The Generation of Phrase-Structure Representations From Principles

by

David C. LeBlanc

Technical Report 90-38
January, 1991

Department of Computer Science
University of British Columbia
Vancouver, B.C. V6T 1W5

Email: leblanc@cs.ubc.ca

Abstract

Implementations of grammatical theory have traditionally been based upon Context-Free Grammar (CFG) formalisms which all but ignore questions of learnability. Even implementations which are based upon theories of Generative Grammar (GG), a paradigm which is supposedly motivated by learnability, rarely address such questions. In this thesis we will examine a GG theory which has been formulated primarily to address questions of learnability and present an implementation based upon this theory. The theory argues from Chomsky's definition of epistemological priority that principles which match elements and structures from prelinguistic systems with elements and structures in linguistic systems are preferable to those which are defined purely linguistically or non-linguistically. A procedure for constructing phrase-structure representations from prelinguistic relations using principles of node percolation (rather than the traditional $\overline{X}$-theory of GG theories or phrase-structure rules of CFG theories) is presented and this procedure integrated into a left-right, primarily bottom-up parsing mechanism. Specifically, we present a parsing mechanism which derives phrase-structure representations of sentences from Case- and Θ-relations using a small number of Percolation Principles. These Percolation Principles simply determine the categorial features of the dominant node of any two adjacent nodes in a representational tree, doing away with explicit phrase structure rules altogether. The parsing mechanism also instantiates appropriate empty categories using a filler-driven paradigm for leftward argument and non-argument movement. Procedures modelling learnability are not implemented in this work, but the applicability of the presented model to a computational model of language is discussed.

Contents

List of Figures

Acknowledgements

First and foremost, I would like to express my gratitude to my supervisors: Dr. Henry Davis for his constant, invaluable assistance with linguistic theory and guidance throughout the course of this thesis, and Dr. Richard Rosenberg for his constructive criticism and support.

I would like to thank Matthew Crocker for introducing me to the field of computational linguistics and Dr. David Ingram for pointing me in the direction of Henry's thesis and providing support and encouragement along the way. Special thanks go out to my parents, John and Ruth, for the friendly words of encouragement from home. Most importantly, my heartfelt thanks to Deborah Wilson for her continuous support and understanding through the period of this work.

Finally, I would like to thank the Department of Computer Science for their financial support.

Chapter 1

Introduction

Computational approaches to the derivation of phrase-structure have traditionally been based upon the implementation of strategies which use explicit phrase-structure rules. Whether it was parsing tables, the ATN framework [Woods 70], or logic grammars [Pereira *et al.* 80], early approaches to parsing normally relied upon a list of simple rules which claimed to provide a broad coverage of the language of interest. Although rule-based systems remain popular, recent developments in the field of natural language processing (NLP) suggest that researchers are looking more to well developed, current linguistic theories for a working paradigm. This shift away from the simplistic enumeration of phrase-structure rules seems to be the result of two developments in the NLP field. Firstly, researchers have experienced difficulties in developing rule bases for natural languages. As Barton points out, rule-based systems are both unconstrained and stipulative in nature [Barton 84]. The number of rules can often become unmanageably large and the correctness of the rules is difficult to ensure given the lack of underlying principles. Secondly, the emergence of cognitive science has led to the questioning of traditional computational models on the grounds of psychological validity. It can be argued that, given the rather dismal record of NLP, the most sensible approach is to try to emulate the only functioning language processor in existence - the human brain. Of course, the human brain is far too complicated to attempt an emulation of its specific functioning (at least at this time); but one can use linguistic theory, which is based upon the study of the 'inputs and outputs' to and from the brain, to attempt 'black-box' models.

Paramount in the cognitive science criticism of rule-based systems are questions of learnability. If one is to design a system that emulates human processing, it must not only work in a way that is arguably similar to an adult's processing, it must also be plausibly learnable by a child. This thesis will be concerned with just such a goal, the implementation of a system which is conceivably learnable. To actually demonstrate language acquisition would be an overly ambitious goal for this thesis, therefore we will concentrate only upon the implementation of a 'mature' parsing system, and leave acquisition for future research. We will still present acquisition arguments, but only in the context of justification for the presented parsing mechanism.

Questions of learnability within a system are usually difficult in and of themselves, but natural language acquisition systems must overcome two additional problems which are products of the child's learning environment - the poverty of stimulus argument and Gold's paradox.

1.1 The Poverty of Stimulus Argument

Researchers have long noted that the acquisition of language proceeds in an environment which is not conducive to learning. Children are exposed to limited examples of natural language, sometimes of ungrammatical form. Yet all children (not afflicted with a major intellectual disability) in a given environment manage to achieve a complete and structurally similar grammar. Hornstein and Lightfoot outline these deficiencies on three levels [Hornstein *et al.* 81]:

> (i) Children hear speech which does not consist of completely grammatical sentences, but also sentences with pauses, incomplete statements, slips of the tongue, etc.
>
> (ii) Despite being presented with finite data, children become able to deal with an infinite range of utterances.
>
> (iii) People attain knowledge of the structure of language, despite the absence of such data. That is, people are able to make judgements concerning complex/rare sentences, ambiguity relations, and grammaticality using knowledge which is not available as primary linguistic data (PLD) to the child.

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

1.2 Gold's Paradox

In 1967, the publication of [Gold 67] posed questions of language learnability in a mathematical framework, and presented researchers with a plethora of issues which remain largely unresolved. The work defined language learning in terms of a paradigm known as identification in the limit. By definition, a language is identified in the limit when no string from that language will force a learner to alter the grammar s/he has hypothesized on the basis of previous strings. Gold showed that, given a completely general learner, certain classes of languages within the Chomsky hierarchy were identifiable in the limit, depending upon the way in which the strings were presented. If appropriately labelled grammatical and ungrammatical strings were presented, then the class of primitive recursive languages could be identified. If only grammatical strings were presented, then only the finite cardinality languages were identifiable.

The effect of these findings were to create a learnability paradox for researchers of natural language since: a) it appears that children only make use of grammatical strings in their acquisition process (see [Brown *at al.* 70] or [Newport *et al.* 77]), and b) human languages are certainly of greater complexity than the finite cardinality languages (which simply consist of a finite list of strings). Thus, researchers wishing to postulate a theory of language acquisition based on a general learning procedure (and classes of language in the Chomsky hierarchy) must adopt at least one of a number of working techniques, the most popular of which are as follows:

(i) ordering the data presentation

(ii) relaxing the identifiability criterion

(iii) introducing a stochastic element into the learning procedure

(iv) constraining the learner's hypothesis space

It has been shown that any of these procedures will enable a learner to identify the class of primitive recursive languages (which certainly does include natural language).

This thesis will be adopting the fourth assumption, the most widely accepted of the techniques. As explained below, we will posit that the child's hypothesis space includes but a small subset of possible languages (this subset being at least all human languages) which the child must 'choose from' according to the language s/he is exposed to.

1.3 Universal Grammar

Theories of Generative Grammar (GG) posit the same solution to both the poverty of stimulus argument and Gold's paradox; a universal knowledge of grammar which underlies all human language ability. This Universal Grammar (UG), which is inherent in all persons, consists of a number of principles and parameters which guide the development of grammar. Differing adult grammars (corresponding to different languages) represent but a variation in parameter settings; all human language is based upon the same, unaltering principles.

The existence of UG answers both of the problems raised above. UG provides intrinsic knowledge of language to the learner, reducing the process of acquisition from a general learning problem to a process of parameter setting. Children do not have to be exposed to all language phenomena, only that necessary to select the proper setting for the language of exposure (which would then regulate similar phenomenon which the child has yet to encounter). Thus, the poverty of stimulus argument is answered.

UG also limits (to the extent of parameter variation) the learner's hypothesis space. Instead of having to select a valid grammar from all possible grammars, the language learner need only select one of the grammars represented by parametric variation. This reduction of the hypothesis space avoids Gold's paradox and allows the learning of any primitive recursive language represented in the hypothesis space, which certainly includes human languages.

1.4 Principle-Based Approaches to Grammar

The development of GG theory is, in fact, the formulation of theories on the structure of UG; what principles and parameters does it contain, and how do the principles interact to produce observed human grammars? Given that GG is motivated by questions of language acquisition, one would think that such questions would be paramount in any theory of GG. Sadly, this is often not the case. Most GG theories all but ignore questions of 'real-time' acquisition, concentrating instead on the development of theories which explain observed adult grammar phenomena. Such theories are often highly elegant and descriptive of mature grammars, but present a language learner with a highly complex system to acquire. These theories also tend to ignore the child's mapping of prelinguistic relationships onto grammars, instead positing descriptiveness in purely linguistic terms.

One should not be too surprised by this. Theories in any immature discipline tend to ignore issues of high complexity to concentrate upon specific, bounded domains, even if these complex issues directly affect the domain in question. Indeed, many linguistic theories ignore the problems of acquisition entirely, not giving a second thought to poverty of stimulus arguments or learnability theory. This, of course, does not mean that these theories are completely wrong, just that they are incomplete at this time. One cannot realistically hope for a complete and correct linguistic theory so early in the development of this field, only for theories that contribute to the overall understanding of language.

This does not mean that one cannot compare and evaluate differing theories, just that one must choose evaluation criteria to suit one's needs. The overall goal of this thesis is to investigate linguistic theories which are amenable to both computational implementation and child language acquisition[1]. Therefore, a linguistic theory which addresses questions of learnability and psychological validity will be considered superior to one that does not. Thus, the reader should not be surprised to learn that we will be investigating theories of GG in general (which is, at least at the basic level, driven by questions of learnability), and a GG theory which is

[1]Actually, we will be examining, in detail, one linguistic theory which meets these criteria.

based upon questions of learnability in particular. Such a theory, if amenable to a computational implementation, could be the basis for a functioning model of language acquisition. Such a goal is beyond the scope of this thesis, but we can at least perform the preliminary 'ground work' by implementing a model of a mature grammar.

1.5 Contents of the Thesis

In this thesis we will present an example of such a 'conventional' approach to GG theory (chapter 2) and then a recent GG proposal which is based upon questions of language acquisition (chapter 3). Specifically, this latter theory relies upon prelinguistic relationships to build a phrase-structure representation for a sentences. In chapter 4, we will present a computational model of phrase-structure representation based upon these prelinguistic relationships and, in chapter 5, argue that such a model is superior to those which rely upon purely linguistic specifications of phrase-structure. Finally, chapter 6 presents a general summary of the work and results. Appendix A presents a number of examples of phrase-structure construction generated by the described system, while appendix B details the Prolog source code of the system[2].

[2]The presented parsing system is written in CProlog. A working knowledge of Prolog on the part of the reader is assumed throughout this thesis and may be necessary for the discussion of the implementation. For an introduction to Prolog, the reader is referred to [Hogger 84], [Sterling *et al.* 86] and/or [Clocksin *et al.* 81]. For a more theoretical discussion see [Lloyd 87].

Chapter 2

Government and Binding Theory

The principle goal of theories like Government and Binding is the identification and formalization of the principles which compose our innate knowledge of grammar (i.e., UG). These principles are usually stated in a general form and include parameterization for specific types of language. Most theories postulate six main components (modules) to the model which apply to one or more of the six main levels of the grammar (see Fig. 2.1). The six components are:

1) $\overline{X}$-Theory

2) Θ-Theory

3) Case Theory

4) Bounding Theory

5) Binding Theory

6) Government Theory (ECP)

2.1 Θ-Theory

Θ-theory is concerned with the mapping of arguments bearing thematic roles into syntactic trees. Although the concept of thematic roles is derived from semantics, GB theory is primarily concerned only with the syntactic behaviour of those elements which assign or receive Θ-roles and not their semantic content. In English, and apparently the vast majority of natural languages, there is a consistent assignment of the 'agent' Θ-role to the external argument position (the subject) and of other Θ-roles to internal argument positions (complements).

The actual assignment of Θ-roles itself is secondary in importance to the behaviour of Θ-marked arguments in syntactic derivations and representations. This is constrained by the Θ-criterion which states:

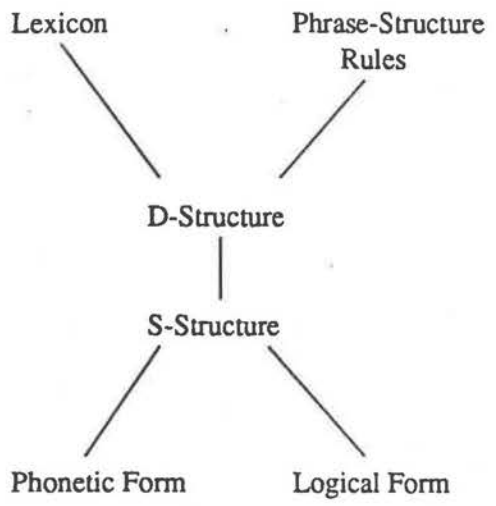


Figure 2.1: Government and Binding model of grammar

The Θ-Criterion:

> Every syntactically expressed Θ-role must be assigned to one and only one argument chain.

where an argument chain is a series of coindexed positions within the representation which constitute a history of movement. This version of the Θ-criterion is a generalization, as there are many competing versions in existence which make certain differing predictions about certain problematic cases; but for the vast majority of 'normal' utterances, this generalization is valid for most theories.

There is a significant difference in the application of the Θ-criterion when it comes to external and internal arguments. Internal argument positions are generated solely to bear Θ-roles - i.e., there is a one-to-one correspondence between internal arguments and internal argument positions. On the other hand, subjects are generated independently of Θ-assignment - i.e., a subject position will exist whether it receives a Θ-role or not. In turn, non-Θ-marked subject positions are the only possible targets for NP-movement, since movement from one Θ-marked position to another (to receive obligatory Case) would violate the Θ-criterion. Additionally, Θ-theory explains the distribution of pleonastic subjects such as "it" and "there" (pleonastics receive no Θ-roles), which occur only in non-Θ argument positions.

According to [Chomsky 81], the Θ-criterion applies at all levels of the grammar (except the lexicon), thus making it a general well-formedness condition on every level of syntactic representation. At S-structure and LF this serves as a severe constraint on possible A(rgument)-chains.

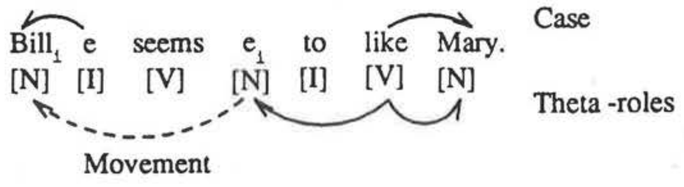

Case
Bill$_i$ e seems e$_i$ to like Mary.
[N] [I] [V] [N] [I] [V] [N]
Theta -roles
Movement

Figure 2.2: NP-movement

2.2 Case-Theory

Case-theory in English is rather straightforward. Certain elements always assign Case in a predefined direction (always under government, see Section 2.3), and all NPs in argument positions must receive Case. In certain other languages, Case morphology is much richer and requires a more complex discussion than is presented here. We need to distinguish only three Cases in English: genitive (assigned to the subject of NP), nominative (assigned by the abstract AGR element to the subject of Tensed clauses), and accusative/objective (assigned by [-N] elements to NPs immediately to their right).

The Case-filter, which blocks the derivation of representations in which an NP receives no Case, is expressed as:

The Case-Filter:

> *[Chain], where Chain contains a lexical element which has no Case.

We say that the Case-filter applies at S-structure, although arguments have been made for application at PF and LF.

Together with Θ-theory, Case theory has the effect of forcing NP-movement. In typical NP-movement cases, an NP in a complement position (which must receive a Θ-role) which does not receive Case is forced to move to a subject position which receives Case but no Θ-role (see Fig. 2.2).

2.3 Government Theory

The relation of government, a constraint on the form of derived representations, is crucial to at least three (and possibly five) of the major components of GB: Case theory, Binding theory, and the ECP (and possibly Bounding theory and Θ-theory). There are so many competing theories of government in existence that we will limit ourselves to a discussion of only the 'core' concepts.

There are usually two parts to any definition of government, a list of possible governors and a description of a structural relation between a governor and the category it governs. In English, the set of relevant governors varies from module to module. For Θ-theory, it is a Θ-assigner; for Case theory, either a [-N] lexical head or AGR; for Binding theory, a lexical head; for Bounding theory, a Θ-marking lexical head; and for the ECP, a lexical head or coindexed antecedent. However, all of the relations are centered around the 'core' notion of government as a relation between a lexical head and its complements.

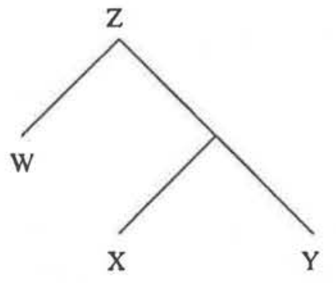
Turning to the structural part of the definition of government, we find that the core notion of c-command

> c-command:
>
> > A node X will c-command a node Y just in case there is a node Z such that Z immediately dominates X and Y.

is matched by a less restrictive notion called m-command

> m-command:
>
> > A node X will m-command a node Y just in case there is a node Z such that Z is the Minimal Maximal Projection containing X and Y.

The notion of Minimal Maximal Projection (MMP) refers to the 'next maximal projection up' from both X and Y. Thus, given the configuration:



we say that X m-commands W and m-commands and c-commands Y, if Z is a maximal projection.

The theory of government incorporates c-command/m-command by simply claiming that such a relationship must exist between a governor and the category it governs. Far more complicated is the definition of a barrier to government which limits the notion of command downwards in a tree (the definitions of m-command/c-command limit government upwards). Many different proposals have been made concerning what constitutes a barrier, but none is uncontroversial. We will not discuss these proposals here, but instead direct the reader to [Chomsky 86b] and [Kayne 84] for further discussion.

2.4 $\overline{X}$-Theory

$\overline{X}$-theory is a set of restrictions on the phrase-structure (base) component of a generative grammar. While all conditions which characterize context free grammars also characterize $\overline{X}$-grammars (i.e., $\overline{X}$-grammars are a proper subset of context free grammars), several additional restrictions apply to $\overline{X}$-grammars. The most important of these restrictions are endocentricity, maximality, and succession; as well, some $\overline{X}$-grammars adhere to uniformity, optionality and centrality [Pullum 85].

Before we may discuss these restrictions, we must first (informally) define a number of terms:

> A **head** is a category X^n which projects up to a phrase X^{n+1}, where n refers to the number of bar-levels.
>
> A **lexical head** is of zero bar-level, its immediately dominating projection is of bar level one, and so on.
>
> A **projection** of a category X^n is a category $X^{>n}$ bearing the same categorial features as X^n and which dominates X^n.
>
> A **maximal projection** is a phrasal category X^n which is the 'highest' projection of a lexical category X^0.
>
> **Categorial features** are those features which determine the category of the lexical item and all its projections in a phrase-structure tree. Chomsky's feature system employs the features [+/- N], [+/- V] to characterize the major syntactic categories V, N, A, P [Chomsky 70]. Note that this system does not incorporate the 'minor' categories DET, INFL, COMP or other 'functional' elements such as quantifiers, conjunctions, and markers of negation. Thus, N, V, A, and P are all lexical heads which project their categorial features up to the maximal projections NP, AP, VP, and PP, respectively (definitions taken from [Davis 89]).

The first of the restrictions is endocentricity. An endocentric category is one whose categorial features are those of a category it immediately dominates. The 'main' categories; VP, AP, NP, and PP; are all endocentric in that they are all phrasal projections which contain a head with the same categorial features. There is debate as to whether all phrasal categories are endocentric (S and $\overline{S}$ have been claimed to lack heads). There also exist well known exceptions to endocentricity (eg., nominal gerunds [Abney 87]).

Maximality refers to the condition that all non-head categories be generated as maximal projections. Although this seems generally true of complements, it is not necessarily true for specifiers such as NP and $\overline{S}$ subjects, as well as lexical elements such as determiners.

Succession refers to the condition that each member of a projection has a bar-level one more than its head. Most current versions of GB theory have relaxed this condition to allow a projection to have the same bar-level. This modification allows for the existence of adjunction structures in the base component.

Uniformity refers to the condition that all maximal projections have the same bar-level. There is little agreement as to what this bar-level should be, and it may be that uniformity is too strict a condition for natural language.

Optionality refers to the condition that all non-head projections are optional. Once again, specifiers seem to violate this condition as, for example, S must always have a subject (SPEC I), though this could conceivably be a result of other components of the system.

Finally, centrality refers to the condition that the 'start symbol' in the grammar (i.e., either S or $\overline{S}$) be part of the $\overline{X}$-system. This condition is also somewhat controversial as there is a debate over the endocentricity of S and $\overline{S}$.

There have been numerous phrase-structure schemas proposed which generally meet these $\overline{X}$ conditions; we will examine that of [Chomsky 86].

$$\overline{\overline{X}} \rightarrow \text{(SPEC X)}\ \overline{X}$$
$$\overline{X} \rightarrow \text{X (COMP X)}$$

where both SPEC X and COMP X are maximal projections. S is analyzed as $\overline{\overline{I}}$ and $\overline{S}$ as $\overline{\overline{C}}$.

This schema meets all of the $\overline{X}$-grammar restrictions. It satisfies endocentricity (each phrase X^n has a head X^{n-1}), maximality (non-heads are all maximal projections), succession (each projection X^n contains a head X^{n-1} which is one bar-level less than itself), uniformity (all maximal projections are of bar-level two), optionality (SPEC X and COMP X are optional), and centrality (S and $\overline{S}$ are both described within the $\overline{X}$ schema.

Having introduced $\overline{X}$ schemata, we must now ask ourselves what advantages such schemata have over traditional versions of phrase-structure. The strongest argument in favour of the $\overline{X}$ system is generality. It seems to be generally true that most categories are endocentric (eg., NP $\rightarrow$ AP V is very unlikely if not impossible), a prediction made by $\overline{X}$-theory but not by ordinary phrase-structure grammars. Perhaps more importantly, a system which incorporates categorial features can capture cross-categorial generalization (eg., only [-N] categories assign objective Case). Such generalization may play an important role in explanations of language learning and cross-linguistic generalization.

2.5 Binding Theory

The purpose of binding theory is to establish coreference between NPs. For the purposes of binding theory, NPs are grouped into three classifications:

Anaphors include reflexive pronouns, the reciprocal expression "each other", the trace of NP-movement, and PRO.

Pronouns include non-reflexive pronouns, pro and PRO (which is also an anaphor).

R(eferring)-expressions include ordinary referential NPs (names) and $\overline{A}$-traces (variables).

Each of the types of expressions are associated (for the purposes of binding theory) with anaphoric and pronominal features. Anaphors are said to be [+A,-P], pronouns [-A,+P], R-expressions [-A,-P], and PRO [+A,+P] (since it is both an anaphor and a pronoun).

Binding theory is said to have three Binding Conditions:

Condition A: An anaphor must be bound in its Minimal Governing Category (MGC).

Condition B: A pronoun must be free in its MGC.

Condition C: An R-expression must be free.

where bound is defined as being coindexed with a c-commanding antecedent. We also define **locally A-bound**, where the antecedent is the 'closest' available binder occupying an argument position. Free is the opposite of bound, and an MGC is the smallest category containing x (a pronoun or an anaphor), the governor of x, and a SUBJECT accessible to x.

Accessible refers to two conditions on SUBJECTs. Firstly, they must c-command x, the anaphor or pronoun whose governing category they are helping to define. Secondly, they must not violate the i-within-i condition, which stipulates that no category may be coindexed with a category containing it.

A SUBJECT is defined informally as "the most prominent subject" and formally as:

SUBJECT:

> The SUBJECT of a clause is [AGR_i, S] if there is one, otherwise [NP_i, S] or [NP_i, NP] (where [X, Y] means "the X immediately dominated by Y" (modulo such nodes as INFL and AUX)).

2.6 Bounding Theory

Bounding theory, unlike any of the other major components of the GB system, is a constraint on derivation rather than representation. This means that unlike the other components of the theory which apply to one or more levels of representation, bounding applies to the transformational mapping between D-structure and S-structure (i.e., syntactic movement).

Bounding constrains the Move-α transformational rule (move anything, anywhere) by specifying that no single application of Move-α may cross more than one bounding node, where a bounding node is defined as NP and S (and, in some languages, $\overline{S}$). This constraint is known as subjacency and was introduced to incorporate many features of constraints proposed by [Ross 67] (CNPC and SSC) and [Postal 69] (WhIC):

> **Complex NP Constraint (CNPC):** No element contained in an S dominated by an NP may be extracted from that NP.
>
> **WH-Island Constraint (WhIC):** No element contained in an indirect question, $\overline{S}$, may be moved out of that $\overline{S}$.
>
> **Sentential Subject Constraint (SSC):** No element may be extracted from an S if that S is a (sentential) subject.

In cases of apparently unbounded movement (eg., Wh-movement) subjacency forces the adoption of a COMP-to-COMP analysis in which the moving element moves through a number of empty COMP elements by repeated applications of Move-α, no single movement violating subjacency.

Recently, there have been attempts (sparked by [Chomsky 86b]) to incorporate the CED of [Huang 82]

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

into the subjacency condition, leading to the development of barriers. A barrier is defined as:

α is a barrier for β iff:

i) α is a maximal projection,

ii) α is not L-marked, and

iii) α dominates β

or,

i) α is L-marked,

ii) α dominates β, and

iii) β is a Bounding Category.

where α is L-marked by β iff β is directly Θ-marked by α [Lasnik *et al.* 86], and Bounding Category is defined as follows:

Bounding Category

> β is a Bounding Category if β is a maximal projection and β is not L-marked.

Subjacency is now expressed as follows:

> **Subjacency:** β is subjacent to α if for every γ a barrier for β, the maximal projection immediately dominating γ dominates β.

2.7 The ECP

The Empty Category Principle (ECP) is a complicated, much debated theory on the distribution of empty categories (e) within a sentence. As the ECP is not particularly relevant to this thesis, we will avoid the in-depth analysis and controversy which normally accompanies such a discussion and present only the simplest definition. Interested readers are referred to, for example, [van Riemsdijk *et al.* 86] for further discussion.

Empty Category Principle:

> [e] must be properly governed.

Proper Government:

> X properly governs Y iff X governs Y and X is either X^0 (i.e., V, N, A, P) or NP_i, where Y (the governee) equels NP_i.

Chapter 3

A New Government and Binding Theory

As previously discussed, Government and Binding Theory is not one monolithic theory, but rather a number of coexisting theories which share an approach to the definition of grammar. New theories are constantly being brought forward to account for a few more extraordinary cases, or to subsume one more linguistic phenomena, or (as is most often the case) to dispute an existing theory of a portion of GB. A particularly radical reinterpretation of GB theory has recently appeared in the form of [Davis 87]. This version of GB interests the author because it has been developed specifically to account for language acquisition. Davis bases his theory on Chomsky's definition of epistemological priority

> ... we want the primitives to be concepts that can plausibly be presumed to provide a preliminary, prelinguistic analysis of a reasonable selection of presented data, that is, to provide the primary linguistic data that are mapped by the language faculty to the grammar ... [Chomsky 81, p.10]

Thus, principles which match elements and structures from prelinguistic systems with elements and structures in linguistic systems are considered to have epistemological priority over those principles which are defined purely linguistically or nonlinguistically. It is on this basis that Davis proposes to eliminate $\overline{X}$-theory from GB theory and replace it with a series of percolation principles which map prelinguistic relations onto phrase-structure representations (formed in a level of representation called NP-structure, as first proposed by Van Riemsdijk and Williams [81] - see Fig. 3.1). In this chapter, we will recount the development of this theory, introducing new concepts of government (Section 3.1), Θ- and Case- theory (Sections 3.2 and 3.3 respectively), and a series of feature percolation principles that determine which categorical features filter up to form the dominating node of two sister nodes in a representational tree (Section 3.5).

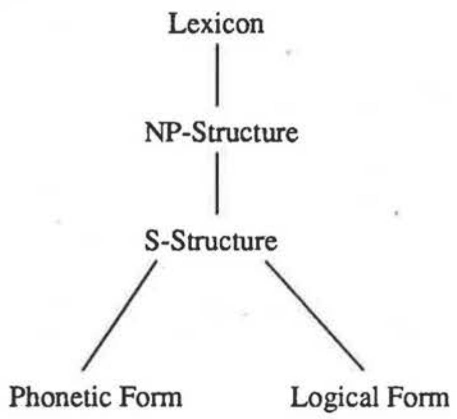


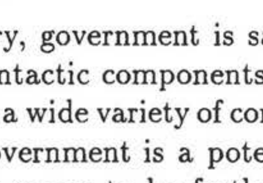
Figure 3.1: Davis's GB structure

3.1 Government

In most versions of GB theory, government is said to impose a general condition on the mapping of various syntactic components onto one another. Because of its importance and application to a wide variety of component theories (i.e., components of the overall GB theory), government is a potentially unifying concept. Yet, no unified version of government seems to be forthcoming. Davis argues that this is the way it should be, because there is no unified concept of government. Instead, he posits two distinct forms of government, internal and external. Internal government concerns the relationship between a lexical governor and elements within its maximal projection. External government concerns the relation between a governor and the elements within a maximal projection which it governs. It seems natural to treat these two concepts separately, and thus we shall, starting with internal government.

3.1.1 Internal Government

The core case of internal government is defined as a lexical head governing its complements (eg., a verb governing its direct object). This would give us the configuration:



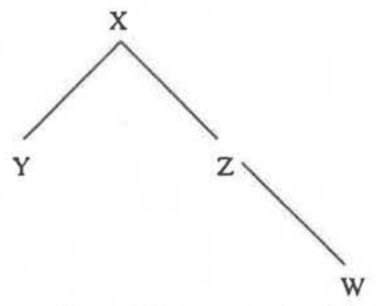
where W is a Case-assigning, Θ-assigning head, Z is its complement, and Y is a projection of W. In this case, we say that W and Z are in a Canonical Government

Configuration (CGC). More formally:

> W and Z, immediately dominated by some Y, are in a Canonical Government Configuration (CGC) iff:
>
> a. V(erb) Case- and Θ- marks NP to its right in the grammar of the language in question and W precedes Z or
>
> b. V Case- and Θ- marks NP to its left in the grammar of the language in question and Z precedes W.

English is an example of an a. type grammar. We refer to this core type of government relation as minimal government, as opposed to maximal government as described below.

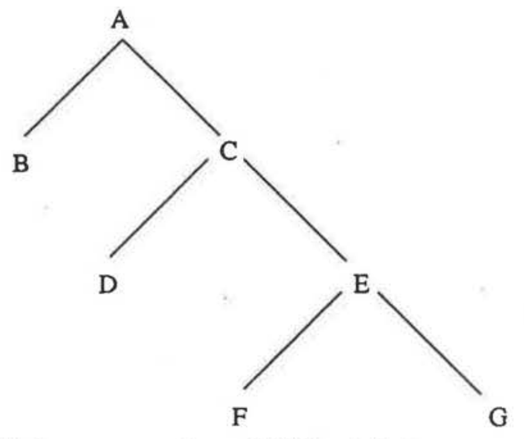
Given the configuration:



where W is a lexical governor, Z and X are projections of W, and Y is a specifier to W, we say that W maximally governs Y. This corresponds to the widely accepted version of government first proposed by Aoun and Sportiche [83], where a governor governs everything up to a maximal projection (equivilent to the m-command of [Chomsky 86b]).

3.1.2 External Government

External government is a rather murky concept which (fortunately) is unnecessary for the generation of phrase-structure. Hence, we will not go into great detail but rather present a cursory summation of its definition as a contrast to internal government. Readers with an interest in this topic are referred to [Chomsky 86b] for an in-depth analysis. Given the configuration:



where B is a potential governor in a CGC with its complement C, D is a specifier to F, and C and E are projections of the lexical head F; and given that B governs C (and assuming that C is not a barrier to government - see [Chomsky 86b]), then we can state that:

a. B governs D,

b. B governs F by the percolation of government down from the projections of F, and

c. B does not govern G, since the government of G by F blocks government by B.

Having characterized internal and external government, we can now distinguish (and formally define) two distinct forms of government, minimal and maximal.

Minimal government:

> α minimally governs β iff α minimally c-commands β and there is no τ such that α governs τ and τ governs β.

Maximal government:

> α maximally governs β iff α maximally c-commands β and there is no τ such that α governs τ and τ governs β.

3.2 Θ-Theory

Central to the GB approach to the assignment of Θ-roles is the Θ-criterion:

The Θ-Criterion:

> Each argument bears one and only one Θ-role, and each Θ-role is assigned to one and only one argument. [Chomsky 81, ch.2]

This is a slight simplification ([Chomsky 81, ch.6] reformulates this definition in terms of Argument-chains) but it is sufficient for our purposes.

3.2.1 PRO

The Θ-criterion has the effect of preventing movement from one Θ-marked position to another. Thus, raising to object is always prohibited (since complement positions are always Θ-marked) but raising to subject is permitted as long as the subject position is not assigned a Θ-role (for example, monadic predicates such as "seems" do not assign a Θ-role to their subject). If one adheres to this strict version of the Θ-criterion, one must formulate a way of dealing with defective predicates. These predicates are of two types. Firstly, adjunct predicates (so called "small clauses").

John ate the meat raw.

Here "ate" assigns a Θ-role to "John" and "the meat", saturating their Θ-role receiving capabilities, while "raw" assigns one Θ-role to its left. Since all Θ-role receiving capabilities are saturated, an empty subject pronoun (PRO) must be postulated to receive "raw"'s Θ-role.

John ate the meat PRO raw.

Davis dismisses this formulation by referring to the work of Williams:

> However, Williams (1980,1983) provides several arguments against this type of structure... the basic thrust of the argumentation is to show that the adjunct predicates behave differently in the syntax from "true" clauses, and treating the two as structurally parallel merely obscures these differences. [Davis 87, p.104]

The second type of defective predicate is exemplified by certain types of non-clausal complements. These include non-nominal gerunds, complements to verbs of perception, and temporal aspect, and the complements "to make" and "let". [Koster *et al.* 82] claim defective complements uniformly contain PRO; [Emonds 85] takes a 'mixed' position, claiming PRO is restricted to infinitivals. Davis argues PRO may be disposed with altogether. The arguments are quite detailed and, again, beyond the scope of this thesis. The main point made is that there are no strong structure-based arguments for the existence of PRO. We must have a mechanism to deal with defective predicates whether or not PRO is accepted; simply advancing PRO as a solution is insufficient. Davis does this (as have others) by modifying the Θ-criterion. His approach is to observe that the Θ-criterion can be broken into two constituent parts:

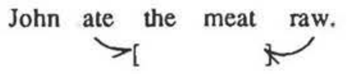


Figure 3.2: the NP "the meat" receives two Θ-roles

The Θ-Criterion:

a. each argument bears one and only one Θ-role, and

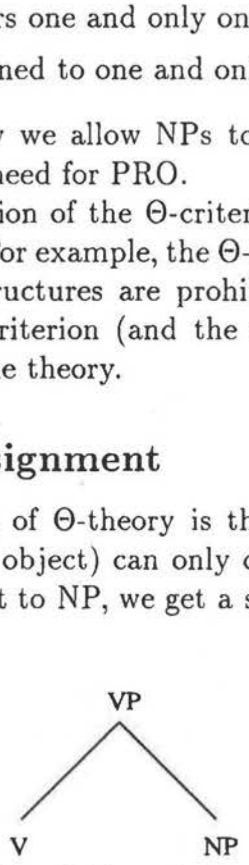
b. each Θ-role is assigned to one and only one argument

and then eliminating a. Now we allow NPs to receive two Θ-roles, as shown in Fig 3.2, thus eliminating the need for PRO.

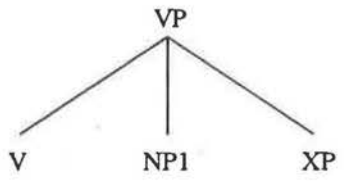
Note that this 'weak' version of the Θ-criterion allows many analyses that the strong Θ-criterion prohibits. For example, the Θ-criterion no longer prohibits raising to object. However, such structures are prohibited by the Projection Principle, so adoption of the weak Θ-criterion (and the elimination of PRO) has actually eliminated a redundancy in the theory.

3.2.2 Internal Θ-Assignment

Also central to the definition of Θ-theory is the fact that internal Θ-assignment (assignment of Θ-roles to the object) can only occur within a CGC. Thus, for the simple case of verb assignment to NP, we get a structure such as:



However, there are some verbs which assign two internal Θ-roles, either to two objects or to one object and one phrase. We still must adhere to Θ-assignment within a CGC, thus we derive structures with a tertiary branching structure:



a) I would like John to leave.

b) I would like very much *(for) John to leave.

Figure 3.3: The insertion of the Adverbial phrase "very much" necessitates the insertion of the Case-assigner "for".

3.2.3 External Θ-Assignment

Predication is the theory which stipulates how Θ-assigners (i.e., verbs or possibly VPs) are linked to their 'external arguments' (i.e., subjects). Most accounts of predication are Θ-based, in that they rely upon thematic linking between the verb and subject (for example, see [Williams 80] and [Rothstein 83]). Davis proposes that predication is in fact based upon the relationship between the verb and its AGR bearing INFL, and case assignment by the INFL to the subject. The fact that this relationship coincides with external Θ-assignment is merely a coincidence, not a given. Although quite complicated, and formulated more for the case of $\overline{A}$ predication (which is beyond the scope of this thesis), the predication theory highlights the fact that internal and external Θ-assignment differ quite dramatically. External Θ-assignment does not occur within a CGC, and in fact only indirectly involves the positioning of the Θ-assigner to the subject.

3.3 Case-Theory

Case-theory in GB can be informally defined by stating that all lexicalized NPs must bear Case. The formal definition of the theory is complicated by the fact that Case receivers (Arguments) can move, leaving behind a trail of co-indexed (Argument position) categories called an A(rgument)-chain. Most theories are further complicated by the need to include PRO as the head of an A-chain, but as we have done away with PRO, we can formally define the constraint on Case-assignment as:

an A-chain must be associated with Case.

As with Θ-assignment, Case-assignment is divided into two types, internal and external, each of which is governed by its own rules.

3.3.1 Internal Case-Assignment

Internal Case-assignment usually takes place in a CGC, although in certain circumstances it is able to penetrate a derived XP to (exceptionally) Case-mark its specifier. Internal Case is assigned by [V] and [P] and is subject to an adjacency condition which (in English) constrains Case-assignment to strict adjacency. Thus,

a) John probably did not realize how late it was.

b) John, I think, did not realize how late it was.

Figure 3.4: Adverbial and parenthetical information may intervene between a an external Case-assigner and receiver.

in Fig. 3.3, the insertion of an Adverbial phrase between the Case-assigner "like" and the Case-recipient "John" necessitates the insertion of the Case-assigner for.

3.3.2 External Case-Assignment

External Case-assignment differs from internal Case-assignment in at least the following ways.

i. In direction of Case-assignment. In English, the two external Cases, nominative and genitive, are assigned to the left, whereas the canonical direction of Case-assignment is to the right. However, there is evidence that this distinction does not hold over cross-linguistic analysis.

ii. In adjacency parameters. As mentioned above, there is a strict adjacency requirement on internal Case-assignment. This does not hold for external Case-assignment, as adverbial and parenthetical information can intervene, as in Fig 3.4.

iii. In level of application. In conventional accounts of GB theory, internal Case-assignment occurs at D-structure (along with internal Θ-assignment) while external Case is assigned at S-structure (since NP-movement, a syntactic process, is forced by external Case requirements). Davis proposes that both external and internal Case-assignment take place at NP-structure, but that the latter type involves 'virtual' movement from a D-structure position.

iv. In obligatoriness. Internal Case-assigners will only assign Case if they also assign an internal Θ-role (even if the Case- and Θ- assignments do not coincide). External Case-assigners always assign Case, whether an external Θ-role is assigned or not. This means that nominative Case must be assigned whenever AGR is present and genitive Case must be assigned whenever N is present.

v. In structural conditions on Case-assignment. Whereas internal Case is always assigned in a minimal government configuration, external Case is always assigned in a maximal government configuration, as shown in Fig. 3.5 for S and NP.

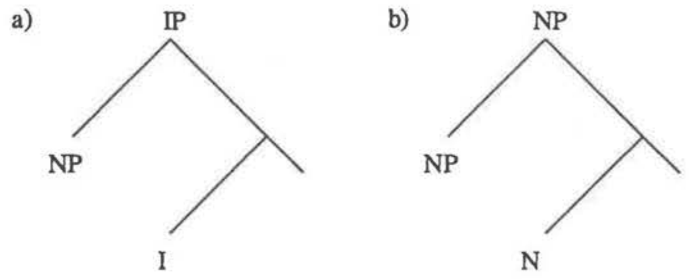


Figure 3.5: maximal government

3.4 Categorial Features

Most GB work assumes the schema adopted in [Chomsky 70] which accounts only for the 'major' categories [A], [N], [V], and [P].

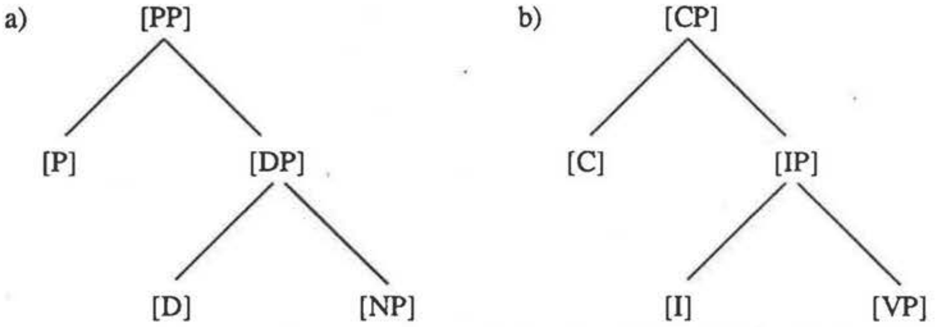
A = [+N,+V]
N = [+N,-V]
V = [-N,+V]
P = [-N,-V]

Davis proposes an alternative categorization based upon distinguishing three types of categories: Θ-heads, which includes [N], [V] and [A]; G-heads, which includes INFL and DET; and C-heads, which includes complementizers and prepositions. These three types of categories enter into categorical associations with each other based on a notion of functional discharge [as elaborated by [Abney 85], [Fukui *et al.* 86], [Higginbotham 86] and [Speas 86]) which forms the basis of the Percolation Principles to be presented below.

Θ-heads include the primary Θ-related categories, [N] and [V], and the inflectionally deficient Θ-head, [A]. G-heads (INFL and DET) do not assign primary Θ-roles but instead contain a small set of syntactically relevant features including those involving tense, definiteness, and number and person agreement. G-heads assign external Case and enter into categorical associations with Θ-heads. C-heads (prepositions and complementizers) act as linkers between Θ-heads and their complement and adjunct dependents. They may also assign Case or 'secondary' Θ-roles. In addition, they may bear G-features such as Tense and AGR. It may well be that C-heads have no intrinsic features of their own, and must 'borrow' features from G-heads and Θ-heads. Just as G-heads enter into categorical association with Θ-heads, so C-heads enter into association with G-heads. Thus, we get the following schema:

	C-heads	G-heads	Θ-heads
Nominal system	P	D	N
Verbal system	C	I	V

We will assume that the heads {P,D,N} are canonically linked in the nominal system, and the heads {C,I,V} are canonically linked in the verbal system. Thus, we get the following phrase-structure representations:

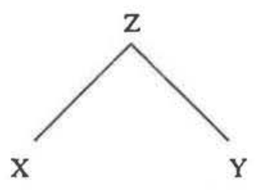


The above representations do not include the Θ-head [A] which, because of its special status, must be considered separately. [A] is special in that it assigns no Case at all and cannot be associated with INFL, yet it is still considered to be an (inflectionally defective) Θ-head. Since they are inflectionally defective, in order to assign internal Θ-roles, adjectives must subcategorize for PP complements; and in order to assign an external Θ-role, the dummy verb "be" must be inserted to bear Tense which in turn enables AGR to assign Case to the external argument. A special property of the 'empty' Θ-assigner be allows it to transmit a Θ-role from an adjectival head to its external argument. This can be done by having "be" 'adopt' the Θ-role of an [A] head and treating it as its own for external argument.

3.5 Percolation Principles

Given the feature-association model presented in Section 3.4, it is now possible to formulate explicit feature-percolation principles which will determine the structure of derived structural parse trees. Davis proposes four principles which, given two adjacent nodes on a parse tree, determine which categorical features will 'percolate up' to the dominating node. These principles are not meant as primitives to the theory since if we are to derive phrase-structure from the interaction of other, more primitive principles, then percolation rules should be as unnecessary as explicit phrase-structure rules [Speas 86]. Rather, they are meant as "'operating principles' which relate licensing conditions to strings to derive labelled trees" [Davis 87, p.150].

The first two principles straightforwardly define the relationships between Θ- and Case- assigners and receivers.



Percolation Principle I:

> Where X Θ-governs Y, the categorical features of Z will be those of X.

Percolation Principle II:

> Where X assigns Case to Y, the categorical features of Z will be those of X.

Percolation Principle III deals with the 'adjunction' set of a phrase, i.e., those elements which are not bound to others by Θ- or Case- relations. Such elements typically have no effect at all on categorical structure. In other words, the category dominating a member of an adjunction set will have the exact features as the other category to which it is joined.

Percolation Principle III:

> Where X is a member of the adjunct set and Y a member of the subcategorization set of a phrase Z, the categorical features of Z will be those of Y.

A more formal definition of categorical set and adjunct set requires a modification of the Revised Extended Projection Principle of Chomsky, which Davis calls the Generalized Revised Extended Projection Principle (GREPP).

Generalized Revised Extended Projection Principle:

> Subcategorization requirements must be satisfied by all phrase-structure configurations, where "subcategorization requirements" refer both to subcategorized and subcategorizing elements.

This redefinition of the Projection Principle differs in two significant ways from the traditional definitions. Firstly, by applying the principle to "all phrase-structure configurations", rather than all levels of the grammar, Davis has removed the restriction from D-structure. The theory we have presented here requires that D-structure be "nothing more than a set of (internal) Θ- and Case- relations, with at most a partial representation in configurational terms" [Davis 87, p.155]. It is not until the application of percolation principles that phrase-structure begins to emerge.

The second (and more important to PPIII) difference in the definition of the GREPP is the extension of the concept of subcategorization to include the subcategorized elements as well as the subcategorizing elements. This has the effect of generalizing the Projection Principle from a constraint on the distribution of NPs to a general constraint on the recoverability of lexical information.

Finally, we turn our attention to adjunct elements which are not found in any of the above defined relations. In this case Davis defines a simple feature-percolation hierarchy that determines which categorical features should dominate over others This hierarchy is given as Percolation Principle IV.

Percolation Principle IV:

> Where X and Y are in a CGC, no Case or Θ-relation holds between them, and both are part of the subcategorization set of Z, the following hierarchy determines which features will percolate:
>
> a. C-features of X and Y will percolate to Z
>
> b. G-features of X and Y will percolate to Z
>
> c. Θ-features of X and Y will percolate to Z

3.6 An Example Parse

This concludes the definition of Davis' theory of GB and the percolation principles which allow us to build phrase-structure. As a demonstration of these principles, we now present an example taken from [Davis 87].

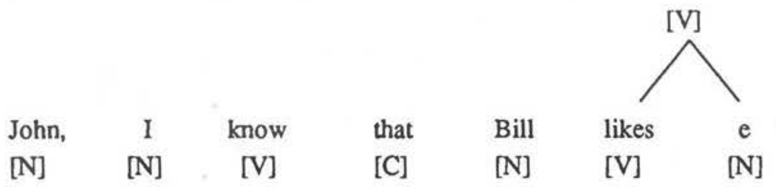
Let us consider the sentence:

John, I know that Bill likes.

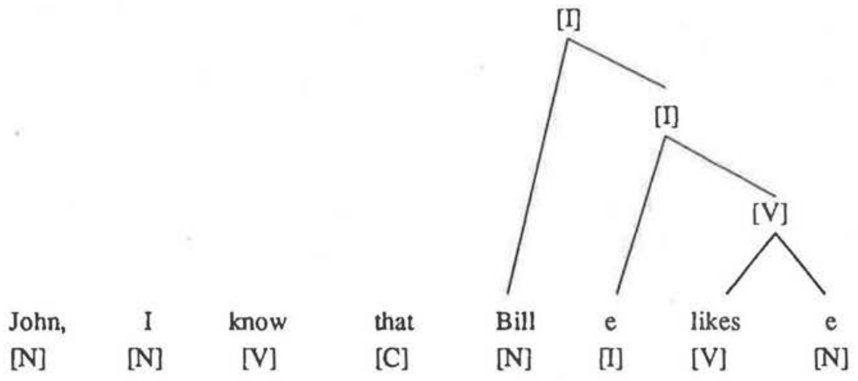
Now, assuming that all lexical categories have been correctly identified, we will get the lexical string:

John,	I	know	that	Bill	likes
[N]	[N]	[V]	[C]	[N]	[V]

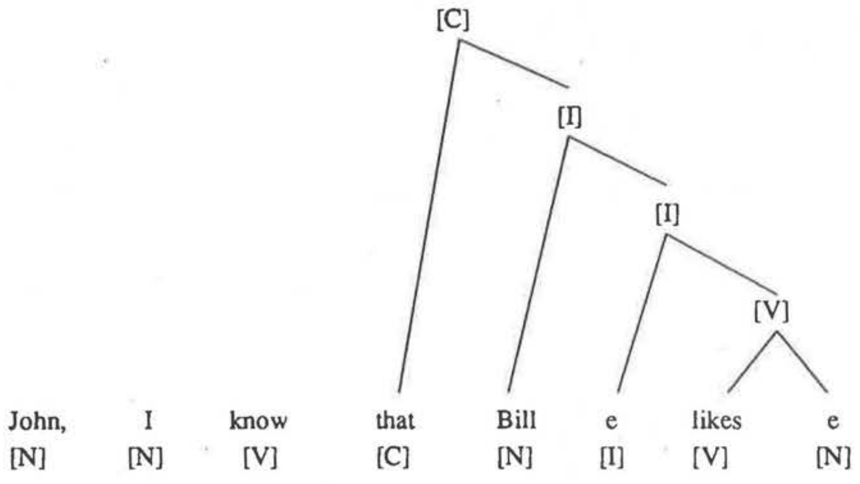
Again, assuming that the subcategorization properties of the various lexical items are known (since this is important to the operation of the GREPP, which in turn determines the distribution of empty categories,) we can begin to apply the principles themselves, beginning with the most embedded phrase, since we are assuming a bottom-up procedure. First of all, by Principles I and II, and by the GREPP, we construct the local tree:



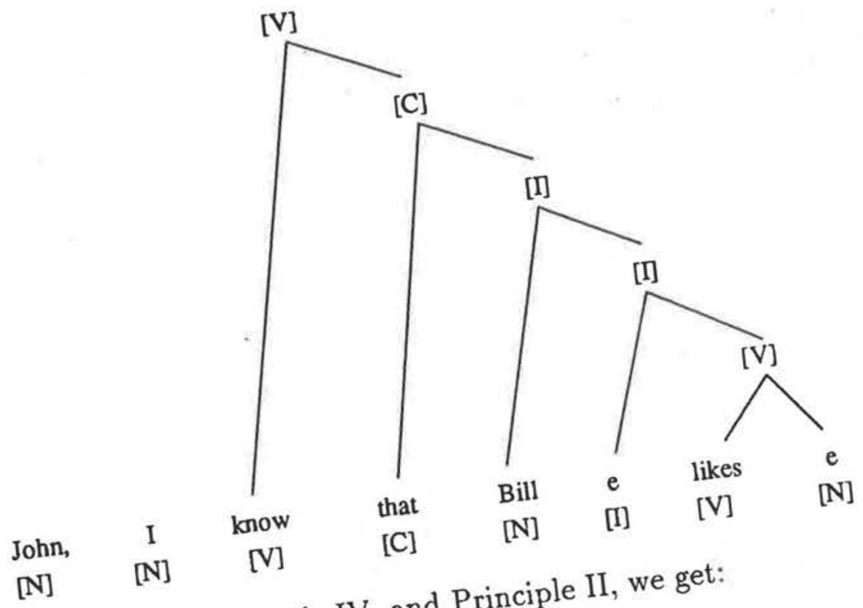
Next, we come to the Case-marked NP "Bill". By the GREPP, there must be an element which assigns Case to this NP; since the NP receives its Case in a non-canonical (external) government configuration, that element must be a G-head; since the G-head associated with [V] is [I], the missing element must be [I], and by Principles IV and II the relevant structural representation will be:


[I]
[I]
[V]
John, I know that Bill e likes e
[N] [N] [V] [C] [N] [I] [V] [N]

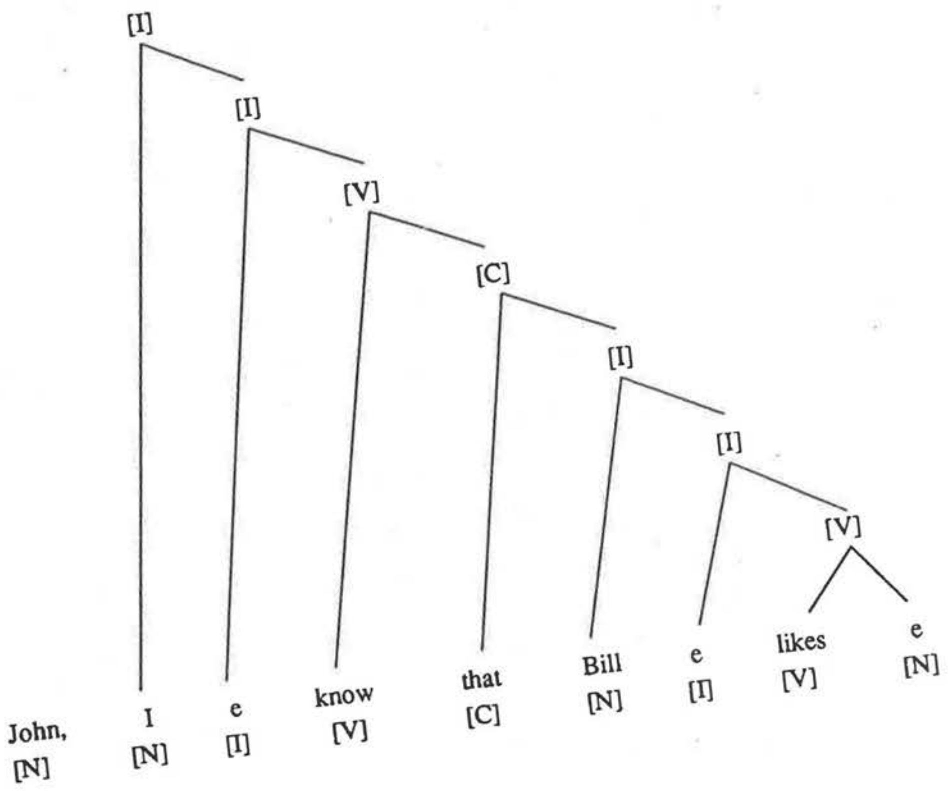
Next, by Principle IV, we will get the structural representation:


[C]
[I]
[I]
[V]
John, I know that Bill e likes e
[N] [N] [V] [C] [N] [I] [V] [N]

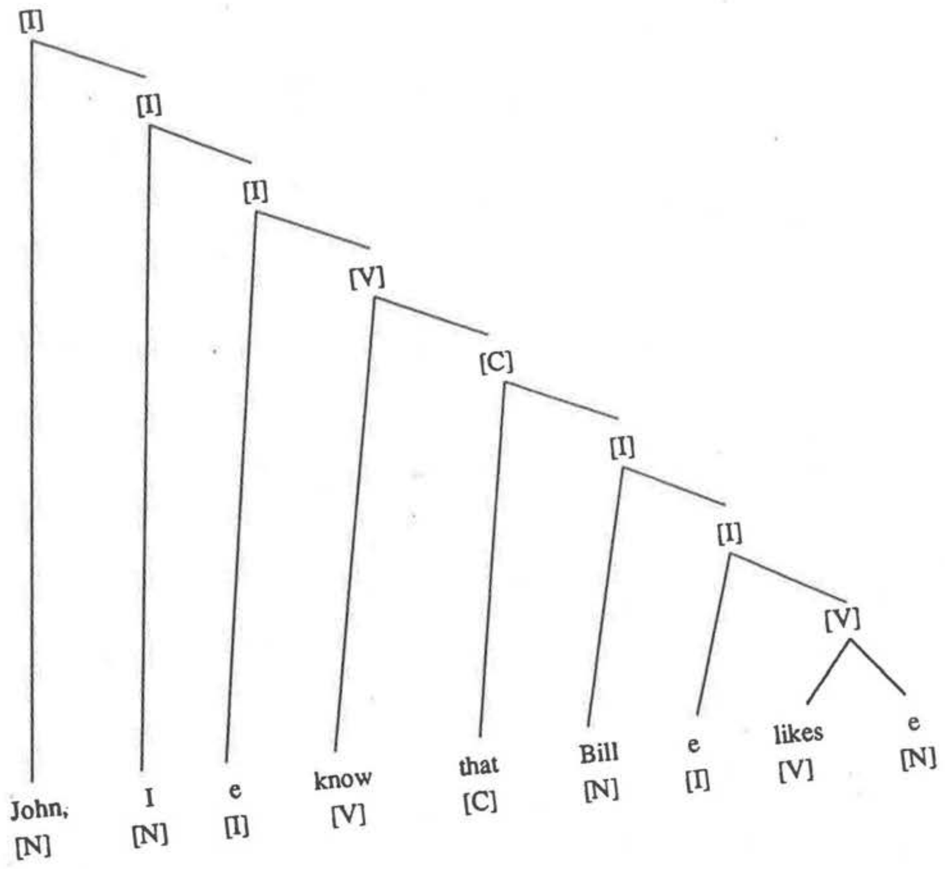
By Principle I, we then construct:



Next, by the GREPP, Principle IV, and Principle II, we get:



Finally, by Principle III, we end up with the phrase-structure representation:



Chapter 4

The Implementation

Now that we have presented the GB theory of Davis, we can introduce a parsing mechanism based on the theory. As Davis' theory mandates, this parser forms phrase-structure representations of sentences without the use of explicit phrase-structure rules. Instead, we use Percolation Principles to determine the categorical features of the dominating node of any two adjacent nodes. The system can be viewed as forming an augmented NP-structure representation of the sentence. We say augmented NP-structure because, although none of the 'higher level' modules of GB (such as Bounding, the ECP, etc.) have yet to act upon the representation (this would occur between NP-structure and S-structure, as well as between S-structure and LF), certain movement constraints already exist within the parsing mechanism itself.

4.1 The Lexicon

The lexicon constitutes the vocabulary of the system. It is implemented in a rather straightforward way as a list of elements representing each word the system recognizes and the lexical features associated with it. The representation of each word is the same, although various lexical features apply only to certain categories of words. This is done to provide a general form for all words, regardless of their category; which, in turn, simplifies processing. Lexical features not relevant to a particular category are assigned a value which represents a lack of this feature.

The general form of all entries in the lexicon is:

[Cat,Word,EC,ET,RC,RT,Ten,S,[Subcat]]

where the meaning of each identifier is as follows.

Cat - The category of the word. The lexicon provided for this implementation contains words of the following categories:

n - Noun

d - Determiner

v - Verb

c - Complementizer

i - INFL

adj - Adjective

adv - Adverb

p - Preposition

Word - The actual word. Note: different inflectival forms of the same root word appear in the lexicon as no morphological analysis is performed.

EC - The assignment of external Case. A value of 1 is specified if external Case is assigned, 0 otherwise.

ET - The assignment of an external Θ-role. A value of 1 is specified if an external Θ-role is assigned, 0 otherwise.

RC - The word's status as a Case receiver. A value of 1 is specified if the word receives a Case assignment, 0 otherwise.

RT - The word's status as a Θ-role receiver. A value of 1 is specified if the word receives a Θ-role assignment, 0 otherwise.

Ten - The Tense of the word. A value of "+" is specified if the word bears Tense, "*" otherwise.

S - An initially 'empty' identifier, it is used to 'carry' a subscript value associated with movement. All words have this identifier initially set to 0, indicating the word is not associated with movement, and will only have this value altered if processing recognizes the word to be associated with a movement trace. Although this is not in any way a lexical feature, it has been included in the lexical feature list for ease of processing.

Subcat - Subcat is a list containing the subcategorization information of the word. Of course, this will only bear information if the word subcategorizes for other elements (for our purposes, this includes a subcategorization definition of intransitive). All other words will simply be specified with an empty list. The internal structure of the subcategorization list can be quite complicated, owing to the complex subcategorizations associated with some words. For example, most subcategorizers subcategorize for more than one type of complement (i.e. the verb "loves" can be transitive or intransitive). Thus, the list Subcat must contain multiple entries, one for each subcategorized type.

[[Subcat1], [Subcat2],...]

Each subcategorization specification can possibly subcategorize for one or two elements (for example, the verb "gave" can subcategorize for two NPs - "I gave Mary a book."). Therefore, each Subcat entry must have the ability to contain up to two subcategorized elements.

[[[Element1], [Element2]], [Subcat2],. . .]

Each element which is subcategorized for has the same specification:

[C,IC,IT,W]

where each identifier represents:

C - The categorical type of the (possibly maximal) element being subcategorized for.

IC - The assignment of internal Case. A value of 1 is specified if internal Case is assigned, 0 otherwise.

IT - The assignment of an internal Θ-role. A value of 1 is specified if an internal Θ-role is assigned, 0 otherwise.

W - A small set of syntactic features which must be present on the head of the phrase immediately following[1] the subcategorizer (eg., +WH). This element is presently unused, thus every lexical entry bears the 'null' value "*".

Thus, the general structure of Subcat can be represented as:

[[[C,IC,IT,W], [Element2]], [Subcat2],. . .]

When the user inputs a sentence to the system (as a string of characters), each word is extracted and placed in a list. Each word is then compared to the lexicon and a representation list is built up containing the matched lexical entries as well as an end-of-sentence marker which is added to the end of the list. This list is then passed to the main parsing system where it is referred to as the Buffer.

4.2 Description of General Processing

The parsing mechanism is, in general, a bottom-up parser, although there are some elements of top-down processing involved in the creation of phrases. There are two distinct mechanisms within the parser, building of NPs and building of other phrasal elements (including S). We describe each mechanism separately, starting with parsing of phrasal elements.

[1]The **W** specification may include phrases which have moved.

```
process_input(Input,Tree) :-
                process_buffer(Input,[[bot_of_stack]],
                               [0,*,0,0,*,0],Buf,Tree,Infolist).


process_buffer([[end_of_sentence]],Stack,Infolist,
               [[end_of_sentence]],Stack,Infolist).

process_buffer(Buffer,Stack,Infolist,NewBuf,NewStk,NewInfo) :-
                clause_check(Buffer,Stack,Infolist,Buf2,[Con1|Stk2],Info2),
                clausal_move([Con1|Buf2],Stk2,Info2,[Con2|Buf3],Info3),
                process_buffer(Buf3,[Con2|Stk2],Info3,
                               NewBuf,NewStk,NewInfo).
```

Figure 4.1: the `process_input` and `process_buffer` predicates

4.2.1 The Building of Phrases

The processing of phrases, other than NP, is driven by the fulfillment of the subcategorization of subcategorizing elements. All phrases contain one subcategorizing element[2]: verb for IP, VP and CP; preposition for PP. A phrase is 'completed' by satisfying this subcategorization. So, when a phrase is 'triggered' (see below), all elements encountered until the subcategorization is completed are considered to be part of the phrase. The subcategorization of an element is said to be completed when a maximal element corresponding to the type subcategorized for is constructed and returned to the buffer position immediately following the subcategorizer. Of course, a subcategorizer which is intransitive is said to be complete. Subcategorizers which subcategorize for a phrasal element will cause a recursive invocation of the build phrase procedure, thus only intransitive subcategorizers and NPs (which do not rely upon subcategorizers for completion - see below) can truly complete a phrase (in that they do not cause a recursive invocation of the build phrase procedure[3]).

The parsing procedure is driven by the `process_input` and `process_buffer` predicates (see Fig. 4.1). `process_input` accepts the input Buffer and invokes the parsing mechanism (`process_buffer`), passing the Buffer, an empty Stack, and the Infolist. Infolist is used for monitoring movement of elements and will be discussed later. `process_buffer` builds the constituent sentence phrases until the end-of-sentence marker (found at the 'end' of the Buffer) is encountered. As each completed phrasal element is returned, it also checks for clausal movement (i.e., fronted clauses) and, if encountered, activates the appropriate movement flag in Infolist. Clausal movement is recognized as the return of a PP or CP which was not subcategorized

[2]Phrases may contain embedded phrases within them. It is correct to say that each phrase contains one matrix subcategorizing element.

[3]NPs may contain phrases, and thus will invoke the build phrase procedure, but they are not dependent upon subcategorizer satisfaction. See Building NPs.

```
clause_check([[[p | Info] | Constit] | Buf_content],Stack,Infolist,
             NewBuf,NewStk,NewInfo) :-
               process_pp([[[p | Info] | Constit] | Buf_content],Stack,
                          Infolist,NewBuf,NewStk,NewInfo).

clause_check([[[c | Info] | Constit] | Buf_content],Stack,Infolist,
            NewBuf,NewStk,NewInfo) :-
               process_cp([[[c | Info] | Constit] | Buf_content], Stack,
                          Infolist,NewBuf,NewStk,NewInfo).

clause_check([[[v | Info] | Constit] | Buf_content],Stack,Infolist,
            NewBuf,NewStk,NewInfo) :-
               process_ip([[[v | Info] | Constit] | Buf_content], Stack,
                          Infolist,NewBuf,NewStk,NewInfo).

clause_check([[[d,Word | Info] | Constit] | Buf_content],Stack,Infolist,
            NewBuf,NewStk,NewInfo) :-
               Word \== max,
               build_np([[[d,Word | Info] | Constit] | Buf_content], Stack,
                        Infolist,Buf1,Stk1,Info1),
               process_ip(Buf1,Stk1,Info1,NewBuf,NewStk,NewInfo).

clause_check([[[n | Info] | Constit] | Buf_content],Stack,Infolist,
            NewBuf,NewStk,NewInfo) :-
               process_ip([[[n | Info] | Constit] | Buf_content], Stack,
                          Infolist,NewBuf,NewStk,NewInfo).
```

Figure 4.2: the clause_check predicate

for (and is therefore returned to process_input as an independent phrase, and not as a subcategorized constituent of a larger phrase).

The building clause procedure is triggered by the invocation of the clause_check predicate (see Fig. 4.2). This predicate simply examines the category of the first element of the Buffer and invokes the appropriate process_xp predicate (where xp is either pp, cp or vp). Phrases fronted by nouns cause process_ip to be invoked, phrases fronted by determiners cause an NP to be constructed and then process_ip to be invoked. As we shall see, because all non-NP phrases are processed using the subcategorization technique described above, the important distinction made by clause_check is that between NPs and other phrases, as other phrases are processed almost identically.

Building non-NP Phrases

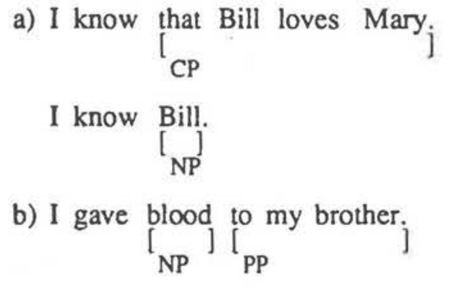
The process_xp predicates (excluding np) all function basically the same way since they are all based on the principle of subcategorization satisfaction. We show process_ip as an example - see Fig. 4.3. All non-subcategorizing elements are processed through a preliminary check to trigger any 'special case scenarios' (see

```
process_ip([[[Cat,Word,EC,ET,RC,RT,Ten,S,[[[]]]] | Constit] | Buf_content],
          Stack,Infolist,NewBuf,NewStk,NewInfo) :-
               word_check([[[Cat,Word,EC,ET,RC,RT,Ten,S,[[[]]]] |
                          Constit] | Buf_content], Stack, Infolist,
                          [First_Buf | Buf1], Stk1, Info1),
               process_ip(Buf1,[First_Buf | Stk1],Info1,
                          NewBuf,NewStk,NewInfo).

process_ip(Buffer,Stack,Infolist,NewBuf,NewStk,NewInfo) :-
               word_check(Buffer, Stack, Infolist, Buffer, Stack, Infolist),
               process_ip2(Buffer,Stack, Infolist, NewBuf, NewStk, NewInfo).

process_ip(Buffer,Stack,Infolist,NewBuf,NewStk,NewInfo) :-
               word_check(Buffer,Stack,Infolist,
                          [First_Buf | Buf1], Stk1, Info1),
               process_ip(Buf1,[First_Buf | Stk1],Info1,
                          NewBuf,NewStk,NewInfo).

process_ip2(Buffer,Stack,Infolist,NewBuf,NewStk,NewInfo) :-
               select_subcat(Buffer,Stack,Infolist,Buf0,NewInfo),
               build_stack(Buf0,Stack,NewBuf,NewStk).
```

Figure 4.3: the `process_ip` predicate

Section 4.5) and then placed on the Stack. Subcategorizing elements are also processed through the preliminary check, and if they do not trigger a 'special case' which causes a change in the contents of the Buffer, Stack or Infolist, they are passed on to `process_ip2`. If the subcategorizer does trigger a 'special case scenario' which does alter the representational structures (such as the insertion of an element into the Buffer), then `process_ip` is invoked again. `process_ip2` drives the satisfaction of the subcategorizing element and then builds the representation of the IP by invoking the build stack procedure.

Satisfying the subcategorization of a subcategorizing element is complicated by two factors: subcategorizers often possibly subcategorize for one of many differ-



a) I know that Bill loves Mary.
[CP]
I know Bill.
[NP]
b) I gave blood to my brother.
[NP] [PP]

Figure 4.4: example subcategorizations

ent specified elements, and some subcategorizers subcategorize for two phrasal elements. For example, the verb "know" subcategorizes for a CP or an NP, while the verb "gave" can subcategorize for an NP and a PP (as well as other possible subcategorizations) (see Fig. 4.4).

When dealing with verbs which subcategorize for more than one possible subcategorization, the parser must select the 'correct' subcategorization for the particular sentence presented (i.e., the subcategorization which corresponds to the phrasal element which is present). This is done by trying to build each possible subcategorization (starting with the first specified) and comparing the completed element with the subcategorization specification[4]. Note that because of the essentially bottom-up nature of this parser, specifying that `process_xp` should be invoked to build the phrase in no way biases the parser towards returning this element[5]. The possible subcategorizations are specified in a predetermined order (PP, IP, CP, NP) to ensure that the subcategorization of a phrasal element is tried before the subcategorization of an element it may contain (i.e., IP tried before NP)[6]. When a match between subcategorizer and the subcategorized (constructed) phrase is found, all other possible subcategorizing specifications are eliminated and processing returns to `process_xp` where a representation of the phrasal element is formed.

Verbs which subcategorize for two elements are handled in essentially the same way (double subcategorizations appear first in the list of possible subcategorizations). Rather than building up one phrasal element and comparing it to the subcategorization specification, two separate build subcategorized element procedures are invoked and both returned elements are compared to the subcategorization. Special processing must also be included in the tree building process in order to recognize a two phrasal element subcategorizer and to form the required tertiary branching structure.

Building NPs

NPs differ from other phrases in that they can include post-noun modifying phrases for which they do not subcategorize[7].

The man in the park...

They also can include any number of these post-noun modifiers. Because they do not subcategorize for their complements, NPs cannot be processed by the general

[4]We will intentionally ignore the effect of moved elements at this point, leaving that discussion for Section 4.3.

[5]This is not entirely accurate. To process CPs which are 'fronted' by complementizers, the parser must be aware that it is trying to process a CP (so that the empty complementizer may be inserted). Thus, we cannot claim that clause processing is entirely bottom-up.

[6]This ordering is used to specify the difference between believe-type verbs and try-type verbs. Believe must subcategorize first for IP, since it Case-marks the subject of the subcategorized phrase. Try must subcategorize for CP first, as we want to 'block' such a Case-marking from occurring.

[7]The same problems affect NP-extraposition, but we will deal only with post-noun modifiers.

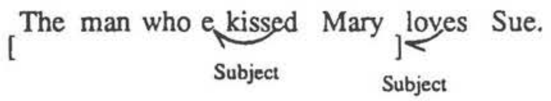


Figure 4.5: Subject/matrix verb connections

subcategorization satisfaction paradigm described above. Instead, we must theorize another method which not only attaches post-noun modifiers to the NP, but that distinguishes where the NP 'ends'

GB theory assumes that all languages have a general descriptor of NP/VP distribution within their grammar. English is known as an SVO language, meaning that all English sentences have the general 'ordering' Subject-Verb-Object. Thus, argument NPs are terminated by either the matrix verb of the phrase, encountering another noun or determiner element, or by the end of the phrase. Determining the end of an object is trivial, the end of an object phrase is the end-of-sentence marker[8]. Subject NPs appear to be much harder to distinguish. The end of a subject NP is signified by the appearance of the matrix verb, but how does the system differentiate between the matrix verb of the 'main' phrase, and the matrix verbs of post-noun modifying phrases (see Fig. 4.5)?

In order to distinguish the matrix verb of the phrase, we must examine the Case- and Θ- assignment structures of the sentence. Within the subject NP, we see that "kissed" assigns internal Case- and Θ- roles to the NP "Mary", and external Θ- and Case-[9] roles to an empty NP which is co-indexed with "who". Thus, we can say that "kissed" is saturated within the post-noun modifying phrase - that is, all of its Case- and Θ- assignments are associated with receivers. Examining the matrix verb "loves", we see that it assigns internal Case- and Θ- roles to the NP "Sue" and external Case- and Θ- roles to the subject NP.

Having examined the Case- and Θ- role assignments of the verbs, we can now explore how the parsing mechanism distinguishes the two (i.e., that one is the verb of a post-noun modifying phrase and the other is the matrix verb of the entire phrase).The first thing we notice is that, because of the left-to-right processing of the parsing mechanism, the parser cannot determine the saturation of internal Case- and Θ- roles since they are assigned to elements which have yet to be constructed. Therefore, internal Case- and Θ- assignment is not significant for identifying the matrix verb. We then examine the external Case- and Θ- assignments of the verbs. The assignments of the post-noun modifying verb are saturated by the subject of this embedded phrase (the empty NP co-indexed with "who"), whereas the assignments

[8]Sentential complements and adverbial information may appear after the object, but they are ruled out by the subcategorization of the matrix verb, or are ambiguous.

[9]Verbs do not assign external Case; but, if they are tensed, we know that an AGR bearing INFL (which assigns external Case to the NP) must immediately precede the verb. Therefore, if a Tense bearing verb is encountered, we know that external Case is assigned.

of the matrix verb are saturated by the entire subject NP (including the post-noun modifying phrase). Therefore, whenever we encounter a verb which cannot be saturated by anything but the subject NP, it must be the matrix verb. Conversely, if we encounter a verb when there are two possible Case- and Θ- receivers, it is the matrix verb of a post-noun modifying phrase.

We capture this generalization within the parser by the predicate `build_rel_clause`, pictured in Fig. 4.6. After constructing the DP consisting of the determiner, the first noun, and all intervening adjectives, then the features of the next element are examined[10]. If the next element is not a noun or determiner, and its the external Case - and Θ- assigning attributes are zero, then a post-noun modifying phrase follows. The modifying phrase is then built using the procedures described previously, it is joined to the DP, and the matrix verb check is done again (see the last instance of the `build_rel_clause` predicate). Note that building the phrase ensures that the Case- and Θ- assignments of the phrasal verb are saturated. When a Case- (actually, a Tense bearing verb) or Θ- assigner is encountered by this check (the first two instances of `build_rel_clause` pictured), the build-NP procedure is completed and the constructed NP is inserted into the front of the Buffer to receive the assignments of the matrix verb.

4.3 Movement

The movement of elements within the sentence to fulfill lexical requirements is controlled by the Empty Category Principle (ECP) which determines the proper configurations under which movement may occur. The ECP is considered to be one of the 'higher level' GB constraints which affects the representation at S-structure or 'above'. As we are postulating a theory which forms correct phrase-structure representations at NP- structure, an alternative to the ECP must be found for controlling movement. The alternative is the newly defined GREPP, repeated below.

Generalized Revised Extended Projection Principle

> Subcategorization requirements must be satisfied by all phrase-structure configurations, where "subcategorization requirements" refers both to subcategorized and subcategorizing elements.

The GREPP states that all subcategorization requirements must be satisfied at all phrase-structure configurations, including NP-structure. The GREPP can be

[10]Adverbial and parenthetical information may intervene between the subject and the matrix verb. Adverbials to the verb are dealt with, in general, by `word_check` and `chk_adv`. Although this has not been done at this point in processing, a check for adverbial information could easily be done at this point. Parenthetical information will, incorrectly, be processed as a post-noun modifier.

```
build_rel_clause([Constit1,[[Cat,Word,EC,ET,RC,RT,+ | Info] | Constit2] |
                  Buf_content],
                 Stack,Infolist,
                 [Constit1,[[Cat,Word,EC,ET,RC,RT,+ | Info] | Constit2] |
                  Buf_content],
                 Stack,Infolist).

build_rel_clause([Constit1,[[Cat,Word,EC,1 | Info] | Constit2] | Buf_content],
                 Stack,Infolist,
                 [Constit1,[[Cat,Word,EC,1 | Info] | Constit2] | Buf_content],
                 Stack,Infolist).

build_rel_clause([Constit1,[[d | Info] | Constit2] | Buf_content],
                 Stack,Infolist,
                 [Constit1,[[d | Info] | Constit2] | Buf_content],
                 Stack,Infolist).

build_rel_clause([Constit1,[[n | Info] | Constit2] | Buf_content],
                 Stack,Infolist,
                 [Constit1,[[n | Info] | Constit2] | Buf_content],
                 Stack,Infolist).

build_rel_clause([Constit1,[end_of_sentence] | Buf_content],
                 Stack,Infolist,
                 [Constit1,[end_of_sentence] | Buf_content],
                 Stack,Infolist).

build_rel_clause([Constit1 | Buf_content],Stack,Infolist,
                 NewBuf,NewStk,NewInfo) :-
                clause_check(Buf_content,Stack,Infolist,
                             Buf1,[Clause | Stk1],Info1),
                [MaxInfo | Rest_con] = Constit1,
                build_rel_clause([[MaxInfo,Constit1,Clause] | Buf1],
                                 Stk1,Info1,NewBuf,NewStk,NewInfo).
```

Figure 4.6: the build_rel_clause predicate

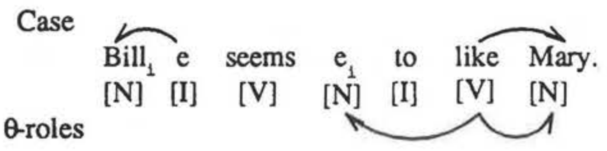

Case
Bill$_i$ e seems e$_i$ to like Mary.
[N] [I] [V] [N] [I] [V] [N]
θ-roles

Figure 4.7: NP movement

viewed as more than just a definition of constraint, but as a definition of the phrase-structure that will be present in a (grammatical) sentence. Thus, if at some point in processing we encounter a subcategorizer which subcategorizes for an element which is not present, then the element must have moved from that position to some leftward position in the sentence[11]. Unfortunately, processing movement is not as simple as this. Since most subcategorizing elements subcategorize for more than one possible element, we cannot blindly assume that a subcategorized element which is not present has simply moved. The element may, in fact, not be present in the sentence at all, and this subcategorization is meant to fail in order that the correct subcategorization may be tried. Therefore, the parser must be able to recognize an element which has moved (without encountering its proper subcategorizer), and record this fact for later use (i.e., when the proper subcategorizer is encountered). In other words, we wish to use a filler-driven rather than gap-driven procedure. Fortunately, it is easy to recognize a moved element - it does not have its Case- and Θ- requirements saturated.

For example, in the sentence:

Bill seems to like Mary.

the NP "Bill" requires both Case- and and Θ- assignment. The verb "seems" is Tensed, and thus is related to an INFL which assigns external Case, but does not assign an external Θ-role. Therefore, "Bill" must have moved from a position which receives a Θ-role (but not a Case-assignment) to its present position in order to receive Case. In this example, the verb "like" assigns an external Θ-role but, because "like" is unTensed, its related INFL "to" does not assign external Case. Thus, we have the configuration shown in Fig. 4.7.

The above is a simple form of movement, but there are a number of 'types'. Movement is normally divided into two main categories: Argument-movement (A-movement), which is the movement of those elements which appear at S-structure or NP-structure in positions which require Case- and Θ- assignments; and non-Argument-movement ($\overline{A}$-movement), which is the movement of WH-elements and topicalized elements.

[11]The direction of movement is always leftward except in 'special cases' such as extraposition and right-node raising.

```
chk_a_move([[[n,Word1,EC1,ET1,1,1,Ten,S | Info1] | Constit1] | Buf_content],
          [A_move,A_type,Scr1,Abar_move,Abar_type,Scr2],
          [[[n,Word1,EC1,ET1,1,1,Ten,NewScr1 | Info1] | Constit1],
           [[v,Word2,EC2,0,RC2,RT2,+ | Info2] | Constit2] | Buf_content2],
          [1,n,NewScr1,Abar_move,Abar_type,Scr2]) :-
               chk_adv(Buf_content,[[[v,Word2,EC2,0,RC2,RT2,+ | Info2] |
                                    Constit2] | Buf_content2]),
               NewScr1 is Scr1 + 1.
```

Figure 4.8: the chk_a_move predicate

4.3.1 A-movement

NP-movement

NP-movement of the type described above requires that any NP which is not in a position in which it receives a Θ-role be flagged for movement. "Flagged for movement" refers to the process of setting a flag value to 1 (from its default of 0) to signify that a moved constituent has been encountered (and the empty category associated with it has yet to be instantiated). The movement flags (one for A-movement and $\overline{A}$-movement respectively) are found in the list structure Infolist, which is passed, along with the Buffer and the Stack, to most predicates. Infolist also contains a subscript counter (to link the flagged constituent to its corresponding empty category) and a type holder which records the category which has moved. The value of the subscript holder will be assigned to the S field in the list of the NP's lexical features, and will be printed out beside the the NP in the printing of the phrase-structure representation. The subscript will also be assigned to, and printed beside, the empty category associated with the movement, producing a visual trace of the movement.

The parser recognizes that movement has occurred, and flags this movement, in the chk_a_move predicate, a sample of which is pictured in Fig. 4.8. If the first word in the Buffer has the categorial features of a noun (i.e., NP, DP, or noun) and requires Case- and Θ- assignment[12], and the following verb/INFL does not assign external Case- and Θ- roles, then movement is flagged. This flag (along with the subscript and type) is 'carried' through the parsing process, until a subcategorizer which subcategorizes for an NP is encountered. At this point, an empty NP is inserted into the Buffer. This insertion, as are all other insertions of empty categories unless specified otherwise, is performed by the predicate insert_move, the relevant section of which is shown in Fig. 4.9.

[12]Pleonastics, such as "it", require only Case-assignment.

```
insert_move(Type,Buffer,Stack,[1,Type,Scr1 | Info_content],
          [[[Type,e,0,0,0,1,*,Scr1,[[[]]]]] | Buffer],
          [0,*,Scr1 | Info_content]).
```

Figure 4.9: the insert_move predicate

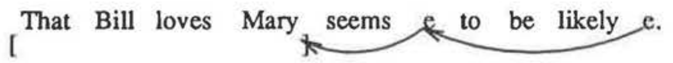


Figure 4.10: Subject Raising

Clausal Movement

The NP movement described above is the simplest form of A-movement; but, NPs are not the only Arguments which can move. Entire phrasal elements, which require at least a Θ-role, can also move in a process called clausal movement (or fronted clauses). Clauses may appear at the front of a sentence where they, obviously, are not subcategorized for.

That Bill loves Mary seems to appear to be obvious.

If a clause is not subcategorized for in the position in which it appears, it must have moved from a position in the sentence where it is subcategorized for. Clausal movement is detected in the parsing mechanism by the predicate `clausal_move`. After a phrasal element has been built and returned to the parse-driving predicate (`process_buffer`), `clausal_move` checks to see if the phrase appeared at the beginning of the sentence (i.e., is a fronted clause). If so, the A-movement flag (part of Infolist) is triggered and the subscript counter, Scr, (used to 'link' the moved constituent to the empty category 'left behind') is advanced. The phrase is also assigned the subscript value, which appears beside the phrasal maximal category in the final printing of the phrase-structure representation. This flag, and subscript, is 'carried' through the parsing process, until a subcategorizer which subcategorizes for the phrasal type is encountered. At this point, an empty category of the phrasal type is inserted into the Buffer.

Subject Raising

Arguments may also 'move through' a number of embedded phrases by a process known as Subject Raising (see example in Fig. 4.10). In the example, the fronted clause has moved from its subcategorized position (attached to "likely") to the front of the sentence. But, the clause is also the subject of the IPs "e to appear" and "e to

```
insert_move(Type,Buffer,Stack,[1,Move_type,Scr1 | Info_content],
            [[[n,e,0,0,0,0,*,Scr1,[[[]]]]] | Buffer],
            [1,Move_type,Scr1 | Info_content]).
```

Figure 4.11: the insert_move predicate

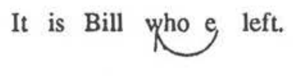


Figure 4.12: WH-movement

be likely". The clause cannot 'stop' its movement in either of these positions since neither "appears" nor "likely" assign an external Θ-role, and the clause requires a Θ-role[13]. Thus, the clause must continue its movement to the right[14] in the sentence, leaving an empty NP (which does not receive Case- or Θ- roles) behind as the subject of the phrases. It is only when the clause encounters a Θ-marked position that the movement can end and an empty category of the movement type (CP in this example) be inserted.

Subject Raising movement is processed by the insert_move predicate pictured in Fig. 4.11. Any position immediately after a subcategorizing element which does not receive a Θ-assignment from verb of the embedded phrase is assigned an empty noun which does not receive Case- or Θ- roles (of course, this only occurs if A-movement has been flagged). The movement flag remains unchanged.

4.3.2 $\overline{A}$-movement

$\overline{A}$-movement differs from A-movement both in how it is flagged and how empty categories are inserted. $\overline{A}$- movement concerns two types of movement, WH-movement and Topicalizations. WH-movement concerns the linking of WH-elements (who, what, when, where) with the appropriate empty position they are referring to. Topicalization is the linking of phrases which are assigned neither Case- nor Θ- roles to their appropriate position.

WH-movement

[13]The verbs "seems" and "appears" assign an internal Θ-role, but these Θ-roles are saturated by the IPs they subcategorize for.

[14]For the purposes of this thesis, movement always occurs in a leftward direction. Here we are talking about the construction of a trace by the parsing mechanism, and this must be done in a rightward direction owing to the left-right processing of the sentence.

```
ins_abar_move([[[i,Word1,1 | Info1] | Constit1],
        [[v,Word2 | Info2] | Constit2] | Buf_content1],
              [[[Cat3,Word3,0,0 | Info3] | Constit3] | Stk_content],
              [A_move,A_type,Scr1,1,Abar_type,Scr2],
              [[[Abar_type,e,0,0,1,1,*,Scr2,[[[]]]]],
               [[i,Word1,1 | Info1] | Constit1],
               [[v,Word2 | Info2] | Constit2],
               [[v,Word4,EC4,1 | Info4] | Constit4] | Buf_content2],
               [A_move,A_type,Scr1,0,*,Scr2]) :-
                have/be_word(Word2),
                chk_adv(Buf_content1,[[[v,Word4,EC4,1 | Info4] |
                            Constit4] | Buf_content2]).

ins_abar_move([[[i,Word1,1 | Info1] | Constit1] | Buf_content1],
              [[[Cat3,Word3,EC3,ET3,0,0 | Info3] | Constit3] | Stk_content],
              [A_move,A_type,Scr1,1,Abar_type,Scr2],
              [[[Abar_type,e,0,0,1,1,*,Scr2,[[[]]]]],
               [[i,Word1,1 | Info1] | Constit1],
               [[v,Word4,EC4,1 | Info4] | Constit4] | Buf_content2],
               [A_move,A_type,Scr1,0,*,Scr2]) :-
                chk_adv(Buf_content1,[[[v,Word4,EC4,1 | Info4] |
                            Constit4] | Buf_content2]).

ins_abar_move(Buffer,Stack,Infolist,Buffer,Infolist).
```

Figure 4.13: the ins_abar_move predicate

WH-movement occurs whenever a WH-element is encountered[15]. Thus, WH-elements can never be Arguments, as they never appear in Argument position. The flagging of WH-movement is trivial - whenever a WH-element is encountered, WH-movement is flagged. Insertion of the appropriate empty category is non-trivial however, as WH-movement can occur from non-subcategorized positions (see Fig. 4.12). Wh-movement can also occur in subcategorized positions, and insertion into these positions is handled the same way as NP-movement (except that the subcategorizer must assign both Case- and Θ- roles). Inserting into an non-subcategorized subject position is performed by the ins_abar_move predicate pictured in Fig. 4.13. If a Tensed verb is encountered which assigns an external Θ-role, and the previous element encountered (which is found on the top of the Stack) does not receive Case; then an empty NP is inserted into the front of the Buffer and $\overline{A}$-movement unflagged.

[15]Special cases of Wh-movement, such as echo questions and multiple WH-questions, are not handled by the parsing mechanism.

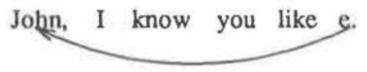


Figure 4.14: topicalization

```
chk_abar_move([[[Cat1,Word,EC,ET,RC,RT,Ten,S | Info1] | Constit1],
               [[Cat2 | Info2] | Constit2] | Buf_content],
              [A_move,A_type,Scr1,Abar_move,Abar_type,Scr2],
              [[[Cat1,Word,EC,ET,RC,RT,Ten,NewScr2 | Info1] | Constit1],
               [[Cat2 | Info2] | Constit2] | Buf_content],
              [A_move,A_type,Scr1,1,n,NewScr2]) :-
                (Cat1 = n;
                 Cat1 = d),
                (Cat2 = n;
                 Cat2 = d),
                NewScr2 is Scr2 + 1.
```

Figure 4.15: the chk_abar_move predicate

Topicalization

Topicalization, the occurrence of an NP[16] which is not assigned Case- nor Θ- roles, occurs when two NPs appear beside each other (see Fig. 4.14). The flagging of Topicalization (as $\overline{A}$-movement) is a relatively straightforward matter. The predicate chk_abar_move (pictured in Fig. 4.15) 'looks for' two NPs occurring beside each other in the Buffer. If this situation is encountered, $\overline{A}$- movement is flagged and processed as described above.

4.4 Building Representational Trees

Once the parsing mechanism has collected all of the elements of a phrase, it must 'form them up' into a representational tree structure. Other than verbs which subcategorize for two elements (which are automatically formed up into a tertiary branching structure dominated by V, in accordance with Percolation Principle I), the process is simply a matter of comparing the two top elements on the stack in accordance with the Percolation Principles. The Percolation Principles determine a dominating node for these two elements. A new element is then constructed consisting of the generated dominating node and the two elements, and this element is then placed onto the top of the stack. Thus, each element is compared with the dominating node of the previous two nodes to form the next dominating node.

[16]Other phrasal types can be topicalized, we deal here with NPs only as an example.

```
chk_tensed_verb(Buffer,
                [[[Cat1 | Info1] | Constit1] | Stk_content],
                [[[i,e,1,0,0,0,*,0,[[[]]]]],
                 [[v,Word2,EC2,ET2,RC2,RT2,+ | Info2] | Constit2] |
                 Buf_content2]) :-
                Cat1 \== v,
                Cat1 \== i,
                chk_adv(Buffer,[[[v,Word2,EC2,ET2,RC2,RT2,+ | Info2] |
                        Constit2] | Buf_content2]).

chk_tensed_verb(Buffer,Stack,Buffer).
```

Figure 4.16: the chk_tensed_verb predicate

4.5 Special Conditions on Processing

A small number of special conditions or circumstances exist within the syntax of English which have not been built into the general processing mechanism of the parser. These conditions tend to be of the form: if a certain condition exists within the Buffer, then perform some action. For example, as has been previously mentioned, all Tensed verbs must be associated with an AGR bearing INFL. Often this INFL element is not present within the sentence, and an empty INFL element must be created (immediately to the left of the verb) to bear AGR. Thus, we have the condition: if a Tensed verb is encountered without an INFL immediately to its left in the Buffer, create an empty INFL element (which assigns external Case) and insert it into the front of the Buffer. This is performed by the predicate `chk_tensed_verb`, pictured in Fig. 4.16.

There are a small number of special case scenarios such as verb/INFL association, and all have the same general form: if X is found in the Buffer, then alter the Buffer in some way. Because these 'special conditions' all have the same general form, they are grouped together 'under' one driving predicate, `word_check`. Whenever `word_check` is invoked (which is every time the system examines a new word in the Buffer), all of the 'special checks' are done. Since all of these checks result in some alteration of the Buffer contents, the parsing mechanism can use a simple comparison to determine if any special cases were encountered and react accordingly (see Section 4.2).

We will now discuss some of the 'special cases', the conditions which trigger them, and their modifications to the Buffer.

`word_check` contains three calls to predicates which deal with the peculiar behaviour of the verbs "be" and "have". Both "be" and "have" can appear as an auxiliary to another verb (for example, "I am sleeping"). Although "be" and "have" are still considered verbs in this instance, they exhibit non-verbal behaviour. Firstly, they 'lose' their subcategorizations, subcategorizations they retain if not followed by another verb. Even more peculiar, if the INFL element associated with the verb

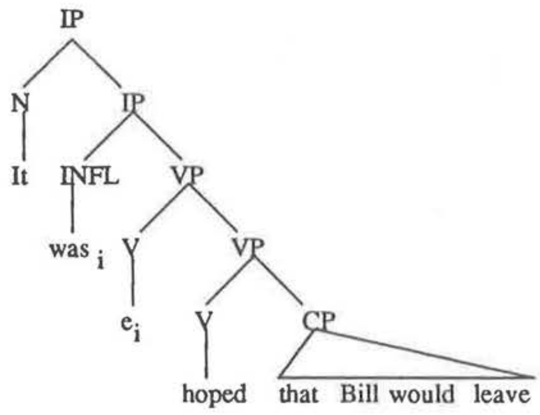


Figure 4.17: aspectual have/be movement to INFL

```
chk_h/b_v([[[v,Word,EC,ET,RC,RT,Ten,S,Subcat] | Constit1],
          [[v | Info2] | Constit2] | Buf_content],
         [[[v,Word,EC,ET,RC,RT,Ten,S,[[[]]]] | Constit1],
          [[v | Info2] | Constit2] | Buf_content]) :-
               have/be_word(Word).

chk_h/b_v(Buffer,Buffer).
```

Figure 4.18: the `chk_h/b_v` predicate

pair is empty, "be" and "have" will 'move into' the empty INFL, leaving behind an empty verb position[17]. Thus, a sentence such as

It was hoped that Bill would leave.

will produce the phrase-structure representation tree pictured in Fig. 4.17.

Although related, the two cases described above are handled by two separate predicates. Elimination of "be" or "have"'s subcategorization is performed by the predicate `chk_h/b_v` pictured in Fig. 4.18. If a "be" or "have" type verb is followed by another verb, the subcategorization of the "be" or "have" type verb is eliminated. The movement of such a verb into an empty INFL element is handled by the predicate `chk_aspec_h/b`, pictured in Fig. 4.19. If a "be" or "have" type verb is encountered which is followed in the Buffer by another verb which is Tensed, and the previous element (found on the top of the Stack) is not an INFL; an INFL is created, the "be" or "have" word moved into it, and an empty V left on the Buffer.

The verb "be" also exhibits peculiar behaviour when coupled with an adjective, as described in Section 3.4. As was previously discussed, in such a case the "be" verb is considered to be a dummy verb, inserted simply to bear Tense (and thus be

[17][Pollock 89] associates this with failure to assign Θ-roles.

```
chk_aspec_h/b([[[v,Word | Info] | Constit1] | Buf_content],
              [Top_Stk | Stack],
              [[[i,Word,1,0,0,0,*,0,[[□]]]],
               [[v,e | Info] | Constit1],
               [[v | Info2] | Constit2] | Buf_content2]) :-
                have/be_word(Word),
                Top_Stk \== i,
                chk_adv(Buf_content,[[[v | Info2] | Constit2] | Buf_content2]).

chk_aspec_h/b([[[v,Word | Info] | Constit1] | Buf_content],
              [Top_Stk | Stack],
              [[[i,Word,1,0,0,0,*,0,[[□]]]],
               [[v,e | Info] | Constit1] | Buf_content]) :-
                have/be_word(Word),
                Top_Stk \== i.

chk_aspec_h/b(Buffer,Stack,Buffer).
```

Figure 4.19: the `chk_aspec_h/b` predicate

linked by predication to an INFL which assigns external Case). In such a case, we must replace the normal subcategorization properties of "be" with the subcategorization properties of the adjective (modelling the Θ-role transmitting properties of a dummy "be" verb).

`word_check` also drives the formation of NPs fronted by determiners. Whenever a determiner is encountered, the `build_np` predicate is immediately invoked, forming up the NP and placing it back into the front of the Buffer.

4.6 An Example Parse

Having detailed the functioning of the parser, we now present an example parse to further explain the parsing procedure. The example will be the same as presented in Section 3.6, except, of course, that we will be presenting a computational solution rather than a 'by hand' parse.

The sentence to be parsed is:

John, I know that Bill likes.

The input sentence is first divided up into individual words which are matched to entries in the lexicon. The corresponding lexical entries are then slotted into a buffer, maintaining the original sentence order.

"John" is the first word to be encountered by the system. Because the first word is a noun, the process `process_ip` predicate is activated which will try to construct an IP (i.e., a sentence). If, for example, a COMP had been encountered first, the system would try to form up a CP, which would then be flagged as a moved phrase fronting a sentence which follows. In the processing of the element "John", the predicate chk_abar_move recognizes the fact that the noun is followed by another noun rather than a Θ-assigner. This is a case of topicalization, therefore the predicate flags an instance of WH-type movement. This is done by setting the variable Abar_move to the value one, the variable Abar_type to "n", and adding one to the value of Scr (initially set to 0). These variables are found in the **Infolist** argument list which is used to flag all movement and 'carry along' necessary information about the movement. The S variable in the element "John"'s lexical features is also set to the new Scr value. This number will be used to link the moved element to its corresponding empty category. "John" is then placed on the stack.

The next element to be encountered is "I". Processing is still being driven by the process_ip predicate and, as "I" neither satisfies the predicate (i.e., a complete sentence formed) nor subcategorizes for another element, it is simply placed upon the stack.

The verb "know" now occupies the front of the buffer. Special processing, which occurs before the general processing of the parser, will recognize "know" as a tensed verb which was not preceded by an AGR bearing INFL (as the top of the stack is not occupied by an INFL). An empty INFL is immediately inserted into the front of the buffer and processing continues as normal. Now when the general processing begins, INFL is the element being processed, as it occupies the front of the buffer. As with "I", INFL neither satisfies the current process nor subcategorizes for another element, therefore it is simply placed on the top of the stack. Note: if we had chosen so, we could now build a partial representation of the phrase-structure tree. INFL assigns Case immediately to its left, therefore by PPII, its features will dominate it regardless of the properties of the next element. Once one dominate node is known, all nodes to the left of it in the representational tree can be resolved. We have chosen not to form a partial tree at this point only to simplify the implementation.

The parser again encounters "know" at the front of the buffer. As an INFL occupies the top of the stack, no new empty INFL will be inserted into the front of buffer. Now as general processing starts, "know" occupies the front of the buffer. Because "know" subcategorizes for a phrasal element, we must resolve which of the possibly many subcategorizations listed as part of the verb's lexical features is appropriate for this sentence. This is done by actually trying to form the appropriate phrasal element for each subcategorization, starting with the first specified in the list of possible subcategorizations. When the parser successfully returns from an attempt to build such a phrase, that subcategorization is selected as the correct one for this sentence and all others are eliminated. In this case, CP is listed first in the subcategorization list so the parser will try to construct a CP with the remaining elements of the buffer. A recursive call is made to the parsing mechanism (in the form of an invocation of process_cp) passing in an empty stack and the partial buffer. The Infolist argument list remains intact.

We begin the building of a CP by looking for a COMP element. In this case "that" is found in the front of the buffer and it is simply moved to the top of the stack. If no COMP element was present, an empty COMP element would have been created and placed on the stack (an action particular only to the build-CP procedure).

"Bill" is now encountered by the parser. This noun neither completes the CP nor subcategorizes for another element, so it is simply placed on the top of the stack.

The verb "likes" is now encountered. As it is a tensed verb with no preceding INFL, an empty INFL is created and placed on the top of the stack, as described above. The verb "likes" is then processed. "likes" subcategorizes for an NP, therefore the parser must satisfy this subcategorization before it can complete the CP (and, in turn, complete the IP). Before this subcategorization satisfaction is attempted (i.e., before the call is made to the appropriate predicate), the predicate `insert_move` is invoked. This predicate (which is invoked for every element processed) usually returns the buffer unchanged, but in this case it finds a match between a moved Case- and Θ- receiver ("John") and a Case- and Θ- assigner without a following receiver ("likes"). This is the condition for a WH-movement 'landing site' and thus corresponds to the topicalized NP movement. An empty category is inserted after "likes" which is assigned the categorial features of the type in `Agr_type` (n). The `S` lexical feature of this empty category is given the value in the `Scr` variable (which will link it to "John"), and `Abar_move` is set to zero, unflagging movement.

The parser now tries to satisfy the NP subcategorization by checking for an NP (or a determiner which fronts an NP) in the next buffer location. It finds the empty NP and halts the recursive descent. We return to `process_cp` which we have now completed by satisfying the subcategorization requirement of the subcategorizer in the phrase. "likes" and the empty NP are then placed upon the stack (already holding, from top to bottom, INFL, "Bill" and "that") and the `build_tree` predicate invoked. This forms up a phrase-structure representation of the CP which provides the crucial dominate element of the phrase which is needed for attachment to the rest of the sentence. The completed CP is then returned as the subcategorized element of "know". "know" and the CP are placed on the stack (containing INFL, "I", and "John") and `build_tree` invoked once again. This builds the phrase-structure of the entire sentence, attaching the already formed CP branch by the domination relationship between "know" and the CP.

The parser has now returned through all levels of recursion. The mechanism checks to see if anything remains in the buffer (certain constructions, such as a cleft, hang from an IP) and, finding it empty, the completed representational tree is passed to `ppxmax` where it is printed.

Chapter 5

Evaluation of Results

In this thesis we have introduced a computational model of an existing linguistic theory which heavily stresses questions of learnability. As such, any evaluation of our model must first focus on the faithfulness of the implementation to the theory. We will then focus on the actual parsing mechanism itself. We have introduced a parsing model which does not use explicit phrase-structure rules to derive phrase-structure representations of sentences. This is a rather radical departure from traditional parsing methods. In this chapter we will also examine some of these more traditional parsing methods and argue on the grounds of epistemological priority that the proposed method is superior. We will then examine a parsing method based on similar principles as ours and compare the two. Finally, we will discuss a natural evolution of the system to the field of language acquisition.

5.1 The Theory and the Implementation

The implementation of Davis' version of GB theory has basically been a process of building a parsing mechanism to drive the activation of the Percolation Principles. The parser must check the word order for correctness, insert any empty categories, and recognize phrasal boundaries; the Percolation Principles then trivially build the phrase structure representations. All parts of this mechanism (other than the very basic left-right nature of the parser) are based upon the linguistic theory, the faithfulness of this implementation being the subject of this section. We will examine the parser as the sum of four parts: the recognition of phrasal boundaries and the correctness of word order, movement (or the insertion of empty categories), the Percolation Principles, and special processing modules as described by the theory.

5.1.1 Building Phrases

As we have seen, the building of phrases is a left-right process of subcategorization fulfillment (for subcategorizers) and Case- and Θ- role fulfillment (NPs)[1]. No actual proposal for a parsing paradigm was presented in Davis' work, not surprisingly as it is a linguistic theory, not one based upon computation; but the in-depth analysis of Case- and Θ- role behaviour had a direct influence upon the parsing procedure developed. This includes the analysis of phrase completion upon Θ-role saturation and NP completion upon encountering an external Θ-assigner which saturates the NP (correct word ordering is also partially checked by subcategorization and also by the NP building process). In fact, Davis' presentation of theory leads one to believe that a right-left implementation is favoured, as the percolation principles construct domination relationships in this manner (for example, see Section 3.6). This strategy can be avoided by noting that PPI and PPII are actually contingent upon a single node (not a pair of nodes), and thus one can form the domination relationship without ever examining the other (rightward) node. This, in turn, allows one to 'form up' the representational phrase-structure tree for all nodes to the left of this point. Using this observation, one can process the sentence left-to-right, constructing a portion of the representational tree whenever a node assigns Case or a Θ-role to an adjacent node[2]. This allows one to avoid placing the entire sentence on the stack and then applying the Percolation Principles (in effect, parsing right-to-left). This means that the stack need not be unlimited and that partial representations of the input sentence are constructed 'on the fly', both important psychological considerations. In actuality, for ease of processing, a compromise approach to this problem was implemented in which entire phrases were placed on the stack before application of the Percolation Principles, allowing for simplified tree building. This approach calls for a possibly unlimited stack, but does not affect the psychological validity of the general approach (as the system could always be altered to perform as described above).

One implementation issue which is not part of Davis' theory at all is the ordering of a subcategorizer's different possible subcategorizations. This ordering (double subcategorizations, then PPs, then IPs, etc.) was imposed when processing showed that certain phrases could be construed as other (incorrect) phrases (eg., a CP with an empty COMP rather than an IP) if ordering was random. Interestingly enough, the imposed ordering works for the lexicon described, except for certain verbs which Case-mark the subject of subcategorized phrases. These verbs, referred to in this thesis as "believe-type verbs", have had their subcategorization ordering altered

[1]It should be pointed out that subcategorization fulfillment is, in actuality, a slightly more abstract form of Case- and Θ- fulfillment. Subcategorizations specify the Case- and Θ- role assignments of a subcategorizer, and one can argue that the specification of a subcategorized phrasal type is simply a special restriction upon these role assignments.

[2]In English, this will apply to cases of a node assigning Case to the left (which is always to an adjacent node), or a Θ-role assignment to the right (again, always to an adjacent node).

so that CP is attempted before IP; and such an alteration functions correctly in this implementation. We do not mean to suggest that such results are a case for subcategorization ordering in human language processing. Indeed, many possible explanations could account for subcategorization selection, including possible parallel implementations which try all possible subcategorizations simultaneously and select the 'best', or certain subcategorization selections being 'ruled out' by further processing (i.e., constraint checking) not implemented here and backtracking to select another subcategorization. We raise this point simply as a part of the discussion of the operation of the parsing mechanism, and as an issue for future study.

5.1.2 Movement

The insertion of empty categories as described in Davis' theory is insufficient for a computational implementation. Davis' method, as best illustrated in the example in Section 3.6, is dependent upon the 'parser' (in Davis' example, the author) knowing the assignment of Θ-roles between sentence elements. When a position in the sentence is encountered which would normally receive a Θ-role, and no Θ-receiver is present, an empty category is inserted. One must know in advance that this position receives a Θ-role. With the elimination of PRO and the relaxation of the Θ-criteria, this is no longer possible for subjects (i.e., positions which are not subcategorized for, but do bear a Θ-role). Now when a Θ-assigner is encountered, we do not necessarily know if a corresponding subject Θ-receiver must be present (and an empty category present if one is not found) or if one of the Θ-receivers receives two Θ-roles. Thus, we cannot blindly insert an empty category whenever a Θ-assigner without receiver is encountered. Instead, we must keep careful track of where Θ-assignments have been made, and what possible 'landing-sites' exist for the movement. This processing is described in Section 4.3 and will not be repeated; we simply wished to point out a difference with Davis' theory.

5.1.3 The Percolation Principles

As has been previously mentioned, the implementation of the Percolation Principles was quite straightforward. As they are simply dominance relationships between two adjacent nodes, they could be implemented as simple comparison features. Only PPIII, which refers to subcategorization and adjunction sets, was difficult to implement, and then only because the **W** specification (the specification of particular syntactic features on a subcategorized head - see Section 4.1) was not implemented[3]. It was discovered during testing of this implementation that, for the lexicon described, PPIII could be expressed simply as a relation between S ($\overline{IP}$) and other phrasal

[3]Specificly, PPIII could not determine the set membership (either subcategorization or adjunction) of a CP headed by an empty complementizer without the feature specification contained in the **W** identifier.

nodes, with S being dominant. The proper implementation of the **W** subcategorization feature would eliminate this somewhat ugly 'hack', and should be included in any future implementations of this theory.

5.1.4 Special Processing

Finally, Davis' theory introduced a few 'special cases' of processing which did not fit into the general functioning of the parser. The insertion of an AGR bearing empty INFL immediately before a Tensed verb (not already associated with an INFL) is such a case. This condition, widely accepted in GB theory, is not triggered by the movement of INFL out of this position (as are empty categories) and thus could not be included as part of the more general movement procedures. Empty INFL insertion must also happen before the processing of the Tensed verb (which would be at the front of the buffer) to allow for Case-assignment to the subject of the sentence, therefore it could not be part of the general processing of the verb (as Case-assignment is a function of the INFL, not the verb). Thus, this case was added to the special processing procedure of the parser, which operates before the more general processing of the buffer.

The verbs "have" and "be" exhibit particular behaviour in that they 'move into' an empty INFL position associated with a Tensed verb following them in the buffer (as described in Section 4.5). We cannot wait to encounter "have" or "be" to perform this function as this would mean altering the contents of the INFL entry which has already been placed on the stack. To do so would mean 'backing up' in the processing (i.e., reprocessing an element we have already finished with), something which should be avoided for reasons of psychological validity. Thus we institute a look-ahead in the buffer to spot the INFL - "have" or "be" - tensed verb combinations and move "have" or "be" during the processing of the INFL. Of course, this requires processing beyond the bounds of the general operation of the parser, which deals with only the first element of the buffer.

The verb "be" also exhibits particular behaviour when coupled with a subcategorizing adjective. As explained in Section 3.4, in this case "be" is acting as a dummy verb (i.e., a verb without subcategorization features) to bear Tense, and thus call for the insertion of an empty INFL if no INFL already exists. We need to recognize that "be" is a dummy verb before encountering the adjective (or else the "be" will be processed as a subcategorizing verb), thus we again implement a look-ahead and perform this check before the general processing of the parser.

5.1.5 Final Notes

As a final note, we wish to identify one other alteration of Davis' theory by this implementation. Some recent GB proposals have been advocating the formation of DP structures to accommodate post-noun modifiers. We follow this analysis of such structures and build them as such, but an NP is still placed at the head of these structures (as opposed to a DP). This was done purely to simplify the implementation (as these DPs are processed the same way as NPs, but differently than determiners) and should not be construed to reflect upon the linguistic theory implemented.

5.2 Principle vs Rule-Based Systems

Despite the increasing popularity of principle-based parsing mechanisms in recent years (see, for example, [Crocker 88], [Barton 84], [Stabler 87]), the dominant modelling strategy for the construction of parsers remains the context-free grammar (CFG) rule-based paradigm. This is understandable considering the (relatively) long tradition of CFG-based models and the existence of well proven implementations. CFG-based models have proven to be at least nearly adequate to perform the traditional task of a grammar (to generate all and only the sentences of a language, and to assign to each sentence its proper structure(s)) and to do it efficiently. Yet, CFG-based models fail as an explanation of linguistic ability.

Firstly, CFG-based models say very little (if anything) about questions of language acquisition. CFG-based models, by emphasizing the configurational aspects of language, emphasize an aspect of language which varies greatly from language to language. There are no context-free rules which are universal [Abney *et al.* 86], therefore language learners are forced to acquire the entire grammar of the language they are exposed to. Given the poverty of stimulus argument (see Section 1.1) and the fact that certain aspects of language are so complex as to be seemingly unlearnable (eg., conditions on long-range quantifier-variable dependencies), this seems highly unlikely, if not impossible. Principle-based systems, on the other hand, posit an innate Universal Grammar (UG) which defines the 'core grammar' for all languages. The same set of grammatical principles apply in every human language, modulo limited parameterization. The process of language acquisition is thus reduced to the setting of certain parameters within UG in accordance to the language of exposure. This explains both the fact that linguistic knowledge is often acquired with insufficient or no evidence, and that certain very complex knowledge is available to the learner. Unfortunately, the exact nature of the parameters and precisely what aspects of language they represent is not precisely known at this time, but an incomplete theory is superior to an inadequate one.

Secondly, CFG-based models say little, if anything, about the ungrammaticality of 'incorrect' sentences. If a rule-based system fails in its parsing, it is always for the same reason - no corresponding phrase-structure rule was found to be consistent with the existing structure. The parser may also return a trace of the representation formed to that point in processing, but it cannot say why the sentence is ungrammatical. Yet for some cases of ungrammaticality, such as subjacency violations, humans are able to interpret an ungrammatical sentence; this implies that the human parser assigns structure to at least some ungrammatical sentences. Again, principle-based parsing provides an explanation for an observed linguistic phenomena such as this. Phrase-structure representations are built up in GB theory at an elementary level, before configurational checks are performed by the 'higher level' modules. The violation of certain of these configurational checks (such as subjacency) will not cause a total parsing failure, but will produce a 'most likely' structure and an explanation of what principle(s) was violated. Language users are not only able to interpret the ungrammatical sentence, they are able to determine why it is ungrammatical.

Finally, context-free grammars presuppose that human language is a context-free construction - but it is not. It has been shown that at least two human languages, Swiss-German and Bambara, are not context-free (in that they contain context sensitive constructions - see [Shieber 85] and [Culy 85] respectively). CFG-based parsers may perform well at providing a broad coverage of linguistic phenomenon, but they do not reflect linguistic competence.

5.3 Principle-Based Systems

Assuming that one has decided to approach human language as a principle-based system, one must decide which 'version' of current linguistic theory to implement. On this subject we will confine our discussion to the topic of this thesis, the generation of phrase-structure configurations. Most systems use a 'straight-ahead' implementation of $\overline{X}$-theory to generate D-structure and then apply Binding, Bounding and the ECP modules to generate movement chains. There are a number of arguments against this method of deriving phrase-structure (see [Davis 87]), but we will present only (in the author's opinion) the most compelling, which deals with epistemological priority.

A conventional $\overline{X}$ schema, such as that suggested by [Chomsky 86b] is:

$$\overline{X} = X\ \overline{\overline{X}}*$$
$$\overline{\overline{X}} = \overline{\overline{X}}*\ \overline{X}$$

where $\overline{\overline{X}}*$ represents zero or more occurrences of some maximal projection and X = X^0. Order is parameterized; the ordering pictured here is for English. This schema is based upon the proposals of [Stowell 81] (pictured in Fig. 5.1), which in turn follow from those originally put forward by [Chomsky 70].

Let us consider the proposals of Fig. 5.1 in terms of epistemological priority. Is there any presyntactic basis for terms such as "head", "maximal projection", "bar level", or even "specifier" or "complement"? The answer is obviously no, thus we must reject $\overline{X}$ theoretic terms as plausible candidates for syntactic primitives. In fact, most of the terms are described in terms of each other, leading to a type of circularity which would be extremely difficult to comprehend for a child trying to 'break into' the linguistic system.

a. Every phrase is endocentric.

b. Specifiers appear at the $\overline{\overline{X}}$ level; subcategorized complements appear within $\overline{X}$.

c. The head always appears adjacent to one of the boundaries of $\overline{X}$.

d. The head term is one bar-level lower then the immediately dominating phrasal node.

e. Only maximal projections may appear as non-head terms within a phrase.

Figure 5.1: $\overline{X}$ schema

Since the definition of $\overline{X}$-theory relies upon syntactic non-primitives, we must conclude that any system of phrase-structure generation which relies upon prelinguistic data has epistemological priority over it. This is exactly what the theory of Davis does and, by fiat, this is exactly what our parser does. By relying upon the syntactically primitive concepts of Case- and Θ- relations to generate phrase-structure, we claim that our parsing mechanism has epistemological priority over those which use $\overline{X}$-theory.

5.4 Discussion of a Similar System

We now turn our attention to a parsing system which embodies many similar aspects of the linguistic theory presented here as Davis' theory. In [Abney *et al.* 86] the authors present a parser which relies upon Case- and Θ- relations in much the same way as our model does. Whereas the parser presented in this thesis relies upon the fulfillment of subcategorization specifications, their parsing mechanism is based upon the saturation of Θ-roles (both external and internal); every Argument must be linked to a Θ-assigner, every Θ-assigner to an Argument. Thus, when an Argument is encountered, the mechanism searches to the left or the right looking for a Θ-assigner. When a Θ-assigner is encountered, a Θ-receiver must be located for it. A failure to locate a Θ-assigner for a receiver or vice-versa is recorded as a failure (i.e., an ungrammatical sentence). Once the relationships between Θ-assigners and receivers have been established (called licensing relations), $\overline{X}$-theory and government are used to establish the proper bar-levels, dominance relations, and attachment to the representation tree.

Movement in this parsing mechanism is recorded in a way similar to that of the parser described in this thesis. When a licensing check fails to find a Θ-assigner for a receiver, or encounters an Operator, movement is flagged. Empty categories are inserted by another failure in licensing (this time a failure to find a Θ-receiver for an assigner), and linking of the empty category to the phrase which has moved is done by a procedure which checks that an excessive number of barriers are not crossed.

Abney and Cole's parser can be seen as a rather elementary implementation of many of the concepts described in this thesis (although, of course, it was based on the work of Abney himself). The parser described in this thesis did not follow the licensing, Θ-fulfillment paradigm for two reasons. Firstly, Abney and Cole's parser ignores the fact that the same word can subcategorize for (and thus assign Θ-roles to) different complements. As discussed in Section 4.2.1, this variability can be quite severe and have a major impact on the processing of the sentence. Secondly, use of Abney-type licensing conditions was made impossible in our framework by the elimination of PRO and the subsequent relaxation of the Θ-criterion. Abney-type licensing will only work if one subscribes to a strong Θ-criterion. The adoption of a Θ-criterion which allows Θ-receivers to receive more than one Θ-role makes a matching paradigm impossible.

There are a number of other differences between the two parsing mechanisms, the construction of the phrase-structure representation being the predominant. Abney and Cole still make use of $\overline{X}$-theory to form the representational tree, using a Θ-licensing relation as a simple constraint test for grammaticality. We have eliminated $\overline{X}$-theory all together, using the Θ- (and Case) relations themselves as the basis for forming representational trees. Once again, we must claim epistemological priority over a parsing mechanism which relies upon $\overline{X}$-theory.

5.5 Future Issues

As we have discussed before, there are distinct forms of government within the linguistic theory we are working with. Θ-government, and the slightly less restricted relation which characterizes internal Case-assignment, constitute the most restricted forms of government. The assignment of external Case- and Θ- roles is governed by a much 'looser' form of government, based upon the concept of absolute barriers to upward government. We have also discussed how the Percolation Principles, which determine phrase-structure, are actually non-primitive concepts which are based upon the assignment of Case- and Θ- roles. Given this theoretical basis, Davis proposes that the process of language acquisition is the process of progression from a grammar ruled by only very strict forms of government through to a grammar determined by more general (i.e., both restricted and less restricted) forms of government; and that this gradual relaxation of government corresponds to a 'grammaticalization' of the child's language. Along with this process of government development, Davis posits the setting of two parameters to determine the ergativity of a language (the 'richness' of morphological Case-systems, whether inflectionally-rich or inflectionally-deficient). As the strictness of government is 'loosened', and as the parameters are set, the Percolation Principles should develop naturally; allowing the child to form more complex phrase-structure representations, and thus progress towards a mature grammar.

Davis' proposals for an acquisition system are much more detailed than described above and are beyond the scope of this thesis. Our purpose here is to simply introduce the idea of an acquisition system and to make a few 'bold-faced' predictions on how it may be done. The obvious first step is to reduce the Percolation Principles (at least PPI and PPII) to a state based more upon government relations and assignments, and less upon explicit ordering (as different languages assign Case- and Θ- roles in different directions). These new "statements of configurations" should allow the acquisition system to form elementary (i.e., highly restricted) configurational structures, and eventually to generalize to Percolation Principles configured according to the language of exposure.

Secondly, a language generation system based upon the child's hypothesized 'primitive' conception of language (i.e., arguments and subcategorizers linked by Θ-roles) would be an interesting development. Given this 'primitive' conceptualization of language, we would then monitor the generation of sentences in a gradually developing system. Would these sentences correspond to observed child linguistic development? This would be the true test of the system.

Chapter 6

Conclusion

This thesis has presented a system for building phrase-structure representations without the use of explicit phrase-structure rules. Specifically, we have developed a system which generates phrase-structure based on the Case- and Θ- relationships between component words of a sentence, as well as the subcategorization features of subcategorizing elements. As such, the system can be seen as a principle-based parser, albeit an unusual deviation from the 'standard' approach.

The derivation of phrase-structure from primitive components (without any rules) is a rather radical departure from previous syntactic parsing systems which relied upon either explicit lists of rules (eg., [Woods 70], [Pereira *et al.* 80], [Marcus 80]) or the very generalized phrase-structure rules of $\overline{X}$-theory (eg., [Crocker 88], [Thiersch *et al.* 89;91], [Dorr 87]). We have argued that the elimination of phrase-structure rules (which are purely linguistic in nature) in favor of a system which relies upon prelinguistic primitives demonstrates epistemological priority and therefore such a system is preferred on cognitive grounds.

The described system accounts for a significant subset of the English grammar, including aspectual have/be movement to INFL, argument movement, $\overline{A}$-movement, subject raising, topicalization, and some relative clauses (including stacked relative clauses). The system has difficulty handling some forms of post-noun modifiers[1], sentential adverbs, and a number of difficult linguistic phenomenon such as rightward movement, multiple WH questions, etc. which are seen as problematic in GG theory. Also, the present system makes no claim to recognizing the ungrammaticality of ungrammatical input. Previous approaches to the implementation of principle-based systems have relied upon the 'higher level' components of GB theory to act as constraints upon generated phrase-structure representations (eg., [Sharp 85], [Dorr 87]). In other words, phrase-structure is generated first, and then grammaticality checks are performed. The system presented in this thesis is only for the generation of phrase-structure, the 'higher level' constraint checks would have to be added to

[1]Problems with the processing of post-noun modifiers arises from the ambiguity of phrase attachment within the clauses and can only be resolved through the use of pragmatic information.

the system to produce a robust parser. Since the phrase-structure generated by this system is virtually identical to that generated by existing principle-based systems, and as we posit no alteration to higher-level constraint checking, it was determined to be unnecessary to implement constraint checking modules which have already been demonstrated by other systems. The addition of such modules would produce a system which is at least as robust as the system from which the modules were taken.

The construction of a principle-based system for generating phrase-structure from prelinguistic primitives is relevant to a number of disciplines. As a model of the human language faculty, it is of direct interest to linguistics and cognitive science. As an implementation of a proposed linguistic theory, it provides credence to the theory while, at the same time, suggesting some slight alterations. As a parser which relies upon prelinguistic primitives which are available to very young children, the system presents itself as a natural 'building block' in the development of a model of language acquisition. Finally, as a natural language parser, the presented system provides an efficient and elegant technique for the generation of phrase structure, which is of interest to the fields of computational linguistics and artificial intelligence.

Chapter 7

References

[Abney *et al.* 81] Steven Abney and Jennifer Cole, *A Government-Binding Parser*, unpublished manuscript, MIT.

[Abney 85] S. Abney, *Functor Theory and Licensing: Toward an Elimination of the Base Component* , ms., MIT, Cambridge, MA.

[Abney 87] S. Abney, *The English NP in its Sentential Aspect*, MIT PhD dissertation.

[Anderson 77] John R. Anderson, "Induction of Augmented Transition Networks", in *Cognitive Science* 1: 125-157.

[Aoun *et al.* 83] Y. Aoun and D. Sportiche, "On the Formal Theory of Government", in *The Linguistic Review* 2: 211-236.

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

[Berwick 82] Robert C. Berwick, *Locality Principles and the Acquisition of Syntactic Knowledge* , PhD dissertation, MIT Department of Computer Science and Electrical Engineering.

[Berwick 85] Robert C. Berwick, *The Acquisition of Syntactic Knowledge* , The MIT Press.

[Berwick *et al.* 84] Robert Berwick and Amy Weinberg, *The Grammatical Basis of Linguistic Performance* , The MIT Press.

[Brown *et al.* 70] R. Brown and C. Hanlon, "Derivational Complexity and Order of Acquisition in Child Speech", J.R. Hayes, ed.

[Collins Hill 83] Jane Anne Collins Hill, *A Computational Model of Language Acquisition in the Two Year Old* , PhD dissertation, University of Massachusetts.

[Chomsky 70] Noam Chomsky, "Remarks on Normalization", in *Readings in English Transformational Grammar*, R. Jacobs and P. Rosenbaum, eds. Waltham, MA: Ginn and Co., pp. 184-221.

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

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

[Chomsky 86b] Noam Chomsky, *Barriers* , The MIT Press.

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

[Crocker 88] Matthew W. Crocker, *A Principle-Based System for Natural Language and Translation* , TR-88-18, University of British Columbia.

[Culy 85] C. Culy, "The Complexity of the Vocabulary of Bambara", in *Linguistics and Philosophy* 8: 345-351.

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

[Davis 89] Henry Davis, *Basic Concepts of Government-Binding Theory*, unpublished manuscript, UBC.

[Dorr 87] Bonnie Dorr, *UNITRAN: A Principle-Based Approach to Machine Translation* , ms., MIT, Cambridge, MA.

[Emonds 85] J. Emonds, *A Unified Theory of Syntactic Categories* , Foris Publications, Dordrecht.

[Fukui *et al.* 86] N. Fukui and M. Speas, "Specifiers and Projections", in *MIT Working Papers in Linguistics* 8.

[Gold 67] E. Gold, "Language Identification in the Limit", *Information and Control* 16: 447-474.

[Higginbotham 86] J. Higginbotham, *Elucidation of Meaning* , ms., MIT, Cambridge, MA.

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

[Hornstein *et al.* **81]** Norbert Hornstein and David Lightfoot, *Explanation in Linguistics. The Logical Problem of Language Acquisition* , Longman Group Limited.

[Huang 82] C.-T. J. Huang, *Logical Relations in Chinese and the Theory of Grammar*, unpublished MIT PhD dissertation.

[Hyams 86] Nina Hyams, *Language Acquisition and the Theory of Parameters* , D. Reidel Publishing Company.

[Kayne 84] R. Kayne, *Correctedness and Binary Branching* , Foris Publications, Dordrecht.

[Kelly 67] K.L. Kelly, *Early Syntactic Acquisition* , PhD dissertation, University of California at Los Angeles.

[Koster *et al.* **82]** J. Koster and R. May, "On the Constituency of Infinitives", *Language* 58: 116-143.

[Lasnik *et al.* **88]** Howard Lasnik and Juan Uriagereka, *A Course in GB Syntax* , The MIT Press.

[Lloyd 87] John W. Lloyd, *Foundations of Logic Programming* , Springer-Verlag, 2nd edition.

[Marcus 80] Mitchell Marcus, *A Theory of Syntactic Recognition for Natural Language* , The MIT Press.

[Newport *et al.* **77]** E. Newport, H. Gleitman and L. Gleitman, "Mother, I'd Rather Do It Myself, Some Effects and Noneffects of Maternal Speech Style", in *Talking to Children: Language Input and Acquisition* , C. Snow and C. Ferguson, eds., Cambridge University Press, Cambridge.

[Otsu *et al.* **83]** Y Otsu, H. van Riemsdijk, K. Inoue, A. Kamio, and N. Kawasaki eds., *Studies in Generative Grammar and Language Acquisition. A Report on Recent Trends in Linguistics* , Editorial Committee, Tokyo 1983.

[Pereira *et al.* **80]** F.C.N. Pereira, D.H.D. Warren, "Definite Clause Grammars for language analysis", in *Artificial Intelligence* , 13: 231-278.

[Pollock 89] "Verb Movement, Universial Grammar, and the Structure of IP", in *Linguistic Inquiry* 20(3): 365-425.

[Postal 69] P. Postal, "Anaphoric Islands" in *Chicago Linguistics Society* 5: 205-239.

[Pullum 85] G. Pullum, "Assuming Some Version of X-Bar Theory", in *Papers from the General Session of the Twenty-First Regional Meeting of The Chicago Linguistics Society* , University of Chicago.

[Radford 81] A. Radford, *Transformational Syntax* , Cambridge University Press, Cambridge.

[Roeper *et al.* **85]** Thomas Roeper and Edwin Williams eds., *Parameter Setting* , D. Reidel Publishing Company.

[Ross 67] J.R. Ross, *Constraints on Variables in Syntax* , unpublished MIT PhD dissertation.

[Rothstein 83] S. Rothstein, *The Syntactic Form of Predication* , unpublished MIT PhD dissertation.

[Sampson 83] G. Sampson, "Deterministic Parsing", in *Parsing Natural Language* , M. King ed. Academic Press, pp. 91-116.

[Sells 85] Peter Sells, *Lectures on Contemporary Syntactic Theories* , Center for the Study of Language and Information.

[Selfridge 80] Mallory Selfridge, *A Process Model of Language Acquisition* , Research Report 172, Yale University, Department of Computer Science.

[Selfridge 86] Mallory Selfridge, "A Computer Model of Child Language Learning", in *Artificial Intelligence* 29: 171-216.

[Shieber 85] S. Shieber, "Evidence Against the Context-Freeness of Natural Language", in *Linguistics and Philosophy* 8: 333-343.

[Speas 86] M. Speas, *Adjunctions and Projections in Syntax* , unpublished MIT PhD dissertation.

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

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

[Stowell 81] T. Stowell, *Origins of Phrase Structure* , unpublished MIT PhD dissertation.

[**Thiersch** *et al.* **89;91**] Craig Theirsch and Hans-Peter Kolb, "Levels and Empty Categories in a Principles and Parameters Approach to Parsing", technical report, Universiteit Brabank, Tilburg [89]; also to appear in *Representational and Derivational Approaches to Generative Grammar* , H. Haider and K. Netter eds., Dordrecht: Reidel [91].

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

[**Wexler** *et al.* **80**] Kenneth Wexler and Peter Culicover, *Formal Principles of Language Acquisition* , The MIT Press.

[**White 82**] Lydia White, *Grammatical Theory and Language Acquisition* , Foris Publications.

[**Woods 70**] W. Woods, "Transition Network Grammars for Natural Language Analysis", in *CACM* 13(10): 591-606.

Appendix A

Sample Parses

1. A very simple sentence consisting of a noun and a Tensed intransitive verb.

```
|: bill slept.

0: infl
   1: n: bill
   1: infl
      2: infl: e
      2: v: slept
```

2. An example of transitive verb usage.

```
|: bill kissed mary.

0: infl
   1: n: bill
   1: infl
      2: infl: e
      2: v
         3: v: kissed
         3: n: mary
```

3. Example of a simple NP (the man).

```
|: the man kissed mary.

0: infl
   1: n
      2: d: the
      2: n
         3: n: man
   1: infl
      2: infl: e
      2: v
         3: v: kissed
         3: n: mary
```

4. A verb which subcategorizes for a PP.

```
|: the man sat on a chair.

0: infl
   1: n
      2: d: the
      2: n
         3: n: man
   1: infl
      2: infl: e
      2: v
         3: v: sat
         3: p
            4: p: on
            4: n
               5: d: a
               5: n
                  6: n: chair
```

5. A verb which subcategorizes for two Objects.

```
|: the man gave mary a book.

0: infl
   1: n
      2: d: the
      2: n
         3: n: man
   1: infl
      2: infl: e
      2: v
         3: v: gave
         3: n: mary
         3: n
            4: d: a
            4: n
               5: n: book
```

6. An example of be/adjective feature passing.

```
|: i am fond of asparagus.

0: infl
   1: n: i
   1: infl
      2: infl: am
      2: v
         3: v: e
         3: adj
            4: adj: fond
            4: p
               5: p: of
               5: n: asparagus
```

7. Aspectual have/be movement into the empty INFL, leaving behind an empty Verb.

```
|: it was hoped that bill would leave.

0: infl
   1: n: it
   1: infl
      2: infl: was
      2: v
         3: v: e
         3: v
            4: v: hoped
            4: c
               5: c: that
               5: infl
                  6: n: bill
                  6: infl
                     7: infl: would
                     7: v: leave
```

8. Insertion of an empty COMP into the head position of a phrase.

```
|: i tried to leave.

0: infl
   1: n: i
   1: infl
      2: infl: e
      2: v
         3: v: tried
         3: c
            4: c: e
            4: infl
               5: infl: to
               5: v: leave
```

9. An Adverb intervening between a Verb and its associated (Tense bearing) INFL.

```
|: i quietly asked bill to leave.

 0: infl
    1: n: i
    1: infl
       2: infl: e
       2: v
          3: v
             4: adv: quietly
             4: v: asked
          3: n: bill
          3: c
             4: c: e
             4: infl
                5: infl: to
                5: v: leave
```

10. NP movement.

```
|: bill seems to like mary.

 0: infl
    1: n: bill (1)
    1: infl
       2: infl: e
       2: v
          3: v: seems
          3: infl
             4: n: e (1)
             4: infl
                5: infl: to
                5: v
                   6: v: like
                   6: n: mary
```

11. The PP is an adjunct to the sentence.

```
|: bill was seen by mary.

 0: infl
    1: infl
       2: n: bill (1)
       2: infl
          3: infl: was
          3: v
             4: v: e
             4: v
                5: v: seen
                5: n: e (1)
    1: p
       2: p: by
       2: n: mary
```

12. Argument movement.

```
|: a book was given to mary.

 0: infl
    1: n (1)
       2: d: a
       2: n
          3: n: book
    1: infl
       2: infl: was
       2: v
          3: v: e
          3: v
             4: v: given
             4: n: e (1)
             4: p
                5: p: to
                5: n: mary
```

13. Clausal movement.

```
|: that bill loves mary i know.

 0: infl
    1: c (1)
       2: c: that
       2: infl
          3: n: bill
          3: infl
             4: infl: e
             4: v
                5: v: loves
                5: n: mary
    1: infl
       2: n: i
       2: infl
          3: infl: e
          3: v
             4: v: know
             4: c: e (1)
```

14. $\bar{A}$-movement, in this case topicalization.

```
|: john i know loves mary.

 0: infl
    1: n: john (1)
    1: infl
       2: n: i
       2: infl
          3: infl: e
          3: v
             4: v: know
             4: c
                5: c: e
                5: infl
                   6: n: e (1)
                   6: infl
                      7: infl: e
                      7: v
                         8: v: loves
                         8: n: mary
```

15. Subject raising through intervening phrases.

```
|: that bill loves mary seems to appear to be likely.

 0: infl
    1: c (1)
       2: c: that
       2: infl
          3: n: bill
          3: infl
             4: infl: e
             4: v
                5: v: loves
                5: n: mary
    1: infl
       2: infl: e
       2: v
          3: v: seems
          3: infl
             4: n: e (1)
             4: infl
                5: infl: to
                5: v
                   6: v: appear
                   6: infl
                      7: n: e (1)
                      7: infl
                         8: infl: to
                         8: infl
                            9: infl: be
                            9: v
                               10: v: e
                               10: v
                                   11: v: likely
                                   11: c: e (1)
```

16. WH-movement, an example of $\bar{A}$-movement.

```
I: who did you say saw bill?

 0: c
    1: c: who (1)
    1: infl
       2: infl: did
       2: infl
          3: n: you
          3: infl
             4: infl: e
             4: v
                5: v: say
                5: c
                   6: c: e
                   6: infl
                      7: n: e (1)
                      7: infl
                         8: infl: e
                         8: v
                            9: v: saw
                            9: n: bill
```

17. WH-movement within a post-noun modifying phrase.

```
I: the man who bill saw left.

 0: infl
    1: n
       2: d: the
       2: n
          3: n
             4: n: man
          3: c
             4: c: who (1)
             4: infl
                5: n: bill
                5: infl
                   6: infl: e
                   6: v
                      7: v: saw
                      7: n: e (1)
    1: infl
       2: infl: e
       2: v: left
```

18. A relative clause (the man in the park).

```
|: the man in the park kissed mary.

 0: infl
    1: n
       2: d: the
       2: n
          3: n
             4: n: man
          3: p
             4: p: in
             4: n
                5: d: the
                5: n
                   6: n: park
    1: infl
       2: infl: e
       2: v
          3: v: kissed
          3: n: mary
```

19. The following two examples highlight the problem of building relative clauses - where to join the constituent clauses. The clauses can be joined so that each clause dominates all following clauses (as we have done), or they can be constructed so that subordinate clauses are all attached to the 'head' up. The only way to determine which is correct for a given sentence is by pragmatic information, which is clearly beyond the abilities of a purely syntactic parser. So, we have arbitrarily chosen to construct relative clauses by attaching to the immediately dominating clause, while recognizing that this will often times generate incorrect parses.

```
|: the man in the park with the big trees kissed the girl.

 0: infl
    1: n
       2: d: the
       2: n
          3: n
             4: n: man
          3: p
             4: p: in
             4: n
                5: d: the
                5: n
                   6: n
                      7: n: park
                   6: p
                      7: p: with
                      7: n
                         8: d: the
                         8: n
                            9: n
                               10: adj: big
                               10: n: trees
    1: infl
       2: infl: e
       2: v
          3: v: kissed
          3: n
             4: d: the
             4: n
                5: n: girl
```

20. An example of an incorrectly parsed relative clause; incorrectly parsed for the reasons stated above.

```
|: the man in the park with the little boy kissed the girl.

 0: infl
    1: n
       2: d: the
       2: n
          3: n
             4: n: man
          3: p
             4: p: in
             4: n
                5: d: the
                5: n
                   6: n
                      7: n: park
                   6: p
                      7: p: with
                      7: n
                         8: d: the
                         8: n
                            9: n
                               10: adj: little
                               10: n: boy
    1: infl
       2: infl: e
       2: v
          3: v: kissed
          3: n
             4: d: the
             4: n
                5: n: girl
```

21. A stacked relative clause.

```
|: the man who i liked who had a red car kissed the girl.

 0: infl
    1: n
       2: d: the
       2: n
          3: n
             4: n
                5: n: man
             4: c
                5: c: who (1)
                5: infl
                   6: n: i
                   6: infl
                      7: infl: e
                      7: v
                         8: v: liked
                         8: n: e (1)
          3: c
             4: c: who (2)
             4: infl
                5: n: e (2)
                5: infl
                   6: infl: had
                   6: v
                      7: v: e
                      7: n
                         8: d: a
                         8: n
                            9: n
                               10: adj: red
                               10: n: car
    1: infl
       2: infl: e
       2: v
          3: v: kissed
          3: n
             4: d: the
             4: n
                5: n: girl
```

Appendix B

The Prolog Code

```
%-------------------------------------------------------------------------
% Section: Process Input
%
% Paradigm: process_input(Input,Tree)
%
% Description: Constructs a representational tree of the inputted Buffer.
%

process_input(Input,Tree) :-
        process_buffer(Input,[[bot_of_stack]],
                             [0,*,0,0,*,0],Buf,Tree,Infolist).


%-------------------------------------------------------------------------
% process_buffer(Buffer,Stack,Infolist,NewBuffer,NewStack,NewInfolist)
%
% Builds the phrasal type represented by the front of the Buffer,
% checks the phrase for movement (clausal movement), and recursively
% calls itself if the Buffer isn't empty.
%

process_buffer([[end_of_sentence]],Stack,Infolist,
               [[end_of_sentence]],Stack,Infolist).

process_buffer(Buffer,Stack,Infolist,NewBuf,NewStk,NewInfo) :-
                clause_check(Buffer,Stack,Infolist,Buf2,[Con1|Stk2],Info2),
                clausal_move([Con1|Buf2],Stk2,Info2,[Con2|Buf3],Info3),
                process_buffer(Buf3,[Con2|Stk2],Info3,
                               NewBuf,NewStk,NewInfo).

%-------------------------------------------------------------------------
% clausal_move(Buffer,Stack,Infolist,NewBuffer,NewInfolist)
%
% Flags clausal movement and sets the subscript values.
%
```

```
clausal_move([Constit1,[end_of_sentence] | Buf_content],Stack,Infolist,
             [Constit1,[end_of_sentence] | Buf_content],Infolist).

clausal_move([[[c,max,EC,ET,RC,1,Ten,S | Info] | Constit] | Buf_content],
             [[bot_of_stack]],
             [A_move,A_type,Scr1,Abar_move,Abar_type,Scr2],
             [[[c,max,EC,ET,RC,0,Ten,NewScr1 | Info] | Constit] | Buf_content],
             [1,c,NewScr1,Abar_move,Abar_type,Scr2]) :-
                NewScr1 is Scr1 + 1.

clausal_move([[[p,max,EC,ET,RC,1,Ten,S | Info] | Constit] | Buf_content],
             [[bot_of_stack]],
             [A_move,A_type,Scr1,Abar_move,Abar_type,Scr2],
             [[[p,max,EC,ET,RC,0,Ten,NewScr1 | Info] | Constit] | Buf_content],
             [1,p,NewScr1,Abar_move,Abar_type,Scr2]) :-
                NewScr1 is Scr1 + 1.

clausal_move(Buffer,Stack,Infolist,Buffer,Infolist).
```

```
%---------------------------------------------------------------------------
% Section: Word Check
%
% Paradigm: word_check(Buffer,Stack,Infolist,NewBuffer,NewStack,NewInfolist)
%
% Description: This module performs the preliminary checks on the contents
%              of the buffer; the 'special processing.
%

word_check(Buffer,Stack,Infolist,NewBuf,Stack,NewInfo) :-
                chk_det(Buffer,Infolist,Buf0,Info1),
                chk_h/b_v(Buf0,Buf1),
                chk_be_adj(Buf1,Buf2),
                chk_aspec_h/b(Buf2,Stack,Buf3),
                chk_tensed_verb(Buf3,Stack,Buf4),
                chk_a_move(Buf4,Info1,Buf5,Info2),
                chk_abar_move(Buf5,Info2,Buf6,Info3),
                ins_abar_move(Buf6,Stack,Info3,Buf7,NewInfo),!,
                chk_adv(Buf7,NewBuf).

%---------------------------------------------------------------------------
% chk_det(Buffer,Infolist,NewBuffer,NewInfolist)
%
% Builds an NP (should be a DP for theoretical correctness) if a determiner
% is found in the front of the Buffer. The constructed NP is then inserted
% into the front of the Buffer.
%

chk_det([[[d | Info] | Constit] | Buf_content],Infolist,
        NewBuf,NewInfo) :-
                build_np([[[d | Info] | Constit] | Buf_content],
                         [[bot_of_stack]],Infolist,NewBuf,NewStk,NewInfo).

chk_det(Buffer,Infolist,Buffer,Infolist).


%---------------------------------------------------------------------------
% chk_h/b_v(Buffer,NewBuffer)
%
% Strips "have"- or "be"- type verbs of their subcategorizing features if
% they are followed by another verb.
%

chk_h/b_v([[[v,Word,EC,ET,RC,RT,Ten,S,Subcat] | Constit1],
          [[v | Info2] | Constit2] | Buf_content],
         [[[v,Word,EC,ET,RC,RT,Ten,S,[[[]]]] | Constit1],
          [[v | Info2] | Constit2] | Buf_content]) :-
                have/be_word(Word).

chk_h/b_v(Buffer,Buffer).
```

```
%---------------------------------------------------------------------------
% chk_be_adj(Buffer,NewBuffer)
%
% Eliminates the subcategorizing features of "be"-type verbs if they are
% followed by a subcategorizing adjective.
%

chk_be_adj([[[i,Word | Info1] | Constit1],
            [[v,e,EC2,ET2,RC2,RT2,Ten2,S2,Sub2] | Constit2],
            [[adj | Info3] | Constit3] |
            Bufcontent],
            [[[i,Word | Info1] | Constit1],
            [[v,e,EC2,ET2,RC2,RT2,Ten2,S2,[[[]]]] | Constit2],
            [[adj | Info3] | Constit3] |
            Bufcontent]) :-
                be_word(Word).

chk_be_adj([[[v,Word,EC2,ET2,RC2,RT2,Ten2,S2,Sub2] | Constit2],
            [[adj | Info3] | Constit3] |
            Bufcontent],
            [[[v,Word,EC2,ET2,RC2,RT2,Ten2,S2,[[[]]]] | Constit2],
            [[adj | Info3] | Constit3] |
            Bufcontent]) :-
                be_word(Word).

chk_be_adj(Buffer,Buffer).


%---------------------------------------------------------------------------
% chk_aspec_h/b(Buffer,Stack,NewBuffer)
%
% Moves a "have"- or "be"- type verb into an empty INFL position, which
% it creates (given that no INFL is already present).
%

chk_aspec_h/b([[[v,Word | Info] | Constit1] | Buf_content],
              [Top_Stk | Stack],
               [[[i,Word,1,0,0,0,*,0,[[[]]]]],
              [[v,e | Info] | Constit1],
              [[v | Info2] | Constit2] | Buf_content2]) :-
               have/be_word(Word),
               Top_Stk \== i,
               chk_adv(Buf_content,[[[v | Info2] | Constit2] | Buf_content2]).

chk_aspec_h/b([[[v,Word | Info] | Constit1] | Buf_content],
              [Top_Stk | Stack],
               [[[i,Word,1,0,0,0,*,0,[[[]]]]],
              [[v,e | Info] | Constit1] | Buf_content]) :-
               have/be_word(Word),
               Top_Stk \== i.
```

```
chk_aspec_h/b(Buffer,Stack,Buffer).


%-------------------------------------------------------------------------
% chk_tensed_verb(Buffer,Stack,NewBuffer)
%
% Inserts an empty INFL element into the front of the Buffer upon
% encountering a Tense bearing verb, provided no INFL already immediately
% preceeds the verb.
%

chk_tensed_verb(Buffer,
                [[[Cat1 | Info1] | Constit1] | Stk_content],
                [[[i,e,1,0,0,0,*,0,[[[]]]]],
                 [[v,Word2,EC2,ET2,RC2,RT2,+ | Info2] | Constit2] |
                 Buf_content2]) :-
                Cat1 \== v,
                Cat1 \== i,
                chk_adv(Buffer,[[[v,Word2,EC2,ET2,RC2,RT2,+ | Info2] |
                        Constit2] | Buf_content2]).

chk_tensed_verb(Buffer,Stack,Buffer).


%-------------------------------------------------------------------------
% chk_a_move(Buffer,Infolist,NewBuffer,NewInfolist)
%
% Flags argument movement and sets the subscript values to be used in
% creating a movement trace.
%

chk_a_move([[[n,Word1,EC1,ET1,1,1,Ten,S | Info1] | Constit1],
            Infl_constit,
            [[v | Info2] | Constit2] | Buf_content1],
           [A_move,A-Type,Scr1,Abar_move,Abar_type,Scr2],
           [[[n,Word1,EC1,ET1,1,1,Ten,NewScr1 | Info1] | Constit1],
            Infl_constit,
            [[v | Info2] | Constit2],
            [[v,Word3,EC3,0 | Info3] | Constit3] | Buf_content3],
           [1,n,NewScr1,Abar_move,Abar_type,Scr2]) :-
                chk_adv(Buf_content1,[[[v,Word3,EC3,0 | Info3] |
                              Constit3] | Buf_content3]),
                NewScr1 is Scr1 + 1.

chk_a_move([[[n,Word1,EC1,ET1,1,1,Ten,S | Info1] | Constit1],
            Infl_constit | Buf_content1],
           [A_move,A_type,Scr1,Abar_move,Abar_type,Scr2],
           [[[n,Word1,EC1,ET1,1,1,Ten,NewScr1 | Info1] | Constit1],
            Infl_constit,
            [[v,Word3,EC3,0 | Info3] | Constit3] | Buf_content3],
```

```
           [1,n,NewScr1,Abar_move,Abar_type,Scr2]) :-
                chk_adv(Buf_content1,[[[v,Word3,EC3,0 | Info3] |
                              Constit3] | Buf_content3]),
                NewScr1 is Scr1 + 1.

chk_a_move([[[n,Word1,EC1,ET1,1,1,Ten,S | Info1] | Constit1] | Buf_content],
           [A_move,A_type,Scr1,Abar_move,Abar_type,Scr2],
           [[[n,Word1,EC1,ET1,1,1,Ten,NewScr1 | Info1] | Constit1],
            [[v,Word2,EC2,0,RC2,RT2,+ | Info2] | Constit2] | Buf_content2],
           [1,n,NewScr1,Abar_move,Abar_type,Scr2]) :-
                chk_adv(Buf_content,[[[v,Word2,EC2,0,RC2,RT2,+ | Info2] |
                                    Constit2] | Buf_content2]),
                NewScr1 is Scr1 + 1.

chk_a_move(Buffer,Infolist,Buffer,Infolist).


%-------------------------------------------------------------------------
% chk_abar_move(Buffer,Infolist,NewBuffer,NewInfolist)
%
% Flags non-argument movement and sets the subscript values to be used
% in creating a movement trace.
%

chk_abar_move([[[c,Word,EC,ET,RC,RT,Ten,S | Info] | Constit] | Buf_content],
              [A_move,A_type,Scr1,Abar_move,Abar_type,Scr2],
              [[[c,Word,EC,ET,RC,RT,Ten,NewScr2 | Info] | Constit] |
               Buf_content],
              [A_move,A_type,Scr1,1,n,NewScr2]) :-
                is_whword(Word),
                NewScr2 is Scr2 + 1.

chk_abar_move([[[Cat1,Word,EC,ET,RC,RT,Ten,S | Info1] | Constit1],
               [[Cat2 | Info2] | Constit2] | Buf_content],
              [A_move,A_type,Scr1,Abar_move,Abar_type,Scr2],
              [[[Cat1,Word,EC,ET,RC,RT,Ten,NewScr2 | Info1] | Constit1],
               [[Cat2 | Info2] | Constit2] | Buf_content],
              [A_move,A_type,Scr1,1,n,NewScr2]) :-
                (Cat1 = n;
                 Cat1 = d),
                (Cat2 = n;
                 Cat2 = d),
                NewScr2 is Scr2 + 1.

chk_abar_move(Buffer,Infolist,Buffer,Infolist).


%-------------------------------------------------------------------------
% ins_abar_move(Buffer,Infolist,NewBuffer,NewInfolist)
%
% Inserts an empty category corresponding to a moved non-argument in
% that case of the empty category appearing to the left of a
% subcategorizing element. Also unflags non-argument movement.
%

ins_abar_move([[[i,Word1,1 | Info1] | Constit1],
               [[v,Word2 | Info2] | Constit2] | Buf_content1],
              [[[Cat3,Word3,0,0 | Info3] | Constit3] | Stk_content],
              [A_move,A_type,Scr1,1,Abar_type,Scr2],
              [[[Abar_type,e,0,0,1,1,*,Scr2,[[[]]]]],
               [[i,Word1,1 | Info1] | Constit1],
               [[v,Word2 | Info2] | Constit2],
               [[v,Word4,EC4,1 | Info4] | Constit4] | Buf_content2],
               [A_move,A_type,Scr1,0,*,Scr2]) :-
                have/be_word(Word2),
                chk_adv(Buf_content1,[[[v,Word4,EC4,1 | Info4] |
                             Constit4] | Buf_content2]).

ins_abar_move([[[i,Word1,1 | Info1] | Constit1] | Buf_content1],
              [[[Cat3,Word3,EC3,ET3,0,0 | Info3] | Constit3] | Stk_content],
              [A_move,A_type,Scr1,1,Abar_type,Scr2],
              [[[Abar_type,e,0,0,1,1,*,Scr2,[[[]]]]],
               [[i,Word1,1 | Info1] | Constit1],
               [[v,Word4,EC4,1 | Info4] | Constit4] | Buf_content2],
               [A_move,A_type,Scr1,0,*,Scr2]) :-
                chk_adv(Buf_content1,[[[v,Word4,EC4,1 | Info4] |
                             Constit4] | Buf_content2]).

ins_abar_move(Buffer,Stack,Infolist,Buffer,Infolist).


%-------------------------------------------------------------------------
% chk_adv(Buffer,NewBuffer)
%
% Forms verb/adverb (or adverb/verb) combinations into a VP and inserts
% the VP into the front of the Buffer.
%

chk_adv([[[adv | Info1] | Constit1],
          [[v,Word2 | Info2] | Constit2] | Buf_content],
         NewBuf) :-
                chk_adv([[[v,max |Info2],[[adv |Info1] | Constit1],
                           [[v,Word2 | Info2] | Constit2]] | Buf_content],
                          NewBuf).

chk_adv([[[v,Word1 | Info1] | Constit1],
          [[adv | Info2] | Constit2] | Buf_content],
         NewBuf) :-
                chk_adv([[[v,max |Info1],[[v,Word1 |Info1] | Constit1],
                           [[adv | Info2] | Constit2]] | Buf_content],
                          NewBuf).
```

```
chk_adv(Buffer,Buffer).

% checks only for verb/adverb combinations.

verb_adv([[[v,Word1 | Info1] | Constit1],
           [[adv | Info2] | Constit2] | Buf_content],
          NewBuf) :-
                 verb_adv([[[v,max |Info1],[[v,Word1 |Info1] | Constit1],
                            [[adv | Info2] | Constit2]] | Buf_content],
                           NewBuf).

verb_adv(Buffer,Buffer).

%-------------------------------------------------------------------------
% selected word checks for membership in special word sets.
%

have/be_word(Word) :- member(Word,[have,has,had,having,be,is,am,are,was,
                                   were,been,being]).

be_word(Word) :- member(Word,[be,is,am,are,was,were,been,being]).

is_whword(Word) :- member(Word,[who,what,when,where]).
```

```
%---------------------------------------------------------------------------
% Section: Clause Check
%
% Paradigm: clause_check(Buffer,Stack,Infolist,NewBuffer,NewStack,NewInfolist)
%
% Description: Drives the formation of phrases. Triggered by encountering
%              a phrasal head or being specified to fulfill the
%              subcategorization requirements of a subcategorizer.
%

clause_check([[[p | Info] | Constit] | Buf_content],Stack,Infolist,
             NewBuf,NewStk,NewInfo) :-
               process_pp([[[p | Info] | Constit] | Buf_content],Stack,
                          Infolist,NewBuf,NewStk,NewInfo).

clause_check([[[c,e | Info] | Constit] | Buf_content],Stack,Infolist,
            NewBuf,NewStk,NewInfo) :-
               process_ip([[[c,e | Info] | Constit] | Buf_content], Stack,
                          Infolist,NewBuf,NewStk,NewInfo).

clause_check([[[c | Info] | Constit] | Buf_content],Stack,Infolist,
            NewBuf,NewStk,NewInfo) :-
               process_cp([[[c | Info] | Constit] | Buf_content], Stack,
                          Infolist,NewBuf,NewStk,NewInfo).

clause_check([[[v | Info] | Constit] | Buf_content],Stack,Infolist,
            NewBuf,NewStk,NewInfo) :-
               process_ip([[[v | Info] | Constit] | Buf_content], Stack,
                          Infolist,NewBuf,NewStk,NewInfo).

clause_check([[[d,Word | Info] | Constit] | Buf_content],Stack,Infolist,
            NewBuf,NewStk,NewInfo) :-
               Word \== max,
               build_np([[[d,Word | Info] | Constit] | Buf_content], Stack,
                        Infolist,Buf1,Stk1,Info1),
               process_ip(Buf1,Stk1,Info1,NewBuf,NewStk,NewInfo).

clause_check([[[d | Info] | Constit] | Buf_content],Stack,Infolist,
            NewBuf,NewStk,NewInfo) :-
               process_ip([[[d | Info] | Constit] | Buf_content], Stack,
                          Infolist,NewBuf,NewStk,NewInfo).

clause_check([[[n | Info] | Constit] | Buf_content],Stack,Infolist,
            NewBuf,NewStk,NewInfo) :-
               process_ip([[[n | Info] | Constit] | Buf_content], Stack,
                          Infolist,NewBuf,NewStk,NewInfo).


%---------------------------------------------------------------------------
% process_xp(Buffer,Stack,Infolist,NewBuffer,NewStack,NewInfolist)
%
```

```
% Builds a phrase of the specified type (i.e., xp) by processing
% (placing constituents on the stack) until the subcategorization of
% the phrase's subcategorizing element is satisfied.
%

process_pp([[[p,max | Info] | Constit] | Buf_content],Stack,Infolist,
           [[[p,max | Info] | Constit] | Buf_content],Stack,Infolist).

process_pp([[[Cat,Word,EC,ET,RC,RT,Ten,S,[[[]]]] | Constit] | Buf_content],
           Stack,Infolist,NewBuf,NewStk,NewInfo) :-
                word_check([[[Cat,Word,EC,ET,RC,RT,Ten,S,[[[]]]] |
                             Constit] | Buf_content], Stack, Infolist,
                             [First_Buf | Buf1], Stk1, Info1),
                process_pp(Buf1,[First_Buf | Stk1],Info1,
                             NewBuf,NewStk,NewInfo).

process_pp(Buffer,Stack,Infolist,NewBuf,NewStk,NewInfo) :-
                select_subcat(Buffer,Stack,Infolist,Buf0,NewInfo),
                build_stack(Buf0,Stack,NewBuf,NewStk).


%
%

process_cp([[[c,e,EC,ET,RC,RT,Ten,S | Info] | Constit] | Buf_content],
           Stack,Infolist,Buf_content,
           [[[c,e,EC,ET,RC,RT,Ten,S | Info] | Constit] | Stack],Infolist) :-
                S \== 0.

process_cp([[[Cat | Info] | Constit] | Buf_content],
           Stack,Infolist,NewBuf,NewStk,NewInfo) :-
                Cat \== c,
                Cat \== n,
                Cat \== d,
                process_cp1([[[c,e,0,0,0,0,*,0,[[[]]]]],
                              [[Cat | Info] | Constit] | Buf_content],
                              Stack,Infolist,NewBuf,NewStk,NewInfo).

process_cp(Buffer,Stack,Infolist,NewBuf,NewStk,NewInfo) :-
                process_cp1(Buffer,Stack,Infolist,NewBuf,NewStk,NewInfo).


process_cp1([[[Cat,Word,EC,ET,RC,RT,Ten,S,[[[]]]] | Constit] | Buf_content],
           Stack,Infolist,NewBuf,NewStk,NewInfo) :-
                word_check([[[Cat,Word,EC,ET,RC,RT,Ten,S,[[[]]]] |
                             Constit] | Buf_content], Stack, Infolist,
                             [First_Buf | Buf1], Stk1, Info1),
                process_cp1(Buf1,[First_Buf | Stk1],Info1,
                             NewBuf,NewStk,NewInfo).

process_cp1(Buffer,Stack,Infolist,NewBuf,NewStk,NewInfo) :-
```

```
            word_check(Buffer, Stack, Infolist, Buffer, Stack, Infolist),
            process_cp2(Buffer,Stack, Infolist, NewBuf, NewStk, NewInfo).

process_cp1(Buffer,Stack,Infolist,NewBuf,NewStk,NewInfo) :-
            word_check(Buffer,Stack,Infolist,
                      [First_Buf | Buf1], Stk1, Info1),
            process_cp1(Buf1,[First_Buf | Stk1],Info1,
                      NewBuf,NewStk,NewInfo).

process_cp2(Buffer,Stack,Infolist,NewBuf,NewStk,NewInfo) :-
            select_subcat(Buffer,Stack,Infolist,Buf0,NewInfo),
            build_stack(Buf0,Stack,NewBuf,NewStk).


%
%

process_ip([[[Cat,Word,EC,ET,RC,RT,Ten,S,[[[]]]] | Constit] | Buf_content],
        Stack,Infolist,NewBuf,NewStk,NewInfo) :-
            word_check([[[Cat,Word,EC,ET,RC,RT,Ten,S,[[[]]]] |
                      Constit] | Buf_content], Stack, Infolist,
                      [First_Buf | Buf1], Stk1, Info1),
            process_ip(Buf1,[First_Buf | Stk1],Info1,
                      NewBuf,NewStk,NewInfo).


process_ip(Buffer,Stack,Infolist,NewBuf,NewStk,NewInfo) :-
            word_check(Buffer, Stack, Infolist, Buffer, Stack, Infolist),
            process_ip2(Buffer,Stack, Infolist, NewBuf, NewStk, NewInfo).

process_ip(Buffer,Stack,Infolist,NewBuf,NewStk,NewInfo) :-
            word_check(Buffer,Stack,Infolist,
                      [First_Buf | Buf1], Stk1, Info1),
            process_ip(Buf1,[First_Buf | Stk1],Info1,
                      NewBuf,NewStk,NewInfo).

process_ip2(Buffer,Stack,Infolist,NewBuf,NewStk,NewInfo) :-
            select_subcat(Buffer,Stack,Infolist,Buf0,NewInfo),
            build_stack(Buf0,Stack,NewBuf,NewStk).


%-----------------------------------------------------------------------
% build_np(Buffer,Stack,Infolist,NewBuffer,NewStack,NewInfolist)
%
% Builds an NP by first forming the determiner-noun, and then
% attaching any post-noun modifiers. The determiner-noun is built
% by simply attaching all constituents to a dominating NP until a
% noun is encountered.
%

build_np([[[n | Info] | Constit] | Buf_content],Stack,Infolist,
```

```
        NewBuf,NewStk,NewInfo) :-
              build_to_noun([[[n | Info] | Constsit] | Stack],Stk1),
              [NP1,[[d,Word2,EC2,ET2,RC2,RT2 | Info2] | Con2] | Stk2] = Stk1,
              build_rel_clause([NP1 | Buf_content],[[bot_of_stack]],
                          Infolist,[NP2 | Buf1],NewStk,NewInfo),
              NewBuf = [[[n,max,EC2,ET2,1,1 | Info2],
                     [[d,Word2,EC2,ET2,RC2,RT2 | Info2] | Con2],NP2] |
                     Buf1].

build_np([ Constit | Buf_content],Stack,Infolist,NewBuf,NewStk,NewInfo) :-
              build_np(Buf_content,[Constit | Stack],Infolist,
                     NewBuf,NewStk,NewInfo).


%-----------------------------------------------------------------------
% build_to_noun(Stack,NewStack)
%
% Attaches the contents of the stack to a dominating node.
%

build_to_noun([[[n,Word1 | Info1] | Constit1],
             [[d,Word2 | Info2] | Constit2],[bot_of_stack] | S],
            [[[n,max | Info1],[[n,Word1 | Info1] | Constit1]],
             [[d,Word2 | Info2] | Constit2],[bot_of_stack] | S]).

build_to_noun([[[n,Word | Info] | Constit1],Constit2 | Stack],NewStk):-
              Word \== max,
              build_to_noun([[[n,max | Info],Constit2,[[n,Word | Info] |
                     Constit1]] | Stack],NewStk).

build_to_noun([[Top_max | Top_constit],Constit2 | Stack],NewStk) :-
              build_to_noun([[Top_max,Constit2,[Top_max | Top_constit]] |
                     Stack],NewStk).

%-----------------------------------------------------------------------
% build_rel_clause(Buffer,Stack,Infolist,NewBuffer,NewStack,NewInfolist)
%
% Builds a post-noun modifier by forming phrases until the NP's Case
% and Theta requirements are saturated.
%

build_rel_clause([Constit1,[[Cat,Word,EC,ET,RC,RT,+ | Info] | Constit2] |
                Buf_content],
               Stack,Infolist,
               [Constit1,[[Cat,Word,EC,ET,RC,RT,+ | Info] | Constit2] |
                Buf_content],
               Stack,Infolist).

build_rel_clause([Constit1,[[Cat,Word,EC,1 | Info] | Constit2] | Buf_content],
               Stack,Infolist,
               [Constit1,[[Cat,Word,EC,1 | Info] | Constit2] | Buf_content],
                 Stack,Infolist).

build_rel_clause([Constit1,[[d | Info] | Constit2] | Buf_content],
                 Stack,Infolist,
                 [Constit1,[[d | Info] | Constit2] | Buf_content],
                 Stack,Infolist).

build_rel_clause([Constit1,[[n | Info] | Constit2] | Buf_content],
                 Stack,Infolist,
                 [Constit1,[[n | Info] | Constit2] | Buf_content],
                 Stack,Infolist).

build_rel_clause([Constit1,[end_of_sentence] | Buf_content],
                 Stack,Infolist,
                 [Constit1,[end_of_sentence] | Buf_content],
                 Stack,Infolist).

build_rel_clause([Constit1 | Buf_content],Stack,Infolist,
                 NewBuf,NewStk,NewInfo) :-
                clause_check(Buf_content,Stack,Infolist,
                             Buf1,[Clause | Stk1],Info1),
                [MaxInfo | Rest_con] = Constit1,
                build_rel_clause([[MaxInfo,Constit1,Clause] | Buf1],
                                 Stk1,Info1,NewBuf,NewStk,NewInfo).


%-----------------------------------------------------------------------
% select_subcat(Buffer,Stack,Infolist,NewBuffer,NewStack,NewInfolist)
%
% Attempts to satisfy the subcategorization specification of a
% subcategorizer by building the subcategorized phrase. If a
% specific subcategorization fails, then it is eliminated from the
% subcategorization features of the element, and the next tried.
% If a specified subcategorization successfully completes, then all
% other subcategorizations are eliminated. If all specified
% subcategorizations fail, then the parse fails.
%

select_subcat(Buffer,Stack,Infolist,NewBuf,NewInfo) :-
                tryone_subcat(Buffer,Stack,Infolist,NewBuf,NewInfo).

tryone_subcat([[[Cat,Word,EC,ET,RC,RT,Ten,S,[Type | Subcat]] |
               Constit] | Buf_content],
              Stack,
              Infolist,
              [[[Cat,Word,EC,ET,RC,RT,Ten,S,[Type]] |
               Constit] | Buf3],NewInfo) :-
                length(Type,2),
                word_check([[[Cat,Word,EC,ET,RC,RT,Ten,S,[Type]] |Constit] |
                            Buf_content],Stack,Infolist,
                            [Constit2 | Buf1],Stk1,Info1),
```

```
                [[Stype | Rest] | Sub] = Type,
                insert_move(Stype,Buf1,[Constit2 | Stk1],Info1,
                            Buf_content2, Info2),
                build_subcat(Type,Buf_content2,[[bot_of_stack]],Info2,
                             Buf3,Stk3,NewInfo), !,
                chk_subcat(Type,Buf3).

tryone_subcat([[[Cat,Word,EC,ET,RC,RT,Ten,S,[Type | Subcat]] |
               Constit] | Buf_content],
              Stack,
              Infolist,
              [[[Cat,Word,EC,ET,RC,RT,Ten,S,[Type]] |
               Constit] | Buf3],NewInfo) :-
                word_check([[[Cat,Word,EC,ET,RC,RT,Ten,S,[Type]] |Constit] |
                            Buf_content],Stack,Infolist,
                            [Constit2 | Buf1],Stk1,Info1),
                [[Stype | Rest] | Sub] = Type,
                insert_move(Stype,Buf1,[Constit2 | Stk1],Info1,
                            Buf_content2, Info2),
                build_subcat(Type,Buf_content2,[[bot_of_stack]],Info2,
                             Buf3,Stk3,NewInfo), !,
                chk_subcat(Type,Buf3).

select_subcat([[[Cat,Word,EC,ET,RC,RT,Ten,S,[Type | Subcat]] |
               Constit] | Buf_content],
              Stack,
              Infolist,
              NewBuf,NewInfo) :-
                select_subcat([[[Cat,Word,EC,ET,RC,RT,Ten,S,Subcat] |
               Constit] | Buf_content],Stack,Infolist,NewBuf,NewInfo).


%-----------------------------------------------------------------------
% chk_subcat(Subcat_spec,Buffer)
%
% Checks to see that the constructed element matches the
% subcategorization requirement.
%

chk_subcat([[Type1 | Sinfo1],[Type2 | Sinfo2]],
           [[[Type1 | Info1] | Constit1], [[Type2 | Info2] | Constit2] |
            Buf_content]).

chk_subcat([[intr | Sinfo]],Buffer).

chk_subcat([[Type | Sinfo]],[[[Type | Info] | Constit] | Buf_content]).


%-----------------------------------------------------------------------
% insert_move(Subcat_spec,Buffer,Stack,Infolist,NewBuffer,NewInfo)
%
```

```
% Inserts an empty category immediately after a subcategorizing
% element, if the element subcategorizes for the flagged mavement
% type and the proper Case and Theta assignments are present.
%

insert_move(Type,Buffer,Stack,[1,Type,Scr1 | Info_content],
            [[[Type,e,0,0,0,1,*,Scr1,[[[]]]]] | Buffer],
            [0,*,Scr1 | Info_content]).

insert_move(i,
            [[[i | Info1] | Constit1] ,
             [[v | Info2] | Constit2] | Buf_content],
            Stack,
            [1,n,Scr1 | Info_content],
            [[[n,e,0,0,0,1,*,Scr1,[[[]]]]],
             [[i | Info1] | Constit1],
             [[v | Info2] | Constit2],
             [[v,Word3,EC3,1 | Info3] | Constit3] | Buf_content1],
            [0,*,Scr1 | Info_content]) :-
                chk_adv(Buf_content,[[[v,Word3,EC3,1 | Info3] | Constit3] |
                        Buf_content1]).

insert_move(i,
            [[[v | Info1] | Constit1] | Buf_content],
            Stack,
            [1,n,Scr1 | Info_content],
            [[[n,e,0,0,0,1,*,Scr1,[[[]]]]],
             [[v | Info1] | Constit1],
             [[v,Word2,EC2,1 | Info2] | Constit2] | Buf_content1],
            [0,*,Scr1 | Info_content]) :-
                chk_adv(Buf_content,[[[v,Word2,EC2,1 | Info2] | Constit2] |
                        Buf_content1]).

insert_move(i,
            [[[i | Info1] | Constit1] | Buf_content],
            Stack,
            [1,n,Scr1 | Info_content],
            [[[n,e,0,0,0,1,*,Scr1,[[[]]]]],
             [[i | Info1] | Constit1],
             [[v,Word2,EC2,1 | Info2] | Constit2] | Buf_content1],
            [0,*,Scr1 | Info_content]) :-
                chk_adv(Buf_content,[[[v,Word2,EC2,1 | Info2] | Constit2] |
                        Buf_content1]).

insert_move(i,[Constit | Buf_content],
            Stack,
            [1,n,Scr1 | Info_content],
            [[[n,e,0,0,0,1,*,Scr1,[[[]]]]],
             Constit | Buf_content],
            [0,*,Scr1 | Info_content]):-
                chk_adv(Buffer,[[[v,Word,EC,1 | Info] | Constit] |
```

```
                        Buf_content]).

insert_move(Type,Buffer,Stack,[1,Move_type,Scr1 | Info_content],
            [[[n,e,0,0,0,0,*,Scr1,[[[]]]]] | Buffer],
            [1,Move_type,Scr1 | Info_content]).

insert_move(Type,[[[Cat1,Word1,EC1,ET1,0,0 | Info1] | Constit1] | Buf_content],
            [[[Cat2,Word2,EC2,ET2,RC2,RT2,Ten2,S2,[[[Type2,1,1,W2]]]] |
             Constit2] | Stk_content],
            [A_move,A_type,Scr1,1,Abar_type,Scr2],
            [[[n,e,0,0,0,0,*,Scr2,[[[]]]]],
             [[Cat1,Word1,EC1,ET1,0,0 | Info1] | Constit1] | Buf_content],
            [A_move,A_type,Scr1,0,*,Scr2]).

insert_move(Type,[[end_of_sentence] | Buf_content],
            [[[Cat2,Word2,EC2,ET2,RC2,RT2,Ten2,S2,[[[Type2,1,1,W2]]]] |
             Constit2] | Stk_content],
            [A_move,A_type,Scr1,1,Abar_type,Scr2],
            [[[n,e,0,0,0,0,*,Scr2,[[[]]]]],
             [end_of_sentence] | Buf_content],
            [A_move,A_type,Scr1,0,*,Scr2]).

insert_move(Type,Buffer,Stack,Infolist,Buffer,Infolist).


%-----------------------------------------------------------------------
% build_subcat(Subcat_spec,Buffer,Stack,Infolist,
%                          NewBuffer,NewStack,NewInfolist)
%
% Attempts to build the subcategorized element by invoking the
% appropriate process_xp predicate for PPs, CPs and IPs; build_np
% for NPs fronted by determiners; and no processing for NPs fronted
% by nouns and intrasitive specifications.
%

build_subcat([[Type1 | Sinfo1],[Type2 | Sinfo2]],Buffer,Stack,Infolist,
             [Front_Buf1 | Buf2],Stk2,Info2) :-
                build_subcat([[Type1 | Sinfo1]],Buffer,Stack,Infolist,
                             [Front_Buf1 | Buf1],Stk1,Info1),
                build_subcat([[Type2 | Sinfo2]],Buf1,[[bot_of_stack]],Info1,
                             Buf2,Stk2,Info2).

build_subcat([[intr | Sinfo]],Buffer,Stack,Infolist,Buffer,Stack,Infolist).

build_subcat([[p | Sinfo]],Buffer,Stack,Infolist,
             [Top_Stk | Buf1],Stk1,Info1) :-
                process_pp(Buffer,Stack,Infolist,Buf1,[Top_Stk | Stk1],Info1).

build_subcat([[i | Sinfo]],Buffer,Stack,Infolist,
             [Top_Stk | Buf1],Stk1,Info1) :-
                process_ip(Buffer,Stack,Infolist,Buf1,[Top_Stk | Stk1],Info1).
```

```
build_subcat([[c | Sinfo]],Buffer,Stack,Infolist,
            [Top_Stk | Buf1],Stk1,Info1) :-
               process_cp(Buffer,Stack,Infolist,Buf1,[Top_Stk | Stk1],Info1).

build_subcat([[n | Sinfo]],[[[n | Info] | Constit] | Buf_content],Stack,
            Infolist,[[[n | Info] | Constit] | Buf_content],Stack,Infolist).

build_subcat([[n | Sinfo]],[[[d | Info] | Constit] | Buf_content],Stack,
            Infolist,Buf1,Stk1,Info1) :-
               build_np([[[d | Info] | Constit] | Buf_content],Stack,
                        Infolist,Buf1,Stk1,Info1).
```

```
%--------------------------------------------------------------------------
% Section: Build Stack
%
% Paradigm: build_stack(Buffer,Stack,NewBuffer,NewStack)
%
% Description: Forms the required phrase-structure representations by
%              building a tree structure with the contents of the
%              stack and the first element in the Buffer. The Buffer
%              element represents the completed subcategorization
%              requirement of the top element of the stack.
%              Tree building is accomplished by comparing two
%              adjacent elements in the epresentation with the
%              percolation principles and forming dominant nodes.
%              The completed tree (possibly fragment) is placed into
%              the front of the Buffer.
%

build_stack([[[v,Word,EC,ET,RC,RT,Ten,S,[Subcat]] | Constit1],
            Constit2,Constit3 | Buf_content],Stack,Buf_content,NewStk) :-
               length(Subcat,2),
               build_stk([Constit3,Constit2,
                          [[v,Word,EC,ET,RC,RT,Ten,S,[Subcat]] | Constit1] |
                          Stack],NewStk).


build_stack([Constit1,[end_of_sentence] | Buf],Stack,
           [[end_of_sentence]],NewStk) :-
               build_stk([Constit1 | Stack], NewStk).

build_stack([Constit1,Constit2 | Buf_content],Stack,Buf_content,NewStk) :-
               build_stk([Constit2,Constit1 | Stack],NewStk).


build_stk([Constit,[bot_of_stack] | Stk],[Constit,[bot_of_stack] | Stk]).

build_stk([Constit1,Constit2,[[v,Word,EC,ET,RC,RT,Ten,S,[Subcat]] | Constit3] |
            Stack],Newstk) :-
               length(Subcat,2),
               build_stk([[[v,max,EC,ET,RC,RT,Ten,S,[Subcat]],
                           [[v,Word,EC,ET,RC,RT,Ten,S,[Subcat]] | Constit3],
                           Constit2,Constit1] | Stack],Newstk).

build_stk([Constit1,Constit2 | Stack],NewStk) :-
               pp(Constit1,Constit2,Max),
               build_stk([[Max,Constit2,Constit1] | Stack],NewStk).


%--------------------------------------------------------------------------
% Percolation Principle I
%
% Where X Theta-governs Y, the categorial features of Z will be those of X.
```

```
%

pp([[Cat1,Word1,EC1,ET1,RC1,1 | Info1] | Constit1],
   [[Cat2,Word2,EC2,ET2,RC2,RT2,Ten2,S2,[[[Subcat2,IC2,1,W2]]]] | Constit2],
   Max) :-
        chk_max([Cat2,max,EC2,ET2,RC2,RT2,Ten2,S2,[[[Subcat2,IC2,1,W2]]]],Max).


%-----------------------------------------------------------------------
% Percolation Principle II
%
% Where X assigns Case to Y, the categorial features of Z will be those of X.

pp([[Cat1,Word1,1 | Info1] | Constit1],
   [[Cat2,Word2,EC2,ET2,1 | Info2] | Constit2],
   Max) :-
        chk_max([Cat1,max,1 | Info1],Max).

pp([[Cat1,Word1,EC1,ET1,1 | Info1] | Constit1],
   [[Cat2,Word2,EC2,ET2,RC2,RT2,Ten2,S2,[[[Subcat2,1,IT2,W2]]]] | Constit2],
   Max) :-
        chk_max([Cat2,max,EC2,ET2,RC2,RT2,Ten2,S2,[[[Subcat2,1,IT2,W2]]]],Max).


%-----------------------------------------------------------------------
% Percolation Principle III
%
% Where X is a member of the adjunct set and Y a member of the
% subcategorization set of a phrase Z, the categorial features of Z
% will be those of Y.
% In fact implemented to specify that the features of S (IP) will
% dominate over those of any other phrase.
%

pp([[Cat,max | Info1] | Constit1],
   [[i,max | Info2] | Constit2],Max) :-
        chk_max([i,max | Info2],Max).

pp([[i,max | Info1] | Constit1],
   [[Cat,max | Info2] | Constit2],Max) :-
        chk_max([i,max | Info1],Max).


%-----------------------------------------------------------------------
% Percolation Principle IV
%
% Where X and Y are in a CGC, no Case or Theta relations holds between
% them, and both are part of the subcategorization set of Z, the
% following hierarchy determines which features will percolate:
%   a. C-features of X and Y will percolate to Z
%   b. G-features of X and Y will percolate to Z
```

```
%   c. Theta-features of X and Y will percolate to Z
%

pp([[Cat1,Word1 | Info1] |Constit1],[[Cat2,Word2 | Info2] | Constit2],
   Max) :-
        (Cat2 = c;
         Cat2 = p),
        chk_max([Cat2,max | Info2],Max).

pp([[Cat1,Word1 | Info1] |Constit1],[[Cat2,Word2 | Info2] | Constit2],
   Max) :-
        (Cat1 = c;
         Cat1 = p),
        chk_max([Cat1,max | Info1],Max).

pp([[Cat1,Word1 | Info1] |Constit1],[[Cat2,Word2 | Info2] | Constit2],
   Max) :-
        (Cat2 = i;
         Cat2 = d),
        chk_max([Cat2,max | Info2],Max).

pp([[Cat1,Word1 | Info1] |Constit1],[[Cat2,Word2 | Info2] | Constit2],
   Max) :-
        (Cat1 = i;
         Cat1 = d),
        chk_max([Cat1,max | Info1],Max).

pp([[Cat1,Word1 | Info1] |Constit1],[[Cat2,Word2 | Info2] | Constit2],
   Max) :-
        chk_max([Cat2,max | Info2],Max).


%-----------------------------------------------------------------------
% chk_max(Max_element,NewMax_element)
%
% Adds a Theta requirement to any maximal PP, CP or IP.
%

chk_max([p,max,EC,ET,RC,0,Ten,S | Info],
        [p,max,EC,ET,RC,1,Ten,0 | Info]).

chk_max([c,max,EC,ET,RC,0,Ten,S | Info],
        [c,max,EC,ET,RC,1,Ten,0 | Info]).

chk_max([i,max,EC,ET,RC,0,Ten,S | Info],
        [i,max,EC,ET,RC,1,Ten,0 | Info]).

chk_max(Constit,Constit).
```

```
%---------------------------------------------------------------------------
% The 'convert' predicate, takes an input sentence, and returns a list
% of the lexical and categorial features of the original words.
%

convert_input([],[]).
convert_input([Punc|RestW],RestF) :-
        member(Punc,['.','?','!']),
        convert_input(RestW,RestF).
convert_input([Word|RestW],[[[Cat,Word,EC,ET,RC,RT,Ten,S,Subcat]]|RestF]) :-
        lex(Cat,Word,EC,ET,RC,RT,Ten,S,Subcat),
        convert_input(RestW,RestF).


lex(n,bill,0,0,1,1,*,0,[[[]]]).
lex(n,mary,0,0,1,1,*,0,[[[]]]).
lex(n,man,0,0,1,1,*,0,[[[]]]).
lex(n,boy,0,0,1,1,*,0,[[[]]]).
lex(n,lips,0,0,1,1,*,0,[[[]]]).
lex(n,girl,0,0,1,1,*,0,[[[]]]).
lex(n,chair,0,0,1,1,*,0,[[[]]]).
lex(n,book,0,0,1,1,*,0,[[[]]]).
lex(n,asparagus,0,0,1,1,*,0,[[[]]]).
lex(n,it,0,0,1,0,*,0,[[[]]]).
lex(n,john,0,0,1,1,*,0,[[[]]]).
lex(n,park,0,0,1,1,*,0,[[[]]]).
lex(n,trees,0,0,1,1,*,0,[[[]]]).
lex(n,car,0,0,1,1,*,0,[[[]]]).
lex(n,i,0,0,1,1,*,0,[[[]]]).
lex(n,you,0,0,1,1,*,0,[[[]]]).

lex(d,the,0,0,0,0,*,0,[[[]]]).
lex(d,a,0,0,0,0,*,0,[[[]]]).

lex(i,would,1,0,0,0,*,0,[[[]]]).
lex(i,did,1,0,0,0,*,0,[[[]]]).
lex(i,to,1,0,0,0,*,0,[[[]]]).
lex(i,dont,1,0,0,0,*,0,[[[]]]).

lex(adj,fond,0,0,0,0,*,0,[[[p,0,1,*]]]).
lex(adj,obvious,0,0,0,0,*,0,[[[c,0,1,*]]]).

lex(adj,fond,0,0,0,0,*,0,[[[]]]).
lex(adj,obvious,0,0,0,0,*,0,[[[]]]).
lex(adj,red,0,0,0,0,*,0,[[[]]]).
lex(adj,big,0,0,0,0,*,0,[[[]]]).
lex(adj,little,0,0,0,0,*,0,[[[]]]).

lex(adv,quietly,0,0,0,0,*,0,[[[]]]).

lex(c,that,0,0,0,0,*,0,[[[]]]).
```

```
lex(c,who,0,0,0,0,*,0,[[[]]]).
lex(c,what,0,0,0,0,*,0,[[[]]]).
lex(c,where,0,0,0,0,*,0,[[[]]]).

lex(p,on,0,0,0,0,*,0,[[[n,1,1,*]]]).
lex(p,in,0,0,0,0,*,0,[[[n,1,1,*]]]).
lex(p,of,0,0,0,0,*,0,[[[n,1,1,*]]]).
lex(p,by,0,0,0,0,*,0,[[[n,1,1,*]]]).
lex(p,with,0,0,0,0,*,0,[[[n,1,1,*]]]).
lex(p,to,0,0,0,0,*,0,[[[n,1,1,*]]]).

lex(v,kissed,0,1,0,0,+,0,[[[n,1,1,*],[p,0,1,*]],[[n,1,1,*]]]).
lex(v,sat,0,1,0,0,+,0,[[[p,0,1,*]]]).
lex(v,gave,0,1,0,0,+,0,[[[n,1,1,*],[p,0,1,*]],
                          [[n,1,1,*],[n,1,1,*]],[[n,1,1,*]]]).
lex(v,know,0,1,0,0,+,0,[[[c,0,1,*]],[[n,1,1,*]]]).
lex(v,loves,0,1,0,0,+,0,[[[n,1,1,*]]]).
lex(v,seems,0,0,0,0,+,0,[[[i,0,1,*]],[[c,0,1,*]]]).
lex(v,given,0,0,0,0,*,0,[[[n,1,1,*],[p,0,1,*]],[[n,1,1,*],[n,1,1,*]],
                           [[p,0,1,*]],[[n,1,1,*]]]).
lex(v,hoped,0,0,0,0,*,0,[[[c,1,1,*]]]).
lex(v,leave,0,1,0,0,+,0,[[[intr,0,0,*]]]).
lex(v,seen,0,0,0,0,*,0,[[[p,0,1,*]],[[n,1,1,*]]]).
lex(v,tried,0,1,0,0,+,0,[[[c,0,1,*]],[[p,0,1,*]],[[i,0,1,*]]]).
lex(v,asked,0,1,0,0,+,0,[[[n,1,1,*],[c,0,1,*]],[[c,0,1,*]]]).
lex(v,appear,0,0,0,0,+,0,[[[i,0,1,*]]]).
lex(v,likely,0,0,0,0,*,0,[[[c,0,1,*]]]).
lex(v,see,0,1,0,0,+,0,[[[n,1,1,*]]]).
lex(v,saw,0,1,0,0,+,0,[[[n,1,1,*]]]).
lex(v,say,0,1,0,0,+,0,[[[c,0,1,*]]]).
lex(v,left,0,1,0,0,+,0,[[[intr,0,0,*]]]).
lex(v,expect,0,1,0,0,*,0,[[[n,1,1,*]]]).
lex(v,slept,0,1,0,0,+,0,[[[intr,0,0,*]]]).
lex(v,like,0,1,0,0,+,0,[[[n,1,1,*]]]).
lex(v,likes,0,1,0,0,+,0,[[[n,1,1,*]]]).
lex(v,liked,0,1,0,0,+,0,[[[n,1,1,*]]]).
lex(v,had,0,1,0,0,+,0,[[[n,1,1,*]]]).
lex(v,likely,0,0,0,0,*,0,[[[c,0,1,*]]]).
lex(v,am,0,1,0,0,+,0,[[[p,0,1,*]],[[n,1,1,*]]]).
lex(v,was,0,0,0,0,+,0,[[[n,1,1,*]],[[intr,0,0,*]]]).
lex(v,is,0,0,0,0,+,0,[[[c,0,1,*]],[[n,1,1,*]],[[intr,0,0,*]]]).
lex(v,be,0,0,0,0,+,0,[[[]]]).
```

```
%_________________________________________________________________________
% The 'begin' predicate is used to initiate the reading of input from
% the keyboard, call the parser, and the print the parse tree.
%

begin :-
        read_in(Input),
        write(Input),nl,!,
        convert_input(Input,Output),
        append(Output,[[end_of_sentence]],Buffer),
        process_input(Buffer,Tree),nl,
%        ppxmax(0,Tree),nl,nl.
        ppxmax2(0,Tree),nl.

% The following is taken from C&M. It reads an input sentence, and puts
% it in list form.

read_in([W|Ws]) :- get0(C), readword(C,W,C1), restsent(W,C1,Ws).

restsent(W,_,[]) :- lastword(W),!.
restsent(W,C,[W1|Ws]) :- readword(C,W1,C1) , restsent(W1,C1,Ws).

readword(C,W,C1) :- single_char(C), !, name(W,[C]), get0(C1).
readword(C,W,C2) :-
    inword(C,NewC),!,
    get0(C1),
    restword(C1,Cs,C2),
    name(W,[NewC|Cs]).
readword(C,W,C2) :- get0(C1), readword(C1,W,C2).

restword(C,[NewC|Cs],C2) :-
    inword(C,NewC), !,
    get0(C1),
    restword(C1,Cs,C2).
restword(C,[],C).

single_char(44).
single_char(59).
single_char(58).
single_char(63).
single_char(33).
single_char(46).

inword(C,C) :- C>96, C<123.
inword(C,L) :- C>64, C<91, L is C+32.
inword(C,C) :- C>47, C<58.
inword(95,95).
inword(39,39).
inword(45,45).

lastword('.').
```

```
lastword('!').
lastword('?').
```

```
%-----------------------------------------------------------------------
% The following routines are used to pretty-print the parse tree that
% is generated by the parser. 'ppxmax' prints the tree with all
% available information. 'ppxmax2' prints it with some of the features
% removed, for easier readability.
%

ppxmax(Indent,[]) :- !.
ppxmax(Indent,[[bot_of_stack]]) :- !.

ppxmax(Indent,[[[Cat,max | Rest] | Constit] |Remainder]) :-
        spaces(Indent),
        write([Cat,max | Rest]),nl,
        NewIndent is Indent + 1,
        ppxmax(NewIndent,Constit),
        ppxmax(Indent,Remainder).

ppxmax(Indent,[LexItem|Rest]) :-
        spaces(Indent),
        write(LexItem),nl,
        ppxmax(Indent,Rest).

ppxmax2(Indent,[]) :- !.
ppxmax2(Indent,[[bot_of_stack]]) :- !.

ppxmax2(Indent,[[[i,max,EC,ET,RC,RT,Ten,S | Rest] | Constit] |Remainder]) :-
        S \== 0,
        spaces(Indent),
        write('infl'),write(' ('),write(S),write(')'),nl,
        NewIndent is Indent + 1,
        ppxmax2(NewIndent,Constit),
        ppxmax2(Indent,Remainder).

ppxmax2(Indent,[[[Cat,max,EC,ET,RC,RT,Ten,S | Rest] | Constit] |Remainder]) :-
        S \== 0,
        spaces(Indent),
        write(Cat),write(' ('),write(S),write(')'),nl,
        NewIndent is Indent + 1,
        ppxmax2(NewIndent,Constit),
        ppxmax2(Indent,Remainder).

ppxmax2(Indent,[[[i,max | Rest] | Constit] |Remainder]) :-
        spaces(Indent),
        write('infl'),nl,
        NewIndent is Indent + 1,
        ppxmax2(NewIndent,Constit),
        ppxmax2(Indent,Remainder).

ppxmax2(Indent,[[[Cat,max | Rest] | Constit] |Remainder]) :-
        spaces(Indent),
        write(Cat),nl,
```

```
        NewIndent is Indent + 1,
        ppxmax2(NewIndent,Constit),
        ppxmax2(Indent,Remainder).

ppxmax2(Indent,[[[i,Word,EC,ET,RC,RT,Ten,S | Rest] | Constit] |Remainder]) :-
        S \== 0,
        spaces(Indent),
        write('infl'),write(': '),write(Word),write(' ('),
        write(S),write(')'),nl,
        ppxmax2(Indent,Remainder).

ppxmax2(Indent,[[[Cat,Word,EC,ET,RC,RT,Ten,S | Rest] | Constit] |Remainder]) :-
        S \== 0,
        spaces(Indent),
        write(Cat),write(': '),write(Word),write(' ('),
        write(S),write(')'),nl,
        ppxmax2(Indent,Remainder).

ppxmax2(Indent,[[[i,Word | Rest] | Constit] | Remainder]) :-
        spaces(Indent),
        write('infl'),write(': '),write(Word),nl,
        ppxmax2(Indent,Remainder).

ppxmax2(Indent,[[[Cat,Word | Rest] | Constit] | Remainder]) :-
        spaces(Indent),
        write(Cat),write(': '),write(Word),nl,
        ppxmax2(Indent,Remainder).


write_spaces(0) :- !.

write_spaces(N) :-
     write(' '),
     N1 is N - 1,
     write_spaces(N1).

spaces(Level) :-
     HowMany is 3*Level + 1,
     write_spaces(HowMany),
     write(Level),write(': ').

member(X,[X|_]).
member(X,[_|Y]) :- member(X,Y).
```

```
%-----------------------------------------------------------------------------
% utility predicates
%


writestring([]).

writestring([N|Ns]) :-
     name(Name,[N]),
     write(Name),
     writestring(Ns).

file(P,File) :-
     telling(Old),
     told,
     tell(File),
     exec(P),
     told,
     tell(Old).

exec(P) :- P.
exec(_).

writel([nl|Xs]) :- !,
     nl,
     writel(Xs).

writel([X|Xs]) :- !,
     write(X),
     writel(Xs).

writel([]).

difference(X-Y,L) :-
     append(L,Y,X).


more(File) :-
   name(File,FileString),
   append("more ",FileString,Command),
   system(Command).

vi(File) :-
   name(File,FileString),
   append("vi ",FileString,Command),
   system(Command),
   [-File].

ls :-
   system("ls").
```

```
append([],X,X).
append([A|X],Y,[A|Z]) :- append(X,Y,Z).
```