} \end{bmatrix} \right\rangle \oplus \boxed{2} \\ \text{SEM}: \boxed{3} \end{bmatrix}$$

The basic verb entry has a list of subcategorized arguments, or SUBCAT list, consisting of a predicate $\boxed{1}$ plus a subject $\boxed{2}$. The SUBCAT list for a verb derived from an application of the rule consists of the predicate, plus an adjunct, shown as a feature structure of type **adverbial**, plus the subject. A verb, then, may be realized with or without adjoined adverbials, without a separate syntactic rule for adjunction. The entire process of adverbial adjunction is "located" in the lexicon. That the adverb can adjoin to the main verb or the auxiliary verb by different applications of the rule accounts for differences in scope of the adverb.

A lexical theory of *argument extraction* is another example of a lexicalized grammar. Extracted complements are complements that appear outside the argument position of a headed phrase, such as topicalized noun phrases (**Kim**, *Sandy likes*), and the Wh-constituents in questions (**Which book** *did you read?*). Constraint-based extraction enables the removal from the grammar of syntactically motivated features, such as traces, which are central to generative accounts of these phenomena (Ross 1967; Chomsky 1981) and also part of GPSG (Gazdar 1981).

Bouma *et al.*'s (2001) Argument Realization constraint on the subcategorization list of a word is shown as example 11. In this example, the feature COMPS refers to the *local* complements subcategorized for by the head. These are the complements that appear inside the phrase headed by the word. The COMPS plus the subject make up the SUBCAT list (not shown). The feature DEPENDENTS is an intermediate level of representation for the local (non-extracted) complements and those that have been extracted.

All subcategorized complements are pointed to by the index $\boxed{2}$. By describing the complements list as

$\boxed{2} \ominus list(\textbf{gap-synsem})$

the authors leave open the possibility that there are some extracted arguments, the list of elements of type **gap-synsem**. Extracted complements, if any, appear as being "removed" from the complements list in the lexical entry.

Note that the list of extracted complements, $list(\textbf{gap-synsem})$, could be empty.

(11)
$$\begin{bmatrix} \textbf{word} \\ \text{SUBJECT}:\boxed{1} \\ \text{COMPS}:\boxed{2} \ominus list(\textbf{gap-synsem}) \\ \text{DEPENDENTS}:\boxed{1} \oplus \boxed{2} \end{bmatrix}$$

With similar treatment, the semantic QSTORE and QRETRIEVAL features for quantifier scope and retrieval which originate from Montague semantics (Montague 1974; Cooper 1975; Cooper 1983) may be expressed as constraints in the lexicon (Pollard & Yoo 1997; Manning & Sag 1998).

1.2 Processing Background

We consider existing methods for solving constraints in NLP systems. In particular, the constraint resolver may be the parser itself, which decides among competing grammar rules, or it may be a separate module that works in conjunction with a traditional parser. Early HPSG grammar systems were modelled as general constraint resolvers (Franz 1990; Emele & Zajac 1990). Over time, researchers have found that interleaving constraint solving with traditional parsing methods, such as chart parsing, retains the advantages of bottom-up methods. This is preferable for obtaining fast, real-time processing. Other approaches for implementing constraint-based grammars include compiling lexical entries into constraints which are fired during runtime (Meurers & Minnen 1995; Meurers & Minnen

1996). Their approach for solving lexical constraints is constraint propagation. With this technique, many constraints at once provide information to each other until no more can be solved.

One solution to the problem of associating constraints of all types with particular feature structures is to adopt the strategy of delays used in logic programming languages. The notion of guarding a constraint dates from Prolog II (Colmerauer 1982). This technique is useful for processing terms that change information state dynamically. For this reason, guarded constraints allow highly lexicalized grammars to be processed.

The `when` predicate in SICStus Prolog (SICStus (1995)) is an example of a delaying predicate.

(12) `when`(+$Condition, Goal$)

A call to the predicate in 12 blocks *Goal* until the *Condition* is true. For example, `when(nonvar(X),append(X,Y,Z))` says that when the value of the term `X` is nonvariable, we can solve the goal `append(X,Y,Z)`. Else, the goal is blocked, and it will not be called if `X` is still variable. We are especially interested in statements such as `when(`*Condition*`,` *Goal*`)` as they apply to conditions on the arguments of subcategorization lists of heads, because these arguments may be shared across many structures, and information about them may be gained at different times. Furthermore, we would like for these arguments to be not single feature values or types but entire feature structures using a feature structure description language. These might be adverbial adjuncts such as the feature structure of type **adverbial** in 10 or an entire list of extracted complements such as the **gap-synsem** arguments in 11.

1.3 Outline of the Thesis

By generalizing delays over feature structures, I make three primary contributions. First, I give the operational semantics of guarding on typed feature structures. I show how either rules can be guarded, or descriptions of feature structures themselves can be guarded. If descriptions are guarded, SLD resolution can be used. Otherwise, descriptions constrain the choice of goal during resolution. The compilation of inequations as delays illustrates how the control of guarded constraints can be handed from the grammar writer to the program. Second, I place the work in a linguistic context, presenting a variety of linguistic analyses for which delaying is good approach. This enables a uniform treatment of a variety of phenomena,

for which underspecification is key. Third, I apply guarding in a system already in use by computational linguists writing large-scale grammars. My generalized description of guarding can be used in lexical rules, grammar rules, constraint descriptions, or definite clause procedures, and I provide examples of how this may be done. The work is language independent and independent of the underlying parsing or generation algorithm.

When coding up a linguistic theory for implementational purposes, one can express it in such a way that makes implementation more practical. It is possible to create HPSG grammars in a system such as ALE (Carpenter & Penn 1994) using phrase-structure rules, for example, while no such rules exist in the theory per se. Rather, schemata exist which are constraints on well-formed phrasal signs. The idea behind adding guards to a grammatical theory is that the theory can be implemented as closely as possible to the original specification. That is to say, if a constraint appears in the lexicon in the theory, then it can be coded in the lexicon in the implemented grammar as well. Simply put, the work described in this thesis makes it not only easier but possible to implement lexical constraints directly in the lexicon.

The writer of the implementational grammar must have some instruction as to where to put the guards. Ideally, they would be automatically derived by inspection from a grammar without guards. I have written them by hand. The places where guards are necessary is where there is argument sharing projected from a lexical head. This happens with argument raising in general, and specifically where the arguments of a head are underspecified but expected to gain information from other sources. This occurs in the expression of traditionally syntactic operations, such as complement extraction, in the lexicon. In the latter case, information is needed about as yet uninstantiated arguments in order for the constraints to successfully apply. Several examples of such cases are given in chapter two.

A linguist would need to look first for shared arguments in a lexical entry, and second, for constraints on variable arguments. Furthermore, the linguist would need to know how to express guarding on these arguments, or, which portions of the feature structures to guard. In my example cases I have waited until local features are instantiated. This is either syntactic category or semantic content, in the case of verb raising, or the quantifier store, in the case of constraints on quantifiers. Therefore, it is non-trivial to write guarded rules, but a pattern does emerge. Guards are compiled automatically for inequations (the case of the binding theory). In chapter 5, I show how guarded descriptions can be added into the theory of feature structures itself, so that the linguist can write the original theory with guards. In this case, the implementation directly follows the theory.

The thesis is organized as follows. In chapter two, I review the relevant linguistic literature, providing a wide range of linguistic data for which the technique of delaying will prove useful in implementation. I show an example of a problem which is encountered when attempting to evaluate underspecified feature structures during processing using traditional means. In chapter three, I review the field of Constraint Logic Programming (CLP) and its application in Computational Linguistics. In chapter four, I lay out the framework of feature structures, and review the existing CLP systems for processing feature based grammars. Next, in chapter five, I give the operational semantics of delaying on typed feature structure descriptions. I show how delaying is used in the case of verb raising by auxiliary in German. I discuss my approach to inequations as an instance of guarded descriptions. In chapter six, I use the binding theory as a linguistic test case for the system. I present this example for Japanese. I also present quantifier retrieval in English as an example of guarding on types. I evaluate all examples with respect to the specification, the grammar and lexicon, and the parsing algorithm. Finally, in chapter seven, I draw conclusions about this work. The thesis is extensible to a wide range of linguistic analyses, and is language independent. The resolution algorithm is abstracted away from the parsing architecture, providing a practical advantage for implementation efforts.

Chapter 2

Describing Linguistic Objects

I begin by presenting a cross section of linguistic data which analyzes arguments as being shared by more than one head. For example, in the case of raising structures, a verb like *seem* in English might share an NP subject with another verb, a predicate for which it subcategorizes. A variety of contexts include argument raising by Germanic and Romance auxiliaries, sharing by semantic predicates in Chicheŵa and Japanese, lexical rules for syntactic operations such as the passive and complement extraction, quantifier raising, and argument sharing in word order domains. These analyses are unified by the fact that arguments are shared across more than one structure. This may be a semantic predicate, a subcategorization frame, a word order domain, or a nonlocal set (such as the SLASH value or the quantifier store).

Argument sharing is relevant in processing because one head may rely on information about an argument that is provided only within the context of the other head. And so, the order of argument instantiation becomes relevant during natural language processing. In this chapter I address the impact of structure sharing on evaluation. Our interest in argument sharing is established. I use the case of raising by the German auxiliary *werden* as a specific example. The question we consider in particular is whether typical processing methods have been sufficient for cases of argument sharing.

2.1 Raising by Auxiliary

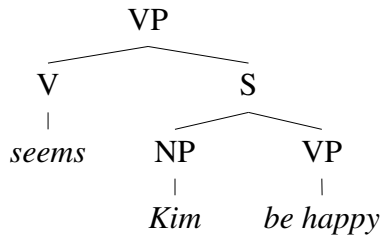
Raising verbs are a class of verbs that subcategorize for an argument that is assigned a semantic role by an embedded predicate. The term *raising* comes

from transformational grammar (see e.g. Chomsky 1973; Soames & Perlmutter 1979; Chomsky 1981). We look here at raising-to-subject verbs, because we will next consider auxiliary verbs, which can be analyzed as a particular case of these. Raising-to-subject verbs "raise" the subject of the embedded clause up to the subject position of the main clause. The main clause verb does not assign a semantic role to the subject.[1] Example 13 is an example of this from English.

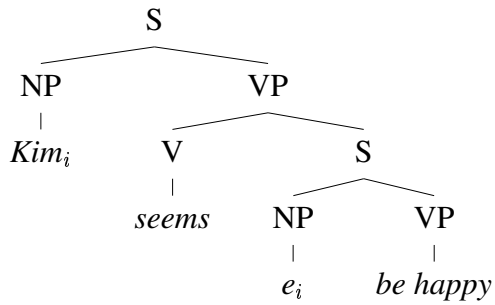
(13) Kim seems to be happy.

Chomsky described raising as a transformation from an underlying form to a surface form. For example, the underlying representation may be the tree structure in 14 and the surface representation that in 15. The raised subject *Kim* is the grammatical subject of *seems* in 15. But *seems* does not assign a semantic role to *Kim*; *Kim* is not "seeming" anything in any sense, though *Kim* is the thematic argument of the predicate *be happy*; Kim may be the theme or experiencer, for example.

(14)



(15)



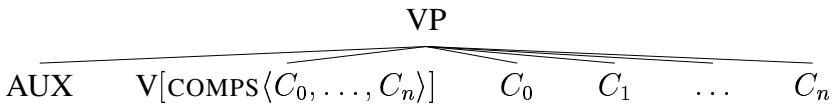
Structure sharing within a single graph is another framework which can be used to describe raising (Bresnan 1982a). For an explanation of raising phenomena in HPSG, the reader is referred to Pollard & Sag (1994:chapter 3). In their

[1]This contrasts with *Equi* verbs, which assign semantic roles to all of their arguments, but have an argument in common with the embedded predicate. e.g. *Kim tries to leave.* Kim is both the one who tries and the one who leaves.

account of example 13, there is not movement of NP constituents from one clause to another, but rather a structure sharing of the subject NP by the raising verb and the head of its verbal complement. Structure sharing in this example means that the subject for each of the two verbs is one and the same object. Example 16 shows a simplified lexical entry for *seem*. Structure sharing is marked in the example by the index [2]. If the feature structure were drawn as a graph, structure sharing would be characterized by two arcs entering the same node.

(16)
$$\begin{bmatrix} \textbf{local} \\ \text{CAT:} \begin{bmatrix} \textbf{category} \\ \text{HEAD:}\textbf{verb} \\ \text{COMPS:}\langle \begin{bmatrix} \textbf{vp} \\ \text{VFORM:}\textbf{inf} \\ \text{SUBJ:}\langle \boxed{2}\textbf{np} \rangle \end{bmatrix} : \boxed{1} \rangle \\ \text{SUBJ:}\langle \boxed{2} \rangle \end{bmatrix} \\ \text{CONTENT:} \begin{bmatrix} \textbf{seem} \\ \text{SOA-ARG:}\boxed{1} \end{bmatrix} \end{bmatrix}$$

The work of Johnson (1986) and Hinrichs & Nakazawa (1989), Baker (1999), and Abeillé & Godard (1994) has analyzed auxiliary verbs as raising the arguments of the verbs for which they subcategorize in German and French. Monachesi (1993a); Monachesi (1993b) has a similar analysis for Italian.[2] The auxiliary raises all of the arguments of the verb for which is subcategorizes. In each case, the structural result is a flat tree structure (17) as opposed to a binary branching structure. A template for a lexical entry for an auxiliary which raises the arguments of a verb is shown in example 18. This template applies across different languages. This is in contrast to a more traditional analysis, in which an auxiliary subcategorizes for a full verb phrase as a single argument (e.g. Gazdar *et al.* 1982 for English, Pollock 1989 for French).

(17)


[2]The use of argument composition continues in other languages as well, for Korean, Dutch, and Polish.

(18)
$$\begin{bmatrix} \textbf{auxiliary} \\ \text{HEAD:} \begin{bmatrix} \textbf{verb} \\ \text{VFORM:}\textbf{base} \\ \text{AUX:}+ \end{bmatrix} \\ \text{SUBJ:}\boxed{1}\langle \textbf{np} \rangle \\ \text{COMPS:}\boxed{2} \oplus \left\langle \begin{bmatrix} \text{HEAD:} \begin{bmatrix} \textbf{verb} \\ \text{VFORM:}\textbf{base} \end{bmatrix} \\ \text{COMPS:}\boxed{2} \\ \text{SUBJ:}\boxed{1} \end{bmatrix} \right\rangle \end{bmatrix}$$

In feature-based analyses of examples from German, French and Romance clitics (19b through 24b) below, the lexical entry for raising auxiliary is notable because the arguments that the auxiliary verb shares with the subcategorized verb are not instantiated in the lexicon. In fact, no information about the individual members of the verb's valence list is available from this entry alone. The information must be combined with information provided by the lexical entry for the head verb and from the specific arguments in an instance of verb usage.

2.1.1 German

The evidence for a flat tree structure for German sentences stems primarily from constituent scrambling. A verb's arguments are not always adjacent to the main verb when an auxiliary is present. The auxiliary verb appears between nominal constituents and the main verb in the so-called "modal flip" construction, which is example 19c.

(19) a. Peter wird das Buch lesen.
Peter will the book read.
Peter will read the book.

b. Peter das Buch lesen können wird
Peter the book read be-able-to will
Peter will be able to read the book.

c. (dass) Peter das Buch **wird** lesen können
(that) Peter[nom] the book will read be-able-to
Peter will be able to read the book.

An instantiated version of the template in example 18 for the auxiliary *werden* (19a) is simplified in example 20. As arguments have been encountered in the sentence, we see that the verb's arguments include an accusative nominal argument *Buch*, which is subcategorized for lexically both by the main verb *lesen* and by the auxiliary *werden*.

(20)
$$\left[\begin{array}{l}\textbf{werden}\\ \text{HEAD:}\left[\begin{array}{l}\textbf{verb}\\ \text{VFORM:}\textbf{base}\\ \text{AUX:}\textbf{+}\end{array}\right]\\ \text{SUBJ:}\boxed{1}\langle NP[nom]Peter\rangle\\ \text{COMPS:}\boxed{2}NP[acc]Buch + \left\langle\left[\begin{array}{l}\textbf{verb}\\ \text{HEAD:}\left[\begin{array}{l}\textbf{lesen}\\ \text{VFORM:}\textbf{base}\end{array}\right]\\ \text{COMPS:}\boxed{2}\\ \text{SUBJ:}\boxed{1}\\ \text{NPCOMP:}\textbf{-}\end{array}\right]\right\rangle\end{array}\right]$$

2.1.2 French

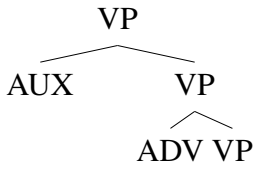
In the case of French, Abeillé and Godard argue for raising of all complements by auxiliary and against a VP constituent, which results in the flat structure as shown in example 17. One argument is that manner adverbs may appear between the auxiliary and the main verb, yet do not have properties that allow them to be fronted.

(21) a. Jean a attentivement écouté son professeur.
Jean has attentively listened-to his teacher
Jean has attentively listened to his teacher.

b. ??Attentivement, Jean a écouté son professeur.
Attentively, Jean has listened-to his teacher
Attentively, Jean has listened to his teacher. (Abeillé & Godard 1994:9a,b)

If the adverb *attentivement* were a sister node to a VP node *écouté son professeur*, one would expect that the adverb would be free to be topicalized, but it is not. More specifically, arguments are presented against the structures for verb phrase in 22a and 23a:

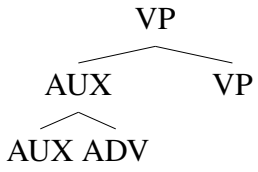
(22) a.



b. ∗Attentivement prendre des notes ne suffit pas à faire un bon étudiant
Attentively taking of notes not suffice [neg] to make a good student
To attentively take notes is not sufficient to make a good student. (Abeillé & Godard 1994:13a)

If the adverb *attentivement* were assumed to be part of an embedded VP node in 22a, then one would expect manner adverbs to occur generally in VP-initial position, but they cannot.

(23) a.



b. Jean a attentivement écouté son professeur et pris des notes.
Jean has attentively listened-to his professor and taken of notes.
Jean has attentively listed to his teacher and taken notes.

> This can only mean that Jean listend to his teacher with attention and took notes; it cannot mean that Jean attentively took notes.

(Abeillé & Godard 1994:14a)

If the adverb were assumed to be adjoined to the auxiliary, separately from an embedded VP, as in 23a, then one would expect it to take wide scope over a sequence of participles and their complements, but it takes narrow scope in this case.

Abeillé and Godard, in the course of establishing argument composition by French auxiliaries, also contest the assumption of a separate VP constituent for participles in French on the basis of the of weakness of standard constituency tests for this type of VP. These are tests such as pronominalization of the VP, VP deletion (also called null complement anaphora), etc. These have been provided as evidence for a VP node for French infinitival verb complements in the literature (Emonds 1978; Fradin 1993).

2.1.3 Romance Clitics

"Clitic climbing" in Romance languages motivates argument sharing by an auxiliary and a subcategorized verb in Italian (Monachesi 1993b),(Monachesi 1993a) and French (Miller & Sag 1995). The basic argument is that, because a clitic can attach to either an auxiliary or the subcategorized verb, that the clitic must be an argument of both. More will be said in section 2.3.2 about a lexical treatment of this phenomenon.

(24) a. Anna vuole comprarlo.
Anna wants-to buy-clitic[acc]
Anna wants to buy it.

b. Anna lo vuole comprare.
Anna clitic[acc] wants-to buy
Anna wants to buy it. (Monachesi 1998:9a,b)

The tree structure for Monachesi's analysis of "restructuring" verbs as in 24b is shown in 25. These verbs include not only modal verbs but also aspectual verbs (Rizzi 1982).

(25) (Monachesi 1998:18)

2.2 Morphology

In this section I examine morphological valence-changing operations. In a more traditional view, these add or subtract an argument or arguments from the valence list of a head (Williams 1981; Comrie 1981; Marantz 1984). Taking this additive view of the causative, for example, the presence of the causative morpheme adds an external subject argument to the embedded predicate to yield a new causative predicate with one more argument than the embedded predicate, or a valency that is one higher than that of the predicate. The causee is expressed as being "demoted" from e.g. direct object to indirect object along a hierarchy of grammatical relations (26).

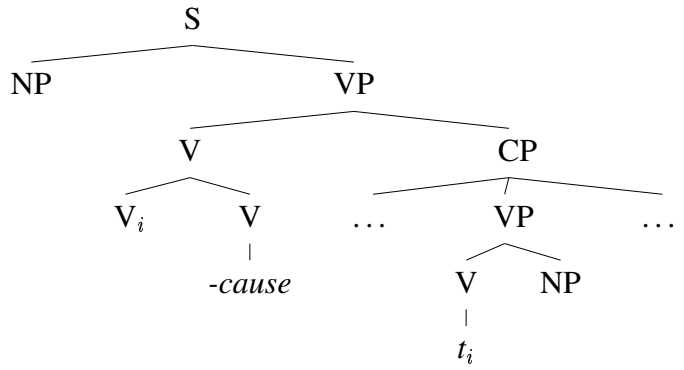
(26) subject > direct object > indirect object > oblique object
Keenan & Comrie (1977)

In the analyses I focus on here, a semantic argument of both the causative predicate and the embedded predicate is shared, linking the two in a single representation. By this linking, the causee may inherit properties from either subcategorizing verb, resulting in the different syntactic positions or semantic interpretations that it may take. Furthermore, this is done as a constraint rather than an operation. We look at analyses of causatives in Chichewa and Japanese. A third, related case is an example of the passive in German, which can be viewed as a valence reducing operation. In the structure sharing version of the passive, information about a nominal is shared.

2.2.1 Alsina's analysis of Chichewa

Alsina's (1992) analysis of Chichewa is a semantic treatment of the causative, in contrast with the syntactic incorporation accounts of Baker (1988) and Li (1990). In an incorporation account, a morpheme at the level of V_0 in X-bar theory (a lexical verb) (Jackendoff 1977) adjoins to the causative affix, which is itself a verb which subcategorizes for a CP (a complement phrase), to form a new verb. A tree diagram is shown in (27). The verb V_i which adjoins to the causative morpheme is moved from a position from within the CP. I have left out the details of the structure of the CP here because it is not central to the discussion, and also because there are a few different scenarios.

(27)



An example of a causative verb is *i-na-phík-íts-a* 'make cook' in Chichewa in 28a. In a theory in which verbs assign case to their arguments based on syntactic position, such as the incorporation accounts, this verb can assign a thematic or semantic role only to the external subject *Nŭngu* 'porcupine.' This is because it is

presumed that the embedded verb 'cook' has case marked its predicate arguments *kadzīdzi* 'owl' and *maûngu* 'pumpkins' in the complement clause.[3]

(28) a. Nŭngu i-na-phík-íts-a kadzīdzi maûngu
9 porcupine 9 s-PS-cook-CST-FV 1a owl 6 pumpkins
The porcupine made the owl cook the pumpkins.

b. Nŭngu i-na-phík-íts-a maûngu kwá kádzīdzi
9 porcupine 9 s-PS-cook-CST-FV 6 pumpkins to 1a owl
The porcupine had the pumpkins cooked by the owl.

Alsina argues instead that the patient of a causative predicate shares a thematic role with one of the arguments of the embedded predicate, without deriving one from the other. Alsina explains his theory using argument sharing as it constrains the surface forms of the syntax. He proposes a three-place causative relation for Chichewa and other Bantu languages. The three arguments are a causer, a causee, and an event. The causer is an agent which acts on the second argument, a causee, which has the semantic role of patient; the patient brings about the third argument, an event; and the patient is itself an argument of the event. A diagram of the relationship is shown in (29). *ag* refers to the agent and *pt* to the patient of the causative verb.

(29) $CAUSE\langle ag\ pt\ \overbrace{PRED\langle \ldots\theta\ldots\rangle}^{caused\ event}\rangle$
$\underline{\uparrow \qquad\qquad \uparrow}$

In (29) some thematic role θ within the embedded predicate is linked, or shared, with the patient of the causative predicate, shown by the two up arrows (↑). This is in contrast to syntactic accounts, in which the causing predicate has two arguments, an agent and a predicate, and no argument sharing. The theory correctly predicts that the causee of the causative verb can be linked with more than one argument of the embedded predicate. These can have different syntactic

[3]Alsina explains his diacritics: Tones and vowel length are marked in the Chichewa sentences as follows: long vowels may be low ¯, high ", rising ˇ, and falling ˆ, and short vowels are either high ', or low, which is unmarked. Each noun in Bantu belongs to one of eighteen noun classes, denoted in the glosses by Arabic numerals. Roman numeral I designates first person singular. The following abbreviations are used:

S	subject	PS	past	CST	causative	FV	final vowel
O	object	INF	infinitive	PAS	passive	FOC	focus
PR	present	ERG	ergative	ACC	accusative	REL	relative

positions. Alsina's analysis explains the fact that the causee may be either the direct object of the verb (as in 28a) or an oblique object of a preposition (28b). In (28a) the causee is *kadzīdzi* 'owl.' The causee is expressed syntactically as an object of the causative predicate. In this case, the causee 'owl' shares a thematic role with the agent of the embedded predicate 'cook.' In example 28b, the causee is the 'pumpkins.' The causee here is an oblique, which is an intermediary who carries out the action. In this sentence, the causee shares its role with the patient of 'cook.'

2.2.2 The Japanese Causative

Manning *et al.* (1999) present an analysis of Japanese causatives which is compatible with Alsina's Chicheŵa analysis. As in Alsina (1992), the causee argument of the causative predicate is linked to the arguments of the embedded predicate. Manning *et al.*'s presentation goes further, however, in that it also describes the morphosyntax of the causative construction. Their analysis casts the causative as a lexical constraint, and the causative verb as a lexical entry with a simple valence list.

Japanese causatives are formed by adding the causative morpheme *-(s)ase* to a verb stem, as shown in example 30. The subject of the causative is marked with the nominative particle *ga*, and the causee is marked with the dative particle *ni*. The accusative particle *o* is optional if the verb stem is intransitive.

(30) Taroo ga Ziroo ni Hanako o tazune-sase-ta
Taroo NOM Ziroo DAT Hanako ACC visit-CAUSE-PAST
Taroo made/had Ziroo visit Hanako.

Traditional transformational analyses of the Japanese causative, including Kuroda (1965) and Shibatani (1976), present the causative as a verb which subcategorizes for an NP argument, which is itself an embedded sentence, or S node. Gunji (1987), in the framework of a unification-based Japanese grammar, argues that instead, the causative morpheme subcategorizes for VP. Whether the embedded argument is a VP or S node is non-crucial in Manning *et al.*'s (1999) approach, since the causative morpheme does not subcategorize for any arguments. Instead, it constrains the entire causative verbal form. This is a crucial departure from previous analyses.[4] The reader is referred to (Manning *et al.* 1999) for a complete

[4]The authors note the similarity of their work in some respects to lexical Government-Binding accounts, including Miyagawa (1980) and Kitagawa (1986).

account of the motivation for a lexical rather than a syntactic approach. The authors explain that Japanese causatives behave as a single clause with respect to case, word order and other properties, following the Lexical Integrity Principle of Bresnan & Mchombo (1995). Construal processes such as honorification, binding, and quantifier scoping, which have been used to motivate a complex syntactic structure, are used here to motivate instead a complex argument structure, which can be located in the lexicon.

In Manning *et al.* (1999) the causative morpheme *-sase* is a lexical constraint (in the style of derivational morphology following Riehemann 1993) which constrains both the content of the causative verb as a whole, and the features of the stem of the embedded verb. Lexical constraints, as they are used in modern analyses, rely greatly on a fine-grained type inheritance hierarchy for the lexicon. The constraint is a partial feature structure, as shown in example 31. Following Pollard & Sag (1994) and Levine & Green (1999), we assume that fully-specified feature structures correspond with linguistic objects in the theory, while constraints on members of a class of objects only contain enough feature values to distinguish the particular class. The constraints in the theory are referred to either as partial feature structures, or feature structure descriptions. Alternatively, the term *AVM*, for attribute value matrix, is used for the familiar feature-value pair notation which gives partial information about a class of objects.

(31)
$$\left[\begin{array}{l}\textbf{causative-stem}\\ \text{PHONOLOGY}:F_{sase}(\boxed{1})\\ \text{ARGUMENT-STRUCTURE}:\langle NP_i, NP_j, \ldots\rangle\\ \text{CONTENT}:\left[\begin{array}{l}\textbf{cause-relation}\\ \text{ACTOR}:i\\ \text{UNDERGOER}:j\\ \text{EFFECT}:\boxed{3}\end{array}\right]\\ \text{LEXICAL-DAUGHTER}:\left[\begin{array}{l}\textbf{verb-stem}\\ \text{PHONOLOGY}:\boxed{1}\\ \text{CONTENT}:\boxed{3}\end{array}\right]\end{array}\right]$$

The causative has three semantic roles, an ACTOR, an UNDERGOER, and an EFFECT. The causative morpheme inherits some constraints also from the types related to **actor** and **undergoer** verbs. The embedded verb stem is considered a *lexical daughter* of the causative morpheme. This means that it is the basic stem from which the morphologically derived sign is derived. The content of the

embedded verb stem is co-indexed with the EFFECT of the causative predicate. The actor and undergoer are co-indexed with the semantic indices (*i, j*) of the nominal arguments of the causative verb stem.

The feature structure for the causative form of the simple transitive verb *tazune* ('visit') is shown as example 32. The combination of the root plus the phonological suffix *-sase* is done in the phonology. *-sase* is part of a phonological combination rule and has no constraints associated with it independently. The phonological combination rule used to derive the phonology of the mother is the function F_{sase}, which yields $X + sase$, if X is vowel-final, and $X + ase$ otherwise.

(32)
$$\begin{bmatrix} \textbf{verb} \\ \text{PHON:}\langle tazune + sase \rangle \\ \text{SUBJ:}\langle \boxed{1}\textbf{np}[N]_i \rangle \\ \text{COMPS:}\langle \boxed{2}\textbf{np}[D]_j, \boxed{3}\textbf{np}[A]_k \rangle \\ \text{ARG-S:}\langle \boxed{1}, \boxed{2}, \boxed{3} \rangle \\ \text{CONTENT:}cause_3(i, j, visit(j, k)) \end{bmatrix}$$

In example 32, the CONTENT of the subcategorized verb, *visit(j,k)*, is embedded as the third argument of the three-place *cause* predicate, *cause$_3$(i,j,visit(j,k))*. The complement object of the embedded verb *tazune*, indexed with [3], is *structure shared* with the second object of the causative verb *tazunesase*. This highly lexicalized analysis of the Japanese causative has been proposed because it enables the argument structure of the embedded predicate to be the site of adverb scoping. It also predicts that the lower object may be subject to Principle A of the binding theory in either clause. Principle A says roughly that anaphors must be bound in their immediate clause (see Chomsky 1986, Pollard & Sag 1994, chapter 6). *Taroo* is the first argument on the ARG-S list of the causative verb, and *Ziroo* is the first argument on the ARG-S list of the verb stem. In this case, the binding of the reflexive *zibun-zisin* is predicted to be ambiguously either the causer *Taroo* or the causee *Ziroo* in example 33. I will re-visit these facts in chapter 6, where I present an implementation of this analysis.

(33) Taroo ga Ziroo ni zibun-zisin o hihan-sase-ta
Taroo$_i$ NOM Ziroo$_j$ DAT himself$_i$/j ACC criticize-CAUSE-PAST
Taroo made/had Ziroo criticize him$_i$/himself$_j$.

2.2.3 The German Passive

We now consider the passive, which, like the causative, is traditionally viewed as morphological valence-changing operation. First we mention *lexical rules*, since they are prominent in many analyses. Lexical rules apply to lexical entries create additional entries in the lexicon. We may think of them as unary re-writing rules, or as templates which get filled in with information from an "input" word, or as operations which create new lexical entries from old ones. Pollard & Sag (1987); Pollard & Sag (1994) speak of lexical rules as "rules of inference" and state that

> Mathematically, we think of the base forms and lexical rules as a signature from which the full lexicon is generated as a free algebra. ...This algebraic perspective, which is netrual between the declarative and procedural interpretations, lends much conceptual clarity to the analysis of word formation; e.g. inflection, derivation, and compounding correspond to unary sort-preserving, unary sort-changing, and binary algebra operations respectively (Pollard & Sag 1987: chapter 8, note 11).

Whether the forms which are the "outputs" to lexical rules are derived or declared in the lexicon, we are challenged when the arguments in the "inputs" are themselves underspecified. For example, it is not possible to extract one complement out of a list of members before the members of the list are known, as in the case of raising by auxiliary.

In the German personal passive, as in English, the accusative direct object of the active voice is realized as the subject of the passive:

(34) a. Otto[nom] schenkt seinem Neffen[dat] ein Klavier[acc].
Otto gives to-his nephew a piano
Otto gives piano to his nephew.

b. Ein Klavier[nom] wird seinem Neffen[dat] geschenkt.
a piano is to-his nephew given.
A piano is given to his nephew. (Pollard 1994:1a,b)

The main verb is *schenken* 'give' and the passive participle is *geshenkt* 'given.' In the view of passive as lexical rule (Pollard & Sag 1987), the active verb in the input to the rule subcategorizes for a subject and an object. This is shown here in example 35. The semantic indices of these are $\boxed{4}$ and $\boxed{3}$, respectively. On the subcategorization list of the output to the rule, the subject has been dropped and it

has been reassigned as the object of the optional prepositional phrase headed by *von* 'by' (PP[VON]). The object becomes the subject.

(35) Passive Lexical Rule

$$\begin{bmatrix} \textbf{base-and-trans} \\ \text{PHON:}\boxed{1} \\ \text{PAST-PART:}\boxed{2} \\ \text{SYN:LOC:SUBCAT:}\langle \ldots, NP_{\boxed{3}}, NP_{\boxed{4}} \rangle \\ \text{SEM:CONT:}\boxed{5} \end{bmatrix} \longrightarrow$$

$$\begin{bmatrix} \textbf{base-and-trans} \\ \text{PHON:}f_{psp}(\boxed{1}, \boxed{2}) \\ \text{SYN:LOC:SUBCAT:}\langle (PP[VON]_{\boxed{4}}), \ldots, NP_{\boxed{3}}, \rangle \\ \text{SEM:CONT:}\boxed{5} \end{bmatrix}$$

(36) schenken $\longrightarrow$ geschenkt

We consider next the analysis of Kathol (1994), and the related analysis of Pollard (1994), in which the German passive may be represented with a lexical entry for a raising auxiliary. The analysis in example 37 differs from the operative view of passive (e.g. the lexical rule in 35) in that the relationship between the arguments of the active and passive verb is wholly contained within the lexical argument for the auxiliary. This obviates the need for a separate rule that mediates the arguments, though there is still place for a rule of morphological inflection.

(37)
$$\begin{bmatrix} \textbf{werden} \\ \text{SUBJ:}\langle \textbf{np}[NOM]_{\boxed{1}} \rangle \\ \text{COMPS:}\boxed{2} + \langle \begin{bmatrix} \textbf{verb} \\ \text{HEAD:} \begin{bmatrix} \textbf{vform} \\ \text{VFORM:}\textbf{part ii} \end{bmatrix} \\ \text{COMPS:}\textbf{np}[ACC]_{\boxed{1}} \oplus \boxed{2} \\ \text{SUBJ:}\langle \textbf{np} \rangle \end{bmatrix} \rangle \end{bmatrix}$$

For the noun which is subject of the auxiliary and object of the embedded verb, the semantic information is shared by the tag $\boxed{1}$. Sharing semantic rather than syntactic information enables the case of that noun to be realized in the nominative case as the subject of the auxiliary, or as an accusative object as the direct object

of the subcategorize verb. The other complements of the subcategorized verb (indexed with 2) are shared by the auxiliary. The auxiliary does not know, and indeed does not need to know, exactly which verb it is that is in the passive form, nor what its arguments are.

2.3 Complement Extraction and Romance Clitics

There are (at least) two approaches to the encoding of operations on arguments. We may encode them as lexical rules (Pollard & Sag 1987, Flickinger 1987) or define them declaratively in terms of constraints (Riehemann 1993), as shown in the analysis of the Japanese causative in section 2.2.2. In the following examples I show lexical rules, but note the theoretical equivalence of the two approaches.

2.3.1 Complement Extraction

We consider complement extraction by lexical rule (CELR), which has been proposed in Pollard & Sag (1994) and adopted freely for other lexical operations in HPSG, including clitic extraction, as we will show in the next section. Extraction in the framework of "HPSG III," that is, the version of HPSG in chapter 9 of Pollard & Sag (1994), is handled without empty categories. The departure from the use of traces to indicate the deep structure position of a "moved" element, standard in transformational accounts of extraction, is motivated by the psycholinguistic work of Pickering & Barry (1991). The authors seek to explain Pickering and Barry's data, which suggests that comprehension of extracted elements is completed during human sentence processing not when a trace position for that element is reached, but rather when the head which subcategorizes for the extracted element is encountered.

In the CELR, a lexical entry is created for a head with one of its arguments extracted. The extracted argument is found in SLASH, the position for extraction (Gazdar *et al.* (1985)). A simplified version of the rule is given in example 38. In its most general conception, the CELR can extract any argument from the subcategorization list of the verb and put it into SLASH, e.g. *Kim, Sandy likes*; *I wonder who Sandy loves*; *On Kim, Sandy depends*.

(38) Simplified Complement Extraction Lexical Rule:

$$\left[\begin{array}{l}\textbf{synsem}\\ \text{LOCAL}:\left[\begin{array}{l}\textbf{local}\\ \text{CAT}:\left[\begin{array}{l}\textbf{category}\\ \text{COMPS}:\langle \ldots, \boxed{1}, \ldots \rangle\end{array}\right]\end{array}\right]\end{array}\right]$$

$$\longmapsto$$

$$\left[\begin{array}{l}\textbf{synsem}\\ \text{LOCAL}:\left[\begin{array}{l}\textbf{local}\\ \text{CAT}:\left[\begin{array}{l}\textbf{category}\\ \text{COMPS}:\langle \ldots \ldots \rangle\end{array}\right]\end{array}\right]\\ \text{NONLOCAL}:\left[\begin{array}{l}\textbf{nonlocal}\\ \text{SLASH}:\{\boxed{1}\}\end{array}\right]\end{array}\right]$$

The case of *Partial Verb Phrase Fronting*, or PVP fronting, in German and Dutch, is one example of the operation of the CELR on an input with uninstantiated arguments (Hinrichs & Nakazawa (1994), Nerbonne (1994), Baker (1994), van Noord & Bouma (1995), and others). A PVP is a verb phrase which may include some number, but not necessarily all, of the verb's arguments, as shown in examples 39 and 40. This variety of verb phrase appears in topicalized position:

(39) a. **Das Märchen erzählen** wird er ihr.
The fairy-tale[acc] tell will he[nom] to-her[dat]
He will tell her the fairy-tale.

b. **Ihr erzählen** wird er das Märchen.
to-her[dat] tell will he[nom] the fairy-tale[acc
He will tell her the fairy-tale.

(40) **Das Examen bestehen** wird er können.
The exam[acc] pass will he[nom] be-able-to
He will be able to pass the exam.

The PVP takes as input a lexical entry for auxiliary, such as the one shown in example 18, and outputs a similar entry with the subcategorized verb appearing as the head of a partial or complete verb phrase in the SLASH feature of the auxiliary. I show the SYNSEM values of the resulting auxiliary in example 41. "synsem" stands for features pertaining to syntax and semantics.

(41)
$$\left[\begin{array}{l}\textbf{synsem}\\ \text{LOCAL}:\left[\begin{array}{l}\textbf{local}\\ \text{CAT}:\left[\begin{array}{l}\textbf{category}\\ \text{HEAD}:\left[\begin{array}{l}\textbf{verb}\\ \text{AUX}:\textbf{+}\end{array}\right]\\ \text{COMPS}:\boxed{2}\end{array}\right]\end{array}\right]\\ \text{NONLOCAL}:\left[\begin{array}{l}\textbf{nonlocal}\\ \text{SLASH}:\left\{\left[\begin{array}{l}\textbf{vp}\\ \text{HEAD}:\left[\begin{array}{l}\textbf{verb}\\ \text{VFORM}:\textbf{bse}\end{array}\right]\\ \text{COMPS}:\boxed{2}\end{array}\right]\right\}\end{array}\right]\end{array}\right]$$

Any arguments "left behind" remain on the COMPS lists of the slashed VP and of the auxiliary.[5] The PVP case is interesting for this reason: If an auxiliary verb is analyzed as raising the complements of the verb for which it subcategorizes, then the lexical entry for the auxiliary simply leaves the verb's, and hence its own, arguments unspecified, since no information about them is available. Therefore, "unknown" arguments appear in the input and output to the PVP lexical rule. This results in a potentially infinite number of lexical instances, since without further restrictions, the rule can keep applying to its own output. This may be a rather unpleasant consequence in implementations of this phenomenon, because it leads to non-termination and/or unruly lexicons. Delaying the application of lexical rules is one solution to the problems of nontermination or unneeded lexical entries.

2.3.2 Romance Clitics

The CELR is closely tied to the lexical handling of clitics in Romance languages as described in a series of papers including Miller (1992), Sag & Godard (1993), and Abeillé *et al.* (1998). The following example, from Miller & Sag (1995) illustrates how the simple case of clitic extraction works.

(42) a. Le garçon à qui Marie croit que Jean-Paul parle.
The boy to whom Marie thinks that Jean-Paul speaks.

b. Jean-Paul lui parle
Jean-Paul speaks to him.

[5]In the analysis of Hinrichs & Nakazawa (1994), the fronted VP is always full, and any "missing" complements are slashed back to the main clause.

First, the CELR is applied to the base form *parler* ('speak') to create a form of that verb in which a complement, *le garçon* in example 42a, is extracted. Operating on that form is a lexical rule for clitic affixation, which takes an argument out of the SLASH set of extracted elements and affixes it onto the main verb, yielding the base form *lui-parler* and the inflected form of 42b. The clitic affixation rule of Sag & Godard (1993) is given in example 43.

(43) Simplified Clitic Affixation Lexical Rule:

$$\left[\begin{array}{l}\textbf{synsem}\\ \text{LOCAL:}\left[\begin{array}{l}\textbf{local}\\ \text{CAT:}\left[\begin{array}{l}\textbf{category}\\ \text{HEAD:}\left[\begin{array}{l}\textbf{verb}\\ \text{CLTS:}\mathbf{S}_1\end{array}\right]\end{array}\right]\end{array}\right]\\ \text{NONLOCAL:}\left[\begin{array}{l}\textbf{nonlocal}\\ \text{SLASH:}\{\boxed{1}\}\cup\mathbf{S}_2\end{array}\right]\end{array}\right]$$

$$\longmapsto$$

$$\left[\begin{array}{l}\textbf{synsem}\\ \text{LOCAL:}\left[\begin{array}{l}\textbf{local}\\ \text{CAT:}\left[\begin{array}{l}\textbf{category}\\ \text{HEAD:}\left[\begin{array}{l}\textbf{verb}\\ \text{CLTS:}\mathbf{S}_1\cup\{\boxed{1}\}\end{array}\right]\end{array}\right]\end{array}\right]\\ \text{NONLOCAL:}\left[\begin{array}{l}\textbf{nonlocal}\\ \text{SLASH:}\mathbf{S}_2\end{array}\right]\end{array}\right]$$

We recall Abeillé & Godard's (1994) analysis of French tensed auxiliaries in which *avior* and *être* are argument-raising verbs. For these auxiliaries, the arguments are not lexically specified beyond the fact that they are shared with the main verb. Like the Complement Extraction lexical rule, the Clitic Affixation lexical rule (CALR) may be dealing with unknown arguments, this time in the SLASH set of the head verb. Order of rule application is important, here. In order to identify the slashed arguments, the rule must first obtain an instantiated instance of the CELR.[6]

[6]The rules are freely ordered, but it is a so-called rule "feeding" relation, in that once the CELR applies, then the CALR can apply to its output.

2.4 Quantifier Raising

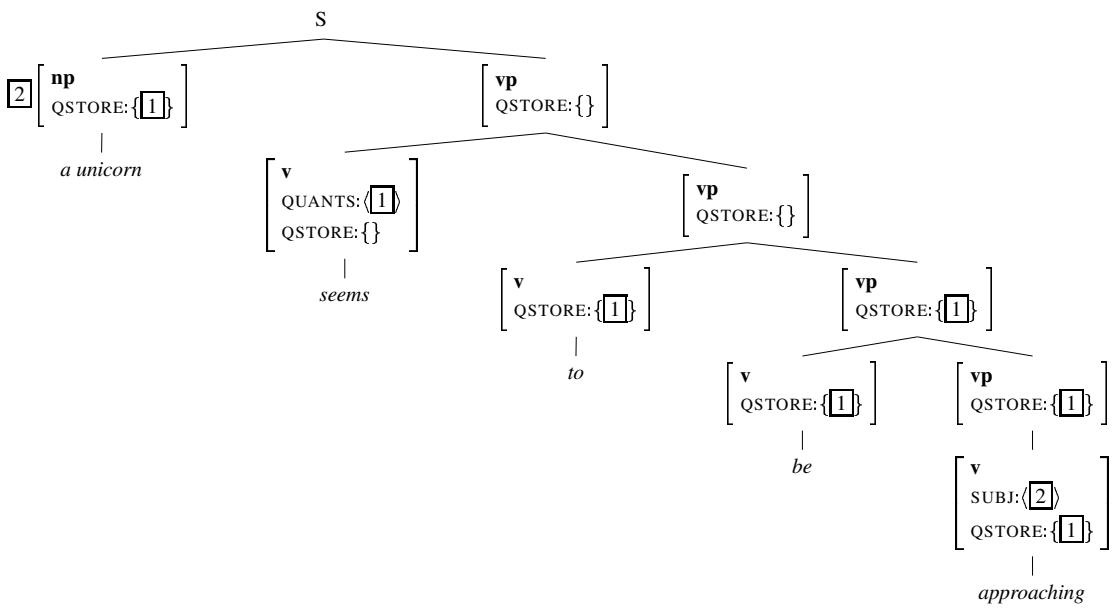
In Pollard & Sag (1994:chapter 8), all quantifiers begin in storage, using a standard interpretation of Cooper storage (Cooper (1983)). Quantifiers from the daughters of a phrase are passed up until a site for scope assignment is reached, at which time one or more quantifiers may be retrieved and given a wide scope interpretation. Pollard & Yoo (1997) show a departure from Cooper storage of quantifiers. Pollard & Yoo propose to make QSTORE a local feature, rather than a feature on the type **sign**, which is the most basic type of objects in HPSG. By being a local feature, it is part of the bundle of basic syntactic and semantic information of the sign. Unscoped quantifiers in QSTORE are passed up to the mother from the semantic head daughter by means of the semantics principle. The Pollard & Yoo analysis makes it possible to derive the *de dicto* or narrow scope reading for the following example:

(44) a. A unicorn seems to be approaching.

b. It seems that there is a unicorn approaching. (*de dicto*)

c. There is a unicorn that seems to be approaching. (*de re*)

(45)



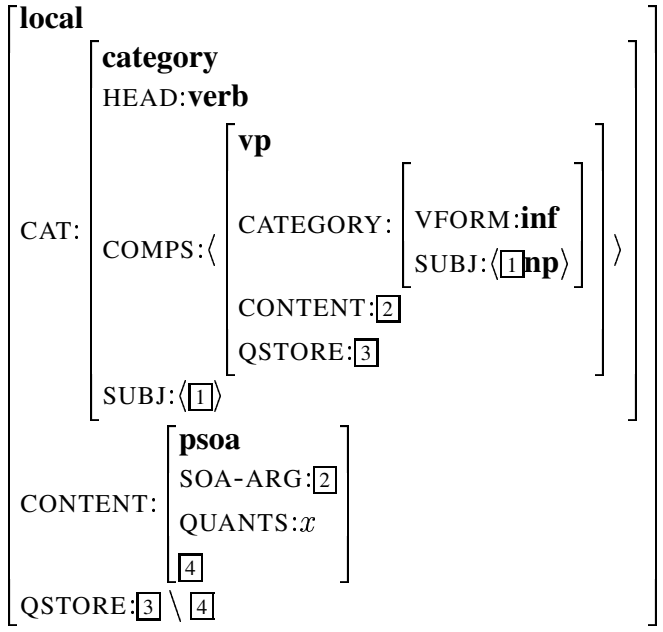
The tree in example 45 shows a derivation for example 44c. In 45 the feature RETRIEVED is not shown at VP nodes, but could be assumed, or could be absent, following the analysis of Manning & Sag (1998). We start unwrapping this tree from the bottom up. The QSTORE value of the subject NP *a unicorn* is the set containing the quantifier value for *a unicorn*, which we abbreviate with $\exists x.\mathbf{unicorn}(x)$ (following Pollard & Sag (1994)). Since the main verb *approaching* subcategorizes only for one argument, its subject *a unicorn*, and since its own QUANTS value is empty, then its QSTORE value will be the same as its subject's (noted in example 45 with the structure shared argument $\boxed{2}$).

At each VP node is a chance for the quantifier to be retrieved. A VP node is defined here as any point at which the subtree is headed by a verb. We follow the VP nodes in the tree up to the VP node *to be approaching*. At this node the quantifier for the subject NP *a unicorn* has been retrieved, inside the scope of the raising verb *seems*. We note this as the value of QUANTS. The QSTORE value for the VP *to be approaching* is the empty set. This QSTORE value is equal to the set of quantifiers stored for the subcategorized VP *be approaching* – still equal to the QSTORE value of the subject NP – minus its own QUANTS value. The wide scope reading that is available occurs when retrieval does not occur until the topmost S node.

Manning & Sag (1998) point out a problem with an earlier version of the Pollard & Yoo analysis of in that it lets quantifier retrieval happen in several places. Both this method for quantifier storage and retrieval and the analysis of Pollard & Sag (1994) allow spurious analyses of every available reading.[7] Manning & Sag propose to let quantifier scope assignment and retrieval be entirely lexical in nature, eliminating the feature RETRIEVED. The value for QSTORE for a verb is the union of the quantifiers stored for its arguments, minus its own quantifiers. Using the Manning & Sag (1998) analysis for quantifier storage and retrieval, we obtain the lexical entry for *seems* shown in example 46.

[7]Pollard and Yoo address the spurious ambiguity problem by constraining a phrase with a semantically vacuous head to have an empty RETRIEVED value. They offer evidence that phrase-level quantifier retrieval accounts for *wh*-scoping, following Carpenter (1994). They also suggest that

> ...the best solution lies in the direction of abandoning the whole notion of retrieval at structural nodes, and instead constraining the relation between CONTENT and phrase structure in some other way that makes no reference to retrieval in this sense.(Pollard & Yoo 1997:30)

(46)


For the verb *seems*, the value for QSTORE is the set of quantifiers stored for the subcategorized VP minus its own quantifiers, i.e. its QSTORE is the QSTORE of the embedded verb minus its own QUANTS value. We don't know, however, just what the quantifiers of the embedded verb are at this lexical level, since the embedded VP is not instantiated at this level. Furthermore, it is only by looking at the entire sentence tree in 45 that we know that the QUANTS value of any of the VP is $\exists x.\mathbf{unicorn}(x)$. Had the subject been a proper NP, such as *Kim*, then the QUANTS values would all be empty. The lexical entries for verbs employing lexical quantifier raising allow for this possibility by structure sharing quantifiers with subcategorized thematic complements, yet leaving these underspecified.

This program of gathering quantifier values for any SIGN is complemented by the analysis of Sag (1997), where lexical amalgamation of the features SLASH and REL is assumed.

2.5 Constraints on Linear Order

In West-Germanic languages, the order of words in the *Mittelfeld*, is somewhat free, resulting in discontinuous constituents. The Mittelfeld is the part of the German clause between the finite verb and the clause final verb or verb cluster, if any.

(47) Dann wird die Pille der Doktor dem Patienten geben.
Then will the pill the doctor to-the patient give.
Then the doctor will give the pill to the patient.

The arguments of *geben* (give) in sentence 47, including the subject *Doktor*, may be permuted (Uszkoreit 1987:93a-b):

(48) a. Die Pille gibt der Doktor dem Patienten.

b. Dem Patienten gibt die Pille der Doktor.

Linear precedence (LP) constraints are the second component of ID/LP grammars, which include GPSG (Gazdar *et al.* (1985)). Separating dominance and precedence relations disassociates the daughters in a phrase from the order(s) in which they are admissible. LP constraints order daughters of an unordered phrase structure rule with respect to each other. In HPSG work of the last decade, starting with Reape (1996) and including Kathol & Pollard (1995) and Kathol (1995), the domain of the linear precedence relation has been contained in the DOM or domain feature of a phrasal projection. A word order domain for a phrase is an ordered sequence of constituents, and consists of their PHON or phonology values

(49)
$$\begin{bmatrix} \textbf{phrasal-sign} \\ \text{PHON}: \boxed{1} \circ \ldots \circ \boxed{n} \\ \text{DOM}: \left\langle \left[\text{PHON} \boxed{1}\right], \ldots, \left[\text{PHON} \boxed{n}\right] \right\rangle \end{bmatrix}$$

The ordering of elements of a domain with respect to each other and the ordering of domains with respect to each other allows for accounts of complex word order phenomena. Elements from different domains may be "shuffled" or interspersed into the domain of the mother, as long as the relative order of each domain is preserved. A simple example of domain ordering from the German Mittelfeld is the constraint that pronouns precede nonpronouns.

(50) a. Ich gebe ihm das Buch.
I give him the book.
I give him the book.

b. *Ich gebe das Buch ihm.
I give the book him[dat].
I give him the book.

c. *Ich gebe ihm ihn.
I give him[dat] it[acc].
I give it to him.

d. Ich gebe ihn ihm.
I give it[acc] him[dat]
I give it to him.

Erbach *et al.* (1995:example 8) express the constraint on domains as in example 51, in the context of discussing tools for grammar development:

(51) $\forall x, y \in Dom$: if $(x = pron) \wedge (y \neq pron)$ then precedes(x, y)

A rendering of the constraint using an HPSG type signature looks as follows, relative to the complements of a verb phrase:

(52) $[\text{LOC:CONT} : \textbf{pron}] \preceq [\text{LOC:CONT} : \textbf{npro}]$

The constraint can neither be satisfied nor fail until all nominal contributors to the ordering domain DOM have their NOM-OBJ values specified, showing the types of noun phrases that they are. And so it shows that ordering must be done at a time when each constituent that maps to an ordering domain is sufficiently instantiated.

Penn (1999) uses embedded word order domains to formalize the placement of second-position clitics in Serbo-Croatian. This grammatical placement of the clitic *je* 'he' in 53b makes the syntactic NP *U lepi grad* discontinuous.

(53) a. U lepi grad je stigao.
in beautiful city cl.3s arrived
He has arrived in the beautiful city.

b. U lepi je grad stigao.

To show the contribution of prosody to word order, Penn posits a PROSODY feature with its own domain lists (PDOM). The domain DOM of the entire sign is the result of appending the lists of domain objects in the list of prosodic words. In this way a prosodic word may be the relevant object with respect to which the clitic is ordered, demonstrating interdependence between the syntactic and prosodic features.

2.6 Problems for Evaluation

The examples in this chapter have all been selected because they pose particular problems for evalution during natural language processing. We look now at the evaluation of argument sharing, first introduced in section 2.1. The problem we

discover is that one head might need to gain information which is projected from a different head, but underspecified. In the processing paradigm, one way a feature structure is underspecified is if the path value at the node which is being evaluated is undefined. It may also be that the type in a typed feature structure is not a maximally specific type, or that an entire piece of the structure is co-referenced or shared with another. In a parsing or generation process, we store all the information we gain as we go along in a common feature structure, updating that information and creating new path values as we process. It may be the case that a feature value that is undefined or not maximally resolved early in the computation will be more specific later, as a result of gaining new information during the parse.

A simple feature structure showing the semantic features for the noun *sheep* might look as in example 54. The type **nom-object** refers to the fact that a sheep belongs to the class of things that are nominal objects.

(54)
$$\begin{bmatrix} \textbf{nom-object} & \\ \text{CONTENT:} & \begin{bmatrix} \textbf{content} \\ \text{PERSON:}\textbf{3rd} \\ \text{NUMBER:}\textbf{number} \\ \text{RELATION:}\textbf{sheep} \end{bmatrix} \end{bmatrix}$$

Clues from other parts of the grammar could help to make the feature or type more specific when the feature structure gains information during natural language processing. We refer to these "clues" as constraints. In the sentence *The sheep graze*, the "filled-in" version of the feature structure has the subtype of **number** resolved to **plural** in 55:

(55)
$$\begin{bmatrix} \textbf{nom-object} & \\ \text{CONTENT:} & \begin{bmatrix} \textbf{content} \\ \text{PERSON:}\textbf{3rd} \\ \text{NUMBER:}\textbf{plural} \\ \text{RELATION:}\textbf{sheep} \end{bmatrix} \end{bmatrix}$$

The sentence *The sheep bleats* would likewise resolve to the singular form. The number for the noun is obtained via unification with the number form on the inflected verb. Typically, the number for the verb form is added into a basic, also underspecified feature structure for the verb's base form, by way of some morphological analysis.

For our test case, we look at a lexical rule that has the potential to extract underspecified complements from an auxiliary at the point of the lexicon. The

feature structure template for the raising auxiliary *werden* is repeated here as example 56.

(56)
$$\left[\begin{array}{l}\textbf{werden}\\ \text{HEAD:}\left[\begin{array}{l}\textbf{verb}\\ \text{VFORM:}\textbf{bse}\\ \text{AUX:}\textbf{+}\end{array}\right]\\ \text{SUBJ:}\boxed{1}\langle\textbf{NP}\rangle\\ \text{COMPS:}\boxed{2}+\left\langle\left[\begin{array}{l}\text{HEAD:}\left[\begin{array}{l}\textbf{verb}\\ \text{VFORM:}\textbf{bse}\end{array}\right]\\ \text{COMPS:}\boxed{2}\\ \text{SUBJ:}\boxed{1}\\ \text{NPCOMP:}\textbf{-}\end{array}\right]\right\rangle\end{array}\right]$$

In section 2.3.1 I introduced PVP fronting as a technique from HPSG for obtaining fronted verb phrases in Germanic languages. We saw that a partial verb phrase has a verb for its head and some number of the verb's complements, but it may not be saturated.[8] A specific instance of the Complement Extraction Lexical Rule is the Partial Verb Phrase (PVP) Fronting Rule of Baker (1994). A naive or preliminary implementation of the PVP rule as it has been written using the ALE system of Carpenter & Penn (1994) is shown in figure 2.1. The `append` relation that is assumed states that the concatenation of outstanding complements of the partial verb phrase (`PVPComps`) and the PVP itself (`loc:PVP`) yields the list of all of the arguments on the SUBCAT or subcategorization list of the output auxiliary (`SubcatComps`):

(57)
```
append(PVPComps,
       [(loc:PVP,non_loc:(inherited:(slash:(elt:PVP))))],
       SubcatComps)
```

I will now consider various strategies which could be considered generally in the implementation of the PVP rule. I describe three methodologies for handling constraints or rules such as this, which appear in the lexicon: *Brute Force*, *Accommodation*, and *Control*. These methods can be roughly characterized as follows:

- Brute Force: Explicit rendering of a constraint to be satisfied via instantiation of its possibilities.

[8]The word "saturated" is used to mean that a head has picked up all of its complements and no longer subcategorizes for any of them in the phrase.

```
%%   % Partial Verb Phrase Fronting Rule
pvp lex_rule
   (word,
    subcat:[Subj|OldComps],
    synsem:(loc:(cat:(head:(Head,verb,vform:bse,aux:plus,flip:minus),
                      subj:[Subj],
                      comps:OldComps)),
            non_loc:(inherited:(slash:OldSlash))))
   **>
   (word,
    subcat:[Subj|SubcatComps],
    synsem:(loc:(cat:(head:Head,
                      subj:[Subj],
                      comps:PVPComps),
                 cont:Cont),
            non_loc:(inherited:(slash:(elt:(PVP,cat:(head:(verb,vform:bse),
                                                     lex:minus,
                                                     subj:[Subj],
                                                     comps:PVPComps),
                                                cont:Cont),
                                       elts:OldSlash)))))
%% The next line is key
if
   (append(VComps,[(loc:cat:(head:(verb,vform:bse),
                             lex:plus,
                             comps:VComps,subj:[Subj]))],
                             OldComps),

    three_or_less(PVPComps),
    append(PVPComps,
           [(loc:PVP,non_loc:(inherited:(slash:(elt:PVP))))],
           SubcatComps)
```

Figure 2.1: PVP rule – naive implementation

- Accommodation: Hand-coded software changes to account for anticipated problems.
- Control: Keeping certain constraints from consideration during the constraint resolution algorithm, while stating them where they most naturally occur.

2.6.1 Brute Force

One approach is to use a "brute force solution" to lexical underspecification. In order to make sure that a lexical constraint is satisfied early, one adds a lexical entry for every instantiation of the constraint. The problem with this approach is that some instantiations may not be needed, or worse, may not be reasonable, but are generated due to the generality of the constraint. They also load up the lexicon with entries that lead to parse ambiguity.

For example, for the case of an auxiliary verb raising the complements of the verb it governs, there is no natural way to use the list data structure to state that the last element of a list is the head verb and that all the other arguments are the verb's argument. That is because a list is described as the head, or first element of the list, and the rest, or tail, of the list. In Prolog, this is rendered simply as in example 58. Unless *the length of the list is known* and the list can be reversed, the last element cannot be referred to with a variable.

(58) `[Head|Tail]`

The following examples 59a through 59c are instantiations of the auxiliary *können* where the verb that is governed may have one, two, or three possible arguments. The PVP rule can be applied to *each* of these lexical entries. In these entries, I show only the relevant SYNSEM values.

(59) a.
```
minus flip aux,
taking an intransitive verb

koennen --->
  word,
  synsem:loc:(cat:(head:(verb,
                         mod:none,
                         vform:bse,
                         aux:plus,
                         flip:minus),
                   subj:[ (NP, @ np(_)) ],
      comps: [loc:(cat:(head:(verb,vform:bse),
```

```
                                              subj:[NP],
                                              comps:e_list,
                                              lex:plus),
                                        cont:Prop)]
```

b. minus flip aux,
taking a verb plus one complement

```
koennen --->
  word,
  synsem:loc:(cat:(head:(verb,
                         mod:none,
                         vform:bse,
                         aux:plus,
                         flip:minus),
                   subj:[ (NP, @ np(_)) ],
      comps: [Comp1,(loc:(cat:(head:(verb,vform:bse),
                                              subj:[NP],
                                              comps:[Comp1],
                                              lex:plus),
                                       cont:Prop))]
```

c. minus flip aux,
taking a verb plus two complements

```
koennen --->
  word,
  synsem:loc:(cat:(head:(verb,
                         mod:none,
                         vform:bse,
                         aux:plus,
                         flip:minus),
                   subj:[ (NP, @ np(_)) ],
      comps: [Comp1,Comp2,(loc:(cat:(head:(verb,vform:bse),
                                              subj:[NP],
                                              comps:[Comp1,Comp2],
                                              lex:plus),
                                       cont:Prop))]
```

With a simple grammar and lexicon and the rule in figure 2.1, we are able to parse German sentences with fronted verb phrases such as example 60.

(60) Kim sehen wird Sandy.
Kim see will Sandy
'Sandy will see Kim.'

We obtain four more lexical entries for *wird* after the application of the PVP rule at the time of compilation of the lexicon, for the cases where the governed verb has zero, one, two or three things on its COMPS list of complements. We have limited the number of complements to three; this is discussed in more detail as the strategy of accommodation, below. Having more than one lexical entry for *wird* populates the chart in a bottom-up chart parsing strategy, but is not a source multiple solutions for this particular example (60).

Things get more interesting, however, when we add to the grammar another variant of the CELR which extracts a noun from a base form verb. Such a version of the CELR enables examples with NP fronting such as 61 to be parsed.

(61) Kim wird Sandy sehen.
Kim will Sandy see.
'Sandy will see Kim.'

The feature structure template version of this rule is shown as example 62.

(62) Lexical rule to extract a noun complement:

$$\left[\begin{array}{l}\textbf{synsem}\\ \text{LOCAL}:\left[\begin{array}{l}\textbf{local}\\ \text{CAT}:\left[\begin{array}{l}\textbf{category}\\ \text{COMPS}:\langle \ldots, \boxed{1} NP, \ldots \rangle\end{array}\right]\end{array}\right]\end{array}\right]$$

$$\longmapsto$$

$$\left[\begin{array}{l}\textbf{synsem}\\ \text{LOCAL}:\left[\begin{array}{l}\textbf{local}\\ \text{CAT}:\left[\begin{array}{l}\textbf{category}\\ \text{COMPS}:\langle \ldots \ldots \rangle\end{array}\right]\end{array}\right]\\ \text{NONLOCAL}:\left[\begin{array}{l}\textbf{nonlocal}\\ \text{SLASH}:\{\boxed{1}\}\end{array}\right]\end{array}\right]$$

Having both rules in the grammar creates numerous entries in the lexicon. Besides the correct parse of sentence 61, we obtain a bizarre parse for the sentence as a phrase with a mythical VP in SLASH, which itself has a VP in SLASH, which is headed by the verb 'see.' This is certainly incorrect. This is the result of three separate steps conspiring. First, allowing the auxiliary *werden* to be slashed for PVP; second, joining the auxiliary with the verb *sehen*, which is slashed for NP; and third, picking up the NP *Kim*, but no VP, in slash. The constraint on the

PVP lexical rule says that the PVP in SLASH subcategorizes back for the elements that the auxiliary has already picked up. The bottom line, of course, is that in example 61 we have no need for lexical entries for auxiliary with slashed VP. These will only create unwanted ambiguity during parsing. And if they are there, the grammar must be fine-tuned to prevent over-generation, stating the proper conditions on slashed elements.

The most obvious improvement to the brute force approach would be to leave the COMPS list as a single uninstantiated variable and then state separately that auxiliary raising is taking place. It seems important to know at the outset, of course, that the auxiliary must take at least a verb as its primary argument.

2.6.2 Accommodation

Accommodation is hand-coding the knowledge sources in order to account for any program behavior which may result from premature solving of constraints. An example of accommodation is adding a limit on the length of an uninstantiated list, so that an infinite search path does not result. This actually changes the meaning of the program.

The conditions in the `if` clause at the bottom of the rule in figure 2.1 include the definite clause `three_or_less(PVPComps)`. This instantiates the list of complements in the fronted verb phrase, the list `PVPComps`, to be of length three complements or less. Without this statement, the compiler goes into an infinite loop as it tries to solve the `append` relation. In that case, neither the complements list of the fronted verb nor of the auxiliary output to the lexical rule are known, and so the Prolog engine enters into an infinite search branch trying to come up with arguments that will satisfy the relation.

As we have mentioned, the interaction of partial verb phrase fronting with the CELR causes some interesting results. The output of the CELR is grammatical in HPSG theory. In practice, however, we need to be a bit wary:

- First, we want to curb the application of the rule over and over, potentially infinitely, on entries where the argument structure is largely variable, and the output could easily match the input. This problem is described by Höhle (1995), who suggests suitable restrictions on the ARGSTORE of output to the rule. (such a constraint is present on the feature SUBCAT, for subcategorization, in figure 2.1).

  As van Noord & Bouma (1995) indicate, if lexical rules are written instead as unary grammar rules encountered in the syntax, we may have just pushed

off to a different processing stage the problem of infinite derivations.

- Second, we wonder if our process will terminate. In Prolog, for example, the *concatenate* operation, which relates two lists, e.g. the subcategorization lists of a verb and its governing auxiliary, will go into an infinite search loop if the first or third arguments are variable.
- Third, even for non-auxiliaries, whose arguments are available in the lexicon, we will increase local ambiguity during parsing if our lexicon contains a priori an entry for every word with every possible complement extracted.

Are these real problems during a parsing or generation process? Certainly the CELR could be tailored for a specific case via greater specification of inputs and outputs (as in Meurers 1994), or the processing system could set bounds on the number of rule applications (as in Carpenter & Penn 1994), which provides for limiting the total number of lexical rule applications) in order to curb these problems. However, the former solution will certainly result in a loss of generality, and the second, though easy to implement, is only a partial fix. In operational terms, a lexical argument extraction should only take place once when a single extracted argument is present in SLASH.

2.6.3 Control

Control is a strategy for constraint resolution. It results in a constraint ordering. Control may be done explicitly, by ordering constraints by hand, or implicitly, by using the program's goal resolution algorithm. Or, the two methods can be combined. In our example of complement extraction, the most straightforward solution may be to code the constraint on extracted elements separately, as in example 63, and fire the constraint during the parsing process, when we know that the extracted argument has been found. The grammar writer may simply list the constraint among the all the constraints fired, and at the appropriate time.

(63)
$$\begin{bmatrix} \textbf{word} \\ \text{SUBJECT}:\boxed{1} \\ \text{COMPLEMENTS}:\boxed{2} \ominus \textbf{gapped-args} \\ \text{DEPENDENTS}:\boxed{1} \oplus \boxed{2} \end{bmatrix}$$

Example 64 is the verb-phrase grammar rule in a German grammar. It is an instance of Schema 2 in HPSG theory, which is implemented as a phrase-structure rule. The `goal` keyword indicates definite clause goals which must

be solved before the rule is complete. In this example, one goal, the constraint `aux_raising`, checks that the subcategorized verb plus its complements list equals the complements list of the auxiliary. The means that the "problem" associated with stating the constraint on empty arguments in the lexicon is not an issue, since the constraint has been moved out of the lexicon and into the syntax.

(64)
```
schema2pvp rule
(Mother,phrase)
===>
goal> three_or_less(Comps),
cats> Comps,
cat> (HeadDtr,word,synsem:(HeadSyn,
                           loc:cat:(head:
                                    (verb,inv:minus,flip:minus),
subj:[SubjSyn]),
non_loc:inherited:slash:e_set)),
goal> (synsems_to_signs(FoundCompSyns,Comps),
       head_feature_principle(Mother,HeadDtr),
       valence_principle(Mother,HeadDtr,[],FoundCompSyns,[]),
       semantics_principle(Mother,HeadDtr,Comps),
...
       aux_raising(HeadSyn,FoundCompSyns,SubjSyn)).
```

The complement extraction case is somewhat more complicated than simple raising by auxiliary, because the head of a filler-head construction is a verb phrase, and not a lexical head. In traditional HPSG grammars, the schema 2 rule or "verb phrase rule" is the only site for checking constraints between a lexical head and its complements. So, locating a lexical constraint on extraction at the site of the filler-head rule may take some gymnastics, or perhaps the percolation of the ARG-S feature to the phrase as a head feature, which I presume is not the intent of Bouma *et al.* (2001).

Control can still be achieved by stating the constraint on auxiliary raising in the lexicon, where the arguments to which it applies are introduced into the grammar, but indicating that it should be delayed. With this strategy, the program control can check for the satisfaction of the constraint once the word has actually encountered its arguments during processing. In this way, the grammar writer need not be concerned about which phrase structure rule it is that actually unites these constituents.

In order to use control explicitly in grammar descriptions, yet not be worried unduly with attaching delay statements to lexical or grammar constructs, one can build delays into the description language. The solution we seek is to extend the description language for typed feature structures in Carpenter (1992) to

include waits on satisfaction of feature structure descriptions. The waits may occur any place feature structure terms are used, including lexical descriptions, constraint descriptions, or definite clause statements. We will show how delayed rules may also be used. The idea of waiting began with negations in Prolog II Naish (1985) and has been generalized to the paradigm of Constraint Logic Programming, which is introduced in the following chapter.

2.7 Summary

In this chapter we have looked at a variety of analyses in which arguments are shared across structures. Having established the linguistic data, we look ahead to implementational issues. How does sharing impact the ability to process these structures in a parsing or generation context? Constraints are sensitive to the information available to solve them. Sometimes, information, such as complement structure, is shared by more than one head. This means that one head may need to wait for that part of the data which is provided by the other head. In this chapter I have shown the inadequacy of some methods for processing feature structure grammars, including brute force approaches and hand-tailored approaches. Beginning in the following chapter, I describe delaying, which is used to apply constraints only after there is enough information to make them effective. Delaying also avoids non-termination problems. This methodology will be useful in any number of linguistic contexts, including the causative, the passive, complement extraction, clitic movement, quantifier raising, and word order.

Chapter 3

Logic Programming

As a parallel exposition to the linguistic literature in chapter two, I review in this chapter the field of Logic Programming. I review the methodology for SLD resolution, and include a definition of resolution for Constraint Logic Programming (CLP). I introduce the concept of delaying and show guarded rules for CLP. I give an example of guarding from NLP and consider some other approaches to constraint solving. A formal description of feature structures follows in chapter 4. We will then be able to show how guarded rules and feature structures can be used together to solve the general problem of processing underspecified linguistic objects.

The definitions for concepts in logic programming (section 3.1) have been written largely with reference to Sterling & Shapiro (1986), and the definitions for CLP (section 3.2) have been written after a close reading of Marriott & Stuckey (1998).

3.1 Logic Programming

In logic programming, a program is made of a series of statements, or *goals*. Goals are also called definite clauses.

Definition 3.1 (Terms) *A* term t *is a variable, a constant, or a function symbol* $f(t_1, \ldots, t_n), n \geq 0$. *Variable names begin with an upper case letter, e.g.* X, Y, Feats, Cat.
Constants *are integers or names that begin with a lower case letter, e.g.* kim, sandy.
A function symbol *is a name that begins with a lower case letter, e.g.* successor.

Some function symbols may be built in to the language. For example, *lists* in Prolog are built underlyingly using the function operator `cons`. The first argument of `cons` is the first element of the list and the second argument is the whole remainder, or tail, of the list. The empty list `[ ]` is a constant. Instead of writing `cons(a,cons(b,c))`, one may use square brackets around the members of the list, i.e. `[a,b,c]`. Therefore, a list is a valid term. A vertical bar may be used to separate the head of the list from the rest of the list, e.g. `[a|b,c]` or `[X|Y]`.

Definition 3.2 (Goal) *A goal is a expression of the form* $p(t_1, \ldots, t_n), n \geq 0$ *where* p *is a relation symbol and the* t_i *are terms. A relation symbol is a name that begins with a lower case letter.*

Example goals are *father(kim,sandy)* and *append([ ],X,X).*

Definition 3.3 (Clause) *A clause is a goal or a statement of the form*
$A \leftarrow B_1, B_2, \ldots, B_n, n \geq 0$ *where* $A, B_1, B_2, \ldots, B_n$ *are goals. A is the head of the clause.*

An example clause is
$append(cons(X, Xs), Ys, cons(X, Zs)) \leftarrow append(Xs, Ys, Zs)$.

Definition 3.4 (Program) *A logic program is a finite set of clauses.*

A simple program for appending two lists together is

(65)
```
append([ ],Xs,Xs).
append([X|Xs],Ys,[X|Zs]) <-- append(Xs,Ys,Zs)
```

Definition 3.5 (Resolvent) *A resolvent is a conjunction of goals to be proved.*

In logic programming the technique of *SLD resolution* is used to deduce a goal given a program. This technique dates to Robinson (1965) and is described in Lloyd (1984). Each clause in logic program has a goal on the left hand side that can be proven from instances of the goals on the right hand side. During resolution, clauses work together to prove new instances of goals that can be reused in other clauses, until the input goal is proven. The operational definition of SLD resolution from Sterling & Shapiro (1986) is shown in figure 3.1. Two concepts are key in understanding resolution. The first is *substitution* and the second is *unification*. A substitution is a choice of a term t to take the place of each occurrence of a particular variable X as it occurs throughout a term s. Unification occurs when two terms s_1 and s_2 are made identical through a series of variable substitutions.

Definition 3.6 (Substitution) *A* substitution *is a finite set (possibly empty) of pairs of the form X=t, where X is a variable and t is a term, and with no two pairs having the same variable as left-hand side. For any substitution* $\theta = \{X_1 = t_1, X_2 = t_2, \ldots, X_n = t_n\}$ *and term s, the term* $s\theta$ *denotes the result of simultaneously replacing in s each occurrence of the variable* X_i *by* t_i, $1 \leq i \leq n$. *The term* $s\theta$ *is called an* instance *of s.(Sterling & Shapiro 1986)*

The substitution $\theta = \{Xs = [a], Ys = [b]\}$ applied to the term $append(Xs, Ys, Zs)$ yields $append([a], [b], Zs)$.

Definition 3.7 (Unification) *A substitution* θ *unifies* s_1 *and* s_2 *if* $s_1\theta = s_2\theta$.

The terms $append(Xs, Ys, [a, b])$ and $append([a], [b], Zs)$ may be unified with the substitution $\theta = \{Xs = [a], Ys = [b], Zs = [a, b]\}$ to yield the common term $append([a], [b], [a, b])$.

Definition 3.8 (Most General Unifier) *A term* s *is an* instance *of a term* t *if* s *is obtainable from* t *via substitution. A term* s *is* more general *than* t *if* s *is an instance of* t *but* t *is not an instance of* s. *For each unifiable set of terms* s_1 *and* s_2 *there is a substitution that produces a unique most general term* s. *This substitution is known as the* most general unifier*, or MGU, of the two terms. The uniqueness of* s *is up to the renaming of variables, taking care not to reuse names which occur elsewhere in the term.*

For the two terms $s_1 = append(Xs, Ys, Zs)$ and $s_2 = append([a], Rest, List)$ a most general unifier is $\theta = Xs = [a], Ys = Rest, Zs = List$ resulting in the term $s = append(Xs, Rest, List)$. Another MGU would be $\theta = \{Xs = [a], Ys = Ys1, Zs = Ys1, Rest = Ys1, List = Ys1\}$ resulting in the term $append([a], Ys1, Zs1)$. The unifier $\theta' = \{Xs = [a], Ys = Rest = [b], Zs = Rest = [a, b]\}$ applied to s_1 and s_2 would yield the term $s' = append([a], [b], [a, b])$, but θ' would not be the most general unifier since s is an instance of s' but s' is not an instance of s.

The steps in solving a goal with resolution can be shown as a search tree, with the goal to solve as the root node of the tree. Each branch from that node goes to a clause from the program unified with the goal selected from the resolvent. The search tree in figure 3.2 shows a proof of goal $append([a, b], [c, d], Q)$. This is the goal of concatenating the lists $[a, b]$ and $[c, d]$ to yield $[a, b, c, d]$. Figure 3.3 shows all the steps in resolving this goal. There is one goal to choose from in the resolvent and there are two clauses to choose from in the program P, which is shown in

Input: A logic program *P*
A goal *G*

Output: $G\theta$, if this was the instance of *G* deduced from P,
or *failure* if failure has occurred.

Algorithm:

Initialize the resolvent to be *G*, the input goal.

While the resolvent is not empty do
Choose a goal *A* from the resolvent
and a (renamed) clause $A' \leftarrow B_1, B_2, \ldots, B_n, n \geq 0$, from *P*
such that *A* and A' unify with MGU θ
(exit if no such goal and clause exist).
Remove *A* from and add B_1,B_2,. . ., and B_n to the resolvent.
Apply θ to the resolvent and to *G*.
If the resolvent is empty output *G*, else output *failure*

Figure 3.1: SLD Resolution. (Sterling & Shapiro 1986: Figure 4.2)

$append([a, b], [c, d], Q])$ $\{X = a$,$Xs = [b]$,$Ys = [c, d]$,$Q = [X|Zs]\}$
$\downarrow$
$append([b|[\,]], [c, d], [b|Zs1])$ $\{Xs = [X1|Xs1]$,$X1 = b$,$Xs1 = [\,]$,
$Zs = [X1|Zs1]\}$
$\downarrow$
$append([\,], [c, d], [c, d])$. $\{Ys = Zs1\}$

Figure 3.2: Search tree for proof of $append([a, b], [c, d], Q)$. By composition, the value of Q is equal to the composition of substitutions in the proof tree.

program P:

```
append([],X,X).
append([X|Xs],Ys,[X|Zs]) <-- append(Xs,Ys,Zs).
```

goal G = $append([a,b],[c,d],Q)$

1. Resolvent: $\{append([a,b],[c,d],Q)\}$. Choose this goal.
   Choose `append([X|Xs],Ys,[X|Zs]) <-- append(Xs,Ys,Zs)` from P.
   Rename clause $append([a|b],[c,d],[X|Zs]) \leftarrow append(b,[c,d],Zs)$.
   $\theta = \{X = a, Xs = b, Ys = [c,d], Q = [a|Zs]\}$
   Resolvent: $\{append([b,[c,d],Zs)\}$
   Apply theta to G: $append([a,b],[c,d],[a|Zs])$
2. Choose goal $append([b,[c,d],Zs)$ from the resolvent.
   Choose `append([X1|Xs1],Ys1,[X1|Zs1]) <-- append(Xs1,Ys1,Zs1)` from P.
   Rename clause $append([b|[]],[c,d],[b|Zs1] \leftarrow append([],[c,d],Zs1)$.
   θ = above substitutions plus $\{X1 = b, Xs1 = [], Ys1 = [c,d], Zs = [b|Zs1]\}$
   Resolvent: $\{append([],[c,d],Zs1)\}$
   Apply theta to G: $append([a,b],[c,d],[a|[b|Zs1]])$
3. Choose goal $append([],[c,d],Zs1)$ from the resolvent.
   Choose `append([],X3,X3)` from P.
   Rename clause $append([],[c,d],[c,d])$.
   θ = above substitutions plus $\{X3 = [c,d], X3 = Zs1\}$
   Apply theta to G: $append([a,b],[c,d],[a|[b|[c,d]]])$
   Resolvent is empty { } so output G: $append([a,b],[c,d],[a|[b|[c,d]]])$

Figure 3.3: Steps in resolving the goal of concatenating the two lists [a,b] and [c,d] to output list [a,b,c,d].

$append([X|Xs1],[c,d],[X|Ys1])\{Q1=[X|Xs1],Q2=[X|Ys1]\}$
$\downarrow$
$append(Xs1,[c,d],Ys1)\{Xs1=[X1|Xs2],Ys1=[X1|Ys2]\}$
$\downarrow$
$append(Xs2,[c,d],Ys2)\{Xs2=[X2|Xs3],Ys2=[X2|Ys3]\}$
$\downarrow$
$append(Xs3,[c,d],Ys3)$
$\downarrow$
$\ldots$

Figure 3.4: Search tree with an infinite branch. Goal is $append(Xs,[c,d],Ys])$. The first and third list arguments are variable and so can grow infinitely long, by pushing down a new first element on the tail of the list.(Sterling & Shapiro 1986: Figure 5.4)

example 65. In figure 3.3, the correct clause can be chosen deterministically, but in general, the process is non-deterministic.

In fact, it is possible to keep selecting clauses which lead to an infinite path. Suppose we are solving the goal $append(Q1,[c,d],Q2)$. If we choose the clause $append([\,],Xs,Xs)$ then we have a successful derivation with Q1=[ ],Q2=[c,d]. But if we choose the clause $append([X|Xs],Ys,[X|Zs]) \leftarrow append(Xs,Ys,Zs)$ then we can push down infinitely on the tails of the first and third arguments of the goal, which are lists. This type of infinite search tree is shown as figure 3.4.

3.1.1 Negation as Failure

The way negation is used in logic programming is called *negation as failure*. The symbol not is applied to goals. In the search tree for G, if all solutions for a goal G fail in the search without any infinite search trees, then the goal $not\ G$ succeeds. Conversely, if one instance of G succeeds, then $not\ G$ fails. The substitution used to solve the goal $not\ G$ extends the current substitution and is scoped to G. The classic work on the semantics of negation as failure is Clark (1978).

Definition 3.9 (Finite Failure) *A goal G finitely fails with respect to a program P if every branch of an SLD search tree having G as its root terminates in failure. Otherwise, $not\ G$ finitely fails if G succeeds. (Kowalski 1992)*

Prolog II (Colmerauer 1982) and MU-Prolog (Naish 1985) are instances of logic programming. Prolog is an implementation of logic programming in which

an order is used to control search of both clauses in the program and goals in the body of a clause. In Prolog II, the domain of Prolog is extended to infinite trees and unification is achieved by solving systems of equations and inequations. Inequations allow a *not equals* operator to be used on variables, in order to declare that two variables are not unifiable. These inequations will become important in a linguistic context, as the constraints of the binding theory are discussed. In Prolog II and MU-Prolog, strategies for control, including wait declarations, are introduced. These provide the ability to delay the evaluation of a call to a goal until one or more argument variables of the goal are bound. We show delay constraints in more detail in the following section.

3.2 Constraint Logic Programming

In particular, the Constraint Logic Programming (CLP) paradigm of Jaffar & Lassez (1987) is relevant for the current work. In this model of logic programming, the operational step in execution is based upon SLD resolution plus determining the solvability of constraints. Jaffar & Lassez characterized systems such as CLP(R) (Prolog with real numbers) (Jaffar & Michaylov 1987) and Prolog III (Colmerauer 1990). Höhfeld & Smolka (1988) generalized the CLP framework for arbitrary constraint languages, so that a constraint could be represented in a domain specific fashion. This meant that constraints were not limited to the syntax of first-order logic. Höhfeld & Smolka established their work for the domain of Knowledge Representation, which includes the feature logics used to express linguistic theories such as HPSG.

Definition 3.10 (Atomic Constraint) *An atomic constraint of the form* $p(t_1, t_2, \ldots, t_n)$ *is a relation symbol over terms. A relation symbol* p *is a name with a lower case letter or a arithmetic relation symbol such as <,>,=. A constraint has a value of* true*,* false *or* unknown*, relative to an assignment of values for variables in p. A relation that is not an arithmetic relation is defined elsewhere.*

Examples of an atomic constraints are $nonvar(X)$, $X < 2$, and $X = Y$. These constraints narrow down the search space for a solution by restricting the set of values a variable could take. This might include, for example, a constraint on whether the variable has been instantiated. e.g. $nonvar(X)$ is true when X is instantiated.

Definition 3.11 (Constraint) *A constraint ϕ for CLP is a conjunction of atomic constraints. ϕ is true if every atomic constraint is ϕ is true and is false if any atomic constraint in ϕ is false, and is unknown otherwise.*

Definition 3.12 (Satisfiability) *A constraint ϕ over a list of terms $t_1, t_2, \ldots, t_n$ is satisfiable if there is an assignment θ of values for variables in $t_1, t_2, \ldots, t_n$ such that ϕ is true. A constraint ϕ is unsatisfiable if there is no assignment of values for variables such that ϕ is true.*

Definition 3.13 (CLP Goal) *A CLP goal is defined as $\phi \wedge B_1 \wedge B_2 \wedge \ldots \wedge B_n$, $n \geq 0$, for ϕ a constraint. $B_1, B_2, \ldots, B_n$ are goals.*

Definition 3.14 (CLP Clause) *CLP clauses are of the form $A \leftarrow (\phi \wedge B_1 \wedge B_2 \wedge \ldots \wedge B_n)$ where ϕ is a constraint and $A, B_1, B_2, \ldots, B_n$ are goals. A substitution θ of values for variables that satisfies ϕ must also satisfy $B_1 \wedge B_2 \wedge \ldots \wedge B_n$*

Resolution for CLP is similar to standard SLD resolution, except that the solvability of the constraints that have accumulated is checked at each new goal. At any time that the constraints are not satisfiable, then the resolution fails. Refer to figure 3.5. It is possible to complete a derivation without being able to verify that the accumulated constraints, called the *constraint store*, are satisfiable.

3.2.1 Guarded Rules in CLP

A CLP clause can be used a means of controlling the evaluation of a CLP program. Beginning with IC-Prolog, Prolog II or MU-Prolog, simple delay primitives have been available to the programmer to delay the evaluation of a particular goal until constraints are satisfied. The constraint language for delays is made up of relations which are primitives of comparison between two terms, such as equality, inequality, and less than or greater than comparisons. Prolog also allows *delaying* on the instantiation and grounding of terms. This is very useful when processing underspecified terms, which are used in natural language, as we shall see in chapter 6. The idea of guarding a rule is really no different than establishing preconditions for its completion, an idea that dates in Computational Linguistics at least to the Augmented Transition Networks (ATNs) of the 1970's (Woods 1970). In the case of preconditions, there is not explicit delaying, but implicit rule ordering, such that one rule is able to "feed" another. With guarding, rules enter a state of *supension* if the rule matches but the constraints cannot be satisfied. This means that the

Input: A logic program *P*
A goal *G*
A constraint store *C*

Output: *Gθ and C, if* $C \neq false$,
or *failure* if failure has occurred or $C = false$.
If $C = true$, then Gθ is an answer for G.

Algorithm:

Initialize the resolvent to be *G*, the input goal.
Initialize the constraint store *C* to be empty.

While the resolvent is not empty do
Choose a goal *A* from the resolvent
and a (renamed) clause $A' \leftarrow \phi, B_1, B_2, \ldots, B_n, n \geq 0$, from *P*
such that *A*, *A'* and ϕ unify with MGU θ
(Exit if no such goal and clause exist).

Add ϕ to C. Continue if C is not false, and exit otherwise.

Remove *A* from and add B_1,B_2,. . ., and B_n to the resolvent.

Apply θ to the resolvent, to *G*, and to *C*.

Figure 3.5: SLD Resolution with Constraints. The derivation fails if the constraints in the constraint store are not satisfiable, and succeeds otherwise. If the constraint store is true, then an answer has been found. Otherwise, the derivation has completed without an answer.

rule is out of consideration until its guard is satisfied during the course of another search path.

Delaying the evalutaion of a constraint ϕ is also known as guarding a constraint, and ϕ is a *guarded constraint*. The $when$ predicate shown in example 66 is an example of a delaing predicate from SICStus Prolog (SICStus 1995). We will wait to evaluate a constraint in a clause from a CLP program until we have "enough information."

(66) `when(+Condition,Goal)`

The predicate `when` blocks `Goal` until `Condition` is true, where `Condition` is a Prolog goal with restricted syntax. `Condition` is restricted in SICStus Prolog to `nonvar(X)`, `ground(X)`, or `?=(X,Y)`.

Definition 3.15 (Delay Constraint) *A delay constraint is a constraint used to delay the evaluation of a goal. The set of delay constraints is a subset of the set of constraints in the language.*

Definition 3.16 (Guard) *A guard is a conjunction or disjunction of delay constraints.*

Lists are often used in feature structure grammars. Guarding can be attached to list $append$ in order to assure termination. We recall the program for appending two lists together in example 65. We have seen that a depth-first search will not terminate if either the first or third arguments of `append` are variable, because the search will continue to push down on the recursive call in the second clause, given that each of these lists has a variable tail. A guarded version of `append` ensures that either of these two lists has been instantiated before the clause is selected. A guarded version of `append` using SICStus Prolog appears as in example 67. The semicolon is used for disjunction of constraints.

(67)
```
append([ ],Xs,Xs).

append([X|Xs],Ys,[X|Zs]) <--
  when((nonvar(Xs);nonvar(Zs)), append(Xs,Ys,Zs)).
```

Definition 3.17 (Guarded Rule) *A guarded rule has the syntax* $Gd|A \leftarrow (\phi \wedge B_1 \wedge B_2 \wedge \ldots \wedge B_n)$ *where Gd is a guard,* ϕ *is a constraint and* $A, B_1, B_2, \ldots, B_n$ *are goals.*

Using the syntax in definition 3.17 the rule in example 67 looks as in 68.

program P:

```
append([],X,X).
append([X|Xs],Ys,[X|Zs]) <-- when((nonvar(Xs);nonvar(Zs)),
append(Xs,Ys,Zs)).
```

goal G = $append(Q1, [c, d], Q2)$
$\downarrow$
$append([X|Xs1], [c, d], [X|Ys1])\{Q1 = [X|Xs1], Q2 = [X|Ys1]\}$
$\downarrow$
$(nonvar(Xs1); nonvar(Ys1))|append(Xs1, [c, d], Ys1)$ This goal is suspended since `Xs1` and `Ys1` are variable.

goal G = $append(Q1, [c, d], Q2)$
$\downarrow$
$append([], [c, d], [c, d]]).\{Q1 = [\,], X = [c, d], Q2 = X\}$

Figure 3.6: Search tree with a suspended goal. The infinite search path is avoided by putting a guard on the goal which could trigger it.

(68)
```
nonvar(Xs);nonvar(Zs) | append([X|Xs],Ys,[X|Zs]) <--
    append(Xs,Ys,Zs)
```

The guard Gd is `nonvar(Xs);nonvar(Zs)` which says that the list argument `Xs` or the list argument `Zs` must be non-variable (instantiated) before the CLP goal `append([X|Xs],Ys,[X|Zs]) <-- append(Xs,Ys,Zs)` is solved for. The constraint ϕ is understood to be either $true$ or the same as Gd. We can now solve the problem $append(Xs, [c, d], Ys)$ from figure 3.4 without going into an infinite search path, as shown in figure 3.6.

3.2.2 Resolution with Guarded Rules

Resolution with CLP clauses can be represented as a state space search. We have as a state a pair $\langle G_i, C_i\rangle$ where G is a CLP goal and C is a set of constraints in the constraint store.

(69) $\langle G_0, C_0\rangle \longrightarrow \langle G_1, C_1\rangle \longrightarrow \ldots \longrightarrow \ldots \langle G_n, C_n\rangle$

Definition 3.18 (Derivation Step for CLP search) *A search state for CLP is a pair $\langle G_i, C_i\rangle$ where G is a CLP goal and C is a set of constraints in the constraint*

store. A derivation step $\langle G_i, C_i\rangle \longrightarrow \langle G_{i+1}, C_{i+1}\rangle$ *for the search is defined as follows:*

1. Let G_i *be* $\phi, B_1, B_2, \ldots, B_n$*. Then* C_{i+1} *is* C_i *plus* ϕ*. If* C_{i+1} *is false, then* G_2 *is the empty goal and the state is a fail state. Otherwise,* G_{i+1} *is* $B_1, B_2, ..., B_n$*.*

2. Let G_i *be* $B_1, B_2, \ldots, B_n$*. Then* G_{i+1} *is* $B_1, B_2 \ldots, B_{i-1}, B_{i+1}, \ldots, B_n$ *plus a rewriting for* B_i *such that* B_i *is the head of a CLP clause and the rewriting is the right hand side of that clause. The clause is rewritten using a substitution* θ *of values for variables.* C_{i+1} *is* $C_i(\theta)$*.*

Definition 3.19 (Success and Failure States for CLP search) *A success state is* $\langle G, C\rangle$ *where G is the empty goal and C≠false. A fail state is* $\langle G, C\rangle$ *where G is the empty goal and C=false. If G is a goal, then an* answer *for G is a derivation for the state* $\langle G, true\rangle$*. (Marriott & Stuckey 1998)*

Definition 3.20 (Suspension) *A suspended state* $\langle\langle Gd, G\rangle, C\rangle$ *is a guarded goal* G *plus a constraint* C*, such that the guard* Gd *is not satisfied by the constraints in* C *plus a substitution* θ *of values for the variables in* G *and* C*.*

Search with guarded rules proceeds much the same as regular CLP search. The difference is that if a goal is suspended, search cannot continue from this point. Search must proceed with a goal that is not suspended. A suspended goal does not necessarily halt the search; it simply removes that goal from immediate consideration. Search continues with goals whose guards can be satisfied. If another goal enables the guard on the suspended goal to be satisfied, search can resume by selecting the suspended goal next. Some systems automatically return to the suspended goal as soon as a variable substitution θ triggers it, even before it has been determined whether the goal which triggered the unsuspension succeeds with the same substitution.

Definition 3.21 (Qualified Answer) *A derivation* $\langle G_0, C_0\rangle \longrightarrow \langle G_1, C_1\rangle \longrightarrow \ldots \longrightarrow \ldots \langle G_n, C_n\rangle$ *is successful if* $C_n \neq false$ *and either* $\langle G_n, C_n\rangle$ *is suspended or* G_n *is the empty goal. If* G_n *is suspended then* $\langle G_n, C_n\rangle$ *is a* qualified answer *to the goal* $\langle G_0, C_0\rangle$*.*

(70) program P:

```
ne(X,Y) <-- not(X=Y).
when((ground(X),ground(Y)), ne(X,Y)).
```

goal G: X=2,ne(X,Y)

G_0: $\langle ne(X, Y), X = 2\rangle$. (Marriott & Stuckey 1998: chapter 9)

There are no transitions available from the state G_0 in example 70 because Y is not ground. Therefore, $\langle ne(X,Y), X = 2\rangle$ is a final state. Since G_0 is not empty then $\langle ne(X,Y), X = 2\rangle$ is a qualified answer to the goal G.

In order to prefer suspended goals over goals that have not been suspended, we can add a third element to the state space, in order to keep track of which goals are suspended and which are not. The new variable S stands for a list of suspended goals. The procedure is shown in figure 3.7. It is this algorithm that we will adapt for feature structures in chapter 5. Goals are moved to the list S as they are found to be suspended, and their reduction is put back on G when they are unsuspended. The algorithm is simplified in that it does not handle the case of automatically triggering suspensions by linking them together.

3.2.3 An Example from NLP

van Noord & Bouma (1994) introduce the technique of using Prolog delaying in a linguistic context, to handle adverbial adjunction to verb clusters in Dutch. In their analysis, an adjunct adjoins to a verb phrase to create a new verb phrase with an additional element on the complements list. The linguistic phenomenon in support of the analysis is that, in Dutch subordinate clauses, the arguments of a main verb as well as adverbials can be realized to the left of an intervening auxiliary verb, such as a modal verb.

(71) a. dat Arie Bob cadeautjes wil geven
that Arie Bob presents wants to-give
that Arie wants to give presents to Bob

b. dat Arie Bob zou moeten kunnen willen kussen
that Arie Bob should must can want to-kiss
that Arie should be able to want to kiss Bob

c. dat Arie Bob vandaag wil kussen
that Arie Bob today wants to-kiss
that Arie wants to kiss Bob today (van Noord & Bouma 1994: examples 5,7,11)

Applying the technique shown in our example 67, van Noord & Bouma delay the concatenation of verb arguments if the tail of the verb's subcategorization list is variable. Recall that verb arguments may be underspecified if they have been raised by an auxiliary, as occurs in several linguistic analyses including Hinrichs & Nakazawa (1989). Refer to chapter 2, section 2.1.

A search state for guarded CLP is a triple $\langle G, S, C\rangle$ where G and S are CLP goals and C is the constraint store. Transition from a state $\langle G_i, S_i, C_i\rangle \longrightarrow \langle G_{i+1}, S_{i+1}, C_{i+1}\rangle$ is defined as follows:

1. Let CLP goal S_i be $B_1, B_2, \ldots, B_n$.
   B_i is the head of a CLP clause with guard Gd.
   - If $Gd = true$ then S_{i+1} is $B_1, B_2 \ldots, B_{i-1}, B_{i+1}, \ldots, B_n$.
     G_{i+1} is G_i plus a rewriting for B_i, such that the rewriting is the right hand side of the clause headed by B_i. The clause is rewritten using a substitution θ of values for variables.
     C_{i+1} is $C_i(\theta)$.
   - If $Gd = false$, then S_{i+1} = S_i minus B_i. G_{i+1} is the empty goal and C_{i+1} is false.
   - If Gd is unknown, then there is no transition from this state.
2. Let CLP goal G_i be $\phi, B_1, B_2, \ldots, B_n$. Then C_{i+1} is C_i plus ϕ. If C_{i+1} is false, then G_{i+1} is the empty goal and the state is a fail state. Otherwise, G_{i+1} is $B_1, B_2, ..., B_n$ and $S_{i+1} = S_i$.
3. Let CLP goal G_i be $B_1, B_2, \ldots, B_n$.
   B_i is the head of a CLP clause with guard Gd.
   - If $Gd = true$ then G_{i+1} is $B_1, B_2 \ldots, B_{i-1}, B_{i+1}, \ldots, B_n$ plus a rewriting for B_i such that the rewriting is the right hand side of the clause headed by B_i. The clause is rewritten using a substitution θ of values for variables. C_{i+1} is $C_i(\theta)$, and S_{i+1} is S_i.
   - If $Gd = unknown$, then B_i is suspended. G_{i+1} is G_i minus B_i. S_{i+1} = S_i plus B_i.
     C_{i+1} = C_i.
   - If $Gd = false$, then G_{i+1} is the empty goal and C_{i+1} is false.

Figure 3.7: Search with Guarded Rules. This search prefers the unsuspension of suspended goals over the execution of goals which have not been suspended. Search proceeds as follows: For S the set of suspended goals not empty, choose a goal from S, until all goals have been tried. Then choose a goal from G. Repeat until all states are fail states or there are no more goals in G.

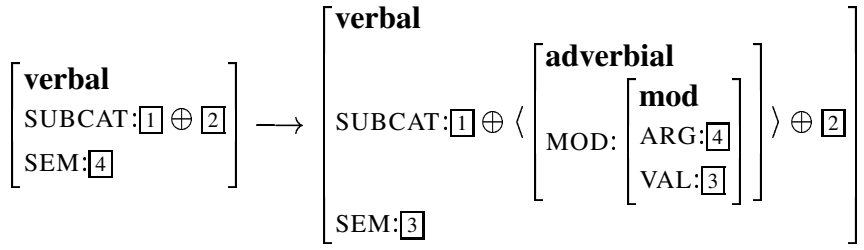


Figure 3.8: A lexical rule that adds an adjunct to the subcategorization list of a verb. The input to the rule is a verb template which is used to parse the phrase *Arie Bob wil kussen*, while the output is used for *Arie Bob vandaag wil kussen*. In both cases, 1 is the object *Bob* and 2 is the subject *Arie*. Separate rules order the elements of the phrase. (van Noord & Bouma 1994).

Delays are applied to the Prolog relation `add_adj` (for "add adjuncts"). The code is shown in 72, while the accompanying HPSG lexical rule is in figure 3.8. In the two-place predicate in 72, the first argument is the feature structure for the verb sign, and the second argument is the feature structure for the adverbial sign. In the four-place predicate, the first two arguments are the subcategorization lists of the verb and adverbial, and the third and fourth arguments are the semantics of the same. We are looking in particular at the subcategorization lists in the first and second places of the four-place predicate.

(72) add_adj([sign; SUBCAT:A; SEM:B; SUBJ:$Subj$], [sign; SUBCAT:J; SEM:K; SUBJ:$Subj$]) ← add_adj(A,J,B,K).

add_adj(⟨⟩,⟨⟩,A,A).

add_adj(⟨C—D⟩,⟨C—E⟩,A,B) ← add_adj(D,E,A,B).

add_adj(A,⟨ [adverbial; MOD: [mod; ARG:B; VAL:E]] —D⟩,B,C) ← add_adj(A,D,E,C)

(van Noord & Bouma 1994: Figure 8)

For the four-place predicate, delaying is achieved by using the Prolog `block` declaration.

(73) `:- block add_adj(?,_,??)`

This declaration says that the `add_adj` goal is blocked if the second argument is variable. This prevents an infinite chain of pushing adjuncts onto an verb subcategorization list with a variable tail. This is semantically equivalent to using the `when` predicate introduced in section 3.2.1:

(74) `when(nonvar(D),add_adj(A,D,E,C)).`

Other linguistic phenomena are handled similarly, including the addition of adjuncts to the subcategorization list of a verb and long distance dependencies. The context of the delaying operation is lexical rules. This implementation does not precompile lexical rules but rather handles lexical rules as complex constraints on lexical categories. The delay statements are fulfilled during parsing as subcategorized elements are realized. We will show a way to handle delay statements in the context of compilation of lexical entries before parsing.

3.3 Other Techniques for Constraint Solving

3.3.1 Constraint Satisfaction and Propagation

Constraint satisfaction is an entire area of concentration in Artificial Intelligence that focuses on obtaining a result given an initial set of constraints. Planning and scheduling problems are often solved by constraint satisfaction. An initial set of independent constraints is known and yet the output, such as an effective plan or an optimized schedule, is difficult to obtain.[1] Constraint propagation is a technique used within the scope of constraint satisfaction for sharing information among constraints and quickly eliminating unsatisfiable hypotheses for variable assignment, often by focussing on a small subset of the problem.

Constraint propagation is used in CLP when there many constraints in the constraint store. Constraint propagation can not only rule out bad hypotheses, but also reduce the search space by making the arguments of calls to goals more specific. It can be used as an alternative to co-routining (delaying those predicates which are not specific enough) or in conjunction with it. Meurers & Minnen (1995) describe compilation of lexical rules into definite clause predicates. They write

> ... one is forced to execute the call to [the predicate] directly when the lexical entry is used during processing, independent of the processing

[1] Koller & Niehren (2000) point out that these problems are notoriously combinatoric, and typically NP-hard.

> strategy used. Otherwise, there is no information available to restrict the search space of a generation or parsing process.

The authors write that they would prefer to perform lexical expansion "on the fly',' or sometime after lexical lookup. They add the technique of off-line constraint propagation (Meurers & Minnen 1996). Their methodology is to use automatic methods to identify those places in the theory where linguistically motivated underspecification would lead to inefficient processing, and then apply constraint propagation at those places, which results in more specific arguments. The authors point out that even delayed goals can be used in the propagation step.

3.3.2 Concurrent Constraint Programming

Concurrency allows more than one process to run simultaneously. In Concurrent Constraint Programming (Saraswat 1993) user-defined constraints are viewed as *processes*, and a state is regarded as a network of processes linked through shared variables via the constraint store. The constraint store contains all known information about variables and their values. Processes communicate by adding constraints to the store and synchronize by waiting for the store to enable a delay condition (Marriott & Stuckey 1998).

Oz (Smolka 1995) is a programming language with concurrency. Oz handles constraints and AVMs (attribute value matrices) as terms, which are unifiable via term unification.[2] The term AVM is used for the familiar feature-value pair notation, and is independent of a particular theory of feature structures. The conditional has blocking: a statement is true, false or suspended based on the constraint store. An OR statement blocks until only a single clause is left. The different disjunctions are solved in parallel. Constraint propagation and distribution are both used in Oz. Distribution is making a non-deterministic choice once the partial information about a variable assignment can no longer be improved. Oz has been used for expressing linear precedence constraints in German, which are parsed using dependency grammars (Hudson 1990, Duchier 1999). A good survey paper on constraint programming in Computational Linguistics, which includes concurrent constraint programming, is Koller & Niehren (2000).

[2]In Oz terminology, the terms are called predicates, but "predicates are values and thus can be assigned to variables."(Duchier *et al.* 1998)

3.3.3 Memoizing

An alternative to using delay constraints in parsing is to use the technique of memoizing. Chart parsers (Earley 1970) are memoizing parsers because active edges are available in the chart. Standard memoizing techniques are associated with variable copying because of the many attempts that are made to form new edges from existing edges. If an attempt eventually fails, the original edge must remain intact, hence the need for copying. In a memoizing implementation of CLP, a constraint is carried along dynamically, or stored without a need to recompute, during processing, until a point at which it can be effectively resolved. Johnson & Dörre's (1995) work of the mid-1990s combines the coroutining (delaying) facilities of Prolog with memoization. Johnson and Dörre formalize their approach within the outline of CLP given in Höhfeld & Smolka (1988). They suggest how their work might be used in chart parsing. Constraints can be propagated along with variable bindings. Because a goal consists of both relations and constraints, both can then associated with edges in the chart. By propagating constraints at the point of applying the fundamental rule (combining rule), the amount of variable copying is reduced, because certain combinations are ruled out. Johnson & Dörre's work is in the framework of constraint-based Categorial Grammar (Bouma & van Noord 1994).

3.4 Summary

This chapter provides a thorough introduction to the main ideas of Logic Programming and Constraint Logic Programming. The concepts of SLD resolution, satisfaction, negation as failure, and guarding have been defined. We look at feature structures in more detail in chapter 4, and then can apply the ideas of this chapter to feature structures in chapter 5. Related work discussed above includes constraint satisfaction, concurrency, and memoization.

Chapter 4

Feature Structures and Grammar Processing

This chapter introduces feature structures. After reviewing feature structures in linguistic theory, I introduce a notation for describing them (Carpenter 1992). I then review computational systems that process feature based grammars. These are in the family of Constraint Logic Programming (CLP) languages, which have been introduced in chapter 3. I note where delay techniques, which are part of CLP, have been incorporated into these systems. This thesis fits into the body of work in this chapter in two ways. First, it extends the feature structure description language by generalizing the language for delay statements to feature structure descriptions. Second, it describes an implementation which is a logic programming system in general, and in particular, an extension of the ALE system (Carpenter & Penn 1994) with delays.

4.1 Introduction

A number of feature-based grammar formalisms have been used for computation, including Functional Unification Grammar (FUG) (Kay 1979; Kay 1983; Kay 1985), PATR-II (Shieber *et al.* 1983), Lexical-Functional Grammar (LFG) (Bresnan 1982b), Generalized Phrase Structure Grammar (GPSG) (Gazdar *et al.* 1985), and Head-Driven Phrase Structure Grammar (HPSG) (Pollard & Sag 1987; Pollard & Sag 1994). The combining operation for these grammars is unification, which is a method of uniting two terms, or in this case, two feature structures (refer to definition 3.7). Shieber (1986) provides a good overview of the first formalisms

for feature based grammars. Shieber divides these into two categories, the *tool* type (e.g. PATR-II, FUG, DCG (Definite Clause Grammars) (Pereira & Warren 1980)) and the *theory* type (e.g. LFG, GPSG). He reviews the fact that tools for computation generally possess general mechanisms for expression, while theories incorporate "very specific" mechanisms related to a particular type of linguistic analysis. It is this distinction that we will address. We will explore what type of tool or tools are appropriate for implementing modern feature based grammars. An overview of the theory we will use is in section 4.2. We extend this theory in chapter 5. A review of the tools that have been developed for processing feature structures, using logic programming, is in section 4.3. We will be extending one of these, the ALE system (Carpenter & Penn 1994), in chapter 6.

4.2 Formalization of Feature Structures

Carpenter (1992) provides a theory for the specification and implementation of typed feature structures. Carpenter employs a general attribute-value logic, following Rounds and Kasper (Kasper & Rounds 1986; Kasper & Rounds 1990), in order that his work may be applied to the domains of phrase structure grammars, definite clause programs, and general constraint resolution systems. Feature structures in Carpenter's sense can be represented by directed, finite, labeled attribute-value graphs. Each node in a feature structure has a *type*, and a type is appropriate for a feature structure relative to a *type signature*, also called a type inheritance hierarchy.

Definition 4.1 (Type Inheritance Hierarchy) *An inheritance hierarchy is a finite bounded complete partial order* $\langle \mathbf{Type}, \sqsubseteq \rangle$ *(Carpenter 1992: Definition 2.1)*

We assume a finite set **Type** of types, ordered according to their specificity. A type τ is said to inherit information from another type σ if τ is more specific than σ. This is written as $\sigma \sqsubseteq \tau$. If $\sigma \sqsubseteq \tau$, then σ is a *supertype* of τ, or inversely, τ is a *subtype* of σ.

A sample hierarchy for types of semantic objects in HPSG is shown in figure 4.1. In this figure, semantic objects can be of three subtypes: nominal objects, parameterized states of affairs, or *psoas*, or quantifiers. Psoas are associated with relations. We focus on the hierarchy of nominal objects and have left some details out of the other two branches of the picture. We will make use of the nominal hierarchy later on when we study the binding theory in chapter 6. At the type **nom_obj**, two features are introduced, and are inherited by all subtypes. The

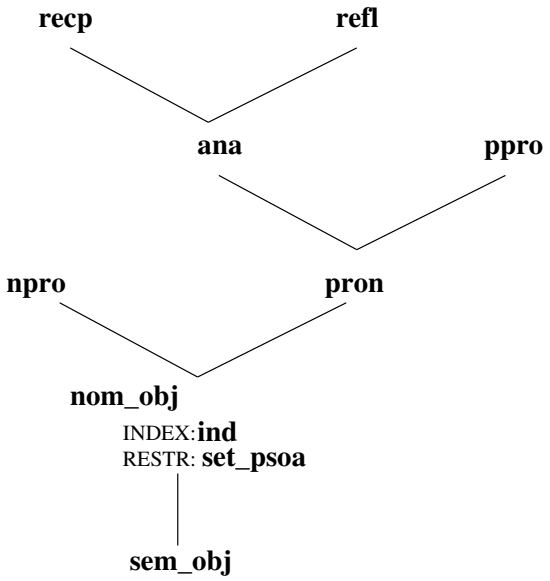


Figure 4.1: A type hierarchy for the types of nominals.

first is an INDEX, which is similar to the lettered subscripts used for reference in theoretical linguistics e.g. *John$_i$ saw himself$_i$*. The second is a set of semantic restrictions or RESTR on the noun, such as qualifying information. Nominal objects can be pronouns (**pron**) or non-pronouns (**npro**). Pronouns can be anaphors (**ana**) or personal pronouns (**ppro**), and anaphors are reciprocal (**recp**) or reflexive (**refl**).

The interesting thing for us about feature structures is that they may provide only partial information about types and features. The case of partial information is a result of the fact that the function δ, which computes the value of a path at a node, is a *partial function* (example 4.2). Partial information also results because there is underspecification of types and structure sharing. Viewed as a state of a process, a set of feature structures represents what is known about an object during each step of computation.

Definition 4.2 (Feature Structure) *For this definition we assume a finite set* **Feat** *of features and a type inheritance hierarchy* $\langle \mathbf{Type}, \sqsubseteq \rangle$.

A feature structure over **Type** *and* **Feat** *is a tuple* $F = \langle Q, \overline{q}, \theta, \delta \rangle$ *where:*

- Q*: a finite set of nodes rooted at* $\overline{q}$

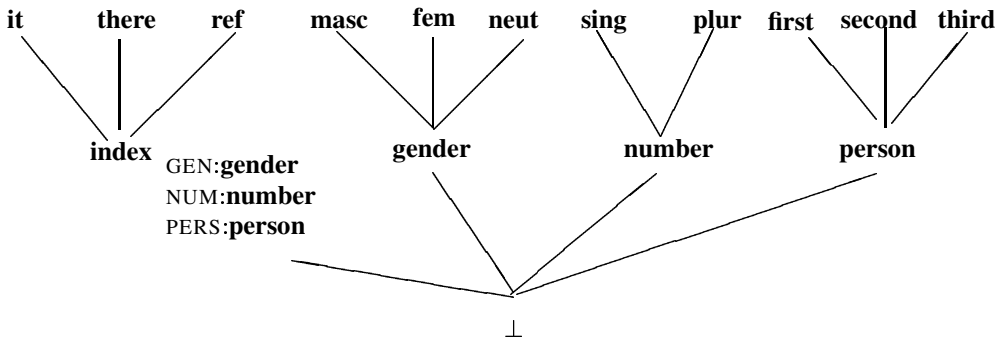


Figure 4.2: Hierarchy of types which are required to construct a nominal index. The sign $\perp$ or bottom is the most general or universal type.

- $\overline{q} \in Q$*: the root node*
- $\theta : Q \rightarrow \mathbf{Type}$*: a total node typing function*
- $\delta : \mathbf{Feat} \times Q \rightarrow Q$*: a partial feature value function*

Let $\{$ *denote the collection of feature structures. (Carpenter 1992: Definition 3.1)*

In the framework we have adopted, a description is a shorthand notation for a feature structure. We follow Carpenter (1992), after Kasper & Rounds (1986) in that a non-disjunctive description picks out a feature structure. That there is a most general feature structure structure that satisfies the description is a theorem which we will introduce in this section. Because of the morphism between descriptions and feature structures, descriptions and feature structures can be used interchangeably relative to a type signature. The description in 75 is relative to the type hierarchy in figure 4.2. A feature structure that is a most general satisfier for 75, i.e. a feature structure picked out by 75, is shown in 76:

(75) NUM:**sing** $\wedge$ PERS:**third**

(76)
$$\begin{bmatrix} \textbf{index} \\ \text{NUM:}\textbf{sing} \\ \text{PERS:}\textbf{third} \\ \text{GEN:}\textbf{gender} \end{bmatrix}$$

The features NUM, PERS and GEN are appropriate in a hierarchy of HPSG types for the type **index**, which is the type for a bundle of referential or index features

for nominal objects. With regards to this hierarchy, we can infer that the feature structure in 75 must be of type **index**. This is true even though the type is not explicitly mentioned in the description, because the features in the structure are appropriate for that type. Furthermore, the type **index** has subtypes appropriate to nouns or expletive pronouns. We do not know whether this bundle of features is for a noun or a pronoun, so the subtype is not resolved further by the description itself. Another way of saying this is that the type is *underspecified*. A last point to make is that, since the feature GEN is appropriate for the type **index**, it is part of the feature structure in 76, even though this feature is not given a value in the description in 75. This is in accordance with version of typing called *strong typing* which says that every feature that is appropriate for a type must be present. The gender is left unresolved to any of the sub-types.

We define descriptions:

Definition 4.3 (Description) *The set of descriptions over the collection* **Type** *of types,* **Feat** *of features and* **Var** *of variables is the least set* **Desc** *satisfying these conditions:*

- $\sigma \in \mathbf{Desc}$ *if* $\sigma \in \mathbf{Type}$
- $\bot \in \mathbf{Desc}$
- $x \in \mathbf{Desc}$ *if* $x \in \mathbf{Var}$
- $\pi : \phi \in \mathbf{Desc}$ *if* $\pi \in \mathbf{Path}$, $\phi \in \mathbf{Desc}$
- $\pi_1 \doteq \pi_2 \in \mathbf{Desc}$ *if* $\pi_1, \pi_2 \in \mathbf{Path}$
- $\phi \wedge \psi, \phi \vee \psi \in \mathbf{Desc}$ *if* $\phi, \psi \in \mathbf{Desc}$

(Carpenter 1992: Definition 4.1)

Subsumption and *satisfaction* are important notions not only for talking about feature structures in general, but also for explaining delays on feature structures. Subsumption is an ordering on feature structures. If a feature structure contains at least as much information as another, then the more general feature structure is said to *subsume* the more specific. For the definition of subsumption in 4.4 we assume that we are dealing with feature structures defined over a fixed inheritance hierarchy $\langle \mathbf{Type}, \sqsubseteq \rangle$.

Definition 4.4 (Subsumption) $F = \langle Q, \bar{q}, \theta, \delta \rangle$ *subsumes* $F' = \langle Q', \bar{q}', \theta', \delta' \rangle$ *if and only if there is a total function* $h : Q \rightarrow Q'$*, called a* morphism *such that:*

- $h(\bar{q}) = \bar{q}'$
- $\theta(q) \sqsubseteq \theta'(h(q))$ *for every* $q \in Q$
- $h(\delta(f,q)) = \delta'(f, h(q))$ *for every* $q \in Q$ *and feature f such that* $\delta(f,q)$ *is defined*

(Carpenter 1992: Definition 3.4)

We can order the feature structure in example 76 in the subsumption relation with other feature structures, using the type hierachy for index features in figure 4.2. In 77a, the type **masc** of the gender feature GEN is more specific on the right hand side of the relation than on the left hand side. In 77b the feature values have not changed, but the subtype of **index** has been resolved to **it**.

(77) a.
$$\begin{bmatrix} \textbf{index} \\ \text{NUM}:\textbf{sing} \\ \text{PERS}:\textbf{third} \\ \text{GEN}:\textbf{gender} \end{bmatrix} \sqsubseteq \begin{bmatrix} \textbf{index} \\ \text{NUM}:\textbf{sing} \\ \text{PERS}:\textbf{third} \\ \text{GEN}:\textbf{masc} \end{bmatrix}$$

b.
$$\begin{bmatrix} \textbf{index} \\ \text{NUM}:\textbf{sing} \\ \text{PERS}:\textbf{third} \\ \text{GEN}:\textbf{gender} \end{bmatrix} \sqsubseteq \begin{bmatrix} \textbf{it} \\ \text{NUM}:\textbf{sing} \\ \text{PERS}:\textbf{third} \\ \text{GEN}:\textbf{gender} \end{bmatrix}$$

An isomorphism is a mapping between two feature structures such that they subsume each other. We will discuss isomorphism more formally in chapter 6, where it is used in implementation. A feature structure F will always be subsumed by infinitely many other feature structures, but these may not be unique with respect to isomorphism. The most general of all of these, that is, the one with the least specific information, is the *most general satisfier* for F. The feature structure in (76) is the most general satisfier of the description in (75). Due to the correspondence between feature structures and non-disjunctive descriptions, satisfaction is actually defined as a relation between a feature structure and a description. We say that a description ϕ has a most general satisfier, where we understand that ϕ may be "satisfied" or semantically implied by many features structures.

Subsumption is a syntactic notion, and satisfaction is its semantic counterpart. If a feature structure is satisfied, then all feature structures that it subsumes are also satisfied. The formal definition of satisfaction is in 4.5. θ is a substitution operating on a variable x.

Definition 4.5 (Satisfaction conditions for feature structures)

- $F \models^{\theta} \sigma$ *iff* **Type***(F)* $\sqsubseteq \sigma$
- $F \models^{\theta} x$ *iff* $\theta(x) = F$
- $F \models^{\theta} \pi : \Phi$ *iff* $F@\pi \models^{\sigma} \Phi$
- $F \models^{\theta} \pi_1 \doteq \pi_2$ *iff* $F@\pi_1 = F@\pi_2$
- $F \models^{\theta} (\Phi, \Psi)$ *iff* $F \models^{\theta} \Phi$ *and* $F \models^{\theta} \Psi$
- $F \models^{\theta} (\Phi; \Psi)$ *iff* $F \models^{\theta} \Phi$ *or* $F \models^{\theta} \Psi$

(Carpenter 1992: Definition 4.2)

Theorem 4.6 (Non-Disjunctive Most General Satisfier) *There is a partial function* $MGSat : NonDisjDesc \rightarrow \mathcal{F}$ *such that:*
$F \models \phi$ *(read "F satisfies* ϕ*") if and only if* $MGSat(\phi) \sqsubseteq F$ *(Carpenter 1992: Theorem 4.5)*

Theorem 4.6 says that, for each description Φ_n, there is a feature structure F_n such that F_i is the most general satisfier of Φ_i. Given θ, a substitution of values for variables in the feature structure F, then $F_0 \models^{\theta} \Phi_0$ and $F_j \models^{\theta} \Phi_j$. The function $MGSats$ returns a set of possible satifiers, in case the description is disjunctive. This means finding a satisfier for at least one of the description's disjuncts.

Theorem 4.7 (Disjunctive Most General Satisfier) *There is a total function MGSats mapping descriptions to sets of pairwise incomparable feature structures such that* $F' \models \phi$ *if and only if* $F \sqsubseteq F'$ *for some* $F \in MGSats(\phi)$. *(Carpenter 1992: Theorem 4.7)*

4.3 Logic Grammar Systems

In this section we move from the language used to describe feature structures to systems that have been built to interpret them (Shieber's "tools"). I start with early systems for feature structure processing, including FUG (Kay 1983; Kay 1985), PATR-II (Shieber *et al.* 1983), and LOGIN (Aït-Kaci & Nasr 1986). I then move on to more recent systems for feature structure processing, including the work of Dörre & Dorna (1993), Erbach *et al.* (1995), Carpenter & Penn (1994), Elhadad *et al.* (1997), Götz *et al.* (1997), and Dörre & Manandhar (1997). These systems are frameworks for resolving constraints on feature structures. These

have included delay functions, in order for the grammar writer to impose some control on specific feature values. The goal of this thesis is to allow more freedom with respect to the placement of constraints in the grammar, and to remove infinite resolution chains in particular. I also note the LinGO system (Copestake *et al.* 1999; Copestake & Flickinger 2000) because this system is among the largest and most widely used system for processing HPSG grammars, although this is not a delaying system. However, none of the applications of the logic grammar systems discussed here is truly large-scale. The main and crucial difference is the motivation for the systems. Using a theoretical, grammar-based approach, one attempts to achieve complete and accurate grammatical coverage for part or all of a theory of language. In contrast, a particular corpus usually motivates commercial and statistical development systems, and leads to larger grammars and lexicons.

4.3.1 Early Work

Constraint Logic Programming includes the Prolog family of programming languages. Colmerauer and colleagues (Colmerauer *et al.* 1973) developed these languages for NLP, and definite clause grammars (DCGs) fall out of their definite clause structure. One of the first systems making use of Prolog for NLP was the CHAT-80 system (Pereira & Warren 1980).

Basic formalisms for feature structures appeared in the late 1970's and early 1980's. The Functional Unification Grammar (FUG) (Kay 1983; Kay 1985) and the PATR-II system (Shieber *et al.* 1983) were both general systems for expressing feature structures, with unification as the combining operation. In PATR-II, feature structures are expressed as directed acyclic graphs, or *dags*. In FUG, the feature structure is called a *functional structure*, and the features are regarded as functions. FUG allows re-entrancy (shared values), and extends the formalism by adding features with special interpretations in their unificational behavior. Feature structures were also part of the linguistic theory of LFG (Kaplan & Bresnan 1982), which began during this same time period.

At the same time that tools for writing and manipulating feature structures were becoming more sophisticated, a line of research in logic programming led to the use of feature structures, and typed feature structures in particular, as terms in a logic programming language. Aït-Kaci showed how to add a type inference operation to a unification process (Aït-Kaci 1984). Aït-Kaci & Nasr's (1986) LOGIN language is an example of a definite clause language over typed feature structure terms, with type consistency checking. Earlier knowledge representation

systems such as KL-ONE (Brachman & Schmolze 1985) inspired Aït-Kaci, and led the way for the implementation of typed linguistic theories such as HPSG.

The next set of systems, in use during the 1990's and into the present, are presented in alphabetical order by system name.

4.3.2 Advanced Linguistic Engineering Platform (ALEP)

Erbach *et al.* (1995) devised constraint solvers for grammar development as part of the Advanced Linguistic Engineering Platform (ALEP). Their premise is that while the formalisms used in computational linguistics can encode very high-level constructs, grammar development environments have provided only basic features. The authors have provided grammar developers with tools for manipulating sets and managing constraints. Sets are used in computational linguistic theories to express such things as the quantifier store and the SLASH set of extracted elements. The ALEP project allows manipulation of sets by providing operations on sets, such as set disjointness and set union. Prolog typically operates on lists. The constraints in this project include an extension of the guarded constraints of the LIFE system (Aït-Kaci & Podelski 1994), which like the Oz system (Smolka 1995) include functional terms, or predicates. The ALEP tool set extends guards to include guards on linear precedence constraints and guards on set constraints (e.g. set membership). The extensions were implemented as Prolog modules and may be used either standalone or integrated with existing grammar formalisms. The work of Erbach *et al.* (1995) has been used in the implementation of German word order.

4.3.3 Attribute Logic Engine (ALE)

The Attribute Logic Engine (Carpenter & Penn 1994) integrates phrase structure parsing and constraint logic programming, using typed feature structures as terms. The theory behind ALE is (Carpenter 1992). ALE allows type inheritance and appropriateness specifications for features and values. ALE was designed to generalize logic programming over PATR-II style feature structures in a Rounds-Kasper logic (Kasper & Rounds 1990). It allows for the specification of HPSG grammars with complete detail (Pollard & Sag 1987; Pollard & Sag 1994). The phrase structure component of ALE allows definite clause attachments to rules and includes a lexical rule component. Grammars may interleave unification steps with logic programming calls, allowing parsing to be interleaved with other system components. Currently ALE allows for the evaluation of inequations. Penn's (2000)

extension to the system, called TRALE, includes delaying as implemented in the Constraint Handling Rules, or CHR package of SICStus Prolog. Delays may be done on inequation-free and function-free descriptions which are characterized as Prolog terms. This differs from the ALE terms described in (Carpenter & Penn 1996), which are the basis for the implementation in this thesis. While theoretically elegant, there are drawbacks to his encoding related to efficiency. The term encoding available for typed feature structures, which uses attributed variables, does not have any notion of appropriateness. Also, direct encoding of delays using Prolog `when` is found to run much faster (60 times) than using the CHR library for delays.

4.3.4 Categorial Grammar Frameworks

Ingria (1990) showed how unification is inadequate for a theory of coordination. It is impossible for the feature specification of a conjunctive NP to be consistent with both conjuncts, if they are inconsistent with each other (e.g. dative and accusative NPs are inconsistent). In an account based on subsumption rather than unification, the use of partial agreement feature specifications plays a different role than that of being a shorthand for a fully specified feature structure. It indicates rather a feature set from which the types of the conjuncts may be implied. Given a logical interpretation, the feature set of the two conjuncts subsumes each of them. Following Bayer & Johnson (1995), Dörre & Manandhar (1997) give rules for entailment in the context of a Categorial Grammar style, constraint based feature grammar. The authors use subsumption checking in the binding of their arguments. This is motivated by linguistic data from coordination, and enables a Lambek style proof theory. The logic they give for their simple feature based system includes rules for delaying if neither entailment nor disentailment can be proven. In chapter 5 of this thesis I will also give rules for entailment, disentailment and delay in the context of a full unification grammar.

4.3.5 Constraint Unification Formalism (CUF)

The CUF system of Dörre & Dorna (1993) is characterized by these authors as being roughly a feature structure description language similar to Kasper/Rounds logic (Kasper & Rounds 1990), combined with the possibility of stating definite clauses over typed feature terms. A **sort** in CUF is feature term similar to a type, but it can take variable arguments. Since a sort has definite clause attachments,

it is used as a goal in resolution. Delay patterns are part of CUF. Delay patterns have this format:

(78) **delay**(*sort/arity, Delayed Parameters*).

The delay parameters are integer indices into the parameters (arguments) of the sort. If the value of the path $f_1 : \ldots : f_n$ at the i^{th} parameter is uninstantiated, then evaluation of the sort is delayed. If the list of delay specifications has more than one element, e.g., for different parameters or different paths of one paramenter, the statements mean a conjunction of the delay conditions. Disjunction of delay conditions is expressed using more than one delay pattern for the same sort.

An example of delayed *append* is in 79.

(79) The sort `append/2` is defined as:

```
append(list,lkist) -> list.
append([], L) := L.
append([F|R], L) := [F|append(R,L)].
```

and the corresponding delay pattern is:

```
delay(append/2,[0,1])
```

which indicates that either the first parameter or the result (numbered 0) must be instantiated. (Dörre & Dorna 1993: examples 11 and 13)

SLD resolution in the context of CLP (resolution plus constraint satisfaction) is the proof strategy used in CUF. As the authors note,

> ... SLD-resolution describes only the general scheme of the proof. The actual strategy for goal selection, i.e. the computation rule, has a dramatic influence on the size of the search space that has to be considered. It is this computation rule that is refined by the addition of delay statements. As a general strategy the computation rule employed in the CUF system always selects *deterministic goals*, if such goals are present. A deterministic goal is a goal for which no choice point needs to be introduced, i.e. for which only one clause has satisfiable constraints. Only if no such goals are leftover the left-most nondeterministic goal which is not delayed according to a delay statement will be expanded. The search induced by the nondeterminism is performed depth-first, using chronological backtracking. In the exceptional case

> where all goals are delayed, the first one of the goal list will be expanded, optionally indicating this condition to the user. The user may restrict the recursion using a depth bound. This guarantees termination, but the proof procedure is no longer complete.(Dörre & Dorna 1993: p. 19)

Simple variable equality or inequality can be checked with variable unification, as with Prolog `when`. CUF also delays implicitly in the solving of term inequality constraints. We will show how this same strategy works for solving inequations between feature structures.

4.3.6 ConTroll

ConTroll is a logic grammar system that allows for the solving of constraint based grammars (Götz *et al.* 1997; Götz & Meurers 1997; Götz & Meurers 1998). ConTroll and ALE are both based on a typed feature logic with appropriateness conditions for the domains and ranges of features (for ConTroll cf King (1989)). Unlike ALE, ConTroll allows for implicational descriptions . Implication such as $\phi \rightarrow \psi$ is assumed to be the same as $\neg\phi \vee (\phi \wedge \psi)$. Thus, negation of features is compiled out to disjunctive normal form in the sense that the description $\neg\phi$ is assumed to be $\psi \vee \upsilon \vee \ldots$ for every type which is not consistent with ϕ. The drawback to this approach is that it results in a system in which every possible disjunction of "species" (King's terminology for a partitioning of the objects in the domain) is represented by a type. As Carpenter (1992) points out, this means that there are 2^n types derived from King's construction if there are n species. In ConTroll this has been given some tools to delay compilation to disjunctive normal form or to "hide" it in an auxiliary relation. Götz & Meurers indicate that a debugger is an "indispensible" tool as grammars increase in size and the data structures become highly complex.

Delay statements are used in ConTroll to partially determine the order in which goals will be executed by the system. The `delay/2` delay statement takes a relation name as its first argument, and a delay pattern as its second. Delay patterns are a a subset of feature terms with no negation, relation calls or list syntax. An example is a delay on the argument of a relation

(80)
```
delay(append,arg1:(e_list ; ne_list))
```

The statement above says that the first argument must be more specific than

the plain `list` type, i.e., an instantiated list.[1] A second type of delay is applied to constraints on a type:

(81) `delay(phrase,subcat:list)`

This delays the principles on type **phrase** (also known as type constraints) until the subcategorization information is known. Another form of delay statement is `delay_deterministic/1`. It requires that a goal of a given relation only be executed if at most one clause matches it. The user is also allowed to delay a particular constraint by name.

A subsumption check determines when the delay pattern is "undelayed." This is the same check that is used on the antecedent of an implication by the Constraint Handler CHR which is part of SICStus Prolog (Frühwirth & Abdennadher 1997), i.e. the head of a rule is matched by a subsumption check (see also Penn (2000)). Implicational principles in ConTroll are compiled out to an implication with a type antecedent. The negation of the complex antecedent is added to the consequent, which can result in highly disjunctive specifications. Götz & Meurers (1997) indicate that their delay strategy, which prefers deterministic goals, is similar to the one in CUF, and which too is based on the Andorra Model (Haridi & Janson 1990). The authors say that CUF does not include what they call "universal principles" such as the principles of the linguistic theory HPSG. ALE can express these as definite relations; CUF allows for definite relations as well. The authors may be referring to the fact that unlike CUF, they have a separate syntactic construct for implicational constraints with complex antecedents.

4.3.7 Functional Unification Formalism (FUF)

The Functional Unification Formalism (FUF) (Elhadad & Robin 1992; Elhadad *et al.* 1997) is a logic programming language with constraints, as an extended version of FUGs (Kay 1979). The generator SURGE uses FUF in the context of English generation.

A wait statement in FUF specifies that the decision corresponding to a disjunction depends on the value of certain features. The waiting is applied to the value of a feature or set of features. An `alt` is a disjunction of possible unifiers for a functional description (feature structure). In example 82, the alt TP depends on the value of the feature P. If feature P is instantiated, then evaluate normally.

[1] It is not clear that one can delay until the first *or result* argument is more specific than **list**.

If P is not instantiated, the whole disjunction is delayed. It is put on hold, on an agenda.

(82) `(alt TP (:wait P) ...)`

Periodically, the unifier checks the agenda to determine if one of the frozen alts (alternatives in the disjunction) can be awakened. In FUF, this period is whenever a new choice point is met, or, whenever a top level disjunction is entered. The author notes that this needs further optimization. "I am experimenting with a "granularity" control system, which controls how often the agenda of frozen disjunctions is checked. This is currently not implemented." (footnote p. 137)

At the end of the unification stage, the determination stage checks if any decision is still on the agenda. This situation is reached if not enough information could be gathered to evaluate the frozen `alts`. Then evaluation is "forced," even though some of the requested information is missing. A unique agenda identifier is assigned to each frozen `alt`. Standard control flow in FUF is top-down, breadth-first, in a tree of constituents.

Floating constraints are semantic constraints on the outcome of generation which are context-sensitive. That is, they may be realized at different places in the syntactic grammar, depending upon the input constraints. An example of a floating constraint is a manner adverbial. It can be realized as implicit in the verb as in *nipped* or as a verb plus an adverb as in *narrowly beat*. The default is to use an adverb. The code for this is shown as example 83. In this example, delay is achieved through using the keyword `:bk-class`, or backtracking class.

(83)
```
(manner ((alt manner-adverbial (:bk-class manner)
( ;; Can be realized by other means -- delay
(( manner-conveyed any))

;; Map manner to an adverbial adjunct
;; and mark that manner has been realized
(({adverb} ((synt-cat adverb) (concept { ^ ^ concept})))
(manner-conveyed adverb))))))

...)
```

The feature `manner-conveyed` is used to record the syntactic category of the constituent realizing the manner constraint. It remains `nil` as long as the constraint is not conveyed elsewhere in the syntax. This `alt` waits to generate manner as an adverbial as long as the `manner-conveyed` feature has some value (specified by the keyword "any"). If all of the locations for manner realization in the backtracking class (`:bk-class`) have been checked and there is still

no value, then manner defaults to assignment via an adverbial adjunct. The BK-CLASS mechanism is a way of restricting the points at which backtracking retries a search, to keep it tractable.

4.3.8 Linguistic Grammars Online (LinGO) and the LKB system

The Linguistic Grammars Online (LinGO) project at Stanford University includes a broad-coverage implementation of English using HPSG (Copestake *et al.* 1999; Copestake & Flickinger 2000). The LKB system on which the grammar runs is designed for efficiency and scalability.[2] This system supporting an English grammar of "over 15,000 lines of code" and a lexicon of about 5000 entries as of 2000.[3] On measure of size of implementations, this may be the most successful of the systems discussed in this section. This contrasts, however, with e.g. the KANT lexicon for Caterpillar, Inc., based on the LFG formalism, with some 70,000 lexical entries (Nyberg & Mitamura 1992; Kamprath *et al.* 1998). Another comparison is the statistical parser of Collins (Collins 1996), which has been trained on 40,000 sentences from a 1-million word corpus of the Wall Street Journal, available through the Penn Treebank(Marcus *et al.* 1993).

An implementation of the binding theory using the LinGO formalism is available online with the LKB system (Copestake *et al.* 1999). Much of the rules are commented out. The grammar is part of an implementation of Sag & Wasow's (1999) syntax textbook. The comments indicate that is costly to implement the argument realization constraint of Manning *et al.*, which is one of the constraints which motivates the work in this thesis, along with the binding constraints formulated as the Anaphoric Agreement Priciple.[4] Assuming a tradeoff between development cost or efficiency and theoretical accuracy, the quest is to bridge the gap, and ask what constructs could be added to practical systems to allow them to

[2]LKB formerly stood for Lexical Knowledge Base, but is now known as the LKB system for grammar development.

[3]Size of the grammar and lexicon are the standard measure of the scope of an implementation, though admittedly, the constraints in a constraint based grammar may not resemble context free rules. The number of entries in the lexicon generally is a measure of the breadth of corpora that could be covered by a system, though again, a constraint or inheritance based lexicon may have a smaller number of entries. The number of semantic concepts or lexical types would be the true measure of lexicon size in this case.

[4]The Anaphoric Agreement Principle states that coindexed elements must share the same agreement value.

handle theoretically interesting problems as well. We will look at delaying in this light.

4.4 Summary

The logic for feature structures has been introduced in this chapter. This includes definitions both for feature structures and for feature structure descriptions, which are a shorthand way of picking out a partial feature structure. Computational systems for processing feature-based grammars have also been introduced. Past and present systems for feature structure processing have included means for the grammar writer to wait on specific feature values in order to delay evaluation. These have provided a way for the grammar writer to have a hand in the control of the evaluation process. They have generally been restricted to waiting on a specific part of the feature structure, and may be limited to particular operations (e.g. list append).

In the next chapter, I present a formal description of delaying for feature structures. This is a complete description which follows the paradigm of Jaffar & Lassez (1987) and Höhfeld & Smolka (1988). The advantage of my approach over other approaches is that a feature structure description may be used as a delay term, which is both a compact and complete way to express a delay term. I also allow negation in the descriptions by allowing inequations. These waits are more general than waits on only types or feature values in that they employ the full description language, including conjunction, disjunction and inequality with other descriptions. Examples of the implementation follow in chapter 6.

Chapter 5

Guarded Constraints on Feature Structures

We have looked at a number of linguistic analyses which pose a challenge for constraint resolution. When arguments are shared, more than one head is providing the pieces of information required to process an argument during parsing or generation. This requires us to adopt a strategy for constraint resolution that is sensitive to the amount of information available at a given time. The paradigm of Constraint Logic Programming (CLP) has guarding in place as a way to combine constraint solving and goal resolution. In order to describe delaying for linguistic problems in a feature based framework, we need to have a language for the rule set and also for feature structure terms, which are the objects of the linguistic theory. The guarded rules which are part of CLP have been introduced in chapter 3. We have also introduced the logic of typed feat-ure structures in chapter 4. We have, then, all the tools that we need to start using linguistic descriptions as the guards on the evaluation of feature structure terms.

In this chapter I present two specific ways to describe guarding on feature structures. One is by using *guarded descriptions*, which adds to the expressive power of the description language, but does not affect the rule set. The other is via guarded rules, as have been introduced in chapter 3. I provide the operational semantics for guarded constraints on typed feature structures in the context of SLD resolution. I also describe how to write negations such that they are an instance of delaying. This will be important in linguistics when handling constraints which are written as statements of negation, such as inequations, as we will see in more detail in chapter 6.

In the work of Jaffar & Lassez (1987) and Höhfeld & Smolka (1988), the

attribute-value description language defines terms over which definite relations are solved, using a process of constraint resolution. It is possible to view Carpenter's (1992) feature structure description language as an instance of CLP in two different ways. On the one hand, arbitrary descriptions provide the constraints on feature structures, and feature structure unification is employed as the method of constraint resolution. The eventual goal is to find a feature structures with substructures that satisfy the constraints on their types. Or, one can employ a definite clause resolution scheme in which the clauses can be mapped e.g. to a phrase structure grammar. The mother and daughter nodes are feature structures, each of which has a mapping to a clause. The two ways of applying CLP have been combined in the ALE system (Carpenter & Penn 1994). We make use of the fact that either methodology provides an opportunity for adding delay constraints.

Starting with the notion of satisfaction from chapter 4, we want to define a constraint that can be used as a delay condition on the evaluation of a feature structure. The delay term that we use will be a feature structure description. We will ask whether the feature structure satisfies the description. Use of subsumption and satisfaction is related both to equality (do two feature structures subsume each other?) and to instantiation (do we have enough information to answer the question?), which are the primitives used to build up user-defined constraints in logic programming. It turns out that simple type checking (does a feature structure have a particular type?) is a particular instance of satisfaction. We return later to the question of whether we need to know if two feature structures are identical. Gerald Penn (personal communication) has also suggested asking whether a given feature structure is maximal, that is, fully instantiated. We would need an additional function `maximal(F)` to include that check in our constraint set.

5.1 Guarded Descriptions

One way to guard feature structures is to expand the definition of descriptions to include *guarded descriptions*. In this case it is descriptions, rather than rules, that are guarded. Of the two approaches, this one may be the more appealing representation of guarding on feature structures. First, it captures the fact that satisfaction is the relevant constraint relation for guarding. Second, we are able to use the regular algorithm for resolution, that is, an algorithm without guarded rules. The context is an NLP parsing or generation process, where we are incrementally building up feature structures that are answers to queries. Descriptions are the arguments of definite clause rules, and are a shorthand for feature structure

terms.

Definition 5.1 (Guarded Description) $\phi \rightarrow \psi; \upsilon \in \mathbf{Desc}$ *if* $\phi, \psi, \upsilon \in \mathbf{Desc}$

Definition 5.2 (Satisfaction Conditions for Guarded Descriptions) $F \models (\phi \rightarrow \psi; \upsilon)$ *iff*

- $F \models \phi$ *and* $F \models \psi$ *(entailment) OR*
- $(\forall F')F \sqcup F' \not\models \phi$ *and* $F \models \upsilon$ *(disentailment)*

Definition 5.2 says that in order for a feature structure F to satisfy a guarded description $\phi \rightarrow \psi; \upsilon$ then either F must satisfy both ϕ and ψ ($\phi \wedge \psi$) or F will not satisfy ϕ and it will satisfy υ.

We now look at a few examples of feature structures and the guarded descriptions that they satisfy. We start with the verb *are*, which can be the second person, present tense form of the verb *be* (*You are in school*). Or, it can be any plural form (*we are, you are, they are*). The AVM notation for a partially completed lexical entry for *are* is shown in figure 5.1. One way to express the possibilities for the person and number of the subject is by using a guarded description. We put a guard on the INDEX value of the subject. We say that the feature structure tagged with [2] in figure 5.1 satisfies the description in example 84:

(84) [2] $\models$ NUMBER:**plural** $\rightarrow$ **true** ; PERSON:**second**

This says that if my number is plural, then my person value is consistent with any value. Else, my person is second person.

Another example showing the satisfaction of a guarded description by a feature structure is the example of German determiners. The determiner *der* may be the masculine nominative, the feminine dative or genitive, or the genitive plural. An AVM for the German determiner is shown in figure 5.2. A noun is the value of the SPEC feature for the determiner. This is the noun that the determiner specifies for. The noun has a determiner as its specifier or SPR value (not shown). We guard on the LOCAL value for the noun ([1] in figure 5.2), which includes the head features and agreement features for the noun. These include the head feature CASE, as well as the INDEX feature values. We say that if the case of the NP is nominative, then the noun must be masculine. Else, the noun is feminine or plural. The guarded description is shown as 85.

(85) [1] $\models$ CAT:HEAD:CASE:**nom** $\rightarrow$ CONT:INDEX:GENDER:**masculine** ;
CONT:INDEX:(GENDER:**feminine** $\vee$ NUMBER:**plural**)

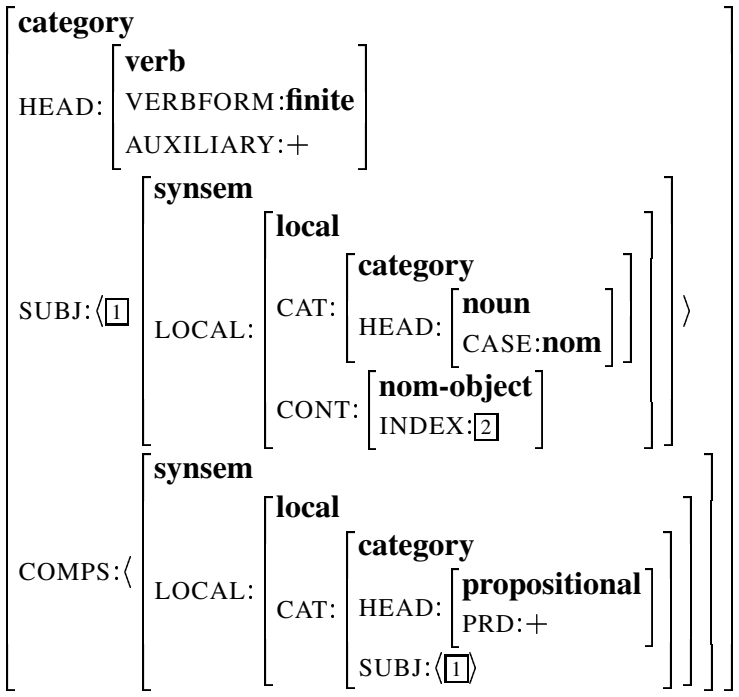


Figure 5.1: AVM notation for the syntactic category of the verb *are*. This is the auxilary form of the verb *to be* that takes a predicative complement (*I am happy, you are in trouble*). We will show how the person and number of the subject can be filled in with a guarded description for the index value 2.

We now consider the most general satisfier for guarded descriptions. As in chapter 4, we start with the non-disjunctive definition, and then move on to the disjunctive definition $MGSats$. In order to define the most general satisfier, we introduce a notion counter to it, that is to say, the set of most general inconsistent feature structures for a description ϕ. This is the set of feature structures that can't be extended to satisfy ϕ.

Definition 5.3 (Most General Inconsistent Feature Structure) *For a description ϕ, $MGIncons(\phi)$ is the set of most general inconsistent feature structures.*
$MGIncons(\phi) = \{F \mid \textit{for all } F', F \sqsubseteq F', F' \not\models \phi\}$

We now consider the case where the description may be disjunctive, and so we may have more than one most general satisfier for a description.

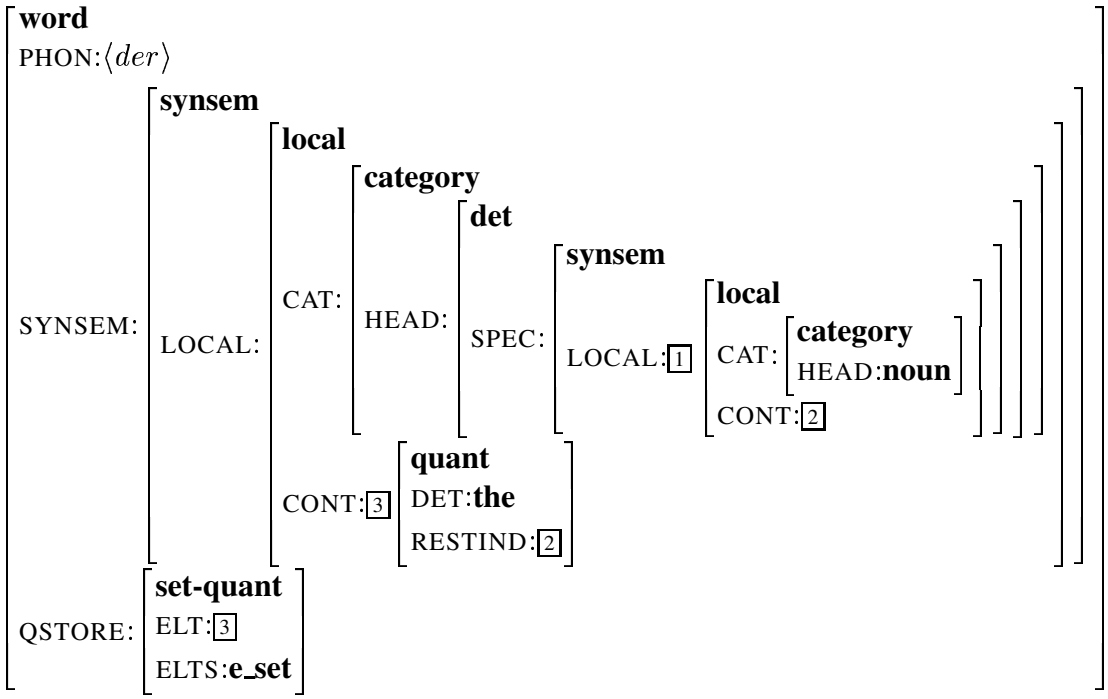


Figure 5.2: AVM for German determiner *der*

Definition 5.4 (Disjunctive Most General Satisfier for Guarded Descriptions) $MGSats(\phi \rightarrow \psi; \upsilon) = MGSats(\phi \wedge \psi) \cup \{F \sqcup G \mid F \in MGIncons(\phi), G \in MGSats(\upsilon)\}$

Procedurally, the way to think about guarded descriptions is to view them as if-then-else statements appropriate for variables. In the rules of the program, an unguarded description from the description language in definition 4.3 will not necessarily trigger a match on a guarded description. In other words, the satisfaction conditions for the description ϕ are not the same as the satisfaction conditions for the description $\phi \rightarrow \psi; \upsilon$.

Following Carpenter (1992) we define $Sats(\phi)$ as the set of all feature structures that satisfy ϕ.

(86) $Sats(\phi) = \{F \in \mathcal{F} \mid F \models \phi\}$

5.2 Using Descriptions as Constraints

In chapter 3, we introduced first logic programming and then logic programming with constraints. In this section we will show how the general algorithm for resolution with constraints can be applied in the context of feature terms. We use descriptions as the constraints on feature structures. The terms of the CLP rules are feature structure terms, and these terms may be constrained to satisfy particular descriptions in the constraint store. As long as the arguments in a goal are consistent with the constraints in the constraint store, the goal may be chosen from the resolvent.

Definition 5.5 (Constrained Feature Structure) *A constrained feature structure is a pair F:ϕ of a feature structure F plus a description $\phi \in$* **Desc***. F is said to be constrained to satisfy ϕ.*

We will use the notation F:ϕ for a constrained feature structure. We can use one or more of these as a constraint set in constraint logic programming. If F is one of the arguments of a goal in the logic program, F:ϕ indicates that F is constrained to satisfy ϕ. Goals and Clauses for CLP are defined as in chapter 3. So that we do not confuse a description ϕ with a constraint, we refer here to a constraint as C. First we use feature structures as the objects in a goal.

Definition 5.6 (Goal) *A goal is a expression of the form $p(f_1, \ldots, f_n), n \geq 0$ where p is a relation symbol and $f_i \in \mathcal{F}$.*

Definition 5.7 (CLP Goal) *A CLP goal is defined as C or $C, B_1, B_2, \ldots, B_n$, for C a (possibly empty) set of constrained feature structures $f_1 : \phi_1$, $f_2 : \phi_2$, ..., $f_n : \phi_n$, and $B_1, B_2, \ldots, B_n$ goals.*

When writing clauses in a program, descriptions of feature structures can be used to "pick out" the feature structure objects that will satisfy them.

Definition 5.8 (Clause) *For p a relation symbol and $\phi_n \in$* **Desc***, a clause is written $p_0(\phi_{0_1}, \phi_{0_2}, \ldots, \phi_{0_n}) \leftarrow p_1(\phi_{1_1}, \phi_{1_2}, \ldots, \phi_{1_n})$, $p_2(\phi_{2_1}, \phi_{2_2}, \ldots, \phi_{2_n}), \ldots, p_n(\phi_{n1}, \phi_{n2}, \ldots, \phi_{nn})$.*

We can attach constraints to a clause in the program. The feature structures in the constraints can be written using the variables available in the description language, e.g. Number:plural, Comps:ne_list.

Definition 5.9 (Constrained Clause (CLP Clause)) *A CLP clause is a clause plus a set C of constrained variables $X_0 : \phi_0$, $X_1 : \phi_1$,..., $X_n : \phi_n$. We write $C \mid Clause$.*

Guarded descriptions could be used in constraints, or even in the feature structures themselves. Resolution with guarded descriptions is an instance of logic programming. A description must first be satisfied by the current state of knowledge, or with the addition of information. Then, a goal constrained by this description can proceed. A description such as $(\phi \rightarrow \psi; \upsilon)$ is self-guarding, in that information about a term f_i and ϕ must be known before the guarded description is satisfied by f_i. There is no need for the addition of a guard. In this case, one would use the usual algorithm for resolution over ordinary goals (figure 3.1), but allow guarded descriptions anywhere in the goal descriptions.

Figure 5.3 describes the process of resolution with the addition of constrained feature structures in a constraint store. We assume that the constraint store $CStore$ has the value of true if all of the constraints in it are not false, else it has a value of false. In definition 5.10 we show how resolution is done step-wise, in case guarded descriptions are included in the constraint set.

Definition 5.10 (Resolution with Guarded Descriptions) *Given f_i, a feature structure, ϕ, ψ and $\upsilon \in$ **Desc**, goal $p(f_1, \ldots, f_i, \ldots, f_n)$, a (renamed) clause $C \mid A' \leftarrow B_1, B_2, \ldots, B_m$, such that goal p, A' and C unify with mgu θ, and C_i a constraint on f_i, $C_i = X_i : (\phi \rightarrow \psi; \upsilon) \in C$:*

- *If $f_i \models \phi$, then unify X_i with f' in $MGSats(\psi)$.*
- *If $f_i \not\models \phi$, then unify X_i with f' in $MGSats(\upsilon)$.*

If f_i neither satisfies nor dissatisfies ϕ, then goal p cannot unify with C_i, and so p cannot be selected from the resolvent.

Use of guarded constraints is part of a working grammar for local quantifier storage. The constraint (Pollard & Yoo 1997) says that the pool of quantifiers for a head comes from its thematic arguments.

(87) The POOL is the union of the QSTOREs of all *selected arguments*, defined as either

- thematic elements selected via the SUBJ or COMPS feature,
- elements selected via the SPR feature, or

Input: A logic program *P*
An input goal *G*
A constraint store *CStore*

Output: *G[θ] and CStore, if* $CStore \neq false$,
or *failure* if failure has occurred or $CStore = false$.
If $CStore = true$, then G[θ] is an answer for G.

Algorithm:

Initialize the resolvent to $\{G\}$.
Initialize *CStore* to $\{\ \}$.

While the resolvent is not empty do
Choose a goal *A* from the resolvent
and a (renamed) clause $C \mid A' \leftarrow B_1, B_2, \ldots, B_n, n \geq 0$, from *P*
such that *A*, A' and C unify with mgu θ.

For each C_i in C do
For C_i a constraint on term f_i in *A*, $C_i = X_i : \phi$,
If $f_i \models \phi$, unify X_i with f' in MGSats(ϕ).
(Exit if no such goal and clause exist).

Add the constraints in C to $CStore$.
Continue if $CStore$ is not false, and exit otherwise.

Remove *A* from and add B_1,B_2,. . ., and B_n to the resolvent.

Apply θ to the resolvent, to *G*, and to *CStore*.

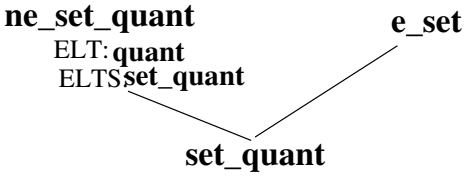
Figure 5.3: SLD Resolution over Feature Structures with Descriptions as Constraints. Terms are feature structure terms which must be unifiable with the descriptions that constrain them in the constraint store. This algorithm has nondeterministic selection of goals.

- elements selected via the MOD feature.

(Pollard & Yoo 1997:15)

A specific instance of this constraint is that for verbs with thematic arguments, the quantifiers will be pooled from the subject and object. This POOL can be built explicitly in each verb's lexical entry. Alternatively, it can be stated as a constraint across most verbs, noting that verbs with non-thematic arguments such as raising verbs (e.g. *appear* in the working grammar) and modal verbs are handled separately.[1] The description of the constraint on the POOL value is, "for each argument on my argument list, make sure it has its QSTORE value, before the POOL is determined." The QSTORE value itself is a set of quantifiers. This set may be a non-empty set of index values, or it may be empty. We assume a type hierarchy for sets of quantifiers as follows:

(88)



A guarded description which describes the behavior we want from the QSTORE value is shown in 89.

(89) ELT:RESTIND:INDEX:**ref** $\rightarrow$ $true$; **e_set**

The description 89 says that a value contributed to my QSTORE needs to be of type **ref**, else my QSTORE value is an empty set. The description ELT:RESTIND:INDEX:**ref** in 89 is only part of the story, in that it is not just one element of the set that must have an index element of type **ref**, it is any remaining elements as well. The rest of the set of quantifiers, stored as the value of the feature ELTS, has the same description as 89.

90 is a bit of a grammar for the guarded description in 89. It appears in a type constraint on certain verbs. `loc:qstore` is the path value for the QSTORE, which is an attribute of the bundle of syntactic and semantic features having type **synsem**. `then_else` is a keyword that takes the place of "then true, else..."[2]

[1] In a hierarchy of lexical types such as in Riehemann (1993), the constraint can be stated on a type at the point of inheritance by all such verbs. This is the strategy which has been used in the following example.

[2] The keyword syntax for guards using ALE is described fully in chapter 6. We will introduce the keyword `then_else_set` to mean that the description is meant to recurse through a set.

Either each set member must satisfy a description of a referential element, or the whole is an empty set. Waiting may indeed occur until the verb obtains one of its QSTORE values from its subject. Only after a verb's subject has correctly been identified during parsing, and a determination made of whether it has a quantifier, will the verb's QSTORE set be complete.

(90) `loc:qstore:(elt:restind:index:ref then_else e_set)`

In the constraint in 87, for example, there is another instance of guarding. It is necessary to wait on the SUBCAT values of a verb to be instantiated before the QSTORE values of those arguments can be checked. I have implemented this and other constraints in HPSG theory using *guarded rules*. In the quantifier grammar, the guarded description in 90 appears nested inside a guarded rule. In other instances as well, much use is made of guarded rules. These are described beginning with the following section. First, one needs a way to express constraints negatively, so that the descriptions in the constraints can be either guarded descriptions or regular descriptions.

Negation can be expressed as a guarded description. We can write "not ϕ" as a guarded constraint without using a negation symbol. *Fail* and *true* are assumed to be unification with the types $\bot$ and $\top$, respectively.

Definition 5.11 (Negation of Descriptions) $\neg(\phi) = (\phi \rightarrow fail;\ true)$

It follows from definition 5.11 that a negative description is equivalent to failing when the positive description is satisfied. A negative description can be used in implementation as a shorthand for succeeding on dissatisfaction conditions. The algorithm for mutual satisfaction between feature structures is shown in the next chapter.

We can use a negative description as guarded constraint. In an implementation, we need to know when a feature structure F has enough information to either satisfy or not satisfy ϕ. The constrained feature structure $F : (\phi \rightarrow fail;\ true)$ is true when F satisfies this description (example 91).

(91) $F \models (\phi \rightarrow fail;\ true)$

5.3 Using Guarded Rules

5.3.1 Goal Selection

With a way to express negation, we can now use guarded rules to describe guarding over feature structures, without using guarded descriptions in the description language or in the constraint language. The allowable constraint in the guard is a feature structure constrained by a regular description. We explicitly delay rules until certain satisfaction conditions are met. A constrained feature structure $F : (\phi \rightarrow \psi; \upsilon)$ can be written alternatively with guarded rules as follows:

Definition 5.12 (Rewriting Rules for Guarded Descriptions) *A rule*
$\{X_i : (\phi \rightarrow \psi; \upsilon)\} \mid Clause$ *can be written with regular descriptions as follows:*
$\{X_i : \phi, X_i : \psi\} \mid Clause$
or
$\{X_i : \phi\} \mid fail$, $\{X_i : \upsilon\} \mid Clause$

The algorithm in figure 5.3 does not describe an ordering on goals or clauses. In Prolog, goals are stored in a list structure, and the first goal on the list is chosen. In chapter 3 we described an algorithm for guarding which prefers goals which have tried but whose constrants are found to be unsatisfiable (figure 3.7). We call these suspended goals. As for the list of suspended goals, any of these is triggered immediately once its guard is satisfiable. The objective of the program is to provide a set of rules which will awaken goals.

Definition 5.13 (Suspended Goal) *Given goal $p(f_1, \ldots, f_n)$ and a (renamed) clause $C \mid A' \leftarrow B_1, B_2, \ldots, B_n$, such that goal p, A', C unify with mgu θ, and C_i a constraint on f_i, $C_i = X_i : \phi \in C$:*

- *If f_i neither satisfies nor dissatisfies ϕ, a* suspension *of a goal $p(f_1, \ldots, f_n)$is created. A suspension is a pointer to the conjunction of goals which are delayed by the constraint. If ϕ is already constrained to satisfy any $p'(f_1, \ldots, f_n)$, then p is added to the conjunction of goals.*
- *p is wakened if X_i is unified with a feature structure that satisfies ϕ. Then, the action taken is to first execute the suspension of p, and then resume the present goal.*

The definition 5.13 always tries to expand the newest unresolved goal, and first tries to resolve any suspended goals before unsuspended ones. Prolog makes use of a stack as well as a list. Using a queue rather than a stack would result in breadth-first evaluation of goals, rather than depth-first. Note that clauses need to be ordered in the program as well as the set of goals remaining to be solved, in order to provide a precise order of evaluation.

5.3.2 Example: Guarding in the Lexicon

It is with definition 5.12 that we have implemented guarding for German. We look at guarded rules in the case of argument raising by auxiliary. During processing, we wish to "check off" the noun arguments we encounter from the subcategorization list of the head verb. In this case the head is an auxiliary whose subcategorization list is not fully instantiated. We must find the verb that is the semantic head, unify the head verb's semantics with the semantics of the auxiliary, and then go back and fill in the missing pieces for the auxiliary. Then we can check off the same nouns from the auxiliary list, and move on to a completed sentence. The sentence will gain its information ultimately from the auxiliary, since it is the head of the sentence.

The example of German partial verb phrase fronting shows how extraction from the auxiliary is possible in the lexicon, even though the auxiliary's arguments are not yet realized. In this implementation I use a *guarded lexical rule*. The application of the lexical rule for complement extraction from a verb passes along the guards on the extracted complements, and the goals associated with the rule are suspended. These goals are constrained by the verb arguments in SLASH. These wait until the verb arguments are realized during parsing. This prevents the user from having to put an artificial constraint on the length or form of the complements list of the head verb. Consequently, one does not overgenerate numerous lexical entries with various combinations of complements in SLASH at the time of lexical compilation, which is the former solution available without guarding. The guarded rule is shown as example 92 (with the arguments of the *append* relation simplified), and 93 is its implemented form.

`If` is the keyword in ALE for goals associated with a lexical rule.[3] `If_guard` is a version of `If` that signals that in addition to the goals, a guard will be given first. `Prop guard nucleus:relation` is a feature structure constraint, with `Prop` the variable for a feature structure, and NUCLEUS:RELATION the de-

[3] `If` is used elsewhere also to signal a list of goals.

```
pvp lex_rule
     (word,
      synsem:(loc:(cat:(head:(Head,
                              verb,vform:bse,aux:plus,flip:minus),
                        subj:[Subj],
                        comps:PVPComps),
                   cont:Cont),
              non_loc:(inherited:(slash:e_set)))),

     ***>
     (word,
      subcat:[Subj|SubcatComps],
      synsem:(HeadSyn,
      (loc:(cat:(head:Head,
                         subj:[Subj],
                         comps:PVPComps),
                  cont:(Cont,nucleus:modal_arg:Prop)),
               non_loc:(inherited:
                        (slash:(elt:(PVP,
                                     cat:(PVPHead,
                                     head:(verb,vform:bse),
                                               lex:minus,
                                               subj:[Subj],
                                               comps:PVPComps),
                                               cont:Prop),
                                elts:e_set)))))),

if_guard
  Prop guard nucleus:relation then_else_fail

(append(PVPComps,
                [(loc:PVP,
                  non_loc:(inherited:(slash:(elt:PVP))))],
                SubcatComps),
        aux_raising(HeadSyn,PVPComps,Subj))
```

Figure 5.4: Partial verb phrase fronting as guarded lexical rule. The guard statement says that once the relation type of the verb (the modal argument) is known, then the value of the SUBCAT list of the derived verb is the concatenation of the complements list of the Partial Verb Phrase, plus the verb phrase itself as the last element of the list. Once the arguments are known, check the constraint on argument raising.

scription that `Prop` must satisfy. The keyword `then_else_fail` stands for "then true, else fail." It gives the constraint which should be added to the constraint store with satisfaction of the guard. In this case the alternative to satisfaction of the description by the feature structure `Prop` is not a different description but simply failure. In other words, if $Prop \not\models nucleus : relation$, then fail.

(92) $Prop \models nucleus : relation \mid lex\_rule(word(\ldots), word(\ldots, Prop, \ldots)) \leftarrow$
$append(PVPComps, PVP, SubcatComps)$,
$aux\_raising(HeadSyn, PVPComps, Subj)$.

(93)
```
if_guard
   Prop guard nucleus:relation then_else_fail

(append(PVPComps,
                [(loc:PVP,
                  non_loc:(inherited:(slash:(elt:PVP))))],
                SubcatComps),
        aux_raising(HeadSyn,PVPComps,Subj))
```

The guard in the lexical rule is a description in the description language, which the feature structure extracted to SLASH must satisfy. The description is `nucleus:relation`. This means that the semantic relation of the verb has to be determined before the arguments can be raised. The complete rule as it appears in the grammar is given in figure 5.4.

In addition, raising of all verb arguments by a head auxiliary is achieved by a *guarded lexical entry*. Because it is impossible to specify all of the arguments of an auxiliary verb in the lexicon, the list is simply left underspecified. Constraints on the semantics of the auxiliary and the verb are executed after the verb is found. It is especially important to wait since the auxiliary is the right-most complement (least oblique) in the list rather than the left-most. The head of a list may be selected with a variable, but not the last element of the list. The lexical entry with a guard on auxiliary raising is shown as figure 5.5. The key word `if_guard_all` signals a "for all" version of guarding for lists that waits with the same constraint on each member of a list argument. This constraint is that each member of the list of complements must be instantiated as a substantive, or type **subst**. The subtype of its head feature will be either **noun**, **verb**, etc.

Instead of `then_else_fail` the next keyword used here is `then_else`. This points to the alternative description that the variable `Comps` satisfies if it is

```
werden --->
   word,
   synsem:(Synsem,loc:(cat:(head:(verb,
                                  mod:none,
                                  vform:bse,
                                  aux:plus),
                            subj:[(NP,@ np(_)) ],
                            comps:Comps,
                            spr:[],
                            marking:unmarked,
                            lex:plus),
                       cont:(nucleus:future,
                             quants:[]),
                       conx:backgr:e_set)),
  (@ empty_non_loc),
  qstore:e_set

if_guard_all
Comps guard loc:cat:head:subst then_else e_list
 (aux_raising(Synsem,Comps,NP)).
```

Figure 5.5: Lexical entry for auxiliary with guarding. A guard is placed on the on the head value of each member of the COMPS list. Once all arguments satisfy the type **subst**, then apply the auxiliary raising constraint (`aux_raising`).

not a list of substantives. That is an empty list, or **e_list**. (This is the description υ in the guarded constraint $F : (\phi \rightarrow \psi; \upsilon)$.)

The guards on both the lexical rule and the lexical entry successfully wait together to yield unique parses for fronted PVP constructions such as: `gehen wird Sandy`, `Sandy sehen wird Kim`, `Sehen wird Kim Sandy`, etc. as well as double infinitive constructions `Sandy wird Kim sehen können`, `Sehen können wird Kim Sandy`, etc. and non-fronted constructions `Wird Kim Sandy sehen können?`. The implementation is true to the analysis in Baker (1999).

5.3.3 Inequations

Inequations are a specific instance of guarding, where the formula being negated is a path equation. Where Π is a path value in a feature structure, example 94 is the representation for inequations.

(94) $\Pi_1 \not\doteq \Pi_2$

Using definition 5.11 this can be written as the negation of the positive path value:

Definition 5.14 (Inequations) $\Pi_1 \not\doteq \Pi_2 = \neg(\Pi_1 \doteq \Pi_2)$

This is the same notion of negation used by Moshier (Moshier & Rounds 1987),(Moshier 1988). Definition 5.14 also gives the same results for negation of inequations as the ALE system (Carpenter & Penn 1994). This fact will be exploited in the implementation described in chapter 6. We introduce an implementation of the binding theory in this chapter.

Inequations can be used in an implementation of the binding theory. We review the principles of the binding theory. These are cast in the framework of HPSG; the relevant comparisons between this version and a Government/Binding account are given in Pollard & Sag (1994:chapter 6).

(95) HPSG Binding Theory:

Principle A. A locally o-commanded anaphor must be locally o-bound.
Principle B. A personal pronoun must be locally o-free.
Principle C. A nonpronoun must be o-free.

(Pollard & Sag 1994: chapter 6, 40)

O-command is defined as a relation on members of SUBCAT lists:

(96) HSPG o-command:

One referential **synsem** object *o-commands* another provided they have distinct LOCAL values and either

1. the second is more oblique (to the right on the SUBCAT list) than the first,
2. the second is a member of the SUBCAT list of a **synsem** object that is o-commanded by the first, or
3. the second has the same LOCAL:CATEGORY:HEAD value as a **synsem** object that is o-commanded by the first.(Pollard & Sag 1994)

Co-indexing is defined for objects of type **synsem** as having token-identical values for the path value LOCAL:CONTENT:INDEX. A **synsem** object is referential if its LOCAL:CONTENT:INDEX value is of type **ref**. In order to verify principles B and C for feature structures, one must verify that there is no structure sharing between the LOCAL:CONTENT:INDEX value of the personal pronoun (Principle B) or the nonpronoun (Principle C) and the same value for a referential **synsem** object which o-commands it. In order to do this, two path values must be compared and found to be non-identical. We define inequality for path values in the following section, and then move on to an example.

In our example we assume following Manning (1994) and Manning *et al.* (1999) that the lexical entry for a verb specifies a list of arguments in order of obliqueness for binding, and that the arguments on this list are token-identical with, but possibly differently ordered than, the concatenation of the arguments on the valence lists of the verb (e.g. SUBJ, OBJ). This list has been named e.g. as the SUBCAT list of Pollard & Sag (1994:chapter 9) and the ARG-S of Manning *et al.* (1999) and others.

(97)
$$\begin{bmatrix} \textbf{like} \\ \text{HEAD:} \begin{bmatrix} \textbf{verb} \\ \text{VFORM:}\textbf{bse} \end{bmatrix} \\ \text{SUBJ:}\langle \boxed{1} \rangle \\ \text{COMPS:}\langle \boxed{2} \rangle \\ \text{ARG-S:}\langle \boxed{1}, \boxed{2} \rangle \end{bmatrix}$$

We wish to obtain the following grammaticality judgments for the following examples using the single lexical entry in example 97. The subscripts *i,j,k,* etc. are used to mark reference to a unique person or object:

(98) a. John$_i$ likes himself$_i$.

b. John$_i$ likes him$_j$.

c. *John$_i$ likes him$_i$.

We can think of the binding theory as a set of constraints on the sign for *John likes himself*, in which the arguments of *like* are fully instantiated, or on the lexical entry for *likes*, in which the arguments are not. If in fact the ARG-S list is a lexical feature, as is argued in Pollard & Sag (1994) and Manning & Sag (1998), the site of binding will not be propagated to the phrasal head of the sign for *John likes him/self*. In this case the binding constraints must be specified in the lexicon.

But anaphoric, nominal and pronominal INDEX values are underspecified in the lexicon. Hence the need for delayed evaluation during processing.

For example 98a, the subject *John* and the reflexive object *himself* must be co-indexed. This means that feature structures tagged with [1] and [2] in 97, for the subject and object, respectively, will share a value for the path LOCAL:CONTENT:INDEX (tagged in examples 99 and 100 with [3]). 99 shows the LOCAL values for the subject (these are the syntactic and semantic features of concern to us here) and 100 shows the shared index value as part of the LOCAL value for the reflexive object.

(99)
$$\begin{bmatrix} \textbf{local} \\ \text{CAT:} \begin{bmatrix} \textbf{category} \\ \text{HEAD:}\textbf{noun} \\ \text{COMPS:}[] \\ \text{SUBJ:}[] \\ \text{SPR:}[] \end{bmatrix} \\ \text{CONTENT:} \begin{bmatrix} \textbf{npro} \\ \text{INDEX:}\boxed{3} \begin{bmatrix} \textbf{ref} \\ \text{GENDER:}\textbf{masculine} \\ \text{PERSON:}\textbf{third} \\ \text{NUMBER:}\textbf{singular} \end{bmatrix} \\ \text{RESTRICTION:}\textbf{e\_set} \end{bmatrix} \\ \text{CONX:} \begin{bmatrix} \textbf{context} \\ \text{BACKGR:} \begin{bmatrix} \textbf{background} \\ \text{ELT:} \begin{bmatrix} \textbf{psoa} \\ \text{NUCLEUS:} \begin{bmatrix} \textbf{naming} \\ \text{BEARER:}\boxed{3} \\ \text{NAME:}john \end{bmatrix} \\ \text{QUANTS:}[] \end{bmatrix} \\ \text{ELTS:}\textbf{e\_set} \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

(100)
$$\begin{bmatrix} \textbf{local} \\ \text{CONTENT:} \begin{bmatrix} \textbf{refl} \\ \text{INDEX:}\boxed{3} \\ \text{RESTRICTION:}\textbf{e\_set} \end{bmatrix} \end{bmatrix}$$

For examples 98b and 98c, the direct object is a personal pronoun. Since the subject *John* o-commands the object *him*, then they must not be co-indexed. If

they are co-indexed as in 98c, the sentence is ungrammatical. The inequation between subject and object which is in effect in 98b is shown in 101.

(101) $\boxed{1} \neq \boxed{2}$

The semantic index for both the noun and the pronoun in 98b is this:

(102)
```
index:(ref,Ind,
              gender:masculine
              person:third
              number:singular)
```

There is no description available to distinguish the INDEX features of the noun *John* from those of the pronoun *him*. That is, in the feature structures above, there is no description that the index value of *him* could satisfy in order to show that it was not the same exact feature structure as $\boxed{3}$. This means that satisfaction alone is not enough to prove or disprove an inequation. Different approaches to using guarding for solving inequations are discussed in the following chapter. It is there that we will discuss the notions of intensional and extensional identity.

5.4 Summary

In order to process the many underspecified linguistic theories in chapter 2 I have defined guarded constraints on feature structures. I use the description language in Carpenter (1992). Guarding may be done either with guarded rules or by adding *guarded descriptions* to the language. The latter approach enables rules to be delayed by subsumption conditions and without altering the algorithm for resolution introduced in chapter 3. The example of German partial verb phrase fronting describes a method for compiling lexical rules with delay statements attached, such that the lexicon may still be closed under lexical rules at compile time. This is a departure from the methods of Meurers & Minnen (1995), who wait until run time to apply such lexical rules. Inequations are described as instances of guarding on satisfaction conditions. In the next chapter, more detail is provided about the implementation of guarded constraints, and evaluation metrics for the system are discussed.

Chapter 6

Implementation and Evaluation

The ALE system (Carpenter & Penn 1994) combines phrase structure parsing and constraint logic programming with typed feature structures as terms. The addition of guarding to this system allows a number of linguistic analyses to be processed more directly using guarded descriptions. First I describe a general approach to the implementation of guarding for the case of typed feature structures. Then, I present specific test cases. These are quantifier raising in English and binding in Japanese. The quantifier raising example associates delays with a general type, the type **sign**. This demonstrates that delays may be associated with virtually any or all objects in the domain, by virtue of their types. A trace of the parse of this example is provided. I then move on to a specific instance of binding, which has been introduced in chapter 5. Linguistic binding conditions are stated in part negatively. The Japanese example shows the handling of inequations as delays in detail.

The set of examples, including the example of German partial verb phrase fronting introduced in the previous chapter, can be evaluated with respect to three criteria. First, are these grammars a faithful rendering of the specification in chapter 5? Second, is the satisfaction algorithm independent of the parser? Third, what are the advantages of using this approach compared with an implementation without guarding?

6.1 Implementing Delays

We use the satisfaction of a description by a feature structure to signal when constraints are fired. If a feature structure entails a description, then the constraint

associated with the description is fired. If the description is inconsistent with the feature structure, we either fire an alternative constraint, or fail. We have a question, however, to answer: What do we actually do during processing if we have neither entailment nor disentailment? In this case we need to wait and check back to see if the main feature structure, which we will call F, gains more information. A second question is, how do we define "enough information"? We may only be interested in a particular piece of the puzzle, for example, the subcategorization list of a head.

In implementing delays, we put a delay on the type of the information that we are looking for. When the type is specific enough, then we have our answer. If we don't get an answer, then our goal can't apply. That is because we never delay any goals that could have been solved with a less specific type. This is important if we are to find a solution.

The work is implemented in SICStus Prolog (SICStus 1995). The data structure *Tag-SVs* is the Prolog term for our feature structure. The structure is fully described in Carpenter & Penn (1996). The tag is a variable which acts as pointer, and gets instantiated to a new value each time the feature structure is updated. The `SVs` value is an array of sort values for the feature structure, which may themselves be feature structures. When new information is added to the whole, it is added as an instantiation of the tag at the front of the structure. By fully dereferencing the feature structure, the chains are compacted and a new variable tag becomes the new pointer and the site of further guarding.

The main reason for guarding during processing is that a feature structure F may not be fully instantiated. The type may be $\perp$ or it may have a non-maximal type. If the type of the feature structure variable is not specific enough to determine whether F satisfies the type T, a wait placed on the tag of the feature structure in case it gains a type more specific which satisfies T. The reason to place a wait on the tag of the feature structure is that the tag changes if the feature structure gains a more specific type. I have defined the `satisfies` relation over feature structures and descriptions. Guarding in `satisfies` occurs when asking whether a feature structure F satisfies a type T. The code for `satisfies` is shown in Appendix A.

In asking whether a feature structure satisfies a feature-value pair *Feat:Desc*, a wait is also placed on the tag of the feature structure if it does not already satisfy a type for which *Feat* is appropriate. Recursively, `satisfies` is called on the feature structure at *Feat*, with respect to *Desc*. This is a simply a specific instance of waiting for a type to change.

In the code we use, descriptions are used to stand for feature structure terms,

and the descriptions are compiled out to feature structures at runtime. This means that as we ask whether a feature structure satisfies a particular description, that in practice we are checking whether one feature structure subsumes another. A specific adaptation of guarding for feature structures involves checking whether two feature structures are *isomorphic*. Morphisms have been introduced in definition 4.4, repeated below as 6.1 for convenience.

Definition 6.1 (Subsumption) $F = \langle Q, \bar{q}, \theta, \delta \rangle \sqsubseteq F' = \langle Q', \bar{q}', \theta', \delta' \rangle$ *if and only if there is a total function* $h : Q \rightarrow Q'$*, called a* morphism *such that:*

- $h(\bar{q}) = \bar{q}'$
- $\theta(q) \sqsubseteq \theta'(h(q))$ *for every* $q \in Q$
- $h(\delta(f, q)) = \delta'(f, h(q))$ *for every* $q \in Q$ *and feature* f *such that* $\delta(f, q)$ *is defined*

(Carpenter 1992: Definition 3.4)

An isomorphism occurs if there is a mapping from the nodes and arcs of one feature structure to those of the other, and vice versa.

Definition 6.2 (Isomorphism) *Two features* F *and* F' *are isomorphic iff* $F \sqsubseteq F'$ *and* $F' \sqsubseteq F$.

By definition 6.2 it is possible for two feature structures not to have the same value for a feature and still be isomorphic. For example, consider the case of the two feature structures in 103, which have a different value for the feature F, and yet still subsume each other:

(103) a. $\begin{bmatrix} \sigma \\ \text{F:}\boxed{1} \end{bmatrix}$

b. $\begin{bmatrix} \sigma \\ \text{F:}\boxed{2} \begin{bmatrix} \sigma \\ \text{F:}\boxed{1} \end{bmatrix} \end{bmatrix}$

These two feature structures could both be collapsed to the simpler form in 103a using a process called extensional unification (see Carpenter 1992, chapter 8). We adopt a stricter notion of isomorphism for the purposes of implementation. We determine not the isomorphism in 6.2, but rather whether two feature

structures are *extensionally identical*. This means that they can be unified to produce a common extensional feature structure. The code refers to the procedure as `iso`, but technically speaking, `iso` will not succeed on the case of determining whether the cyclic feature structures in 103 are isomorphic.

Theorem 6.3 (Extensionality) *Let $\mathcal{TF}$ be the collection of well-typed feature structures and let $\mathcal{ETF}$ be the collection of extensional well-typed feature structures. There is a partial function $Ext : \mathcal{TF} \rightarrow \mathcal{ETF}$ such that if $F \in \mathcal{TF}$ and $F' \in \mathcal{ETF}$ are such that $F \sqsubseteq F'$, then $Ext(F) \sqsubseteq F'$.*

Proof: To carry out extensionalization, we iterate the following step: select a pair of nodes assigned a common extensional supertype and sharing all appropriate features and identify them (unify the feature structures rooted at these nodes). (Carpenter 1992: Theorem 8.3)

We introduce the procedure used in the code for finding an isomorphism. For the case where $Desc$ is compiled to a feature structure F', determine recursively from the top down whether the two feature structures F and F' are isomorphic. Assume that for F and F', the respective data structures are `Tag1-SVs1` and `Tag2-SVs2`. Then, follow this procedure:

1. If the tags `Tag1` and `Tag2` are identical, stop. This means they are the same object, or *intensionally* identical.
2. If the tags `Tag1` and `Tag2` of two feature structures `Tag1-SVs1` and `Tag2-SVs2` can be unified, do unify them and recursively determine whether the arrays of feature values `SVs1` and `SVs2` are isomorphic. Do this by determining whether, for each feature $s1_1, s1_2, \ldots, s1_n$ in `SVs1`, the feature structures rooted at $s1_1, s1_2, \ldots, s1_n$ are isomorphic to the the feature structures rooted at $s2_1, s2_2, \ldots, s2_n$ in `SVs2`.
3. If the types of two feature structures are not incompatible, then wait until the type of one if the feature structures becomes more specific, and retry the isomorphism, starting from step 1.

Currently the isomorphism is retried every time one of the two tags changes. The program is not smart enough to wait until the tags of the feature structures are unifiable, because determining this in itself would involve awakening and re-freezing of a variable. The pseudo-code for the procedure for processing an isomorphism with delaying is shown in example 104. If a goal is called, the procedure has a return value of 'true', or if the isomorphism fails, it has a return

value of 'fail.' Else, since there is information that is still being waited on, the return value is undefined. Assume a guarded description $\phi \rightarrow \psi; \upsilon$. F_1 is a feature structure and F_2 is a feature structure which satisfies ϕ.

(104) iso(F_1,F_2,ψ,υ) iff
if $F_1 = F_2$, add ψ
else
if Type(F_1) $\sqcup$ Type(F_2) = $\perp$, add υ
else
if Type(F_1) = Type(F_2) then
iso_values(Feats(F_1),Feats(F_2),ψ,υ)
else
delay(F_1,Type(F_2),iso(F_1,F_2,ψ,υ),υ)
else
delay(F_2,Type(F_1),iso(F_1,F_2,ψ,υ),υ)

iso_values([F_{11},F_{12},. . .,F_{1n}],[F_{21},F_{22},. . .,F_{2n}],ψ,υ) if
iso(F_{11},F_{21},iso_values([F_{12},. . .,F_{1n}],[F_{22},. . .,F_{2n}],ψ,υ),υ)

6.2 ALE syntax

The implementation is an extended version of ALE. Here we show the BNF grammar for definite clause rules, type constraints, and guarded rules and constraints. <pred_sym> and <prolog_goal> are predicate symbols and goals from Prolog. <type> is a type from the inheritance hierarchy. The type hierarchy specifies the types and appropriateness conditions for feature structures, and is declared as part of an ALE program.

(105) Literals.

```
<literal> ::= <pred_sym>
            | <pred_sym>(<desc_seq>)
```

(106) Goals.

```
<goal> ::= true
         | <literal>
         | (<goal>,<goal>)
```

```
          | (<goal>;<goal>)
          | (<desc> =@ <desc>)
          | !
          | (\+ <goal>)
          | prolog(<prolog_goal>)
```

(107) Descriptions.

```
<desc_seq> ::= <desc>
             | <desc>,<desc_seq>

<desc> ::= <type>
         | <variable>
         | (<feature>:<desc>)
         | (<desc>,<desc>)
         | (<desc>;<desc>)
         | (=\= <desc>)
         | (<path> == <path>)
         | <guarded_desc>

<guarded_desc> ::= <desc> then_else_fail
                 | <desc> then_else <desc>
                 | <desc> then_else_set <desc>
```

(108) Guards.

```
<guard> ::=  <variable> guard <desc>
```

`then_else_fail` signals that if the `<variable>` does not satisfy the guard, then fail. `then_else` signals that if the `<variable>` does not satisfy the first description, then it must satisfy the other (second) description. `then_else_set` applies to sets.[1]

The guarded feature structure `<variable>` could actually be any description, but in practice, it is always a variable that stands for a feature structure that the grammar writer picks out. It is unified with a relevant description in a lexical rule, grammar rule or type constraint, by taking advantage of Prolog unification on variables.

[1] `then_else_set` means every member of the set must satisfy the description, or the description must be an empty set (**e_set**). This is parochial to this implementation.

(109) Type constraints, including guarded constraints.

```
<cons_spec> ::=  | <type> cons <desc>
                    goal <goal>
                 | <type> cons <desc>
                    guarddesc <guard>
                    goal <goal>
```

(110) Clauses.

```
<clause> ::= <literal> if <goal>.
           | <literal> if_guard <guard> <goal>.
           | <literal> if_guard_all <guard> <goal>.
```

`if_guard_all` is a keyword that signals that the description must be satisfied recursively across the entire list or set, if the variable has the type of a list or set.

It is noted here that there is an inconsistency between the syntax of type constraints and clauses. The syntax for both could no doubt be standardized so that clauses would be written as follows:

```
<clause> ::= <literal> if <goal>.
           | <literal> if
              guarddesc <guard>
              goal  <goal>.
```

Then the notation of `if_guard_all` might be transferred to the `<guard>` syntax, by using there two keywords, e.g. `guard` and `guard_all`.

(111) Phrase Structure Rules.

```
<rule> ::= <rule_name> rule <desc> ===> <rule_body>.

<rule_body> ::= <rule_clause>
              | <rule_clause>, <rule_body>

<rule_clause> ::= cat> <desc>
                | cats> <desc>
                | goal> <goal>
```

6.3 Parsing with Delayed Constraints

6.3.1 Quantifier Raising with Delays

We have explained how guarding involves satisfaction between a feature structure and a description, which can be broken down as a wait on a specific type. Waiting on types is employed in the analysis of quantifier raising in Pollard & Yoo (1997), in which quantifier storage is a feature local to the type **sign**. A sign is the basic linguistic entity in HPSG and is comprised of words and phrases (these have subtypes **word** and **phrase**). The quantifier store, or QSTORE, of a sign is derived from a pool of quantifiers (the value of the feature POOL) and a list of any of those quantifiers which may have been retrieved for scoping, in the case of a verbal sign (the RETRIEVED value). Because of this, it is necessary to associate the constraint on the QSTORE value of a sign directly with the type **sign** in the grammar.

We will guard the constraint on the QSTORE value of signs. This means that we will establish a particular guard that must be satisfied before the goal which expresses the constraint can be solved. In our formalism, a guard is a variable plus a description. The guard is satisfied if it is instantiated to a feature structure that satisfies the description. In particular, we will guard the type of the POOL feature. The pool is a set. We say that each member of the pool (if it is non empty) must satisfy the type **ref** (it is referential). The guard says once we have information from all of the members of the POOL, compute the RETRIEVED and QSTORE values from that pool. Because words and phrases are signs, a guarded constraint is then attached to all phrases and all words as they are encountered during parsing. However, only if more information is needed will resolution actually need to wait on the satisfaction of the guard. "More information" might mean knowing what the subject is in the case of a verb phrase, for example, because the verb phrase inherits any quantifiers that might be associated with its subject. In the case of semantically vacuous lexical entries, such as the words *to* and *be* (as shown in Pollard & Yoo (1997)), the constraint will be immediately satisfied; that is, there is no waiting. This is because these don't have any POOL of quantifiers to pick from.

The original constraint on the type **sign** as described by Pollard and Yoo is in 112.

(112) Given a POOL value P of a sign, the set of elements in the RETRIEVED value will form a subset R of P, and the QSTORE will be the set of unretrieved values, that is the set difference between P and R.

The HPSG expression of the constraint is in 113. There is a technical issue here, in order to relate lists of quantifiers with sets of the same. The type of retrieved elements in the theory is **list_quant**, which is a list of quantifiers, the same as it is in HPSG theory. The relation $set\text{-}of\text{-}elements$ is a relation that holds between the list of quantifiers that has been retrieved, and a set of these same quantifiers. Think of it as a conversion from list to set format. This is simply a convention so that there can be a uniform set operation over the retrieved, pooled, and stored quantifiers, the POOL and QSTORE being sets in the theory.

(113) $sign \longrightarrow$

$$\begin{bmatrix} \text{SYNSEM:} \begin{bmatrix} \textbf{synsem} \\ \text{LOCAL:} \begin{bmatrix} \textbf{local} \\ \text{QSTORE:}\boxed{1} \\ \text{POOL:}\boxed{2} \end{bmatrix} \end{bmatrix} \\ \text{RETRIEVED:}\boxed{3} \end{bmatrix}$$

$\wedge set\text{-}of\text{-}elements(\boxed{3}, \boxed{4})$
$\wedge \boxed{4} \subset \boxed{2}$
$\wedge \boxed{1} = \boxed{2} - \boxed{4}$

The guarded type constraint from the implementation of the Pollard & Yoo analysis is shown in example 114. `guarddesc` signals that the guarded description is the sign's POOL value. The constraint says that for a sign with a QSTORE value of `QStore`, a POOL value of `Pool` and a list of retrieved quantifiers `Retrieved`, then wait on the index value of every element of the `Pool` before computing the QSTORE from the `Retrieved` and `Pool` values. The `set_sublist` clause is the goal. It calls the routine that computes the QSTORE. The value of `Pool` may alternatively be an empty set; the `then_else_set` syntax simply ensures that `Pool` will be properly considered a set and not a list.

(114)
```
sign cons (synsem:loc:(qstore:QStore,pool:Pool),
               retrieved:Retrieved)
    guarddesc Pool guard (restind:index:ref)
           then_else_set e_set
           goal
    (set_sublist(Retrieved,Pool,QStore)).
```

A simple constraint (115) on headed phrases ensures that the POOL value of the phrase is token-identical with the QSTORE value of the semantic head daughter. [2]

[2]For illustrational purposes, the unification shown is between the mother and the head daughter.

(115)
```
phrase cons (synsem:loc:pool:Pool,
       head_dtr:synsem:loc:qstore:Pool).
```

The semantics principle, a general principle which applies to all headed phrases, is implementable as a constraint on the type **headed_phrase**. There is a reason to apply guarding in the application of the principle to verb phrases. This is that the QUANTS value of such a phrase depends upon its RETRIEVED value. The list of retrieved quantifiers is found non-deterministically at any sign of sort **psoa**. These signs are verbs, verb phrases, and sentences in a standard grammar. We have seen that the value of of RETRIEVED depends upon satisfying the relation in the constraint in 113. And so once this constraint is satisfied, we may apply the semantics principle:

Semantics Principle (Pollard & Yoo (1997:examples 23 and 24)):

(116) a. For a headed phrase whose **cont** is of sort **psoa**, the NUCLEUS value is identical with that of the semantic head, and the QUANTS value is the concatenation of the RETRIEVED value and the semantic head's QUANTS value.

b. For a headed phrase whose CONT is not of sort **psoa**, the CONTENT value is token-identical to that of the semantic head.

The implementation of the semantics principle as a constraint on the type **verb_phrase** is shown in 117. The implementation has been simplified somewhat to apply to phrases headed by verbs, as these have heads whose content is of type **psoa**. For other headed phrases, the content value of the head and the phrase can be unified at the point of the application of the relevant phrase structure rule; it is clear that in this case, there is no need for a delay statement to apply.

(117)
```
verb_phrase cons (synsem:loc:(cont:(Cont,
                                    nucleus:Nucl,
                                    quants:MQuants)),
      retrieved:Retrieved,
              head_dtr:synsem:loc:(cont:(HeadCont,
                                     psoa,nucleus:Nucl,
                                     quants:HQuants)))

    guarddesc Retrieved guard (restind:index:ref)
```

In the case of a head-adjunct phrase, the unification of the mother's POOL value and the daughter's QSTORE would be between the mother and the adjunct daughter, since the adjunct is the semantic head in that case.

```
then_else e_list

else

     append(Retrieved,HQuants,MQuants).
```

The variable `Retrieved` stores the list of retrieved quantifiers. This is the feature structure that is being guarded. Once we know whether this is a list of quantifier values or an empty list, we can append this with the quantifier list of the head to determine the quantifier list of the mother. If the quantifier list of the head is not yet determined, the `append` call will still terminate. In that case, the tail of the mother's quantifier list will simply unify with the head's quantifier list, but will not vary. We show this in more detail in the following section.

Finally, the SUBCAT value on "content" verbs in the lexicon must be guarded.[3] The pool of quantifiers for a verb equals all of the quantifiers from its meaning-bearing complements (subject and objects in the standard case). As a test for guarded type constraints, I put all such verbs under a common type **verb_word** in the type hierarchy, and wrote the constraint as attaching to all verbs. Alternatively, one could attach it to each individual verb in the lexicon. This goes to show that there are different ways of achieving the same effect. Attaching this constraint in only one place is probably simpler.

(118)
```
verb_word cons (synsem:loc:(pool:Pool,
                            cat:head:
                                (verb,
                                 aux:minus,
                                 control:minus)),
                subcat:Subcat)

guarddesc Subcat
       guard
          (loc:qstore:(restind:index:ref then_else_set e_set))
          then_else e_list
  goal
  (qstores_of(Subcat,e_set,Pool)).
```

[3]These are regular verbs with non-vacuous semantics, e.g. *go, swim, run, take, find* etc. as opposed to auxiliaries, control verbs, and the like.

6.3.2 Parsing Steps

In this section we show how parsing steps are interleaved with guarding, in the case of the sentence in 119.

(119) A unicorn appears to be approaching.

We use phrase structure rules for parsing. There are 6 phrase structure rule schemata in HPSG. The reader is referred to these in Appendix B. Schema 2 is used for creating a verb phrase from a lexical head verb and its complement or complements. It has other applications also, such as promoting an N to an N′ or a Det to a determiner phrase (DetP). Schema 1 is used to put phrasal head together with one of its complements. This could be a verb phrase with its subject in the case of the rule $S \rightarrow NP\ VP$, or an N′ and its specifier, in the case of $NP \rightarrow DetP\ N'$. The rules in the trace appear labelled as `schema1` (for sentence), `schema2` (for verb phrase, N′, or DetP), and `head_spr` (the case of schema1 for NP).

The ALE rules for schema 1 (the head-subject schema) and the head-specifier rule are shown in figures 6.1 and 6.2.[4] The rules are compiled in this fashion: First, compile the description for the subject or specifier and match it to an edge in the chart; then, compile the description for the head daughter and match it; solve the goals; then, compile the mother and add it to the chart. Information about the mother (result) is gained during the course of the rule, but the description at the beginning of the rule is added into the feature structure procedurally at the end, after the goals are solved. The code which compiles rules into a series of definite clause statements is provided in Appendix C.

The parser is the bottom-up chart parser which is part of the ALE package. The parser enters edges for words into the chart from right to left, and then parses from left to right.[5] In table 6.1 I show for each edge which guarded constraints are unblocked by adding that edge to the chart, and when the QSTORE and RETRIEVED values are computed. Each time a word or phrase is entered into the chart, its POOL value is guarded, as part of the type constraint on the type **sign** in example 114. I will not mention this explicitly; it will be assumed.

[4]The semantics principle is not listed as one of the principles in the goal list since it has been implemented as a constraint on a type (example 117). Another methodology would be to implement these principles as constraints on the type **headed_phrase**. This grammar follows the HPSG grammar for English by Gerald Penn, released with ALE, in showing principles as definite clause attachments on phrase structure rules. Type constraints have been added to test guarding on types.

[5]Rules for phrases fire as soon as all the necessary words have been added to the chart, even if there are more words to be added to the left in the sentence.

```
schema1 rule
(Mother,verb_phrase,synsem:loc:cat:comps:e_list,head_dtr:HeadDtr)
===>
cat> (SubjDtr,phrase,synsem:(SubjSynsem,loc:cat:comps:e_list)),
cat> (HeadDtr,phrase,synsem:loc:cat:(head:(func;(prd:minus)),
                                comps:e_list,
     subj:[SubjSynsem])),
goal> (head_feature_principle(Mother,HeadDtr),
       inv_minus_principle(Mother),
       valence_principle(Mother,HeadDtr,[SubjSynsem],[],[]),
       marking_principle(Mother,HeadDtr),
       spec_principle(SubjDtr,HeadDtr),
       nonlocal_feature_principle(Mother,HeadDtr,[SubjDtr]),
       single_rel_constraint(Mother),
       clausal_rel_prohibition(Mother),
       relative_uniqueness_principle(Mother,[SubjDtr,HeadDtr]),
       conx_consistency_principle(Mother,[SubjDtr,HeadDtr]),
       deictic_cindices_principle(Mother,[SubjDtr,HeadDtr])).
```

Figure 6.1: Schema 1 in ALE format

Table 6.1: **Trace of Parse of quantifier retrieval, showing guards unblocked**

Parser Action	**Edge Added**	**Guards**
lexicon	approaching	
schema2	approaching	
lexicon	be	
schema2	be approaching	
lexicon	to	RETRIEVED value is empty list
schema2	to be approaching	
lexicon	appears	
schema2	appears to be approaching	
lexicon	unicorn	

continued on next page

Table 6.1 *bottom-up parse, continued*

Parser Action	Edge Added	Guards
schema2	unicorn	
lexicon	a	
schema2	a	
head_spr rule fires; daughters are compiled. Head and specifier not yet related.		
Principles fire; spec_principle fires. Determiner identified as specifier of N bar. RESTIND:INDEX value for unicorn is found (PER:THIRD,NUM:SING,GEN:NEUT)		POOL value for Detp a is unblocked.
		POOL value for N bar unicorn is unblocked.
		POOL value for N unicorn is unblocked.
		POOL value for det a is unblocked.
Rest of principles fire.		
Mother is compiled. N bar identified as head daughter and QSTORE values are unified.		POOL value for NP a unicorn is unblocked.
	a unicorn	
Rule schema1 fires; daughter descriptions are compiled and matched to existing edges. Subject of verb phrase daughter is unified with subject daughter (SubjDtr with synsem SubjSynsem). (Mother edge not yet compiled)		SUBCAT value for verb approaching is unblocked. (since subject found)

continued on next page

Table 6.1 *bottom-up parse, continued*

Parser Action	Edge Added	Guards
		POOL value for verb `approaching` is computed. POOL value on verb `approaching` is unblocked (sign constraint). POOL value for VP `approaching` is unblocked. (phrase's POOL is same as head daughter's QSTORE, by unification) RETRIEVED, QSTORE values for VP `approaching` is computed. (nondeterministically) RETRIEVED value for VP `approaching` is unblocked. POOL value for VP `be approaching` is unblocked. (QSTORE of `be` is known, same as QSTORE of complement `approaching`) RETRIEVED, QSTORE values for VP `be approaching` is computed. RETRIEVED value for VP `be approaching` is unblocked. POOL value for VP `to be approaching` is unblocked. (QSTORE of `to` is known, same as `qstore` of complement `be approaching`)

continued on next page

Table 6.1 *bottom-up parse, continued*

Parser Action	Edge Added	Guards
		RETRIEVED, QSTORE values for VP `to be approaching` is computed. RETRIEVED value for VP `to be approaching` is unblocked. POOL value for VP `appears to be be approaching` is unblocked. (based on QSTORE of head) RETRIEVED, QSTORE values for VP `appears to be approaching` is computed. RETRIEVED value for VP `appears to be approaching` is unblocked.
⟨principles associated with schema1 fire here⟩		
Description for mother compiled. Head daughter unified with VP edge `appears to be approaching`		POOL value for S `a unicorn appears to be be approaching` is unblocked. (based on QSTORE of head daughter) RETRIEVED, QSTORE values for S `a unicorn appears to be approaching` are computed. RETRIEVED value for S `a unicorn appears to be approaching` is unblocked.
	`a unicorn appears to be approaching`	

continued on next page

Table 6.1 *bottom-up parse, continued*

Parser Action	Edge Added	Guards

The implemented analysis of 119 returns the 6 possible parses described in Pollard & Yoo (1997). These show quantifier retrieval for the quantifier associated with *a unicorn* at each the following nodes in the parse trees in figure 6.3:

1. at the verb phrase VP_4 *approaching*
2. at the verb phrase VP_3 *be approaching*
3. at the verb phrase VP_2 *to be approaching*
4. at the verb phrase VP_1 *appears to be approaching*
5. at the sentence node
6. no retrieval; the quantifier is still in storage.

Retrieval at node VP_2, VP_3 or VP_4 corresponds with the narrow scope reading of the sentence (it appears that there is a unicorn approaching) $(appears(\exists(x)|uncorn(x) \wedge approach(x))$. One such tree is shown in figure 6.3. Retrieval at S and VP_1 correspond with the wide scope reading (there is a unicorn which appears to be approaching) $(\exists(x)|unicorn(x) \wedge appears(approach(x))$. This tree is figure 6.4.

The grammar has been implemented so that lexical entries have a RETRIEVED value of the empty list, so as to reduce spurious ambiguity, but retrieval at a lexical verb (*approaching* or *appears*) is possible in the theory (Pollard & Yoo 1997). In fact, Pollard & Yoo point out that retrieval is only possible at a lexical node in the related analysis of Manning & Sag (1998). We have discussed the ambiguities arising from this analysis in chapter 2. I add here that, while the implementation is true to the original analysis, adjustments to retrieval sites could certainly be made. For example, retrieval could be blocked on a phrase headed by a lexically vacuous particle such as *to* (VP_2).

6.3.3 Using Different Parsers

One significant advantage of guarded grammars is that they are blind to the particular parsing strategy being adopted. In test scenarios, two guarded HPSG grammars for German and English were each evaluated by two different parsers, a

```
head_spr rule
(Mother,phrase,synsem:loc:(cont:HeadCont,cat:comps:e_list),
               retrieved:e_list,
head_dtr:synsem:loc:(cont:HeadCont,qstore:HQStore))
%% Cannot structure share entire Head_dtr.  Causes a
%% cycle since the two FS are reciprocal.
===>
cat> (SprDtr,phrase,synsem:(SprSynsem,loc:cat:comps:e_list)),
cat> (HeadDtr,phrase,synsem:loc:(cat:comps:e_list,
                                 cont:HeadCont,
                                 qstore:HQStore)),
goal> (head_feature_principle(Mother,HeadDtr),
       inv_minus_principle(Mother),
       valence_principle(Mother,HeadDtr,[],[],[SprSynsem]),
       marking_principle(Mother,HeadDtr),
       spec_principle(SprDtr,HeadDtr),
       nonlocal_feature_principle(Mother,HeadDtr,[SprDtr]),
       single_rel_constraint(Mother),
       clausal_rel_prohibition(Mother),
       relative_uniqueness_principle(Mother,[SprDtr,HeadDtr]),
       conx_consistency_principle(Mother,[SprDtr,HeadDtr]),
       deictic_cindices_principle(Mother,[SprDtr,HeadDtr])).
```

Figure 6.2: ALE rule for a head plus its specifier.

bottom-up chart parser and a left corner parser, which uses a combination of top-down and bottom up search. The chart parser fills in edges in the chart from right to left and then goes back and parses from left to right. The left corner parser works strictly from left to right, asserting mother categories of rules from the top down once the leftmost daughter has been identified. However, this parser always works with respect to a list of top-down goals. The top-level code for the left-corner parser is shown in example 120.

(120)
```
% connect(+C1:<cat> ,+C2:<cat>, +Ws:<words>, ?WsOut:<words>)
% ------------------------------------------------------

% found C1 and need to complete C2 using difference Ws-WsOut
% ------------------------------------------------------
```

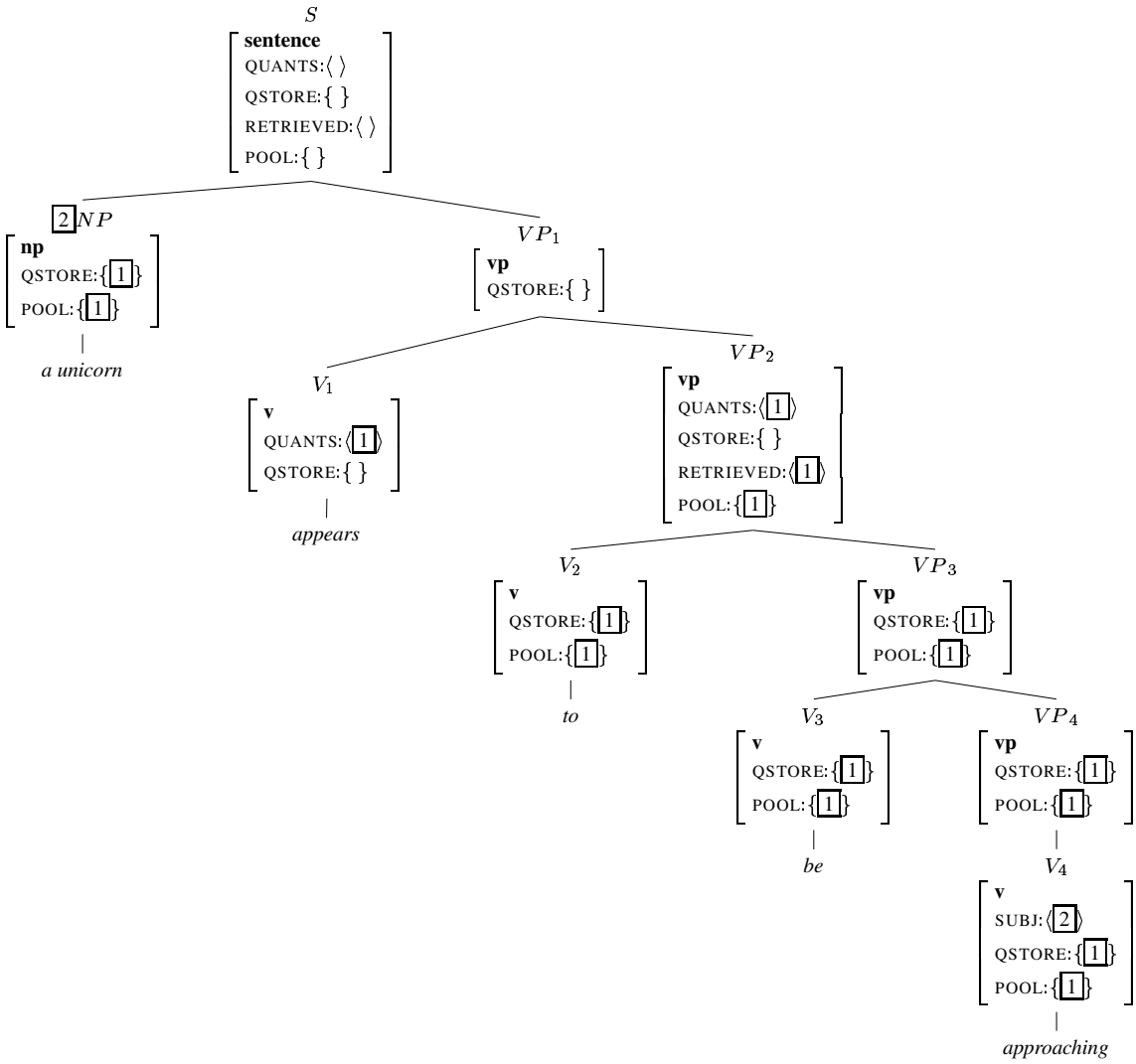


Figure 6.3: Derivation tree narrow scope reading, lexical quantifier retrieval (“de dicto” reading). (Pollard & Yoo 1997: example 25)

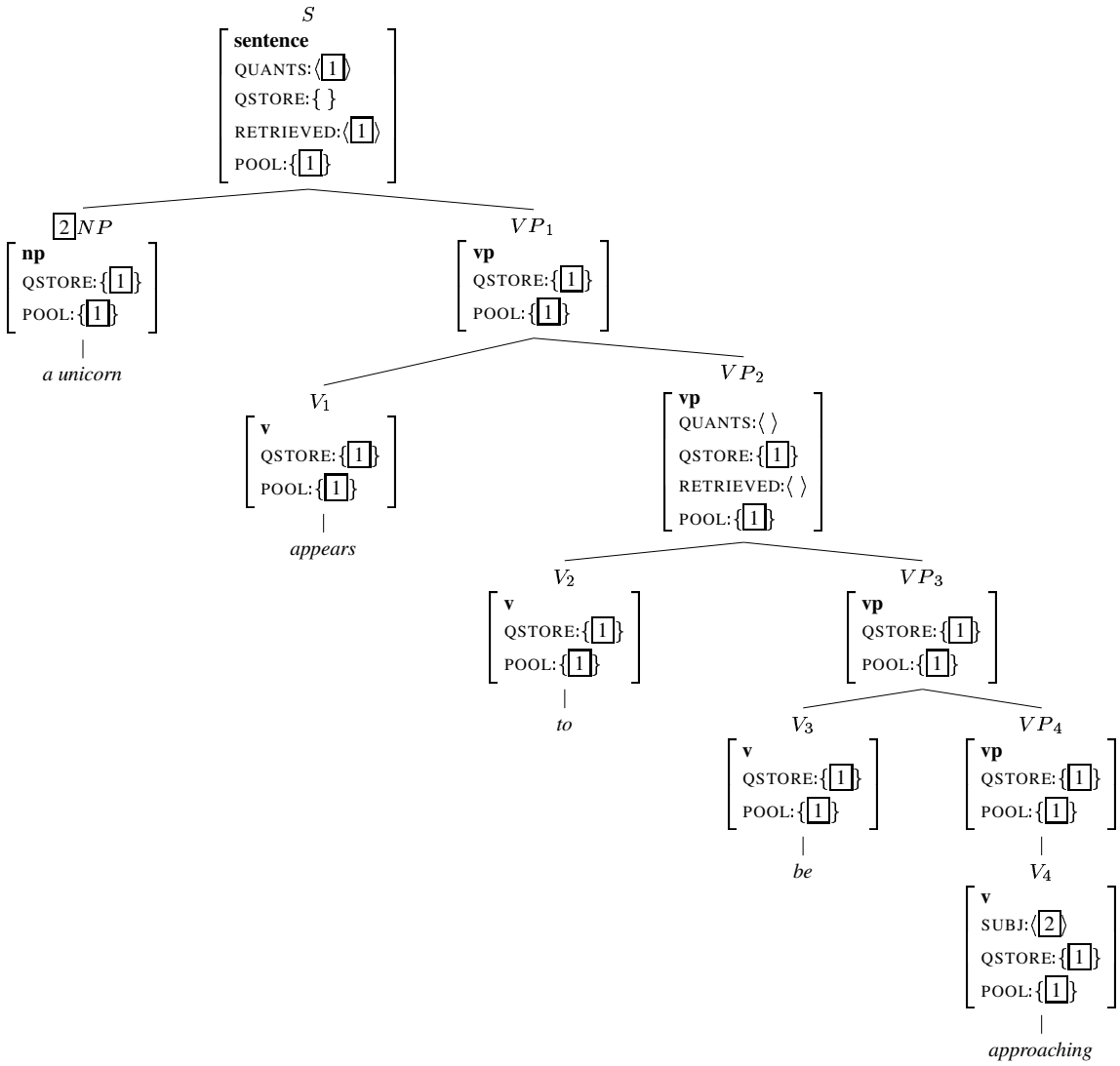


Figure 6.4: Derivation tree for wide scope reading, lexical quantifier retrieval ("de re" reading) (Pollard & Yoo 1997: example 26)

```
lc_rec(TagOut-SVsOut,[W|Ws],WsOut):-
  lex(W,Tag,SVs),
  connect(Tag-SVs,TagOut-SVsOut,Ws,WsOut).

connect(Tag-SVs,TagOut-FSOut,Ws1,Ws2):-
  deref(Tag,SVs,Tag2,SVs2),
  connect2(Tag2,SVs2,TagOut-FSOut,Ws1,Ws2).

connect2(Tag,SVs,Tag-SVs,Ws,Ws).

connect2(Tag,SVs,TagOut-SVsOut,Ws,WsOut) :-
  rule(Tag,SVs,Ws,WsMid,Tag2-SVs2),
  connect2(Tag2,SVs2,TagOut-SVsOut,WsMid,WsOut).
```

The 5-place predicate `rule` calls a phrase structure rule. More interesting is the way in which rules are *compiled*. We use the same phrase structure rules with both the chart parser and the left corner parser. First the leftmost daughter is compiled; then the mother is compiled; then a left corner search proceeds on the rest of the daughters. Therefore, the mother category in a rule is compiled from a description before the search is performed on the rest of the daughters. The code for the compilation of the rules is for each of the two parsers is provided in Appendix C. Search on the left-corners proceeds in a depth-first fashion. Recall the way in which rules are compiled the chart parser, such that the mother is the last of the feature structures in a rule (among mother, daughters) to be compiled. Guards are still attached in the left corner case once the feature structure has a type associated with it, even though much of the basic information is missing at first. So the guards are in fact established in a very different order, though they become unblocked in the same order as before.

We look at the example of quantifier raising in English to note the different orders in which the signs are guarded using the two different parses. In the case of the chart parser, edges are entered into the chart in the following bottom-up order for the example we have been following, *A unicorn appears to be approaching*:

1. VP_4 *approaching*
2. VP_3 *be approaching*
3. VP_2 *to be approaching*
4. VP_1 *appears to be approaching*

5. NP *a unicorn*
6. S node *a unicorn appears to be approaching*

Each time a verb phrase is entered into the chart, the value for its RETRIEVED feature is guarded, depending ultimately upon the outcome of its POOL. Its POOL in turn cannot be determined until the subject NP *a unicorn* is parsed and connected via subject-verb agreement to the VP_1 node. Each sub-VP then inherits its subject from its mother, and then can determine its POOL.

In the left corner scenario, phrasal signs are instantiated for the sentence constituents in this order instead:

1. NP *a unicorn*
2. S node *a unicorn appears to be approaching*
3. VP_1 *appears to be approaching*
4. VP_2 *to be approaching*
5. VP_3 *be approaching*
6. VP_4 *approaching*

The RETRIEVED values for the verb phrases are guarded in a reverse order from the chart parser, but the unblocking of the guards proceeds in exactly the same order. In the case of the chart parser, information proceeds from the head up to the mother, which is the last edge produced. In the left-corner scenario, the RETRIEVED value for the S node, although established early, must wait until the VP_4, the smallest VP, has been parsed. This completes the subcategorization list of each parent VP node up the tree. Once the subcategorization list for the sentence node is completed, then the POOL can be determined, and the guard on the RETRIEVED value is unblocked. Since a phrase inherits its POOL value from the QSTORE of its head in the theory, information percolates upward in both cases, regardless of the order in which the trees or edges are established.

The use of guarded rules did not cause differences in the set of answers returned by the parsers. In both cases, the same complete set of answers is obtained for auxiliary raising and quantifier scoping. It is noted that in the case of German, the use of a right corner parser may be preferable to parse verb phrase rules anchored on the right by a head verb.

We show the parsing order for the left corner parser in table 6.2.

Table 6.2: **Trace of quantifier retrieval, using left corner parser**

Parser Action	Rule Completed	Guards
lexicon	a	
schema2	a	
lexicon	unicorn	
schema2	unicorn	
head_spr rule fires; first daughter and mother are compiled.		
Parser called recursively on second daughter.		
Principles fire; spec_principle fires. Determiner identified as specifier of N bar. RESTIND:INDEX value for unicorn is found (PER:THIRD,NUM:SING,GEN:NEUT)		POOL value for Detp a is unblocked.
		POOL value for N bar unicorn is unblocked.
		POOL value for N unicorn is unblocked.
		POOL value for det a is unblocked.
Rest of principles fire.		
N bar identified as head daughter and QSTORE values are unified.		POOL value for NP a unicorn is unblocked.
	a unicorn	
Rule schema1 fires with first daughter a unicorn is connected. Mother is compiled. Left-corner parser is called on head daughter.		
lexicon	appears	
Rule schema2 fires		

continued on next page

Table 6.2 *left-corner parse, continued*

Parser Action	Rule Completed	Guards
lexicon	to	
Rule schema2 fires		
lexicon	be	
Rule schema2 fires		
lexicon	approaching	
schema2	approaching	
schema2	be approaching	
schema2	to be approaching	
schema2	appears to be approaching	
in schema1, subject daughter is unified with subject of head daughter. SUBCAT value for verb approaching is unblocked. (since subject found)		POOL value for verb approaching is computed.
		POOL value on verb approaching is unblocked.
		POOL value for VP approaching is unblocked, by unification
		RETRIEVED, QSTORE values for VP approaching is computed.
		RETRIEVED value for VP approaching is unblocked.
		POOL value for VP be approaching is unblocked.
		RETRIEVED, QSTORE values for VP be approaching is computed.

continued on next page

Table 6.2 *left-corner parse, continued*

Parser Action	Rule Completed	Guards
		RETRIEVED value for VP `be approaching` is unblocked. POOL value for VP `to be approaching` is unblocked. RETRIEVED, QSTORE values for VP `to be approaching` is computed. RETRIEVED value for VP `to be approaching` is unblocked. POOL value for VP `appears to be be approaching` is unblocked. RETRIEVED, QSTORE values for VP `appears to be approaching` is computed. RETRIEVED value for VP `appears to be approaching` is unblocked. POOL value for S `a unicorn appears to be be approaching` is unblocked. RETRIEVED, QSTORE values for S `a unicorn appears to be pproaching` are computed. RETRIEVED value for S `a unicorn appears to be approaching` is unblocked.

continued on next page

Table 6.2 *left-corner parse, continued*

Parser Action	**Rule Completed**	**Guards**
⟨principles associated with schema1 fire here⟩	`a unicorn appears to be approaching`	

Although a top-down parser has not been implemented, we can show in the same manner how the blocking and unblocking of contraints would proceed with a top-down search. The guards on the POOL value at the topmost S node would be established right away and would wait once again for information to percolate all the way up from the lexical entry for the verb *approach*, which would be one of the last tree nodes to be instantiated. The guards on the SUBCAT value for that verb would fire immediately because the noun *unicorn*, its subject argument, would already have been found. This is the reverse case from the bottom up parse in table 6.1, where the guards on *approach* must wait through much of the parse. Not shown in this table is the interaction between the guards and the principles of the theory, which are written as definite clause goals attached to phrase structure rules. These principles might have to be guarded themselves, since they have been written assuming fully instantiated daughters.

Table 6.3: **Outline of quantifier retrieval, using top down parser**

Parser Action	**Rule Completed**	**Guards**
Rule schema1 fires. Mother is compiled.	S $\longrightarrow$ NP VP	guard established on POOL value of entire sentence. In schema1, subject daughter is unified with subject of head daughter.
top-down parser is called on subject NP		
head_spr rule fires; mother is compiled. N bar identified as head daughter and QSTORE values are unified.	NP $\longrightarrow$ DetP NP	
schema2	`a`	

continued on next page

Table 6.3 *top-down parse, continued*

Parser Action	Rule Completed	Guards
lexicon	a	
schema2	unicorn	Determiner identified as specifier of N bar.
lexicon	unicorn	RESTIND:INDEX value for unicorn is found (PER:THIRD,NUM:SING,GEN:NEUT) POOL value for Detp a is unblocked. POOL value for det a is unblocked. POOL value for N bar unicorn is unblocked. POOL value for NP a unicorn is unblocked. (not sure if before or after N) POOL value for N unicorn is unblocked.
	a unicorn	
schema2	VP $\longrightarrow$ V VP	
lexicon	appears	
schema2	VP $\longrightarrow$ V VP	
Rule schema2 fires		
lexicon	to	
schema2	VP $\longrightarrow$ V VP	
lexicon	be	
schema2	VP $\longrightarrow$V	
lexicon	approaching	SUBCAT value for verb approaching is unblocked right away (since subject found) POOL value for verb approaching is computed. POOL value on verb approaching is unblocked.

continued on next page

Table 6.3 *top-down parse, continued*

Parser Action	Rule Completed	Guards
		POOL value for VP `approaching` is unblocked. RETRIEVED, QSTORE values for VP `approaching` is computed. RETRIEVED value for VP `approaching` is unblocked. POOL value for VP `be approaching` is unblocked. RETRIEVED, QSTORE values for VP `be approaching` is computed. RETRIEVED value for VP `be approaching` is unblocked. POOL value for VP `to be approaching` is unblocked. RETRIEVED, QSTORE values for VP `to be approaching` is computed. RETRIEVED value for VP `to be approaching` is unblocked. POOL value for VP `appears to be be approaching` is unblocked. RETRIEVED, QSTORE values for VP `appears to be approaching` is computed. RETRIEVED value for VP `appears to be approaching` is unblocked.

continued on next page

Table 6.3 *top-down parse, continued*

Parser Action	Rule Completed	Guards
		POOL value for S `a unicorn appears to be be approaching` is unblocked. RETRIEVED, QSTORE values for S `a unicorn appears to be pproaching` are computed. RETRIEVED value for S `a unicorn appears to be approaching` is unblocked.

6.4 Detecting Inequations

Our next example, that of binding in Japanese, shows how one can use guards to solve for inequations. Inequations can appear anywhere that descriptions can appear. This might be in constraints, lexical descriptions, and phrase structure rule descriptions. There are two cases of inequation to consider. One is the case of semantic disentailment, which of the two is somewhat easier to understand. If a feature structure and a description do not unify, then disentailment is proven, and, in turn, inequality is proven as well. The other is the case where two feature structures have identical values but are not considered token identical. That is, they must be assigned an inequation to be understood as separate objects. By associating an inequation between feature structures, one is stating that they should not be seen as the identical object, that is, never unified. The former case of semantic disentailment is handled as the negation of the case of semantic entailment described in the previous section. The procedure for proving semantic disentailment simply returns true if the `satisfies` procedure for adding descriptions returns false. The procedure for determining an isomorphism in example 104, however, eagerly unifies tags if they can in fact be unified, which is the step Type(F_1) = Type(F_2), using Prolog =.

For the case of semantic disentailment, the user can choose either to eagerly

unify (causing token identity) or not. If unification is not done, then one continues to wait until there is enough information to disprove isomorphism. Only then can an inequation be proven. Otherwise, inequation as defined by token inequality is still a possibility. The case of delaying on semantic entailment does not explicitly cover the case of token inequality. The routine for isomorphism can only indicate either that semantic entailment has occurred, or that the tags of the two feature structures have already been unified. Then the program must then handle token inequality by separate means.

When an inequation appears in a description, the compiler compiles that as as a call to the `not_satisfies` predicate, which fires when the description is encountered the first time during processing. An example of a description containing an inequation from the "zebra puzzle," which accompanies the ALE package, is given in 121. These constraints, all associated with the type **background**, indicate that the nationalities, pets and beverages of the owners of three houses are all distinct.

(121) Inequational constraints from the "zebra puzzle:"

```
background cons
   (house1:nationality:N1,
    house2:nationality:(N2,(=\= N1)),
    house3:nationality:((=\= N1),(=\= N2)),

    house1:animal:A1,
    house2:animal:(A2,(=\= A1)),
    house3:animal:((=\= A1),(=\= A2)),

    house1:beverage:B1,
    house2:beverage:(B2,(=\= B1)),
    house3:beverage:((=\= B1),(=\= B2))).
}
```

The compilation of one of these descriptions, e.g.

```
(=\= N1)
```

for the nationality of the resident of house 1, looks as follows in the context of the compiled type constraint for the type **background**:

(122) Compilation of Inequations (simplified):

```
typecons_background(bot,SVs,Tag):-
   featval(house1,SVs-Tag,House1),
   featval(nationality,House1,N1),
   featval(house2,SVs-Tag,House2),
   featval(nationality,House2,N2),
   not_satisfies(N1,N2,[]),
   ...
```

`typecons_background` is a list of the type constraints for the type `background` as applied to the feature structure `Tag-SVs`. The predicate `featval` has three arguments, a feature name, a feature structure, and the value of the feature for that feature structure. The inequation is compiled into the `not_satisfies` predicate. The third argument of `not_satisfies` is for passing along any delayed goals that may have been acquired during compilation. In this example, there are none, hence the empty list `[]`. A more detailed example of compiling out inequational descriptions follows next section. This is the example of lexical binding.

6.4.1 Binding and the Japanese Causative

In Manning & Sag (1998) the argument structure of a lexical entry is proposed as the site of variable binding. This is a revised notion of performing binding on the SUBCAT (subcategorization) list of Pollard & Sag (1992); Pollard & Sag (1994). I implement binding theory using delayed evaluation of constraints on the INDEX values of the members of the argument list. In such an implementation delaying over feature structures may be explored in some detail. One reason binding is a good test case is because of the variables embedded within the arguments. Another is that this example calls for a strategy for evaluating inequations, as was pointed out by Penn (1993). We will use the strategy for handling inequations introduced in section 6.4.

The Japanese causative as described in (Manning *et al.* 1999) is an interesting test case for binding constraints because there are two possible co-referents for *zibun-zisin*, the long distance reflexive anaphor, as in example 123. *zibun-zisin* can be co-referenced with either the subject of the causative, the person named *Taroo* or the subject of the embedded predicate, the person named *Ziroo*.

(123) Taroo$_i$ ga Ziroo$_j$ ni aete zibun-zisin$_{i/j}$ o hihan s-ase-ta.
Taroo NOM Ziroo DAT purposefully self ACC criticism do-CAUS-PAST
'Taroo purposefully made Ziroo criticize himself.' (Kitagawa 1986:92)

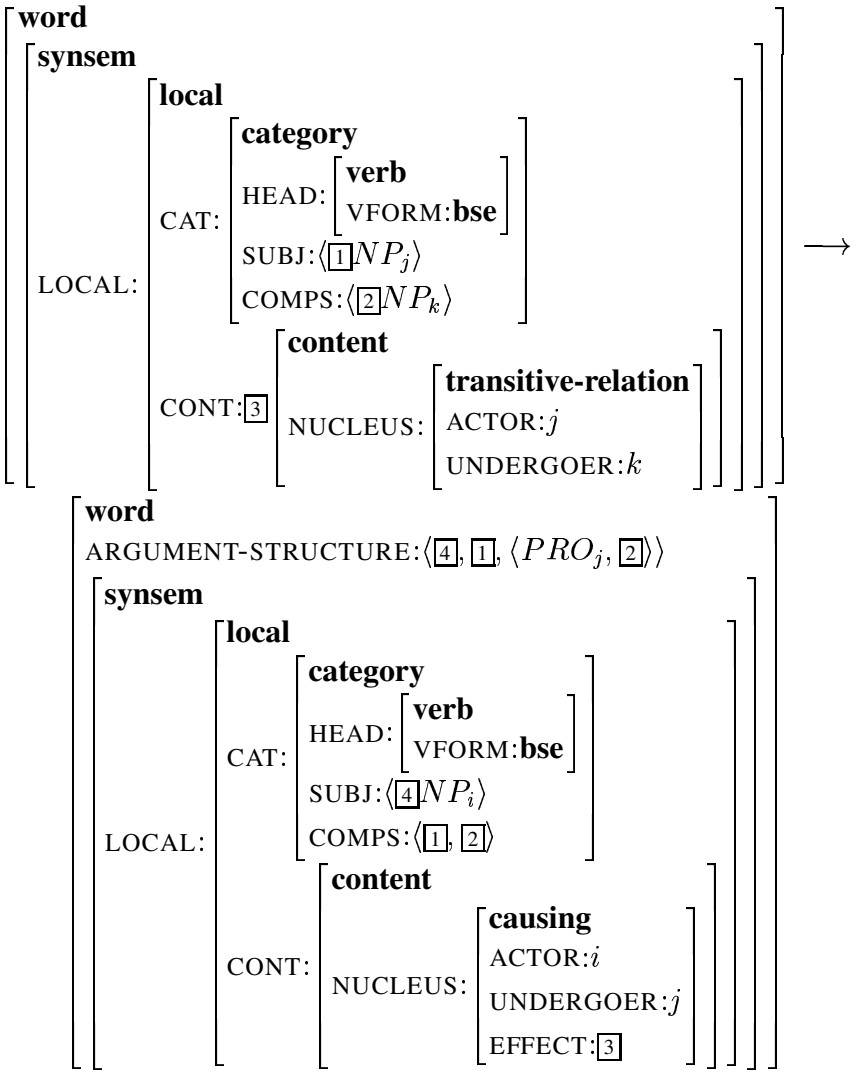


Figure 6.5: Causative lexical rule for Japanese. A causative verb form is derived from a transitive, base form verb. The actor in the transitive action becomes the undergoer in the causative action, and the entire content of the transitive verb is embedded as the content of the causative verb.

A lexical rule, shown in figure 6.5, derives the causative form of a verb from the base form of a transitive verb with the semantic roles of an ACTOR and an UNDERGOER. The ACTOR of the input verb is co-referenced with the UNDERGOER of the causative output verb. This means that the object of the input verb, indexed with k, can be co-referenced with either the subject of the causative NP_i or with the subject of the embedded verb NP_j. The EFFECT of the output verb is structure shared with the semantic content of the input verb. This content is an embedded predicate in the semantics of the whole causative predicate. A subject of type **noun** or **personal_pronoun** must not have the same index as a noun or personal pronoun which is more oblique on the argument list of the causative. This includes nouns and pronoun objects in the embedded predicate. In example 123, *Taroo* and *Ziroo* must not be co-indexed.

A set of binding constraints is realized as a definite clause constraint on the ARG-ST list of the lexical head of a phrase. The subject of the causative must satisfy the description in example 124; that is, it must be either a non pronoun (**npro** or a personal pronoun, and its index value must not be identical to the index value of the embedded argument. The variable `Ind` refers to that index value.

(124) `loc:cont:((npro;ppro),index:(=\= Ind))`

The complete binding constraints appear in example 125. `if_guard` is the implementation of definite clause `if` with a feature structure (`Subj`) specified, and a description as the constraint on which to guard (example 124). The first clause is guarded, and the second is not.

(125) a.
```
check_binding(Subj,[(Ref,loc:cont:((npro;ppro),index:Ind))|Rest])
if_guard
Subj guard loc:cont:((npro;ppro),index:(=\= Ind)) then_else_fail
check_binding(Subj,Rest).
```

b.
```
check_binding((Subj,loc:cont:((npro;ppro),index:Ind))
[loc:cont:(ana,index:Ind)|Rest]) if
 check_binding(Subj,Rest).
```

The indices of the subject and the embedded argument are *inequated*. This means that a guard waits on the feature structure for the subject to satisfy the description in example 124, at which point co-reference between the two indices can be disproved. Then, any waiting goals are fired, which in this instance amounts to recursively checking the reference of the subject (its INDEX value, i)) with any remaining arguments to the right. No guarding is needed to co-reference the subject

and any embedded anaphor (e.g. *zibun-zisin*). This is achieved via unification, associating each with the variable `Ind` in the second clause.

Guarding is used to wait until we can *disprove* that two feature structures are the same. This is the way the system solves inequations. Search is done top-down (or outside-in), depth-first in proving or disproving an isomorphism. If at any point the two feature structures do not unify, the isomorphism fails, and satisfaction is disproved.

6.4.2 Inequation and Token Identity

Implementing example 123 raises for discussion the distinction between token isomorphism and token identity. In particular, the index values of the two masculine nouns *Taroo* and *Ziroo* are the same, even though Taroo and Ziroo are understood to be different people, and hence different "linguistic objects."

The semantic index for the noun in both cases is this:

(126)
```
cont:(npro,index:(ref,Ind,
                  gen:masc,
                  per:third
                  num:sing))
```

This raises the question of how to indicate that the two are inequated. Waiting to prove distinctness of the index values will not help in this case. There are some alternatives:

1. Look outside the index values to see whether the complete feature structures are identical.
2. Assign unique meta "indices" to the index values to indicate that they are distinct. This would be analogous to assigning the *i, j, k* subscripts used in theoretical linguistics. (examples 127 and 128)
3. Assume token inequality unless token equality can be proven via *prior* unification (example 130).

The first solution may not be generalizable in all cases, and so it has not been considered in this implementation. Both the second and third suggestions have been tried. The second solution, of assigning indices, is a grammar change which is transparent to the machinery of guarding. Lexical representations are shown in examples 127 and 128. Use of these indices does return a semantic disentailment. In this case, it is up to the grammar writer to assign unique indices to the nominals in the lexicon. This is only an approximation for knowing the object that a noun

actually refers to in a discourse context, and so use of this method is discouraged. For example, there may be more than one person named *Taroo* in the room, in which case one would need two different lexical entries.

(127)
$$\begin{bmatrix} \textbf{local} \\ \text{CONTENT:} \begin{bmatrix} \textbf{npro} \\ \text{INDEX:}\boxed{3} \begin{bmatrix} \textbf{ref} \\ \text{UNIQUE:}\textbf{i} \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

(128)
$$\begin{bmatrix} \textbf{local} \\ \text{CONTENT:} \begin{bmatrix} \textbf{ppro} \\ \text{INDEX:}\boxed{4} \begin{bmatrix} \textbf{ref} \\ \text{UNIQUE:}\textbf{j} \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

The third solution is a more generalizable solution and has been implemented as an alternative version of the `not_satisfies` procedure. The linguist could perhaps indicate via a switch whether to assume that token inequality holds or token identity holds as a default for feature structures which are unifiable. This is an issue for further study.

The difference between the two versions is subtle. The first (129) attempts to find an isomorphism between description `Desc` and feature structure `Tag-SVs` by passing a return value of 'fail.' The call to `iso` (example 104) is implicitly guarded. `not_satisfies` will succeed as long as `iso` does not know enough to return `true`. The goals will be solved as long as the initial call succeeds. The pruning step simply rewrites goals in an easier to solve form, based on new information which might have been learned about the variable arguments. If the initial call eventually fails, then the goals will be retracted during backtracking. The point here is that if enough information is not gained, there may not be backtracking. The second version (130) passes the guarded goals `Gs` to `iso`. They will only be executed when the isomorphism is disproved. Otherwise, they will remain guarded.

(129)
```
%% this version is eager; as long as two feature
%% structures are still not proven isomorphic,
%% the guards are solved.  In some linguistic
%% instances, that might be preferable if some
%% values are generally never going to be
%% instantiated
not_satisfies(Desc,SVs2,Tag2,Gs) :-
iso(Desc,Tag2-SVs2,[],fail),
```

```
    prune(Gs,Gsout),
    solve(Gsout). %% since you may still be waiting.
```

(130)
```
    %% this version is "true" delaying; don't solve Gs
    %% until you're sure whether two FS are isomorphic
    not_satisfies(Desc,SVs,Tag,Gs) :-
    iso(Desc,Tag-SVs,Gs,fail).
```

The implemented solution (130) correctly finds two possible co-referents for *zibun-zisin*, the long distance reflexive anaphor. The code for `iso` is in Appendix A. *zibun-zisin* can be co-referenced with either the subject of the causative or the subject of the embedded predicate. The two results show the object of the input verb, indexed with k, co-referenced with the subject of the causative NP_i and with the subject of the embedded verb NP_j. The example of Japanese presents a case in which specific indices for gender, number and person were not explicitly assigned by the grammar, and so the feature values for the feature INDEX were unresolved. The program can either assume uniqueness of all instances or unify all instances that have not had uniqueness disproved. Another possible solution would be to use default values, and to assume a default value for features which are waiting on a value that is never set. This is a way of breaking "deadlocks" which occur when constraints are waiting on each other, and do not fire. Other ways to break deadlocks would be to start e.g. a breadth-first search.

6.4.3 Related Work

One alternative to the above implementation of binding is a publicly available implementation of the HPSG binding theory is the teaching grammar available as part of the Linguistic Grammars Online Project (LinGO) at Stanford University (Copestake 1992; Copestake *et al.* 1999). The grammar is an implementation of the textbook grammar of Sag & Wasow (1999). This grammar implements the Anaphoric Agreement Principle, which is paraphrased thus:

> Principle A states that 'an [ANA +] **synsem-struc** must be out ranked by a co-indexed **synsem-struc**' and B that 'An [ANA -] **synsem-struc** must not be.' ... co-indexed elements share the same AGR value.

The grammar is apparently a brute force implementation of binding constraints. Nine possible instantiations of the binder and bound arguments are handled by rules, and these rules apply to a type of word, prior to the application of grammatical rules. These rules are provided in Appendix D. They perform the same basic function as the `check_binding` constraints in 125.

Rules for argument realization are applied at the same point as the binding rules. These rules set up the correspondence between the arguments on the ARG-ST list and the SUBJ, COMPS, and SPR lists. There is also a set of GAP rules which account for the correspondence between items on the ARG-ST list and extracted elements on the GAP list. This a situation analogous to the complement extraction example in chapter 5. A few examples are shown in figure 6.6. It is not a surprise that both the binding rules and the GAP (complement extraction) rules are commented out, with notes about performance and the "virtual unreadability" of the parse chart, because of a proliferation of entries. The approach used in this grammar is a dead-end, as its writers have found. Problems are due to the adoption of the "brute force" strategy outlined in chapter 2.6.1. Search time is increased due to an increase in possibilities that will never be needed. This is particularly obvious with bottom-up parsing. With a top-down parser, the grammar rule selected for a verb phrase will impose constraints on the lexical entry for the verb which heads the rule. Some search will come into play in deciding which of the many lexical entries is consistent with the constraints in the grammar rule, but it may be minimal, if lexical entries are indexed. In top-down parsing, the movement of the above-mentioned argument-realization and binding constraints to the point of selecting the lexical entry would be ideal, which is essentially what happens with the guarded approach we present in this chapter.

6.5 Comparison with Other Methods

In chapter 2 we introduced three different methods that might be used for evaluating underspecified arguments in the absence of guarding. These were brute force, accomodation, and control. Brute force is the explicit expression of the constraint to be satisfied via instantiation of all its possibilities. Accomodation is hand-coded software or grammar changes, to account for anticipated problems. Control is stating constraints where they optimally occur. We return to these here and compare their performance with the performance of guarding.

6.5.1 Brute Force

We have looked at the implementation of lexical binding in the case of the Japanese causative.

```
arg_real-principle-GAP2 :=  word+arg_real &
[ SYN [ HEAD #head,
        SPR #spr,
COMPS < #1 >,
GAP <! #2 !> ],
  SEM #sem,
  ARG-ST #arg-st,

  ARGS < word+arg_real & [ SYN [  HEAD #head,
                                  SPR #spr,
                                  COMPS < #1, #2 >,
                                  GAP <! !> ],
                           SEM #sem,
                           ARG-ST #arg-st] > ].

arg_real-principle-GAP2b :=  word+arg_real &
[ SYN [ HEAD #head,
        SPR #spr,
COMPS < #1, #3 >,
GAP <! #2 !> ],
  SEM #sem,
  ARG-ST #arg-st,

  ARGS < word+arg_real & [ SYN [  HEAD #head,
                                  SPR #spr,
                                  COMPS < #1, #2, #3 >,
                                  GAP <! !> ],
                           SEM #sem,
                           ARG-ST #arg-st] > ].
```

Figure 6.6: Excerpt from a teaching grammar which shows lexical overgeneration. The rules are two of the realizations of a complement (#2) from the COMPS list in the GAP position (topicalized). Later, the authors revise the rules so that the extracted complement is identified with type **topic**. (Callison-Burch & Guffey 1999).

(131) do-CAUS-PAST Taroo$_i$ ga Ziroo$_j$ ni aete zibun-zisin$_{i/j}$ o hihan s-ase-ta.
Taroo NOM Ziroo DAT purposefully self ACC criticism
'Taroo purposefully made Ziroo criticize himself.' (Kitagawa 1986:92)

One can compare the results for guarding on this example with the "brute force" method of listing all possible lexical instantiations. Because inequations can be handled without delays, the Japanese example, as is true of all of the examples presented here, does not require delays to be parsed properly. However, a naive implementation of this same grammar without delays may result in 20 lexical entries for the 3-place causative, reflexive verb *hihansaseta*.[6] This number of entries corresponds with the legal permutations of personal pronoun, noun, and reflexive anaphoric pronoun in a 3-place predicate, as shown in example 132. This is the case where all of the possible binding relations are compiled out in advance, rather than waiting to apply the relations after the binder and bound objects have been located. Disjunctions between a personal pronoun and a noun, for example, are compiled out. The parse time on this example is greatly reduced with the reduction in the number of lexical entries, due to the reduction in local ambiguity. Table 6.4 shows binding facts result in 22 lexical entries (no delays) vs. 2 (with delays). The corresponding run time is 2.5 sec. (no delays) v .54 sec (delays) on a SPARC 5 (only 21% of the former time).

(132) noun noun noun
noun noun pron
noun noun$_i$ ana$_i$
noun pron noun
noun pron pron
noun pron$_j$ ana$_j$
noun$_i$ ana$_i$ noun
noun$_i$ ana$_i$ pron
noun$_i$ noun ana$_i$
noun$_i$ pron ana$_i$
pron noun noun
pron noun pron
pron noun$_j$ ana$_j$
pron pron noun
pron pron pron
pron pron$_j$ ana$_j$

[6]The actual test grammar resulted in 22 lexical entries.

pron$_i$ ana$_i$ pron
pron$_i$ ana$_i$ noun
pron$_i$ noun ana$_i$
pron$_i$ pron ana$_i$
noun ana ana
pron$_i$ ana ana
pron$_i$ ana$_i$ ana

Table 6.4: Parse time for the Japanese causative verb *hihansaseta*

Guards	**Num lexical entries**	**Parse time**
no	22	2.5 sec
yes	2	.54 sec

The final version of the binding constraint implemented with delays is given in figure 6.5.1. The difference between this version and the version discussed in the previous chapter is that here, there is no guarding in the definite clauses themselves. All of the guarding takes place on the index values in the lexical entry, in the style of the German auxiliary raising example. The lexical rule for the causative with guarding added is in figure 6.5.1. Inequations continue to be implicitly guarded. Therefore the binding constraint is independent of the lexicon, and may be shared by two different test grammars. In a grammar without delays, in order to avoid a proliferation of lexical entries, the constraint must be threaded by hand after the verb arguments have been parsed. Two lexical entries are produced in the guarded version due to having a disjunctive description in the guard. The guard fires when it is known whether the index value of the subject belongs to either a non-pronoun or a pronoun.

(133) `Subj guard loc:cont:(npro;ppro)`

6.5.2 Accommodation

The German example of partial verb phrase fronting is parsed by attaching guards onto a lexical entry. A variety of methods may be used to parse this grammar without guarding. The most effective technique in this case is accommodation.

```
check_binding(e_list) if true.

check_binding([Subj|Rest]) if
check_binding(Subj,[],Rest).

check_binding(Subj1,Prevs,[[Subj2|Tail2]]) if
(check_binding([Subj2|Tail2])
;
check_binding(Subj1,Prevs,Tail2)).

check_binding((Subj,loc:cont:((npro;ppro),index:Ind))
,Prevs,
[loc:cont:(ana,index:Ind)|Rest]) if
 check_binding(Subj,Prevs,Rest).

check_binding(loc:cont:(npro;ppro),_,e_list) if true.

check_binding(Subj,
Prevs,
[(Ref,loc:cont:(npro;ppro))|Rest]) if
 not_bound(Ref,Prevs),
 check_binding(Subj,[Ref|Prevs],Rest).


not_bound((Ref,index:Ind),[index:(=\= Ind)|Rest]) if
not_bound(Ref,Rest).

not_bound(_Ref,e_list) if true.
```

Figure 6.7: Lexical binding constraints as a Prolog definite clause. The disjunction in the third clause accounts for the fact that the objects in an embedded predicate, such as is found in the causative form, may be bound by either the subject of the embedded clause or the subject of the embedding clause.

```
causative lex_rule
  (word,
   synsem:(loc:(cat:(head:(verb,
                           vform:bse,
                           aux:minus),
                     subj:[StemSubj],
        comps:[StemObj]),
                cont:(Cont,nucleus:(transitive_reln,actor:Indj)),
                conx:Conx),
          non_loc:NL),
   qstore:QStore,
   qretr:QRetr)
  ***>
  (word,
   arg_struct:(Arg_struct,[Subj,
                           StemSubj,
                           [(@ np(Indj),loc:cont:ppro),StemObj]]),
   synsem:(loc:(cat:(head:(verb,
                           vform:bse,
                           aux:minus),
                     subj:[(Subj,@ np(Indi))],
            comps:[StemSubj,StemObj],
                     spr:[]),
                cont:nucleus:(causing,
                              actor:Indi,
                              undergoer:Indj,
                              effect:Cont),
                conx:Conx),
          non_loc:NL),
   qstore:QStore,
   qretr:QRetr)

if_guard

Subj guard loc:cont:nom_obj then_else_fail

check_binding(Arg_struct)

  morphs
    X becomes (X,s,a,s,e).
```

Figure 6.8: Causative lexical rule in Japanese (ALE version), showing guarding on the subject argument of a verb (`Subj`) before the binding constraints fire.

One way to address the problem of non-termination is to prescribe a limit on the length of the complements list of a verbal head. This is added to the PVP lexical rule in the naive implementation of PVP fronting introduced in chapter 2. (The lexical rule is repeated here as figure 6.9 for convenience.) The lexical rule in figure 6.9 avoids an infinite depth-first search with the relation `three_or_less(PVPComps)`. Under the guarded approach, this relation disappears. This is because the complements of the partial verb phrase, `PVPComps`, are guarded until all complements have been found. The conditions attached to the lexical rule are simplified as well by assuming that auxiliary raising is a separate constraint. The guarded lexical rule, discussed in chapter 5, is shown as figure 6.10.

We find that the naive grammar in figure 6.9 introduces 10 lexical entries for the base form of each auxiliary, each of which generates inflected forms. A different, less ambiguous way to use accomodation in this lexical rule is to put constraints on the types of arguments a PVP may have, before it takes it arguments. e.g. `check_phrasal` and `check_nominal` in 134 are requirements on the types of the members of the complements list of a partial verb phrase, to reduce the overgeneration of a brute force method. Performance of both of these grammars is reviewed in the following section.

(134)
```
check_nominal([]) if
  !,true.

check_nominal([loc:cat:head:(noun;prep)|Synsems]) if
  check_nominal(Synsems).

check_phrasal([]) if
  !,true.

check_phrasal([loc:cat:lex:minus|Synsems]) if
  check_phrasal(Synsems).
```

6.5.3 Control via Ordered Rules

The guarded approach is in general a move toward abstraction, of attaching a principle to a type, and away from a position in the grammar where the principle applies. It is thus a move away from attaching ordered conditions on phrase structure rules. For example, the only way to handle the German example of raising by auxiliary without delaying the constraint is to hand-thread the constraint on

```
%%   % Partial Verb Phrase Fronting Rule
pvp lex_rule
   (word,
    subcat:[Subj|OldComps],
    synsem:(loc:(cat:(head:(Head,verb,vform:bse,aux:plus,flip:minus),
                      subj:[Subj],
                      comps:OldComps)),
            non_loc:(inherited:(slash:OldSlash))))
   **>
   (word,
    subcat:[Subj|SubcatComps],
    synsem:(loc:(cat:(head:Head,
                      subj:[Subj],
                      comps:PVPComps),
                 cont:Cont),
            non_loc:(inherited:(slash:(elt:(PVP,cat:(head:(verb,vform:bse),
                                                     lex:minus,
                                                     subj:[Subj],
                                                     comps:PVPComps),
                                                cont:Cont),
                                       elts:OldSlash)))))
%% The next line is key
if
   (append(VComps,[(loc:cat:(head:(verb,vform:bse),
                             lex:plus,
                             comps:VComps,subj:[Subj]))],
                             OldComps),

    three_or_less(PVPComps),
    append(PVPComps,
           [(loc:PVP,non_loc:(inherited:(slash:(elt:PVP))))],
           SubcatComps)
```

Figure 6.9: PVP rule – naive implementation

```
pvp lex_rule
     (word,
      synsem:(loc:(cat:(head:(Head,
                              verb,vform:bse,aux:plus,flip:minus),
                        subj:[Subj],
                        comps:PVPComps),
                   cont:Cont),
              non_loc:(inherited:(slash:e_set)))),

     ***>
     (word,
      subcat:[Subj|SubcatComps],
      synsem:(HeadSyn,
      (loc:(cat:(head:Head,
                         subj:[Subj],
                         comps:PVPComps),
                    cont:(Cont,nucleus:modal_arg:Prop)),
               non_loc:(inherited:
                        (slash:(elt:(PVP,
                                     cat:(PVPHead,
                                     head:(verb,vform:bse),
                                               lex:minus,
                                               subj:[Subj],
                                               comps:PVPComps),
                                               cont:Prop),
                                elts:e_set))))),

if_guard
  Prop guard nucleus:relation then_else_fail

(append(PVPComps,
                [(loc:PVP,
                  non_loc:(inherited:(slash:(elt:PVP))))],
                SubcatComps),
        aux_raising(HeadSyn,PVPComps,Subj))
```

Figure 6.10: Partial verb phrase fronting as guarded lexical rule.

extracted complements through the grammar, such that it fires at the appropriate time, that is once the auxiliary, matrix verb and complements have been identified.

In the example of quantifier raising with local quantifier storage, the semantics principle is used to illustrate how one can guard on a specific type in the grammar, in this case, on phrases with verbal heads. But it is not necessary to make the unification of the content values on the mother and semantic head daughter in this principle be a type constraint, much less a guarded one. It is possible to unify these directly in the head-complement phrase structure rule. And so the semantics principle can be divided into portions which are guarded and portions which are not. This departs from a "pure" implementation, in which the principle is implemented as a single constraint, but performance needs to be considered.

There are different ways to implement constraints on types. In a grammar with phrase structure rules, it is possible to build some constraints between a mother and its head directly into the rule itself. Guarded constraints are appropriate for situations where there are several arguments which contribute information to a head which subcategorizes for them, and there may be quite a distance between that head and one or more of those arguments. Especially in a binary branching grammar, it is not transparent how to apply a constraint across multiple arguments within the context of a single rule. They can instead be attached to the sort which is the type of the mother node of the rule.

Guarded constraints are only as effective as the conditions on which the constraints are blocked. The verb raising example points out a problem with early or late unblocking of guarded constraints. In the sentence

(135) Wird Kim Sandy sehen können?

the system waits to evaluate the arguments of *werden* until all have been found. Since *können* is also an auxiliary verb, its semantics are not known until it has found its own arguments. An unforeseen interaction occurs if the constraint which triggers auxiliary raising has not been unblocked at the time that the first auxiliary *werden* encounters *können* as a possible argument. The parser attempts to use PVP-slashed versions of both *werden* and *können* in a parse of example 135. The SLASH values of the two auxiliaries are unified as the Non-local Feature Principle is executed, since they are consistent. As a result, the first auxiliary is seen as subcategorizing for itself, since it subcategorizes for anything that the PVP in SLASH subcategorizes for. An infinite cycle results. The solution to this problem is to change the conditions on unblocking. The generality "all arguments are known" means knowing the key facts about each element, generally, either syntactic head

features or semantic state of affairs. In this example, the constraint on auxiliary raising can, and should, fire instead as soon as syntactic features are available.

Sentences in table 6.5 are shown parsed with four grammars: a "naive" grammar with the accomodation strategy in figure 6.9; a grammar with auxiliary raising as a separate constraint but without guarding; the same grammar with guarding; and an "ideal" grammar with constraints added to the lexical rule for partial verb phrase fronting, to reduce the ambiguity that results from overgeneration by this rule. The constraints added are that the complements of the PVP must be phrasal and that they must be nominal. The PVP lexical rule for this grammar is shown in Appendix E.

Table 6.5: Parse times (in msec) for argument raising by auxiliary

Sentence	**Naive Unguarded**	**Unguarded**	**Guarded**	**Threaded**
Sandy sieht Kim.	70	60	70	70
Sandy wird Kim sehen können.	5590 (mult. parses)	4990 (mult. parses)	3620	840
Sandy sehen wird Kim.	1070 (m)	470 (m)	350	200
Sandy wird Kim sehen.	1240 (m)	810 (m)	710	280
Sandy sehen können wird Kim.	3450 (m)	2010 (m)	1110	390
Wird Kim gehen können?	2620 (m)	2120 (m)	2210	330
Wird Kim sehen können?	4260 (m)	3990	3270	610
Sehen können wird Kim Sandy?	3860 (m)	no	1020	no
Wird Kim gehen?	540 (m)	1460	430	120
wird Kim können (VP)	1860 (m)	940 (m)	580	160
Wird Kim Sandy sehen?	1220 (m)	1230 (m)	1570	320
Wird Kim Sandy sehen können?	5850 (m)	8240 (m)	8900	880

The guarded German grammar performs faster than the unguarded grammar in the average case, and, unlike that grammar, has no ambiguities. However, both of these grammars perform up to ten times slower than the best grammar without guards for sentences with an initial and final auxiliary e.g. *Wird Kim Sandy sehen können*. In the grammars without guards, raising by auxiliary occurs in phrase-structure rules, rather than the lexicon, and so constraints have been hand-threaded. In the guarded grammar, argument raising by auxiliary is guarded in

lexical entries for auxiliary and also in the PVP lexical rule. This is essentially an experiment to see whether such a constraint can be expressed wholly in the lexicon. The answer is that yes, it can. However, we learn that guards need to be implemented more efficiently in order to speed up performance. The slowdown in the guarded grammar is due to blocking and unblocking of guards when an auxiliary is the head of a phrase structure rule. As each new constituent added to the chart for phrases headed on the right, the constraints unblock under the premise that all arguments may have been found. If the constraint is not satisfied, the blocking continues under backtracking. The process repeats for verb-initial phrases headed on the left. In sum, the grammar keeps checking to see whether an auxiliary has found all of its arguments or not. The advantage of guarding is that this grammar will handle all of the PVP constructions. Furthermore, no accomodation strategies need to be developed.

It is important to point out that only the naive grammar and the guarded grammar properly allow a fronted double infinitive (136):

(136) Sehen können wird Kim Sandy?

This is because the arguments of a partial verb phrase (`PVPComps`) are concatenated with the PVP (`loc:PVP`) to form the argument of the main clause auxiliary (`SubcatComps`) in the PVP lexical rule:

(137)
```
append(PVPComps,[(loc:PVP,
                      non_loc:(inherited:(slash:(elt:PVP))))],
               SubcatComps)
```

In the guarded grammar, `append` is guarded. In the unguarded grammar, `append` is not guarded. Since the fronted PVP's arguments are not instantiated yet at the point of lexical rule application, they match the first clause of `append`, which automatically instantiates them to be an empty list. The lexicon compilation will go on indefinitely if there is no "cut" (halt) in `append`, as we have seen. This means that in the unguarded grammar, there is no chance that example 136 will parse. One needs to use a constraint similar to the naive strategy discussed in section 6.5.2 in order for the complements list to be of variable size. Parse times for the naive strategy in table 6.5 show that overall, it a less desireable strategy than adding the constraints in example 134 (the "ideal" situation).

6.6 Summary

In this chapter we find a successful implementation of guarded constraints as an instance of Constraint Logic Programming. We try the same guarded grammars with different parsers, and find no differences in the output of the parse, though the constraints have been delayed in different orders. We revisit brute force, accommodation and control as alternatives to guarding for the chosen grammars. Practical advantages of guarding are that lexical size may be reduced substantially, and grammar rules may streamlined in the direction toward a statement of constraints on more abstract grammatical concepts, such as types, or typed feature structures. Performance improvements are thus tied to gains associated with improvements to the grammar. Guarding itself is associated with performance overhead, so must be balanced with performance gains elsewhere in order to show an overall improvement in the parse. In our implementation, a guard may be unblocked prematurely and reblocked, which slows down the parse.

The examples in this chapter provide some detail for guarding on feature structures. Examples from Japanese, German, and English are successfully parsed using guards. The main thread in all of these examples is that arguments which provide information to a head may be processed at a later time than the head itself, and yet the constraint may be "attached to," or associated with, the head. In the example of the causative, the arguments of a verb have indices which must be instantiated in order to apply binding constraints. In the example of argument attraction by auxiliary, the auxiliary waits for its verbal head as well as the verb's arguments before a constraint on fronted constituents can fire. And in the example of quantifier raising, the pool of possible quantifiers that a verb has must wait until all of the verb's arguments, including the subject, have made their contribution. Yet, retrieval still occurs at verbal nodes. I show that the Japanese example results in a total of two lexical entries for the causative, compared with over 20 entries otherwise, which is more efficient to process. The German and English examples are full and correct implementations of these theories as written. In the case of German, the fronted double infinitive construction does not parse without either guarding or using a naive strategy that, like the case of the Japanese causative, also increases local ambiguity. In the last chapter I review the main points of the thesis and discuss future work.

Chapter 7

Conclusions

The major contribution of this work is a specification and implementation of delays on typed feature structure descriptions. An implementation of this specification enables the successful evaluation of modern grammars as written, while retaining the succinctness of the original grammar. Beginning with a formal description for guarding and then proceeding to an implementation, the highlights are the following:

1. Specification of guarded descriptions for typed feature structures.
2. An algorithm for resolution over feature structure terms with delays. Because any description may be a guard, one is able to guard various objects in the grammar. In turn, grammatical constructs such as lexical rules and lexical entries can be guarded by referencing these descriptions.
3. Three independent examples of guarding in practice, which are predicate sharing in the Japanese causative, raising by auxiliary in the case of German verb phrases, and locally determined quantifier scoping in English.

The thesis is generalizable such that the mechanism of guarding can be applied to a wide range of linguistic phenomena and also across various languages. Future work includes researching its applicability across various feature structure based grammar formalisms.

7.1 Guarded Descriptions

The specification of a guarded feature structure description from chapter 5 is repeated below. The gain in expressive power is that the descriptions themselves

become procedural statements whereas the resolution algorithm can remain unchanged.

A guarded description is a description of the form

- $\phi \rightarrow \psi; \upsilon \in \mathbf{Desc}$ if $\phi, \psi, \upsilon \in \mathbf{Desc}$

The entailment conditions for guarded descriptions are the following:

(138) for ϕ, ψ, $\upsilon \in$ Desc
$F \models (\phi \rightarrow \psi; \upsilon)$ iff
- $F \models \phi$ and $F \models \psi$ (entailment) OR
- $(\forall F')F \sqcup F' \not\models \phi$ and $F \models \upsilon$ (disentailment)

We defined negation as a special case of a guarded description:

- $\neg(\phi) = (\phi \rightarrow fail;\ true)$

With the addition of negation, inequations may be implemented as guarded descriptions, or with guarded rules. The implementation of the binding theory in the case of the Japanese causative handles inequations as guarded constraints. The algorithm for resolution with guarded rules is the following:

(139) Given goal $p(f_1, \ldots, f_n)$ and a (renamed) clause $C \mid A' \leftarrow B_1, B_2, \ldots, B_n$, such that goal p, A', C unify with mgu θ, and C_i a constraint on f_i, $C_i = X_i : \phi \in C$:

- If f_i neither satisfies nor dissatisfies ϕ, a *suspension* of a goal $p(f_1, \ldots, f_n)$is created. A suspension is a pointer to the conjunction of goals which are delayed by the constraint. If ϕ is already constrained to satisfy any $p'(f_1, \ldots, f_n)$, then p is added to the conjunction of goals.
- p is wakened if X_i is unified with a feature structure that satisfies ϕ. Then, the action taken is to first execute the suspension of p, and then resume the present goal.

7.2 Extensibility

The data covered in implementation supports the generality of the thesis such that it is extensible to a wide range of linguistic analyses, and is language independent. The specification is generalizable for linguistic analyses of morphologically

shared predicates, verb raising, long-distance dependencies, clitics, quantifier raising, and linear order constraints. The reason that all of these phenomena can be unified by the analysis presented is that in all cases, there are arguments which are shared across structures by more than one head. A constraint may exist between a head and one of its arguments, yet another head may project the features which fully instantiate that argument at a later time.

The description of the resolution algorithm is abstracted away from the parsing architecture, providing a practical advantage for implementation efforts. The gain is advancement along a continuum toward the statement of independent constraints and away from the ordering of rules. Another approach, the proliferation of possible instantiations in the case of "brute force," is also eliminated, and gains are made especially by reducing the number of locally ambiguous lexical entries.

7.3 Scalability

We have seen that the addition of guards to the German grammar slows down its performance compared with a non-guarded grammar, up to 10x for simple sentences. This would only get worse with longer sentences and more guarded constraints in the grammar. It is not hard to see that for speed alone, it is preferable to write a grammar without guards. We have shown, however, that certain linguistic analyses make it necessary to have an ordering on constraint resolution, such that evaluation of shared or uninstantiated arguments should be delayed. Guards are a way to state constraints where they appear in a linguistic theory and have them evaluated at the proper time as well. Properly writing a hand-threaded grammar without guards (a grammar with constraints stated at the place where they are best evaluated) would be in the least case knowledge intensive, and may be more difficult to teach in the long run than delays. This is because a grammar writer would need to become familiar with the algorithm of the parser or generator. In the case of delays, the training would be focused more on the facets of the expressive grammar that might be problematic for evaluation in the general case.

The description language used in this thesis is a change from the HPSG theory of Pollard & Sag (1994). I have advocated an addition to typed feature structures that is not in the original theory. Carpenter (1991) showed that grammars with typed feature-structures and rewriting rules are undecidable. This means that there is no way of stating whether we can find a solution for a fragment of linguistic theory without guards vs. its guarded counterpart. A grammar with guards would be harder to write than one without guards, but a grammar without guards might

not terminate. On the other hand, a grammar with guards may run into deadlocks that would need to be resolved with a strategy such as breadth-first search. In the cases presented in this thesis, deadlocks were not a problem.

Feature based grammars are knowledge-based as compared with the empirical methods that became popular as a viable way to do NLP beginning in the late 1980's (e.g. Church 1988, Collins 1996). Statistical systems can be trained from very large corpora with minimal linguistic input. A good reference on this subject is Manning & Schütze (1999). Knowledge-based systems require a staff of linguistically-trained people managing a large lexicon and grammar, largely by hand, and are generally less robust than statistically trained systems. The relative disfavor of such grammars has contributed to the fact that logic programming is not widely used for NLP. Another contributing factor is that goals in standard CLP are solved for top-down, while a combination of top-down and bottom-up methods is probably best for NLP parsing systems. ALE, for example, has a bottom-up chart parser.

Concurrent Constraint Programming has seen the introduction of Constraint Handling Rules (CHR) (Frühwirth & Abdennadher 1997) into Prolog and other logic programming language.s These allow for dynamic scheduling of constraints. Once a constraint is introduced, a constraint handler takes care of deciding when it should be evaluated, by a subsumption check. This is, then, the next generation of delays in logic programming. One advantage to using the older guards for this thesis is that guards could be built into the description language, so that it became straightforward to show how the theory doubled as an implementation. Penn (2000) used CHR with ALE with a very different data structure for feature structures, closely related to a Prolog term, but one requiring attributed variables. Early results were very slow compared with Prolog guards.

Whether or not logic programming is the actual programming scheme used, the fact of having to solve linguistic constraints occurs across implementations, and also across linguistic theories. One assumes interdependence among the constraints. This thesis shows one way to cope with that interdependence during implementation.

7.4 Future Work

Future work should include re-implementing the satisfaction algorithm to maximize efficiency. In a test situation, hand-threading guards in the German grammar decreases processing time vs. the guarded version for sentences with initial auxil-

iaries, as there is no overhead due to the unblocking and reblocking of constraints. There is no absolute need to trigger a constraint check each time the type variable for a feature structure changes, unless the desired type has been obtained. In other words, the satisfaction conditions are currently a constraint that wait on the gain of any new type information by a feature structure. The satisfaction condition itself could be the delay constraint in a more sophisticated implementation. That is, wait on the satisfaction conditions themselves. That is the approach that I favor here.

One possibility for future work relates to the generalizability of the approach for other grammar formalisms. This would be to try the guarding mechanism on feature structures that are untyped. The KANT knowledge based machine translation system (Nyberg & Mitamura 1992; Mitamura & Nyberg 2000) has a large working grammar (544 rules) with rules in the Lexical Functional Grammar formalism. Each rule can have many lines of equations attached. The grammar writer must be aware of the order in which the equations fire so that tests on existing values can be performed at the proper place. This knowledge of ordering is also very much reliant on knowledge of the parser strategy. One can imagine a scenario in which the feature value names are presumed to be types, and the values of the features are tested once they are instantiated, if not in the current rule, then in a future rule. It is also possible with to remove equations to a unique set of constraints that become rule independent.

Appendix A

Satisfaction of a description by a feature structure

```
satisfies(Desc,Tag-SVs,Gs) :-
var(Desc),
!,iso(Desc,Tag-SVs,Gs,true).

satisfies(Desc,FS,Gs) :-
var(FS),
!,when(nonvar(FS),satisfies(Desc,FS,Gs)).

satisfies(Desc,Tag-SVs,Gs) :-
var(Tag),var(SVs),!,
when(nonvar(Tag),satisfies(Desc,Tag-SVs,Gs)).

satisfies(Desc,Tag-SVs,Gs) :-
        !,satisfies(Desc,SVs,Tag,Gs).

satisfies([],SVs,Tag,Gs):-
!,satisfies(e_list,SVs,Tag,Gs).

satisfies([H|T],SVs,Tag,Gs):-
!,satisfies(hd:H,SVs,Tag,[]),
satisfies(tl:T,SVs,Tag,Gs).

%% variation on satisfies(Type) (see below)
%% ---------------------------------------
```

```
satisfies(Feat:Desc,SVs,Tag,Gs) :-
!,introduce(Feat,Type),
type(Type),
nonvar(SVs),
deref(Tag,SVs,TagNew,SVsNew),
SVsNew =..[Type2|_],
%% The result has to be the type of SVs
%% means that Type2 is type or subtype of Type
(unify_type(Type2,Type,Type2),!,
 featval(Feat,SVsNew,TagNew,Tag2-SVs2),
 satisfies(Desc,Tag2-SVs2,Gs)
     ;
extensional(Type2),!,fail
    ;
when(nonvar(TagNew),
(
    fully_deref_k(TagNew,SVsNew,TagOut,SVsOut),
    satisfies(Feat:Desc,TagOut-SVsOut,Gs))
    )).
% ----------------------------------------

%% satisfies(Type...) uses Prolog guarding.

satisfies(Type,SVs,Tag,Gs) :-
type(Type),!,
nonvar(SVs),
deref(Tag,SVs,Tag2,SVs2),
SVs2 =..[Type2|_],
(unify_type(Type2,Type,Type2),!,
         prune(Gs,GsOut),solve(GsOut)
     ;
extensional(Type2),!,fail
    ;
when(nonvar(Tag2),
(fully_deref_k(Tag2,SVs2,TagOut,SVsOut),
 satisfies(Type,TagOut-SVsOut,Gs)))).

%% If the type of the feature structure Tag-SVs is not rich enough to
%% determine satisfaction, then delay on its tag.  If the type of the
```

```
%% FS changes, the tag will also change.

% ------------------------------------------------------------

satisfies((D1,D2),SVs,Tag,Gs) :-
!,satisfies(D1,Tag-SVs,
[fully_deref_k(Tag,SVs,NewTag,NewSVs),
 satisfies(D2,NewTag-NewSVs,Gs)]).

satisfies((D1;D2),SVs,Tag,Gs) :-
  !,
  (satisfies(D1,Tag-SVs,Gs);
  satisfies(D2,Tag-SVs,Gs)).

satisfies((=\= D1),SVs,Tag,Gs) :-
!,not_satisfies(D1,SVs,Tag,Gs).

satisfies(Tag-SVs,SVs2,Tag2,Gs) :-
iso(Tag-SVs,Tag2-SVs2,Gs,true).

% iso/4 (FS1:<fs>, FS2:<fs>, Gs<goals>, Done<boolean>)
% ------------------------------------------------------------

% determines whether structures FS1 and FS2 are isomorphic.
% Gs are guarded goals which the user may define in the case the isomorphism
% proves satisfaction.

% Done is true if the feature structures are isomorphic or fail if
% the isomorphism cannot be proven.  iso succeeds as long as guards persist.
%
% ------------------------------------------------------------


%% Assume that two feature structures are isomorphic, by unifying
%% their tags.  Then recursively see if their feature values are isomorphic.


iso(Tag1-SVs1,Tag2-SVs2,Gs,Done) :-
fully_deref_k(Tag1,SVs1,Tag1Out,SVs1Out),
fully_deref_k(Tag2,SVs2,Tag2Out,SVs2Out),!,
```

```
prune(Gs,GsOut),
delay_iso(Tag1Out-SVs1Out,Tag2Out-SVs2Out,GsOut,Done).

delay_iso(Tag1-SVs1,Tag2-SVs2,Gs,Done) :-
(
    Tag1 == Tag2,!,solve(Gs),Done=true
;
nonvar(SVs1),SVs1 =..[Type1|_],
extensional(Type1),
nonvar(SVs2),SVs2 =..[Type2|_],
extensional(Type2),
(
Tag1 = Tag2, %% need to nest the Done=true for the "undefined index"
%% example of the Japanese grammar (always still waiting)
iso(SVs1,SVs2,[Done=true|Gs]),!
;
Done=fail, \+ (iso(SVs1,SVs2,[])), solve(Gs))).



delay_iso(Tag1-SVs1,Tag2-SVs2,Gs,Done) :-
 %% if a type of one of the feature structures is not a
 %% maximal subtype, wait to see if it becomes
 %% more specific.
 %% if it does, then retry the isomorphism.
 (var(SVs1),!,when(nonvar(Tag1),iso(Tag1-SVs1,Tag2-SVs2,Gs,Done))
 ;
var(SVs2),!,
when(nonvar(Tag2),iso(Tag1-SVs1,Tag2-SVs2,Gs,Done))
 ;
SVs1 =..[Type1|_],
\+extensional(Type1),!,
%% First check Type against FS to see if they are unifiable
 %% when that succeeds, then wait if necessary on the type

 satisfies(Type1,Tag2-SVs2,[]),
 when(nonvar(Tag1),iso(Tag1-SVs1,Tag2-SVs2,Gs,Done))
 ;
SVs2 =..[Type2|_],
\+extensional(Type2),!,
```

```
satisfies(Type2,Tag1-SVs1,[]),
when(nonvar(Tag2),iso(Tag1-SVs1,Tag2-SVs2,Gs,Done))
 )
 .

%%iso(_FS1,_FS2,_Gs,fail).

% ------------------------------------------------------------

% iso/3 (SVs1:<feature values>,SVs2<feature values>,Gs<guarded goals>) mh(0)

% recursive top down search to see whether sorted list of fea-
ture values SVs1
% is isomorphic with SVs2.  Guarded goals are passed down from iso/4.

% ------------------------------------------------------------


iso(_,_,_) if_h [fail] :-
  \+extensional(_).

iso(SVs1,SVs2,Gs) if_h [solve(Isos)] :-
  extensional(Sort),
  approps(Sort,FRs),
  length(FRs,N),
  functor(SVs1,Sort,N),
  functor(SVs2,Sort,N),
  new_isos(N,SVs1,SVs2,Gs,Isos).
```

Appendix B

HPSG phrase structure schemata

Pollard & Sag (1994: Appendix A.3)

Schema 1 (Head-Subject Schema)
The SYNSEM:LOCAL:CATEGORY:SUBCAT value is ⟨ ⟩, and the DAUGHTERS value is the object of sort **head-comp-struc** whose HEAD-DAUGHTER value is a phrase whose SYNSEM:NONLOCAL:TO-BIND:SLASH value is { }, and whose COMPLEMENT-DAUGHTERS value is a list of length one.

Schema 2 (Head-Complement Schema)
The SYNSEM:LOCAL:CATEGORY:SUBCAT value is a list of length one, and the DAUGHTERS value is the object of sort **head-comp-struc** whose HEAD-DAUGHTER value is a word.

Schema 3 (Head-Subject-Complement Schema)
The SYNSEM:LOCAL:CATEGORY:SUBCAT value is ⟨ ⟩, and the DAUGHTERS value is the object of sort **head-comp-struc** whose HEAD-DAUGHTER value is a word.

Schema 4 (Head-Marker Schema)
The DAUGHTERS value is an object of sort **head-marker-struc** whose HEAD-DAUGHTER:SYNSEM:NONLOCAL:TO-BIND:SLASH value is { }, and whose MARKER-DAUGHTER:SYNSEM:LOCAL:CATEGORY:HEAD value is of sort **marker**.

Schema 5 (Head-Adjunct Schema)
The DAUGHTERS value is an object of sort **head-adjunct-struc** whose

HEAD-DAUGHTER:SYNSEM value is token-identical to its ADJUNCT-DAUGHTER:SYNSEM:LOCAL:CATEGORY:HEAD:MOD value and whose HEAD-DAUGHTERS:SYNSEM:NONLOCAL:TO-BIND:SLASH value is $\{\ \}$.

Schema 6 (Head-Filler Schema)

The DAUGHTERS value is an object of sort **head-filler-struc** whose HEAD-DAUGHTER:SYNSEM:LOCAL:CATEGORY value satisfies the description $\left[\text{HEAD:} \begin{bmatrix} \textbf{verb} \\ \text{VFORM:}\textbf{finite} \\ \text{SUBCAT:}\langle\ \rangle \end{bmatrix}\right]$, whose HEAD-DAUGHTER:SYNSEM:NONLOCAL:INHERITED:SLASH value contains an element token-identical to the FILLER-DAUGHTER:SYNSEM:LOCAL value, and whose HEAD-DAUGHTER:SYNSEM:NONLOCAL:TO-BIND:SLASH value contains only that element.

Appendix C

Compilation of phrase structure rules

C.1 Chart Parser

From Carpenter & Penn (1994).

```
rule(Tag,SVs,Left,Right,N) if_h SubGoals :-
  (RuleName rule (Mother ===> Daughters)),
  compile_dtrs(Daughters,Tag,SVs,Left,Right,N,SubGoals,[],Mother,RuleName).

compile_dtrs((cat> Dtr,Rest),Tag,SVs,Left,Right,N,PGoals,PGoalsRest,
             Mother,RuleName):-
  !, compile_desc(Dtr,Tag-SVs,PGoals,PGoalsMid),
  compile_dtrs_rest(Rest,Left,Right,PGoalsMid,PGoalsRest,Mother,
                    [N|DtrsRest],DtrsRest,RuleName).

compile_dtrs_rest((cat> Dtr,Rest),Left,Right,
                  [edge(N,Right,NewRight,Tag,SVs,_,_)|PGoals],
                  PGoalsRest,Mother,PrevDtrs,[N|DtrsRest],RuleName):-

  !,compile_desc(Dtr,Tag-SVs,PGoals,PGoalsMid),
  compile_dtrs_rest(Rest,Left,NewRight,PGoalsMid,PGoalsRest,Mother,
                    PrevDtrs,DtrsRest,RuleName).

 compile_dtrs_rest((cats> Dtrs,Rest),Left,Right,PGoals,PGoalsRest,
```

```
                    Mother,PrevDtrs,DtrsRest,RuleName) :-
   !,compile_desc(Dtrs,Tag-bot,PGoals,
                                  [deref(Tag,bot,_,SVs),
                                   SVs =.. [Sort|Vs],
      match_list_rest(Sort,Vs,Right,NewRight,DtrsRest,DtrsRest2)|
                                                    PGoalsMid]),
   compile_dtrs_rest(Rest,Left,NewRight,PGoalsMid,PGoalsRest,Mother,
                     PrevDtrs,DtrsRest2,RuleName).

compile_dtrs_rest((goal> Goal,Rest),Left,Right,PGoals,PGoalsRest,
                  Mother,PrevDtrs,DtrsRest,RuleName):-
  !, compile_body(Goal,[],[],PGoalsBody),
  conc(PGoalsBody,PGoalsMid,PGoals),
  compile_dtrs_rest(Rest,Left,Right,PGoalsMid,PGoalsRest,Mother,
                    PrevDtrs,DtrsRest,RuleName).

compile_dtrs_rest((cat> Dtr),Left,Right,
                  [edge(N,Right,NewRight,Tag,SVs,_,_)|PGoals],
                  PGoalsRest,Mother,PrevDtrs,[N],RuleName):-

  !,compile_desc(Dtr,Tag-SVs,PGoals,PGoalsMid),
  compile_desc(Mother,Tag2-bot,PGoalsMid,
               [add_edge_deref(Left,NewRight,Tag2,bot,
                               PrevDtrs,RuleName)|PGoalsRest]).

 compile_dtrs_rest((cats> Dtrs),Left,Right,PGoals,PGoalsRest,
                    Mother,PrevDtrs,DtrsRest,RuleName) :-
   !,compile_desc(Dtrs,Tag-bot,PGoals,
                                  [deref(Tag,bot,_,SVs),
                                   SVs =.. [Sort|Vs],
     match_list_rest(Sort,Vs,Right,NewRight,DtrsRest,[])|
                                                    PGoalsMid]),
   compile_desc(Mother,Tag2-bot,PGoalsMid,
                [add_edge_deref(Left,NewRight,Tag2,bot,
                                PrevDtrs,RuleName)|PGoalsRest]).

compile_dtrs_rest((goal> Goal),Left,Right,PGoals,PGoalsRest,Mother,
                  PrevDtrs,[],RuleName):-
  !, compile_body(Goal,[],[],PGoalsBody),
  conc(PGoalsBody,PGoalsMid,PGoals),
```

```
  compile_desc(Mother,Tag2-bot,PGoalsMid,
               [add_edge_deref(Left,Right,Tag2,bot,
                               PrevDtrs,RuleName)|PGoalsRest]).
```

C.2 Left Corner Parser

```
rule(Tag,SVs,WsIn,WsOut,MomTag-MomSVs,RuleName) if_h SubGoals :-

   (RuleName rule (Mother ===> Daughters)),
   compile_dtrs(Daughters,Tag,SVs,WsIn,WsOut,MomTag-MomSVs,SubGoals,
                [],Mother,RuleName).

compile_dtrs((cat> Dtr,Rest),Tag,SVs,WsIn,WsOut,MomTag-MomSVs,PGoals,
             PGoalsRest,Mother,RuleName):-
  !, compile_desc(Dtr,Tag-SVs,PGoals,PGoalsMid),
  compile_desc(Mother,MomTag-MomSVs,PGoalsMid,PGoalsMore),
  compile_dtrs_rest(Rest,WsIn,WsOut,MomTag-MomSVs,PGoalsMore,PGoalsRest,
                    Mother,RuleName).

%% the lc step is here:

compile_dtrs_rest((cat> Dtr,Rest),WsIn,WsOut,MomTag-MomSVs,
                    [write('**in rule: '),write(RuleName),nl,nl,
     lc_rec(Tag-SVs,WsIn,WsMid)|PGoals],
                    PGoalsRest,Mother,RuleName):-
     compile_desc(Dtr,Tag-SVs,PGoals,PGoalsMid),
     compile_desc(Mother,MomTag-MomSVs,PGoalsMid,PGoalsMore),
     !, compile_dtrs_rest(Rest,WsMid,WsOut,MomTag-MomSVs,PGoalsMore,
                          PGoalsRest,Mother,RuleName).

compile_dtrs_rest((goal> Goal,Rest),WsIn,WsOut,MomTag-MomSVs,PGoals,
                  PGoalsRest,Mother,RuleName):-
  !, compile_body(Goal,[],[],PGoalsBody),
  conc(PGoalsBody,PGoalsMid,PGoals),
  compile_dtrs_rest(Rest,WsIn,WsOut,MomTag-MomSVs,PGoalsMid,PGoalsRest,
                    Mother,RuleName).

compile_dtrs_rest((cat> Dtr),WsIn,WsOut,MomTag-MomSVs,
[write('**in rule: '),write(RuleName),nl,nl,
```

```
 lc_rec(Tag-SVs,WsIn,WsOut)|PGoals],
 PGoalsRest,Mother,RuleName):-
 compile_desc(Dtr,Tag-SVs,PGoals,PGoalsMid),
 !, compile_desc(Mother,MomTag-MomSVs,PGoalsMid,PGoalsRest).

compile_dtrs_rest((goal> Goal),WsIn,WsIn,_MomTag-_MomSVs,PGoals,
                  PGoalsRest,_Mother,RuleName):-
  !, compile_body(Goal,[],PGoalsRest,PGoals).
```

Appendix D

Binding rules as unguarded constraints

From Callison-Burch & Guffey (1999). Constrasts with the binding approach in chapter 6.

```
:begin :instance.
ana_agr-principle-e := word+arg_real+ana_agr &
[ SYN #syn,
  SEM #sem,
  ARG-ST #arg-st & < >,

  ARGS < word+arg_real & [ SYN #syn,
   SEM #sem,
   ARG-ST #arg-st ] > ].
:end :instance.


:begin :instance.
ana_agr-principle-x := word+arg_real+ana_agr &
[ SYN #syn,
  SEM #sem,
  ARG-ST #arg-st & < [ ] >,

  ARGS < word+arg_real & [ SYN #syn,
   SEM #sem,
   ARG-ST #arg-st ] > ].
:end :instance.
```

```
:begin :instance.
ana_agr-principle-x+ := word+arg_real+ana_agr &
[ SYN #syn,
  SEM #sem,
  ARG-ST #arg-st & < [SYN [HEAD [AGR #agr]],
      SEM [INDEX #ind]],  [SYN [HEAD [ANA true,
      AGR #agr]],
   SEM [INDEX #ind]] >,

  ARGS < word+arg_real & [ SYN #syn,
   SEM #sem,
   ARG-ST #arg-st ] > ].
:end :instance.



:begin :instance.
ana_agr-principle-x- := word+arg_real+ana_agr &
[ SYN #syn,
  SEM #sem,
  ARG-ST #arg-st & < [ ],  [SYN [HEAD [ANA false]]] >,

  ARGS < word+arg_real & [ SYN #syn,
   SEM #sem,
   ARG-ST #arg-st ] > ].
:end :instance.



:begin :instance.
ana_agr-principle-x-+ := word+arg_real+ana_agr &
[ SYN #syn,
  SEM #sem,
  ARG-ST #arg-st & < [SYN [HEAD [AGR #agr]],
      SEM [INDEX #ind]],

     [SYN [HEAD [ANA false]]],
```

```
     [SYN [HEAD [ANA true,
 AGR #agr]],
      SEM [INDEX #ind]] >,

  ARGS < word+arg_real & [ SYN #syn,
   SEM #sem,
   ARG-ST #arg-st ] > ].
:end :instance.



:begin :instance.
ana_agr-principle-x+- := word+arg_real+ana_agr &
[ SYN #syn,
  SEM #sem,
  ARG-ST #arg-st & < [SYN [HEAD [AGR #agr]],
      SEM [INDEX #ind]],

                    [SYN [HEAD [ ANA true,
          AGR #agr]],
      SEM [INDEX #ind]],

     [SYN [HEAD [ANA false]]] >,

  ARGS < word+arg_real & [ SYN #syn,
   SEM #sem,
   ARG-ST #arg-st ] > ].
:end :instance.



:begin :instance.
ana_agr-principle-x-- := word+arg_real+ana_agr &
[ SYN #syn,
  SEM #sem,
  ARG-ST #arg-st & < [ ],  [SYN [HEAD [ANA false]]],
     [SYN [HEAD [ANA false]]] >,
```

```
  ARGS < word+arg_real & [ SYN #syn,
   SEM #sem,
   ARG-ST #arg-st ] > ].
:end :instance.



:begin :instance.
ana_agr-principle-xi+ := word+arg_real+ana_agr &
[ SYN #syn,
  SEM #sem,
  ARG-ST #arg-st & < [ ],  [SYN [HEAD [ANA false,
       AGR #agr]],
    SEM [INDEX #ind]],       [SYN [HEAD [ANA true,
 AGR #agr]],
     SEM [INDEX #ind]] >,

  ARGS < word+arg_real & [ SYN #syn,
   SEM #sem,
   ARG-ST #arg-st ] > ].
:end :instance.


:begin :instance.
ana_agr-principle-x++ := word+arg_real+ana_agr &
[ SYN #syn,
  SEM #sem,
  ARG-ST #arg-st & < [SYN [HEAD [AGR #agr]],
      SEM [INDEX #ind]],      [SYN [HEAD [ANA true,
   AGR #agr]],
       SEM [INDEX #ind]],

     [SYN [HEAD [ANA true,
 AGR #agr]],
      SEM [INDEX #ind]] >,

  ARGS < word+arg_real & [ SYN #syn,
   SEM #sem,
   ARG-ST #arg-st ] > ].
:end :instance.
```

Appendix E

Partial verb phrase lexical rule without guarding

```
pvp lex_rule
     (word,
      synsem:(loc:(cat:(head:(Head,verb,
                                    vform:bse,
                                    aux:plus,
                                    flip:minus),
                              subj:[Subj],
                              comps:PVPComps,
                              spr:Spr,
                              marking:Mark),
   cont:Cont,
                     conx:Conx),
                  non_loc:(inherited:(slash:e_set,
                                      rel:Rel,
                                      que:Que),
                           to_bind:ToBind)),
      qstore:QStore,
      qretr:QRetr)
     **>
     (word,
      subcat:[Subj|SubcatComps],
      synsem:(HeadSyn,(loc:(cat:(head:Head,
                          subj:[Subj],
                          comps:PVPComps,
```

```
                        spr:Spr,
                        marking:Mark),
                   cont:(Cont,nucleus:modal_arg:Prop),

                   conx:Conx),
              non_loc:(inherited:
                        (slash:(elt:(PVP,
                                     cat:(PVPHead,
                                          head:(verb,vform:bse),
                                          lex:minus,
                                          subj:[Subj],
                                          comps:PVPComps),
                                     cont:Prop),
                                elts:e_set),
                        rel:Rel,
                        que:Que)))),
      qstore:QStore,
      qretr:QRetr)

if        (check_phrasal(PVPComps),
           check_nominal(PVPComps),
           append(PVPComps,
                  [(loc:PVP,non_loc:(inherited:(slash:(elt:PVP))))],
                  SubcatComps),
           aux_raising(HeadSyn,PVPComps,Subj))

     morphs
       X becomes X.
```

Bibliography

ABEILLÉ, ANNE, & DANIÈLE GODARD. 1994. The complementation of French auxiliaries. In *West Coast Conference on Formal Linguistics*, volume 13, Stanford University. CSLI Publications/SLA.

ABEILLÉ, ANNE, DANIÈLE GODARD, PHILIP MILLER, & IVAN A. SAG. 1998. French bounded dependencies. In *Romance in HPSG*, ed. by Luca Dini & Sergio Balari, number 75 in CSLI Lecture Notes. Cambridge University Press. Distributed for CSLI Publications.

AÏT-KACI, H., & R. NASR. 1986. LOGIN: A logic programming language with built-in inheritance. *Journal of Logic Programming* 3.185–215.

AÏT-KACI, HASSAN, 1984. *A lattice-theoretic approach to computation based on a calculus of partially ordered types*. University of Pennsylvania dissertation.

AÏT-KACI, HASSAN, & ANDREAS PODELSKI. 1994. Functions as passive constraints in LIFE. *ACM Transactions on Programming Languages and Systems* 16.1279–1318.

AJDUKIEWICZ, K. 1935. Die syntaktische Konnexität. *Studia Philosophica* 1.1–27.

ALSINA, ALEX. 1992. On the argument structure of causatives. *Linguistic Inquiry* 23.517–555.

BAKER, KATHRYN L. 1994. An Extended Account of "Modal Flip" and Partial Verb Phrase Fronting in German. Laboratory for Computational Linguistics Report CMU-LCL-94-4, Laboratory for Computational Linguistics, Carnegie Mellon University.

—— 1999. 'Modal flip' and partial verb phrase fronting. In *Studies in Contemporary Phrase Structure Grammar*, ed. by Robert Levine & Georgia Green, 161–198. Cambridge, U.K.: Cambridge University Press.

BAKER, MARK. 1988. *Incorporation: A Theory of Grammatical Function Changing*. Chicago, Illinois: University of Chicago Press.

BAR-HILLEL, Y. 1953. A quasi-arithmetical notation for syntactic description. *Language* 29.47–58.

BAYER, SAM, & MARK JOHNSON. 1995. Features and agreement. In *Proceedings of the 33rd Annual Meeting of the ACL*, Cambridge, Mass. ACL.

BOUMA, G., & G. VAN NOORD. 1994. Constraint-based categorial grammar. In *Proceedings of the Thirty-Second Annual Meeting of the ACL*, Las Cruces, New Mexico. Association for Computational Linguistics.

BOUMA, GOSSE, ROBERT MALOUF, & IVAN A. SAG, 2001. Satisfying constraints on extraction and adjunction.

BRACHMAN, R. J., & J. G. SCHMOLZE. 1985. An overview of the KL-ONE knowledge representation system. *Cognitive Science* 9.171–216.

BRESNAN, JOAN. 1982a. Control and complementation. In *The Mental Representation of Grammatical Relations*, ed. by Joan Bresnan, 282–390. Cambridge, Mass: The MIT Press.

—— (ed.) 1982b. *The Mental Representation of Grammatical Models*. Cambridge, Mass: The MIT Press.

BRESNAN, JOAN, & SAM MCHOMBO. 1995. The lexical integrity principle: Evidence from Bantu. *Natural Language and Linguistic Theory* 13.181–254.

CALLISON-BURCH, CHRIS, & SCOTT GUFFEY, 1999. Implementation of (Sag & Wasow 1999). Distributed with the LKB grammar system (Copestake *et al.* 1999).

CARPENTER, BOB. 1991. The generative power of categorial grammars and head-driven phrase structure grammars with lexical rules. *Computational Linguistics* 17.301–314.

——. 1992. *The Logic of Typed Feature Structures with Applications to Unification-based Grammars, Logic Programming and Constraint Resolution*, volume 32 of *Cambridge Tracts in Theoretical Computer Science*. New York: Cambridge University Press.

——. 1994. Quantification and scoping: A deductive account. In *West Coast Conference on Formal Linguistics*, volume 13, 533–548. Stanford University: CSLI Publications/SLA.

CARPENTER, BOB, & GERALD PENN. 1994. ALE: The Attribute Logic Engine: User's guide. Laboratory for Computational Linguistics Report CMU-LCL-94, Laboratory for Computational Linguistics, Carnegie Mellon University. Version 2.0.1.

——. 1996. Compiling typed attribute-value logic grammars. In *Recent Advances in Parsing Technologies*, ed. by Harry Bunt & Masaru Tomita. Kluwer.

CHOMSKY, NOAM. 1973. Conditions on transformations. In *A Festschrift for Morris Halle*, ed. by Stephen Anderson & Paul Kiparsky. New York: Holt, Rinehart and Winston.

——. 1981. *Lectures on Government and Binding*. Dordrecht: Foris.

——. 1986. *Knowledge of Language*. New York: Praeger.

CHURCH, KENNETH W. 1988. A stochastic parts program and noun phrase parser for unrestricted text. In *Second Conference on Applied Natural Language Processing*, 136–143, Austin, Texas.

CLARK, K. L. 1978. Negation as failure. In *Logic and Databases*, 293–322. New York: Plenum Press.

COLLINS, MICHAEL JOHN. 1996. A new statistical parser based on bigram lexical dependencies. In *Proceedings of the 34th Annual Meeting*, Santa Cruz, CA. Association for Computational Linguistics.

COLMERAUER, A., H. KANOUI, R. PASERO, & P. ROUSSEL. 1973. Un systeme de communication homme-machine en Francais. Technical report, Research Report, Groupe Intelligence Artificialle, Groupe Intelligence Artificielle, Universite Aix-Marseille II.

COLMERAUER, ALAIN. 1982. PROLOG-II, manual de référence et modèle théorique. Rapport technique, Groupe d'Intelligence Artificielle, Université de Marseille.

——. 1990. An introduction to PROLOG-III. *Communications of the ACM* 33.69–90.

COMRIE, BERNARD. 1981. *Language Universals and Linguistic Typology: Syntax and Morphology*. The University of Chicago Press.

COOPER, ROBIN, 1975. *Montague's Semantic Theory and Transformational Syntax*. University of Massachusetts at Amherst dissertation.

——. 1983. *Quantification and Syntactic Theory*. Dordrecht: Reidel.

COPESTAKE, ANN. 1992. The ACQUILEX LKB: Representation issues in semi-automatic acquisition of large lexicons. In *Proceedings of the 3rd Conference on Applied Natural Language Processing (ANLP-92)*, 88–96. Trento, Italy: Association for Computational Linguistics.

COPESTAKE, ANN, JOHN CARROLL, ROB MALOUF, STEPHAN OEPEN, & OTHERS, 1999. *The (new) LKB system*. CSLI, Stanford University, 5.2 edition. A grammar and lexicon development environment for use with constraint-based linguistic formalisms.

COPESTAKE, ANN, & DAN FLICKINGER. 2000. An open-source grammar development environment and broad-coverage English grammar using HPSG. In *Proceedings of the Second conference on Language Resources and Evaluation (LREC-2000)*, Athens, Greece.

DÖRRE, JOCHEN, & MICHAEL DORNA. 1993. CUF – a formalism for linguistic knowledge representation. ESPRIT Basic Research Action 3175, ESPRIT, Institut für maschinelle Sprachverarbeitung, Universität Stuttgart.

DÖRRE, JOCHEN, & SURESH MANANDHAR. 1997. On constraint-based lambek calculi. In *Specifying Syntactic Structures*, ed. by Blackburn & de Rijke, Studies in Logic, Language and Information. Stanford: CSLI Publications.

DUCHIER, DENYS. 1999. Axiomatizing dependency parsing using set constraints. In *Sixth Meeting on Mathematics of Language*, Orlando, Florida.

DUCHIER, DENYS, CLAIRE GARDENT, & JOACHIM NIEHREN, 1998. Concurrent constraint programming in Oz for natural language processing. Lecture notes, Universität des Saarlandes. Version 1.2.0.

EARLEY, J. 1970. An efficient context-free parsing algorithm. *Communications of the ACM* 13.94–102.

ELHADAD, MICHAEL, KATHLEEN MCKEOWN, & JACQUES ROBIN. 1997. Floating constraints in lexical choice. *Computational Linguistics* .

ELHADAD, MICHAEL, & JACQUES ROBIN. 1992. Controlling content realization with functional unification grammars. In *Aspects of Automated Natural Language Generation*, ed. by R. Dale, E. Hovy, D. Rösner, & O. Stock, 89–104. Springer Verlag.

EMELE, M.C., & R. ZAJAC. 1990. Typed unification grammars. In *Proceedings of the 13th International Conference on Computational Linguistics*, Helsinki.

EMONDS, JOE. 1978. The verbal complex V' V in French. *Linguistic Inquiry* 9.151–175.

ERBACH, G., M. VAN DER KRAAN, S. MANANDHAR, H. RUESSINK, W. SKUT, & C. THIERSCH. 1995. Extending unification formalisms. In *Proceedings of the 2nd Language Engineering Convention*, London.

FLICKINGER, DANIEL, 1987. *Lexical Rules in the Hierarchical Lexicon*. Stanford University dissertation.

FRADIN, BERNARD, 1993. *Organisation de l'information lexical: l'interface morphologie-syntaxe dans le domaine verbal*. U. Paris 8 dissertation.

FRANZ, ALEX. 1990. A parser for HPSG. Laboratory for Computational Linguistics 90-3, Carnegie Mellon University.

FRÜHWIRTH, THOM, & SLIM ABDENNADHER. 1997. *Constraint-Programmierung*. Berlin: Springer-Verlag.

GAZDAR, GERALD. 1981. Unbounded dependencies and coordinate structure. *Linguistic Inquiry* 12.155–184.

GAZDAR, GERALD, EWAN KLEIN, GEOFFREY K. PULLUM, & IVAN A. SAG. 1985. *Generalized Phrase Structure Grammar*. Oxford: Basil Blackwell.

GAZDAR, GERALD, GEOFFREY PULLUM, & IVAN A. SAG. 1982. Auxiliaries and related phenomena in a restrictive theory of grammar. *Language* 58.591–638.

GÖTZ, THILO, & DETMAR MEURERS. 1997. Interleaving universal principles and relational constraints over typed feature logic. In *Proceedings of the 35th Meeting of the ACL and 8th Conference of the EACL*, Madrid, Spain. ACL.

GÖTZ, THILO, DETMAR MEURERS, & DALE GERDEMANN, 1997. *The ConTroll Manual*
(ConTroll v.1.0β, XTroll v.5.0β). Seminar for Sprachwissenschaft, Universität Tübingen.

GÖTZ, THILO, & WALT DETMAR MEURERS. 1998. The importance of being lazy - Using lazy evaluation to process queries to HPSG grammars. In *Lexical and Constructional Aspects of Linguistic Explanation*, ed. by Gert Webelhuth, Jean-Pierre Koenig, & Andreas Kathol. Stanford: CSLI.

GUNJI, TAKAO. 1987. *Japanese Phrase Structure Grammar: A Unification-Based Approach*. Dordrecht: Reidel.

HARIDI, S., & S. JANSON. 1990. Kernel Andorra Prolog and its computation model. In *Proceedings of the seventh international conference on logic programming*, ed. by D. H. D. Warren & P. Szeredi. MIT Press.

HINRICHS, ERHARD, & TSUNEKO NAKAZAWA. 1989. Flipped out: Aux in german. In *Papers from the 25th Regional Meeting of the Chicago Linguistic Society*, volume 25, 193–202, Chicago, Illinois. Chicago Linguistic Society, Chicago Linguistic Society.

——. 1994. Linearizing AUXs in German verbal complexes. In *German in Head-Driven Phrase Structure Grammar*, ed. by John Nerbonne, Klaus Netter, & Carl Pollard, number 46 in Lecture Notes, 11–37. Stanford University: CSLI Publications.

HÖHFELD, MARKUS, & GERT SMOLKA. 1988. Definite relations over constraint languages. LILOG Report 53, LILOG, IBM Deutschland GMbH, Stuttgart.

HÖHLE, TILMANN, 1995. CELR and variable argument raising. Presented at the HPSG Workshop, Tübingen, June 22, 1995. Unpublished ms.

HUDSON, RICHARD A. 1990. *English Word Grammar*. Oxford, UK: B. Blackwell.

INGRIA, ROBERT. 1990. The limits of unification. In *Meeting of the Association for Computational Linguistics*, 194–204.

JACKENDOFF, RAY. 1977. $\bar{X}$ *Syntax: A Study of Phrase Structure*. Cambridge, Mass: MIT Press.

JAFFAR, J., & S. MICHAYLOV. 1987. Methodology and implementation of a clp system. In *Proceedings of the Fourth International Conference on Logic Programming*, Melbourne.

JAFFAR, JOXAN, & JEAN-LOUIS LASSEZ. 1987. Constraint logic programming. In *Proceedings of the 14th ACM Symposium on Principles of Programming Languages*, 111–119, New York. Association for Computing Machinery.

JOHNSON, MARK. 1986. A GPSG account of VP structure in German. *Linguistics* 24.871–882.

JOHNSON, MARK, & JOCHEN DÖRRE. 1995. Memoization and coroutined constraints. In *Proceedings of the Thirty-Third Annual Meeting of the ACL*, Cambridge, Mass. Association for Computational Linguistics.

KAMPRATH, CHRISTINE, ERIC ADOLPHSON, TERUKO MITAMURA, & ERIC NYBERG. 1998. Controlled language for multilingual document production: Experience with Caterpillar Technical English. In *Proceedings of the Second International Workshop on Controlled Language Applications*, Pittsburgh, PA.

KAPLAN, RONALD, & JOAN BRESNAN. 1982. Lexical-functional grammar: A formal system for grammatical representation. In *The Mental Representation of Grammatical Relations*, ed. by Joan Bresnan, 173–281. Cambridge, Mass: The MIT Press.

KASPER, ROBERT T., & WILLIAM C. ROUNDS. 1986. A logical semantics for feature structures. In *Proceedings of the Twenty-Fourth Annual Meeting of the ACL*, 235–242. Association for Computational Linguistics.

—— 1990. The logic of unification in grammar. *Linguistics and Philosophy* 13.35–58.

KATHOL, ANDREAS. 1994. Passives without Lexical Rules. In *German in Head-Driven Phrase Structure Grammar*, ed. by John Nerbonne, Klaus Netter, & Carl Pollard, number 46 in Lecture Notes, 237–272. Stanford University: CSLI Publications.

——, 1995. *Linearization-Based German Syntax*. Ohio State University dissertation.

KATHOL, ANDREAS, & CARL POLLARD. 1995. Extraposition via Complex Domain Formation. In *Proceedings of the Thirty-Third Annual Meeting of the ACL*, Cambridge, Mass. Association for Computational Linguistics.

KAY, MARTIN. 1979. Functional grammar. In *Proceedings of the 5th annual meeting*, Berkeley, CA. Berkeley Linguistic Society.

——. 1983. Unification grammar. Technical report, Xerox Palo Alto Research Center.

——. 1985. Parsing in functional unification grammar. In *Natural Language Parsing: Psychological, Computational and Theoretical Perspectives*, chapter 7, 251–278. Cambridge, England: Cambridge University Press.

KEENAN, EDWARD, & BERNARD COMRIE. 1977. Noun phrase accessibility and universal grammar. *Linguistic Inquiry* 8.63–99.

KING, P., 1989. *A logical formalism for Head-Driven Phrase Structure Grammar*. Manchester, England: University of Manchester dissertation.

KITAGAWA, YOSHIHISA, 1986. *Subjects in Japanese and English*. Amherst: University of Massachusetts dissertation. Published by Garland, New York, 1994.

KOLLER, ALEXANDER, & JOACHIM NIEHREN. 2000. Constraint programming in computational linguistics. In *Proceedings of the eighth CSLI Workshop on*

Logic Language and Computation, ed. by D. Barker-Plummer, D. Beaver, J. van Benthem, & P. Scotto di Luzio. CSLI Press. To appear.

KOWALSKI, ROBERT. 1992. Logic programming. *Encyclopedia of Artificial Intelligence* 1.778–783. 2nd edition.

KURODA, SIGE-YUKI, 1965. *Generative Grammatical Studies in the Japanese Language*. Massachusetts Institute of Technology dissertation.

LAMBEK, JOACHIM. 1958. The mathematics of sentence structure. *American Mathematical Monthly* 65.

LEVINE, ROBERT D., & GEORGIA M. GREEN (eds.) 1999. *Studies in Contemporary Phrase Structure Grammar*. Cambridge University Press.

LI, Y. 1990. X^0 binding and verb incorporation. *Linguistic Inquiry* 399–426.

LLOYD, J. W. 1984. *Foundations of Logic Programming*. Springer Verlag.

MANNING, CHRISTOPHER, 1994. *Ergativity: Argument Structure and Grammatical Relations*. Stanford University dissertation.

MANNING, CHRISTOPHER, & IVAN SAG. 1998. Argument structure, valence, and binding. *Nordic Journal of Linguistics* 107–144.

MANNING, CHRISTOPHER, IVAN SAG, & MASAYO IIDA. 1999. The lexical integrity of Japanese causatives. In *Studies in Contemporary Phrase Structure Grammar*, ed. by Robert Levine & Georgia Green. Cambridge, U.K.: Cambridge University Press.

MANNING, CHRISTOPHER D., & HINRICH SCHÜTZE. 1999. *Foundations of Statistical Natural Language Processing*. MIT Press.

MARANTZ, ALEC P. 1984. *On the Nature of Grammatical Relations*. Cambridge, MA: MIT Press. Revision of thesis (Ph.D.) – MIT, 1981.

MARCUS, M., S. SANTORINI, & M. MARCINKIEWICZ. 1993. Building a large annotated corpus of English: the Penn Treebank. *Computational Linguistics* 19.313–330.

MARRIOTT, KIM, & PETER J. STUCKEY. 1998. *Programming with Constraints: An Introduction*. Cambridge, Mass.: The MIT Press.

MEURERS, DETMAR, 1994. On implementing an HPSG theory: Aspects of the logical architecture, the formalization and the implementation of head-driven phrase structure grammars. Master's thesis, Universitt Tbingen. Appeared as Part 2 of SFB Report 58.

MEURERS, WALT DETMAR, & GUIDO MINNEN. 1995. A computational treatment of HPSG lexical rules as covariation in lexical entries. In *Proceedings of the 5th International Workshop on Natural Language Understanding and Logic Programming*, Lisbon, Portugal.

——. 1996. Off-line constraint propagation for efficient hpsg processing. In *Proceedings of the HPSG/TALN Conference*, Marseille, France.

MILLER, PHILIP H. 1992. *Clitics and Constituents in Phrase Structure Grammar*. New York: Garland. Published version of 1991 Doctoral dissertation, University of Utrecht, The Netherlands.

MILLER, PHILIP H., & IVAN A. SAG, 1995. French clitic movement without clitics or movement. Unpublished ms., University of Lille and Stanford University.

MITAMURA, TERUKO, & ERIC NYBERG. 2000. The KANTOO machine translation environment. In *Proceedings of the AMTA*. Association for Machine Translation in the Americas.

MIYAGAWA, SHIGERU. 1980. Complex verbs and the lexicon. In *Coyote Working Papers 1*. Tucson: University of Arizona.

MONACHESI, PAOLA. 1993a. Object clitics and clitic climbing in Italian HPSG grammar. In *Proceedings of the Sixth Conference of the European Chapter of the Association for Computational Linguistics*, Utrecht. European Chapter, Association for Computational Linguistics, European Chapter, Association for Computational Linguistics.

——. 1993b. Restructuring verbs in Italian HPSG grammar. In *Proceedings of the 29th Regional Meeting of the Chicago LInguistic Society*, volume 29, Chicago, Illinois. Chicago Linguistic Society, Chicago Linguistic Society.

——. 1998. Italian restructuring verbs: A lexical analysis. In *Syntax and Semantics: Complex Predicates in Nonderivational Syntax*, ed. by Peter W. Culicover, volume 30 of *Syntax and Semantics*. San Diego: Academic Press.

MONTAGUE, RICHARD. 1974. The proper treatment of quantification in ordinary english. In *Formal Philosophy*, ed. by R. H. Thomason, 188–221. New Haven, Conn.: Yale University Press.

MOSHIER, M. DREW, 1988. *Extensions to Unification Grammars for the Description of Programming Languages*. Ann Arbor: University of Michigan dissertation.

MOSHIER, M. DREW, & WILLIAM C. ROUNDS. 1987. A logic for partially specified data structures. In *Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages*, 156–167, Munich, Germany.

NAISH, L., 1985. *Negation and Control in Prolog*. University of Melbourne dissertation.

NERBONNE, JOHN. 1994. Partial verb phrases and spurious ambiguities. In *German in Head-Driven Phrase Structure Grammar*, ed. by John Nerbonne, Klaus Netter, & Carl Pollard, number 46 in Lecture Notes, 109–150. Stanford University: CSLI Publications.

VAN NOORD, GERTJAN, & GOSSE BOUMA. 1994. The scope of adjuncts and the processing of lexical rules. In *Proceedings of Coling 94*, 250–256, Kyoto.

——, 1995. Dutch verb clustering without verb clusters. Unpubl. ms., Alfa-informatica, Rijksuniversiteit Groningen.

NYBERG, ERIC, & TERUKO MITAMURA. 1992. The KANT system: Fast, accurate, high-quality translation in practical domains. In *Proceedings of Coling 92*. International Conference on Computational Linguistics.

PENN, GERALD, 1993. A utility for typed feature structure-based grammatical theories. Master's thesis, Carnegie Mellon University, Pittsburgh.

——. 1999. A generalized-domain-based approach to Serbo-Croatian second position clitic placement. In *Constraints and Resources in Natural Language Syntax and Semantics*, ed. by Gosse Bouma, Erhard Hinrichs, Geert-Jan M. Kruijff, & Richard Oehrle, Studies in Constraint-Based Lexicalism. Stanford University: CSLI Publications.

——. 2000. Applying constraint handling rules to HPSG. In *Proceedings of the First International Conference on Computational Logic (CL2000)*, Imperial College, UK. Workshop on Rule-Based Constraint Reasoning and Programming.

PEREIRA, F. C. N., & D. H. D. WARREN. 1980. Definite clause grammars for language analysis: A survey of the formalism and a comparison with augmented transition networks. *Artificial Intelligence* 13.231–278.

PICKERING, MARTIN, & GUY BARRY. 1991. Sentence processing without empty categories. *Language and Cognitive Processes* 229–259.

POLLARD, CARL, & IVAN A. SAG. 1987. *Information-based Syntax and Semantics, Vol. 1*. Number 13 in Lecture Notes. Stanford University: CSLI Publications. Distributed by University of Chicago Press.

——. 1992. Anaphors in English and the scope of binding theory. *Linguistic Inquiry* 23.261–303.

——. 1994. *Head-Driven Phrase Structure Grammar*. Chicago: University of Chicago Press.

POLLARD, CARL, & EUN JUNG YOO. 1997. A unified theory of scope for quantifiers and *wh*-phrases. *Journal of Linguistics* .

POLLARD, CARL J. 1994. Toward a Unified Account of Passive in German. In *German in Head-Driven Phrase Structure Grammar*, ed. by John Nerbonne, Klaus Netter, & Carl Pollard, number 46 in Lecture Notes, 273–296. Stanford University: CSLI Publications.

POLLOCK, JEAN-YVES. 1989. Verb movement, universal grammar and the structure of IP. *Linguistic Inquiry* 20.365–424.

REAPE, MIKE. 1996. Getting things in order. In *Discontinuous Constituency*, ed. by Arthur Horck & Wietske Sijtsma, 209–254. Berlin: Mouton de Gruyter.

RIEHEMANN, SUSANNE, 1993. Word formation in lexical type hierarchies: A case study of *bar*-adjectives in german. Master's thesis, University of Tübingen. Also published as SfS-Report-02-93, Seminar für Sprachwissenschaft, University of Tübingen.

RIZZI, LUIGI. 1982. *Issues in Italian Syntax*. Dordrecht: Foris.

ROBINSON, J. A. 1965. A machine-oriented logic based on the resolution principle. *Journal of the ACM* 12.23–41.

ROSS, JOHN R., 1967. *Constraints on Variables in Syntax*. Massachusetts Institute of Technology dissertation. Published as *Infinite Syntax!* Norwood, NJ: Ablex, 1986.

SAG, IVAN A. 1997. English relative clause constructions. *Journal of Linguistics* 32.

SAG, IVAN A., & DANIÈLE GODARD. 1993. Extraction of *de*-phrases from the French NP. In *Proceedings of the North East Linguistic Society 24*, Department of Linguistics, University of Massachusetts. GLSA.

SAG, IVAN A., & THOMAS WASOW. 1999. *Syntactic Theory: A Formal Introduction*. CSLI Lecture Notes. Cambridge University Press. Distributed for CSLI Publications.

SARASWAT, VIJAY, 1993. ACM distinguished dissertation series.

SHIBATANI, MASAYOSHI. 1976. Causativization. In *Japanese Generative Grammar*, ed. by Masayoshi Shibatani, Syntax and Semantics, 239–294. Academic Press.

SHIEBER, S. M., H. USZKOREIT, F. C. N. PEREIRA, J. ROBINSON, & M. TYSON. 1983. The formalism and implementation of PATR-II. Technical report, SRI International, Menlo Park, California.

SHIEBER, STUART. 1986. *An Introduction to Unification-Based Approaches to Grammar*. Number 4 in Lecture Notes. Stanford University: Center for the Study of Language and Information.

SICStus, 1995. *SICStus Prolog User's Manual*. The Programming Systems Group, Swedish Institute for Computer Science, Kista, Sweden, 3.0 edition.

SMOLKA, GERT. 1995. The Oz programming model. In *Computer Science Today*, ed. by Jan van Leeuwen, Lecture Notes in Computer Science, vol. 1000, 324–343. Berlin: Springer-Verlag.

SOAMES, SCOTT, & DAVID PERLMUTTER. 1979. *Syntactic Argumentation and the Structure of English*. Berkeley and Los Angeles: University of California Press.

STERLING, LEON, & EHUD SHAPIRO. 1986. *The Art of Prolog*. Cambridge, MA: MIT Press.

USZKOREIT, HANS. 1987. Linear precedence and discontinuous constituents: Complex fronting in German. In *Discontinuous Constituents*, ed. by G.J. Huck & A.E. Ojeda, volume 20 of *Syntax and Semantics*, 406–427. London: Academic Press.

WILLIAMS, EDWIN. 1981. Argument structure and morphology. *The Linguistic Review* 1.81–114.

WOODS, WILLIAM A. 1970. Transition network grammars for natural language analysis. *Communications of the ACM* 13.591–606.