A UNIFICATION-BASED NATURAL LANGUAGE INTERFACE TO A DATABASE

VOLUME I

NEIL KILBY SIMPKINS

Submitted for the degree of Doctor of Philosophy

THE UNIVERSITY OF ASTON IN BIRMINGHAM

August 1988

The University of Aston in Birmingham

A UNIFICATION-BASED NATURAL LANGUAGE INTERFACE TO A DATABASE

Neil Kilby Simpkins

Submitted for the degree of Doctor of Philosophy

1988

Summary

An implementation of a Lexical Functional Grammar (LFG) natural language front-end to a database is presented, and its capabilities demonstrated by reference to a set of queries used in the Chat-80 system. The potential of LFG for such applications is explored.

Other grammars previously used for this purpose are briefly reviewed and contrasted with LFG. The basic LFG formalism is fully described, both as to its syntax and semantics, and the deficiencies of the latter for database access application shown. Other current LFG implementations are reviewed and contrasted with the LFG implementation developed here specifically for database access.

The implementation described here allows a natural language interface to a specific Prolog database to be produced from a set of grammar rule and lexical specifications in an LFG-like notation. In addition to this, the interface system uses a simple database description to compile metadata about the database for later use in planning the execution of queries.

Extensions to LFG's semantic component are shown to be necessary to produce a satisfactory functional analysis and semantic output for querying a database. A diverse set of natural language constructs are analysed using LFG and the derivation of Prolog queries from the F-structure output of LFG is illustrated. The functional description produced from LFG is proposed as sufficient for resolving many problems of quantification and attachment.

Key Words : Lexical Functional Grammar, unification, database access, natural language processing, artificial intelligence

in memory of

William Saunders Wheeler Wallbank

Acknowledgements

I would like to express my thanks to the following people :

To my parents, for everything,

To my supervisor, Peter, for his patience, advice, humour and service beyond the call of duty,

To Irene for her cheerful assistance in compiling and proof reading this thesis,

To Dr. Brian Gay for his support, advice and guidance,

To Jackie, Chas, Kevin, Rob, Richard and Tony for their help and friendship,

To Gino for maintaining and providing help with the machines I have used,

and finally I acknowledge the financial support provided by the SERC of Great Britain.

Thanks, Neil.

Contents

Volume II

Appendix

List of Tables

List of Figures

Chapter 1
Introduction

1.1 Overview

Lexical Functional Grammar (LFG) was developed by Kaplan and Bresnan [Kaplan & Bresnan, 1982] as a formal theory of the mental representation of grammatical constructs and constraints on constructs. The formalism has been applied to several languages including English, Dutch, Russian and Icelandic. The objective of this thesis is to develop the LFG formalism for the description of natural language database queries and to produce a system that allows a database access system to be generated from LFG descriptions of a language. LFG will thus serve as a meta-language for the actual application (a natural language interface). The suitability of LFG for this purpose is at present unknown. LFG is unusual in being a linguistic formalism with a notation formal enough to be implemented, at least to some extent, as a computational grammar. LFG is also a member of the family of unification grammars.

LFG attempts to uncover an underlying structure directly without any extra mechanisms and thus provides a clarity not found in other grammars. LFG is primarily a tool for syntactic analysis so that it will be necessary to develop a semantic interpretation of the output that LFG produces. A linguistically based formalism may also offer more natural descriptions and a greater descriptive power than computational models.

This work seeks to implement LFG as an efficient computational tool for the description of natural language whilst retaining the formalism's simplicity and clarity. To demonstrate this, it has been decided to develop a domain independent (transportable) system which could be used to produce a natural language front end for simple Prolog databases. The most successful recent database access systems have used computational grammars. For this reason, a comparison is drawn with these systems and more specifically, the Chat-80 system [Pereira, 1982; Warren & Pereira, 1982].

As well as employing LFG for database querying, an LFG 'environment' is to be created which operates at the level of the LFG formalism. This also will include other tools to provide a more user friendly system such as simple spelling correction.

Basic to the problem of natural language processing, is the uncovering of the underlying or "deep" structure of a phrase. This may be near or greatly removed from the original or "surface" structure. A number of grammar formalisms have been developed as tools for describing Natural Language (NL). Several database systems have already been developed using these grammar formalisms.

1.2 Grammar Formalisms

Of particular interest in evaluating grammar formalisms [Shieber, 1985a] are the areas of "linguistic felicity" (the ease with which the desired constructs can be described), "linguistic expression" (the ability to describe the desired constructs at all), and "computational effectiveness" (whether the formalism is computationally practical and the demands made on computation by mechanisms within the formalism). In addition to these criteria, a grammar is required to be application independent and thus "transportable" between different application domains (databases) [Grishman, Hirschman & Friedman, 1983; Grosz, 1982; 1983; Hendrix & Lewis 1981]. Previous systems for database access have used grammars which can be divided roughly into four groups, described below.

1.2.1 Formal Grammars

Formal grammars are of four types (types 0, 1, 2 and 3) [Barr & Feigenbaum, 1981] and are well understood. A type 0 grammar is defined by three sets of non-terminals, terminals, production rules and also the grammar's distinguished symbol. The terminal and non-terminal sets have no common elements. Production rules take the form :

$$\text{LHS} \longrightarrow \text{RHS}$$

where the Left-Hand Side (LHS) and Right-Hand Side (RHS) are strings of elements from the terminal and non-terminal sets. No restrictions are placed upon the form of productions except that the LHS may not be an empty string. Type 1 (context sensitive)

grammars are defined similarly to type 0 with an additional restriction that the RHS contains at least as many elements as the LHS. Type 2 (context free) grammars further restrict productions so that the LHS may only consist of a single element. Type 3 (formal) grammars only allow rules of the form :

$$\text{LHS} \longrightarrow t \quad \text{RHS} \qquad \text{or} \qquad \text{LHS} \longrightarrow t$$

where the LHS and RHS are only a single element and *t* is a terminal.

A rule based grammar can thus be classified as type 0, 1, 2 or 3 according to its formal properties [Chomsky, 1959]. The restrictions, which have been outlined above, not only relate to the form a grammar can have but also to the notion of the power of a grammar (or the complexity of recognition using a grammar). A type 0 grammar is least restrictive but has the power of a Turing machine and thus will contribute little as a computational model of language. Each successive restriction on the form of a grammar (from type 0 to type 3) reduces the power of the grammar and increases the possible efficiency of recognition.

Efficient parsing algorithms exist for both regular and context free grammars but these are not suitable for natural language processing, as natural language is not context free and grammars must handle contextual information such as verb and subject number agreement. This has resulted in a concentration on type 2 grammars to ensure an initially acceptable level of complexity, with additional mechanisms added to the grammar to extend the grammar's power into at least some context sensitive areas (although there is still some disagreement as to wether NL recognition requires context sensitive power).

1.2.2 Semantic Grammars

Semantic grammars are application-biased grammars where the meta-language is close to the application. The PLANES [Waltz *et al*, 1976; Waltz & Goodman, 1977] and LIFER [Hendrix, Sacerdoti & Slocum, 1978] systems both employ grammars which fall into this group. LIFER is an environment for producing semantic grammars for specific applications and also includes a parser. The system has been used to create the INLAND (Informal Natural Language Access to Navy Data) front-end of the LADDER (Language Access to Distributed Data with Error Recovery) system which interfaces to a database

concerning shipping. INLAND uses 'semantic templates' which are matched during parsing against an input query. For example a template to match a WH-fronted interrogative might be :

'what' 'is' 'the' <Attribute> 'of' <Ship> .

where *<Attribute>* and *<Ship>* are variables ("semantic classes") intended to be filled by values or symbols from the input. This template might match, for example :

'what is the *tonnage* of *The Titanic*.'

These templates are very specific, covering only a very small number of phrases in comparison to a more general phrase structure rules such as :

Noun-Phrase , Verb-Phrase .

Semantic grammars suffer from one very important weakness. The grammar is not readily transportable between domains. It is a meta-description of the language of a particular application domain rather than a high-level NL description. Also semantic grammars are not a general description of NL and do not attempt to capture any universals of language. The addition of a new syntactic structure to the coverage of the grammar may require adding many new semantic templates to the grammar, so that all relevant semantic classes are used in a template matching the new syntactic structure.

1.2.3 Linguistic Grammars

Linguistic grammars include informal rules, such as Transformational Grammar (TG) [Jacobsen, 1978; 1986; Radford, 1981; King, 1983] where meta-rules are used to alter the shape of parse trees and the Augmented Transition Network (ATN) formalism [Woods, 1970; Bates, 1978] which changes the contents of global registers. Implicit in these formalisms is the recognition of multiple levels of syntactic representation. An initial (surface) representation is transformed into a deeper representation by the cyclic application of specialized rules.

The TGs have made a basic assumption that a sentence or phrase should be represented as a phrase structure tree. This type of structure is most easily generated by simple phrase structure rules. Phrase structure rules cannot however directly generate

certain types of structural relations that occur in natural language. That is, phrase structure rules cannot generate certain phrase structure trees. TG overcomes this problem by first generating a basic phrase structure and then using an additional set of rules to transform this structure into one that does represent the required structure. This however has implications for the complexity of recognition using TG. Unrestricted TG has been described by Berwick [1981] : " Thus in the worst case TGs generate languages whose recognition is widely recognized to be computationally intractable. "

If the basic assumption behind TG is withdrawn, then it may be found that transformational rules are no longer required to generate a phrase representation. The assumption that a phrase should be represented by a phrase structure has not been incorporated in the design of LFG. LFG uses a single level of phrase (syntactic) representation and no transformations are used to uncover a deeper phrase structure. Instead, the underlying deep structure is represented by a functional structure not a phrase structure. This functional structure is not created by the application of one or more transformational rules after producing an initial structure but by a general mechanism using a single analysis stage which is incorporated in the notational devices of the LFG formalism itself.

1.2.4 Computational Grammars

Computational grammars are precisely defined and designed to be computationally practical, such as logic grammars (CHAT-80). Several computational grammars have been tested in natural language database querying systems and have proved to be practical and efficient. Most notable amongst these have been the logic grammars : Definite Clause Grammar (DCG) [Pereira & Warren, 1980] together with its subsequent development, Extraposition Grammar (EG) [Pereira, 1981; 1982], used in the Chat-80 system, and Modular Logic Grammar (MLG) [McCord, 1985b].

Whilst they have proved effective, computational grammars may become very complex and difficult to understand in practical applications. Logic grammars are very close to the Prolog programming language itself. It is obviously desirable to have a higher-level formalism which is easily understood by linguists and which can be implemented efficiently on computer systems. The relative merits of these four types of formalism with respect to the stated evaluation criteria are given in Table 1.2.4.

Grammar Type	Evaluation Criteria: Linguistic felicity	Expressiveness	Computational effectiveness	Portability
Semantic	non-linguistic basis (very poor)	very poor	very good	very poor
Formal	very poor	unacceptably poor	very good	very good
Computational	poor	poor	good	good
Linguistic	very good	poor	good	good

Table 1.2.4 Evaluation of different grammar formalisms

Computational grammars have perhaps met with greatest success in providing the basis of transportable interfaces. For this reason, the work described here is repeatedly compared to the CHAT-80 system.

1.3 Unification Grammars

LFG is a member of the family of unification grammars which have emerged out of work on TG and ATNs. As a member of the unification family, LFG relies on a method of additive description called "unification" (not to be confused with Prolog unification). The unification process, which either fails or combines information, is a basic mechanism of all the unification grammars, Generalized Phrase Structure Grammar (GPSG) [Gazdar *et al*, 1985], Functional Grammar (FG) [Kay, 1979], Functional Unification Grammar (FUG) [Kay, 1984; 1985a], Simple Unification Grammar (SUG) [Kay, 1985b], Unification Categorial Grammar (UGC) [Zeevat, Klein & Calder, 1987], Categorial Unification Grammar (CUG) [Uszkoreit, 1986] and PATR-II [Shieber *et al*, 1983; Shieber, 1984].

Information is represented as a set of features and values which, taken as a whole, is frequently termed a "feature structure". Unification (U) is illustrated in Figure 1.3.a where the notation of the simplest unification grammar PATR-II has been adopted for clarity.

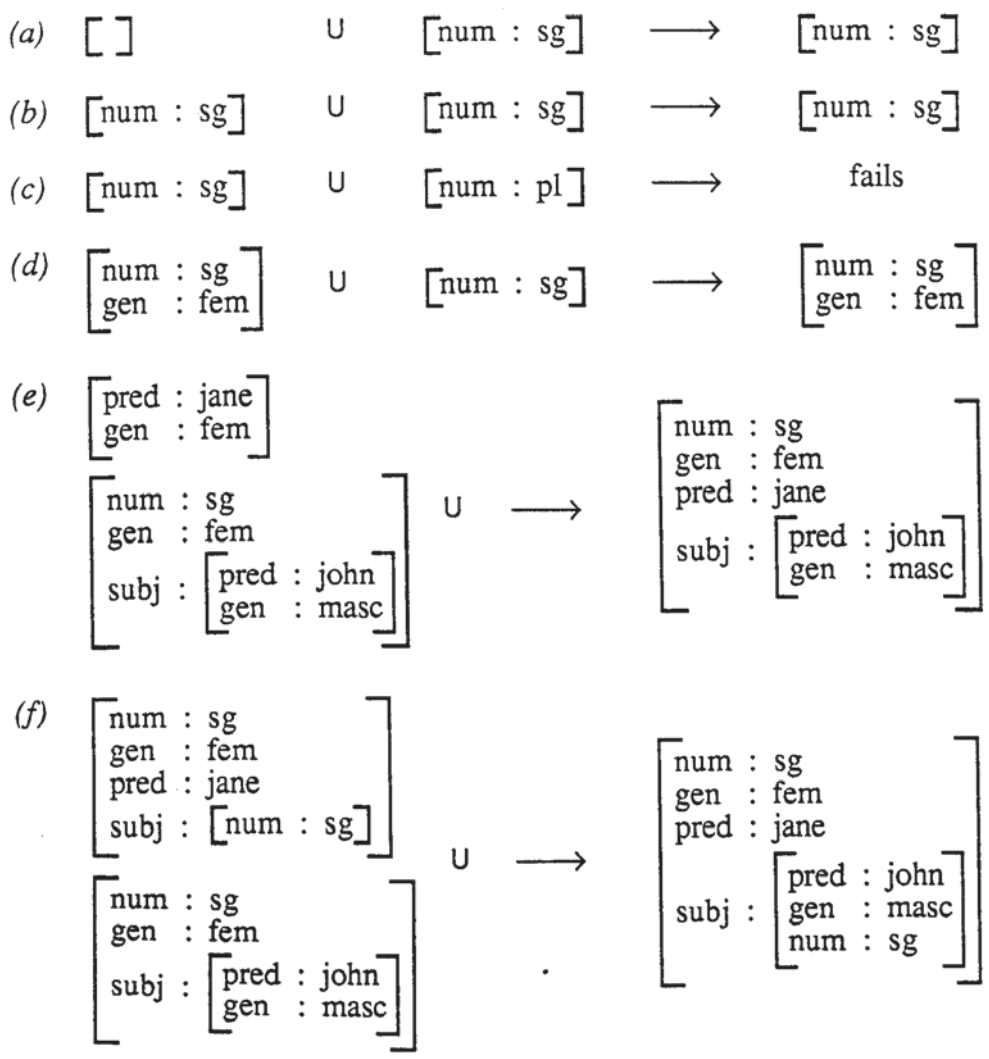


Figure 1.3.a Examples of Unification (U)

A feature and value are represented as a pair *'<feature> : <value>'*. Central to the idea of unification is the notion of "uniqueness" (one value per feature). The first case of unification *(a)* illustrates how unification adds information, a single atomic value 'num : sg' is added to an empty feature structure. Cases *(b)* and *(c)* illustrate the requirement of uniqueness. In case *(b)*, unification succeeds without adding information as both component feature structures have a single feature with the same unique atomic value. Unification of two feature structures with non-unique values *(c)* fails. Cases *(d)*, *(e)* and *(f)* also illustrate how information is added by unification. Cases *(e)* and *(f)* illustrate also the unification of recursive structures.

The unification of two feature structures D' and D'' may be formally defined [Shieber, 1985a, p14] as the most general feature structure D such that $D \sqsupseteq D'$ and $D \sqsupseteq D''$, which is notated $D = D' \cup D''$.

$$\begin{bmatrix} \text{obj} : \begin{bmatrix} \text{agree} : \begin{bmatrix} \text{number} : \text{pl} \\ \text{person} : \text{third} \end{bmatrix} \end{bmatrix} \\ \text{agree} : [1] \begin{bmatrix} \text{number} : \text{pl} \\ \text{person} : \text{third} \end{bmatrix} \\ \text{subj} : [1] \end{bmatrix}$$

Figure 1.3.b Feature Structure with Re-entrancy

The feature structure in Figure 1.3.b illustrates the representation of shared values (often called "re-entrancy" or "structure sharing") by co-indexing the subsidiary feature structures and showing the actual value of the subsidiary feature structure only at one place. It should be noted that there is a critical difference between two features with the same value and two features which share the same value.

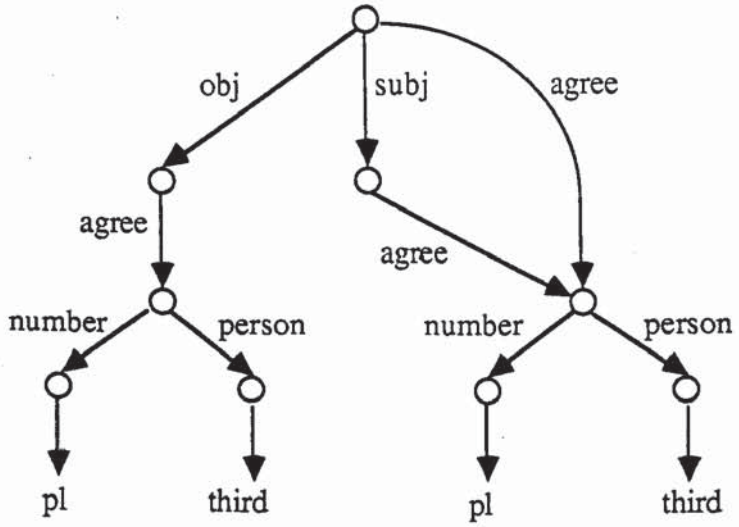


Figure 1.3.c DAG Representation of Feature Structure

Feature structures may be graphically represented as Directed Acyclic Graphs (DAGs) where attributes form arcs, values form nodes and simple symbolic values form leaves. The feature structure shown in Figure 1.3.b can for example, be graphically represented as the DAG shown in Figure 1.3.c. Values which are themselves feature structures, will themselves be DAGs.

Chapter 2

Lexical Functional Grammar

LFG incorporates two levels of description, Constituent structure "C-structure" and Functional structure "F-structure", the latter being the LFG equivalent of PATR-II's feature structure. C-structure defines the relationships between syntactic categories and terminal strings and is recognized by a standard Context Free Grammar (CFG). The F-structure is generated by equations attached to the CFG. An overview of LFG analysis is shown in Figure 2 where arrows mean roughly 'component of' [1].

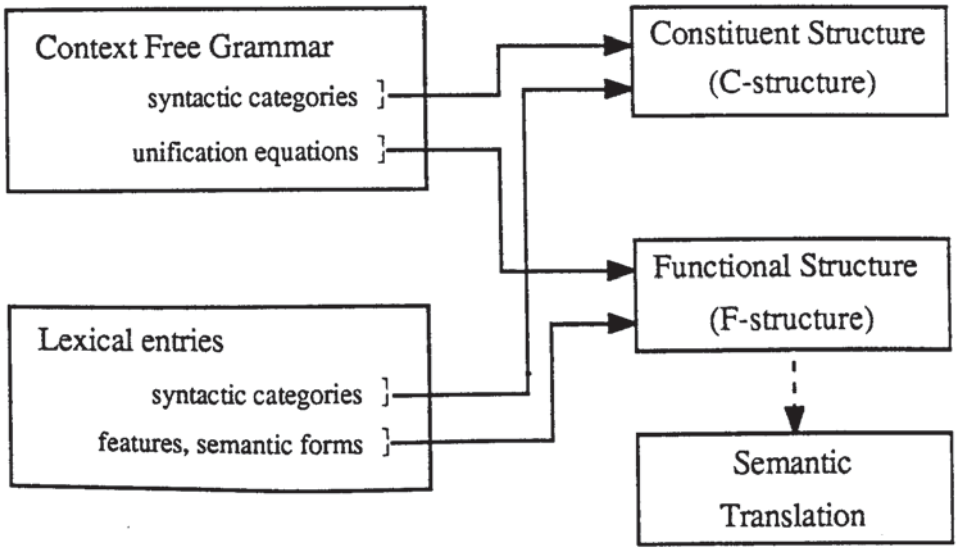


Figure 2 Overview of LFG phrase analysis

F-structures identify constituent (surface) functions and include feature specifications and semantic forms. The CFG very weakly constrains phrase structure but ungrammatical phrases are filtered out because their respective corresponding F-structure is not well-formed. The F-structure is a nested (recursive) structure which has a number of unordered 'attribute value' pairs. Attributes that have values which are themselves F-structures are termed "functions", of which there is presumed to be a small finite set. Attributes with simple atomic values are called "features". Although C-structure is recognized, F-structure is the sole output produced by LFG and intended for further

---

[1] A compact description of the LFG notation is provided by Appendix A in this volume.

semantic translation (Figure 2). The LFG formalism itself does not however encompass any processes past the production of F-structure.

2.1 Context Free Grammar and Equations

The CFG forms a simple set of rewrite rules with a single category (non-terminal) on the Left-Hand Side (LHS) and a number of categories (terminals and non-terminals) on the Right-Hand Side (RHS) :

$$s \longrightarrow np \quad vp \ .$$

As well as grammatical categories, 'literals' may appear in the RHS :

$$pp \longrightarrow (to) \quad np \ .$$

and may also have equations attached [1] :

$$vp' \longrightarrow \begin{pmatrix} to \\ (\uparrow to) = + \\ (\uparrow inf) =_c + \end{pmatrix} \quad \overset{vp}{\uparrow = \downarrow}$$

The CFG recognizes terminal categories during parsing, combining these into non-terminals and building the C-structure (often called a "derivation tree" in other formalisms). The equations attached to these categories in the grammar specify how the corresponding F-structure is to be constructed. The mechanisms of LFG are best illustrated by example. A simple LFG grammar is given in Figure 2.1.a.

s	⟶	np (↑ subj) = ↓	vp ↑ = ↓	
vp	⟶	v ↑ = ↓	np (↑ obj) = ↓	np (↑ obj2) = ↓
np	⟶	det ↑ = ↓	n ↑ = ↓	

Figure 2.1.a Simple LFG

---

[1] Parentheses are used in the normal CFG manner to denote optionality (Appendix A).

This grammar can be used, for example, to analyse the declarative phrase :

(a) 'the girl handed the baby a toy.'

The first rule recognizes a non-terminal *s* (LHS) which has two components, (also non-terminals) these being a *np* followed by a *vp*.

Arrows in the equations refer : '↓' to the subordinate F-structure produced by the nodes of C-structure below ; '↑' to the superior F-structure of the node above. The equation '(↑ subj) = ↓' can thus be read as 'the subj function of the F-structure above this node in the C-structure is to be unified with the F-structure produced from below this node in the C-structure'. This type of equation will be termed a "functional assignment" equation. Most categories are labelled with the "trivial" equation '↑ = ↓' which signifies unification of the F-structure below the node with that above. This equation is also that given to the functional 'head' category in a rule, corresponding to the head-feature convention found in GPSG. Figure 2.1.b lists seven types of simple LFG equation and gives an example of each of these. These equation types are used not only to specify values but also to force agreement between, for example, subject and verb number and person, and auxiliary and main verb tense and number.

- trivial (head)	↑ = ↓
- defining (or assignment)	(↑ subj) = ↓
- existential constraint	(↑ tense)
- negative existential constraint	¬ (↑ tense)
- value constraint	(↑ numb) = $_c$ plur
- negative value constraint	¬ (↑ numb) = $_c$ plur
- cyclic defining	(↑(↓ pcase)) = ↓

Figure 2.1.b Simple LFG Equation Types

Constraint equations are quite distinct from defining equations. Frequently it is necessary to impose a constraint relationship on an attribute's value, without specifying the attribute's actual value. Intuitively, this means imposing a constraint so that if an attribute does occur in an F-structure, it must or must not have a certain value. If however, the attribute does not occur, the attribute is allowed to remain unspecified. Thus a value/negative-value type constraint equation constrains the value a feature can/cannot take but does not imply an existential constraint on the attribute. The existential/negative-

existential constraints on the contrary, obviously cannot constrain the value an attribute takes : they merely ensure the feature's presence/absence. If defining equations were used to constrain values, there would be no way of distinguishing such 'constraints' from defined values in the F-structure produced. This notational difference thus segments F-structure into two classes (definitions and constraints).

The last equation in Figure 2.1.b merits special comment. An equation such as '$(\uparrow(\downarrow F_n)) = \downarrow$' (here termed a "cyclic defining" equation) attached to a node N in C-structure can be described as unifying a function F_v in the F-structure above N in C-structure with the F-structure from below N, where F_v is the value of the function of feature F_n in the F-structure below N. Such equations complicate DAG representation so that F-structure must be represented by a directed cyclic graph (Figure 2.1.c).

Cyclic graph Feature Structure (PATR-II)

Figure 2.1.c Cyclic Graph and Feature Structure After Application of '(↑(↓ pcase)) = ↓'

Any section of a DAG or feature structure can be referenced by a "path name". A path name simply lists (in order) the DAG arcs which must be taken, from the top level of the DAG to arrive at the referenced DAG section. The path naming the *second* value of person in the DAG in Figure 1.3.c is 'obj : agree : person'. LFG is unusually restrictive for a unification grammar in allowing path names in equations (both grammar and lexical) to be a maximum of two elements long. So both examples *(1)* and *(2)* below are allowable, but not example *(3)*.

(1)	(↑ subj) = ↓	(path of length 1 element)
(2)	(↑ subj num) = sg	(path of length 2 elements)
(3)	(↑ vcomp subj num) = sg	(illegal path of length 3 elements)

This restriction is termed the "functional locality" principle [Shieber, 1985a, p36].

2.2 Lexical Entries and Equations

There is an equality between lexical entries and grammar rules. Both have categorical and functional (feature and function specifications) components. Lexical entries in LFG, as the name implies, carry much more information than grammar rules. Each entry specifies the word being described, its category and a number of equations. As well as these, certain words, chiefly verbs and adjectives, have "semantic forms" specified. A minimal lexical entry for each word in the example phrase *(a)* is given in Figure 2.2.

the : det
(↑ spec) = the
(↑ num) = sing

girl : noun
(↑ num) = sg
(↑ pred) = 'girl'

handed : verb
(↑ tense) = past
(↑ pred) = 'hand⟨(↑ subj)(↑ obj2)(↑ obj)⟩'

baby : noun
(↑ num) = sg
(↑ pred) = 'baby'

a : det
(↑ num) = sg
(↑ spec) = a

toy : noun
(↑ num) = sg
(↑ pred) = 'toy'

Figure 2.2 Simple LFG Lexical Entries

As lexical entries will always be terminals of C-structure, they can only refer to the F-structure above ('↑'). Most of the equations in the entries above specify simple features and their atomic values, eg: '(↑ spec) = the' ; '(↑ num) = sg'. The verb entry also has a semantic form ('pred' attribute). This will play play the role of a semantic representation of a phrase described by the final F-structure. A semantic form has zero or more functional arguments which are enclosed in angled brackets. In the case of verbs, the number of arguments is equal to the verb's transitivity. The semantic form is said to "subcategorize" or "govern" the functions named as its arguments. There is assumed to be a small set of governable functions "designators" (subj, obj, vcomp). The semantic form

also plays an important role in determining the well-formedness of F-structures. This role is fully described in Section 2.5 but may be stated informally as 'the requirement that all those functions governed by the semantic form must be present in the F-structure' and that 'no other governable functions are present in the F-structure'.

As well as the normal semantic form (pred), an alternative semantic feature is specifiable for lexical constituents which are semantically empty. Amongst these are *there*, *it* and various idioms. These are given lexical entries containing the semantically empty 'form' feature instead of a semantic form 'pred' :

there	: N	(↑ form) = there
tabs	: N	(↑ form) = tabs
		(↑ form) = pl

A form feature makes no independent contribution to the meaning of a phrase. This allows even semantically empty F-structures to be properly governed.

2.3 Functional Control Equations

As well as the simple equations described earlier, LFG allows several more complex types of equation. Amongst these are "functional control" equations. These lexical equations identify two functions at different levels in the F-structure :

(↑ vcomp subj) = (↑ subj)

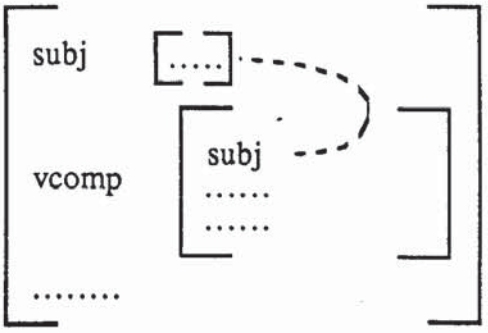
The equation above may be briefly described as stating that the subj function of the vcomp function has the same value as the subj function through structure sharing.

Functional control thus equates the values of two subsidiary F-structures. As semantic translation operates on an F-structure, there must be an indication in some way that each occurrence of the subsidiary F-structure is in fact derived from a single occurrence. Note that two subsidiary F-structures may be identical and not be occurrences of a single subsidiary F-structure (still distinct values). To overcome this problem, Kaplan and Bresnan [1982, p225] propose viewing lexical semantic forms as 'meta' semantic forms. A single "meta-form" represents any number of distinct actual semantic forms. An instance of a meta-form is then identified in an F-description by indexing. A

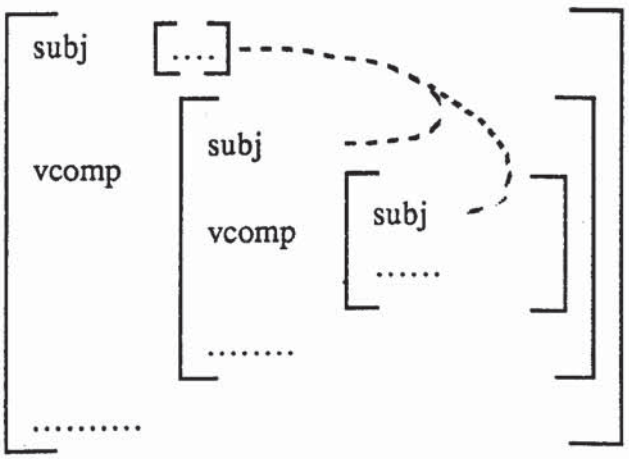
meta-form involved in functional control will then be instantiated to two actual semantic forms with the same index. Two semantic forms are thus assumed to be distinct unless they have the same predicate and argument structure and the same index. In principle, all semantic forms should be indexed but, as differing predicate or argument specifications implicitly identify different actual semantic forms, this is not necessary.

The various occurrences of a meta-form are said to imply that entire subsidiary F-structures will appear repeatedly in an F-structure. The implication of Kaplan and Bresnan is that not only the semantic forms but the entire informational content of the F-structure in which they are enclosed is duplicated. This will have an important consequence for the semantic interpretation of F-structure proposed here (Section 3.5).

The multiple occurrences of subsidiary F-structures are abbreviated by Kaplan and Bresnan in their diagrams by displaying the full value only at one place (the 'source') and linking this to other values by lines :



Functional control may be induced recursively so that a single function's value plays a role at several levels of an F-structure :



Functional control equations are used to account for the peculiarities of the so-called "equi" and "raising" verbs. These two types of constructs are closely related but are usually realized in TG by two (or more) separate rules.

The TG "equi-NP-deletion" rule is an obligatory rule which deletes a noun phrase from a complement clause of a sentence when the meaning of the noun phrase is the same as that of another noun phrase in the main clause of that same sentence. The phrase :

'Peter wants to see the film'

is derived by equi-NP-deletion from the main and complement clauses :

'Peter wants <X>' and 'Peter sees the film'

The noun phrase *Peter* (which is subject of the main clause) is thus deleted from the complement, where it is also the subject in surface structure.

The transformational raising rules (raising to object and raising to subject) apply to clauses with a complement, the subject of which appears to have been raised to function as subject or object of a higher clause. For example, in the phrase :

'Peter believes John to be honest.'

John functions as object of the main clause, even though it functions as subject of the complement :

'Peter believes <X>' 'John is honest'

Functional control can thus be seen as a simple notational device for describing the equi and raising phenomena.

2.4 Lexical Rules

LFG incorporates the notion of lexical redundancy rules. The functional control equations in lexical entries can be produced by one such rule :

Let L be a lexical semantic form and F_L its grammatical function assignment.
If xcomp $\in F_L$, add to the lexical entry of L :

$$(\uparrow \text{xcomp subj}) = (\uparrow \text{obj2}) \quad \text{if obj2} \in F_L$$
$$(\uparrow \text{xcomp subj}) = (\uparrow \text{obj}) \quad \text{if obj} \in F_L$$
$$(\uparrow \text{xcomp subj}) = (\uparrow \text{subj}) \quad \text{if subj} \in F_L$$

Another example of a lexical rule is the active/passive rule. There is a systematic relationship between lexical entries for the passive and those for the active :

kick ⟨(↑ subj) (↑ obj)⟩	(Brian kicked the student)
kick ⟨(↑ by obj) (↑ subj)⟩	(The student was kicked by Brian)

which can be expressed by the rule [1] :

$$\begin{array}{lcl} (\uparrow \text{subj}) & \longmapsto & (\uparrow \text{by obj}) \\ (\uparrow \text{obj}) & \longmapsto & (\uparrow \text{subj}) \end{array} \qquad (\uparrow \text{participle}) = \text{passive}$$

This rule defines the change from active to passive as changing the subj function to an obj function preceded by case *by*, the obj function to a subj function and adding a participle feature with value *participle*. These functional changes must, of course, be accompanied with corresponding morphological changes in the verb form.

A similar rule can be specified for dativizing :

hand ⟨(↑ subj) (↑ obj) (↑ to obj)⟩	(a girl handed the baby a toy)
hand ⟨(↑ subj) (↑ obj2) (↑ obj)⟩	(a girl handed a toy to the baby)

in the form of a rule :

$$\begin{array}{lcl} (\uparrow \text{obj}) & \longmapsto & (\uparrow \text{obj2}) \\ (\uparrow \text{to obj}) & \longmapsto & (\uparrow \text{obj}) \end{array}$$

The complete C-structure of *'the girl handed the baby a toy'* is given in Figure 2.4. Also shown are the lexical entries involved, their feature specifications and the equations on each node of the C-structure, which originate from the grammar.

---

[1] The symbol '$\longmapsto$' represents a simple transformation.

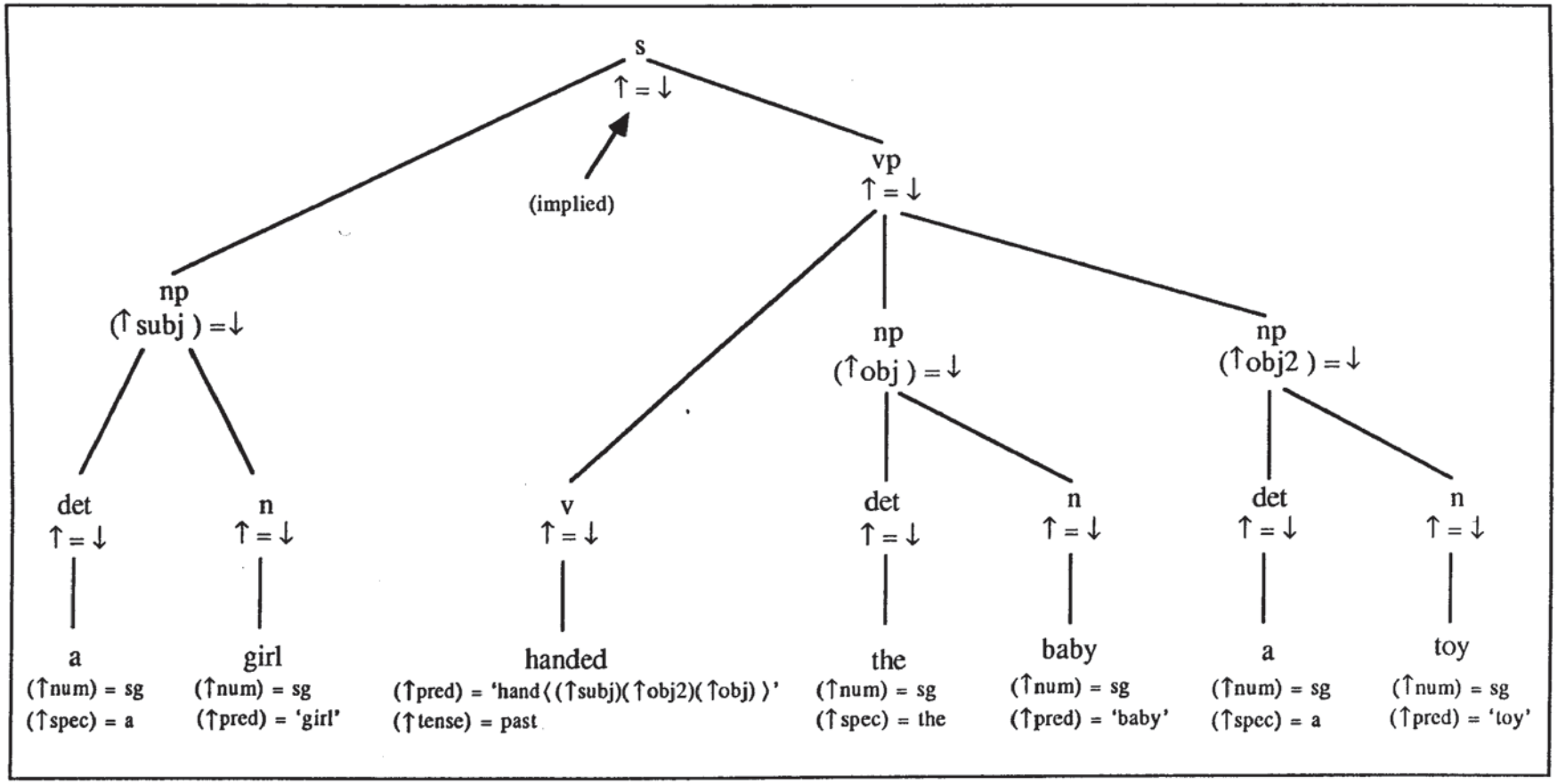


Figure 2.4 Example Annotated C-structure Tree

2.5 F-structure Production

A part of a C-structure with equations is shown in Figure 2.5.a, showing also how the equations construct a portion of the corresponding F-structure. The diagram shows how two lexical entries (F-structure sources) are incorporated into a C-structure. The first entry (Lex 1) is the single daughter of a node with category *np* and the second (Lex 2) the daughter of a node *vp*. The equations in a lexical entry can be viewed as equivalent to a partially specified F-structure. If *f1* is the F-structure originating from Lex 1, then this will be unified with the subj function's F-structure of the F-structure *f3* according to the equation below the *np*. If *f2* is the F-structure originating from Lex 2, then this will be unified with the vcomp function's value in *f3* in accordance with the equation below the *vp*. The F-structure *f3* is in turn unified with the F-structure of *f4*. An outline of the global F-structure representation produced by this analysis (*f4*) is also shown.

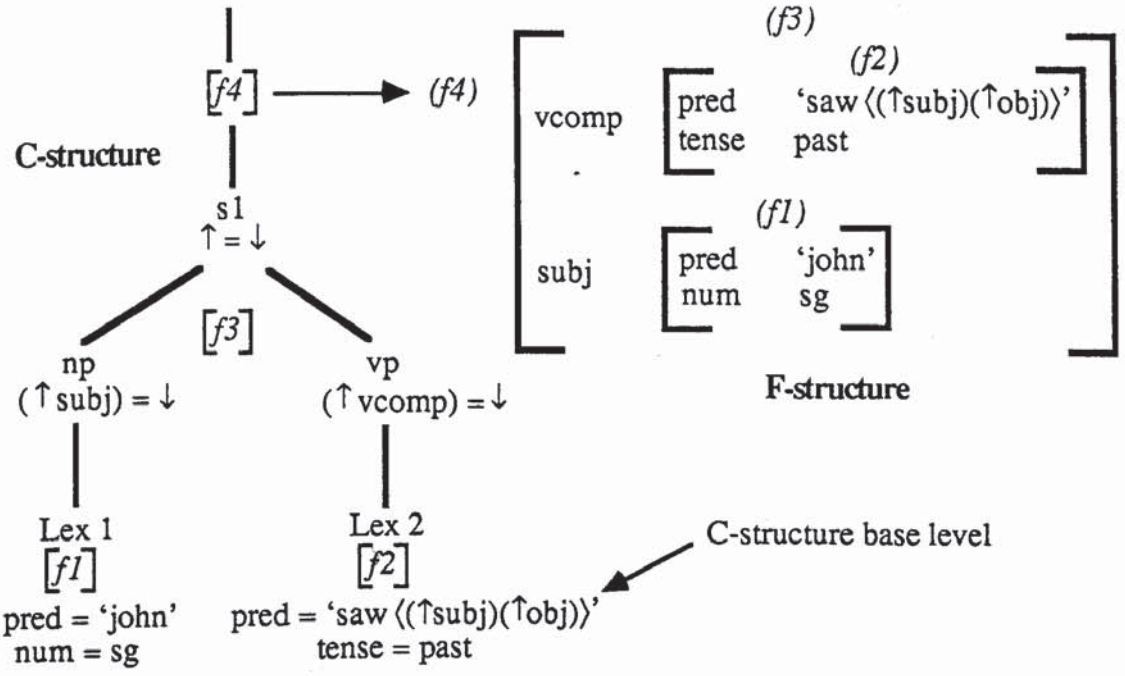


Figure 2.5.a Section of C-structure and Resultant F-structure

Kaplan and Bresnan [1982] describe in detail an algorithm for F-structure production. This algorithm first identifies unknown F-structures at nodes in C-structure with labels (*f1*, *f2*, ... *fn*) as in Figure 2.5.a. An F-structure can then be derived from a set of equations relating its F-structure components :

$$f4 = f3$$
$$f3\ (\text{subj}) = f1$$
$$f3\ (\text{vcomp}) = f2$$

The set of equations which defines an F-structure is termed an "F-description". An F-structure is then taken to be a mathematical function which represents the grammatical functions of a phrase. The arguments of this function are known, and therefore finding a solution requires deriving the function itself (here the final solution will be the F-structure, *f4*).

The equations above are derived from grammar equations which refer to F-structures by the use of the notational devices, '↓' and '↑'. These arrows thus serve to name a particular F-structure (*fn*) and are termed "metavariables". The actual value of a metavariable can only be determined at parse time. Metavariables are of two types :

Immediate domination metavariables : ↓ and ↑
Bounded domination metavariables : ⇓ and ⇑

Immediate domination metavariables, as their name implies, are variables which will be given an F-structure value from immediately above ('↑') or immediately below ('↓') the C-structure node to which they are attached. Bounded domination metavariables also are given F-structure values at parse time but as will be described later, the value may originate from a node more distant than the immediate parent or child C-structure node. If parsing is visualized bottom-up then, when a phrase is parsed, the F-structure values at the C-structure base (originating from lexical entries) become known and the corresponding immediate dominance metavariables can be instantiated :

$$f4 = f3$$
$$f3\ (\text{subj}) = \begin{bmatrix} \text{pred} & \text{'john'} \\ \text{num} & \text{sg} \end{bmatrix}$$
$$f3\ (\text{vcomp}) = \begin{bmatrix} \text{pred} & \text{'saw } \langle(\uparrow\text{subj})(\uparrow\text{obj})\rangle\text{'} \\ \text{tense} & \text{past} \end{bmatrix}$$

These values can then be propagated by substitution through the equations to eventually generate the final F-structure *f4*, assuming a solution exists :

$$f4 = \begin{bmatrix} \text{subj} & \begin{bmatrix} \text{pred} & \text{'john'} \\ \text{num} & \text{sg} \end{bmatrix} \\ \text{vcomp} & \begin{bmatrix} \text{pred} & \text{'saw } \langle(\uparrow\text{subj})(\uparrow\text{obj})\rangle\text{'} \\ \text{tense} & \text{past} \end{bmatrix} \end{bmatrix}$$

The algorithm for finding an F-structure solution described by Kaplan and Bresnan [1982] uses successive approximation. Importantly, they appear only to consider F-structure generation from a complete F-description, implying that C-structure is generated in a previous and completely separate stage. At each step, the algorithm takes the current set of assignments (symbols, semantic forms and partial F-structures) which satisfy the equations considered so far and revises these according to the next equation taken from the F-description. Equations may be considered in any order and a solution is only produced when all equations have been considered.

Evaluating equations in any order often means that equations refer to F-structures, functions or features which have yet to be fully specified. To overcome this problem, Kaplan and Bresnan [1982] introduce the notion of "place-holder" variables in the F-description's solution. These variables represent entities in the F-structure about which nothing is yet known and may, if the equation is not a constraint itself, also be considered to imply existential constraints. For example, if the equation :

$$(f_1 \text{ pred}) = \text{'see}\langle(\uparrow \text{subj})(\uparrow \text{obj})\rangle\text{'}$$

is included in an unspecified F-structure, then the resultant F-structure will include two place-holder variables in addition to the semantic form (a place-holder variable is represented by '___') :

$$\begin{bmatrix} \text{subj} & [\ \_\_\_\ ] \\ \text{pred} & \text{'saw } \langle(\uparrow\text{subj})(\uparrow\text{obj})\rangle\text{'} \\ \text{vcomp} & [\ \_\_\_\ ] \end{bmatrix}$$

To process an F-description, Kaplan and Bresnan [1982] describe three operators : *Include*, *Locate* and *Merge*. The Locate operator finds the current value for any designator (function or feature) in the assignments made so far. When the current values on both sides of an equation have been found by Locate, the Merge operator unifies the two

values. In more formal terms, if brackets represent the application of an operator to its arguments and D_1 and D_2 are the designators, an equation '$D_1 = D_2$' is processed by the evaluation of [Kaplan & Bresnan, 1982, p191] :

$$\text{Merge [Locate[} D_1 \text{], Locate[} D_2 \text{]]}$$

As well as these two operators, Kaplan and Bresnan [1982] define another operator *Include* which is used to implement the set inclusion operator. A membership statement '$D_1 \in D_2$' is processed by performing :

$$\text{Include [Locate[}D_1\text{], Locate[}D_2\text{]]}$$

The definitions of these operators are given in Appendix C. It should be noted that whilst Kaplan and Bresnan's definitions are quite precise, they do not state exactly how F-structures should be represented at the coding level or how these should be manipulated. That is to say, the 'algorithms' are more conceptual than actual.

The output from the LFG analysis (Figure 2.4) will be a single F-structure (Figure 2.5.b). This must be a unique well-formed solution of the grammar and lexical equations involved. The F-structure is devoid of any structural information relating to surface order.

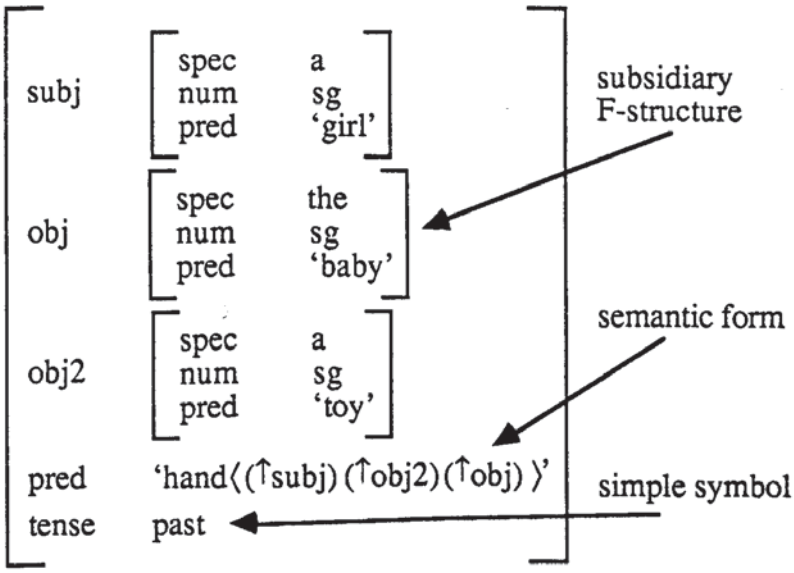


Figure 2.5.b Simple F-structure

The F-structure shown (Figure 2.5.b) contains three governable functions (subj, obj, obj2), a semantic form and a simple atomic value of 'tense'. As mentioned earlier,

the semantic form plays a central role in determining the well-formedness of an F-structure.

2.6 Constraints on Well-Formedness of Analyses

LFG requires that a number of constraints be satisfied in order for an analysis to be accepted. These are "conditions on grammaticality", "well-formedness conditions" and conditions on "completeness" and "coherence". LFG embodies most information in the functional component but as this is constructed from the associated C-structure tree, grammaticality is ensured not only by C-structure but also by F-structure well-formedness. All of these constraints apply directly to F-structures; additional conditions discussed later are imposed to constrain movement.

2.6.1 Condition on Grammaticality

Grammaticality conditions apply to input phrases. A string is grammatical only if it has a valid C-structure with an associated F-structure that is both "consistent" and "determinate". The F-structure assigned to the string is the value in the F-description's unique "minimal" solution in the place-holder variable of the C-structure's root node. A minimal solution is that F-structure which contains only those 'attribute value' pairs explicitly stated in the lexicon and grammar rules, which satisfies all constraints imposed on the F-structure. For any minimal solution F-structure there is an infinite number of other F-structures which are also valid solutions, except that they break the minimality constraint. These F-structures contain all of the information in the minimal solution and additional information which, whilst it does not contradict any constraints imposed on the F-structure, is redundant :

minimal solution :

$$\begin{bmatrix} \langle attribute1 \rangle & \langle value1 \rangle \\ \langle attribute2 \rangle & \langle value2 \rangle \end{bmatrix}$$

an additional solution :

$$\begin{bmatrix} \langle attribute1 \rangle & \langle value1 \rangle \\ \langle attribute2 \rangle & \langle value2 \rangle \\ \langle attribute3 \rangle & \langle value3 \rangle \end{bmatrix}$$

The requirement of a minimal solution limits the set of possible F-structure solutions from being an infinite set to a single member of that set (that with the least informational content).

The requirement for a unique solution means that all place holders (variables in the output structure) must be instantiated. This in turn means that all attributes in the F-structure produced must be assigned a single unique value.

The actual content of F-structures is left open by the LFG theory. Although constraints such as completeness and coherence, in conjunction with semantic forms, specify the expected functional structure (and there is in most cases general agreement about these structures) the more basic feature content of F-structures is undefined. Features can thus be 'created' and used as the grammar writer wishes. As long as these allow unification to succeed in the desired cases and also cause failure when appropriate then the features and values used are considered a 'valid' statement about the language being recognized, as far as the LFG theory is concerned.

There are several reasons which seem likely to explain why a feature system is not specified. Firstly LFG is a language independent theory and the level at which features are defined is certainly language dependent. Secondly there is no set agreement on what features actually exist ('are required' or 'should be used' to describe language) and about the names or values that these features can take. Lastly it is (at least) 'very difficult' to compile a complete list of justifiable features and their values and as language changes or the number of language cases is increased it seems likely that any initial list would have to be continually revised. It would require knowing about some 'correct' and 'complete' model of a language to produce a universally accepted feature system.

When some feature is used in LFG both the feature and the values it may take are an attempt to describe some characteristic of a language, however the feature is only a label it has no explicit meaning itself. A feature called 'number' could represent any characteristic of a language, it is only through convention that a meaning is given to this label.

The question thus arises : under what circumstances should a feature be introduced in a particular LFG grammar ?'. Here the following motivations have been used :

- to mark some linguistic phonema for semantic interpretation.
- to enforce some (usually generally recognized) constraint (such as subject/verb agreement).

In addition there is another motivation for using specific labels and including features and values which may not actually be used to affect the outcome of recognition (initially) : clarity. By calling for example number 'number' we conform to a linguistic convention and produce a much more readable language description than calling this say 'featureA'. Certainly it is desirable that the content of F-structures be justified by (at least) reference to the characteristics of language which the grammar writer is attempting to capture.

2.6.2 Functional Well-Formedness

Functional well-formedness conditions of the LFG theory also cause strings with otherwise valid C-structures to be marked as ungrammatical. The functional component thus acts as a filter on the output of the C-structure component.

The theory does not allow arbitrary predicates to be applied to the C-structure output. Rather, it is expected that a substantive linguistic theory will make available a universal set of grammatical functions and features and indicate how these may be assigned to particular lexical items and C-structure configurations. The most important of the well-formedness conditions, the "uniqueness" condition, ensures that the assignments for a particular sentence are globally consistent so that its F-structure exists.

Further functional conditions must also be satisfied by an F-structure. An F-structure must be both coherent and complete. An F-structure is locally complete if and only if it contains all the governable grammatical functions that its predicate governs. An F-structure is complete if and only if it and all of its subsidiary F-structures are locally complete. An F-structure is locally coherent if and only if all the governable grammatical functions that it contains are governed by a local predicate. An F-structure is coherent if and only if it and all its subsidiary F-structures are locally coherent. Following from this is the grammatical condition that a phrase is grammatical only if it is assigned a complete and coherent F-structure.

The example F-structure (Figure 2.5.b) is both complete and coherent. The top level semantic form of *hand* requires three functions to be present (subj, obj2, obj), which are all present. These functions are also well-formed, their semantic forms require no governable functions be present in their F-structures, which is also true.

2.7 The Kleene-Star (*) and Disjunction

The LFG notation departs slightly from a strict CFG by including the Kleene-star operator and disjunction. Disjunction in rules is represented in grammar rules by use of braces :

$$pp' \longrightarrow \left\{ \begin{array}{c} pp \\ (\uparrow (\downarrow \text{pcase})) = \downarrow \\ \\ pp \\ \downarrow \in (\uparrow \text{adjuncts}) \end{array} \right\}$$

The rule above signifies that a *pp'* may rewrite to a *pp* annotated with the equation '(↑ (↓ pcase)) = ↓' or a *pp* annotated with '↓ ∈ (↑ adjuncts)' [1]. Note that the *pp* categories need not be the same category.

The Kleene-star signifies that a grammatical category in an LFG rule may be repeated any number of times, including zero times. The Kleene-star is signified by an asterisk to the right of a category in a rule :

$$vp \longrightarrow v \quad np \quad pp^*$$

This rule thus states that a *vp* consists of a *v* followed by a *np* followed by zero or more *pp*. The rule thus represents an infinite number of actual CFG rule enumerations :

$$\begin{array}{llllll} vp & \longrightarrow & v & np & & \\ vp & \longrightarrow & v & np & pp & \\ vp & \longrightarrow & v & np & pp & pp \end{array} \qquad \text{(and so on).}$$

Kaplan and Bresnan indicate [1982, p277, n11] that the Kleene-star used in LFG is not intended to be interpreted as an abbreviation for an infinite repetition of some C-structure category. Infinite, or even large, repetitions of constituents do not occur in normal natural language and should thus not be recognized. It is suggested by Kaplan and Bresnan that in this respect the notation remains incomplete.

---

[1] The use of '∈' in LFG is explained in Section 2.8.

The Kleene-Star is useful for describing a number of linguistic constructs such as noun phrases containing adjectives, for example :

The *big tall handsome* man .
The *big tall* man .
The *big* man .
The man .

which can all be described by a single rule :

$$np \longrightarrow det \quad adj^* \quad n$$

if *the* is of category *det*, *man* of category *n* and all the words in italics of category *adj*. Repetitions can also be employed in the description of adjunct sequences (free complements) :

Peter painted the house {at the end of the road} {under the tree by the hill} .
Peter painted the house {at the end of the road} {under the tree} .
Peter painted the house {at the end of the road} .
Peter painted the house .

The braces in each of these sentences represent repetitions of a functional component of each phrase (adjunct). The Kleene-star thus allows a very compact description in a number of situations and may also be used with the disjunction notation :

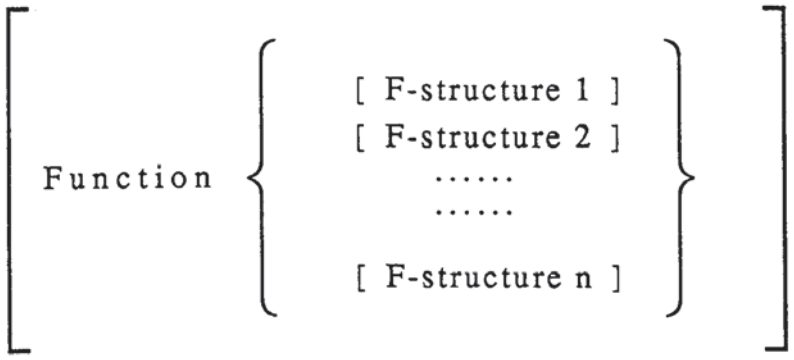
$$pp' \longrightarrow \left\{ \begin{array}{c} pp \\ (\uparrow (\downarrow \text{pcase})) = \downarrow \\ \\ pp \\ \downarrow \in (\uparrow \text{adjuncts}) \end{array} \right\}^*$$

The rule above signifies that *pp'* rewrites as any number of repetitions of the alternatives, *pp* annotated with '$(\uparrow (\downarrow \text{pcase})) = \downarrow$' or *pp* annotated with '$\downarrow \in (\uparrow \text{adjuncts})$'

Whilst C-structure repetition can easily be described in this manner, the problem of dealing with these repetitions in F-structural terms cannot be solved using the simple equations described so far. For this reason, Kaplan and Bresnan [1982] introduce the notion of sets and the set inclusion operator into LFG.

2.8 Sets

Sets are used in LFG when a C-structure element plays some functional role and occurs as a number of repetitions. A set value of a function is signified by enclosure in braces :



A function which has a set value is obviously an exception to the rule of uniqueness that requires a single unique value.

LFG has a set inclusion operator '$\in$'.which is used to assign an F-structure to membership of a set. This device is usually employed in conjunction with the Kleene-Star operator. The Kleene-Star operator can be used to recognize any number of repetitions of a C-structure component and the set inclusion operator can be used to assign the corresponding F-structures to membership of some function's set value. A rule suitable for recognition of the adjunct sequence examples given in Section 2.7 might be :

$$\begin{array}{lcccc} vp & \longrightarrow & v & np & pp \\ & & \uparrow = \downarrow & (\uparrow \text{subj}) = \downarrow & \downarrow \in (\uparrow \text{adjuncts}) \end{array}$$

This rule might be used to recognize the *vp* :

'painted the house {at the end of the road} {under the tree} .'

where *painted* has category *v*, *'the house'* category *np* and, *'at the end of the road'* and *'under the tree'* category *pp*. In outline, the F-structure produced will be :

$$\left[\begin{array}{ll} \text{pred} & \text{'painted}\langle(\uparrow \text{subj})(\uparrow \text{obj})\rangle\text{'} \\ \text{obj} & \left[\begin{array}{ll} \text{det} & \text{the} \\ \text{pred} & \text{'house'} \end{array}\right] \\ \text{adjuncts} & \left\{\begin{array}{l} [\text{ at the end of the road }] \\ [\text{ under the tree }] \end{array}\right\} \end{array}\right]$$

The internal description of the adjunct set members is not shown here. This is the case also in the description given by Kaplan and Bresnan [1982, p216]. Adjuncts, and all other functions which are sets, are not governable and thus not subject to the constraints of completeness and coherence. More will be said about the functional aspect of adjuncts when sample queries are analysed (Chapter 4).

LFG's treatement of many constructs which are analysed as set members using the Kleene-star, is a particular weakness of the formalism. In this respect the grammar can be viewed as incomplete. Especially in for example agreement checking between coordinations and restrictions on adjective sequences. Features can be passed upward from within set members to enforce some agreement but the ommision of restrictions on adjective sequences may require additional (semantic) mechanisms to prevent unreasonable adjective sequences being accepted.

2.9 Long-Distance Dependencies

Bounded domination metavariables, as mentioned in Section 2.4, may receive values originating from some distant section of C-structure. These metavariables can be used to move features through the C-structure tree. One well known case of movement is found in interrogatives, with which the mechanisms in LFG for dealing with these long-distances dependencies can be illustrated.

The underlying deep structure in the case of 'question phrases' (interrogatives) is more removed from the original surface structure than in declaratives by movement. To illustrate how LFG uncovers an underlying structure in such cases, a simple example can be used :

(b) 'What has Fred sold ?'

The underlying structure of an interrogative is the same as the surface structure of the corresponding declarative phrase. In this case, the corresponding declarative is *'Fred has sold <X>'* where *<X>* is whatever Fred has sold (the element under question). An outline of the functional composition of this declarative is shown in Figure 2.9.a.

Declarative form :

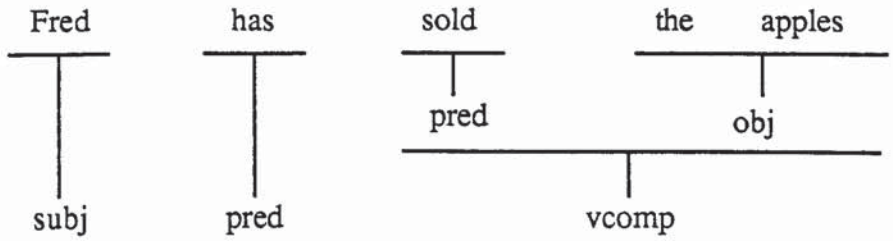


Figure 2.9.a Functional Description of a Simple Declarative

The semantic form to be filled in the complement is :

$$\text{pred} = \text{'sold}\langle(\uparrow \text{subj})\ (\uparrow \text{obj})\rangle\text{'}$$

That is, the verb *sold* in this context governs a subj function and an obj function. The main subj function of the verb *has* is passed to the vcomp function to become its subj function according to the functional control equation '(↑ vcomp subj) = (↑ subj)'. The semantic form of *sold* must be the same in the interrogative version but the obj function from the vcomp function has been moved to a clause initial position (Figure 2.9.b).

Interrogative form (showing subject / verb inversion) :

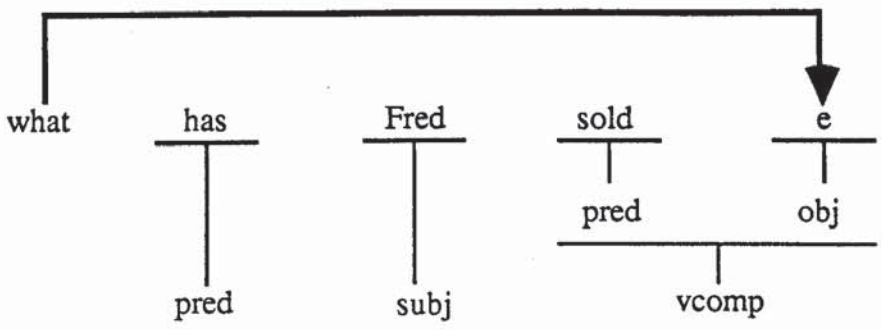


Figure 2.9.b Functional Description of a Simple Interrogative

The problem of uncovering the underlying structure requires undoing the effects of movement. In LFG, constituents can be moved over arbitrary distances in C-structure using bounding dominance metavariables (in most instances it is desirable to restrict this movement in ways described later). These are used in equations annotating the grammar just as immediate dominance variables are used. The notation and functioning of bounding metavariables can be illustrated by example. The rules listed in Figure 2.9.c can be used to parse the example interrogative *(b)*.

The grammar in Figure 2.9.c includes three bounded dominance metavariables, two are included in equations below the *np* category in rule *(1)* and one in the single equation below the *e* category in rule *(5)*. The first two of these are downward pointing metavariables termed "constituent controllers"; the third an upward pointing arrow is termed a "constituent controllee". These will take part in occurrences of "constituent control". Like immediate dominance variables, the actual values of bounded metavariables are instantiated at parse time.

(1)	s	$\longrightarrow$	np $(\uparrow \text{focus}) = \downarrow$ $(\uparrow q) = \Downarrow^{np}_{[+wh]}$ $\downarrow = \Downarrow^{s1}_{np}$		$\boxed{s1}$ $\uparrow = \downarrow$ $(\uparrow \text{aux})$
(2)	s1	$\longrightarrow$	v $\uparrow = \downarrow$	np $(\uparrow \text{subj}) = \downarrow$	vp1 $(\uparrow \text{vcomp}) = \downarrow$
(3)	np	$\longrightarrow$	n $\uparrow = \downarrow$		
(4)	vp1	$\longrightarrow$	v $\uparrow = \downarrow$	np $(\uparrow \text{obj}) = \downarrow$	
(5)	np	$\longrightarrow$	e $\uparrow = \Uparrow_{np}$		
(6)	np	$\longrightarrow$	det $\uparrow = \downarrow$		

Figure 2.9.c LFG Including Movement Mechanisms

The value given to a bounded metavariable is described by the value referenced in its equation. The controller in the first equation :

$$(\uparrow q) = \Downarrow^{np}_{[+wh]}$$

may be viewed as equating the F-structure of the function q with the bounded metavariable's F-structure. The controller also has a superscript *np* and a subscript *[+wh]*. A controller's superscript category matches that of its "domain root". The controller here must be passed into a *np* rooted domain. The root node of a controller is formally defined by Kaplan and Bresnan [1982, p244] :

> " Suppose $\Downarrow^{r}_{c}$ is a controller metavariable attached to a node N. Then a node R is the root node of a control domain for $\Downarrow^{r}_{c}$ if and only if :
>
> - R is a daughter of N's mother, and
> - R is labelled with category r. "

This means that a controller's domain root must be a node, which is a category the same as the controller's superscript, occurring in the same rule as the equation containing the controller (Figure 2.9.d).

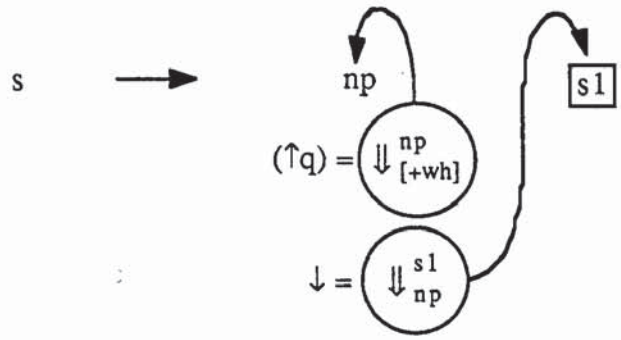


Figure 2.9.d Controller Domains in a LFG Rule

The necessary lexical entries for the example interrogative phrase *(b)* are shown in Figure 2.9.e.

In order to be well-formed, for each controller in the equations attached to a parse tree there must be a corresponding controllee in the controller's domain. In addition to this, the subscripts of the controller and controllee must match.

what : det
($\uparrow$ det) = what
($\uparrow$ wh) = +
$\uparrow = \Uparrow_{[+wh]}$

fred : noun
($\uparrow$ pred) = 'fred'
($\uparrow$ num) = sg

has : verb
($\uparrow$ aux) = have
($\uparrow$ pred) = 'have⟨($\uparrow$ vcomp)⟩ ($\uparrow$ subj)'
($\uparrow$ vcomp subj) = ($\uparrow$ subj)
($\uparrow$ subj num) = sg
($\uparrow$ tense) = past

sold : verb
($\uparrow$ pred) = 'sell⟨($\uparrow$ subj) ($\uparrow$ obj)⟩'
($\uparrow$ tense) = past

Figure 2.9.e Lexical Entries Including Mechanisms for Movement

The threading of controllers or controllees through C-structure is not allowed through "bounding nodes". Bounding nodes are denoted by a square box around a grammatical category in LFG rules. They define 'islands' of the C-structure that constituent control dependencies (movement) may not penetrate (Figure 2.9.f).

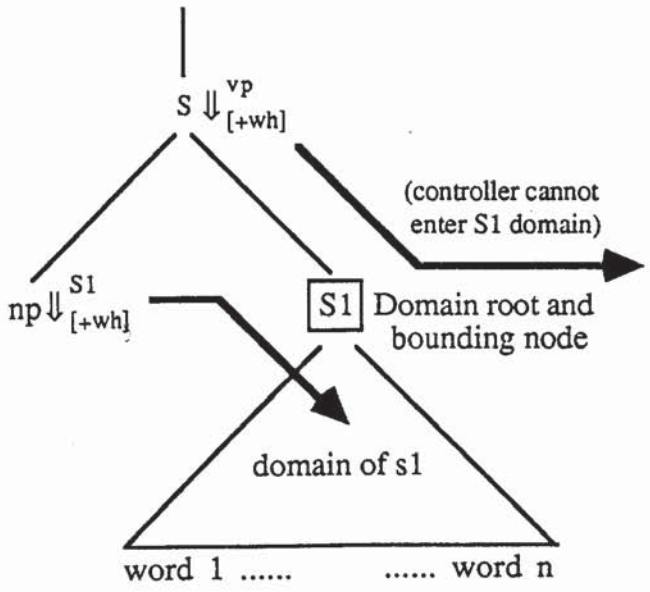


Figure 2.9.f Bounding Node in C-structure

This allows the generation of gaps to be controlled. The C-structure generated for the example movement phrase is outlined in Figure 2.9.g and the threading of controllers (or

controllees) through this structure is illustrated to show controllee/controller correspondences.

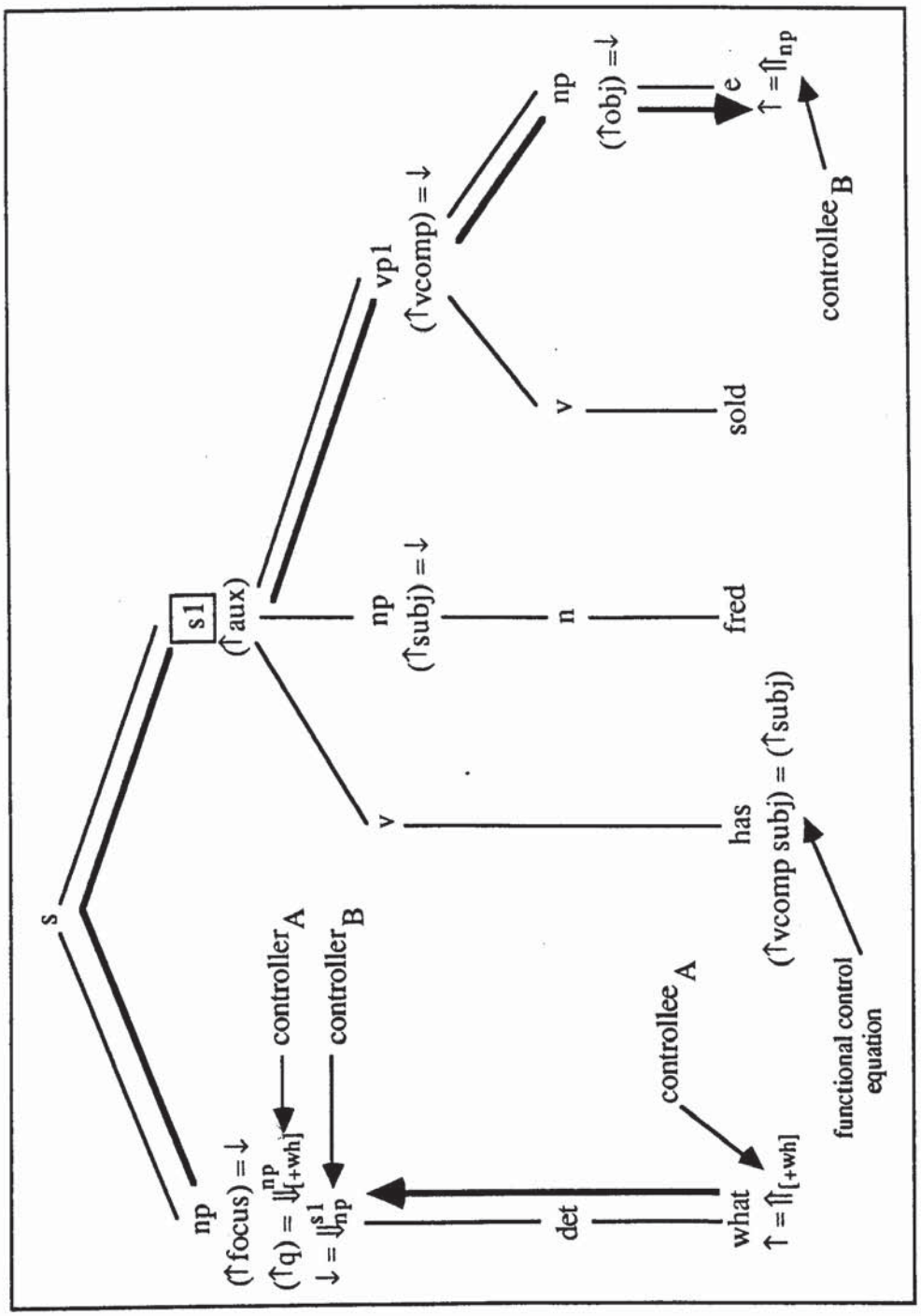


Figure 2.9.g C-structure Tree with Movement

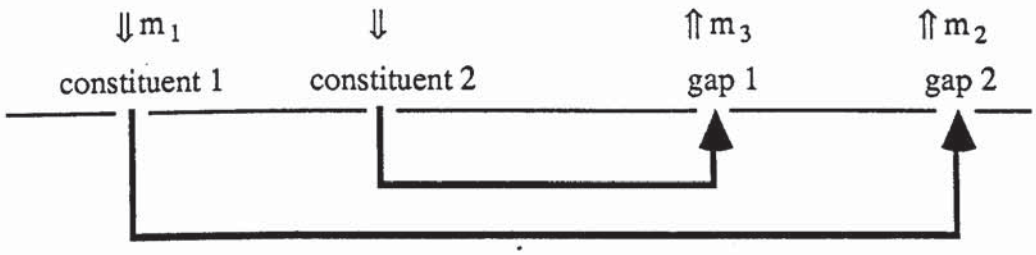
Movement is not only restricted by the presence of bounding nodes in C-structure but also by the notions of "crossing degree" and "nearly nested correspondence". An additional (language dependent) constraint is imposed in LFG ; the "crossing limit".

Crossing relates to the paths between pairs of matching controllers and controllees. Kaplan and Bresnan define crossed correspondence (Figure 2.9.h) :

> " The correspondence of two metavariables m_1 and m_2 is "crossed" by a controller or controllee m_3 if and only if all three variables have compatible categorial subscripts and m_3 but not its corresponding controllee or controller is ordered between m_1 and m_2. "

A metavariable correspondence has an associated number which is called its "crossing degree". This number is defined as the number of controllers and controllees by which a metavariable correspondence is crossed.

(a) zero crossing degree in metavariable correspondences (English) :



(b) crossing degree of one in metavariable correspondences :

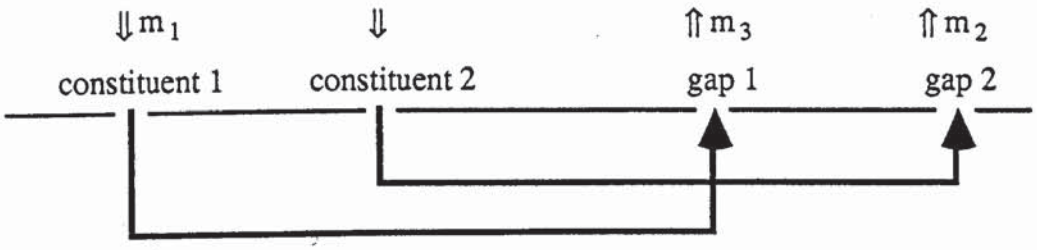


Figure 2.9.h Crossing Degree Examples

Matching of a controller with the 'closest' controllee in C-structure corresponds to a crossing degree of zero *(a)* : matching with the second closest controllee corresponds to a crossing degree of one *(b)*. A crossing limit is then simply the maximum allowable value of crossing degree for a particular LFG (language) and a correspondence is said to be "strictly nested" if its crossing degree is zero. A correspondence is "nearly nested" if its crossing degree is within the grammar's permitted crossing limit.

The F-structure produced from the example movement phrase is shown in Figure 2.9.i. The function q is derived from a controller's F-structure which in turn is derived by matching a controllee in the lexical entry for *what*. Both of the bounded metavariables in this case have the same subscript *[+wh]* which is used for interrogative movement. The focus function (which is derived from an immediate dominance variable on the C-structure's first *np*) has the same value as function q but in other cases (where the interrogative determiner *what* is embedded in a more complex *np*) the values of the focus and q functions will differ. The various values derived from instantiating bounded domination variables (constituent control) are linked by lines in the same way as those derived by functional control.

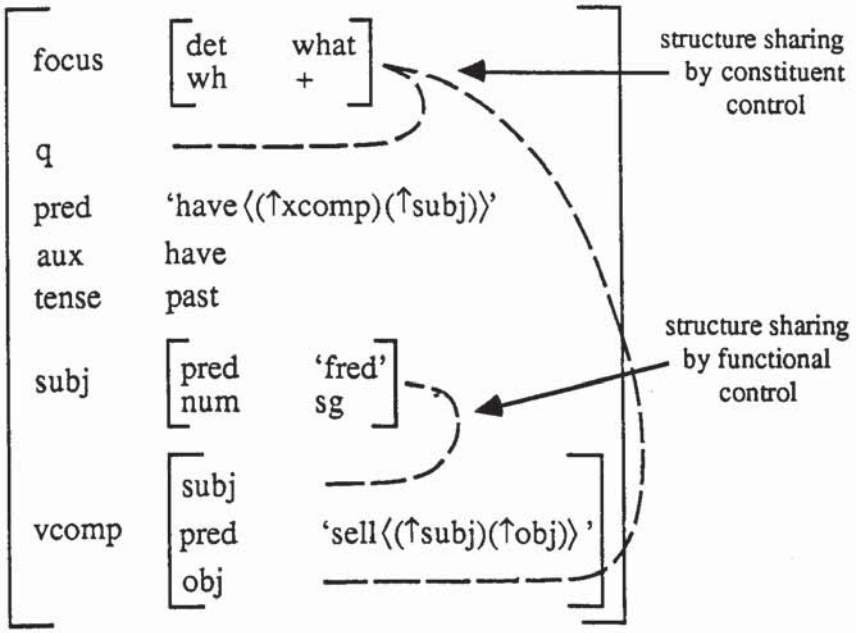


Figure 2.9.i F-structure Produced from Movement

Further conditions must be imposed to account for controller/controllee correspondences in long distance control instances. These state that a string is grammatical only if its F-structure is properly instantiated.

2.9.1 Definition of Proper Instantiation

An F-description produced from a C-structure with attached schemata is properly instantiated if and only if:

- no node is a domain root for more than one controller.

- every controller metavariable has at least one control domain.

- every controller metavariable corresponds to one and only one controllee in each of its control domains.

- every controllee metavariable corresponds to one and only one controller.

- all metavariable correspondences are "nearly-nested".

A metavariable correspondence is nearly-nested if its crossing degree does not exceed the grammar's crossing limit. The significant formal implication of this is that for any string, the degree of departure from strict nesting is bounded by a constant that is independent of the length of the string. Kaplan and Bresnan [1982, p262] suggest that for English, the crossing limit is zero.

2.9.2 By-Passing Bounding Nodes in C-structure

In certain special circumstances, bounding nodes are found to prevent the recognition of well-formed phrases. Kaplan and Bresnan [1982, p252] state that:

> " In contrast to the non-uniform bounding characteristics of *NP*s, it can be argued that in languages like English, Icelandic, and French, all *S*'s are bounding nodes. "

In certain cases however, a bounding *S* node would prevent the recognition of well-formed *S* constructions. In these cases, LFG provides a special equation, which is called here a "linking equation", of the form:

$$\Uparrow = \Downarrow^{dom}$$

where *dom* is the domain root of the controller to which the equation is attached. This equation can be thought of as an abbreviation for an equation:

$$\Uparrow_c = \Downarrow_c^{dom}$$

The linking equation is attached to the bounding node in the grammar rules. If constructing C-structure top-down, the equation may be thought of as matching a controller ('$\Downarrow$') from above the bounding node with its own controllee ('$\Uparrow$') and then

generating a new controller ('⇓'), with the same subscript *c* as the matched controller, which is passed into the domain (*dom*) below the bounding node (of category *dom*). An example where such an equation is required is the phrase [Kaplan & Bresnan, 1982, p252] :

'The girl wondered who the nurse claimed that the baby saw.'

To uncover the underlying structure of this sentence requires the movement of *who* into the domain of the *s* (*'the baby saw'*), where it fulfils the role of object for the verb *see*. In order to achieve this, Kaplan and Bresnan propose the rule :

$$s' \longrightarrow (\text{that}) \quad \boxed{s}\ \uparrow = \downarrow \quad \Uparrow = \Downarrow^{s}$$

This allows a single controller to penetrate the domain of *s*, if the *s* constituent is preceded by the word *that*. There are other circumstances when it is desirable for a controller to be passed across a bounding node. One of these, to be described later, is found in coordinate structures.

2.10 LFG Semantic Component

LFG has its own notion of a semantic component which is represented by the semantic forms found in F-structures. Semantic forms however, only appear in the lexical entries for nouns, verbs and adjectives in Kaplan and Bresnan's account of LFG [1982] and only verbs and adjectives appear to take functional arguments. A semantic form's value consists of an atomic term and then zero or more references (using immediate dominance variables) to functions in the enclosing F-structure, all enclosed in angled brackets. If no functions are subcategorized then the angled brackets are omitted. The order of subcategorized functions enclosed in angled brackets denotes surface grammatical relationships, not the underlying logical relationships (agent, theme, instrument) commonly represented in deep transformational models. For example, the verb *hand* allows :

'hand⟨(↑subj) (↑obj2) (↑obj)⟩'	(a girl handed the baby a toy)
'hand⟨(↑subj) (↑obj) (↑to obj)⟩'	(a girl handed a toy to the baby)

This limited semantic representation is clearly insufficient for database querying but might be extended to encompass other categories and matters of quantification.

2.11 The Computational Complexity of LFG

The transformations used in TG have been replaced in LFG by lexical rules (passivisation, dativization etc) and a more general movement mechanism (bounded domination metavariables). Berwick [1981] concludes that : " The elimination of transformational power naturally gives rise to the hope that a lexically based system would be computationally simpler than a transformational one. " This optimism has partly been responsible for the current trend toward lexically based grammars and the design of grammars such as LFG. However recent findings have shown that, in the worst case, LFG may have a recognition time which is very likely to be computationally intractable.

Kaplan and Bresnan [1982, p271] show that LFG can describe some context-sensitive languages. They observe that context-sensitive power is required in the analysis of constructs such as can be found in Dutch, Mohawk and the English 'respectively' construct. LFG's context sensitive power is shown to originate from the functional composition and unification equations associated with the basic CFG. Kaplan and Bresnan [1982, p271] state that (despite the linguistic pressures for context-sensitive power) :

> " On the other hand, the problem of recognizing languages with context sensitivities can be computationally much more complex than the recognition problem for context-free languages. If our system turns out to have full context-sensitive power, then there are no known solutions to the recognition problem that require less than exponential computational resources in the worst case. "

A more conclusive investigation into LFG's formal properties has been conducted by Berwick [1981]. The method relies on showing that a problem with a known complexity can be transformed into the problem of which a complexity rating is required. The problem with known complexity is that of determining the satisfiability of Boolean formulas which are in Conjunctive Normal Form (CNF). A CNF expression is a simple conjunction of disjunctions which, for the purposes of this proof, is restricted so that each term is a disjunction of just three terms (called 3-CNF) :

$$(X_2 \vee X_3 \vee X_7) \wedge (X_1 \vee \bar{X}_2 \vee X_4) \wedge (\bar{X}_3 \vee \bar{X}_1 \vee \bar{X}_7)$$

This restriction entails no loss of generality is known to be a problem of NP-complete complexity (see below). The 3-CNF above is solved (evaluates to true) if X_1 = True, X_2 = True, X_3 = False, X_4 = False and X_7 = False. The details of the extensive proof will not be repeated here, but briefly each X_i has a separate lexical entry for true and false :

X_i	(↑ truth) = true (↑ vars X_i) = true	(entry for X_i if X_i is true)
Xi	(↑ vars X_i) = false	(entry for X_i if X_i is false)

Each X_i variable (grammar terminal) in the 3-CNF problem can thus be assigned a value of true or false, but inconsistent value assignments across conjunctions are filtered out as unification does not allow a feature to have more than one value. The lexical entry for the true value of a variable in the 3-CNF has a feature called truth and each disjunctive term must have at least one variable with value true to satisfy a semantic form (which subcategorizes this value) introduced as a dummy terminal in the C-structure (Figure 2.11.a).

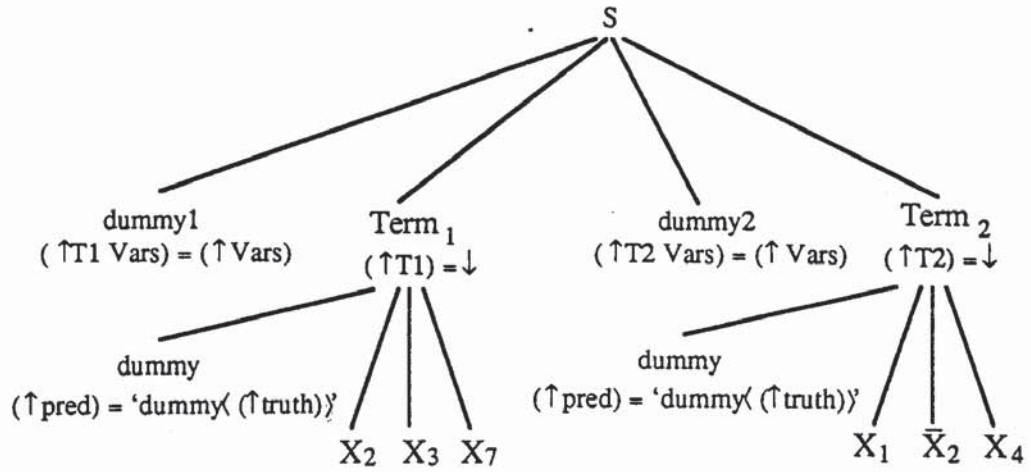


Figure 2.11.a Outline of 3-CNF Solution Using LFG (two conjunctions)

The F-structures produced from each conjunction of the 3-CNF are assigned as functions (T_x) in the top-level F-structure so that their semantic forms do not cause unification failure. The individual variable assignments from each conjunction are however passed up to the top-level of the F-structure by functional control equations attached to another set of dummy terminals in C-structure (dummy1 and dummy2 in Figure 2.11.a). This results in all variable assignments being passed up to the top-level of the F-structure where they are

unified, thus ensuring consistency of variable values across the conjunctions (Figure 2.11.b).

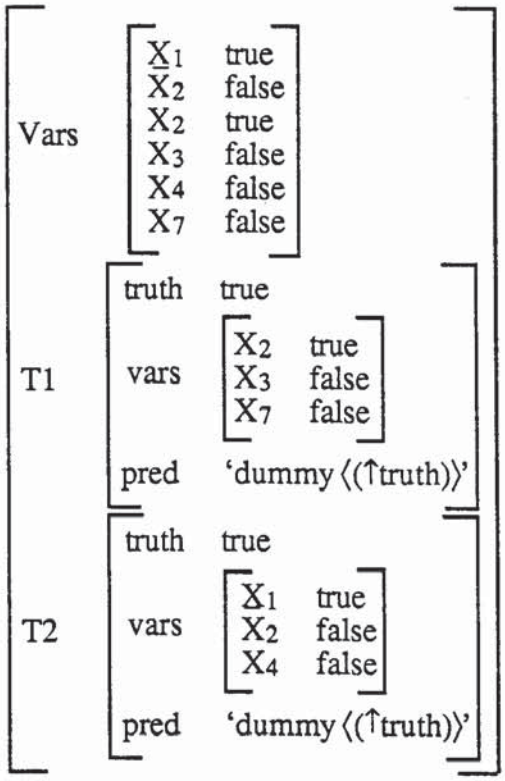


Fig 2.11.b F-structure Produced from 3-CNF Expression

A problem such as 3-CNF is described as NP-complete in being a problem that can be solved by a non-deterministic Turing machine in polynomial time (hence NP) and is complete in that all other languages in the class NP can be reduced to a CNF formula. Berwick's proof thus shows that recognition using an LFG grammar is at least as hard as a NP-complete problem but this is not to say that LFG recognition may not be even harder than this. The fact that LFG recognition is in the class NP (NP-hard in complexity theory) implies that in the worst case LFG will be unsuitable for computational purposes. However there are a number of restrictions, described by Berwick, which can be applied to remedy this situation.

Firstly it is noted that intractability arises because co-occurrence restrictions (in F-structures) may be applied across arbitrary distances in C-structures. If some device could be found so that ambiguity were resolved more locally then LFG's complexity would be reduced. Secondly locality principles might be applied to specific LFG language grammars to ensure efficient recognition or lastly the number of lexical items can be restricted as Berwick states : " The reduction [proof of complexity] depends crucially

upon having an infinite stock of lexical items and an infinite number of features with which to label them. ". The last of these is perhaps the most significant here (it is considered not really desirable to restrict the grammar for computational purposes) as the use of lexicons in the interface system here implies a limited number of lexical items. Also implicit in this is the use of a limited number of features. However both of these limitations are only due to practical considerations (the result of 'writing down and using a grammar and lexicon') and do not imply that a 'complete' lexicon or set of features can be produced.

Chapter 3

A Semantics of F-structure

LFG provides a primarily syntactic analysis of phrases, ensuring agreement and general syntactic well-formedness. The F-structure output of an LFG analysis does however identify surface semantic roles (subj, obj, xcomp). Having performed a syntactic analysis and produced a well-formed F-structure, it is then necessary to translate this F-structure into some semantic representation of the phrase. The application of database access demands a practically orientated representation rather than a powerful or cognitively orientated representation. A number of different semantic representations have been developed in the past (modal logic, lambda calculus, intensional logic) but, of these, variations of first-order logic have proved most practical for database querying. This type of logic limits the phenomena which may be expressed to the same type of information which is found in a simple relational (Prolog) database. Other phenomena, such as temporal information [Hafner, 1985; De, Pan & Whinston, 1985] and events [McCord, 1987] which may be expressed in more powerful logics, are not usually represented in simple relational databases and are still a matter for research in that field.

Here, a semantic theory suitable for database querying is to be developed which will draw upon the experience of other database querying systems developed in the recent past. All previous transportable systems of this type have used a syntactic analysis as the basis for semantic translation. As a functional analysis is to be translated, the explanation of the theory will not be complicated by considering the shortcomings of previous theories.

3.1 Halvorsen's Semantic Theory of F-structure

Consideration of semantics in Bresnan and Kaplan [1982] does not extend past the notion of semantic forms. Halvorsen [1983] has however proposed a semantic theory for LFG. His aim is to begin the construction of a universal interpretation procedure for F-structures. Halvorsen's theory takes an F-structure and produces first, a 'semantic structure' and then translates this into an intensional logic expression. The system employs five levels of representation in all (C-structure, F-structure, semantic structure, intensional logic and a model theoretic interpretation). The level of intensional logic is included only for convenience of comparison with other work by linguistics in the

Montague (logic and grammar) tradition. Whilst intensional logic is a well-founded and well-explored representation, based on lambda calculus, it is not well-suited for use in database querying.

Halvorsen's intermediate semantic representation is a 'reduced' version of an F-structure where only meaning bearing elements are represented and representations are converted into a form suitable for the production of intensional logic expressions. Halvorsen describes a set of rules for translating F-structures into his semantic structures. The detailed construction of Halvorsen's semantic structures will not be described here as the semantic theory described here does not include this level of representation. Instead F-structures themselves will be given a more extensive semantic content than that of basic LFG F-structures and can be seen to contain the basic elements of Halvorsen's semantic representation, although simple predicate logic is used instead of intensional logic as this is more suited to the database querying application. It should be noted that as Halvorsen states [1983, p572]:

> " It is perfectly feasible to construct a composite function, Φ, whose components are the function mapping C-structures to F-structures and the function mapping from F-structure to semantic structure. Φ would take us directly from surface structure to meaning representation without constructing an intermediate level of representation such as F-structure. Eliminating a level of representation in this fashion is not hard. Similarly, it is not hard to construct a theory with an arbitrarily small or large number of levels of representation. The crucial issue is not number of levels involved, but whether or not the levels proposed can be shown to elucidate the description of distinct sets of phenomena. "

In the work described here, it has been chosen not to eliminate F-structure as the production of this facilitates well-formedness checking and it is a useful illustration of any analysis for debugging (grammar and lexical entries). Also a direct mapping from surface structure to semantic representation would require fully integrating syntax with semantics: a practice about which there is no general agreement among linguists and which may not be computationally advantageous in the face of ambiguity. In addition to this, to remove the functional representation stage seems to depart from the spirit of a functional formalism. It remains undecided [McCord, 1987] as to whether it is most computationally efficient to interleave a full semantic interpretation with syntactic analysis or whether these

should be performed in series. If it were chosen to derive a semantic representation directly from C-structure (equations and lexical semantic components), as Halvorsen suggests might be done, it is still open as to how the well-formedness of a string can be ensured, given that not all elements of an F-structure which are used in checking well-formedness have a semantic relevance. So while much of F-structure content could be directly transformed into a semantic representation, it would still be necessary to retain some additional F-structure content to ensure well-formedness.

Several points discussed by Halvorsen are however relevant to the semantic interpretation theory of F-structures being proposed here. Most importantly, the relationship between a semantic form's subcategorized functions (functional arguments) and the semantic translation of a semantic form into logic is considered. In his discussion, Halvorsen examines semantic forms in the well known active/passive transformation. As an example of this, consider:

(1) Brian kicked the student. (active)
(2) The student was kicked by Brian. (passive)

The F-structures for these two phrases (with pointers to subcategorized subsidiary F-structures) are shown in Figure 3.1.a and Figure 3.1.b (overleaf).

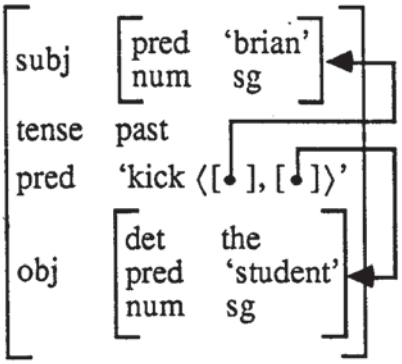


Figure 3.1.a Outline F-structure of Active Phrase Example *(1)*

Notice that in these F-structures (Figures 3.1.a and 3.1.b), the central semantic form of *kick*, although having different functional arguments, fills the functional roles of its arguments with the same subsidiary F-structures in the same order. That is to say, that if a semantic form 'kick⟨(↑ A) (↑ B)⟩' is taken to imply that the entity represented by

subsidiary F-structure *A* performed the act of kicking the entity represented by subsidiary F-structure *B*, then in both active and passive cases, the correct thematic relation is represented by the semantic form.

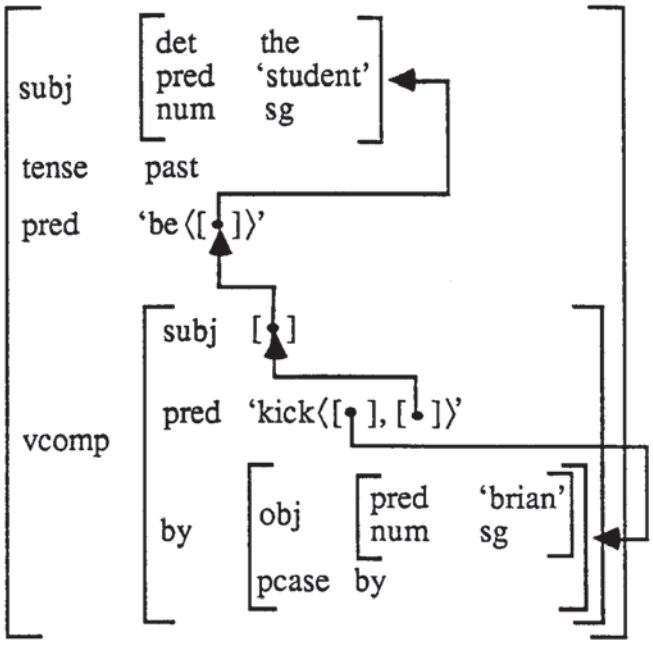


Figure 3.1.b Outline F-structure of Passive Phrase Example *(2)*

In accordance with this notion, the semantic forms for active and passive :

'kick ⟨(↑ subj) (↑ obj)⟩'
'kick ⟨(↑ by obj) (↑ subj)⟩'

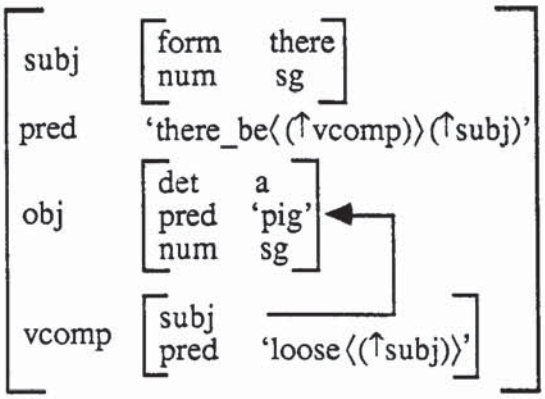
can be translated in the same way regardless of the variance in subcategorization. Winograd [1983, p348] also noted that "These forms are intended to be part of the semantics - they follow conventions for predicate logic and could be used for inference". This is significant for LFG in that it acknowledges LFG's surface grammatical relationships (subj, obj, vcomp) as a sufficient basis for (limited) semantic interpretation rather than requiring a deeper analysis of relationships, such as that found in TG.

Halvorsen also examines the semantic translation of F-structures produced from phrases displaying "there-insertion". As discussed in Section 2.2, there is given a semantic form which makes no semantic contribution. As a general rule, Halvorsen proposes that in cases of there-insertion :

" .... the key to interpreting all the examples above [there-insertion constructions] lies in isolating the semantically relevant aspects of their F-structures by simply eliminating syntactic expressions without independent meaning. "

Consider the following case of there-insertion :

'There is a pig loose.'



At the top level of the F-structure neither the subj function or semantic form are semantically relevant. In addition to this, the obj function plays a functional role of subj within the vcomp function. Thus, the full semantic content of the F-structure lies within the vcomp function and only this need be translated. The principle of isolating semantically relevant portions of F-structure can be applied in other cases, such as idioms ('keep *tabs* on').

Halvorsen also briefly approaches the problem of quantification. It is proposed to use a generalization that the relative scope of quantifiers is given by their linear precedence (surface order) in active passive pairs. Linear precedence, it is suggested, can be represented by indexing C-structure nodes and then using rules to ensure that scoping in the semantic representation is 'legal'. As an example of such a rule Halvorsen examines :

(1) 'each student admires no politician.'
(2) 'no politician is admired by each student.'

In the first case *(1)*, *each* is given scope over *no* and in the second case *(2)* the quantifier scopings are reversed. Halvorsen claims simply that this illustrates the general fact that if *each* and *no* occur in a sentence, the first quantifier tends to scope over the second.

Quantifiers are indexed in C-structure from left to right and a semantic rule is then employed to ensure that the indexes of quantifiers given highest scope are less than those given a lower scope. This rule cannot however be extended to active/passive pairs involving *every* and *a*, in which case sentences are taken to be truly ambiguous :

'every man loves a woman.'
'a woman is loved by every man.'

Halvorsen's approach to quantification obviously requires actually producing a C-structure representation of a string so that indexes can be used. This however is not really in the spirit of LFG, where F-structure is the sole input to semantics and functional roles are analysed, rather than surface structure (order). In addition to this, as will be shown later, the general rule relating to surface order proposed by Halvorsen, is not applicable in all quantification cases.

3.2 Logical Semantics for Database Access

A number of techniques have emerged from work using logic grammars which have become generally accepted as effective components of a semantic theory, at least for database access. Two of these techniques are the use of slot frames with types and the reshaping of syntactic derivation trees to reflect quantifier scoping. Both of these techniques are employed in Chat-80. In addition to these techniques, Dahl [1979, 1981] has argued for the use of a three-valued logic in representing quantifiers. Chat-80 has however illustrated that this extension is not required to provide a useful system. Some of these techniques can obviously be employed directly in the translation of F-structure (slots and typing) but others (tree reshaping) are not applicable to the translation of F-structure and equivalent but new techniques must be developed.

3.2.1 Slot Frames and Typing

McCord [1982] has described a method of attaching argument descriptions to the lexical predicates of referring words (nouns, verbs and adjectives). An argument description is called a "slot" which imposes constraints on the modifiers which a particular word may have. A slot contains a complement name, called the slot identifier (subject 'subj', object 'obj', indirect object 'iobj', prepositional object (with case 'on')

'pobj(on)'), an argument of the words corresponding logical predicate and a domain "type" :

<slot identifier> : <predicate variable> : <domain type>

The set of slots describing a predicate's arguments is called a "frame"; the verb *give* might have a lexical entry (with the frame as a list) :

```
verb(give, give_action(S, O, Io),
                [subj : S : human, obj : O : phys_obj, Io : human] ).
```

where 'give_action(S, O, Io)' is the logical predicate representing a particular sense of the verb (as all finite verbs take a subject, the identifier for this slot can be omitted). Nouns and adjectives are given similar frame descriptions. Nouns such as *dependence* are given frames derived from the corresponding verb (*'to depend on'*), whilst other relational nouns such as *man* are given single slots relating to the subject :

```
noun(dependency, depend_event(Of_obj, On_obj) : E,
        [subj : E : event, pobj(of) : Of_obj : human, pobj(on) : On_obj : _] ).

noun(man, man(M), [subj : M : human] ).

adjective(green, green_coloured(T), [subj : T : object] ).
```

A slot identifier names the surface syntactic function which a predicate may take but says nothing about the expected semantic content of modifiers. The use of types is a simple and elegant way of describing this semantic content (meaningfulness) [Dahl, 1981; Brand, 1986]. Types are also a useful way of associating the universe of predicate calculus to the relations of a particular database domain and improving efficiency by :

- narrowing the search space, as only those values in a variable's associated domain (or type) need be considered.

- avoiding futile access to the database, as absurd queries can be rejected by the analyser on the grounds of domain (type) incompatibility.

Types may also provide an efficient means for discarding readings that are syntactically acceptable but semantically incorrect. Consider for example :

'what is the salary of the employee who lives in London ?'

Syntactically, there is no way to determine whether the antecedent of the relative clause '*who lives in London*' is '*the salary of the employee*' or '*the employee*'. With the aid of type checking the first argument of the relation *live* is associated with the human domain in which employees and not salaries are known to belong (that is a salary cannot live in London but an employee can). Types can also be used to place modifiers other than relatives. Dahl's system does not do this but it does check that the modifiers it encounters are of the expected type. Ambiguities concerning different meanings of a word can also often be resolved through domain type checking.

The expected types of a predicate's arguments are given in lexical entries. Ambiguous words will thus have a lexical entry for each possible combination of meaning and syntactic role they can accept. For example, the adjective *blue* may have the entries :

adjective (sad(person-x)) = blue.	(the mood 'blue')
adjective (blue(object-x)) = blue.	(the colour 'blue')

During parsing, the correct parse is automatically chosen by matching appropriate terms with types, for example :

'which blue door is John painting.'

would generate a formula containing predicates of the form :

door(t-z), paint (person-john, t-z), p(t-z)

where the predicate *p* is either *sad* or *blue*. The lexical rules allow *t* to take a value only if that value is compatible with the types defined for all three predicates. Given that the predicate *door* for example only takes 'object' type arguments, the predicate *p* must be *blue* for the assignment of type *t* to be consistent.

Types are represented by expressions that reflect subcategorization and allow for domain intersections to be found automatically. The Prolog interpreter itself matches type expressions by unification, which are of the form :

nil & type & $type_1$ & $type_2$ .... & $type_n$

where $E(type_x)$ is a set of types such that :

$$E(type_n) \supset .... \supset E(type_2) \supset E(type_1) \supset E(type)$$

In general then, the longer a type expression, the more specialized that type will be and the type *nil* can be seen as the most specialized domain type. For example, the type :

T & employee & human

may be unified with any type contained in or equal to the *employee* type, for example *T* may be unified with :

nil or nil & salesman or nil & manager

The use of types while parsing represents a degree of interleaving of semantic and syntactic analysis (a subject about which there is no general agreement) but the simplicity of this semantic component (typing) and the advantages of its use certainly seem to outweigh the arguments for delaying this analysis until a post-parsing stage.

3.2.2 Three Branched Quantifiers and Presupposition

The evaluation of a logical expression can obviously either fail or succeed but a third logical truth value would be useful because in NL there are two ways in which a statement may fail to be true :

(a) its negation holds (is provable).

(b) something presupposed by the statement fails to be satisfied (not provable).

In the latter case *(b)*, the logical representation is regarded as having a value 'pointless' rather than (proved) false. As an example, consider the phrase :

'The mad hatter hates Alice.'

In this example, if no hatter is mad then this is obviously not true. However if considered false then :

'The mad hatter does not hate Alice.'

would have to be considered true, if only two logical values are available. The non-existence of a referent for the definite article noun phrase makes the whole sentence pointless. This is because the singular definite article introduces a "presupposition" of existence and uniqueness on a noun phrase's referent.

Dahl's treatment of quantification has been devised to account for the presuppositions implied by NL quantifiers. If a sentence contains a determiner then a quantification of the form :

those(x, p)

is introduced, where *x* is a typed variable and *p* is a logical formula. Its evaluation yields the set of all *x*'s in *x*'s associated domain which satisfy *p*. According to the determiner's meaning, presuppositions about the cardinality of such a set are represented within the logical representation :

'Three blind mice.'

equal(card(those(x, and(mice(x), and(blind (x), run(x))))), 3).

Definite articles introduce the formula :

if(f1, f2).

the value of which is pointless whenever *f1* fails to be satisfied, and has the same value as *f2* if *f1* is true, *f1* thus representing presuppositions (in italics below) :

'The mad hatter hates Alice.'

if(*equal(card(1, (those(x, and(hatter(x), mad(x)))))),*
hate(those(Alice (x, and(hatter(x), mad(x)))))).

An alternative approach to false presupposition detection is the pragmatic one, in which false presuppositions are caught by noting their empty extensions in the database. In this way a two valued logic can be preserved.

3.2.3 Semantics in the Chat-80 system

Semantic interpretation in Chat-80 produces a first-order predicate logic expression which can be executed in Prolog to query a Prolog database. The use of more powerful logics, such as modal or intensional logic, is ruled out [Pereira 1982, p118] as the practical requirements of database access require that the logic representation has a readily executable proof procedure available in Prolog. This is not readily possible in the case of certain semantic phenomena which can be expressed in these more powerful logics. The first-order logical expressions produced by Chat-80 contain :

- conjunction 'p & q', where the conjunction holds true simply if both *p* and *q* hold true.

- existential quantifiers 'exists(x, p)' which holds true if there is some instance *y* of *x*, such that if all occurrences of *x* in *p* are substituted by *y* then *p* holds true.

- non-provability '\+p' which holds if *p* is not provable.

- set expressions 'set(x, p, s)' which holds if *s* is the finite non-empty set of instances of *x* such that the corresponding instances of *p* are provable.

- *all*, which is actually reduced to an existential quantifier and non-provability by equating :

  all(x, p => q) to \+exists(x, P & \+ q)

- *numberof*, (number of elements in a set) which is also reduced to an existential quantifier and non-provability by equating :

  numberof(x, p, n) to exists(s, setof(x, p, s) & card(s, n))

  where *card* counts the number *n* of members of a finite set *s*.

The role of existential quantifiers is connected with the use of negation and *setof/3* predicates. Outside the scope of these, existential quantifiers are eliminated through the equalities :

p <= exists(X, q) <=> all(X, p <= q) <=> p <= q

Within the scope of *setof/3* or *\+/1*, the formula 'exists(X, p)' is replaced by 'p1(V_1, V_2 ....., V_n)' where V_1, V_2 ..... , V_n are all the free variables in *p* and *p1* is defined by :

$$p1(V_1, V_2 ....., V_n) <= p.$$

That is to say existential quantifiers inside of a *setof/3* predicate are proved by proving that some value exists for the variables of quantifiers, where all the values for those variables are consistent.

Pereira [1982, p43] gives an account of how the definitions of both *setof/3* and non-provability (*\+/1*), which are defined in terms of provability and not truth, can be mapped into first-order formulas. For the purposes of execution in Prolog, the existential quantifiers are used to prefix the *setof/3* predicate which must not contain any free variables not subject to quantification. This is however not true of the *findall/3* predicate which has the same semantic definition as *setof/3*, except that free variables may occur within the goal and the goal succeeds even if the solution set is empty. Warren's definition of *setof/3* fails if an empty set is returned. In practice, this can lead to some unreasonable responses (see query execution Section 8.3).

Determiners themselves are given the following logical translations, where roughly a variable *X* corresponds to an explicit or implicit *np*, the range *R* of the variable to the translation of the words in the *np* and the scope *S* to the translation of the rest of the sentence where the *np* occurs :

- a, some, the (singular)	exists(X, R & S).
- no	\+ exists(X, R & S).
- every, all	\+exists(X, R & \+S).
- the (plural)	exists(St, setof(X, R, St) & S).
- one, two, ... numeral(N)	numberof(X, R & S, N).
- which, what	answer(X) <= R & S.
- 'how many'	answer(N) <= numberof(X, R & S, N).

Other words which have arguments, filled by quantifier variables, (verbs, nouns, adjectives) are also marked for case and type. Each argument is represented by a tuple called a slot after McCord's slot filling grammars [McCord, 1980, 1982, 1985a, 1985b, 1987] :

slot(case, type, argument)

A case marker is a preposition or verb argument role (subject, object) and a type is the most general class of domain entities which may fill the slot. A word is then given a lexical entry for each of its meanings. The verb *flow* thus has an entry :

```
verb(flow,  flows_through(R, C),
        [slot(subject, river, R), slot(prep(through), country, C)] ).
```

with meaning 'flows_through(R, C)', where *R* corresponds to the first slot's argument which has a subject role and type *river* and *C* corresponds to the second slot's argument value which is case marked by the preposition *through* and has type *country*. Pereira [1982] recognizes some redundancy in lexical types and generalizes the genitive case *of* when used to mark the argument of a property such as *area* or *population*. A general slot description is then produced for these words :

```
noun(Word, Type, Val, Pred, [slot(gen, Arg_type, Arg)]) :-
                property(Word, Type, Val, Arg_type, Arg, Pred).
```

defining a noun *Word* as naming a property *Pred* with value *Val* and type *Type* that takes a genitive argument *Arg* of type *Arg_type*. A template is then defined for each noun with a genitive slot. The template for *area* is :

property(area, measure, A, region, R, area(R, A)).

which defines the property *area*, produced by the predicate 'area(R, A)', as a value *A* of type *measure* having a slot with value *R* of type *region*.

The property template for the noun *area* also illustrates that some predicate arguments are not slots themselves (here the range *R*) but are bound to the quantifier variable associated with the *np* of which the noun is the head. The *np* 'the population of France' is thus translated into :

the(λ(A) . area(france, A), ...)

where *A* is the *np* variable which like any other argument is given a type. The occurrence of arguments which are not slots will have important consequences in terms of a semantic translation of LFG F-structure described later. Types in Chat-80 are based on Dahl's work and dealt with in the manner described in Section 3.2.

Strangely, Chat-80 allows slots to be left unfilled. This simplifies the treatment of passives. Slot filling rules allow the subject slot of a transitive verb to be filled by a phrase case marked with the preposition *by* if the sentence is passive. If the *by* phrase does not exist then the slot is left unfilled and the corresponding argument variable existentially quantified. In this way, unfilled slots are assumed to be optional and their corresponding argument places filled with existentially quantified variables. Pereira [1982, p131] does however note that this approach is too simplistic as the sentence *'the door is closed'* will be given the same meaning as *'the door was closed by something'* (tense variation ignored). The concept of slot filling corresponds to the notion of subcategorization in LFG and allowing unfilled slots corresponds to breaking the condition of completeness. It seems likely that slots are allowed to be unfilled in Chat-80 to reduce lexical ambiguity which can severely reduce the efficiency of parsing when using Prolog's TDB directly.

In addition to simple predicates, Chat-80 provides a treatment of adjectives and higher-order operators suitable for grammars applied to querying a database. Adjectives which simply refer to some set of entities which intersects with those referred to by the *np* in which they appear are simply translated as extra predicates restricting the range of the variable they share with the rest of the *np* (usually the *np*'s quantifier variable) :

'the green book' the(B, green(B) & book(B))

Comparative and superlative adjectives are taken as implicit references to some property of the head noun where the property values must belong to some ordered set for the comparison or the finding of a superlative to be possible. For example, the adjective *larger* is understood as referring to areas when applied in the context of countries, so that the phrase :

'every country larger than France ....'

is roughly translated as :

```
all(C,  country(C) &
        exists((A1,A2),  area(C, A1) &
                         area(france, A2) &
                         A1 > A2)   ....)
```

The actual interpretation of superlatives and comparatives depends on the entity type to which they are applied, so the lexicon will contain a number of entries for each adjective.

The semantic type of the objects to which adjectives are applied can be used to select among these alternative entries.

Comparative adjectives (*larger, smaller*) can, as shown in the example above, be treated as special cases of intersective adjectives. They simply add a number of extra predicates to the logical expression. Superlatives on the other hand, apply to the entire *np* in which they are found. The superlative *largest* in the phrase *'the largest country in Africa'* selects a single country from the set of all African countries. When applied to countries, *largest* is taken as referring to the area of the countries. Superlatives and other non-intersective adjectives are called "aggregations" by Pereira [1982]. An aggregation is represented in Chat-80 as a second-order predicate :

$$\text{adj}(\lambda(R, V) . p, O) .$$

where *p* is the predicate(s) to which the adjective is to be applied, *R* the range, *V* the value the adjective operates on and *O* the selected superlative object. The noun-phrase 'the largest country in Africa' is translated roughly into :

$$\text{largest}(\lambda(C, A) . \text{country}(C) \,\&\, \text{area}(C, A) \,\&\, \text{in}(C, \text{africa}), L) .$$

where the predicate *largest* will select objects from the range *C* for which the value *V* is not smaller than the value for any other object. The aggregations *average* and *total* are treated in a similar way but calculate the average and total of the values over all objects respectively. In addition to aggregations certain words are treated in a special manner. These are words which Pereira describes as "words which look at their arguments".

In Chat-80, the verb *have* is dealt with by special rules which fit occurrences of *have* which match one of the sentence templates :

- <entity> has <attribute> of <value>
- <entity> has <value> as <attribute>
- <entity> has <attribute>

into a rough paraphrase :

<attribute> of <entity> is <value>

The rules in Chat-80 must then relate a slot filler *<entity>* functioning as the subject of the verb to the arguments *<attribute>* and *<value>* of the verb. The *<attribute>* must have an unfilled genitive slot which can be filled by *<entity>* and the type of *<attribute>* must be compatible with the type of *<value>*.

The syntactic analysis of phrases produced by Chat-80, as mentioned previously, is used to produce a first-order logic expression. The syntactic analysis is not however immediately translated into logic, first a "quant tree" is produced. This Pereira states [1982, p140] is not truly a semantic representation between syntax and logical semantics :

> " Quant trees, however, can only be seen as translations of sentences in the weak sense of being precursors of logic, which unlike Quant trees have a semantics and may therefore be seen as translations in a full sense. "

To produce a logical expression for a sentence involves combining the meaning of words together. Doing this in turn requires finding :

(a) the arguments of predicates (the modifiers of the word).

(b) the relative scopes of quantifiers.

There are of course several different possible approaches to both of these. Pereira lists three approaches to *(a)* above :

- the syntactic level describes all possible attachments, and a pragmatic level filters out permissible attachments.
- the concept of modifier attachment is built into the syntactic level in a way that makes it possible to use pragmatic notions in the analysis of the input.
- the syntactic level decides on attachments in a predefined way, from which a pragmatic level can recover other alternative attachments.

In Chat-80 the third approach above is taken. The syntactic analysis produces an initial set of modifier attachments for verb complements, prepositional phrases, relative clauses and adjective phrases. The initial attachments are performed in a rightmost (deepest) normal form. A set of slot filling rules can then reform attachments if the syntactic attachments

conflict with the case marking or type of a slot. This approach is taken to completely decouple syntactic analysis from attachment so that attachment does not cause backtracking during parsing, and to allow easier syntactic manipulation of constructs involving the verb *have*. This separation is in contrast to the work of McCord where slot filling is performed during parsing.

The quant tree is used as an intermediate representation upon which rules of modifier attachment and quantifier scope can act. A set of tree rewrite rules produces a quant tree from the parse tree by slot filling (attachment). This is then passed to another set of rewrite rules which alter the tree to reflect quantifier scoping. Modifiers for attachment fall into two classes : fillers, which fill a noun, verb or preposition argument place, and restrictors (adjectives), which specify some property of the modified entity. In a first-order logic representation, the objects which both modifiers and restrictions apply to are represented by quantified variables. Scoping is concerned only with quantifiers. The rules of attachment and quantifier scoping thus require a representation, different in nature to the syntactic derivation tree, which places quantifiers in a suitable position for scoping and identifies their variables for attachment. This is the function of the quant tree. The attachment rules do much the same as the slot filling process which has been described (case marking and typing is used). Quant trees will not be described in detail here but are made up of three types of nodes, the most important of which is the quantification node (Quant) which represents the translation of a noun-phrase. This type of node has fields for :

- the determiner, which will translate into a quantifier binding the variable of the node.
- the head, which is either the head noun's predicate or a term representing a higher order function.
- the predication, a sub-tree of restrictors modifying this node, any determiners in which have lower scope than the determiner of this node.
- modifiers, a set of trees representing all other modifiers of this node.
- the bound variable introduced by this node.
- the range variable, the variable restricted by the head, predication and modifiers, which represents the entities defined by this quant node. In certain cases (plural determiners represented by sets), this variable will be different from the bound variable.

It should be noted that the post-modifiers (lower nodes) of a node in a quant tree are divided into two classes "predication" and "modifiers" to reflect the different scoping properties of these post-modifiers.

The rules for quantifier scoping (Pereira calls these determiner scope rules) assume as a default the order of scoping given in a quant tree, which will roughly be their linear (surface) order. The scoping rules apply only to exceptional cases. Determiners are divided into two groups : "strong" and "weak". The determiners *each* and *any* are strong determiners, all others are weak. Generally then, a node with a strong determiner will be moved up the quant tree above nodes with weak determiners. Determiners in the predication of a node, which contains post-modifiers such as relative clauses, are not however allowed to be moved up from the node to which they are attached. The scoping rules are applied bottom-up so that a strong determiner may be moved repeatedly up the tree until it is blocked from further raising by another strong determiner.

The Chat-80 semantic analysis is very effective and efficient at producing first-order logic expressions, but is not applicable to F-structure translation. The default quantifier scoping is related to the derivation tree produced by parsing and thus relies heavily on quantifier order in the phrase or sentence. This ordering is not represented in F-structure. Most attachments are performed in LFG automatically by functional assignment although LFG itself does not perform type checking so that many different attachments may be produced. Some attachments such as set values of adjectives and adjuncts are not performed by LFG but left open for subsequent post-parsing stages.

3.3 A Semantics of F-structure for Database Access

3.3.1 Deriving Logical Expressions from F-structure

It has been shown that in some cases, LFG semantic forms parallel logical predicates but instead of variables and values as arguments they have function names as arguments. Semantic forms do not however have arguments related to quantifiers and LFG itself says nothing about quantifiers and quantifier scoping. For the purpose of database querying LFG must be extended to encompass these two notions.

Firstly, in some cases, the semantic forms of LFG must be adapted to the form of first-order logic predicates. The logical arguments of verbs will general correspond, given

the translation of F-structure to be described below, to their functional arguments (subcategorizations). In order to maintain a clear separation of LFG semantic forms and logical predicates, a lexical entry is given an additional "sem" feature describing its semantic predicate. The name of this predicate may not necessarily correspond to that in the semantic form. The semantic form's name can be regarded as a surface (or linguistic) semantic name and the predicate name as a deeper (or application) semantic name, related to the database domain. This also allows several different semantic forms to be explicitly mapped onto the same database predicate. The lexical entry for the verbs *border* and *surround* might thus be :

borders : v
(↑ pred) = 'borders⟨(↑ subj)(↑ obj)⟩'
(↑ sem) = borders(subj, obj)

surrounds : v
(↑ pred) = 'surround⟨(↑ subj)(↑ obj)⟩'
(↑ sem) = borders(subj, obj)

In the case of nouns with complements and simple restrictive words, where semantic forms omit arguments in first-order logic, these are variables bound by a quantifier, the sem feature may differ in its number of arguments from the semantic form. Arguments which are quantifier variables only appear in the sem feature not in the semantic form, as these are not subcategorized functions, and are given the name "quant" :

population : n
(↑ pred) = 'population⟨(↑ of obj)⟩'
(↑ sem) = population(quant, of obj)

man : n
(↑ pred) = 'man'
(↑ sem) = man(quant)

F-structure can also be extended to incorporate domain typing. In general each function and quant argument is viewed as having a domain type in the final F-structure. These types are assigned by the predicates of which they are arguments and assignments

must be consistent with a domain hierarchy of types. Types originate from lexical entries, in which special 'type' features can be specified. The feature name used for types is "domain". The verb *borders* might thus have a lexical entry :

borders : v
(↑ pred) = 'border⟨(↑ subj)(↑ obj)⟩'
(↑ sem) = border(subj, obj)
(↑ subj domain) = country
(↑ obj domain) = country

Lexical entries for words which have only a variable argument (not a subcategorized function name) refer to this argument as the feature named quant :

country : n
(↑ pred) = 'country'
(↑ sem) = country(quant)
(↑ quant domain) = country

The derivation of logical expressions from F-structures can now be introduced by examining an example. The phrase *'no man owns a trunk'* has a simple functional analysis but is ambiguous in that it has more than one logical interpretation (paraphrase). These interpretations come about by giving either the quantifier *no* or the quantifier *a* highest scope. The F-structure produced from the phrase is shown in Figure 3.3.1.a.

$$\begin{bmatrix} \text{subj} & \begin{bmatrix} \text{det} & \text{no} \\ \text{pred} & \text{'man'} \end{bmatrix} \\ \text{pred} & \text{'owns}\langle(\uparrow \text{subj})(\uparrow \text{obj})\rangle\text{'} \\ \text{obj} & \begin{bmatrix} \text{det} & \text{a} \\ \text{pred} & \text{'trunk'} \end{bmatrix} \end{bmatrix}$$

Figure 3.3.1.a F-structure for Phrase with Quantifier Scope Ambiguity

In this particular case, the functional arguments of the sem feature are exactly the same as those of the normal LFG semantic form, so the sem feature is not shown in Figure 3.3.1.a. This structure may be extended to include quantification (transformation can actually be made directly from F-structure to first-order logic). Firstly, a variable can be

associated with each of the two quantifiers and quantifiers represented as normal 3BQ's taking two formulas. A quantifier's basic form 'quantifier(Var, F & Fs)' may be described as translating the explicit or co-indexed subsidiary F-structure in which the quantifier is found and other semantic features of the subsidiary F-structure (chiefly semantic forms and function set values) as *F*, and other subsidiary F-structures which make up the rest of the main F-structure as *Fs*.

Quantifier variables may be represented as the value of a feature "var" in a quantifier's lexical entry. Restrictive predicates which have no functional arguments are given variables as arguments (man(V), trunk(V1)) which are expected to become bound to some quantifier. A "functional variable" is also associated with each of the F-structure functions (here subj and obj functions). These variables are then used to replace the main semantic form's (actually sem feature) arguments. This produces a new 'F-structure' :

$$\begin{bmatrix} \text{subj(M)} & \begin{bmatrix} \text{det} & \text{no(X,F1 \& Fs1)} \\ \text{pred} & \text{man(Y)} \\ \text{var} & \text{X} \end{bmatrix} \\ \text{pred} & \text{owns(M, T)} \\ \text{obj(T)} & \begin{bmatrix} \text{det} & \text{a(P,F2 \& Fs2)} \\ \text{pred} & \text{trunk(Q)} \\ \text{var} & \text{P} \end{bmatrix} \end{bmatrix}$$

Now the var feature values are unified with those of restrictive predicates and the semantic translations of subsidiary F-structures placed inside their respective quantifiers :

$$\begin{bmatrix} \text{subj(M)} & \begin{bmatrix} \text{no(X, man(X) \& Fs1)} \\ \text{var} \quad \text{X} \end{bmatrix} \\ \text{pred} & \text{owns(M, T)} \\ \text{obj(T)} & \begin{bmatrix} \text{a(P, trunk(P) \& Fs2)} \\ \text{var} \quad \text{P} \end{bmatrix} \end{bmatrix}$$

The variables of functions will now be unified with those of the var features inside of their respective subsidiary F-structures (*M* is unified with *X* and *T* with *P*). Two

choices are then possible given that a well-formed formula is to be constructed. There is a single quantifier inside of each function's F-structure and only a single predicate at the top level of the F-structure. The phrase is ambiguous in that we have a choice of either giving the quantifier in the subj function highest scope or giving the quantifier in the obj function highest scope. These two non-equivalent logical expressions correspond to the two different paraphrases of the phrase :

(a) **no**(M, man(M), **a**(T, trunk(T), owns(M, T)))

(b) **a**(T, trunk(T), **no**(M, man(M), owns(M, T)))

The extended F-structure thus represents both paraphrases, as did the original F-structure implicitly, and now quantifiers and variables lacking in the original F-structure are represented. It is obviously desirable that both paraphrases are represented.

Represented as a DAG, the transformations on the F-structure are illustrated in Figure 3.3.1.b. The first DAG *(1)* shows the initial content of the extended F-structure with 3BQs and var features added to functions.

In the second DAG *(2)*, the var features have been substituted for variables in restriction predicates which have quantifier variable arguments and additional variables introduced for each function. The translations on each subsidiary F-structure have also then been placed within their respective quantifiers.

In DAG *(3)*, the var features are unified with the variables associated with the functions in which they reside and the final DAG is produced for application of quantifier scoping rules. This DAG can then be transformed to produce either paraphrase *(a)* or *(b)* by either substituting the subj function's translation and main semantic form's value for Fs_2 or substituting the obj function's translation and main semantic form's value for Fs_1. The two final DAG's produced by these alternative interpretations are illustrated as the DAGs labelled *(4a)* and *(4b)*.

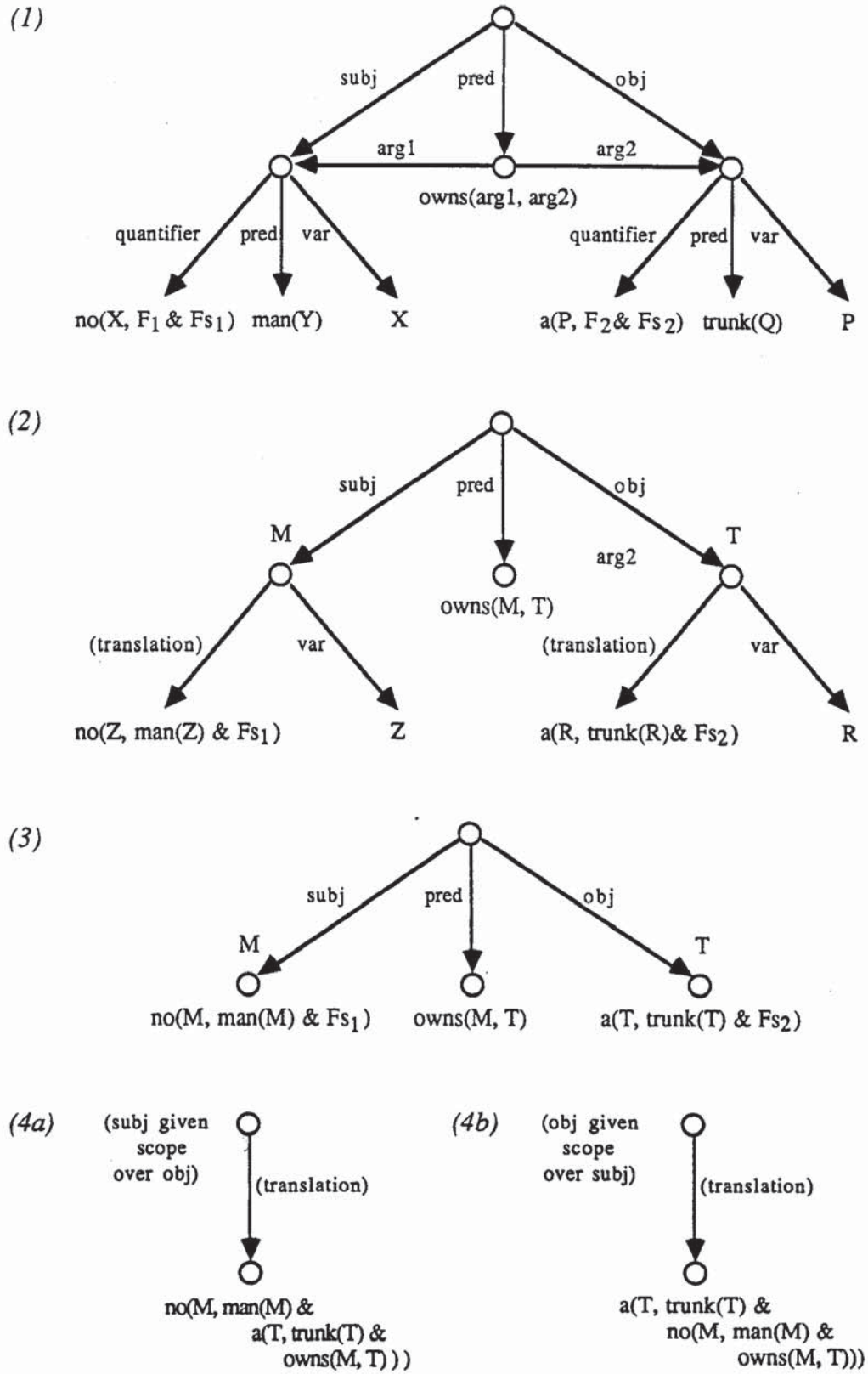
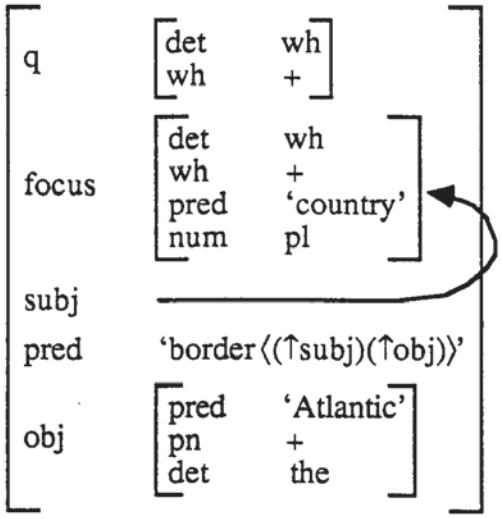


Figure 3.3.1.b DAG Transformations for Semantic Translation of Phrase with Quantifier Scope Ambiguity

A more complicated example involving movement can now be illustrated. This will illustrate the interaction of constituent control with translation and the translation of a Wh-front. The phrase to be translated is :

'which countries border the Atlantic ?'

from which the following F-structure will be produced :



Again the sem feature and semantic form have identical arguments and the semantic form is used in the description given here. The translation of this F-structure begins with the translation of the interrogative function q (see Section 4.1 for a discussion of the LFG analysis of Wh-fronts). It has already been outlined how this function is embedded in the function focus and in addition how the focus function, the Wh-front, will have been moved under constituent control to take up some other functional role in F-structure. Quantification is to be discussed later in this chapter but for now it must be noted that the function q is always given highest scope as a quantifier.

Having translated function q, it is then necessary to prevent the repeated translation of this quantifier when the function induced by constituent control is found. In order to do this, additional indexing of lexical entries with controllees is introduced. Now given that the q function is indexed, the focus function (which is also indexed to reflect constituent control) will contain some quantifier which is co-indexed with function q :

$$
\begin{array}{ll}
q & [\,det \quad wh\,]_1 \\
focus & \left[\begin{array}{ll} det & wh_{\,1} \\ pred & \text{'country'} \end{array}\right]_2 \\
subj & \left[\begin{array}{ll} det & wh_{\,1} \\ pred & \text{'country'} \end{array}\right]_2
\end{array}
$$

Indexes are actually only applied to entire subsidiary F-structures so that in the case shown above the indexes of the functions q, focus and subj will all be the same. If function q were embedded in the F-structure of the focus this would not be so; some function within the focus would be co-indexed with function q.

During translation a global list of indexes for function and their associated variables is passed around. An indexed function is then defined to be in one of two major states :

(1) bearing the first instance of an index, in which case none of the subsidiary F-structure will have been translated previously.

(2) bearing an index already encountered, in which case the subsidiary F-structure may have previously been :

(a) translated in its entirety.

(b) only the quantifier in the subsidiary F-structure has previously been translated.

(c) the function has not been translated at all.

In the example case, function q is translated immediately and its index and var feature value saved, noting that this is a quantifier that has been translated. The focus function is not translated but its index recorded with the same variable as function q. At this stage then, the focus is in translation state *(2c)* and q is in state *(2a)*. The subj function induced by constituent control and co-indexed with the focus function might be translated next. This will be given the same var feature value as the focus (and hence here also function q) and the quantifier in the subj will not be translated again as this function bears the same index as a function on the index list which has had its quantifier translated (q). Even if the quantifier of function q were embedded in the subj more deeply, inside of a function *F*, then the subj would still receive the var feature value of the focus (and hence also function q) and the function *F* would receive the var feature of function q (and hence also the focus) and the quantifier shared with function q in *F* would not be translated again.

The translation of this F-structure is illustrated in more detail by the DAG series show in Figure 3.3.1.c. The initial DAG *(1)* shows the initial extended F-structure with the q function's quantifier and its repetition indicated by enclosure in boxes. In this case, as described above, the function q is part of the focus, not embedded in the focus. Thus the same index will be present on functions q and focus in this case, and of course, the focus and subj functions will also be co-indexed as they are involved in constituent control.

In the second DAG *(2)*, the var feature of the two occurrences of the Wh quantifier have been unified; var features substituted into restrictive predicates; variables introduced for each function (focus and subj functions sharing the same variable) and these variables substituted into the main semantic forms (actually sem feature) argument's positions.

The third DAG *(3)* illustrates the final pre-logic state, where the variables representing functions (entities) have been unified with their corresponding subsidiary F-structure var feature values (the focus function which is not semantically translated has also been deleted).

The final DAG *(4)* is then formed by substituting first the translation of the subj function and then the main semantic form into the top level quantifier from q. The value of the obj function is unified with its functional variable (the entity this function represents), as proper nouns, which are given a special feature 'pn +', are simple constants in the first-order logic representation here.

The translation of function q is a special case of translating co-indexed subsidiary F-structures, complicated by the non-identity of q with its functional realization as part of some other function *F*. The function q is not reproduced as a separate function elsewhere but its value is unified with other features. In other cases, described later, constituent control moves subsidiary F-structures but these are more readily dealt with, through indexing, as the content of moved components is not unified with other additional information.

The translation of F-structure outlined above is obviously only applicable to very simple cases. More complex cases are discussed in Chapter 4.

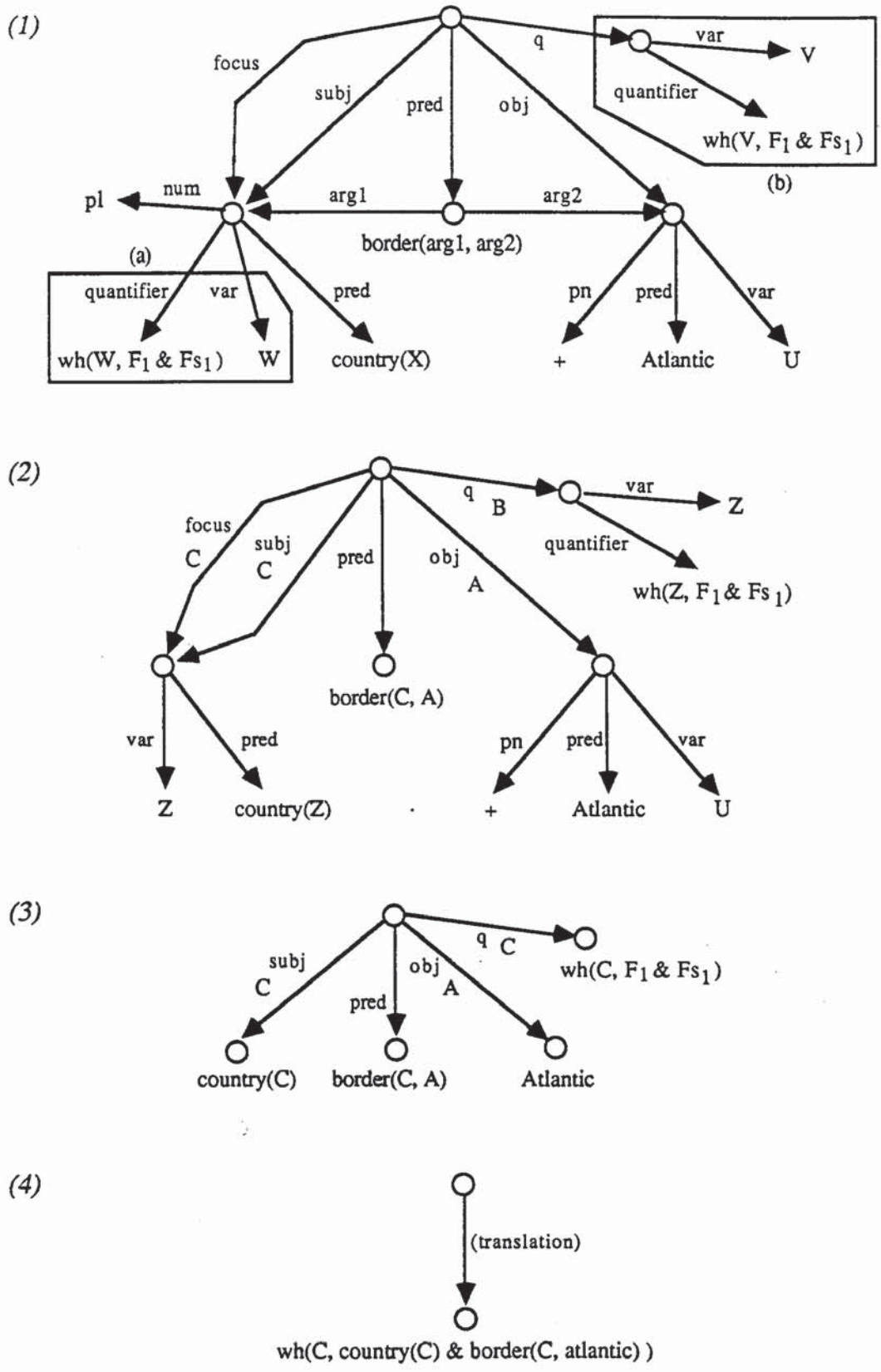


Figure 3.3.1.c DAG Transformations for Semantic Translation of Simple Interrogative

3.3.2 Quantification

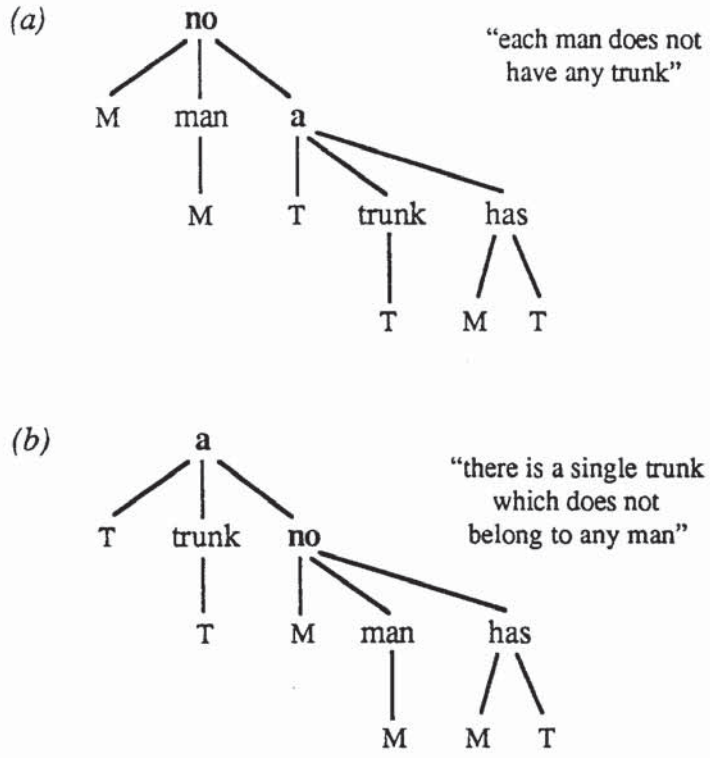
As well as Chat-80, several other database systems have been developed which use predicate logic expressions to query a Prolog database. Colmerauer [1982] examines a number of interesting structures and their translation into logical formulas. These examples may be used as a starting point for illustrating the quantifier scoping rules proposed here. It should be noted that Colmerauer's examination of quantification is from a structural rather than functional perspective. The logical formulas consist of two major components :

- certain words function as three branched quantifiers (3BQs) which unlike classical quantifiers, for example '$\exists$' and '$\forall$', relate not one but two formulas to a variable. The exception to this is 'not' which signifies negation over its scope but does not introduce a variable.

- predicates which have arguments that are either variables or actual values.

Colmerauer introduces heuristic rules which are used to choose between (disambiguate) alternative paraphrases. Of course, counter examples against such a set of rules will always be available and in many cases it may be disputed which of the possible paraphrases of an ambiguous phrase is the 'correct' interpretation. Nevertheless, it has been shown that at least in applications such as database access, it is desirable to use such rules and that in most practical situations, a reasonable interpretation can be produced where a 'reasonable interpretation' is that which most human readers of the phrase would agree upon .

Colmerauer first examines the phrase *'no man has a trunk'*. This phrase exhibits the well known property of quantifier scope ambiguity. That is to say that the phrase has more than one meaning depending on the order of quantifier scoping given to the quantifiers *no* and *a*. The two different quantifier scopings (paraphrases) can most clearly be illustrated in tree form :



In the first case *(a)*, the quantifier *no* is said to have been given a higher scope than the quantifier *a* and in the second case *(b)*, *no* is given a lower scope than the quantifier *a*. Colmerauer states that the 'correct' paraphrase is the first given above *(a)*. There cannot, of course, be a truly correct paraphrase. Here 'correct' means that paraphrase which carries the meaning understood by most readers. In the case above the heuristic used to determine the correct paraphrase is :

> " The quantification introduced by the article of the subject of a verb dominates the quantification(s) introduced by the complement(s) closely related to that verb. "

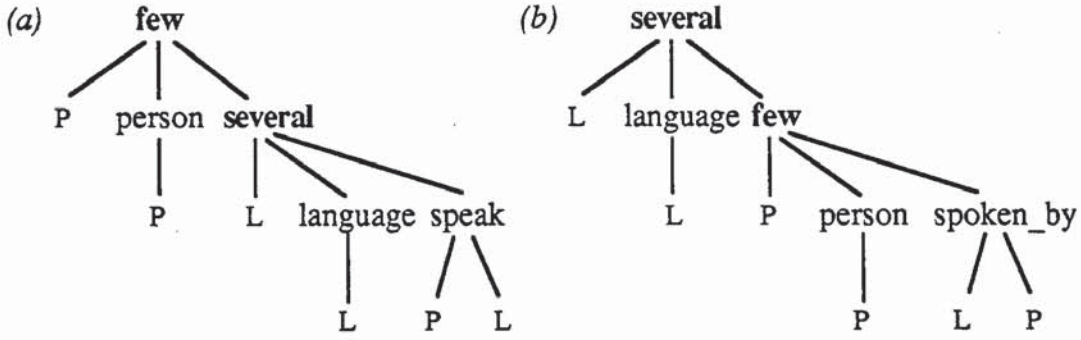
It is interesting to note that the heuristic itself is stated in functional (subject, complement) rather than in syntactic (noun phrase, verb phrase) terms. The F-structure for this phrase is identical in structure to that shown in Figure 3.3.1.a with only a different predicate, *has* not *owns* (treating *have* in the same way as Colmerauer). It can now clearly be seen that this rule can be used in a similar manner on the F-structure which represents the ambiguity ; the subj function's quantifier is given scope over that of the obj function's ('$F_1 > F_2$' means that function F_1 has scope over F_2) :

subj > obj

Colmerauer next examines the effect of active/passive change on quantification. The active/passive pair :

(a)	'few persons speak several languages'	(active)
(b)	'several languages are spoken by few persons'	(passive)

are given the quantification trees :



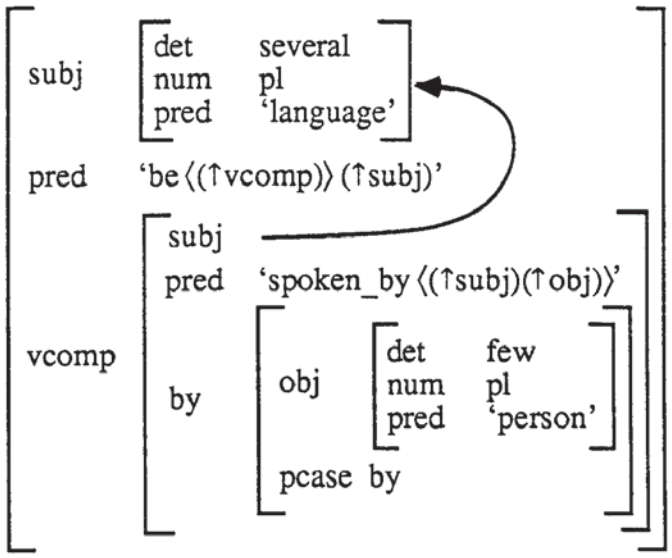
This Colmerauer states clearly shows that :

" change from active to passive inverses the hierarchy of quantification. "

The corresponding F-structures for these phrases are :

$$(a)\begin{bmatrix} subj & \begin{bmatrix} det & few \\ num & pl \\ pred & \text{'person'} \end{bmatrix} \\ pred & \text{'speak}\langle(\uparrow subj)(\uparrow obj)\rangle' \\ obj & \begin{bmatrix} det & several \\ num & pl \\ pred & \text{'language'} \end{bmatrix} \end{bmatrix}$$

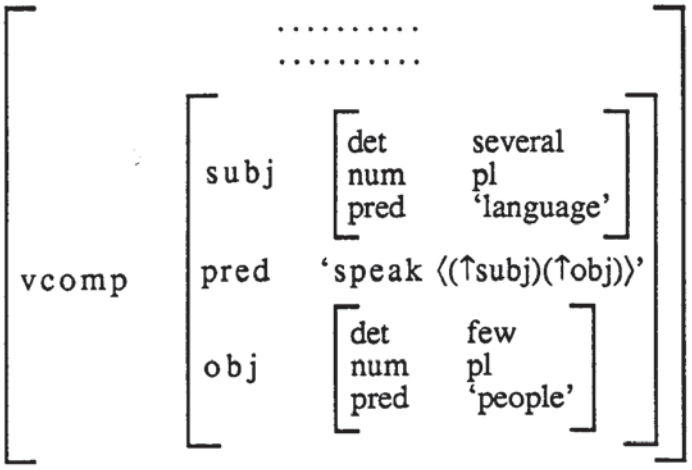
(b)



The first F-structure *(a)* conforms to the first of Colmerauer's heuristic rules (the subj function's quantification dominates those of complements) but the second F-structure does not conform to the active/passive heuristic. This is one case in which functional structure departs from a syntactic analysis. The main clause's subj function is passed to the verb-complement under functional control from the lexical entry of *be*.

The problem of interpreting F-structures involving *there* insertion was discussed earlier, where interpretation of only the (vcomp function) takes place. This position is also adopted here so that only vcomp is interpreted in the case of an auxiliary verb invoking functional control :

(a')



The active/passive rule does not however apply to the vcomp function. The quantifier hierarchy within 'vcomp' is defined by the first of Colmerauer's heuristic rules. The

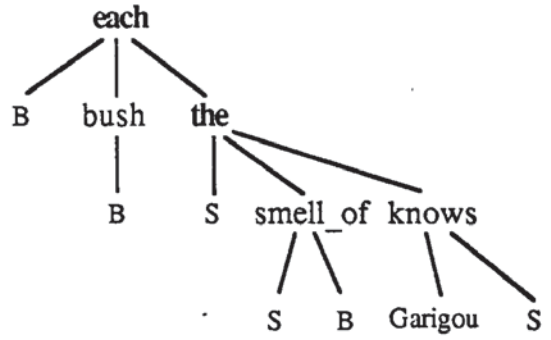
change from active to passive may indeed reverse the order of quantification but the reversal is reflected directly in F-structure as in *(a)* above.

Colmerauer next examines quantification in phrases with noun complements. The three phrases :

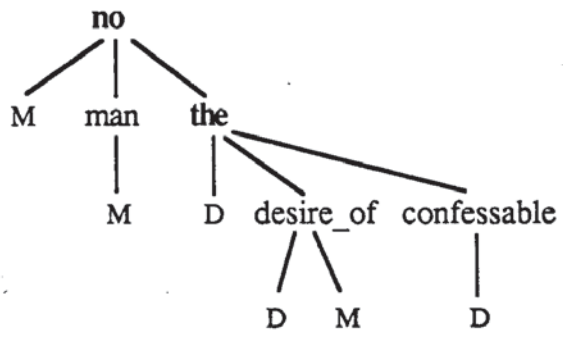
(a) 'Garigou knows the smell of each bush.'
(b) 'The desires of no man are confessable.'
(c) 'Pangloss recommends Candide to the captain of each ship.'

are given quantification trees :

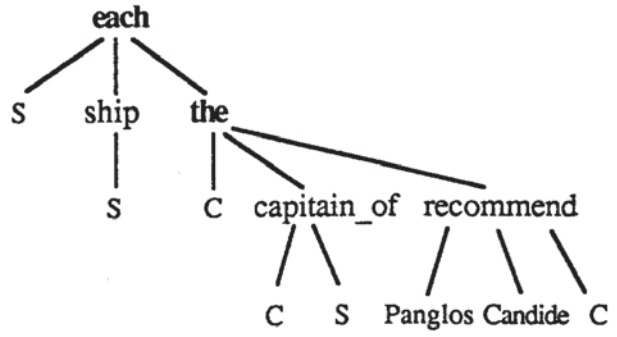
(a)



(b)

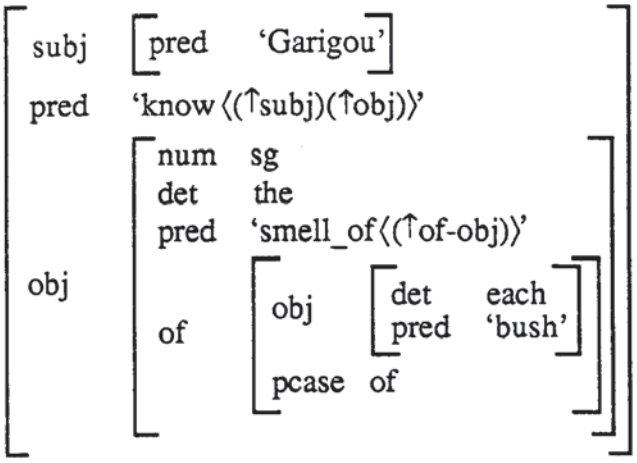


(c)

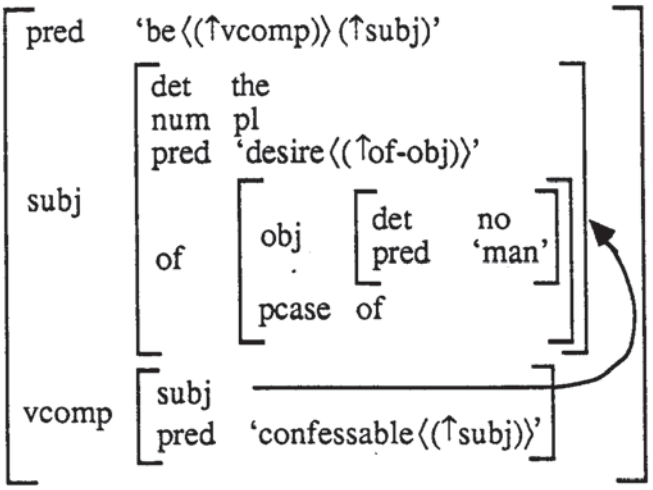


for which the corresponding F-structures will be :

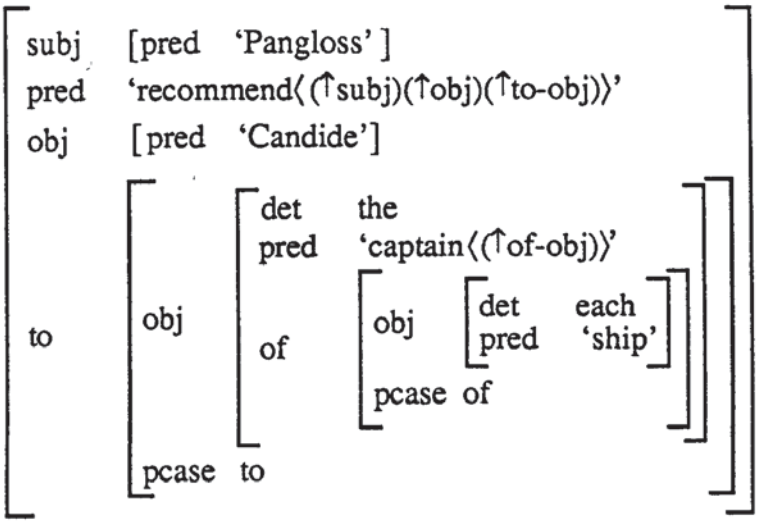
(a)



(b)



(c)



Colmerauer gives the following rule for noun complements :

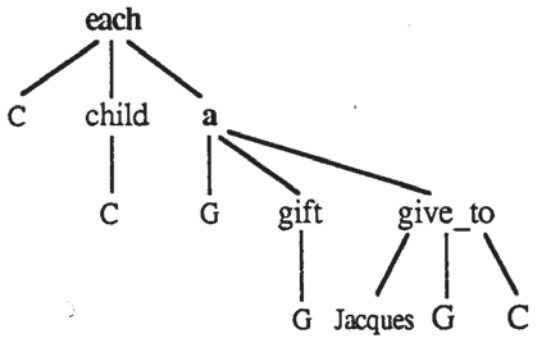
> " In a construction involving a noun and a complement of this noun, the quantification introduced by the article of the complement dominates the quantification introduced by the noun. "

Functionally, noun complements will form functions such as of-obj, by-obj, to-obj. The rule above is thus reflected by giving the sub-function (of-obj, by-obj, to-obj) domination in the quantifier hierarchy, over the function in which a noun occurs. So that in the case of some function *F*, with a noun's semantic form, subcategorizing some other function with a name of the form (pcase)-obj, the following scoping rule is applied :

(pcase)-obj > F

Next, Colmerauer looks at verbs having two or more complements, such as :

'Jacques gives a gift to each child.'



for which Colmerauer introduces the heuristic :

> " Whenever a verb, an adjective or a noun has two complements, the quantification is made in the inverse order of the natural order of their appearance. "

The F-structure for this phrase is :

$$\begin{bmatrix} \text{subj} & \begin{bmatrix} \text{pred} & \text{'Jacques'} \\ \text{num} & \text{sg} \end{bmatrix} \\ \text{pred} & \text{'give\_to}\langle(\uparrow\text{subj})(\uparrow\text{obj})(\uparrow\text{pcomp})\rangle\text{'} \\ \text{obj} & \begin{bmatrix} \text{det} & \text{a} \\ \text{pred} & \text{'gift'} \\ \text{num} & \text{sg} \end{bmatrix} \\ \text{pcomp} & \begin{bmatrix} \text{pcase} & \text{to} \\ \text{det} & \text{each} \\ \text{pred} & \text{'child'} \end{bmatrix} \end{bmatrix}$$

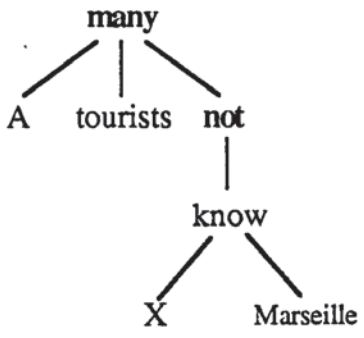
Functionally, the rule proposed by Colmerauer can be mirrored by giving functions such as xcomp, whatever the type of xcomp (pcomp, vcomp, acomp), dominance over the obj function :

xcomp > obj

Quantification involving negation is then considered in several sample phrases, for example :

'many tourists do not know Marseille.'

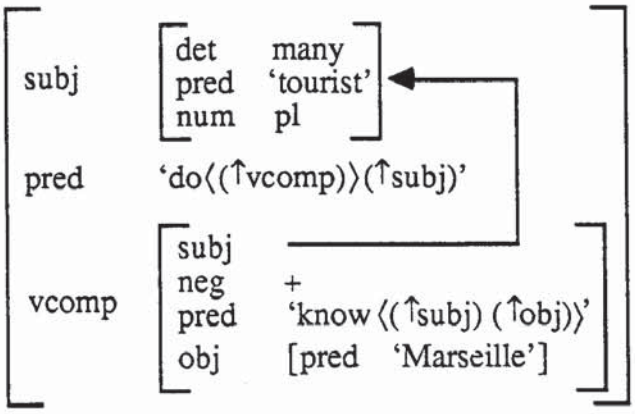
for which the quantification tree is given as :



The negative *not* is not a true quantifier as it does not introduce a variable. Instead it is represented as an operator which takes a single argument. Colmerauer proposes the following :

" The negation introduced by *do not* is translated by the operator *not* placed immediately below the quantification(s) introduced by the subject. Nevertheless if the article of the subject is *each, each of the, every* or *all the* the operator *not* applies to the whole statement. "

This rule differs from those given previously as it does not only relate to sentence structure but also to the actual quantifiers used. The F-structure for the sample phrase is :



The presence of the negative operator *not* is signalled in the F-structure here by the simple feature 'neg +'. This feature is thus an example of a simple semantic feature. The rule proposed by Colmerauer can be represented in an LFG type notation :

If (↑ neg) = +
Then If (↑ subj det) = each or
(↑ subj det) = every or
(↑ subj det) = 'all the' or
Then neg > (all other functions)
Else subj > neg > (other remaining functions) .

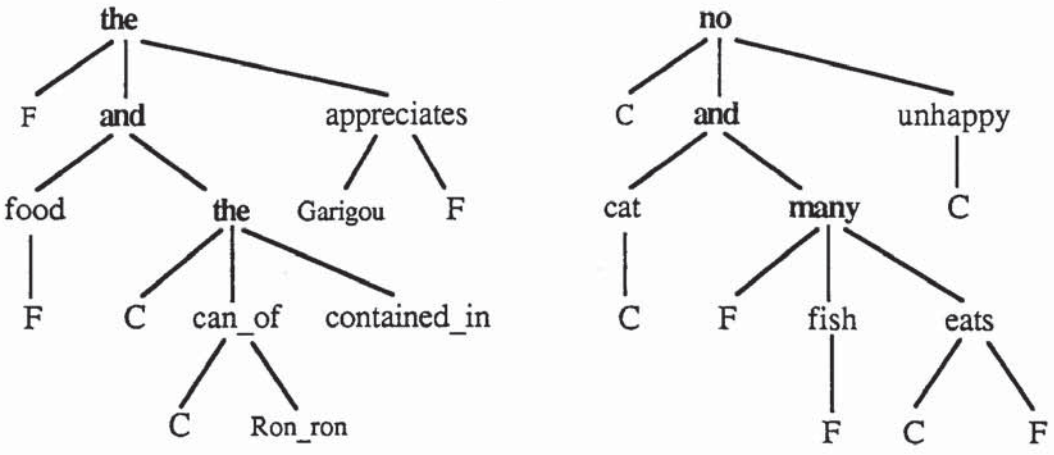
Next, Colmerauer considers restrictive relative clauses. The relative clause :

'Garigou appreciates the food that is contained in the can of Ron-ron.'

where *that* constitutes a marker of relativisation and another example where *which* acts as a relative marker :

'no cat which eats many fish is unhappy.'

These relatives are given the following quantification trees :



Relative clauses may be viewed as similar to conjunctions which is perhaps the reason for insertion of a conjunction by Colmerauer. To deal with relatives, Colmerauer proposes the heuristic :

> " Every relative clause is treated as an ordinary statement; the relative pronoun is replaced by the appropriate variable and the whole is linked to the translation of the noun by the conjunction *and.* "

An explanation of the LFG analysis of relatives will be postponed until Chapter 4, but briefly, a relative is analysed as consisting of two functions "head" and "mod" where the head function is moved under constituent control to play some functional role within mod. In outline the example relatives will be analysed :

$$\begin{bmatrix} \text{head} & [\text{'the food'}] \\ \text{mod} & [\text{'is contained in the can of Ron-ron'}] \end{bmatrix}$$

$$\begin{bmatrix} \text{head} & [\text{'no cat'}] \\ \text{mod} & [\text{'eats many fish'}] \end{bmatrix}$$

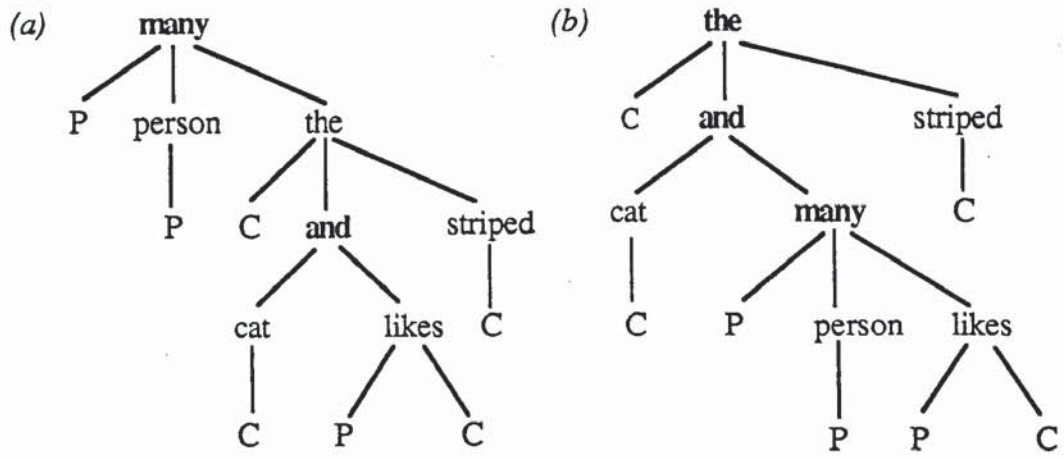
In the functional realm, Colmerauer's rule is replaced by a new rule giving the head function higher scope than that of the modifying function mod and creating a conjunction of the representations of head and mod :

head > mod

In Chat-80 it was noted that a distinction is made between post-modifiers which are 'predicate' and those which are 'modifiers' and that these have different scoping properties. Relative clauses are one type of post-modifier belonging to the predication, from which quantifiers cannot be moved. The quant tree for the phrase :

'the cat that many persons like is striped.'

is thus given the quantifier tree *(b)*, and not *(a)* below.



In a similar manner, it can be stated that quantifiers cannot be moved out from the mod function.

Using Colmerauer's, examples basic quantifier scoping has been illustrated based on the type of functional analysis that F-structure provides. As in the Chat-80 system the inherent properties of quantifiers will in exceptional cases cause the functional scoping rules to assign an incorrect scoping. Just as quant trees are reshaped in Chat-80, in these cases the F-structure can be reshaped to account for the strong determiners (*each* and *any*). The restructuring of F-structures is performed in much the same way as that of quant trees in Chat-80. Pereira examines quantification in the cases of the noun-phrases :

(a) 'the$_2$ population of each$_1$ country.'
(b) 'the$_2$ population of any$_1$ country.'
(c) 'the$_1$ population of every$_2$ country.'

where the the quantifier with subscript *1* is given a wider scope than that with subscript 2. In Chat-80's initial quant tree the quantifier *the* will, in all cases, have wider scope. The two examples *(a)* and *(b)* constitute exceptions to the default rules used in Chat-80. Rules

for exceptions, which may all be paraphrased by "if determiner A appears above determiner B in the Quant tree, give B, contrary to the default, wider scope than A", will be applied to cases *(a)* and *(b)* to give the determiners *each* and *any* wider scope than *the* in each case. When the quant nodes for these determiners are moved up the quant tree their post-modifiers, represented in the quant node, will be moved with them.

F-structures are reshaped in the same manner, by reordering a functional precedence list. This list names the functions in an F-structure and the order of named functions defines their relative scopes. For any list of functions, there is a default ordering of functions which may be modified by rules which can be paraphrased 'if function *A* appears in the list after function *B* and function *A* contains a strong quantifier and function *B* only weak quantifier(s) then place function *A* before function *B* in the list'. The F-structure for all three noun-phrases (forming a corresponding function F_{np} in F-structure) is in outline :

$$F_{np}\begin{bmatrix} \text{det} & \text{the} \\ \text{pred} & \text{'population}\langle(\uparrow\text{of obj})\rangle\text{'} \\ \text{of} & \begin{bmatrix} \text{obj} & \begin{bmatrix} \text{det} & \text{each} \\ \text{pred} & \text{'country'} \end{bmatrix} \\ \text{pcase} & \text{of} \end{bmatrix} \end{bmatrix}$$

For this F-structure, the function list will contain the function name of-obj and (implicitly) the function name F_{np}. The default ordering of this list being '[F_{np}, of-obj]'. The function name 'F_{np}' is taken as referring to the quantifier at the top-level within F_{np}. This order is then reversed in cases *(a)* and *(b)* as the function 'of-obj' contains a strong quantifier and F_{np}, a weak quantifier. Just as in Chat-80, strong quantifiers are repeatedly moved up the quant tree, then in this case the function F_{np}, which now has a strong quantifier in the first function of its functional list, may be placed before other functions in a functional list of the F-structure surrounding F_{np}.

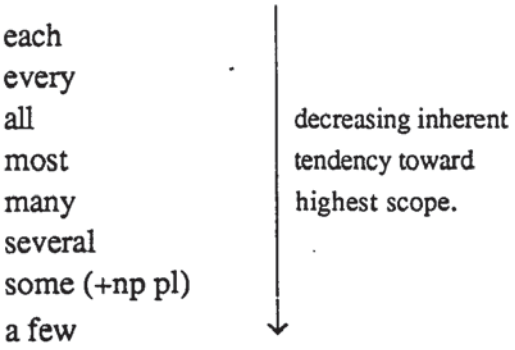
The quantification rules being proposed here are restricted to operate only on the content of F-structure (thus at the semantic stage, C-structure is redundant) and currently relate only to functional roles and the properties of quantifiers themselves. The rules proposed here are by no means 'complete' or even correct when applied to any case. They do however encompass all the scoping problems examined by Colmerauer and those

posed by the query corpus (Appendix D). It is thought that the principles upon which scoping is based here are generally applicable to most cases and the rules themselves could be developed, through greater specialization, to give a wider coverage.

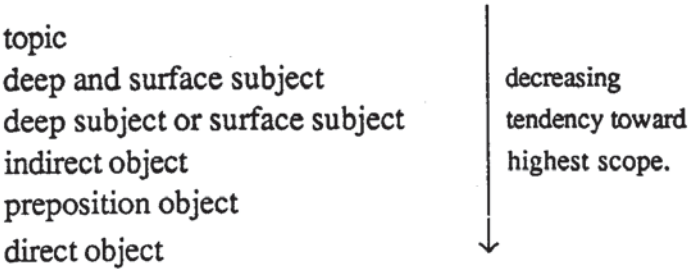
The theory of quantification proposed here is similar in nature to that suggested by Ioup [1975]. Ioup criticizes the traditional view of determining quantifier scope by syntactic structure and suggests, with supporting data from the several languages, that three factors can be seen to universally determine quantifier scoping. These factors are (in decreasing order of importance) :

- the "inherent properties" of individual quantifiers.
- the "grammatical function" of a quantifier within a clause.
- the location of a quantifier in a "salient serial position".

Ioup suggests the following hierarchy as reflecting the relative tendency of quantifiers to take highest scope based on their inherent properties :



and another hierarchy, relating the grammatical function of the *np* that a quantifier occurs in, to the tendency of that quantifier to take highest scope :



It is explained [Ioup, 1975, p57] that the subject and topic functions are at the top of this hierarchy because they occur in salient positions in clauses (clause initial and final positions respectively). Ioup explains :

> " Sentences are uttered as ordered series of elements, and, thus are subject to what is known as the serial position effect. It has long been known in psychology that in an ordered set of items those at the beginning and at the end capture the most attention and are retained the longest. This is due to what is called the primacy and recency effect. Languages use these positions to convey the most important information in the sentence. This is why the element with higher scope will most likely be found at either end of the sentence. English normally utilizes the S-initial position to convey important information. Thus, the noun phrase with highest scope can usually be found at the front of the sentence. This is why to date, linguists, who have built their theories around data in English alone have posited left-right order as the determining factor in assigning scope. One need only look at a variety of languages to see that the leftmost position is just a vehicle to convey the important grammatical information. "

It can be seen that F-structure may contain all of the three informational components Ioup has proposed as important criteria in determining quantifier scope. In the translation shown in Figure 3.3.1.c, the function q is taken as one example of a function representing constituents which have a salient position (clause initial) in Wh-interrogatives. In contrast to linear-order based theories then, it seems that the type of functionally based quantifier scoping theory, proposed here, may have some psychological basis. Certainly, Ioup's data suggests that a functionally based theory may apply universally to a number of languages, as LFG itself has been shown to do.

Chapter 4

LFG Analysis of English Queries and Constructs

The database to be used here is taken from the Chat-80 system which uses a Prolog database of geographical information. The Chat-80 system also provided a set of twenty-three test queries which are to be used as a corpus. These queries appear to have been selected as representative of many different linguistic phenomena, although as might be expected, only interrogatives are represented, no declaratives being present in the set. The queries to be analysed are given in Appendix D and the grammar and lexicons (domain and domain-free) used to parse these are listed in Appendix B.

The queries can be grouped together according to the different linguistic phenomena (relevant to LFG) that they display. It is under the headings of these phenomena that the LFG analysis of constructions is discussed below. In addition to the constructions found in the query corpus, several other constructions discussed by Pereira [1982] are outlined using LFG. An LFG analysis of simple noun phrases and comparatives has been described by Frey [1985].

4.1 Wh-Questions

In Wh-question type interrogatives, the initial *np* contains an interrogative determiner (*which*, *what*, *who*, *where*, *how many*). This *np* as discussed in Section 2.9 has been moved from some position in the phrase where it has some functional role.

LFG analyses a Wh-fronted question as a function q and a function focus, where q corresponds to the interrogative determiner (usually found in clause initial position) and focus to the entire *np* front. An outline of this analysis is shown in Figure 4.1.

This analysis raises the interrogative semantic form to a functional role in the outer F-structure independent of its degree of C-structure embedding in the front. It has been shown that q is used in the semantic interpretation of an F-structure. The focus function is intended to be co-indexed [Kaplan & Bresnan, 1982, p279, n23] with its other functional role, found through movement, for use by rules which interpret anaphors. Anaphora is outside the scope of this thesis but the implementation of LFG should support these rules.

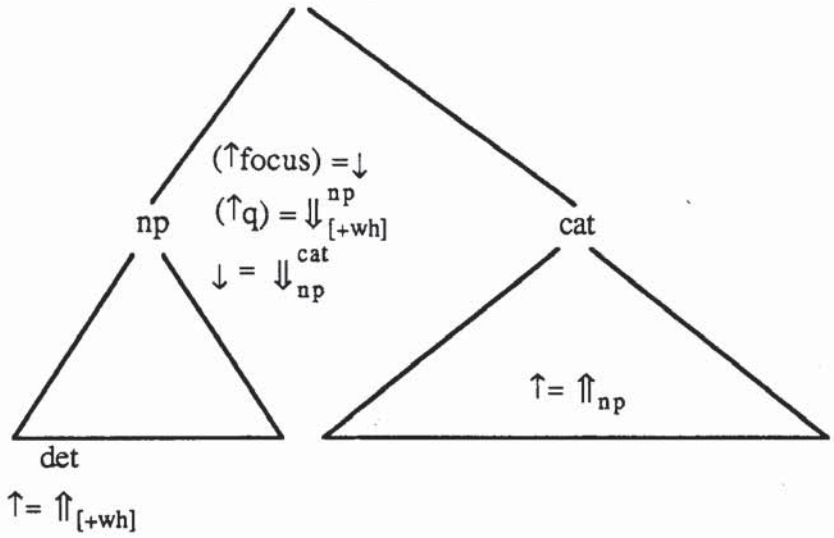


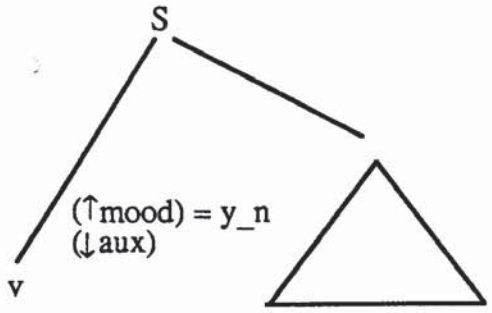
Figure 4.1 LFG Analysis of WH-Front in WH-Question

4.2 Yes / No Questions

Interrogative Yes/No (polar) questions are signalled by an auxiliary verb in clause initial position and also display subject/verb inversion :

'Does Afghanistan border China ?'
'Is China in Asia ?'
'Has China a population exceeding the population of Afghanistan ?'

This type of question will be indicated in F-structure by introducing an atomic feature for mood :



The mood feature is used here to explicitly mark polar interrogatives for semantic interpretation. The presence of the aux feature at the top-level of an F-structure could be used as an alternative to this but the mood feature is more explicit and has been suggested as such a marker (actually for imperatives) by Kaplan and Bresnan [1982, p219]. This

feature, like the function q, will have a semantic significance. This feature will always be at the top-level of an F-structure and will be translated as a 'quantifier' which will have the form : 'y_n(<predicates>)', where *<predicates>* is the first-order logic translation of the rest of the interrogative.

4.3 Instances of the Verb *Be*

Constructions involving *be* are difficult constructs to deal with in LFG. The verb *be* is an intensive verb taking a subject and single complement. Falk [1984] provides an analysis of the English auxiliary system in LFG. A single entry for *be* is proposed [Falk, 1984, p499] where *be* is categorized as a helping verb *hv* (a category which encompasses main verb, passive and progressive *be*) :

be : HV (↑ pred) = 'be⟨(↑ xcomp)⟩ (↑ subj)'
not (↑ vcomp inf)
not (↑ vcomp tense)

The negative existential equations prevent *be* taking tensed or infinitival complements. Also, a functional control equation, which Falk does not give here, would be expected to pass the subject of *be* to the complement. Falk also gives a lexical entry for *is* [Falk, 1984, p489] in combination with *there* (there-insertion) :

is : V (↑ pred) = 'there_be⟨(↑ xcomp)⟩ (↑ subj) (↑ obj)'
(↑ subj form) =$_c$ there
(↑ obj) = (↑ xcomp subj)
(↑ tense) .....

Where the corresponding entry for *there* is given simply as :

there : N (↑ form) = there

The form feature is described by Falk as a dummy element which replaces the normal semantic form in "meaningless forms". To prevent *there* being allowed with other constituents having the usual semantic form (pred), a negative existential equation '¬ (↑ pred)' can be added to Falk's entry for *there*. The nature of there-insertion can be illustrated by the example phrase (taken from [Jacobsen, 1978]) :

'there is a man sitting in the chair.'

which is derived from the sentence :

'a man is sitting in the chair.'

From this, it can be seen that in declarative cases *there* replaces the subject *'a man'* which is moved into the verb complement where it assumes the subject function. Subject and verb agreement is therefore transformed into subject and 'vcomp-subject' agreement.

Falk's proposal of a single lexical entry for *be* is most attractive, for it eliminates the lexical ambiguity which multiple entries would cause. He claims [Falk, 1984, p499] that :

> " English has one verb *be* ; it encompasses 'main verb', passive, and progressive *be*. No syntactic or semantic reason exists to distinguish them. "

This view appears to be in direct conflict with the traditional transformational view of copula *be*, where *be* is ascribed to have at least three different semantic interpretations [Jacobsen, 1986, p152]. These are illustrated by the phrases :

'There is Peter in the room.'	(existential *be* coupled with *there*)
'Peter is very tired.'	(attributive *be*)
'Peter is the first year tutor.'	(equative *be*)

In the first case above of existential *be*, the verb is used in conjunction with *there* and has a meaning which might be paraphrased *'there exists an X'* where *X* is the entity mentioned in the verb complement (here *Peter*). Attributive *be* assigns some quality (attribute) to the subject, so that in the second case above, *Peter* the subject is given the attribute *tired*. In the third case of *be*, equative or identificational *be*, the subject and complement are one and the same entity. The complement and subject might be interchanged without any change in meaning. So a speaker can say just *Peter* or *'the first year tutor'* without any change in reference, as both of these identify the same individual.

It is clear from the examples above that there is some semantic motivation for differentiating between the three uses of *be*. This view may not however conflict with that of Falk. The difference may be seen to be only in the interpretation of the term 'semantic'. Falk's analysis is in terms of LFG's semantic component (which is based only on semantic forms and their subcategorizations) whilst the TG view is based on a much deeper notion of the semantic component. Thus whilst the three cases of *be* may be analysed in LFG's semantic terms as all simply having a semantic form which takes a subj and vcomp function, at a deeper semantic level it becomes necessary to differentiate the occurrences of *be*. This difference obviously cannot be in the functional arguments of *be* but must be in how *be* is interpreted itself to relate its functional arguments to one

another. This has led to the adoption of a view that in essence satisfies both Falk's and the TG view of *be*.

The verb *be*, as Falk describes it, takes a subj function and a vcomp function, and will be given a lexical entry for each of the semantic interpretations proposed by TG :

(a)	(existential)	be : V	pred = 'exists⟨(↑ xcomp)⟩ (↑ subj)' (↑ subj form) ¬ (↑ pred)
(b)	(attributive)	be : V	pred = 'attributive⟨(↑ xcomp)⟩ (↑ subj)'
(c)	(equative)	be : V	pred = 'equate⟨(↑ xcomp)⟩ (↑ subj)'

The exact interpretation and form of these entries will be postponed until some examples are described, but in outline these are translated in the following manner :

(a) the subj function is omitted as the form feature is a semantically irrelevant version of a semantic form and only translation of the vcomp function is performed.

(b) the vcomp function is taken as describing some attribute of the subj function and the two functions are semantically related in this way (note that under functional control subj will appear inside of the vcomp function itself).

(c) the vcomp and subj functions are taken as referring to the same entity and the functions are semantically related in this way.

In a copula phrase or sentence, the auxiliary verb *be* functions as the main and only verb. The analysis of such constructions is notably omitted from Kaplan and Bresnan [1982] but is approached by Falk (English) where he notes [Falk, 1984, p500] that :

> " 'Main verb' *be* is a little more complicated [than *be* occurring in conjunction with a main verb]. In sentences with other auxiliaries, the analysis proceeds straightforwardly. However, in a sentence with tensed 'main verb' *be*, we are left with a verbless *vp*. "

This indicates that the function vcomp (verb complement) may be produced by a verbless portion of C-structure. Falk further suggests that the *vp* be given the trivial equation and that the sub-constituents of the *vp* can then be interpreted as arguments of modal tensed

be. It should also be noted that a verbless *vp* is used by Bresnan *et al* in Dutch analyses [Bresnan *et al*, 1982]. The problem here is however, further complicated by changes due to interrogative formation and the need to produce a functional structure suitable for semantic translation. Consider the following cases of (declarative) main verb *be* :

(a) 'Peter is the senior tutor.'	(equative *be*)
(b) 'The largest country is in Africa.'	(attributive *be*)
(c) 'Where is the largest country ?'	(attributive *be*)

In *(a)*, Peter clearly functions as the subject (subj function) subcategorized by *is* and the verb is an equative occurrence of *be* (*Peter* is the same entity as *'the senior tutor'*). The problem now remains to determine the functional role of the constituent *'the senior tutor'*. If verbless *vp* constructions are allowed, then this constituent can form a verb-phrase (*vp*) and be assigned a functional role of vcomp, subcategorized by *is* (Figure 4.3.a.).

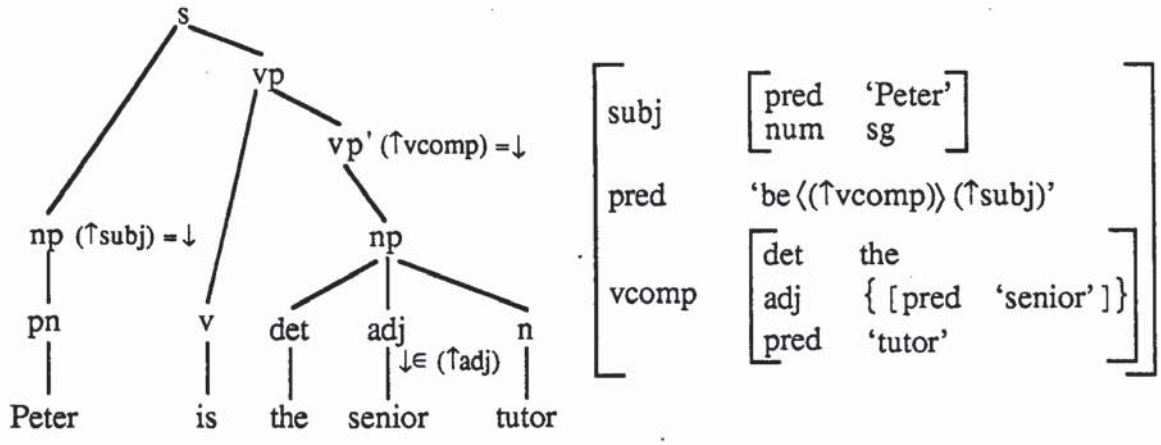


Figure 4.3.a Example of Equative *Be* C-structure and F-structure

The translation of an F-structure containing equative *be* involves an equating of functional variables and their types at the top-level of the F-structure. The sem feature of equative *be* is :

$$\text{sem} = \text{be}(\text{subj} = \text{vcomp})$$

which signifies that subj and vcomp are related (as the same entity) by having the same var feature value and slot type. In this example, *Peter* and *'the senior tutor'* might have the same domain type *human*. In certain instances however, types may not be identical but only compatible, so for example, *Peter* might have type *human* and the vcomp function a specialization of this *tutor*. The DAG series in Figure 4.3.b illustrates the translation of the F-structure produced from this example of equative *be*.

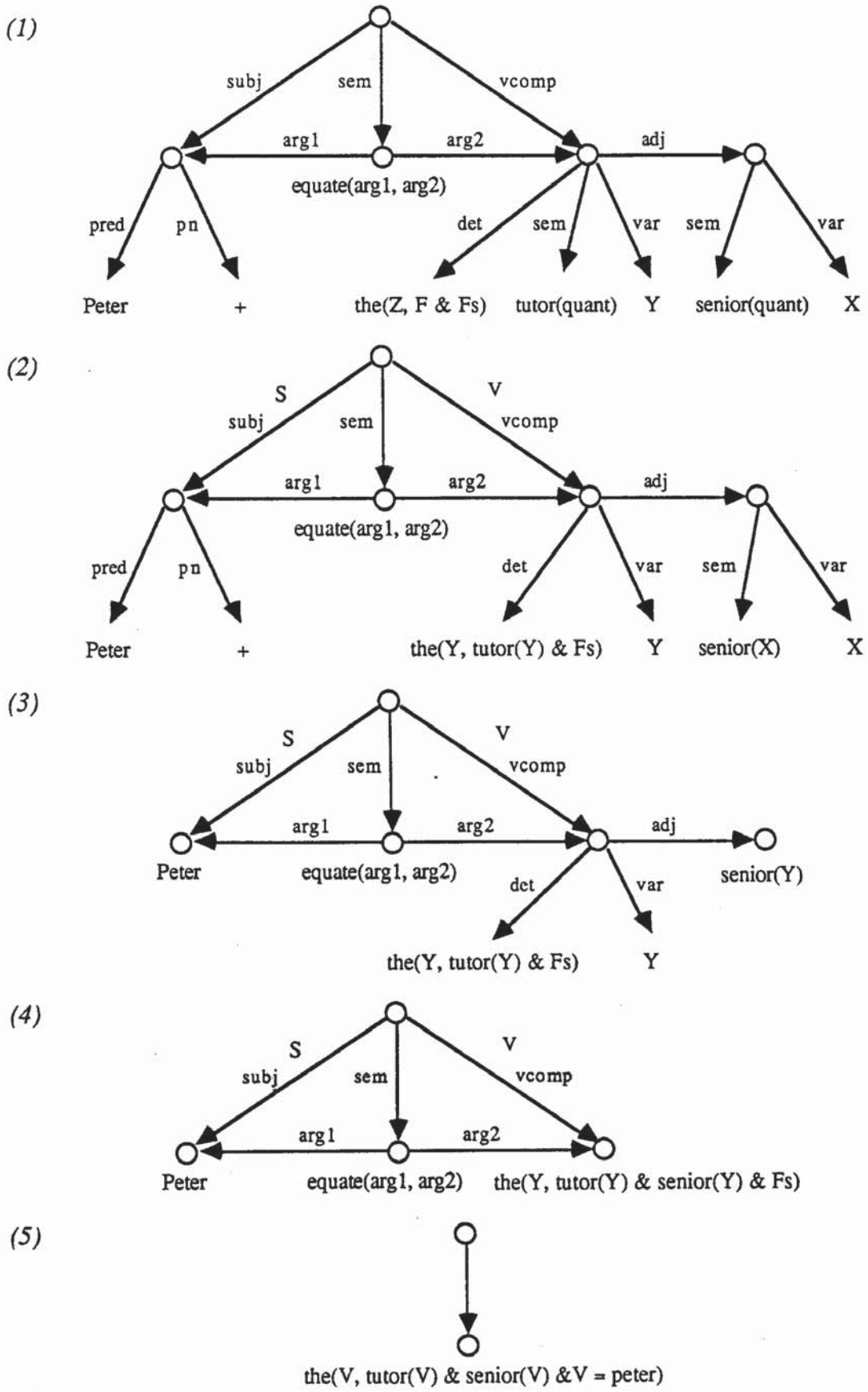


Figure 4.3.b DAGs Illustrating a Translation of Equative *Be*

In case *(b)* (attributive *be*), the subj function is in clause initial position (*'the largest country'*), in which case the prepositional phrase *'in Africa'* will form a verbless *vp* (Figure 4.3.c).

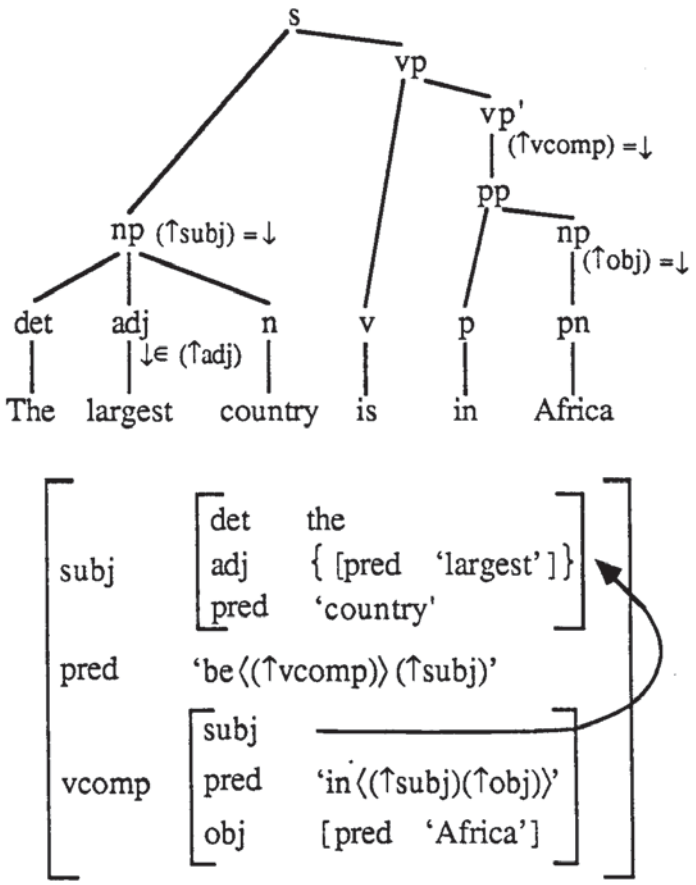


Figure 4.3.c Example Attributive *Be* C-structure and F-structure

Attributive *be* is translated in much the same way as equative *be* except that the subj and vcomp functions are known to be related as an entity and attribute of that entity. This relationship is however directly reflected in F-structure itself, so that type checking and functional variables are dealt with automatically. This is due to functional control. In cases of attributive *be*, the translation thus simply passes to the vcomp function. The sem feature of attributive *be* itself signals the need to only translate vcomp :

sem = passto(vcomp)

The last case *(c)* (interrogative attributive) causes the greatest problems for this analysis. The declarative phrase underlying this phrase would be *'the largest country is in <X>'*, where *<X>* is the questioned element. In this case, *'the largest country'* functions

as subj and the moved Wh-front functions as vcomp. Under functional control the vcomp function receives the subj function of the main verb and must thus subcategorize this. The semantic form in vcomp for the Wh-determiner *where* must therefore subcategorize subj. This does not seem so unreasonable if it is noted that *where* is semantically equivalent to *'in what place'* and can thus be represented by a logical expression :

what(P, in_place(P, subj))

Which is exactly how *where* is to be semantically translated here. This type of interrogative determiner will be called "attributive determiners", as they seem to be confined to asking about entity attributes (also *'how large'*, *when*). The use of attributive determiners will thus be coupled with the use of attributive *be*.

The use of attributive determiners such as *'how large'* requires that the subj function be passed to the focus function, to be subcategorized by *large*. In the case of the attributive determiners *where* and *when*, the subj function must be passed to q, as well as focus and vcomp, to satisfy the completeness condition of LFG (Figure 4.3.d). This is because the attributive aspect of place is included in the WH-word which forms q, whereas in the case of *'how large'*, the attributive aspect of size is separate from the WH-word *how*. The subj function can be passed to focus (and q) by equations attached to grammar rules. An example of an attributive determine (*where*) is show in Figure 4.3.d.

The additional complexity in the analysis of phrases with *be* is due here to the extension of semantics for producing first-order logic expressions. It may be the case that in LFG, in its pure form, determiners such as *where* would only be represented by a simple feature *'det where'* (although this would mean *be* could not pass its subj function to the vcomp). Here however, it is necessary to use a deeper representation of such determiners, suitable for translation into first-order logic, which involves allowing these determiners to have arguments. This might more easily have been accomplished by introducing an additional translation of LFG F-structure into a representation with additional arguments. However, given that the most unlikely part of the analysis shown here is the subcategorization by *where* and that *be*, it is generally agreed, does induce functional control, this peculiarity seems due to the analysis proposed by LFG and not due to the extensions made here.

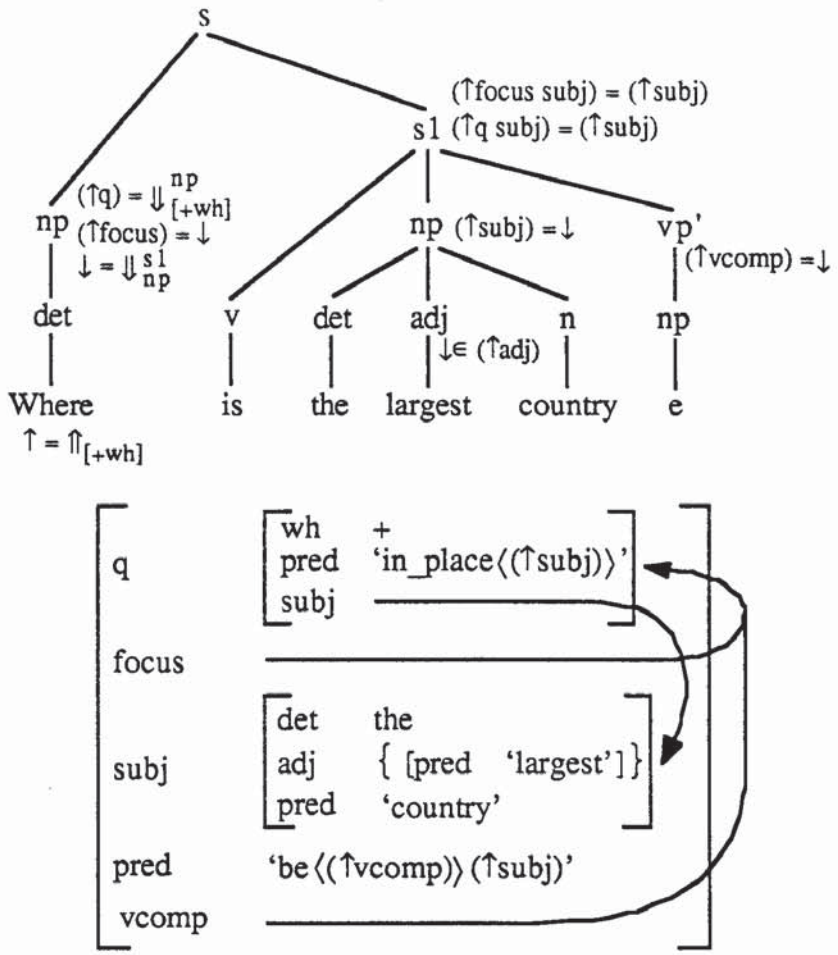


Figure 4.3.d Example Interrogative with Attributive *Be*
C-structure and F-structure

4.4 Instances of the Verb *Have*

The auxiliary *have*, like *be*, causes some problems in analysis. An analysis of *be*, when used as main verb, has already been proposed and the main verb *have* like *be* can be seen to relate its functional arguments. The analysis of *have* in LFG, of course, cannot directly use a paraphrase, as used in Chat-80. Instead a functional analysis must be provided. Consider the following examples (which fit the Chat-80 templates in Section 3.2.3) of each occurrence of *have* :

(a) 'England has a population of 60 million.'
(b) 'England has Scotland as a neighbour.'
(c) 'England has two borders.'

The verb *have* like *be* will take a vcomp function and allow a subj function to occur as well (not subcategorized). In Chat-80 special filling rules act after parsing to fit phrases

with main verb *have* to a paraphrase template. The first case of *have* above *(a)* matches the Chat-80 template :

<entity> has <attribute> of <value>

The C-structure for this phrase *(a)* is shown in Figure 4.4.a.

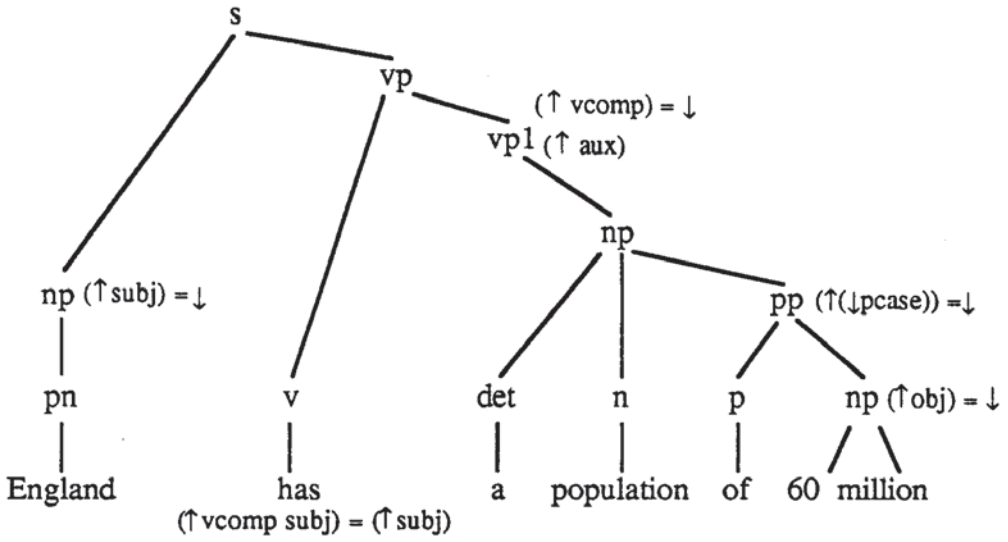


Figure 4.4.a C-structure for *Have* in 'Entity has Attribute of Value' Construct

The vcomp function from the corresponding F-structure, which is to be semantically interpreted, is shown in Figure 4.4.b. The constituent *England* functions as subj to be passed under functional control to the verbless *vp 'a population of 60 million'*. This *vp* serves as vcomp and has an internal head noun *population* with determiner *a* and a complement *'60 million'* which serves as a case marked 'of obj' function.

$$\text{vcomp}\left[\begin{array}{ll} \text{det} & \text{a} \\ \text{pred} & \text{'has\_population}\langle(\uparrow\text{subj})(\uparrow\text{of obj})\rangle\text{'} \\ \text{subj} & \left[\begin{array}{ll} \text{pn} & + \\ \text{pred} & \text{'England'} \end{array}\right] \\ \text{of} & \left[\begin{array}{ll} \text{obj} & \left[\begin{array}{ll} \text{measure} & \text{million} \\ \text{number} & 60 \end{array}\right] \\ \text{pcase} & \text{of} \end{array}\right] \end{array}\right]$$

Figure 4.4.b Vcomp F-structure for *Have* in 'Entity has Attribute of Value' Construct

The semantic form used here for *population* differs from that normally used for such nouns. Here *population* takes both a subj and obj function and thus subcategorizes as if it were a verb. This will only be the case where the main verb is *have* so that this use of the word must be constrained. To do this, constraint equations :

$$(\uparrow \text{main\_verb})_{c}\ \text{have}$$
$$(\uparrow \text{main\_verb})$$

can be added to this lexical entry for population and the value of 'main_verb' satisfied by a value originating from the lexical entry for *have.*

The translation of the vcomp function's F-structure will use the form of sem features (not shown) which are placed in this F-structure from lexical entries. The sem feature given to *population* is :

$$\text{sem} = \text{population(subj, quant)}$$

where during semantic translation, the subj argument will be substituted by the functional variable representing the subj function and the quant argument by a quantifier variable (var feature).

The F-structure of the phrase is produced in the normal manner and then, as in the case of *be* used as main verb, translation uses only the vcomp function. The method of translation can be illustrated using Pereira's [1982] paraphrase template to which this phrase would be fitted :

<attribute> of <entity> is <value> .

The use of *be* in this template is the equative case, in which case the quantifier variable of *<attribute>* (var feature) is equated with that of *<value>*. In the F-structure this means equating the function 'of obj' with the var feature of the vcomp.

The translation process is illustrated in DAG form in Figure 4.4.c [1]. The variables which are equated can be seen in DAG *(2)*. The measure and number features are further examples of simple features which have a semantic interpretation. The variables to be equated are those which do not occur in both sem feature and semantic form, so that, signalled by the *'main_verb = have'* value, these can be examined and the correct

---

[1] The cyclic DAG for 'of obj' has been drawn as a normal DAG for clarity. This is valid as the value of the feature pcase in 'of obj' is a simple symbolic value, used only to label the obj function for case marking.

variables equated. The translation information is thus all contained locally in the vcomp function.

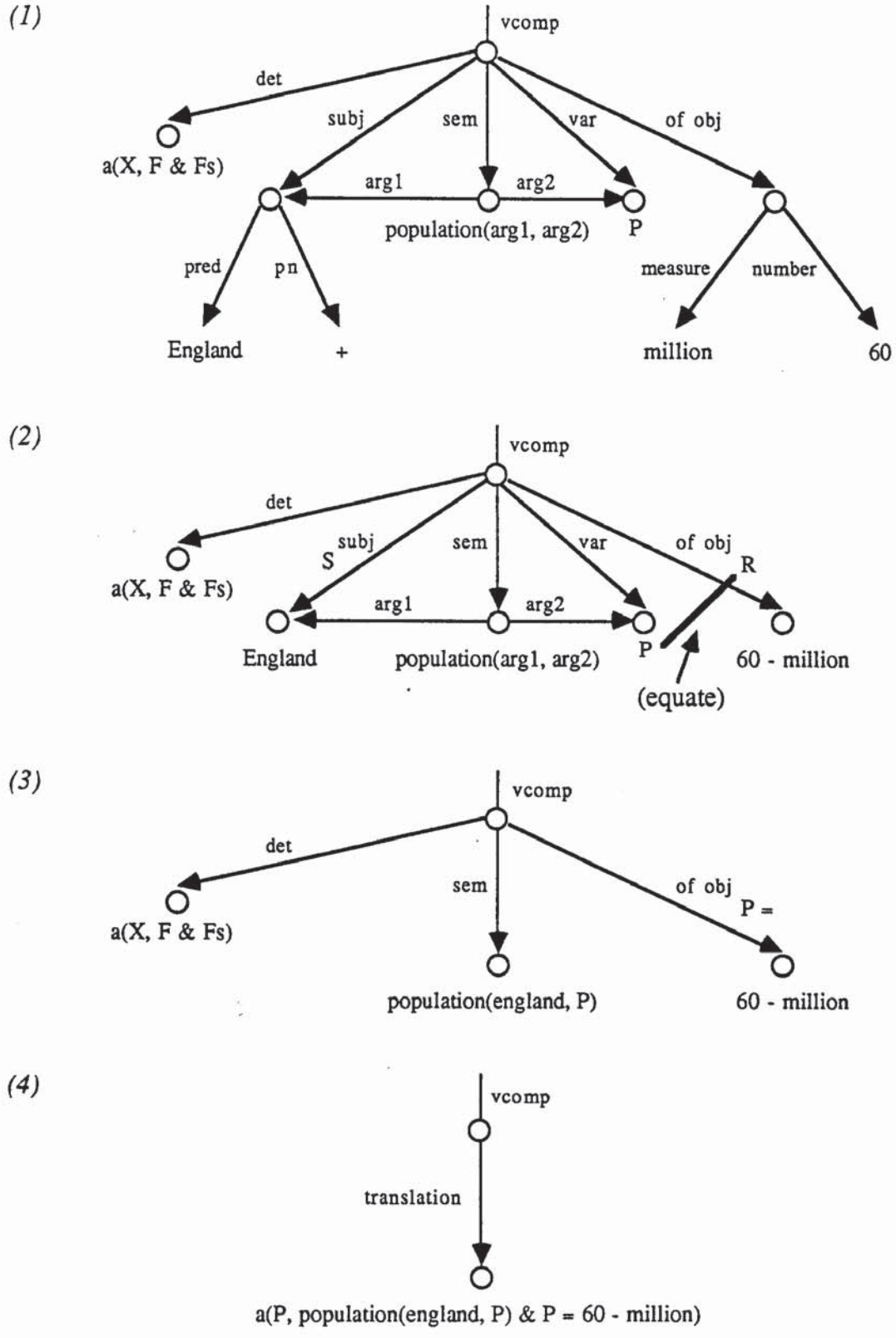


Figure 4.4.c Example Translation of *Have* as Main Verb

The second case of main verb *have (b)* can be dealt with in exactly the same way as the first if the preposition *as* in conjunction with *have* is used to produce the same functional analysis (Figure 4.4.d).

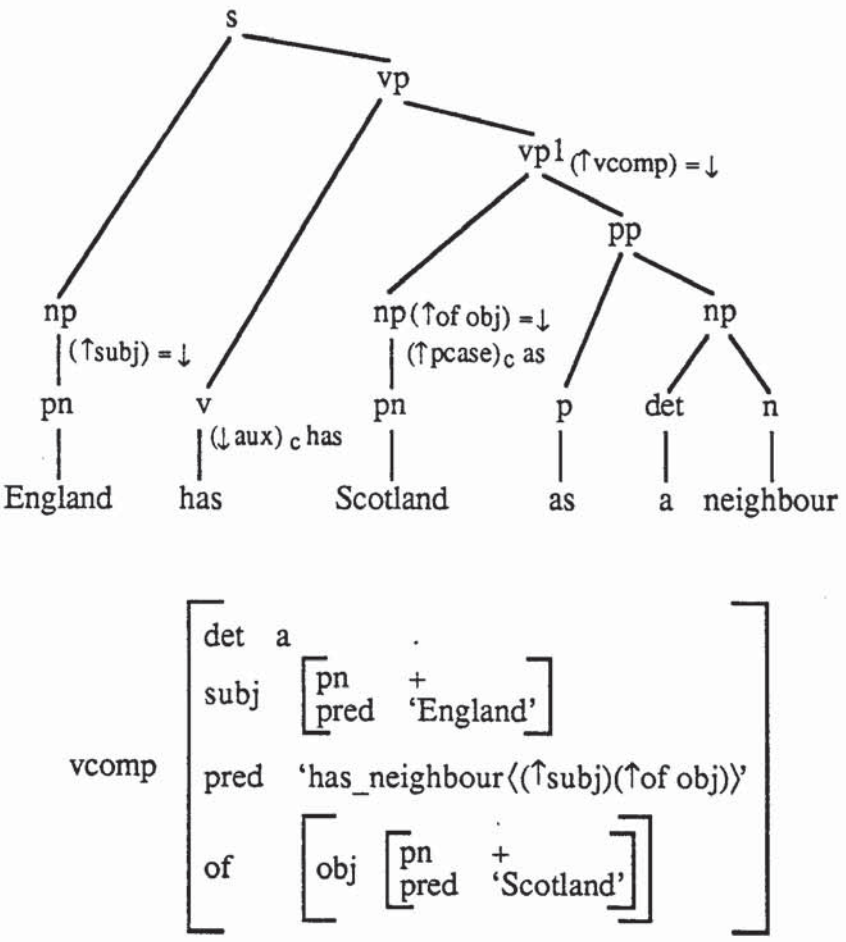


Figure 4.4.d C-structure and Vcomp F-structure for *Have* in 'Entity has Value as Attribute' Construct

The third case of main verb *have (c)* can also be dealt with in a similar manner to the previous cases, but no equating of variables takes place as the semantic form subcategorizes only a subj function (Figure 4.4.e).

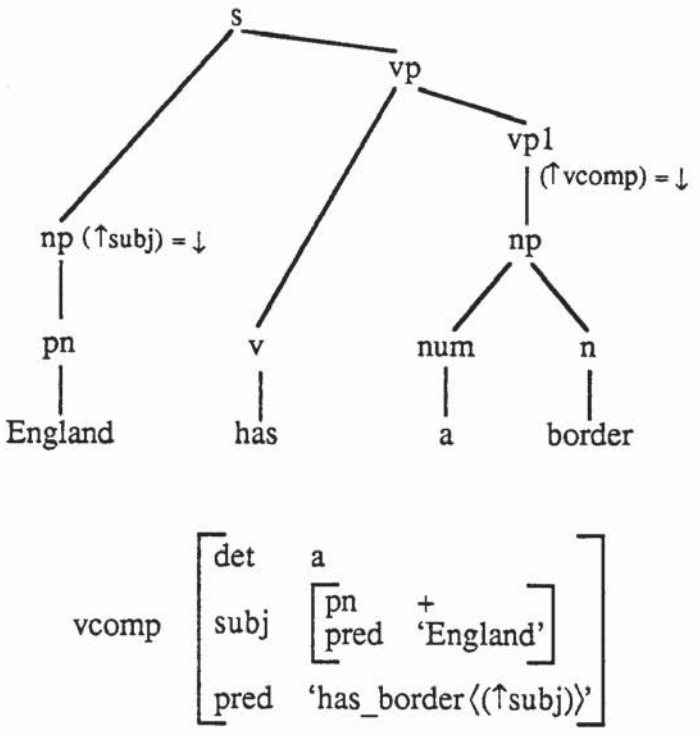


Figure 4.4.e C-structure and Vcomp F-structure for *Have* in 'Entity has Attribute' Construct

When used as an auxiliary, with other main verbs, the analysis proceeds as normal except that the verb has a genitive case marking influence on some function in F-structure. A typical interrogative instance of this is illustrated in Figure 4.4.f.

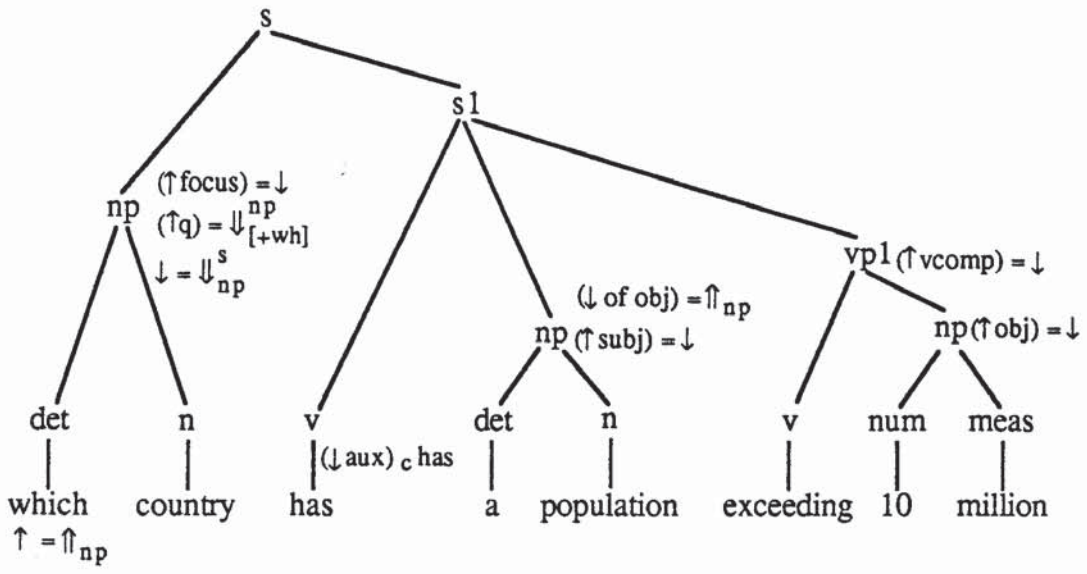


Figure 4.4.f Auxiliary *Have* Used with Other Verb

This analysis corresponds roughly to the analysis of *have* used in Chat-80 (Section 3.2.3). The LFG analysis passes the subject function in Fig 4.4.f to the vcomp function under functional control. The underlying declarative form of this interrogative is *'the population of country <X> exceeds 10 million'*. Where *<X>* is the questioned element represented as usual by the function q in F-structure. Comparing this with the Chat-80 paraphrase template the entities referred to by *'which countries'* are related to the attribute *population* by allowing population to subcategorize over the entity represented as a genitive object. The *<attribute>* in this case is related to the *<value>* by the second verb *exceeding*, in contrast to the examples with *have* as main verb where the value and attribute are related by equating variables.

4.5 Relative Clauses

Relative clauses may be roughly divided into restrictive and non-restrictive. Here, only those which can clearly be seen as restrictive are described. In a restrictive relative clause, as the term implies, the relative clause modifying a noun phrase head restricts the set of entities that the head refers to [Burton-Roberts, 1986]. For example, the relative :

'a baby that the senator kissed.'

refers first to some baby and then restricts this to specifically the baby kissed by the senator. A relative clause forms a *np* and has the general form shown in Figure 4.5.a.

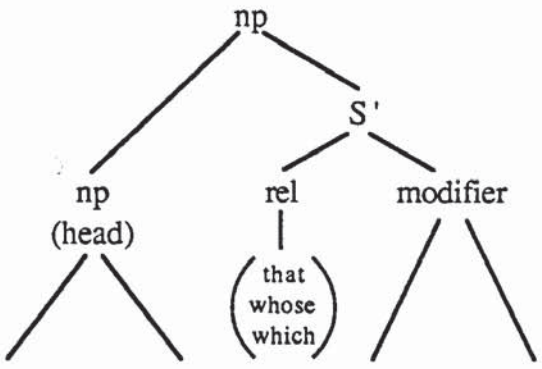


Figure 4.5.a Outline of Relative Clause Structure

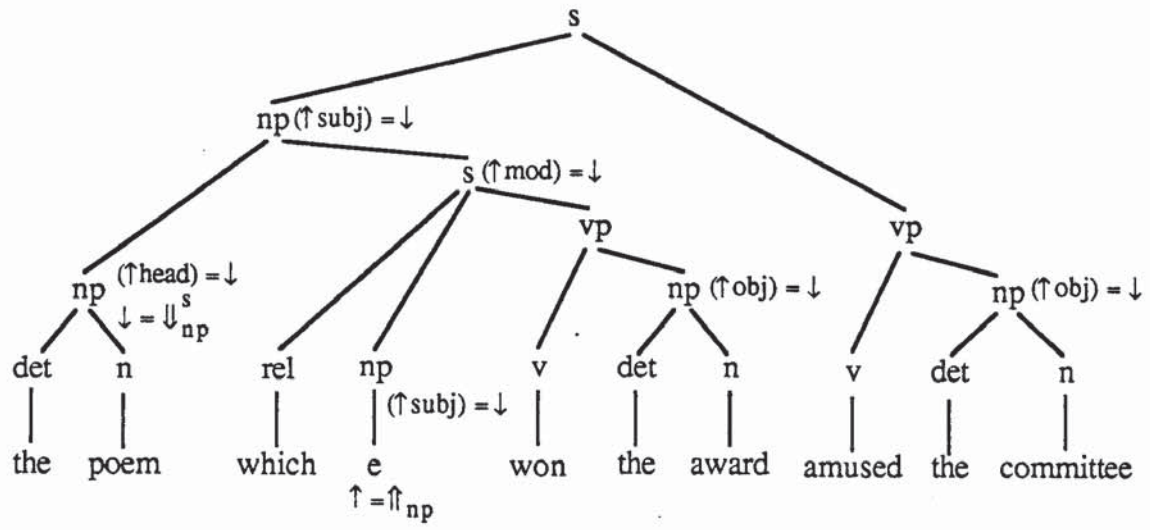
The head *np* plays some functional role within the modifier. A relative phrase such as :

'the poem which won the award amused the committee.'

where the head *'the poem'* also functions as subj within the modifier. The LFG analysis of relatives must move the head back into the modifier. Several methods are available in LFG for accomplishing this (functional control, constituent control or anaphoric control). Here an analysis, using constituent control, described by Pinker [1982, p663] is used. This analysis is based on the rule :

$$\begin{array}{lcll} \text{np} & \longrightarrow & \text{np} & \text{s}' \\ & & \downarrow = \Downarrow^{s1}_{np} & (\uparrow \text{mod}) = \downarrow \\ & & (\uparrow \text{head}) = \downarrow & \end{array}$$

and a rule which rewrites a *np* to a gap *e* with a controllee (with subscript *np*). The example relative above is thus given a C-structure tree :



The corresponding F-structure being :

$$\begin{bmatrix} \text{subj} & \begin{bmatrix} \text{head} & \begin{bmatrix} \text{det} & \text{the} \\ \text{pred} & \text{'poem'} \end{bmatrix} \\ \text{mod} & \begin{bmatrix} \text{rel} & + \\ \text{subj} & \\ \text{pred} & \text{'won}\langle(\uparrow\text{subj})(\uparrow\text{obj})\rangle\text{'} \\ \text{obj} & \begin{bmatrix} \text{det} & \text{the} \\ \text{pred} & \text{'award'} \end{bmatrix} \end{bmatrix} \end{bmatrix} \\ \text{pred} & \text{'amused}\langle(\uparrow\text{subj})(\uparrow\text{obj})\rangle\text{'} \\ \text{obj} & \begin{bmatrix} \text{det} & \text{the} \\ \text{pred} & \text{'commitee'} \end{bmatrix} \end{bmatrix}$$

This F-structure is translated as illustrated by the series of DAGs shown in Figure 4.5.b. Translation is quite straightforward, the only complication is the translation of head and mod functions within the subj function. By convention the head is given quantifier scope over mod and is translated in its entirety. Once this is done, the index of head (which is co-indexed with the mod's subj function) is recorded with its functional variable as completely translated. When the mod function is translated the subj function will not be translated but is given the functional variable recorded with its index.

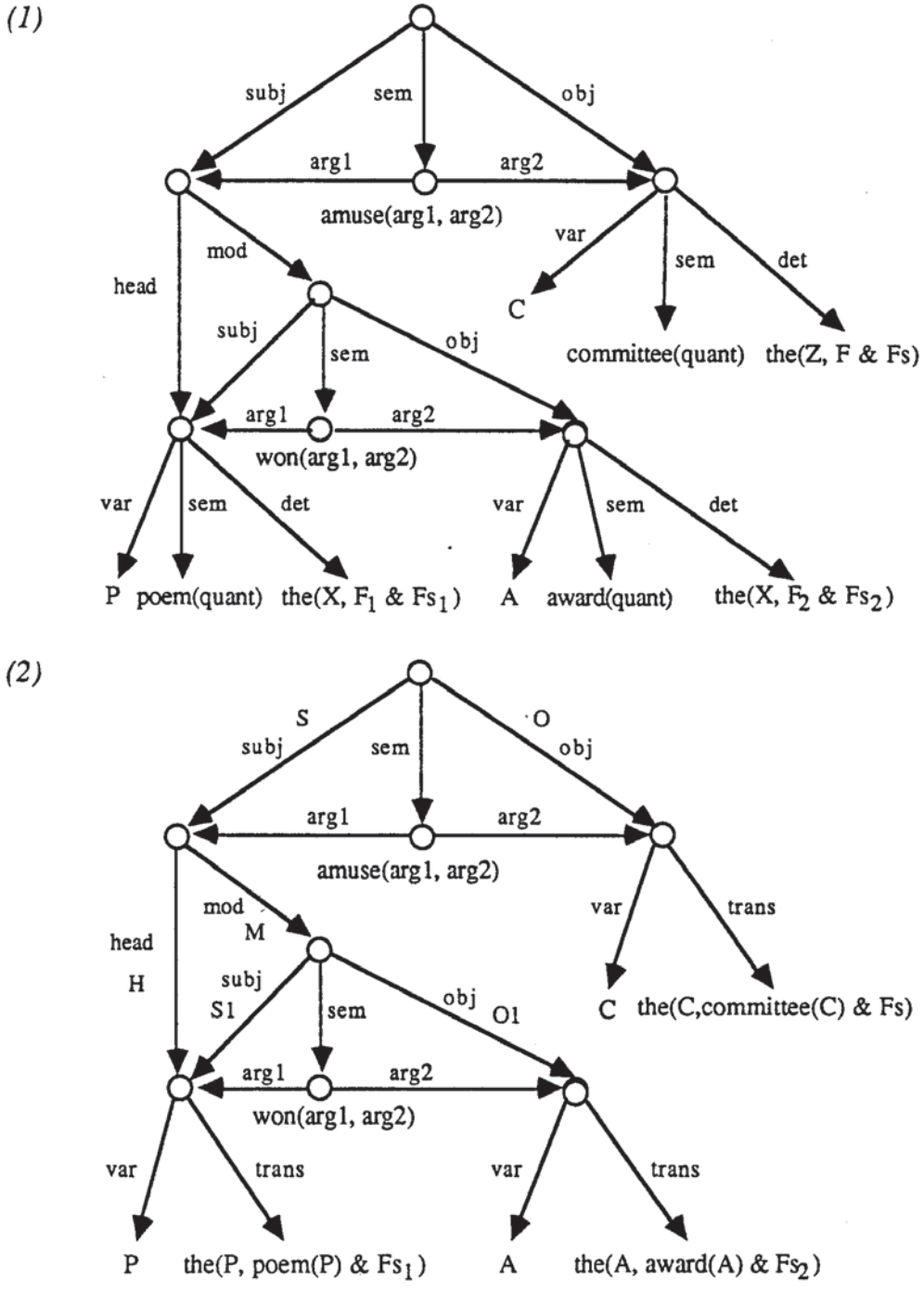
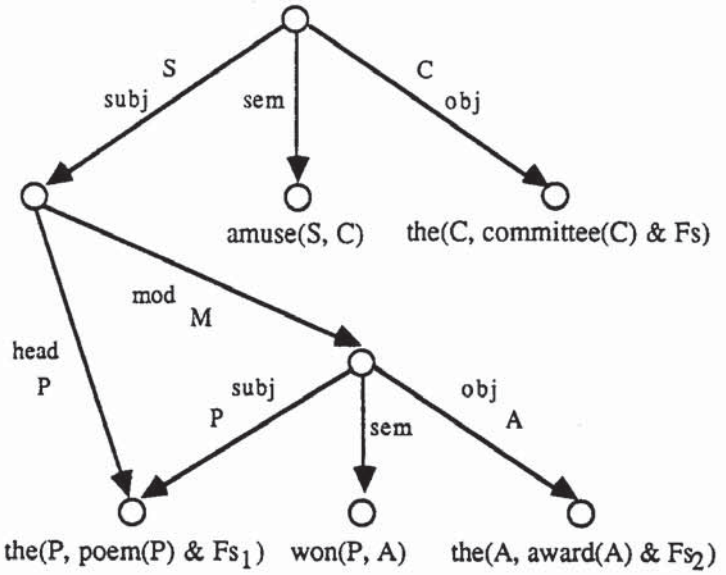
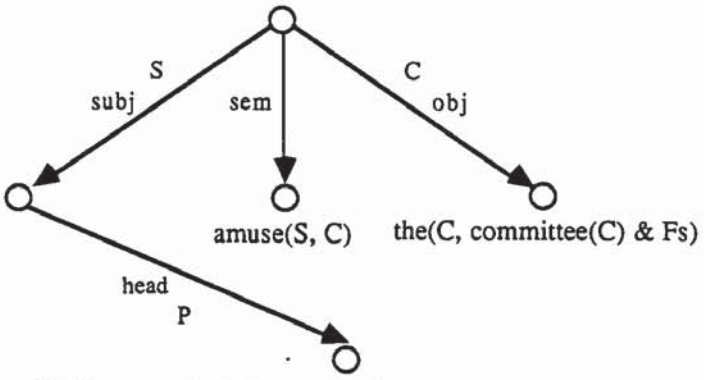


Figure 4.5.b Example Translation of Relative Clause (continued overleaf)

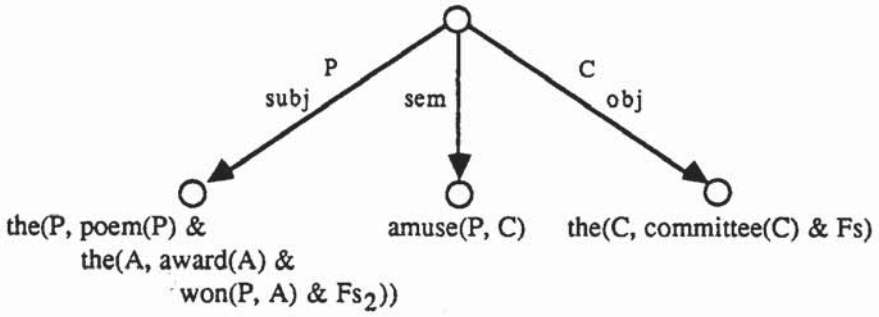
(3)



(4)



(5)



(6)

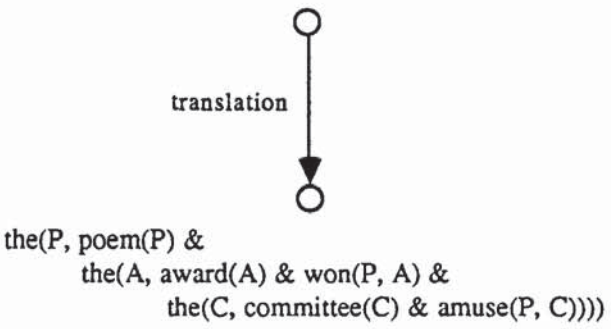


Figure 4.5.b Example Translation of Relative Clause (Continued)

This analysis varies from that proposed by Kaplan and Bresnan [1982, p280, n30] where functional or anaphoric control is used :

> " The preposed item in relative clauses is also Topic. Although the relative Topic might be functionally controlled when the clause is embedded next to the *np* that it modifies, it must be linked anaphorically when the relative is extraposed. "

The problem with this analysis is that anaphoric linking requires determination of the proper antecedent of a relative pronoun. This is itself still a matter for research but it should be noted that the method of identifying the moved constituent with its surface realization is independent of the method of translation described here. The analysis used here has proved sufficient for many cases and the relatives in the query corpus.

The genitive relative marker *whose* causes a particular problem in LFG. Consider the *np* :

'the country whose population exceeds 10 million.'

The underlying structure of which is :

'the country & population of the country exceeds 10 million.'

So in the case of relative *whose* not only is the head function moved into the modifier but it is also given a genitive case. This is reflected in C-structure by giving the moved head a possessive functional role inside mod (Figure 4.5.c).

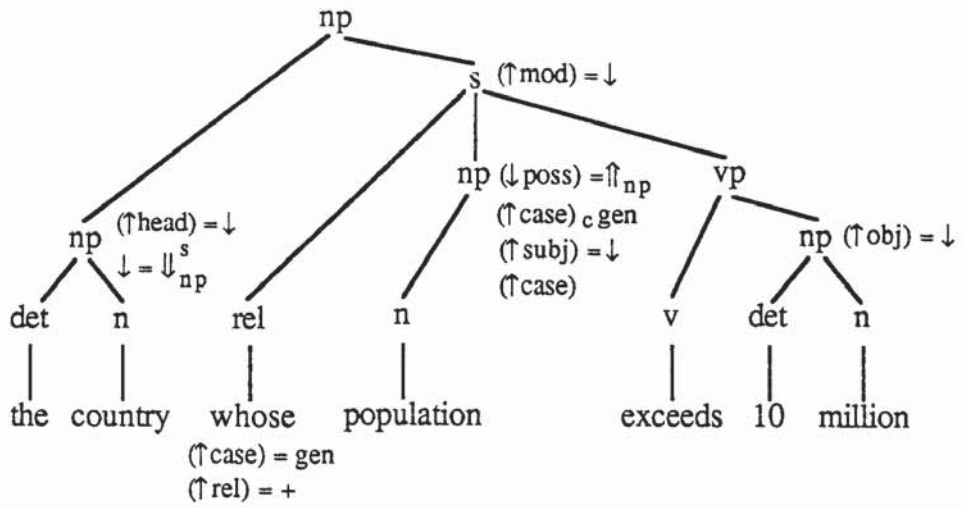


Figure 4.5.c C-structure of Relative with *Whose*

The poss function is subcategorized by *population* :

$$\text{pred} = \text{'population}\langle(\uparrow \text{poss})\rangle\text{'}$$

This function is equivalent to the case marked obj function 'of obj' more usually subcategorized by *population*. Lexical ambiguity could be reduced by substituting the equation '$(\uparrow \text{of obj}) = \Uparrow_{np}$' for the equation '$(\uparrow \text{poss}) = \Uparrow_{np}$' in Figure 4.5.c.

4.6 Reduced-Relative Clauses

Certain noun post-modifiers are analysed as "reduced relatives" in Chat-80. A reduced relative here simply means a relative without a relative marker (*that, which, whose*) where a reduced relative clause can take one of three forms :

- an optional negation and an adjective phrase as a sentence with main verb *be.*
- a participle phrase as a sentence.
- a sentence with an initial subject noun-phrase.

A example of a participle type reduced relative is *'the countries bordering France'* which in an unreduced form would be *'the countries that border France'*.

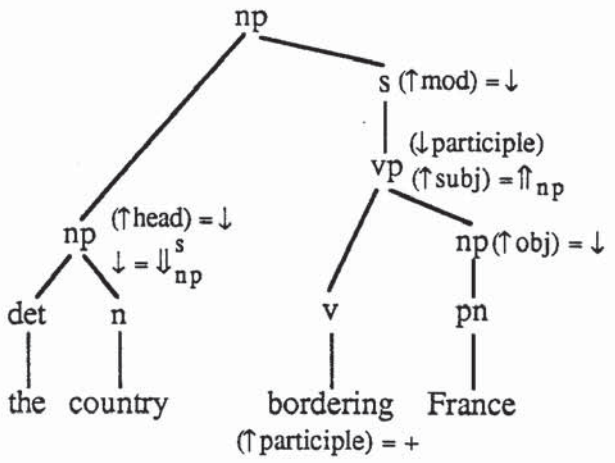


Figure 4.6 Reduced Relative C-structure

The reduced relatives can be analysed in the same way as unreduced relatives, as head and mod functions (Figure 4.6).

4.7 Prepositional Variations

In certain constructs prepositions can be left behind when movement takes place :

(a) 'The house which Peter lives *in* is old.'
(b) 'The house *in* which Peter lives is old.'

The underlying structure of both the relative clauses is *'Peter lives in the house'*. The noun phrase *'the house'* can be viewed as moved (actually deleted as explained in Section 4.5) from the relative modifier. In case *(a)* above, the preposition *in* has been left behind by the *np* (stranded) but in case *(b)*, the preposition has been moved with the *np*.

The first case presents no problems for analysis as the relative head function is simply moved back under constituent control to a position after the preposition and can combine with this to form a prepositional phrase in C-structure and an 'in obj' function in F-structure. The second case *(b)* however requires special attention. This type of prepostional phrase construct is analysed here as a case marked noun-phrase, as this alows movement to be described in the same manner as the more common occurrences of the noun-phrase movement. The C-structure for this phrase is outlined in Figure 4.7.a.

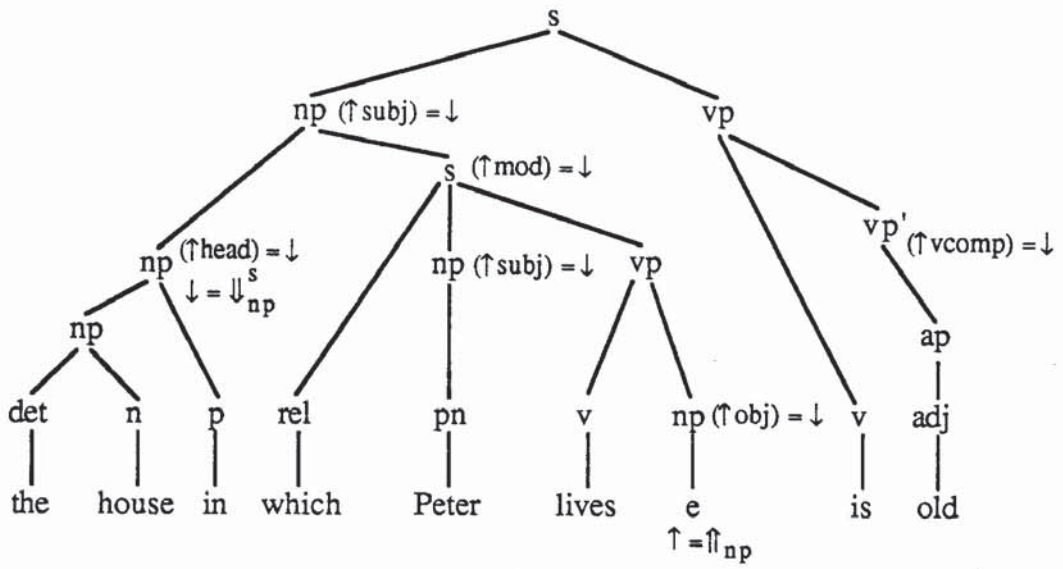


Figure 4.7.a C-structure with Moved Preposition

The verb *lives* in this case would normally have a semantic form subcategorizing subj and 'in obj' functions. In this case however, *'the house in'* cannot form an 'in obj' function as this is not subcategorized. Also, there is no motivation for moving the preposition *in* and *np 'the house'* separately so that they could form such a function when filling the gap *e*. The simplest solution is to introduce a new lexical entry for the verb *lives* :

pred = 'lives⟨(↑ subj)(↑ obj)⟩'
(↑ obj pcase)
(↑ obj pcase) $_c$ in

The additional lexical ambiguity this causes could also be removed if all case marked subcategorizations are replaced by simple obj functions and separate constraints on case.

In some cases of Wh-fronting, a preposition may also may be moved with the front. The declarative phrase :

'you gave the books to Peter.'

can have two interrogative forms :

(a) 'who did you give the book to ?'
(b) 'to whom did you give the book ?'

This case of fronting with a preposition, which is a particular instance of "pied-piping" *(b)* and a peculiarity of English which allows a preposition to be optionally moved with a *np*. The fronted preposition is analysed in the same way as the stranded preposition above. The verb *give* is thus given a semantic form :

pred = 'give⟨(↑ subj)(↑ obj)(↑ obj2)⟩'
(↑ obj2 pcase)
(↑ obj2 pcase) $_c$ to

Pereira [1982, p167] examines a more complex case of pied-piping. The phrase, *'the concepts in terms of which the theory was formulated'* is analysed as a *np 'the concepts'*, a preposition *in*, a Wh-complement *'terms of which'* and a sentence *'the theory was formulated'* into which the preposition and complement are moved. In LFG, the *np* can be first moved into the prepositional phrase which is then moved into the sentence as the head function of the relative (Figure 4.7.b). The *np* moved within the head function is here given a non-subcategorizable functional name 'pied'. This function, co-indexed with its other functional role within head is given scope over the rest of the head and is translated in its entirety. Its corresponding realization inside of the head is then not translated but is given the same functional variable as the pied function.

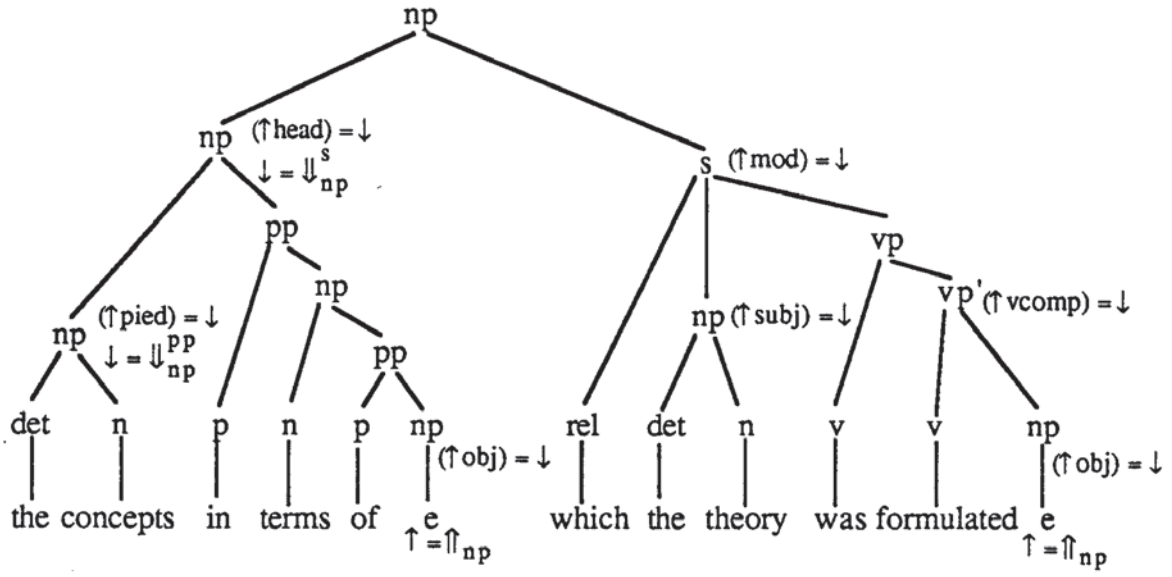


Figure 4.7.b C-structure with Pied-Piping

4.8 Adjuncts and Attachment

A standard syntactic analysis of phrases that include adjunct sets poses the problem of modifier (free complement) attachment [Pereira, 1982, p104]. In the Chat-80 system the syntactic derivation trees of modifiers are attached to the derivation tree of the main clause. In many cases there are several different points at which a modifier may be attached to the main tree. This is the problem termed "modifier attachment ambiguity". In Chat-80 Right-Most Normal Form (RMNF) was employed to determine the point at which to attach post-modifier derivation trees and to overcome the problem of ambiguity. The RMNF technique always attaches the modifier to the rightmost legal node at which a modifier may be attached. The derivation tree is thus made as tall as possible. The RMNF constraint employed in Chat-80 is a global condition involving the comparison of all possible analyses. However, it is not possible to produce a set of DCGs which directly represent a global constraint. Instead RMNF is implemented as a set of local conditions described by extra arguments and tests.

The noun phrase *'Peter painted the house at the end of the road under the tree'* contains two adjuncts which are represented in F-structure as a set (see Section 2.8). The F-structure of a single set member *'under the tree'* might be :

$$\begin{bmatrix} \text{pred} & \text{'under}\langle(\uparrow\text{subj})(\uparrow\text{obj})\rangle\text{'} \\ \text{obj} & \begin{bmatrix} \text{det} & \text{the} \\ \text{pred} & \text{'tree'} \end{bmatrix} \end{bmatrix}$$

This subsidiary F-structure is coherent but not complete. The governable function subj is missing from the adjunct member's F-structure. The problem of attachment in a functional context is thus one of "functional attachment". That is, determining which function in the enclosing F-structure also serves as the function that is missing in the adjunct F-structure. Ambiguity is present when a number of functions are candidates for filling this role.

As an F-structure is defined as well-formed with incomplete set member F-structures, the attachment constraints can be defined globally and applied after F-structure derivation. The RMNF method of attachment can be adopted by selecting the function derived last from the surface order of a phrase. This means, for example, that obj is preferred over subj for fulfilling a functional role in an adjunct function. Attachment is

realized in F-structure translation in the same manner as co-indexing; functions being given the same functional variable.

Some uses of the preposition *with* are dealt with in Chat-80 in the same way as the verb *have*. In addition this preposition may, as other prepositions, be used in an adjunct construction. Consider the interrogative :

'Which countries *with a population exceeding 10 million* border the Atlantic ?'

If the preposition is treated in the same way as *have* then, as in Figure 4.4.e, the Wh-front will be passed to the subsidiary F-structure containing *population* where it will form a case marked obj function 'of obj'. The adjunct is itself analysed as a reduced relative clause and is, in this case, both complete and coherent in contrast to the adjuncts described previously which are only coherent, but not complete.

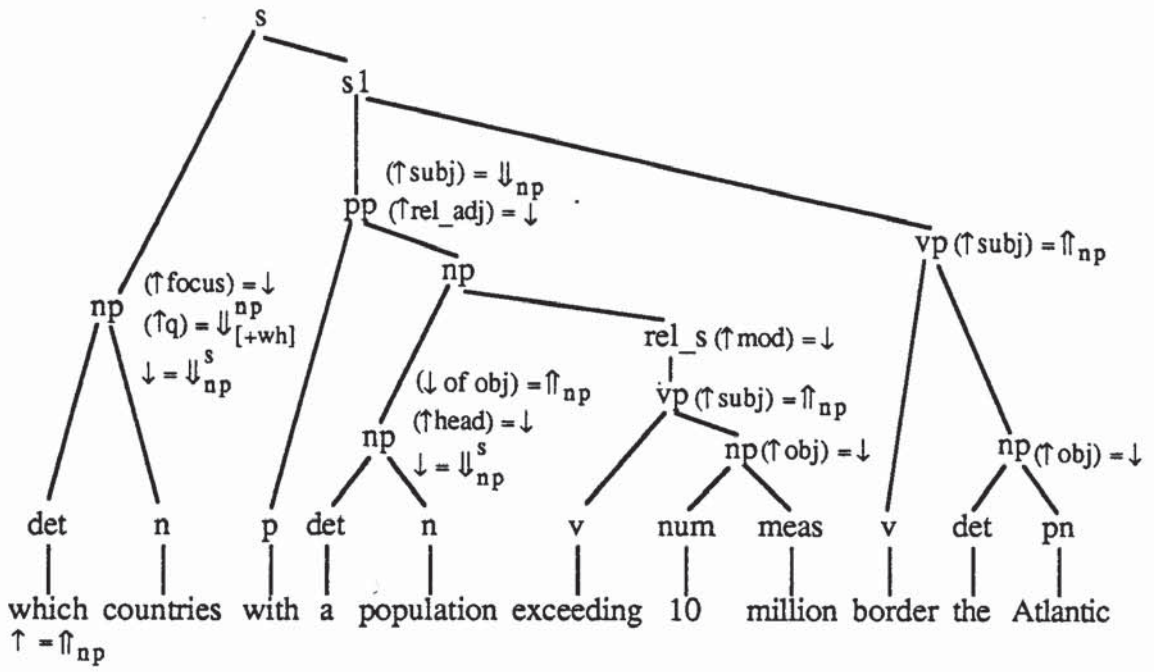


Figure 4.8 C-structure of Adjunct with *Have*

This particular construct is analysed here as a particular case of a "closed adjunct" and in addition, the preposition *with* is analysed as a particular instance of *have* (Figure 4.8). The adjunct forms a function 'rel_adj' (relative adjunct) which is not, like other adjuncts, realized as a set value but as an ordinary non-subcategorizable function which is semantically translated in the same way as a subcategorized function. Currently, this

function is by default given quantification scope over other subcategorized functions, although this may not be appropriate in other cases.

4.9 Coordinate Conjunctions

Coordinate phrases are difficult structures to handle and are a current issue in syntactic analysis. Here, only a small subset of possible coordinations, those represented in the query corpus, will be of concern. Coordinations may be generally described as phrases involving one or more connectives. Connectives include conjunctions (*and*, *or* and *but*), relative pronouns (*who*, *whom*, *which*, *where*, *when*, *how* and *that*), some examples of which have already been described, and binders (*because*, *so*, *since*, *until*, *before* and *while*). Only coordinations involving conjunctions will be discussed; specifically the coordinate *and*.

The coordinate *and* conjoins two phrase structures (coordinations), itself separating the two structures. Any category may be coordinated but the coordinations must be of the same category and basic structure if the coordinate phrase as a whole is to be well-formed (Figure 4.9.a).

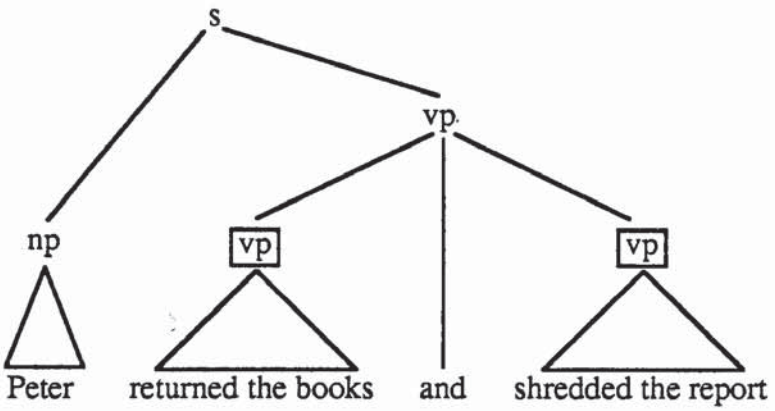


Figure 4.9.a Outline of Conjunction C-structure

It has been shown [Karttunen, 1984] that, if negative constraints are allowable (as they are in LFG), agreement in coordinate structures can be handled by simple unification of features from each coordination. This requires breaking, for example, the person feature into a number of component features. Unification of these new feature sets (values and negative constraints) then reflects the result of 'person coordination' :

1st person coordinated with 2nd person	⟶	1st person
1st person coordinated with 3rd person	⟶	1st person
2nd person coordinated with 3rd person	⟶	2nd person.

An analysis of coordination in LFG is described in Falk [1983]. Individual conjunctions are treated as bounding nodes in C-structure as shown above. However in the case of interrogatives, the bounding nodes may block the correct functional analysis.

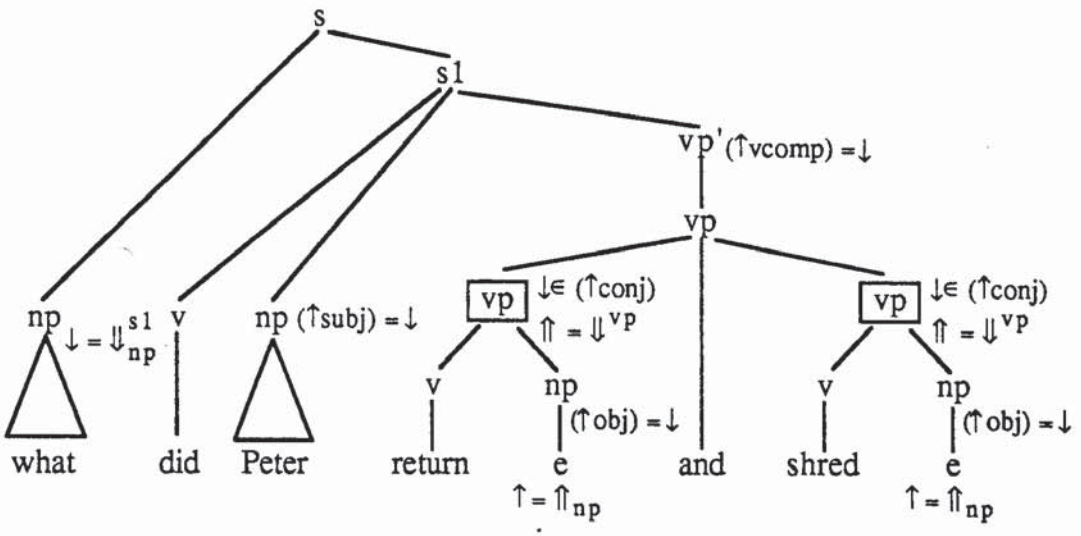


Figure 4.9.b Outline C-structure of Interrogative with Conjunction

In these cases Falk uses the linking equation to by-pass the bounding nodes (Figure 4.9.b). As can be seen in Figure 4.9.b, the controller originating from the Wh-front is matched, via the linking equations, with a controllee in both conjunctions. This is thus a case in which there is not a strict one-to-one correspondence between controllers and controllees. The controller passing into a conjunction must thus be duplicated and passed into subsequent conjunctions. Individual conjunct F-structures are placed in a set function, here called conj. The individual members of the conj set pose a particular problem for analysis.

The F-structure of the vcomp function is shown in Figure 4.9.c below. Each member of the conj set F-structure, it can be seen, is related to the F-structure in which it is contained in such a way as to make that F-structure well-formed. That is to say that unifying any conjunct set member with the remainder (less conj set) of the immediate surrounding F-structure will produce a well-formed F-structure.

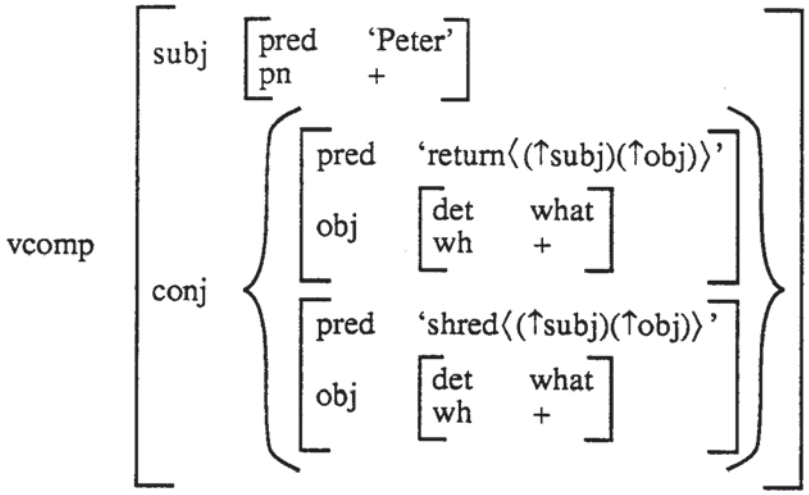


Figure 4.9.c Section of F-structure from Conjunction

This additional unification is however not necessary in cases where the set function forms the sole content of a subsidiary F-structure or function. This is the case when whole sentence structures are conjoined.

In other cases (ellipsed conjunctions), the content of the surrounding F-structure should be deleted, suitably indexed, and unified with each set member. The analysis of conjunctions proposed here thus places additional constraints on the production of sets in the case of conjunctions. Conjunctions are currently translated in the same manner as other F-structures and the resultant predicates simply conjoined. As an example of how an ellipsed conjunction can be dealt with in LFG (and quantifier scoping in such cases), the phrase *'each man drove a car through and completely demolished a glass window'*, taken from Dahl and McCord [1983] can be examined. This phrase is used by them to produce a structure with the basic form :

A X conj Y B

where *X* and *Y* are the coordinated structures. Dahl and McCord use a 'demon' process to parse such structures. On encountering the conjunction, a process is set off which backs-up in the parsing history in order to parse *Y* parallel to (as a repeated structure) *X*. The system does not produce the deeper unreduced form of the coordinate structure :

A X B conj A Y B

for :

> " Not expanding to the unreduced form is important for keeping the modifier relationships straight, in particular, getting the right quantifier scoping .... . This logical form [for the example phrase, which is given below] would be difficult to recover from the unreduced structure, because the quantified noun phrases are repeated in the unreduced structure, and the logical form that corresponds most naturally to the above [here given below] logical form. "

The logical form, produced by Dahl and McCord from the example coordination is :

```
each(M, man(M),
        exists(C, car(C),
                  exists(W, glass(W) & window(W) &
                            drove_through(M, C, W) &
                            completely( demolished(M, W) ) ))).
```

A possible LFG F-structure analysis of the example coordination is shown in outline in Figure 4.9.d. For simplicity, the adverb *completely* has been incorporated in the sense of the verb *demolished.* Adverbs are not dealt with in the Chat-80 semantic representation or in the account of LFG given by Kaplan and Bresnan [1982], but could be dealt with logically in the same way as Dahl and McCord have in the above logical form and in LFG by use of an additional semantic feature.

The F-structure in Figure 4.9.d illustrates the effect of ellipsed coordination. The F-structure as shown is not complete at the top-level (as there is no semantic form to subcategorize subj and obj) in this particular case. The individual members of the conj set are also not complete, they both are missing an obj and subj function.

Various approaches seem possible to the problem posed by the F-structure in Figure 4.9.d. If completeness is not checked until after parsing, then the conj set value could be treated specially during well-formedness checking, and the contents of the F-structure surrounding the conj function 'fitted' into the individual members of the set (as indicated by arrows in Figure 4.9.d). If multiple occurrences of the functions fitted into the conj members are co-indexed, then a 'correct' quantifier scoping is obtained, whichever conj member is given scope over the other. Alternatively, only the functional variables of the

outer functions could be passed into set members and then these functions translated with scope over all set members.

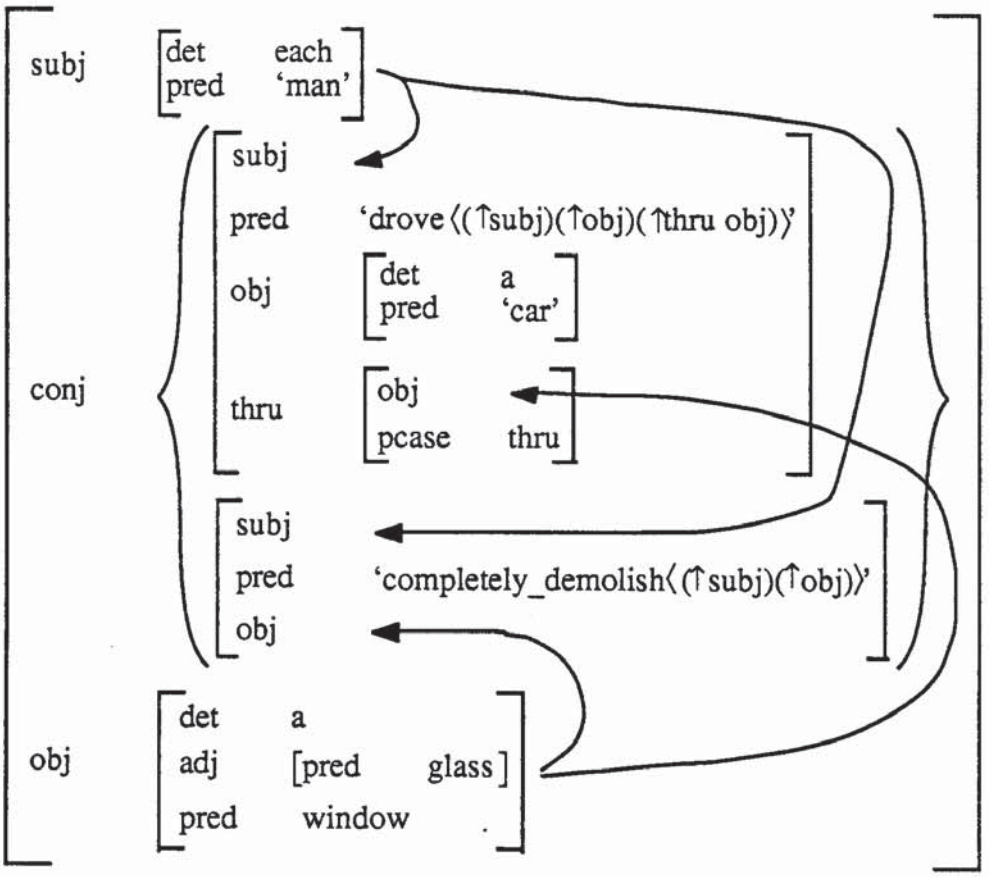


Figure 4.9.d Outline of F-structure from Ellipsed Coordination

In either case, it would seem necessary to treat F-structures containing coordinate sets in a special manner. This would obviously also be true of single ellipses themselves (*'which country has the largest population ?'* followed by a ellipsed query *'the largest area ?'*) where the ellipse's F-structure will not be complete, but is a component of the first query's F-structure. Thus, the ellipsed query's F-structure must be used to substitute some part of the first query's F-structure. In many cases, this would be possible, aided by semantic typing, but there would be the problem of ambiguity in exactly which function (of the candidate functions in the first query) the ellipse fulfils. The greatest problem in dealing with ellipses is to determine the grammar rule(s) with which to parse the ellipse. As a general rule, it seems that the rule(s) will be amongst those used during parsing of the first query and that the function fulfilled by the ellipse's F-structure will tend to be derived from C-structure on the RHS of the tree. It remains to be determined to what extent the inclusion of ellipses in the linguistic coverage of a system would degrade the efficiency of parsing.

4.10 Possessives

The head of a noun phrase may contain possessive constructs. In Chat-80 possessives with *'s* postfixes are analysed as genitive case markers. In LFG the possessive head constituents are used to form a function 'poss' which is subcategorized by the noun following the head function (Figure 4.10).

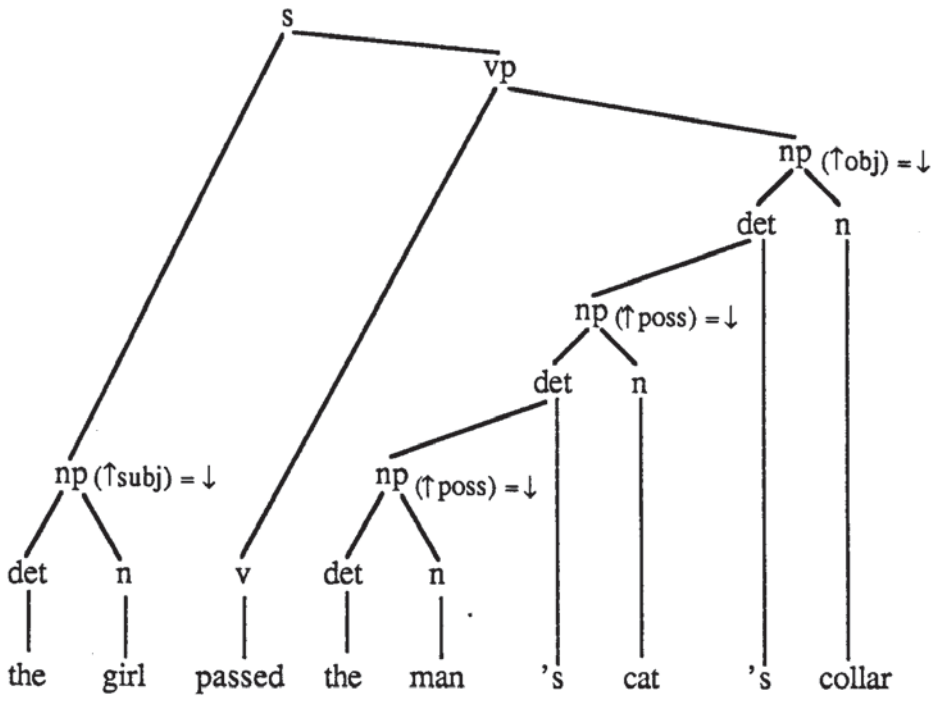


Figure 4.10 Possessive C-structure Outline

The F-structure produced by this analysis can be translated in the usual manner. By convention the function poss is given a default scope over the function in which it is contained.

Chapter 5

Prolog Techniques and Quintus Prolog

During the implementation of LFG, to be described in Chapter 7, several different approaches and Prolog techniques, used in implementations by others and used to produce a new implementation, were tried. Different approaches to parsing suggest different approaches to implementation. The purpose of some of these techniques may not be immediately apparent so that some explanation is required. Although various differing Prologs have been employed in various implementations the techniques described here are illustrated using Quintus Prolog [Quintus, 1987]. This Prolog is a development of Edinburgh Prolog [Pereira, Pereira & Warren, 1978; Clocksin & Mellish, 1984; Bratko, 1986] and represents a 'state of the art' programming environment for artificial intelligence, including, incremental interpretation or compilation, a sophisticated editor and debugging aids.

5.1 Open-Ended Lists

An open-ended list can be described as a ordinary Prolog list which has an uninstantiated tail (cdr in Lisp) :

```
[a, b, c, ..... |Cdr]
```

The variable tail of this list may be instantiated with another Prolog list to produce an ordinary Prolog list :

```
Cdr = [d, e]   ⟶   [a, b, c, d, e]
```

In this way, the list can be extended by any number of elements but once the tail has been unified with a list, no further extension is possible. If however, the tail of the list is unified with another open-ended list, then the resulting list will itself be open-ended and can be extended again :

```
Cdr = [d, e|Cdr1]   ⟶   [a, b, c, d, e|Cdr1]
```

An open-ended list may thus contains any number of elements, perhaps zero, and in addition, may have any number of new elements added any number of times. This data

structure is obviously very suitable in certain circumstances for representing feature structures. If two feature structures, Fs_1 and Fs_2, are represented as open-ended lists and unified (linguistically), the resulting feature structure can be produced by either adding features of Fs_1 not present in Fs_2 to the tail of Fs_2, or by adding features of Fs_2 not present in Fs_1 to the tail of Fs_1. If the list of new elements added to a feature structure is itself an open-ended list then the result of unification will be an open-ended list which may have further information added to it by subsequent unifications.

5.2 Pseudo-Declarative Procedures

Prolog procedures which operate on lists, or any structure of indeterminate size, such as the standard *member/2* procedure are normally declared as purely recursive procedures :

```
member(Element, [Element|_]) :- !.
member(Element, [_,|Rest]) :-
            member(Element, Rest).
```

The execution time and space taken by this procedure is thus dependent upon the list's length. This procedure has an infinite definition in a purely declarative form :

```
member(Element, [Element|_]) :- !.
member(Element, [_, Element|_]) :- !.
member(Element, [_, _, Element|_]) :- !.
member(Element, [_, _, _, Element|_]) :- !.
            ..............
            ..............
```

This declarative definition would execute more efficiently (in time and space) as Prolog's top-down search is more efficient than recursion (in Quintus Prolog). The membership test in the declarative form above is reduced to a simple top-down search for a matching predicate. In practice however, it is not always possible to provide declarative definitions of all procedures. If the length of lists involved in the membership test above cannot be guaranteed to have a reasonably small maximum value, then a declarative solution becomes impracticable. This problem can be overcome by making a compromise between declarative and recursive solutions. As an example, consider the following definition of *append/3* :

```
append([], List, List) :- !.
append([A], List, [A|List]) :- !.
append([A, B], List, [A, B|List]) :- !.
append([A, B, C], List, [A, B, C|List]) :- !.
append([A, B, C|Rest], List, [A, B, C|List1]) :- !,
            append(Rest, List, List1).
```

This definition functions declaratively if the first list passed to *append/3* has three or less elements but will perform some recursion when given a list of greater length. The number of recursive calls will however be less than that performed by the traditional definition of *append/3* as each recursion will consume three elements of the first list, not one as in the traditional definition. This type of definition is here called a "pseudo-declarative" definition.

5.3 Templates

Templates allow a procedure definition to be used in several different ways and uses the higher-order extensions to Prolog described by Warren [1982]. Consider the following procedure *sort/2* which performs a bubblesort on a simple Prolog list:

```
sort(List, Sorted) :-
      swap(List, List1),
      sort(List1, Sorted).
sort(Sorted, Sorted).

swap([Elem, Elem1|Rest], [Elem1, Elem|Rest]) :-
      greater_than(Elem, Elem1).
swap([Elem|Rest], [Elem|Rest1]) :-
      swap(Rest, Rest1).
```

where the predicate *greater_than/2* is an ordering relation which compares two elements of the list according to some predefined ordering. For example, if the list is a list of integers, the *greater_than/2* will simply be defined:

```
greater_than(A, B) :- A  >  B.
```

Given this definition of *greater_than/2*, the bubblesort procedure can only be used on lists of integers. If a list of say, characters were passed to the procedure, this would produce an error as comparison of characters using the operator '>' is illegal. Furthermore, only lists of integers can be sorted but not, for example, lists of elements which are themselves lists of integers.

The *sort/2* procedure can be made applicable to a much wider class of data structures by using templates. A template can be used in this example to specify both the structure of list elements and how they are to be compared. The definition of the *greater_than/2* procedure is altered to become :

```
greater_than(Template, Elem, Elem1) :-
      \+ gt(Elem, Elem1, Template).

gt(Elem, Elem1, temp(Elem, Elem1, Op)) :-
      \+ call(Op).
```

The elements for comparison are passed to *greater_than/3* as before but in addition a template is passed to the procedure which is used to make the comparison. This template is passed to the *sort/3* procedure which then passes it to *greater_than/3*. The template for a simple sort of a list of integers would be :

```
temp(A, B, A > B)
```

The actual values of elements are instantiated in the template by the procedure *gt/3* which then executes the comparison operator in the template. Double negation is used as failure of the *gt/3* predicate uninstantiates the variables in the template so that it can be used again. A more complex example might be the sorting of the list :

```
[ [a, 1], [b, 1], [b, 2], [a, 2] ]
```

If this list is to be sorted into an order so that sublists with lowest numbers appear first and in the case of two sublists having the same number then that with the letter closest to *a* is put first, then the template passed to the *sort/3* procedure would be :

```
temp([Let, Num], [Let1, Num1],
                ( Num > Num1
                ;
                   Num == Num1, Let @> Let1)
                )
```

where '`A == B`' ensures that the terms *A* and *B* are strictly identical and '`A @> B`' ensures that term `A` precedes term `B` in the standard order for terms (*a* before *b* etc). User defined comparison procedures can also be used in the template. The *sort/3* procedure can thus be used to sort lists of any data structure for which a template can be described.

Chapter 6

Implementations of LFG

6.1 DCG Type Implementation

At the time of starting this research only one implementation of LFG had been described in the literature [Frey & Reyle, 1983; Reyle & Frey, 1983]. This implementation is based upon the observation [Pereira & Warren, 1983, p139] that LFG can be realized as an extension of the Definite Clause Grammar (DCG) formalism. The DCG formalism is itself a simple and clear notation for an actual set of Prolog rules, which are produced from the grammar rules by adding string handling arguments. The DCG rule :

```
sentence    --->    noun_phrase ,  verb .
```

will be expanded into the Prolog rule :

```
sentence(S, S1) :-    noun_phrase(S, S2), verb(S2, S1).
```

A DCG rule may also be given extra arguments to ensure, for example, simple number and tense agreement or to build a syntactic tree representation of the phrase analysed. The DCG rule format has formed the basis of the LFG implementation described in Frey and Reyle [1983]. An LFG rule such as :

S	⟶	np	vp
		(↑ subj) = ↓	↑ = ↓
		(↑ tense)	

is transformed into a Prolog rule [Frey & Reyle, 1983, p54] :

```
S (*c10 *c11 *outps)  <----
        NP  (*c10 *c11 *featnp *outpnp),
        VP  (*c12 *c11 (SUBJ (*outpnp *featnp)) TEN *outps) .
```

where variables are denoted by prefixing with asterisks and constants are in uppercase. This rule can only really be understood in the context of a complete set of rules and lexical entries which with this rule analyses a phrase.

Prolog variable unification is used directly to implement (linguistic) unification although there is not a one-to-one correspondence between variables and features. Rather, arguments are used to build up F-structures as Prolog structures from basic elements in the lexicon. A *np* rule, for example, may expand *np* in the previous rule to a determiner and a noun :

```
NP (*c10 *c11 *outpnp)        <----
        DET (*c10 *c11 *outpdet) ,
        N (*outpdet *outpnp) .
```

Here, *outp* variables are instantiated with the output portion of F-structures (semantic forms) for their respective phrases so that *outpdet* is instantiated by matching with the respective lexical entry for the determiner. This is then passed to the noun part of the rule which matches with the *outpdet* variable perhaps also checking, by unification, number agreement and then incorporates this structure in its own output structure *outpnp*. The *n* category is termed the head category which is always that category assigned the trivial equation. The output from the head category is always also the rule's output structure. The *feat* variables carry simple feature values and *c* variables, a list of controllers. Variables which have names that are simple numbers carry a list of subcategorized functions originating from verb lexical entries. Every subcategorizable function appearing in a phrase must be able to shorten this list and the list must be empty at the end of parsing. This ensures both completeness and coherence. As Prolog unification is used directly, the functions found must match the first element of this list of functions. This introduces an additional (non-LFG) constraint, in that functions must be found in the same order in which they appear in lexical entry checklists.

The lexical entry for a verb such as *expects*, for use with the *s* rule above, takes the form :

```
V (  (VCOMP (SUBJ (*outpobj *featobj) ) )
     ( (SUBJ (*outpsubj (SG 3) ) )
            (OBJ (*outpobj *featobj) ) (XCOMP *outpxcomp) )
     TEN
     ( (TENSE PRES) (PRED 'EXPECT (*outpsubj *outpxcomp)' ) )  )
```

The first line of the verb's entry describes part of the vcomp function's structure (the subj). This is instantiated with values from the verb's obj function and thus corresponds to a functional control equation. This section of vcomp is later passed to a subsequent verb entry and incorporated in the vcomp function.

The second and third lines of the entry above constitute the checklist of subcategorized functions. Each function in this list has also an output variable and a feature variable which in the case of the subj function is instantiated with the values *sg* and *3* to ensure subject/verb agreement. The fourth line of the verb's entry simply names the verb feature *tense* as a constant which matches with the existential constraint value in the main *s* rule. The last line of the verb's entry is matched with the verb's output variable in the grammar rule. The output from the verb is the verb's semantic form, with arguments given the value of their respective function's output and the tense feature and value.

Long distance control is handled by variables which pass a list of controllers through grammar rules. The LFG rule :

$$s' \longrightarrow \underset{(\uparrow q) = \Downarrow^{np}_{[+wh]}}{np} \quad \underset{\uparrow = \downarrow}{\boxed{s}}$$

$$(\uparrow \text{focus}) = \downarrow$$

$$\downarrow = \Downarrow^{s}_{np}$$

is translated into a Prolog rule (the tail of a list '|' is represented by '.' in Frey and Reyle's rules) :

```
S' (*c10 *c11 *outpsc)      <——
        NP (((NP [+wh] . *c10) *c11 *featnp *outpnp),
        rest(*c11 *c10)
        S ((*outpnp *featnp (S NP))) nil *outpsc) .
```

where each *c* variable is a list of controllers, *c10* being the input list of controllers to this rule and *c11* the output list of controllers. Each controller in this list is represented by a sublist containing the controller's output variable, feature variable, superscript and subscript. The controller belonging to the *np* domain however does not carry features and output, presumably because the function q is non-subcategorizable or not used in

semantics. The controller belonging to the *np* domain is added to the front of the controller list passed into the *np* domain where a matching controllee must be found, although this does not seemed to be ensured by any checking. The list of controllers is returned as *c11* which would then be passed to the rest of the rule body (*rest*). A new list of controllers is produced for input to the *s* domain as this is a bounded node. This list contains only the controller belonging to the *s* domain and must be empty (nil) when returned.

The linking equation, which allows a controller to penetrate a domain the root of which is labelled as a bounding node, appears to have been misinterpreted by Frey and Reyle. They state that [Frey & Reyle, 1983, p55] :

> " Here [in the case of linking equations] we use a test procedure which puts only the controllers indexed by *s* [the superscript on the controller in the linking equation] onto the controller list going to the *s* goal [the bounding category]. "

It appears that the procedure used to deal with linking equations may thus pass a number of controllers into the bounding node's domain (all those with an *s* superscript). This is not however the interpretation seemingly intended by Kaplan and Bresnan [1982, p253], where it is stated that :

> " Thus, this schema [a linking equation] will link metavariables of any type, passing on to the lower controller the compatibility requirement of the upper one. "

Indeed if a single linking equation is intended to be applied to a number of controllers, then the generation of a number of new controllers into the bounding nodes domain would break the constraint that no node can serve as domain root for more than one controller.

Using Prolog's unification directly to perform linguistic unification in DCG rules requires establishing a fixed set of features for each grammatical category so that feature lists match correctly :

```
[num sg,    gen fem,    pers third]
   ↓           ↓           ↓        ('term' unification)
[num sg,      _,        pers third]
```

Unspecified values can be represented by anonymous variables which will match with any value. The efficiency of unification is thus not greatly affected by the number of features used in lexical entries although memory usage may be quite large. This method of realizing unification has been used to produce quite efficient implementations of unification type grammars [Sharman, 1987].

This DCG based type of implementation allows execution of grammar rules using Prolog's Top-Down Backtracking (TDB) and can use Prolog's unification directly to effect linguistic unification. Despite this, the implementation suffers from a number of weaknesses. The rules are very complex and difficult to understand, and therefore to alter, and require a deep understanding of Prolog itself. This is not desirable in a realization of a high-level notation. Also the implementation is incomplete in that sets and constraints are not dealt with. Set values could be handled by using open-ended lists to which set values are added but implementing the Kleene-star operator poses some problems. Top-down control does not allow left-recursive rules (np → np, mod), a natural description of many linguistic constructs, to be used. Only value type constraints could be dealt with directly when using Prolog unification but the important difference between a value constraint and a value definition would be lost. Prolog's TDB control may also frequently backtrack over previous work in the face of non-determinism in the grammar or ambiguity in the lexicon. It is possible to restructure a DCG so that backtracking is eliminated but this requires an understanding of TDB, inventing unnatural grammatical categories and often produces unclear grammatical descriptions. In the case of LFG, this restructuring would be complicated by the equations attached to rules. Non-determinism due to lexical ambiguity cannot be removed and as LFG incorporates a great deal of information in the lexicon, a relatively high degree of lexical ambiguity can be expected.

This DCG based implementation of LFG is used as the syntactic component of a system for discourse processing [Frey, Reyle & Rohrer, 1983] which uses Hans Kamp's Discourse Representation Theory (DRT) [1981], [Reyle, 1985]. This semantic theory is primarily concerned with anaphora which will not be discussed here.

6.2 Pseudo-DCG Type Implementation

In addition to the DCG based implementation, Eisele and Dörre [Eisele, 1984; Eisele & Dörre, 1986] also at the University of Stuttgart have produced an implementation based on (linguistic) unification using open-ended lists. Instead of using Prolog unification, linguistic unification is performed by a separate unification procedure *merge/2* :

```
merge(F_structure, F_structure) :- !.
merge([Attribute = Value|Rest], F_structure1) :-
        del(Attribute = Value1, F_structure1, F_structure2),
        merge(Value, Value1),
        merge(Rest, F_structure2) .

del(Pair, [Pair|Rest], Rest) :- !.
del(Pair, [Pair1|Rest], [Pair1|Rest1]) :-
        del(Pair, Rest, Rest1) .
```

The *merge/2* predicate is passed two F-structures as open-ended lists and instantiates the tails of these so that the F-structure lists become identical, should they not contain contradicting values for attributes. This is done by inserting into both F-structures the attributes missing with respect to the other. The first *merge/2* rule simply unifies two F-structures, attribute values or open-ended list tails. The second rule takes an attribute and value from the first list, deletes this attribute's value from the second list (*del/3*), returning the remainder of this list, and then merges the attribute's values and the rest of the lists. If however the attribute is not defined in the second list the procedure *del/3* will add this attribute to the list and the subsequent merge on the attribute's value will actually instantiate the attribute's value. The recursive call to *merge/2* on attribute values is also used to unify function values, themselves open-ended lists. A lexical entry of the verb *hand* for example, may be represented by an open-ended list and merged with a subject function representing *'a girl'* :

```
merge([subj = [spec = def, num = sg, pred = girl|Rest_subj1]
          |Rest1],
      [pred = hand(subj, obj2, obj), tense = pres,
          subj = [num = sg|Rest_subj2]
          |Rest2]) .
```

which results in the following instantiations :

```
Rest_subj2 =  [spec = def, pred = girl|Rest_subj1]
Rest1 =       [pred = hand(subj, obj2, obj),
                      tense = present|Rest_subj1]
```

The *merge/2* predicate is also used to process equations in grammar rules. The LFG rule :

s	⟶	np	vp
		(↑ subj) = ↓	↑ = ↓

is translated into a DCG type rule :

```
s(S)  -->  np(NP), { merge([subj = NP|_], S) },
           vp(S) .
```

where the additional call to *merge/2* is enclosed in braces so that when expanded into a Prolog rule, string handling arguments are not added to this predicate.

Constraints are handled by the special Prolog II functions *diff* and *freeze*. The *freeze* function allows 'frozen' conditions (goals) to be attached to variables which are executed when the variable becomes instantiated. An existential constraint, for example, is dealt with by attaching a frozen condition (≠ nil) to the value of the feature concerned, which is given a variable as its value. After parsing, all unbound variables are instantiated to nil, at which time the condition will be executed causing a failure as the F-structure is ill-formed. The function *diff* fails if its arguments can be unified but it does not attempt to do this if its arguments contain parts represented by different variables. This function can thus be used to support negative value and negative existential type constraints. Coherence is handled by passing the F-structure, less subcategorized functions, to a predicate *ngf* which checks that the remainder of the F-structure does not contain any further governable functions. The lexical entry for the verb *promised* :

promised	:	v	(↑ tense) = past
			(↑ pred) = 'promise⟨(↑ subj)(↑ obj)(↑ vcomp)⟩'
			(↑ vcomp to) = $_c$ +
			(↑ vcomp subj) = (↑ subj)

thus takes the form :

```
v(V)    -->    [promised],
               { merge(V,
                [tense = past,
                 pred = promise(subj, obj, vcomp),
                 subj = [pred = Pred_subj|Rest_subj],
                 obj = [pred = Pred_obj|_],
                 vcomp = [to = Plus,
                          pred = Pred_vcomp,
                          subj = [pred = Pred_subj|Rest_subj]|_]
                |Rest]),
                 freeze(Plus, Plus = +),
                 freeze(Pred_subj, Pred_subj \= nil),
                 freeze(Pred_obj, Pred_obj \= nil),
                 freeze(Pred_vcomp, Pred_vcomp \= nil),
                 ngf(Rest) } .
```

Long distance dependencies are dealt with in a similar manner to that in the DCG type implementation, by passing input and output lists of controllers to C-structure nodes. Prolog unification is not however directly employed for matching controllers with controllees. Instead, additional goals (in braces) are added to take controllers off the controller list. The grammar rules :

(1) np $\longrightarrow$ e

$\uparrow = \Uparrow_{np}$

(2) s $\longrightarrow$ np S1

$(\uparrow \text{focus}) = \downarrow$ $\uparrow = \downarrow$

$(\uparrow q) = \Downarrow^{np}_{[+wh]}$ $(\uparrow \text{aux})$

$\downarrow = \Downarrow^{s1}_{np}$

are translated into two corresponding DCG rules :

```
(1)  np(Ctrls, Ctrls1, Fs_np)   -->
                [],
                { subst(np/Fs_np, np, Ctrs1, Ctrls1) }.
```

```
(2)  s1(Ctrls, Ctrls1, Fs_s1)   -->
           np([wh/Q|Ctrls], [wh|Ctrls1], Fs_np),
           s([np/Fs_np], [np], Fs_s1) .
```

The identification of controller domain roots is performed during translation into Prolog. A controller is added to the list of controllers at its domain root in the C-structure. This category is not necessarily the category below which a controller appears in an equation. As can be seen in the sample rules above, a domain root adds a controller to the list of controllers. The controller list elements consist of the controller subscript and a variable representing the F-structure. A controllee removes a controller from the list and replaces it with a receipt marker. The predicate *subst/4* does this and is allowed to match the first *n+1* elements of the list, where *n* is the language dependent crossing limit. The domain root will then ensure that its controller has been matched by removing the receipt marker from the controller list.

Eisele also shows how left-recursion in grammar rules may be eliminated by a transformation which, given the rules :

A ⟶ A, B. A ⟶ C.

will produce a new set of rules :

A ⟶ A1, A2 A2 ⟶ B, A2.

A1 ⟶ C. A2 ⟶ [ ].

This new set of rules is weakly equivalent to the original rules in that it recognizes the same strings (terminal sequences). The C-structures produced by both sets of rules, and the passing of F-structures through these (indicated by arrows), are illustrated by Eisele [1984] using a terminal string 'C B B B B' (Figure 6.2). The C-structure tree produced by the new rules *(b)* will be right-branching rather than left-branching *(a)*.

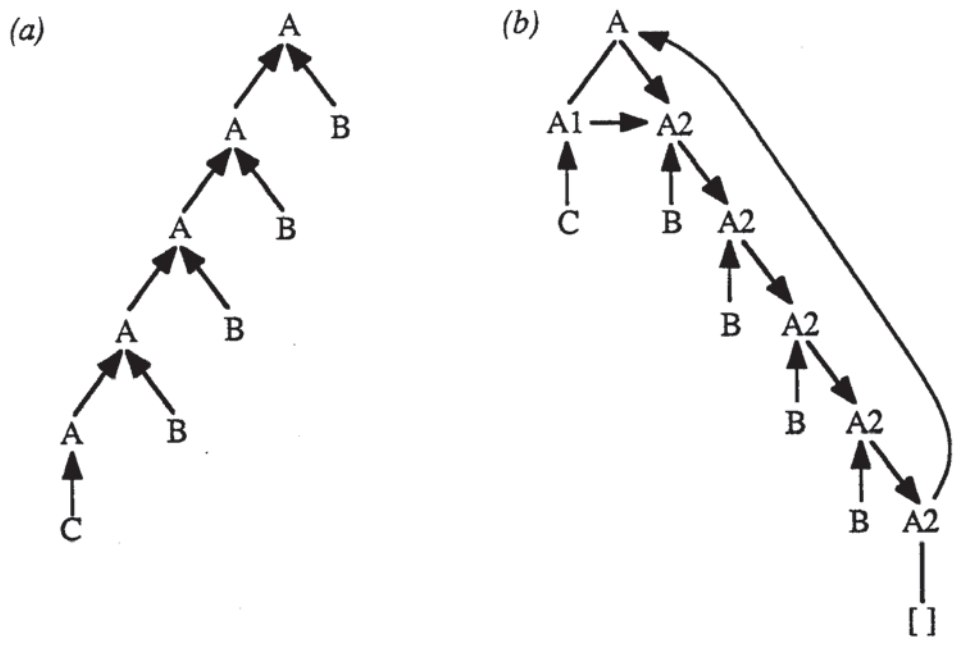


Figure 6.2 F-structure Passing through C-structure Produced from Left-Recursive Rules and Eisele's Equivalent Rules

The relationship between categories in the two sets of rules and building of F-structures using the new rules, is described by Eisele :

> " In the old grammar one could say that each A node takes the F-structure(s) of its subordinate A and B or C node(s) and uses them to build its own F-structure.
>
> In the new grammar, the A1 node behaves just as the A node dominating the C node in the old grammar, but the A2 nodes can be considered as taking the F-structures of the left-sister A1 node or the dominating A2 node and that of the subordinate B node to build its own F-structure.
>
> If the F-structures are evaluated in this way and passed downward, then the F-structure belonging to the A node will be assigned to the A2 node which expands to the empty string.
>
> We have to provide an extra argument for passing the F-structure of A through all the A2 goals [back up to the A node, as illustrated above]. "

This transformation however, requires introducing new grammatical categories (*A1, A2*) for which there is no linguistic motivation and two additional rules. This may result in

parsing actions which are difficult to understand. In addition to this the transformation does not take into account LFG equations and bounding nodes.

Eisele's system has been tested using about thirty grammar rules and two hundred lexical entries but he reports that its performance is very poor. This is mainly caused by backtracking in the top-down parser which is caused by the nature of an LFG grammar. In LFG grammars, there are many optional constituents which can be repeated an arbitrary number of times and in addition to this, a single grammatical category may have alternative equation sets.

More recently, this implementation has been used as the basis of a machine translation system [Netter & Wedekind, 1986] where a set of transfer rules convert from the C-structure and F-structure representing a phrase in one language (German) to those representing the phrase in another language (French). This implementation has also been used [Netter, 1986] as the basis of a system for dealing with German word order. Other translation systems based on LFG have also used transfer rules to convert F-structures between languages [Kudo & Nomura, 1986], [Horsfall, 1986]. The implementation developed at UMIST, used by Horsfall, appears to be very similar to Eisele's. LFG rules are translated into DCGs with embedded extra goals added to perform operations according to equations.

To overcome the inefficiencies of top-down control problem, Eisele proposes using C-Prolog and adopting the bottom-up parsing method described in Matsumoto and Tanaka [1983]. However, as C-Prolog does not have the *diff* and *freeze* functions, much of Eisele's implementation would have to be reworked. Also, the *merge/2* procedure which unifies two F-structures is very computationally expensive, particularly the *del/3* predicate which recurses along open-ended lists looking for a feature and its value. This is the cost of using the more compact F-structure representation where a category may be given any number of features in any order.

Yasukawa [1984] has developed a similar LFG system to that of Eisele and Dörre which is also based on the DCG formalism and embeds extra Prolog goals in DCG rules. F-structures are generated during parsing by executing the equations attached to nodes in C-structure. After parsing a complete string, the F-structure is checked to see if values are consistent with constraints and to see if it is complete. Yasukawa employs the following data types to represent F-structure components :

- simple symbols as atoms or integers.

- semantic forms as 'sem(X)' where *X* is a predicate.

- F-structure as 'Id : Obt' where *Id* is an identifier variable (Id-variable) and *Obt* an ordered binary tree. A unique identifier variable is used to identify the F-structure of each syntactic node and F-structures themselves are represented as ordered binary trees where each tree leaf holds an attribute and value pair.

- sets as $\{value_1, value_2, ..., value_n\}$.

A binary tree is represented by a Prolog term which takes the form :

```
obt(v(Attribute, Value), Less, Greater).
```

where the term *'v(Attribute, Value)'* is the value held at this node in the tree, *Attribute* is used as this node's label and *Greater* and *Less* are ordered binary trees which hold attributes with label values greater than that at this node and less than that at this node respectively. If a tree leaf is not defined, it is represented by a single Prolog variable so that the tree may be extended in the same way as an open-ended list. Ordered binary trees are claimed to cut the processing time of operations on F-structures over the use of ordinary lists by thirty per cent. The predicates which implement LFG primitives are (*dn* is a designator, *s* a set) :

d1 = d2	`equate(d1, d2, Old, New).`
d ∈ s	`include(d, s, Old, New).`
$d1 =_c d2$	`constrain(d1, d2, Old_c, New_c).`
d	`exist(d, Old_c, New_c).`
$\neg(d1 =_c d2)$	`neg_constrain(d1, d2, Old_c, New_c).`
¬ d1	`not_exist(d, Old_c, New_c).`

The variables *Old* and *New* are global value assignments (F-structures) and *Old_c* and *New_c* are constraint lists. In addition to these predicates, additional predicates are used to check the constraint list during parsing to kill off incorrect parses as soon as possible. The predicate calls are actually inserted into DCG rules by expanding a more readable and concise notation. The LFG rule :

s	⟶	np	vp
		(↑ subj) = ↓	↑ = ↓

is written as :

```
s(s(Np, Vp), Id_s, [ ])        -->
        np(Np, Id_np, [eq([subj, Id_s], Id_np])),
        vp(Vp, Id_vp, [eq(Id_s, Id_vp)]) .
```

which is expanded into a DCG rule :

```
s(s(Np, Vp), Id_s, Old, New, Old_c, New_c) -->
        np(Np, Id_np, Old, Old1, Old_c, Old_c1),
        { equate([subj, Id_s], Id_np, Old1, Old2) },
        vp(Vp, Id_vp, Old2, Old3, Old_c1, New_c),
        { equate(Id_s, Id_vp, Old3, New) } .
```

Yasukawa compares his pseudo-DCG implementation with the DCG implementation of Reyle and Frey noting that although the DCG approach may be more efficient, using extra goals allows much more of LFG to be supported, especially constraints, and that rules are much simpler and easier to modify.

6.3 Recent Implementations

An Integrated Parsing (IP) system using a grammar based on LFG has been developed by Uehara *et al* [1984b; 1985]. This system is based upon "actor theory" [Hewitt, 1977], where grammar rules and lexical entries are represented by actors. A grammatical actor has the form :

[<actor_name> <script>]

where *<actor_name>* is the LHS of a grammar rule and *<script>* a set of RHSs of rules which expand the LHS. A script contains a number of "patterns" which represent the RHSs of rules which have a common LHS. A lexical actor has the form :

[<actor_name> [<non_terminal> <schemata>]]

where *<actor_name>* is a terminal word, *<non_terminal>* the word's grammatical category and *<schemata>* a set of lexical equations.

Although actor theory assumes that communication between actors is performed in parallel, the parser is limited to constructing C-structure in a traditional top-down manner with automatic backtracking by the Prolog interpreter. When an actor has some applicable

patterns, it activates the first of these, while all other patterns remain inactive, and evaluates the equations attached to the first element (category) of the pattern, modifies its internal state to reflect this and sends a message to the grammatical actor with the category of the pattern's first element (TD expansion). The actor receiving this message will then evaluate its first pattern according to the message sent and returns the result of the transmitted message to the sender. The sender then evaluates the equations attached to the second element of its first pattern and repeats the process. The flow of messages during parsing is illustrated in a simple example by Uehara *et al* [1985, p84] which is reproduced in Figure 6.3.a. The order of messages and replies (during parsing) has been added to this to clarify the TD control.

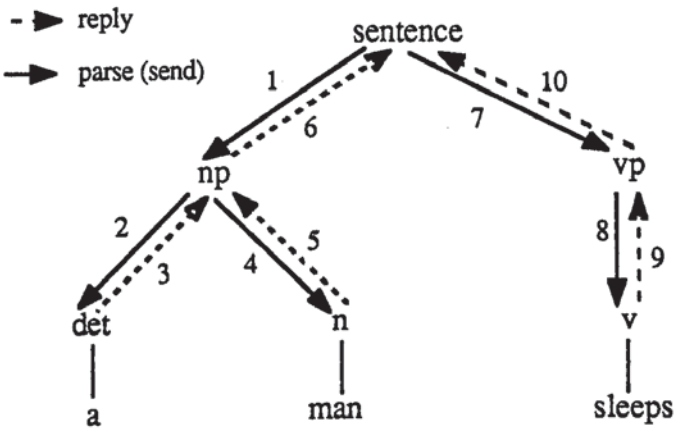


Figure 6.3.a Flow of Messages and Replies in Integrated Parser

If parsing fails, either because an actor cannot construct a proper F-structure of the sentence or has no more patterns (possible expansions to try), then control reverts to the most recently activated actor which transmits a message to abandon the computations made and then selects the next of its patterns to use as an alternative expansion. A message consists of four parts :

(1) F-structure, an identifier for the 'parent' F-structure.

(2) trail list, which is used as a push-down list used to store the names of actors which should be reactivated on backtracking.

(3) hold list, which holds controllers passed down the C-structure in messages.

(4) constraining equation list, which holds constraint type equations which have yet to be satisfied.

The representation of F-structures themselves is not described in detail but each F-structure and subsidiary F-structure is identified by a unique segment identifier (integer). When a message is transmitted the actor sends its F-structure identifier *(1)*. This is then used to replace the '↑' metavariables in the receiver's equations. An actor will then add its own name to the trail list *(2)* in any messages it produces and may add or remove controllers and controllees to the hold list *(3)* according to the equations it evaluates. An actor also creates a unique identifier for the F-structure below (↓) which is used to replace all references to this in the equations evaluated. Constraining equations *(4)* are checked by searching for values in the F-structures below and above the current level and if these refer to values which have not yet been determined, they are recorded in a message and their evaluation postponed (the method of their eventual evaluation is not described).

The basic action of the IP parser then is to view the C-structure as a 'program outline' where each node is realized by an actor (procedure) which interprets messages from other actors. The semantics of LFG equations is reinterpreted according to actor theory, so that the required messages are defined by equations. A '↓' metavariable is read as 'receive' and a '↑' metavariable as 'send'. Uehara *et al* then reinterpret references to F-structures in equations according to which side of an equation they appear on :

> " For example, if the designator '↑ subject' appears at the left-hand side of a schema, it would be read as :
>
> 'send the receiver an F-structure with the attribute subject'
>
> If it appears at the right-hand side, it would be read as :
>
> 'send the receiver a request to get an F-structure with the attribute subject'. "

This interpretation certainly differs from that intended by Kaplan and Bresnan [1982] where no significance is ascribed to the side of an equation on which a F-structure reference or metavariable occurs. It seems likely that these interpretations are not actually those intended as they do not even agree with the (more reasonable) interpretation of a complete grammar rule which is also given. A grammar rule such as :

$$\begin{array}{lcll} s & \longrightarrow & np & vp \\ & & (\uparrow \text{subj}) = \downarrow & \uparrow = \downarrow \end{array}$$

in the actor orientated reading is read as [Uehara *et al*, 1985, p 82] :

" receive an F-structure of an *np* and send it to an *s* with the attribute subject, then receive an F-structure of a *vp* and send it to the *s*."

Message sending seems therefore to be based on the sending of complete F-structures, as indicated by the form of messages themselves (given above).

The IP system has been extended into text (discourse) processing. This extension is based on the use of "prediction" and "presupposition". A semantic form is extended to 'subcategorize' not only surface functions within a sentence but also a presupposition (event) in the previous sentence(s) and a prediction about the next sentence (event). For example, consider the following sentences and their semantic forms with filled arguments :

(1) 'John went to a pet shop.'

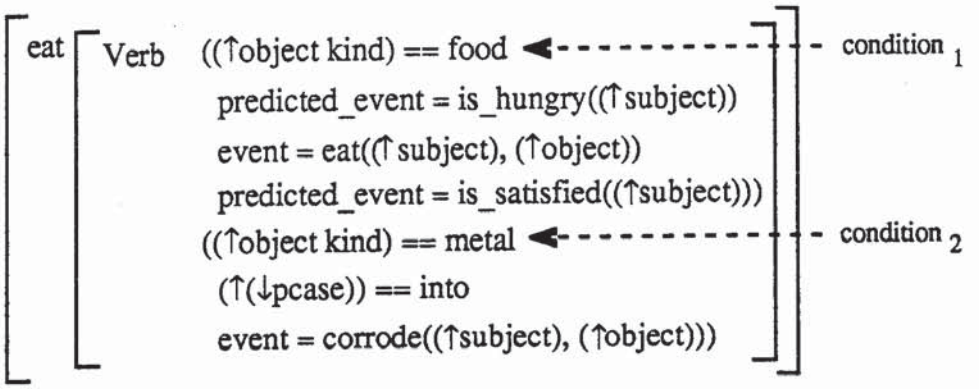
presupposed_event = come(john, (↑ place), pet_shop)
event = go(john, pet_shop)
predicted_event = stay_in(john, pet_shop)

(2) 'There he bought a dog.'

presupposed_event = stay_in(he, pet_shop)
event = buy(he, dog)
predicted_event = pay(he, (↑ obj), money),
get(he, dog)

Processing of the first sentence *(1)* produces a presupposed event that John came to the pet-shop from some other place, specifies an event of John going into the pet shop and predicts that the next event will be John staying in the pet shop. The second sentence *(2)* presupposes that 'he' stays in the pet shop, records the event of *him* buying a dog and predicts that *he* will then pay some some and then have a dog. The prediction produced from sentence *(1)* correctly matches with the presupposition of *(2)* and this relationship can then be used identify *he* in *(2)* as *John* in *(1)* and also *there* in *(2)* as the *pet shop* in *(1)*. This type of referent identification is sometimes not possible using purely syntactic agreements but can be processed using the extra semantic knowledge proposed here.

Lexical actors also have additional conditions added to aid in selecting between ambiguous alternatives. The lexical actor for *eats* is :



The event feature can be seen to be equivalent to a semantic form and the conditions are used to select the appropriate semantic form in any context when ambiguity arises. This additional feature however has exactly the same effect as the typing of slots used by McCord and described earlier.

The IP parser is only described in outline by Uehara *et al* but appears to be a novel approach to constructing F-structure in parallel to C-structure. However, message passing has had to be severely constrained to support practical parsing and fit the generation of C-structure. C-structure is generated in a TD manner so that this forms the basic control structure which is extended to pattern invocation in actors. In addition, F-structure seems to be held as a global data structure, with pointers to F-structure portions passed between actors, rather than being distributed across the actors with equations initiating message passing (as seems to have been intended in part of the description). The extension for text processing relies heavily on very specialized semantic forms which make very precise statements about presuppositions and predictions. This knowledge is however very cleanly added into the lexical entries of LFG and processed on top of the F-structure layer. The number of possible alternatives in presuppositions and predictions may however be very large and thus difficult to describe.

A more recent implementation of LFG, described by Block and Hunze [1986], has moved away from the use of DCGs in favour of the Earley algorithm [Earley, 1970]. F-structure is constructed incrementally in parallel to C-structure. The parser operates on a list of ordered states to the end of which new states are added. A state is a tuple :

(<tree> <left> <right> <dot> <pred_list>)

where *<tree>* is the current parse tree of the path, *<left>* a pointer to the place in the input string the constituent begins with, *<right>* a pointer to the place in the input string that immediately follows the constituent, *<dot>* marks the current position in the RHS of the context-free grammar rule and *<pred_list>* is a set of pointers to all proceeding states which might become the parent of this state's tree. The basic parsing actions of the Earley algorithm are retained :

- "predict" : expands non-terminals to produce a new state for each alternative expansion of the non-terminal.

- "scan" : adjusts states which have the next string terminal as the next part of their RHS moving the dot on one place in the RHS of the rule.

- "complete" : completes a state by adjusting the dot marker in states, via the pointer list, which have the LHS category of the completed rule as the next category in their RHS.

These being integrated into the state representation :

> " For the construction of the C-structure these actions are augmented in the following way : predict creates an empty tree node labeled with the predicted category, scan attaches the next input word as the rightmost daughter to the state's *<tree>*, and complete attaches the state's *<tree>* as the right most daughter to all the tree nodes in the states of the current state's *<pred_list>*. For the construction of the F-structure the following augmentations are performed : the *<dot>* part of a state not only marks the position in the cf-rule's right hand side, but also contains the functional equations associated with that position. When predicting a constituent an empty F-structure is attached to it and incremented by scanning a word or completing the phrase. "

To evaluate the functional equations attached to a category in a grammar rule, the parser instantiates the up and down-arrows in the equations with copies of the mother's and daughter's F-structure. The equations are then evaluated and a copy of the new F-structure associated with the up-arrow becomes the F-structure of the new state. As an

example, Block and Hunze illustrate the parsing of *'this man loved Mary'*, noting the (corrected) trace :

State of analysis	F-structure
predicting *s*	[ ] S
predicting *np*	[ ] NP
scanning *this*	[det = dpron, num = sg] NP
scanning *man*	[det = dpron, num = sg, pred = man] NP
completing *np*	[subj = [det = dpron, num = sg, pred = man]] S
predicting *vp*	[ ] VP
scanning *loved*	[tense = past, pred = love(↑ subj)(↑ obj)] VP
predicting *np*	[ ] NP
scanning *Mary*	[pred = Mary, num = sg] NP
completing *np*	[obj = [pred = Mary, num = sg]] NP
completing *vp*	[tense = past, pred = love(↑ subj)(↑ obj), obj = [pred = Mary, num = sg]] VP
completing *s*	[subj = [det = dpron, num = sg, pred = man], tense = past, pred = love(↑ subj)(↑ obj) obj = [pred = Mary, num = sg]] S

As F-structure is built incrementally in parallel with C-structure, this allows F-structure to act as a filter killing off incorrect parses during the actual parsing process. There are however alternatives in which of the parsing actions are used to carry out unification and detect incorrect parses. Consider parsing the sentence *'these man loved Mary'*. When scanning *man*, the parser tries to unify the F-structure originating from the lexical entry of *man* with that of the *np* parsed so far. At this point, the inconsistency in number (*these* - num = pl, *man* - num = sg) causes unification and thereby this parse to fail. However when parsing, for example, *'these men loves Mary'* the inconsistency in number between the verb and initial *np* could be detected immediately when scanning the verb. This would require that the F-structure built so far for a state is passed to all states created by the predictor. However this is a very computationally expensive practice, not recommended by Block and Hunze, as the Earley algorithm requires that no new state is generated if that exact state has been generated previously. This being a requirement to prevent the parser looping. This means that whenever a new state is added to the set of current states, the entire set of states must be searched to ensure that state has not already been predicted at that point in the string. Comparing F-structures is a very costly operation. The benefits of incrementally building F-structures on the completer stage are however still great, reducing greatly the explosion in number of C-structure parses.

The coherence condition can obviously be used to kill off incorrect parses when building F-structures in parallel to C-structures. Block and Hunze also add additional constraints into LFG in order to support some completeness checking during parsing. As described by Kaplan and Bresnan [1982], completeness cannot be checked until all of an F-description has been taken into account. This means that completeness cannot be checked until parsing is complete. In particular, equations of the type :

$$(\uparrow F2) = (\downarrow F1)$$

where function *F1* is subcategorizable, can introduce a function into an F-structure, sometime after its constituents have been parsed, making the F-structure complete. This problem is overcome by extending the notion of bounding categories and assuming that they define strict islands in C-structure. Not only are controllers and controllees prevented from passing such nodes, but also no equation of the type shown above may be associated with such a node. Then the complete stage of the parsing action may perform completeness checking whenever a state is marked as a bounding node.

Block and Haugeneder [1986] have also described a new treatment of movement which combines ideas taken from LFG and Government Binding (GB) [Sells, 1985] theories. This system is basically the same as that described above, in that it uses the Earley algorithm, but the movement mechanism is changed to a cyclic based mechanism found in GB. As GB theory is outside the coverage of this thesis, this variation of LFG will not be described here.

Wedekind [1986] has developed a version of LFG which also derives F-structure in parallel with C-structure, although this appears not to have been actually implemented at this time. This theory is based on the idea of a "monostratal" version of LFG where only an F-structure, augmented with derived symbols and their linear order, is produced. This is proposed as a more efficient approach :

> " as the F-description solution algorithm is directly simulated during the derivation of these structures instead of being postponed. "

This is an alternative approach to those described previously. Kaplan and Bresnan [1982] describe LFG as producing an F-description which after parsing is then solved to produce

an F-structure. The implementations described previously (here) have, in general, derived F-structure in parallel to C-structure to improve the efficiency of parsing, as F-structure functions as a filter on incorrect C-structures. These implementations use the properties of unification (which may fail) and coherence to kill off incorrect parses as soon as possible in the parsing process. Block and Hunze's implementation also introduces additional constraints into LFG so that in certain circumstances completeness can also be checked.

Wedekind however proposes actually combining C-structure and F-structure rather than just developing them in parallel. Obviously, this will produce the same final result but the representation may be more compact and handled more efficiently. Wedekind develops his monostratal representation in a number of steps which are outlined here. A phrase analysis is represented by a triple :

$$<c, d, f>$$

where c is a C-structure, d a (partial) F-structure and f a mapping from C-structure nodes to (sub) DAGs of the F-structure. Each node in C-structure is given a unique label and the mapping function provides path names to the F-structure of each node in C-structure. A rule application expands each part of the triple. Wedekind illustrates this with an example using the rule $(\overline{r})$:

$$\overline{r} = \quad \text{vp} \longrightarrow \begin{array}{cc} \text{np} & \text{vp} \\ (\uparrow \text{obj}) = \downarrow & (\uparrow \text{vcomp}) = \downarrow \end{array}$$

which is represented as a rule :

$$<p_1(r), <\{<\emptyset, p_2(r)>\}, d_r, f^r>>$$

where $p_1(r)$ is the left-hand side of the rule, $\{<\emptyset, p_2(r)>\}$ the introduced C-structure (for the LFG rule above this would be $\{<\emptyset, \{<1, np>, <2, vp>\}>\}$), d_r the introduced F-structure and f^r the mapping from C-structure to F-structure. The initial parse state triple (s_0) is thus $<\{<1, s>\}, d_0, f^0>$ where f^0 is an empty mapping and d_0 a minimal DAG (a placeholder). This rule is then applied to the triple $S_{i-1} = <c, d, f>$ which is illustrated graphically in Figure 6.1.

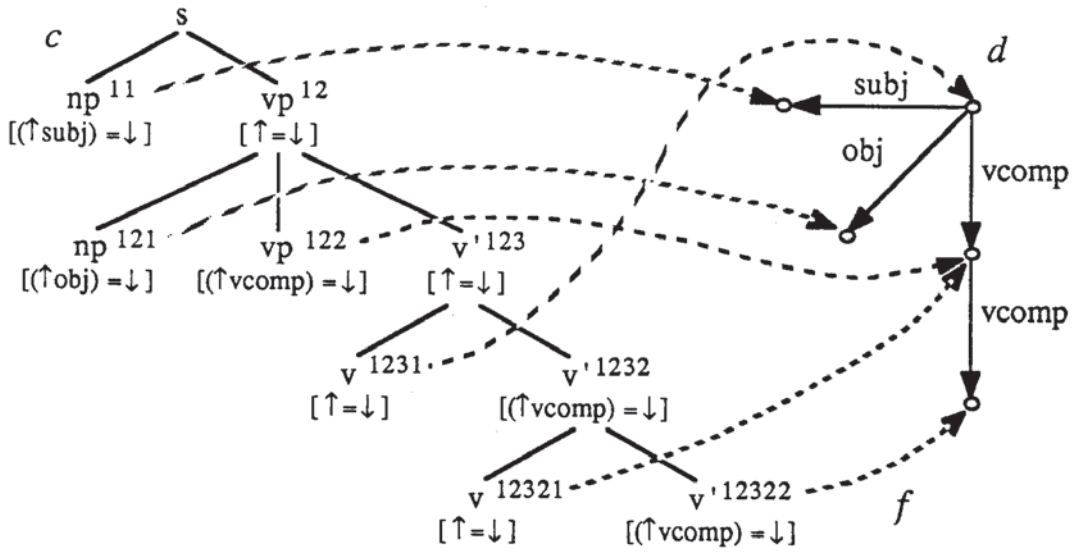
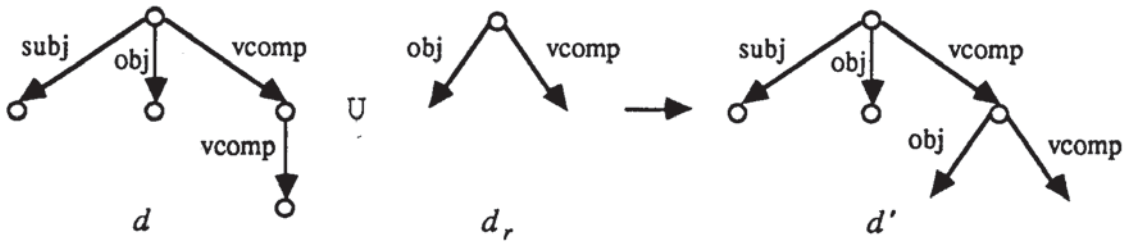


Figure 6.3.b Graphical Illustration of Triple for State S_{i-1}

to produce a new triple $S_i = <c', d', f'>$ in the following manner. The rule $\bar{r}$ is applied to the terminal node vp^{122} in C-structure to produce two new terminal nodes, np^{1221} and vp^{1222} where the new nodes are labelled with the parent node's label suffixed with the nodes number in the rule $\bar{r}$. The F-structure is revised by first expanding all '↑' metavariables in the rule with the F-structure associated with C-structure node vp^{122}, thus $f_{122} = f'^{r}_{\emptyset}$ and then the F-structure of the rule, containing a vcomp and obj function, is unified with the F-structure of node *122* (that is *d*'s vcomp is unified with d_r) and by substitution this new F-structure becomes the value of f'_{122}:



The new mapping f' must include mappings for the new C-structure nodes which are produced by extending the path names of their parent nodes, other mappings remain unaltered. The new path names (mappings) are produced simply by appending each of the path names of the new node's F-structures within the rule's F-structure to the path name of the parent node. Thus the mapping function values of the new nodes are derived:

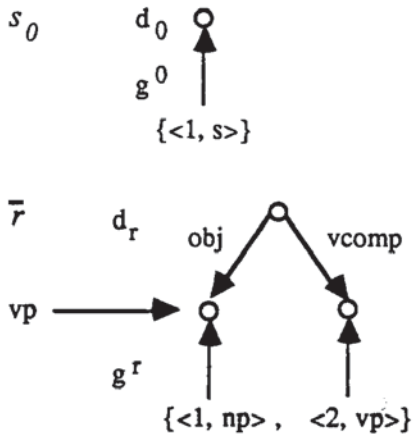
f_1^r = d_r obj f_{122} = d vcomp ⟶ f'_{1221} = d' vcomp obj

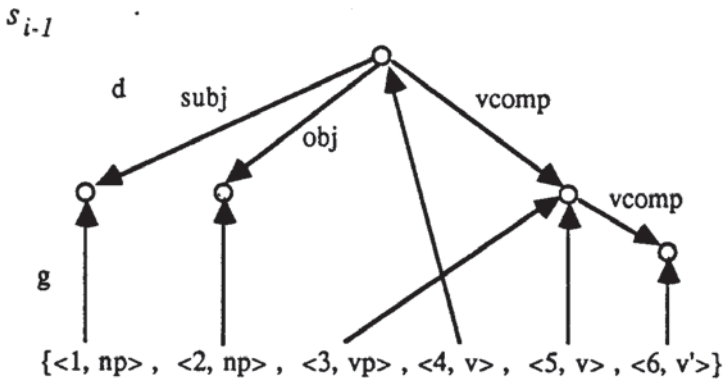
f_2^r = d_r vcomp f_{122} = d vcomp ⟶ f'_{1222} = d' vcomp vcomp

The outline of analysis above still however retains a separation of C-structure and F-structure, these are only connected by a mapping, not by combination. The monostratal representation is then developed by Wedekind by first changing the triple representation so that C-structure is replaced by strings. This is possible, as at any point in the analysis of a phrase, only the values of *f* at terminals in C-structure are used when deriving the next analysis triple. A triple is now represented as :

<w, d, g>

where *w* is a string, *d* an F-structure as before and *g* maps strings to terminals in *d*. Strings are only represented as terminals with their linear order as Wedekind states : " The C-structure information which goes beyond the linear order of the labelled terminal nodes is not required ". As in triples, *g* replaces *f* in rule representations. This results in the following graphical representations of *w*, *d* and *g* for, the initial state, the rule $\bar{r}$ and the state s_{i-1} :

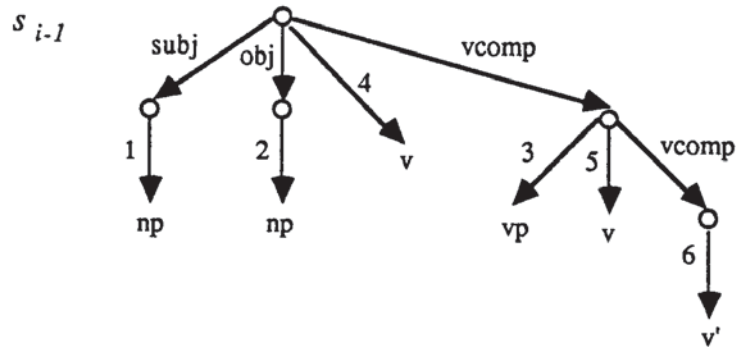




This allows F-structures to be developed in parallel to strings, the production of state s_i will not be described for this representation as it is very similar to that for the final monostratal representation to be described immediately below.

The monostratal representation is derived by combining the terminal strings and their linear order with *d* and thus eliminating the need for *g*. This is done by simply attaching the string and linear order argument of each element in *g* as an additional edge of

the subsidiary DAG referenced by the corresponding path name argument in g. Thus for example, the state s_{i-1} is represented as an augmented DAG :



The application of rule $\bar{r}$ on $<3, vp>$ on state s_{i-1} (with DAG d) is illustrated graphically in Figure 6.3.c.

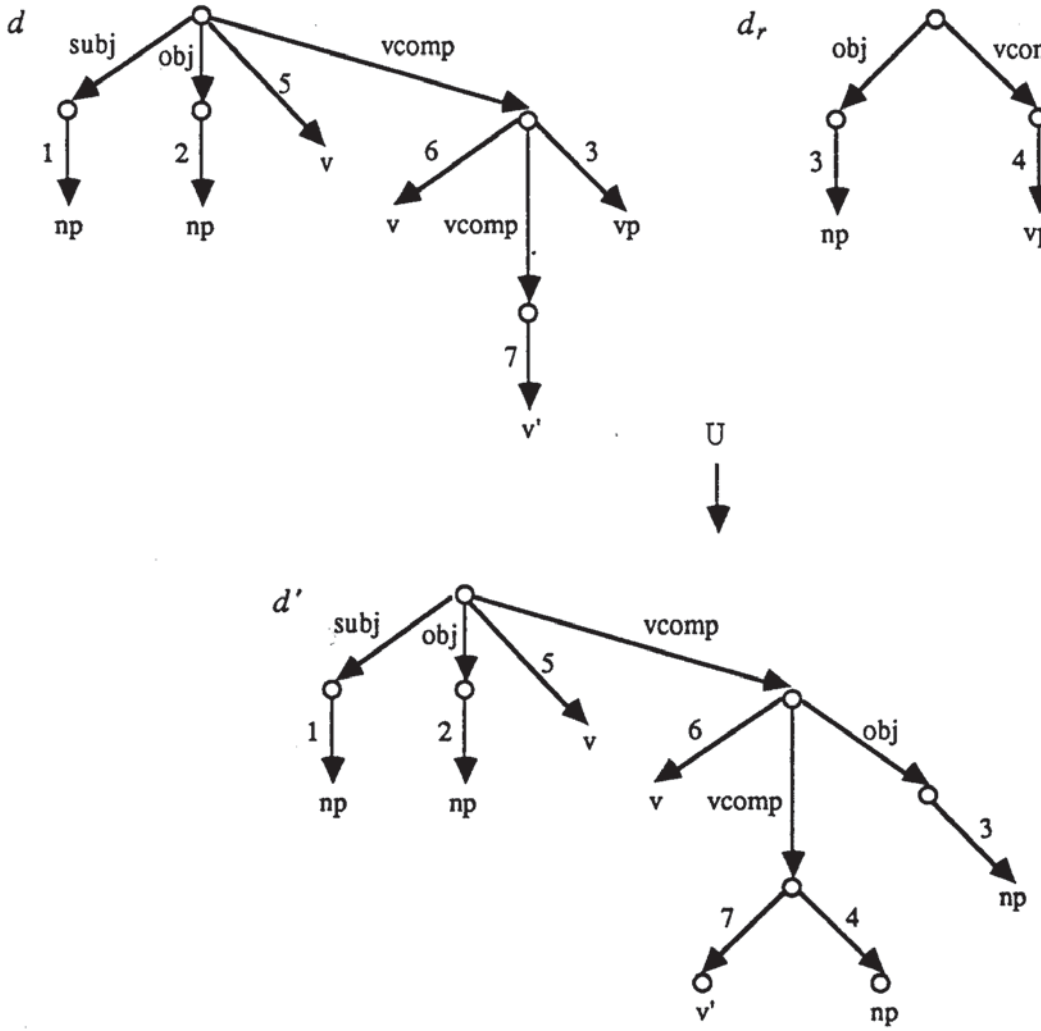


Figure 6.3.c Example of Wedekind's Monostratal Unification

Briefly, the edge to which the rule is applied is deleted (*<3, vp>*) and all edges with indexes greater than the index of this edge are incremented by the number of new nodes introduced by the rule (the length of the rule's right-hand side). A new edge is then created and substituted for the deleted edge. The new edge is created by the unification of d_r with the sub-DAG of *d* to which the deleted edge was attached (this is illustrated in Figure 6.3.c). Unification will not fail due to the additional edges as all indexes are unique and thus the values of the indexes are not unified themselves.

Wedekind only describes in detail his derivation of the monostratal representation. Other parts of LFG are not described although he states that long distance dependencies are dealt with at the F-structure level. The actual implementation of the theory is not described, so that there can be no discussion of the unification method or other aspects. It does seem that the basic theory relies on a top-down parsing method being employed, as terminals of the DAG representation are expanded. This theory has been implemented and used as the basis of a system for generating phrases from F-structures [Momma, 1987]. Generation is a useful method of testing grammars and lexicon for overgeneration during their development.

The general trends in the implementation of LFG appear currently to be : away from DCG TD control toward BU or flexible control stratigies supported by Chart based data structures, and away from the original method of producing an F-structure from an F-description after parsing [Kaplan and Bresnan, 1982] toward an incremental (or even combined) building of F-structure in parallel with C-structure so that constraints in F-structure can be used to guide parsing.

Chapter 7

Implementation of the Interface System

Several different implementation techniques were employed in an attempt to produce a complete and yet efficient implementation of the LFG formalism. The implementation should primarily :

- support fully the LFG formalism and notation as described in Kaplan and Bresnan [1982].
- illustrate that the system could produce reasonable/practical response times in executing natural language queries to a database.

Perhaps the most important characteristic of the desired implementation is that it should provide an environment as close to the LFG formalism (notationally) as possible. This is a highly desirable approach given that LFG is a high-level linguistic descriptive formalism intended for the use of linguistics, not computer programmers. The linguistic felicity and efficiency of the LFG implementation are determined to the greatest extent by the parsing algorithm employed. Tests with various parsing strategies were made.

An initial implementation was based on the DCG type method, which was soon replaced by a pseudo-DCG type implementation in order to facilitate automatic compilation of rules from an LFG-like notation into Prolog rules. The DCG type approach proved efficient at parsing small grammars with small degrees of rule and lexical ambiguity, but was abandoned as it proved extremely difficult to compile an LFG-like notation into DCG type rules. This transformation is easily possible for simple LFGs but is much more difficult when complex equation sets are used. In particular the building of F-structures by direct Prolog variable unification is very difficult to arrange. An F-structure may be passed around (through rules) as a single variable (input and output variables) but suppose an equation is used which imposes a constraint on the subj function's number feature. Then the rule must match the input F-structure's subj function and match this function so as to expose the number feature for checking. The basic problem is that each rule and lexical entry must 'know' about the structure built by others during parsing.and cannot thus operate in a modular fashion.

The pseudo-DCG implementation was then augmented, according to the method described in Matsumoto and Tanaka [1983] and Matsumoto, Kiyono and Tanaka [1985]. Parsing is then performed in a bottom-up manner and partial parses, whether failed or correct, are saved in the database so that they do not have to be redone on backtracking. The parsing in this implementation proved extremely difficult to follow so that grammar debugging was also extremely difficult. In addition to this, it proved quite difficult to support other LFG mechanisms such as sets and the Kleene-star.

The final implementation is based on a specialization of the active chart parser [Winograd, 1983; Varile, 1983] here called Word Incorporation (WI) [Phillips, 1986]. An outline of the system components is given in Figure 7.a.

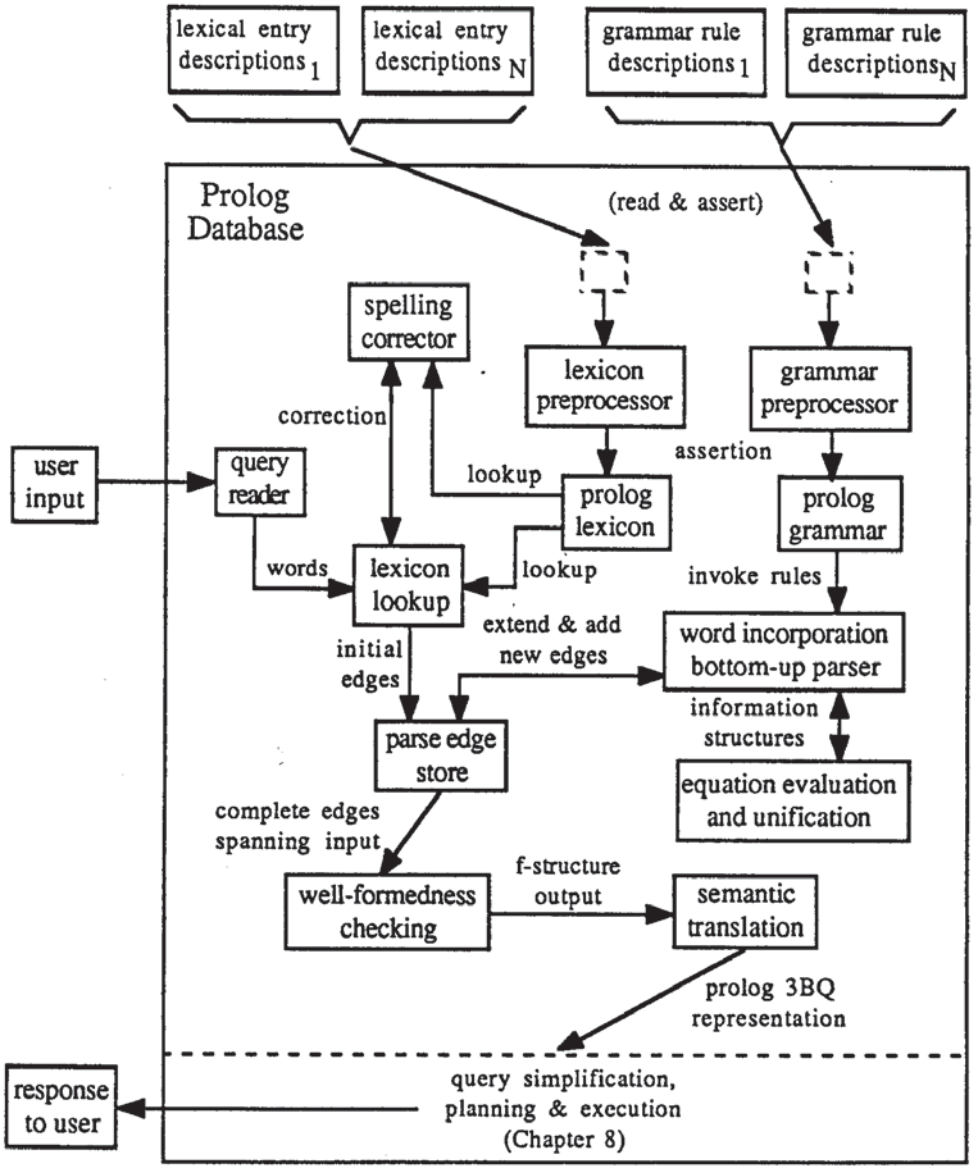


Figure 7.a Interface System Outline

Once the system has been loaded into the Prolog database a top-level menu is presented to the user. To produce an interface for a particular domain database it is necessary to first introduce a grammar, a set of lexical entries and a specification of predicates in the domain database (see Chapter 8). The top-level menu offers options for all of these and provides the necessary prompts for input file names as well as offering the option to return to the top-level menu at each stage. Other options in the top-level menu allow one of the twenty-three queries in the test corpus (Appendix D) to be selected for parsing, a query to be typed into the system, tracing to be turned on or off (traces from the test corpus are given in Appendix F) and the user to quit the interface system returning to the top-level of the Prolog interpreter. The parser will be described in detail in Section 7.3 but first the representation of F-structures and the form of input grammar and lexicon specifications is described.

7.1 F-structure Representation and Unification

The heart of any system based on a unification grammar will be the actual implementation of unification in the chosen programming language. The system described here is completely implemented in Quintus Prolog [Quintus, 1987].

During parsing F-structures are unified (combined) as described earlier. There are obviously a number of different ways to realize unification but in a 'parallel' parser such as the active chart, where edges are combined and a new and totally separate edge is produced, it is appropriate for the unification algorithm to produce a new F-structure F_3 from two component F-structures F_1 and F_2, without changing the values of either F_1 or F_2. Then either F_1 and F_2 may be combined with other F-structures later. The use of open-ended lists for example, is thus quite inappropriate as it is undesirable to change the value of either of the components of F_3 (F_1 and F_2).

F-structures are represented as normal ordered Prolog lists with elements which represent the basic '*<attribute> <value>*' pairs. The list is ordered on *<attribute>* names using Prolog's standard order. The various different types of feature specifications (values and constraints) are represented themselves as Prolog terms (with Prolog operators shown in bold type) :

- simple features with values (*'num sg'*) are represented as :

  <feature> = **atom** <value> eg num = atom sg

where *<value>* is the simple atomic value of *<feature>*.

- existential constraints on feature values as :

  <feature> = exists eg num = exists

- negative existential constraints on feature values :

  <feature> = none eg num = none

- value constraints on feature values :

  <feature> = **val_c** <value> eg num = val_c sg

- negative value constraints on feature values :

  <feature> = **neg_c** <values> eg person = neg_c [first, second]

  where *<values>* is a list of *<value>* which *<feature>* may not take (person may not be first or second).

Certain combinations of constraints are also allowable which will be produced by multiple constraint equations applied to a single feature in grammar or lexicon, or by constraint unification during parsing. These are also represented as Prolog terms produced by combining the component constraint terms. The allowable constraint combinations are :

- an existential constraint and value constraint :

  <feature> = exists
  <feature> = val_c <value>
  U ⟶ <feature> = exists val_c <value>

- an existential constraint and negative value constraint :

  <feature> = exists
  <feature> = neg_c <values>
  U ⟶ <feature> = exists neg_c <values>

Other unifications of constraints with constraints, and constraints with values are allowable but do not produce more complex Prolog terms. The following examples illustrate how constraints are handled by unification :

- <feature> = exists U <feature> = exists → <feature> = exists
- <feature> = none U <feature> = none → <feature> = none

- <feature> = none U <feature> = exists → fails
- <feature> = val_c <value>
  U → <feature> = val_c <value>
  <feature> = neg_c <values>

  unification is conditional on *<value>* not being a member of the list *<values>*.

- <feature> = neg_c <values$_1$>
  U → <feature> = neg_c <values$_3$>
  <feature> = neg_c <values$_2$>

  unification adds any values not in *<values$_1$>* to *<values$_2$>* to produce the new list *<values$_3$>* which is the list of values *<feature>* may not take (for simplicity *<values$_1$>* is actually appended to *<values$_2$>* to produce *<values$_3$>*).

In this way, a feature can only have a single entry in an F-structure list which may represent one or more actual LFG feature specifications. This allows the F-structure list to be simply ordered on attribute names. Functions are given values which are themselves F-structure lists. A function is represented by the Prolog term :

<function> = fs <f-structure list>

where *fs* (F-structure) is a Prolog operator prefixing a subsidiary F-structure.

As well as the F-structure list described above, additional information is attached to an F-structure to support indexing, functional control, bounding metavariables, the additional semantic components described in Chapter 3, and completeness and coherence checking. An F-structure list is thus just part of a much larger "information structure" designed to support unification and carry the additional information required for later semantic interpretation. An information structure takes the form :

<index> **ind** <slot_description> ^
<quantifier> ^ <sem_pred> ^ <variable> ^
<f_structure> ^ <controllees> **glob** <pointers>

where Prolog operators are again shown bold type. The components of this information structure are to be outlined now. The *<index>* component is an integer used to co-index

information structures involved in constituent control. If an information structure is not involved in constituent control this will have a value '[ ]'.

The *<slot_description>* component is a Prolog term :

<slot_type> **stype** <slots>

where *<slots>* is an ordered list of either subcategorizable functions introduced into the F-structure by grammar equations and lexical entries or those functions subcategorized by the semantic form in the F-structure.

As an F-structure is built up, subcategorizable functions may be introduced into the F-structure by grammar equations '(↑ subj) = ↓' or their existence implied by lexical equations '(↑ subj num) $=_c$ pl', these functions are listed in *<slots>*. Until the semantic form (if a semantic form is to be introduced at all) is found, the conditions of completeness and coherence cannot be applied. Before a semantic form is unified with an F-structure, subcategorizable functions introduced or implied are added to the list *<slots>* and the value of *<slot_type>* will be *part* (partial slots). When a semantic form is added to the F-structure all the functions listed in *<slots>* must be subcategorized by the semantic form (listed in the *<slots>* of the semantic form's information structure). The semantic form's list of functions is then used in the unification result. The value of *<slot_type>* in the case of information structures containing semantic forms is *full*, indicating that no additional functions can be added to those in the *<slots>* list. In addition to function names, the *<slots>* list also carries the function's domain types. The *<slots>* list thus takes the form :

$$[<f_1> : <t_1>, <f_2> : <t_2>, \ldots . , <f_n> : <t_n>]$$

where f_x is a subcategorized function name and t_x the domain type of the function. In the case of proper nouns (England, John), no functions are subcategorized and the *<slots>* list is abbreviated to a term :

<predicate> : <type> (england : country, john : human)

The *<quantifier>* component is a skeleton (three-branched quantifier type) semantic representation of the quantifier within the F-structure which has the value '[ ]' if no quantifier is present. The *<sem_pred>* component is the semantic predicate used in semantic translation. This may contain functional variables and a quantifier variable as arguments.

The *<variable>* component is a variable which may become bound to a quantifier in the information structure. If the information structure serves as some function value, then this variable will be the variable associated with that function (see Chapter 3). This variable is also given a domain type. If this information structure is the value of a function *F*, then the type must be compatible with the corresponding type of *F* found in the *<slots>* list of the enclosing F-structure. The *<f_structure>* is the ordered F-structure described previously.

The *<controllees>* component is a list of controllees which have yet to be matched with corresponding controllers. Each controllee takes the form :

controllee(<subscript>, <info_structure>)

where *<subscript>* is the controllee's subscript and *<info_structure>* the controllee's information structure, which has no controllers or controllees itself.

The *<ptrs>* component is a list of pointer values. This list is connected with the use of functional control. Two or more functions involved in functional control share the same information structure value. This value is stored in the pointer value list, where a pointer is simply an integer value (*N*) assigned at parse time. Entries on the pointer list have the form :

N = fs <info_structure>

and the corresponding functions which have this value are represented in the ordered *<f_structure>* list by entries 'F = fs ptr N'. The use of pointers is illustrated in Figure 7.1.

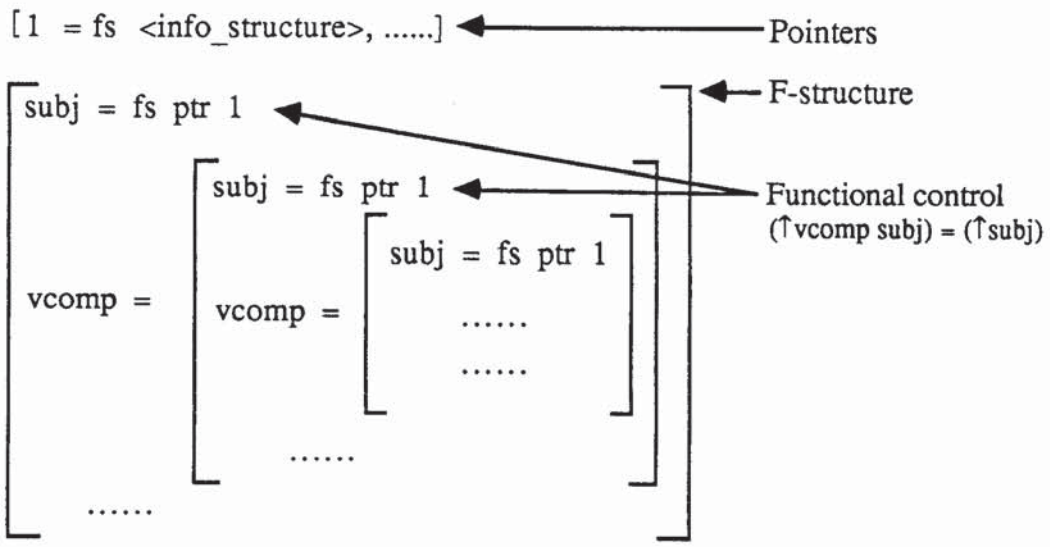


Figure 7.1 Outline of Pointers used for Functional Control

Pointers are only present at the top-level of the information structure, as they are global values which may occur at any and possibly several levels of the F-structure itself. This representation thus overcomes the problem of shared F-structure values by factoring out the shared value and representing this as a single value referenced from multiple occurrences by pointers. The controller and controllee lists are also global in that they are present only at the top-level of an information structure. The value of a function within an information structure's F-structure component thus only has an index, slot description, quantifier, variable, semantic predicate and subsidiary F-structure.

Unification is now defined to act on information structures rather than on just the simple F-structure lists. The unification of two information structures proceeds in the following manner. At the top-level of the structures, the global components (pointers, controllers and controllees) are separated from the rest of the information structures being combined and passed to the unification procedure through which, as global values, they are to be passed. Unification thus acts on the remainder of the information structures with occasional reference to the global components, as will be explained later. Functions themselves, within the F-structure component, are also represented as information structures without the global controller, controllee and pointer components. Discussion of the components involved in long distance dependencies (indexes, controllees and controllers) will be delayed until after the discussion of grammar handling. The basic unification of information structures (slot description, quantifier, semantic predicate, variable and F-structure components) can however be discussed now.

The slot description plays a central role in the incremental construction of F-structure during parsing. It lists the subcategorizable functions either included in the F-structure or included by implication. The unification of two slot descriptions succeeds only under the following circumstances :

- both slot descriptions are prefixed as type *part* (the corresponding F-structure components contain no semantic forms) in which case the functions missing from one slot list are added to the other slot list to produce the unification result. In addition to this, functions appearing in both slot lists must have compatible domain types, for example :

  part stype [subj : human, obj : book]
  U
  part stype [subj : man, vcomp : human]
  ↓
  part stype [subj : man, obj : book, vcomp : human])

- one slot is defined as type *full* (the corresponding F-structure component contains a semantic form) and the other as type *part*. In this case, all of the functions in the *part* slots list must be present in the *full* slots and in addition, the domain types of the duplicate slots must be compatible, for example :

  - full stype [subj : human, obj : book]
    U
    part stype [subj : man]
    ↓
    part stype [subj : man, obj : book]

  - full stype [subj : human, obj : book]
    U
    part stype [subj : man, vcomp : human]
    ↓
    fails

- both slot lists are of type *full*, in which case both slot lists must contain the same functions which all have compatible domain types :

  full stype [subj : human, obj : book]
  U
  full stype [subj : man, obj : object]
  ↓
  full stype [subj : human, obj : book]

Quantifiers are more simply dealt with as by the uniqueness condition only one F-structure involved in unification can have a quantifier (determiner), the other quantifier will have a value of '[ ]'. Unification thus produces a quantifier if either component information structure holds a quantifier, and results in a value '[ ]' if neither information structure contains a quantifier. The uniqueness condition similarly applies to the semantic predicate component so that these are dealt with in the same way as quantifiers. The variable component consists of a Prolog variable and domain type. Unification succeeds if the domain types are compatible and also unifies the variables (Prolog unification) so that the variables become a single variable. This unification changes in some way the component information structures. As a single information structure may be unified during parsing, as will be explained shortly, with several other information structures, all of the resulting information structures will have the same variables. This similarity is not however maintained as after unification, information structures are stored separately in the Prolog database and the identity of these variables is not maintained across information structures.

The F-structure ordered lists are unified by a recursive procedure which examines the first attribute entry on each list and produces a new attribute entry on the output list. This procedure receives the global pointers from both information structures as a single list which is used to hold values of functions involved in functional control. In outline, the procedure (*new_fs/5* in Appendix G) takes the form:

```
new_fs([], Fs, Ptr, Fs, Ptr) :- !.
new_fs(Fs, [], Ptr, Fs, Ptr) :- !.
new_fs([Att = Val|R], [Att = Val1|R1], Ptr, [Att = Val2|R2], Ptr1) :- !,
        { unify values : Val, Val1 & Ptrs to produce Val2 & Ptrs2 (see below) },
        new_fs(R, R1, Ptrs2, R2, Ptrs1).
new_fs([Att = Val|R], [Att1 = Val1|R1], Ptr, [Att2 = Val2|R2], Ptr1) :-
        Att @< Att1 ->
           (Att2, Val2) = (Att, Val),
           new_fs(R, [Att1 = Val1|R1], Ptr, R2, Ptr1)
           |
           (Att2, Val2) = (Att1, Val1),
           new_fs([Att = Val|R], R1, Ptr, R2, Ptr1).
```

The first two rules simply add the elements of a F-structure to an empty F-structure. The third rule unifies the values of a single attribute appearing in both F-structures to produce an output value. There are many different actual cases of this rule, each case performs a unification of one of the possible combinations of values and constraints a single attribute

can have. The last rule simply maintains the ordering of F-structure lists when the next attribute in the component lists is different.

As well as simple atomic values and constraints, an attribute may be a function in which case the value will itself be an information structure or the attribute may be a function involved in functional control in which case it will have a pointer value (integer). If the value is itself an information structure then a recursive call to the information structure unifying procedure is made. If the value is a pointer then the corresponding value of the pointer is found in the pointer list and used as if the value were a function value in the F-structure. The new pointer value is then simply put back into the pointer list. If two pointer values are unified then they are both given the same pointer number in the output F-structure and the two corresponding pointer values are unified and put back into the pointer list under the new pointer number.

7.2 Input Grammar

The LFG grammar used by the system is expressed directly in an LFG type notation where syntactic alterations have only been made to facilitate the symbols available on most keyboards and the legal syntax of Prolog. By defining suitable operators in Prolog, the programming language syntax itself can be extended to encompass that of this LFG notation. A rule is thus made a Prolog term and the syntax checking procedures of the Prolog environment itself can be applied to the rules simply by treating a grammar as a normal Prolog program (*consult/compile*). A simple rule such as :

s	⟶	np	vp
		(↑ subj) = ↓	↑ = ↓

is written in the Prolog notation as :

```
s      --->    np  eqns (up subj) = down ,
               vp  eqns up = down .
```

where the necessary Prolog operators for this particular rule are shown in bold type. A rule has the format 'LHS ---> RHS'. The LHS is a simple Prolog symbol (grammatical category) and the RHS a series of categories, annotated with equations, to which the RHS rewrites. Each RHS category is followed by the operator *eqns* which introduces the equations attached to the category in the LFG rule. There may be one or more equations

attached to a category in a rule and zero or more attached to a literal (*that*, *'s*). Equations are written as a sequence with the operator & placed between individual equations (see rule example below). Immediate dominance variables, '↑' and '↓' are written as *up* and *down* respectively as the LFG arrows are not generally available on computer keyboards. A more complex rule involving controllers such as :

$$\begin{array}{lcll} s' & \longrightarrow & np & \boxed{s} \\ & & (\uparrow q) = \Downarrow^{np}_{[+wh]} & \uparrow = \downarrow \\ & & (\uparrow \text{focus}) = \downarrow & \\ & & \downarrow = \Downarrow^{s}_{np} & \end{array}$$

is expressed in this notation as :

```
s1   --->     np  eqns  (up q) = controller super np sub [+wh] &
                       (up focus) = down &
                       down = controller super s sub np ,
          bnd  s   eqns  up = down .
```

Again, the symbols which are declared as operators are indicated in bold type. Controllers are written as a sequence *'controller super <superscript> sub <subscript>'* and controllees, which do not have a superscript; as *'controllee sub <subscript>'*. A bounding node is denoted by prefixing the grammatical category with the operator *bnd*. The notation of the simple equation type examples, listed in Figure 2.1.b is :

$\uparrow = \downarrow$	`up = down`
$(\uparrow \text{subj}) = \downarrow$	`(up subj) = down`
$(\uparrow \text{tense})$	`(up tense)`
$\neg(\uparrow \text{tense})$	`not (up tense)`
$(\uparrow \text{numb}) =_c \text{plur}$	`(up numb) c plur`
$\neg(\uparrow \text{numb}) =_c \text{plur}$	`not (up numb) c plur`
$(\uparrow(\downarrow \text{pcase})) = \downarrow$	`(up(down pcase)) = down`

The grammar notation also includes versions of the Kleene-star operator (*) and the set inclusion operator (*set_val_of*), for example :

$$\begin{array}{lcccc} np & \longrightarrow & det & adj\,* & n \\ & & \uparrow = \downarrow & \downarrow \in (\uparrow \text{adj}) & \uparrow = \downarrow \end{array}$$

```
np   --->   det    eqns       up = down ,
            adj * eqns       down set_val_of (up adj) ,
            n      eqns       up = down .
```

As noted by Kaplan and Bresnan, natural language does not typically contain unlimited repetitions of constituents. For this reason, an additional version of the Kleene-star operator is made available in the notation. This takes the form '*<cat> *N*' where *N* is the maximum number of repetitions of grammatical category *<cat>* which may occur, for example, the rule :

```
np   --->   det      eqns up = down ,
            adj * 2  eqns down set_val_of (up adj) ,
            n        eqns up = down .
```

can be used in parsing either :

'the largest blue ball' or 'the largest ball'

but not :

'the largest bouncy blue ball'

if *the* is of category *det* and *ball* of category *n*. The optionality allowable in LFG rules, signified by the use of parentheses (Appendix A), can be described in the notation here by using '*1' (one or zero occurrences).

The notation also includes a special operator which is used to specify the linking equation ($\Uparrow = \Downarrow$). There seems to be no motivation for using a linking equation other than in conjunction with a bounding node. For this reason, the operator used here to specify a linking equation also implies a bounding node. The operator *lnk* is used to specify that a grammatical category in a grammar rule is a bounding node with a linking equation attached :

s' ⟶ (that) $\boxed{s}$ $\uparrow = \downarrow$ $\Uparrow = \Downarrow^{s}$

```
s1   --->          that ,
                lnk  s     eqns  up = down .
```

Disjunction is the only part of the Kaplan and Bresnan [1982] description not allowable in the grammar notation. Disjunction could be implemented in principle, but is perhaps undesirable as disjunctions are very difficult to parse efficiently and have a non-declarative nature. This restriction does not reduce the descriptive power of the notation but does lead to a less compact linguistic description. The only usage of disjunction which is difficult to describe is that coupled with use of the Kleene-star operator. As mentioned in Chapter 2 however, large repetitions of constructions (and disjunctions) do not naturally occur in natural language so that in practice, the omission of disjunction does not lead to large numbers of grammar rules. Rather, this restriction constrains the grammar writer to just those repetitions which will occur, where these must be stated as separate rules representing alternative expansions (disjunctions).

A grammar is first added to the Prolog database (Figure 7.a) by a simple procedure which opens the file named by the grammar writer and reads each rule in the file asserting it into the database. This is the same action accomplished by the Prolog primitive *consult/1* procedure but this produces untidy output reports about files consulted and their code size. Each rule forms a Prolog term (which must be terminated by a full stop) and can thus be unified with a single Prolog variable by a single call to the primitive *read/2* (stream, term) procedure. The Prolog interpreter will report the position of syntactic errors in rules when they are read from the input file according to the operator definitions which define the grammar notation's syntax. A complete grammar may be defined across any number of input files (Figure 7.a). All grammar files must be loaded before the rules can be pre-processed as rules with common first RHS category or literal are stored together to support the parser's operation. This will be fully described in Section 7.4.

Grammar pre-processing transforms the rules into a form more easily handled in Prolog and also performs a certain amount of correctness checking on each of the rules. Each grammar rule is retracted from the Prolog database and subjected to the following transformations and checks :

- simple equations referring to features (mainly constraints on features) are converted into information structures so that equation evaluation is realized simply by unifying the information structure with that referred to by the equation's meta-variable ('↑', '↓').

- equations referring to functions are checked to see if the named function is a designator (subcategorizable function),. If this is so, the function name is prefixed with the operator *d* :

```
(up subj) = down   →   (up d subj) = down
```

- the domain roots of controllers are located and the controllers are moved to these. This may require splitting an equation into two parts :

(a) s ⟶ np s1
$\downarrow = \Downarrow_{np}^{s1}$

(b) s ⟶ np s1
$\downarrow = Fs_a$ $Fs_a = \Downarrow_{np}^{s1}$

The equation in rule *(a)* is thus split into two separate equations illustrated in rule *(b)*. The original equation in rule *(a)* equates the F-structure produced from below the *np* to that of a controller with domain root *s1*. The equations in rule *(b)* achieve the same result, the F-structure from below the *np* is equated with a variable Fs_a, which is also equated to that of the controller. The reason for doing this is connected with the operation of the parser and will be explained in Section 7.4. At the same time as controllers are moved to their domain roots, the existence of domain roots can be ensured and it is also possible to check that a grammatical category does not serve as domain root for more than one controller and that there is only one possible domain root for a controller.

- equation paths are checked to ensure that they are not longer than two elements and do not vary from the legal syntax of the equation notation.

After these transformations have been made, the grammar rules are asserted into the parser's Prolog code module. All rules which have a common first RHS grammatical category or literal are stored together. This is done to support the parsing algorithm's operation which is described in Section 7.4.2. The parser thus receives grammar rules in the form :

(a) <cat> rules_which_cat(<starts>, <completes>).

(b) <word> rules_which_word(<starts>, <completes>).

In case *(a)*, *<cat>* is a grammatical category which is the first RHS category of each rule in the list of rules *<starts>* and the only category in the RHS of each rule in the list of rules *<completes>*. In case *(b)*, the literal (a word or other literal in grammar rules such as *'s*) *<word>* is the first RHS component of each rule in the list of rules *<starts>* and the only component in each rule in the list of rules *<completes>*.

Each grammar rule in the lists of rules represents a single grammar rule which has been subjected to the basic transformations described above. For example, a rule (with a number of equations to demonstrate their treatment):

```
s   --->     np  eqns (up subj) = down &
                      (up num) c pl &
                      down = controller super vp sub np ,
          bnd vp  eqns up = down &
                      (down tense) .
```

is transformed into a list of the form:

```
[s, (      np if (up d subj) = down &
                 up [] ind part stype []^[]^_^[num = val_c pl] &
                 down fs_is Fs , .
     bound vp if up = down &
                 down [] ind part stype []^[]^_^[tense = exists] &
                 controller(np, Fs) ) ] .
```

This rule would be added as an element to the list of rules indexed by category *np*, the first category of the RHS.

In addition to the transformations described, the grammar pre-processor adds literals which it finds in grammar rules to a temporary storage area in the lexicon pre-processor. These words will then be added to the system's final dictionary as if they were defined in an input lexicon. For this reason, grammar files should be pre-processed before lexicon files. Literals in grammar rules are currently not allowed to have equations attached and, although only minor modifications would be necessary to allow this, it would then be necessary to include an additional operator in the notation to differentiate literals from grammatical categories. Equations which would be attached to literals can however simply be placed on sister C-structure nodes as these can only refer to the F-structure above ('↑'), not that below ('↓'), the literal itself.

7.3 Input Lexicon

Lexical specifications, like grammar rules, are specified in a notation supported by Prolog operators. Each word can have only one lexical entry but this may specify alternative feature sets for a single word. A lexical entry has the general form shown in Figure 7.3.

```
<word> ~
                    <cat_a>    eqns      <features_a1>
                               or        <features_a2>
                               or            .....
                               or        <features_aN>
               and  <cat_b>    eqns      <features_b1>
                               or        <features_b2>
                               .....
                               or        <features_bN>
               and  .....
               and  <cat_N>    eqns      <features_N1>
                               or        <features_N2>
                               or            .....
                               or        <features_NN> .
```

Figure 7.3 General Form of Lexical Entry

where *<word>* is an atomic Prolog value (the word enclosed in single quotes if it contains a space), *<cat$_x$>* is a grammatical category and *<features$_x$>* is a sequence of equations associated with a specific definition of *<word>* with grammatical category *<cat$_x$>*. A Prolog operator *or* is defined to separate alternative equation sequences which define a word with a single grammatical category, and the operator *and* is defined to separate alternative definitions of a word with different grammatical categories. For example, *population* might have a lexical specification with two different semantic forms :

```
population   ~
             n  eqns  (up pred) = population >> [(up of-obj)] &
                      (up sem) = population(of-obj, quant) &
                      (up num) = sg &
                      (up of-obj-domain) = place &
                      (up quant-domain) = number
                or    (up pred) = population >> [(up poss)] &
                      (up sem) = population(poss, quant) &
                      (up num) = sg &
                      (up poss-domain) = place &
                      (up quant-domain) = number .
```

Both these entries of *population* have grammatical category *n* and also illustrate the lexical notation of equation sequences. Equation sequences take the same basic form as grammar

equation sequences, where equations are separated by the operator *&*. The notation of assignment equations, semantic forms and the sem feature, used in semantic translation, is also illustrated. Path names are specified using the operator '-', for example '(↑ subj-num) = pl'.

The principle of locality (ie path name length greater than two) appears to be broken in the lexical entry for *population* by the use of the path name 'of-obj-domain'. In fact it is not possible to adhere to the locality principle when specifying values or constraints on any such case marked functions ('of-obj', 'to-obj'). The requirement to do this does not seem to arise on a syntactic basis, so that here departure from the locality principle is only allowed when specifying domain features (types) on case marked obj functions.

The use of the operator *and*, which is used when a word has two or more entries with differing grammatical categories, can be illustrated with another example (*which* can be a determiner *det* or a relative pronoun *rel*) :

```
which   ~
                  det   eqns  (up det) = wh &
                              (up wh) = +
                              up = controllee sub [+wh]
            and
                  rel   eqns  (up rel) = + .
```

Lexical entries are treated in much the same way as grammar rules. Any number of files containing lexical specifications can be loaded into the Prolog database. These are read from a file named by the lexicon writer, subjected to a number of transformations, and then asserted into the dictionary accessing Prolog module. Each individual definition of a word is transformed into an information structure so that, during parsing, lexical equations do not have to be evaluated but can simply be unified as described earlier. The lexical entries shown above are represented in the Prolog dictionary :

```
dict(not_in_grammar, population,
     [112, 111, 112, 117, 108, 97, 116, 105, 111, 110],
     [   [n, [] ind full stype
             [of-obj = Var:place]^[]^
             population(Var, Var1)^(Var1:number)^
             [num = atom sg, pred = atom population(of-obj)]^[]],
         [n, [] ind full stype
             [poss = Var:place]^[]^
             population(Var, Var1)^(Var1:number)^
             [num = atom sg, pred  = atom population(poss)]^[]] ]
```

```
dict(not_in_grammar, which,
     [119, 104, 105, 99, 104],
     [   [det, [] ind part stype []^wh(Var, _)^[]^(Var:_)^
             [det = atom wh]^[controllee([+wh], [det = atom wh]],
         [rel, [] ind part stype []^[]^[]^_^[rel = atom +]^[]] ]
```

The format of each of these dictionary entry is :

dict(<gram_flag>, <word>, <chars>, <entry_list>).

where <gram_flag> has the value *not_in_grammar* if the word defined by this entry does not occur in the grammar and the value *grammar* if the word occurs in a grammar rule; *<word>* is the word or literal defined by this dictionary entry; *<chars>* is a list of ASCII numbers which correspond to the characters of the word and *<entry_list>* is a list of the entries for the dictionary word.

The literals found in grammar rules are also incorporated into the dictionary. As lexicon entries are processed, this set of literals is checked to see if the word also occurs in the grammar rules. If this is so, an additional entry with category *word* is added to those for the word in the dictionary entry. After processing all lexical entries the remaining literals are given dictionary entries with a single *word* type definition.

7.4 The Parser

The characteristics of Top-Down (TD) and Bottom-Up (BU) parsing were outlined in Chapter 6, where a number of different LFG implementations using these parsing strategies were described. The most recent of these which has been implemented [Block & Hunze, 1986] is based on the Earley algorithm. This algorithm does not constrain the type of grammar rules which can be used (left-recursive rules can, for example, be used), as TD control does, and can support a variety of parsing strategies. However, the overheads of carrying and processing information to support this flexibility is very high. The Earley algorithm also does not seem suitable for LFG as TD expansion is only weakly constrained in LFG by the CFG. The prediction part of the algorithm thus seems to be a distinct weakness with a view to LFG. The BU combining component on the other hand, seems to naturally fit the requirements for parsing LFG as a great deal of information (values and constraints) is held in the lexicon which should be used early on in parsing to kill-off incorrect parses.

An algorithm has therefore been adopted (based on Phillips [1986]) which has low computational overheads and has a basic BU action, here called "Word Incorporation" (WI). This algorithm is a specialized version of the the Earley algorithm (also using a Chart like data structure, which can be used as the basis of a variety of parsing algorithms including Earley). The Earley algorithm itself can be modified to support a variety of parsing strategies, but WI represents a greater commitment to a BU methodical action than a BU variation of the Earley algorithm.

7.4.1 The Active Chart Parser

The Chart is a graph data structure which can support a range of parsing algorithms (for example, TD or BU and breadth or depth first). The Chart acts as book-keeping storage for all the information produced during parsing. Graph nodes are called "vertices" which can be linked by arcs called "edges". Each edge represents a grammar rule (or dictionary entry) and contains a minimum of : starting and ending vertices numbers, a category (rule LHS category) and a remainder (initially the rule RHS, or nil for a dictionary item). Dictionary items (grammar terminals) may be viewed as rules with an empty RHS, and a LHS equal to the item's dictionary category. Edges may carry any required additional information to produce the required output. In an implementation of LFG, edges may also carry the (partial) F-structure of the phrase parsed so far.

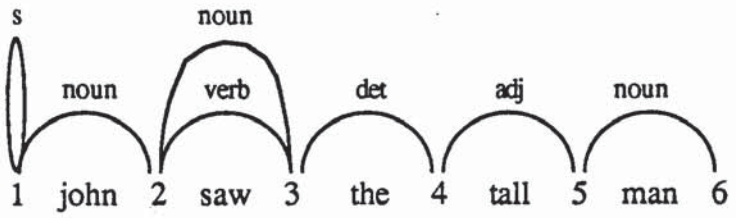


Figure 7.4.1.a Initial Chart State (Base)

Before parsing actually starts, an initialisation step (Figure 7.4.1.a) creates the initial edges and vertices. Each input word is assigned a starting and ending vertex number and forms a complete edge (having no remainder) in the initial "base" of the Chart. A single word in the input may generate a number of initial complete edges, one for each dictionary entry of the word. An active edge representing each grammar rule with LHS which is the distinguished grammar category is proposed at (that is having starting and ending vertices equal to) the first vertex (the vertex with category *s* in Figure

7.4.1.a). These active (incomplete) edges are placed on the list of "pending" edges to be later entered onto the Chart, that is the edges are "proposed".

The Chart algorithm selects a pending edge and enters this onto the Chart. This process has two distinct phases, combining the edge with other edges on the Chart and using the edge to invoke more grammar rules. Each of these activities may create new active or complete edges which are added to the pending edges.

A complete edge which is entered onto the Chart may extend (match) an active edge on the Chart (BU following successful TD expansion), if the active edge's end is equal to the complete edge's start, and the complete edge's category is the first part (or all of) the active edge's remainder. An active edge which is entered may be extended (BU) by a complete edge on the Chart if these edges match in the same way. If the complete edge matches the last part of the active edge's remainder then a new complete edge is created; if some further remainder is left then a new active edge is created. The new edge created (active or complete) spans the vertices of both its component edges and is put on the pending list of edges.

An edge, with category *C*, invokes (TD) new rules (used to produce corresponding edges) at its starting vertex if there is a grammar rule that has the same category *C* as its LHS. Before any edge is added to the pending list, the Chart must be searched to ensure that the edge has not been proposed before (thus preventing an infinite recursion of TD expansions). This requires searching all edges on the Chart (active and complete) and all of the pending edges, for an edge with the same category and starting vertex (and the same informational content, if other information is carried by edges). The complete algorithm iterates until some complete edge with the required category and/or starting and ending vertices is produced.

The Chart contains common goals (substrings) of different parses so that these need only be produced once. These can either be asserted into the Prolog database or can be carried through the algorithm's cycles as Prolog arguments in the algorithm's Prolog rules. The algorithm is obviously cyclic, not a Prolog-like (TD) search, for no backtracking will occur at all. The algorithm can be varied by changing the selection criteria of an edge from the pending list to be entered onto the Chart. For example, if edges with highest vertices (ending) are selected, the different parses will tend to be

developed in series. By selecting those with lowest ending vertices the different parses can be developed in parallel. Selecting completed edges promotes BU combination whilst selecting those only with a remainder tends to result in TD expansion. Whatever the selection criteria, an attempt is made to use each edge both for BU combination and TD expansion.

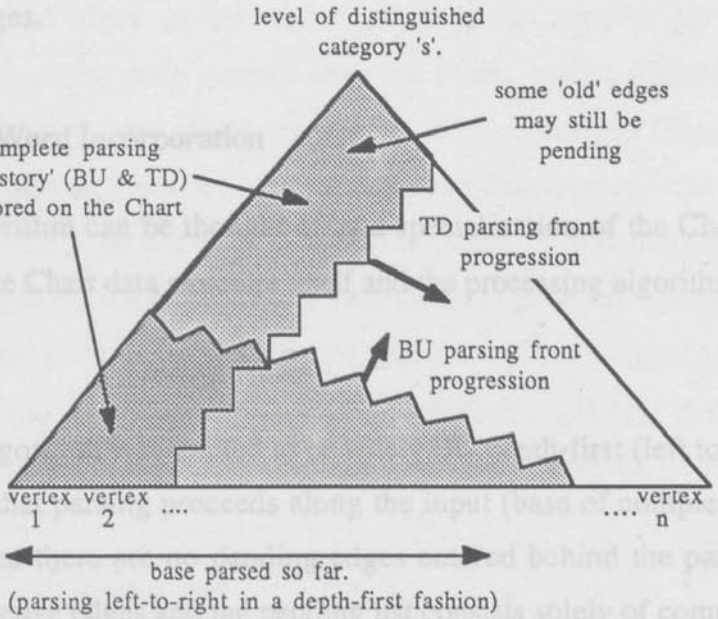


Figure 7.4.1.b Outline of Chart Parsing Operation

The main weakness of the Chart algorithm is its low efficiency, mainly caused by a high overhead when proposing new edges (checking an edge has not previously been proposed). It is a simple matter to index edges by their starting or ending vertex so that the outgoing or incoming set of edges to a vertex ("edgeset") can be quickly accessed, but the Chart algorithm requires access to Chart edges by start and end, and pending edges by their start. This edge check cannot be efficiently supported in Prolog. The check must be performed on all the edges produced (TD or BU). As the Chart carries a complete parsing history (Figure 7.4.1.b) the number of edges to be checked may be very large. The Chart is thus also very expensive in space. This is really the cost of the flexibility in selecting the next pending edge to combine with the Chart. Pending edges, which have yet to be entered, may span any vertices, including those behind the current parsing front (Figure 7.4.1.b), thus creating gaps in the "parsing history" stored on the Chart. When an edge from behind the parsing front is entered onto the Chart, the new edges created may have

been created before. To avoid the possibility of looping, the entire parsing history must be checked to prevent re-proposing an edge.

Another weakness of the Chart algorithm is the use of every edge to both combine BU and expand TD. This generates a large number of edges, many of which (especially those generated TD) may never be used and increases the edge checking problem when proposing new edges.

7.4.2 Word Incorporation

The WI algorithm can be thought of as a specialization of the Chart. At a cost of some flexibility, the Chart data structure itself and the processing algorithm can be greatly simplified.

The Chart algorithm is restricted to be solely BU depth-first (left to right or right to left). This means that parsing proceeds along the input (base of completed edges) in an orderly fashion and there are no pending edges entered behind the parsing front. The Chart holds only active edges and the pending list consists solely of completed edges. To achieve depth-first parsing, the completed (pending incorporation) edge list is handled on a First-In-First-Out (FIFO) basis.

The parser first builds a (parse tree) base of completed edges where each edge represents a word in the input string. This is much the same data structure as that produced by the Chart algorithm. The dictionary is accessed by a look-up procedure which, if no matching entry can be found, performs simple character alterations (deletion, addition and permutation) to the word and attempts to match the new words produced with dictionary entries [Berghel & Traudt, 1986]. If one or more matches is then found, these words are all used in construction of the chart base, otherwise a look-up failure is reported to the user and the parse fails.

In addition, an initial active edge is proposed at the first vertex with a category which is the distinguished grammar symbol (s). Each complete (base) edge is then combined (BU) with all matching active edges. These being the active edges with a final

vertex number equal to the complete edge's first vertex number and a first remaining RHS category which is that of the complete edge. A complete edge is also used to invoke new rules which have a matching RHS. These being all rules with a first RHS category or literal equal to that of the completed edge.

New active edges are placed on the Chart and new complete edges on the pending list. Newly completed edges are also tested to see if the final "target" edge has been created. Completed edges, once entered onto the Chart, can be dispensed with. Thus, only information on the current active "parsing front" is maintained (Figure 7.4.2.a). This does not mean that parsing goals need be pursued more than once, as each completed goal is used (BU) to its full potential when added to the Chart, and is then expended. As the algorithm proceeds along the base in an orderly fashion, processing never takes place at any vertex more than once, so that clearly no check to see if an edge has been proposed before is required. The resultant decrease in the number of edges held in the Chart and the reduction in processing produce a highly efficient parsing algorithm.

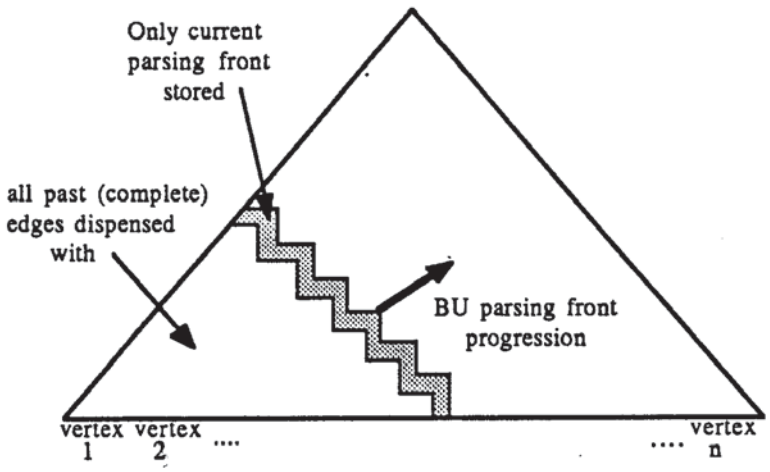


Figure 7.4.2.a Outline of Word Incorporation Parsing Operation

It is characteristic of the WI algorithm that alternative parses are developed in parallel. The efficiency of invoking new rules can be improved by indexing rules by their first RHS category or literal, so that all the required rules can be found when new rules are invoked. Having described the basic operation of the parsing algorithm, some of the other aspects of the parser will now be described.

Edges used by the WI algorithm are only of two types :

- complete (base) edges :

```
base_edge([Sv, Ev, Cat, Info])
```

where *Sv* is the starting vertex, *Ev* the ending vertex, *Cat* the edge's category (word category or rule LHS) and *Info* an information structure built when parsing constituents which are spanned by the edge.

- active edges :

```
active_edge(Ev, Next_cat, [Sv, Ev, Cat, Rem, Info])
```

where *Ev* and *Sv* are the starting and ending vertices, *Cat* the corresponding rule's LHS category, *Info* an information structure representing information parsed so far, *Rem* the remainder of the rule to be parsed, with embedded equation sequences, and *Next_cat* is the next rule category (first grammatical category in *Rem*).

Each cycle of the parser involves removing the next base (complete) edge from the list of these and then using this edge to extend active edges and invoking new rules to produce new edges. Active edges have their ending vertex *Ev* and next remainder grammatical category isolated in their storage format, so that when a complete edge is to be combined with active edges, those active edges which the complete edge may extend can be found simply and efficiently. The set of active edges *Actives* which a complete edge with category *Cat* and starting vertex *Sv* can extend can thus simply be found by use of the *bagof/3* Prolog predicate :

```
bagof(Active, active_edge(Sv, Cat, Active), Actives).
```

Each edge of the list *Actives* will then match the base edge with respect to category and vertex but may only be extended by the base edge if the equations attached to the next category in the remaining part of the active edge's rule can be successfully evaluated.

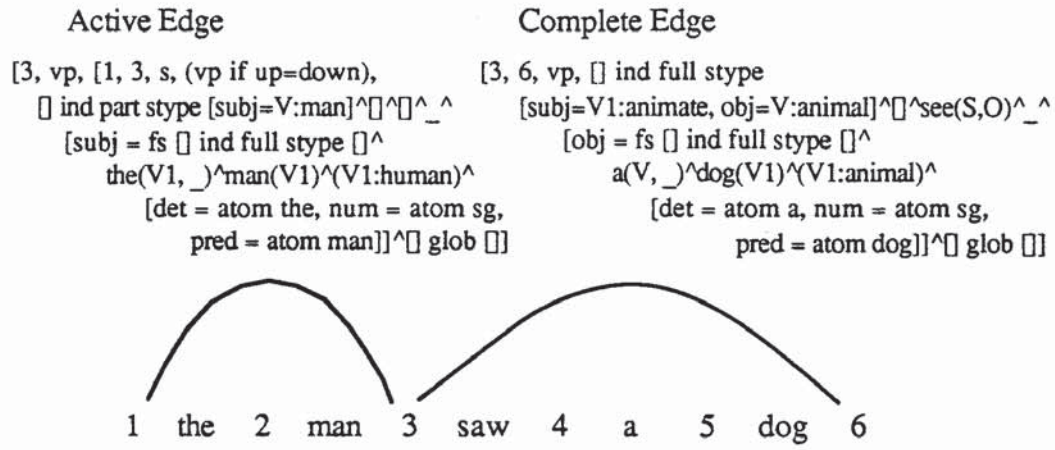


Figure 7.4.2.b Matching Complete (Base) and Active Edges in WI Parser

Figure 7.4.2.b illustrates a WI parsing state when parsing *'the man saw a dog'*. The parser is about to incorporate a complete edge with category *vp* with active edges on the chart. This edge spans the string *'saw a dog'* where the portion *'a dog'* has previously been incorporated into the edge as the obj function. This complete edge will be used to extend all active edges which have a next remainder category *vp* and end at vertex number three. A single such active edge is shown in Figure 7.4.2.b which spans *'the man'* which has been incorporated in the information structure of this edge as the subj function.

The parser will extend the active edge by the complete edge if the equation sequence attached to the *vp* category in the active rule (up = down) can successfully be evaluated. In the evaluation of equations, the complete edge's information structure is that referred to by downward pointing meta-variables (↓) and the information structure of the active edge is that referred to by upward pointing metavariables (↑). The trivial equation following *vp* in the active edge is thus evaluated by simply unifying the information structure of the active edge with that of the complete edge. This will produce a new information structure which will be placed in a new complete edge (as the active edge now has no remainder) which spans the entire phrase. Complete edges which span the entire input string are passed to a well-formedness checking procedure which checks that the edge's information structure (F-structure) is complete. If a complete F-structure is contained in an edge spanning the input string, then this will be a complete parse (a target edge), the information structure of which can then be passed on for semantic translation.

If a complete parse is not found, then the extension of active edges by a complete edge may create a number of new active and complete edges. These new edges are then asserted into the Prolog database. After existing active edges have been extended, the

complete edge is then used to invoke new rules and thus create new active and complete edges. The category of the complete edge is used to retrieve all those rules whose RHS starts with this category (or literal). Each such rule is then given an empty information structure, and the equations attached to the first RHS category of each rule are evaluated using this empty information structure and the complete edge's information structure. Then, if invoking new rules does not produce a target edge, the next base edge (LIFO) is retracted from the Prolog database and used to extend active edges and invoke new rules. This cycle is repeated until a target edge, spanning the input string with a complete information structure, is produced. Coherence is ensured during the incremental construction of information structures themselves.

7.4.2.1 Long-Distance Dependencies

Long distance dependencies are the most troublesome element of LFG to implement correctly. This is partly due to the fact that the exact semantics of the notation is not made entirely clear in Kaplan and Bresnan [1982]. The F-structure values of two functions produced by matching a controller with a controllee are always the same in the examples given by Kaplan and Bresnan. This intuitively seems the correct result, given that the notation is intended to reflect the movement of a complete F-structure component. This interpretation of controller/controllee matching would be best supported, in the implementation described here, by the use of pointers in the same way as pointers are used to support functional control. However, this is not the only possible interpretation of controller/controllee matching. It is to illustrate an alternative implementation of long-distance dependencies that the simpler pointer method has not been used.

The use of pointers to store shared function values in an F-structure means that values are shared throughout (after evaluating the corresponding functional control equation) the parse and the functions will always have the exact same value. In the case of matching controllers with controllees however, the sharing of values may be defined to take place only when the controller is matched with the controllee. Thereafter, the values of the functions which receive this shared value can be treated as separate values. Whilst there does not appear to be any great advantage in this interpretation, it does seem to fit more naturally with the basic concept of passing controllers/controllees through C-structure for matching. When a controllee is matched with a controller, the values of the F-structures carried by these are unified but once the match has taken place, function

values which receive the unification result become separate values. Whilst this interpretation does seem to conflict to some extent with the co-indexing of subsidiary F-structures involved in constituent control, in practice this does not cause any problems.

As parsing is BU, the C-structure's base is constructed first and then extended upward. The parsing of controllers/controllees through this structure is constrained to conform with the development of C-structure by the parser. Thus controllees which are encountered at the base (or at some low point) of C-structure progress upward until a matching controller is encountered. Also during grammar pre-processing, controllers are moved to their domain root categories. Then whenever a controller is found in a sequence of grammar equations, the corresponding controllee must have been found previously.

(c) I ind full stype [subj=V:human, obj=V1:object]^
[]^see(V,V1)^[obj = fs [] ind full stype []^a(V1,_)^book(V1)^(V1:object)^
[det = atom a, num = atom sg, pred = atom book,
subj = fs moved 1 ind full stype []^a(M,_)^man(M)^
[det = atom man, num = atom sg, pred = atom man]
temp 1 ind part stype [J^[J^[J^_^[num = val_c sg]]^[]] glob []

(e2) [] ind full stype []^a(M,_)^man(M)^
[det = atom man, num = atom sg, pred = atom man] glob [] $= \Downarrow_{np}$

(b) I ind full stype [subj=V:human, obj=V1:object]^
[]^see(V,V1)^[obj = fs [] ind full stype []^a(V1,_)^book(V1)^(V1:object)^
[det = atom a, num = atom sg, pred = atom book,
subj = fs moved Final temp 1 ind part stype []^[]^[]^_^[num = val_c sg]]^
[controllee(np, moved Final temp 1 ind part stype []^[]^[]^_^[num = val_c sg]] glob []

(e1) $(\uparrow \text{up subj}) = \Uparrow_{np}$

(a) I ind full stype [subj=V:human, obj=V1:object]^
[]^see(V,V1)^[obj = fs [] ind full stype []^a(V1,_)^book(V1)^(V1:object)^
[det = atom a, num = atom sg, pred = atom book,
subj = fs [] ind part stype []^[]^[]^_^[num = val_c sg]]^[] glob []

Figure 7.4.2.1 Information Structures in Long Distance Dependencies

During the period between the evaluation of an equation containing a controllee and the evaluation of an equation containing the corresponding controller, the F-structure value of the controllee is represented by a Prolog term :

```
moved Final temp <info_struct>
```

where Prolog operators are shown in bold type, *Final* is a Prolog variable and *<info_struct>* is the current information structure value. In Figure 7.4.2.1 an equation *(e1)* is evaluated against an information structure *(a)*, which is that referred to by the '↑' metavariable, to produce a new information structure *(b)*. This information structure is then used in evaluating a set of equations containing a controller *(e2)*, where the controller also carries an information structure. In matching the controller with the controllee, the controller's information structure is unified with the *<info_struct>* value of the controllee on the controllee list and the resultant information structure is used to instantiate the *Final* variable of the controllee. This will also instantiate the corresponding *Final* variable (the same Prolog variable) in the F-structure list. In this way, the instantiation of a function's value, derived from matching a controller with a controllee, can be suspended until the controller is located and constraints on the controllee's value can be held until they can be evaluated.

7.4.2.2 Top-Down Linking

The WI algorithm as described by [Phillips, 1986] exhaustively develops each edge bottom-up. Complete edges, which can never be used to extend active edges, are still developed upwards until they cannot be used to extend any edge. The unnecessary upward development of complete edges can be reduced by extending the parser to include a top-down expectation element [Matsumoto & Tanaka, 1983]. This takes the form of a "linking relation" which can be computed off-line before parsing. Matsumoto and Tanaka [1983, p150] state that:

> " Let the 'link' relation hold between the two categories A and B when there is a grammar rule whose form is 'B → A ..... '. Assume that the 'link' is reflexive and transitive relation. "

The link relation here is produced automatically by the grammar pre-processor when rules are pre-processed. The task of finding elements of the link relation is separated into two parts : finding "immediate" links and then using these to find "extended" links. An immediate link exists from a category $<cat_a>$ to a category $<cat_b>$ if there exists a rule

$<cat_b>$ ---> $<cat_a>$ .....

The extension of these links adds additional links to create the set of extended links. An additional link exists from $<cat_a>$ to another category $<cat_c>$ if there is an immediate link from $<cat_a>$ to $<cat_b>$ and an immediate link from $<cat_b>$ to $<cat_c>$.

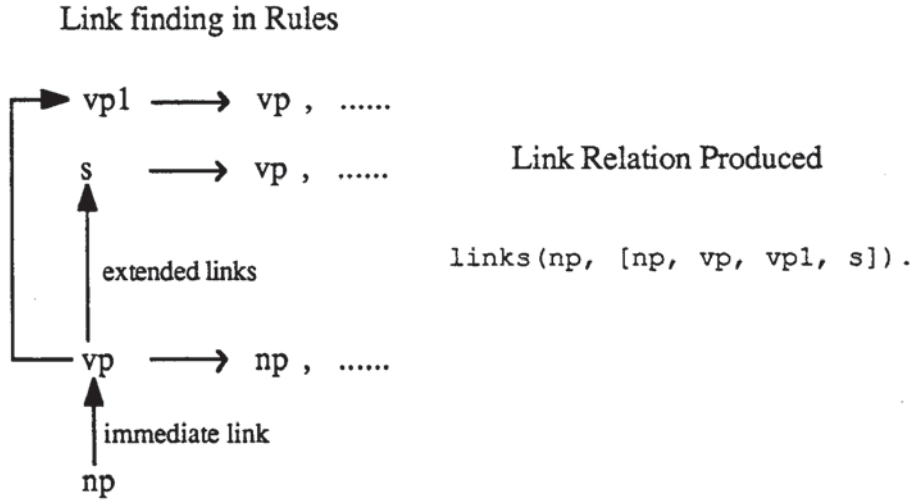


Figure 7.4.2.2 Link Relation Production from Grammar Rules

The link relation can thus be thought of roughly as defining 'the set of grammatical categories of which a single grammatical category may form the first or only component'. An example of immediate and extended linkage is shown in Figure 7.4.2.2.

The link relation is used by the WI parser when invoking new rules. A rule is only invoked when its LHS category might be used to extend some current active edge. For this to be true, there must be at least one active edge with ending vertex equal to the starting vertex of the new edge, with a first remainder category which is linked to the category of the new edge. In this case, if the new edge can be completed, it might then be used to extend the active edge(s) to which its category is linked. If no active edge can be found which is linked to the new edge's category then the corresponding rule is not invoked.

7.4.2.3 Literals

It is a simple matter to extend the WI algorithm to parse literals which appear in grammar rules. The grammar must first also be indexed by literals in just the same way as categories. Only 'single item' literals are allowed so that a literal will always be a single item (edge) in the base of edges. Of course, several contiguous literals may be used in a grammar rule. Essentially, edges are matched, as has been described, by vertex and

grammatical category. Now also, edges are allowed to match by vertex and the literal represented (by a base edge).

Literals in rules could be matched to any complete (base) edge including those representing more than one literal but this would require that every complete and active edge be maintained with information about the string it spans. Instead, literals in rules are confined to match one initial base edge so that the parser need only introduce an initial set of literal type of base edges for matching with these (Figure 7.4.2.3). This reduces the overhead of dealing with literals to entering an extra (literal) complete edge at each successive vertex, and invoking grammar rules which have a RHS starting with a base literal. Parsing can proceed as before as there is no difference between matching a base 'word type' edge with a rule's (RHS) word and matching a base edge with a rule's (RHS) category.

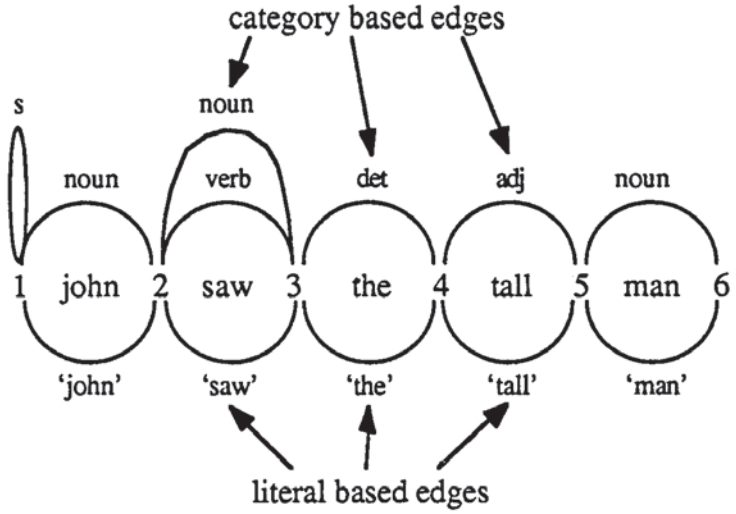


Figure 7.4.2.3 Initial Base of Edges Including Literal Edges

Not all words in the base need generate word type edges, only those words which are mentioned in the grammar, which are marked as grammar words in the Prolog lexicon.

7.4.2.4 Gaps

The method adopted here for dealing with gaps requires that a gap (*e*) be viewed as a dictionary or lexical item rather than a terminal in the grammar. A single entry for gaps can be imagined which has category *e* and a nil lexical realization. In LFG, a gap

corresponds to some other constituent to which the gap is related by the matching of bounding metavariables.

Gaps are dealt with simply by hypothesizing a gap at each vertex in the base of edges. Gaps are generated by the parser during the initialisation phase where each gap edge has category *e*, no remainder (*ie* forms a complete edge) and the same starting and ending vertex (Figure 7.4.2.4). The category *e* is used in the CFG to denote a gap which can match an *e* edge in the C-structure's base.

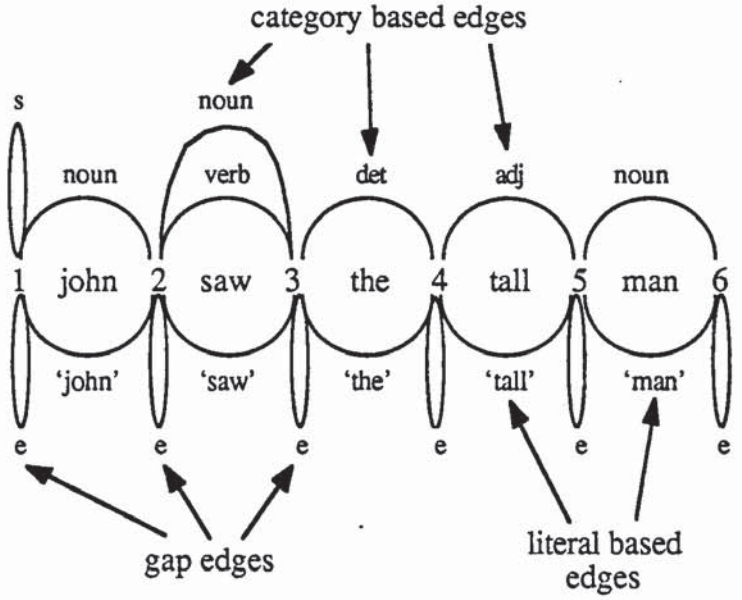


Figure 7.4.2.4 Initial Base of Edges Including Gap Edges

The use of gaps in the C-structure base can in fact be eliminated, which would improve the efficiency of parsing. This is noted by Falk [1983, p251] :

> " Inducing C-control [Constituent control] without traces is actually not very problematic in a framework such as LFG. In LFG, C-control (or syntactic binding) occurs not as a result of the existence of an empty category but rather by virtue of the existence of a pair of long-distance domination metavariables. "

As this implies, gaps in LFG are used merely as a traditional 'category' to which a controllee can be attached. However, constituent control is realized by the controllee, not by the gap. Controllees can be attached to other nodes in C-structure at levels above C-structure terminals (gaps) to match controllers in exactly the same manner as if attached

to terminal *e* categories. The use of *e* categories (gaps or traces) has not been excluded from the implementation described here, as this would depart from the description of LFG given by Kaplan and Bresnan. Other equations can however be used to eliminate gaps, (such as '$(\uparrow \text{subj}) = \Uparrow_{np}$') and if these were to be used to replace all uses of *e*, then their inclusion in the parsing base and upward propagation could be eliminated. This would improve the parser's efficiency.

7.4.2.5 The Kleene Star

The Kleene-star, as described by Kaplan and Bresnan [1982] and in Chapter 2, is used to signify that any number of repetitions of some constituent in C-structure may occur. A restriction to this notion was also described in Section 7.2 where the Kleene-star was postfixed with an integer signifying the maximum number of repetitions which may occur ("limited Kleene-star"). The Kleene-star is retained in the pre-processed grammar rules so that it can be detected by the parser and dealt with at parse time.

Use of the Kleene-star does however complicate several other aspects of the parsing method described previously. In particular, the Kleene-star must be taken into account when producing the TD link relation and indexing rules by their first RHS grammatical category. As an example, consider a rule of the form :

$$\text{<lhs>} \longrightarrow \text{<cat}_a\text{>}^* \quad \text{<cat}_b\text{>}^* \quad \text{<cat}_c\text{>} .$$

which may in fact be invoked (BU) by a complete edge of category $<cat_a>$, $<cat_b>$ or $<cat_c>$. When the parser invokes new rules and looks for active edges with which to combine a complete edge, it collects all those with the appropriate first or next RHS category (as well as checking the ending vertex number of active edges). In the case of a rule with one or more Kleene-stars there may clearly be a number of possible next or first RHS categories, as the rule above illustrates.

Kleene-star operators are currently only allowable on up to two consecutive categories in any grammar rule, although apart from this limitation, they may appear anywhere in a grammar rule. This limitation is imposed because rules are processed by declarative procedures which perform the necessary 'look-ahead' in grammar rules, to see if a Kleene-star repetition may have finished, in which case, the remainder of a rule (if any) is then processed. This limitation could easily be relaxed by use of additional rules

which match larger rule portions, or could be removed by using recursion, but in practice, it seems unlikely that grammar rules with long sequences of categories all marked with Kleene-stars would be required. The restriction means that in a grammar rule only the first two categories of the RHS may have Kleene-stars and thus, there can only be at most three categories which may at parse time be the actual initial category consumed by a rule. The rule pre-processor will thus add a rule (such as that shown above) to three different sets of rules indexed by different first RHS categories. Such a rule can then be invoked by any one of $<cat_a>$ or $<cat_b>$ or $<cat_c>$. Active edges, which are indexed by next RHS category, will now be indexed by a list of up to three categories which can be the next grammatical category used to extend the active edge.

The link relation takes Kleene-star operators into account in much the same way as active edges are indexed. In the case of the example rule, a link is made to the $<lhs>$ from each of the possible first RHS categories $<cat_a>$, $<cat_b>$ and $<cat_c>$. These links are then all extended as before.

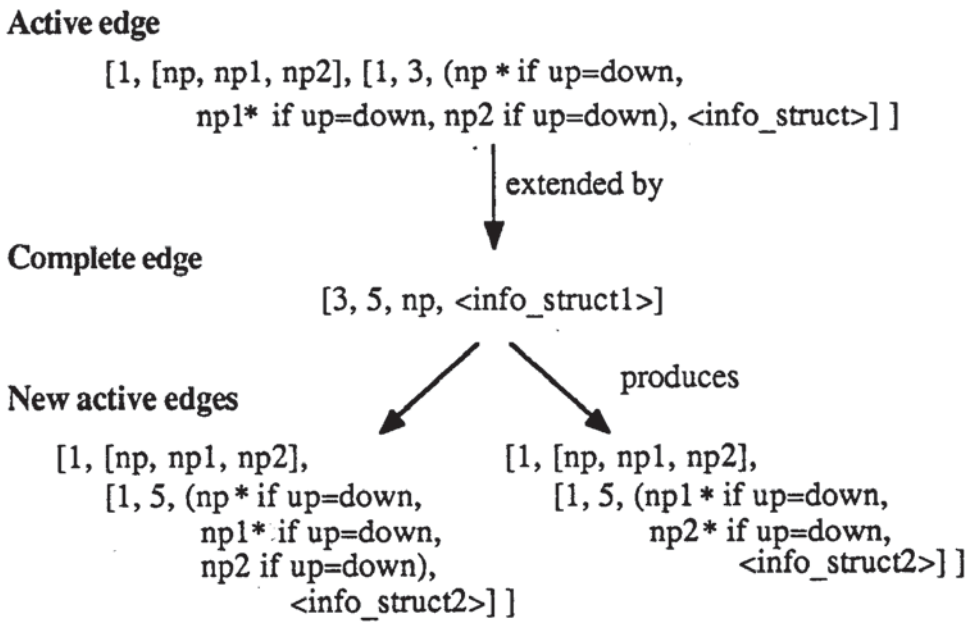


Figure 7.4.2.5 Extension of Active Edge with Kleene-star Operator

During parsing, rules are invoked if the link relation holds between a complete base edge's category and any one of a rule's possible first RHS categories. When an active edge with Kleene-stars is extended by a complete base edge, a number of new active edges and possibly a complete edge can be created. As an example of such a case, Figure

7.4.2.5 shows in outline, the extension of an active edge, where the next RHS category has a Kleene-star attached, to produce two new active edges.

7.4.2.6 Conjunctions

As described in Section 4.9, in the restricted set of conjunctions considered here, a single controller is passed into each branch of a conjunction to be matched with two or more controllees. There are several different approaches to parsing conjunctions but here conjunctions are dealt with explicitly in the grammar (*cf* demons [Dahl & McCord, 1983]). Not only are the C-structure constituents and corresponding equations of a conjunction stated explicitly, but conjunctions are also prefixed in the grammar with a Prolog operator 'conj' (a similar qualifier is used by Sedogbo and Bull [1985]) :

```
rel_s ---> conj  rel_s  eqns  down set_val_of (up conjs) ,
                 and ,
                 rel_s  eqns  down set_val_of (up conjs) .
```

The conj operator is retained in the preprocessed grammar rules and acted upon by the parser at parse time (as shown attached to C-structure in Figure 7.4.2.6).

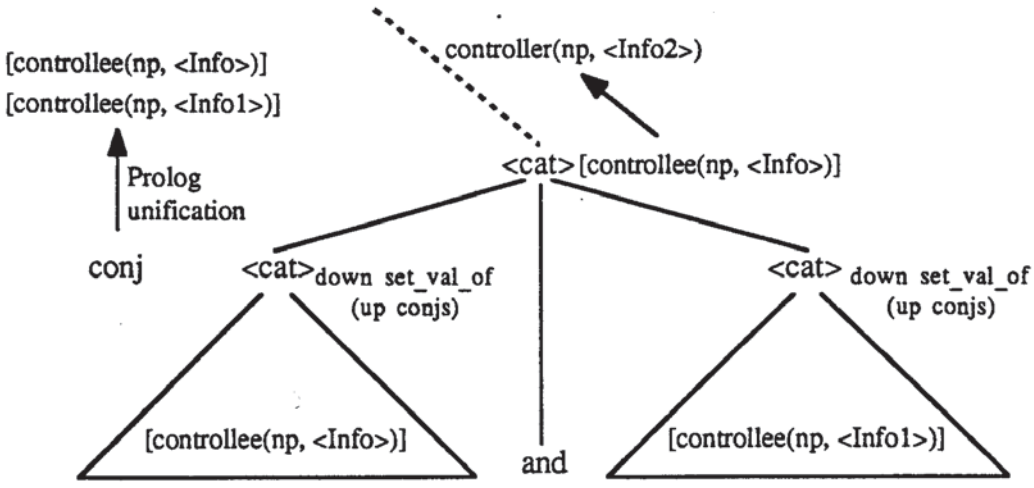


Figure 7.4.2.6 Outline of C-structure Produced from Conj Prefixed Grammar Rule

When a conj operator is found (Figure 7.4.2.6) by the parser prefixing a conjunction of some category, the controllee lists from each conjunction repetition are expected to be the same. These are unified (currently using Prolog unification) and a single controllee list, from one of the conjunctions, is passed upward for matching with the corresponding

controller(s). When a controller is matched against a controllee in this list both of the corresponding controllees in the conjunctions will be instantiated with the controller's value.

7.5 Well-formedness Checking

7.5.1 Well-formedness Checking During Parsing

During parsing, the coherence condition is maintained by the definition of unification on information structures themselves. Under normal conditions, it is not possible to perform any checking of the completeness condition until a complete parse has been produced and the F-structure is known to have its full informational content. If completeness can be checked to some extent during parsing, it might greatly improve the efficiency of parsing as a great number of incorrect parses may be eliminated. This is why Block and Hunze [1986], as described in Section 6.3, have extended the notion of bounding categories to prevent functions being passed through these. This means that, parsing bottom-up, when a bounding category is encountered, the F-structure from below the bounding node must be complete. This is only the case if the linking equation is not used, as this may pass a controller down to the F-structure to make the F-structure complete. An alternative approach has been taken here to allow some completeness checking during parsing. The method used by Block and Hunze could be used in addition to this to further improve parsing.

During parsing, functions, which are the main concern of completeness, are specified as F-structure members by equations such as '$(\uparrow \text{subj}) = \downarrow$'. This equation specifies that the F-structure from below the C-structure node to which the equation is attached is to be unified with the subj function's F-structure of the F-structure above the node. In many cases, the subj function produced after evaluation of this equation will itself be both complete and coherent. Of course this depends on the parsing action (BU or TD). If the parser operates in a predictable fashion, as the WI parser does, then it is possible to know how C-structure is produced and in certain instances, it can be stated when a subsidiary F-structure (function) will be complete.

An optional prefix operator 'complete' is defined in the LFG notation which can be used to specify that the function produced by a functional assignment equation is expected to be complete at the time of the equation's evaluation (actually after evaluation of the equation sequence in which the equation appears). Here, 'complete' means not only

functionally complete, as defined in LFG, but 'in place', that is not subject to constituent control (an F-structure value yet to be found by controller/controllee matching).

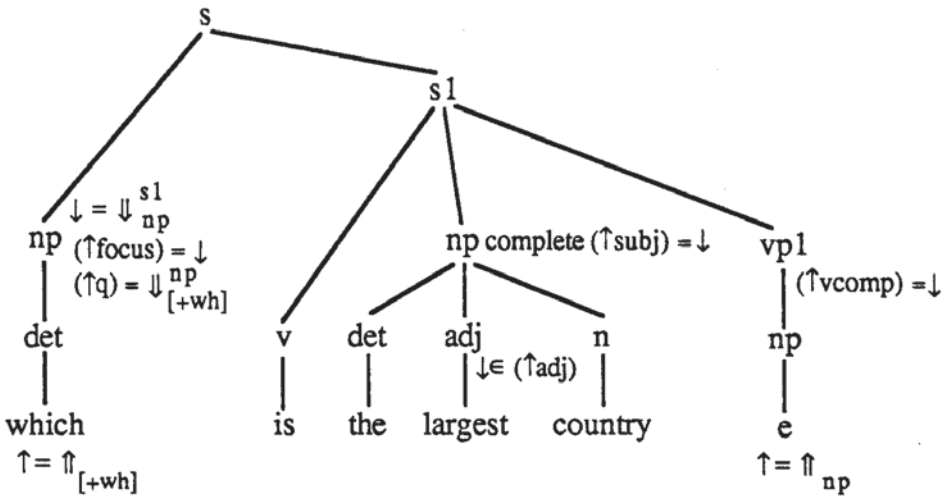


Figure 7.5.1 C-structure Outline with Completion Operator

As an example, consider the subject function in a query *'which is the largest country'*. The annotated C-structure of this query is shown in Figure 7.5.1. In this case (of equative *be*), it is known when the WI parser evaluates the equation '(↑ subj) = ↓' the subj function will be 'in place' and complete. The corresponding grammar rule which specifies this equation in the LFG notation will thus be :

```
s1   --->    v       up = down ,
             np      complete (up subj) = down ,
             vp1     (up vcomp) = down .
```

The evaluation of an equation, pre-fixed with a completion operator, not only checks that the specified function has a well-formed F-structure but also gives the *<slot_type>* part of the function's information structure a value of *cc*. This indicates that the function's F-structure is complete and has been checked for well-formedness. Information structures with a *<slot_type>* value of *cc* are not re-checked after parsing for well-formedness.

It should be noted that the use of this operator is optional. This is because its use requires an understanding of the parser's action. The vcomp function, for example, could not here be specified as 'complete', as at the time that the '(↑ vcomp) = ↓' equation is evaluated, the vcomp function will not be 'in place' as the controller which actually carries the F-structure value which will fulfil this role, will not have been found.

Although use of the 'complete' operator requires an understanding of the parsing action, it can be used to improve parsing efficiency. The use of this operator could be extended so that it could be used in isolation, rather than when a functional assignment equation is used. For example, as shown in Figure 7.5.1, the vcomp function will be in place and complete when the *s* rule is extended by the complete *s1* edge. This might allow a greater usage of the operator and a corresponding improvement in parsing efficiency.

7.5.2 Well-formedness Checking Post-Parsing

When a complete parse is produced (a complete edge spans the entire input string), the F-structure must be checked for well-formedness. The coherence condition is maintained by the definition of unification during parsing, so that only completeness must be ensured after parsing. For completeness to be ensured, it only remains to check that the functions listed in the slot component of the information structure have values in F-structure component and that no existential constraints are left outstanding. Function values as described previously may be implied by constraints, the value assigned to a function in such circumstances may however have no actual content besides a constraint, which may have been satisfied. Functions must therefore not only be present in the F-structure component but must also have some content (features). An 'empty' F-structure is thus not accepted as well-formed.

7.6 Semantic translation

Semantic translation acts on the well-formed information structure of a complete parse and is divided into three stages :

- isolation of semantically relevant features in the F-structure component and re-structuring.
- resolution of quantifier scoping.
- translation of the remaining information structure into a Prolog query.

The first of these stages basically simplifies the information structure for semantic translation. Only part of the information in the F-structure is required for semantic translation. These, of course, include function values, which are recursively simplified themselves, and certain simple features which are semantically relevant in database querying (not the database domain). It is expected that a small number of such features

will exist. Most semantic features are placed in a Prolog list where the position of each element in the list signifies the feature's name. This list takes the form :

[<pro>, <num>, <int>, <adj>, <aggs>]

where the significance of each element will be explained below. If no value exists for an element of this list, the nil value '[]' is used. Other features are promoted to the slot list of functions if they are involved in quantification. Currently the set of semantically relevant features includes :

- proportional (proportional = atom +) features, which signifies a higher order predicate which operates on proportions, such as *'percentage of'*. This type of noun has to be treated specially in semantic translation, where a more complex series of predicates is actually used to realize the taking of proportions. This is much like the treatment of aggregate adjectives. The sem feature value (predicate) of a subsidiary F-structure containing a proportional feature value is held in the *<pro>* element of the output list.

- number features (num = atom sg, num = atom pl), the value of which is given to the *<num>* element in the output list. This feature is primarily used to determine whether a quantifier refers to a set of values (plural), in which case the *setof/3* Prolog predicate is used in the Prolog query.

- cardinality features (card = atom N), which is added to the output list of semantic features (new F-structure component). The card feature, which originates from lexical numbers (1, 2, etc), may however be modified by a comparative (*'more than 3'*) or by a unit of measure (*'3 million'*). The card feature and modifier is used to produce the *<int>* element in the output feature list . For example :

  [card = atom three, q = atom more] ⟶ > 3
  [card = atom three, meas = atom million] ⟶ 3 -- million

- ordinary adjectives, held as set values in an F-structure, are placed in a list which becomes the value of the *<adjs>* element.

- aggregate adjectives, also held as set values, are placed in a list which becomes the value of the *<aggs>* element.

The negation feature (neg = atom +), if present, is added to the information structure's list of functions in the slot description as an atom 'neg'. This is done so that the scope of negation can be determined in relation to the functions in the slot list.

In addition to constructing the list of semantic features, adjunct functions (being a set value) are attached to some function by equating functional variables, as described in Section 4.8, and the list of functions in the slot description is extended to include set value names, such as adjuncts and conjs, and additional function names such as the pied and rel_adj functions, which are also described in Sections 4.7 and 4.8. If however, a relative structure consisting of non-subcategorizable head and mod functions is encountered, these function names are used as a substitute slot list.

The semantic features of the F-structure are thus isolated in a standardized list and the slot list is extended to list the functional components (and negation) which are subject to quantification as well as semantic translation.

The second stage in semantic translation, quantifier scoping, first examines the quantifiers of each function in the slot list and classifies each of these as either strong or weak quantifiers. A special function name *f* is also added to the list of functions, which signifies the quantifier of the function itself, inside of which the other functions named in the slots list are held.

A list is constructed in the same order as the functions in the slot list, where each element of this list is either the symbol strong or the symbol weak, according to the nature of the function's quantifier. This list, together with the slot list of function names, is then used to determine quantifier scoping. For example, the lists might be :

quantifier properties	[weak, weak, weak]
function list	[f, subj=V:T, obj=V1:T1]

in which case the default scoping order for the F-structure's quantifier and those of the subj and obj functions will be applied (subj > obj >f) to produce a function list :

[subj = V:T, obj = V1:T1, f]

If however, the obj quantifier was a strong quantifier, then this would be given highest scope and the corresponding output list would be :

[obj = V1:T1, subj = V:T, f]

Quantification resolution is the last stage before translation of the information structure proper takes place. Translation has been described in some detail in Section 3.3 and the details of this need not be repeated here. The interpretation of the information structure follows roughly the DAG transformations described in that section. The function with highest scope (first on the slot list) is currently translated first and then other functions in the list are translated and placed within the scope of quantifiers encountered previously. The semantic features and quantifier of an F-structure are translated when the function name *f* is encountered in the slot list.

7.7 Efficiency of Parsing

Although parsing efficiency has not been of major concern here, this aspect deserves some comment. Parsing appears quite efficient (Appendix F) but is still much slower than that of the Chat-80 system. For example, queries with four to ten words take from about one to six seconds to parse. Chat-80 parses such queries in under one tenth of a second. There are a number of reasons which contribute to the slow parsing rate of the system described here, in comparison to Chat-80. The most important of these is simply that exhaustively elaborating all possible parses in parallel always involves a great deal of computational work. However, the system's performance is quite close (of the same order of magnitude) to acceptable, if one second is viewed as an adequate response time to a typical query. This is believed to be quite significant, given that a high-level grammar description has been used and the example grammar used to parse queries constructed with a linguistic rather than computational bias. A number of methods of improving the parser's performance are also possible and are outlined below.

The parser currently suffers from a lack of memory (signalled by pauses for garbage collection), when running on a Sun 3/60 with four mega-bytes of memory. Two methods of reducing memory usage are immediately apparent. Either the number of edges, held and produced, can be reduced or the amount of memory used by edges can be reduced.

The number of edges held might be reduced if active edges behind the parsing front, which will not be used, are removed. In most cases, this is not possible as any active edge may be extended by a complete edge. However, those active edges which are waiting to be extended by a terminal category, which is behind the parsing front (has been used to extend active edges), cannot be extended and could thus be removed.

The number of active edges produced might be reduced by use of a look-ahead facility as used in the PAMPS system [Uehara *et al*, 1984a], which also uses a BU parser similar to WI. The amount of memory used by edges might be reduced by using a structure-sharing technique described in Pereira [1985]. This technique does not create (copy) entire DAGs (F-structures) during unification but represents a DAG by a pointer to its basic structure (skeleton) and another pointer to a table of updates to the DAG. When two DAGs are unified, one can be chosen as the basic structure (actually a pointer to this is created) and the other used to produce the table of updates which if applied to the basic structure would produce the new structure. The updates themselves are co-indexed with the basic structure's portion to which they apply and consist of values not specified in the basic structure. These values are in turn defined (largely) by pointers to parts of the DAG used to construct the update table.

A much simpler improvement to the parsing algorithm itself, which became apparent in the latter stages of development, also remains to be implemented. This relates to the use of the top-down link relation. Currently, this is only category based but there is no reason why it could not be extended to F-structure. This has also been suggested by Block and Hunze [1986, p491] as described in Section 6.3, but was rejected by them as too computationally costly when used with the Earley algorithm. This is because the Earley algorithm, when generating new edges, must check that these exact edges have not been generated previously. This requires comparing the F-structures held as part of edges, which is a very expensive operation. The WI algorithm does not however carry out such a check so that F-structure comparisons would not be necessary.

There is however another 'cost' caused by extending TD prediction in this way. Several similar active edges but with different F-structure content may be linked TD with a new edge. In this case, a new edge must be created for each different F-structure of the edges with which the new edge is linked. This means that invoking a new rule will produce a number of initial new edges, which will require more memory than a single edge, and all of which must be processed. It may thus be more efficient to use only perhaps subcategorization lists in TD prediction, as these will vary much less than entire

F-structures but will still impose coherency on predicted edges. A similar feature-based extension to the Earley algorithm's TD prediction step in a PATR II implementation [Shieber, 1985b] has been shown to greatly reduce the number of edges generated.

The current parser implementation adds edges produced during parsing to the Prolog database. This however was not the case in an initial implementation. The Prolog *assert/1* predicate is very computationally expensive, as adding code to the Prolog database requires calculating values for the hashing function which is used to index rules in the Prolog database. In addition to this, predicates, the definition of which may be changed in the Prolog database ("dynamic" predicates), are only executed in an interpreted manner, even when the code is itself compiled. This means that changing or matching edges in the WI Chart is done very slowly.

In an earlier implementation, Chart active edges were held in an array-like data structure. This structure was passed as a single Prolog argument through the parsing rules and thus not asserted or matched in the database. Complete (base) edges were also handled by a single Prolog list passed as a Prolog argument through the Parser's rules. This allows FIFO control to be simply achieved by adding or removing an element from the head of this list. The array for active edges was generated when constructing the initial base, where an array index corresponds to the last vertex number of an active edge. Each element of the array holds a list of active edges, each with the same last vertex number, so that when adding a complete edge to the chart, the required edgeset can simply be found by accessing a single element of the array. These data structures more naturally 'fit' the WI algorithm and allow efficient access to edges. However the cyclic action of the parser when using these data structures leads to an excessive memory usage, so that the time spent garbage collecting and saving information for backtracking (even using cuts) is so great that parsing is slower than when using assertions to the database and then backtracking (when memory can efficiently be recovered). This is viewed as a major weakness of Prolog itself when implementing naturally cyclic algorithms which use (alter) large data structures.

Whilst the alterations to the parsing system above may well increase performance of the system to a level comparable with Chat-80, it is expected that a similar improvement in performance could be achieved by simply providing more memory or recoding part or all of the system in a programming language which is fully compilable, to produce more efficient machine code.

Chapter 8

Query Planning and Database Querying

Having translated the F-structure output from its LFG analysis into a semantic structure, it remains to use this structure to query the database and produce a response to the user. An outline of the processing sequence which takes the logical expression produced by semantic translation and produces a response is shown in Figure 8 below.

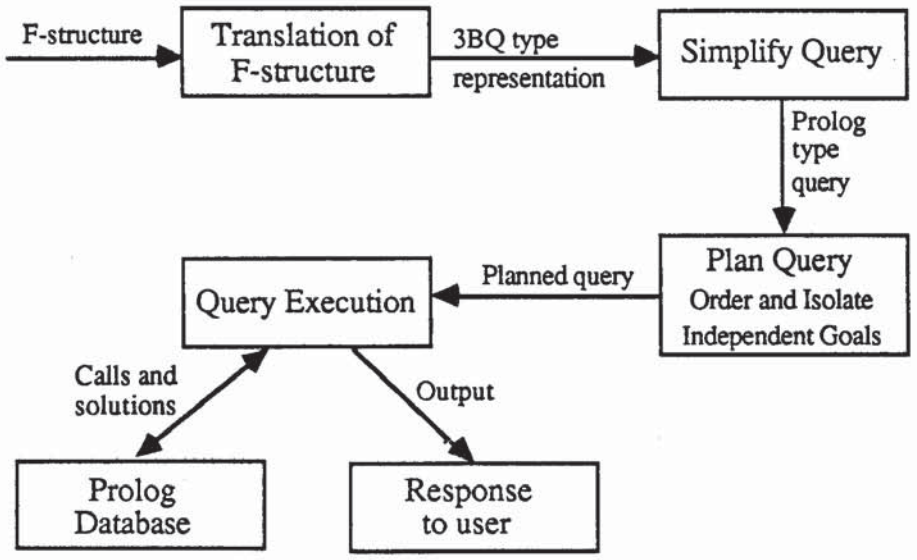


Figure 8 Processing of Queries after Semantic Translation

8.1 Query Simplification

Translation of an F-structure produces a semantic representation which contains three-branched quantifiers (as Colmerauer [1982]). For the purposes of simple database querying, as performed by Chat-80, some of the information in this representation is redundant. Quantifiers, for instance, which signal a single solution, *a* and *the_sg*, may be eliminated from the representation. This reduction does however eliminate presupposition information. That is to say, the implicit assumption by the user that there is a single unique solution to a query is not represented in the simplified query. Information about presuppositions can be used to correct user conceptions about the database but are not a necessary part of query answering.

As well as removing certain 3BQs, the process of simplification transforms other 3BQs into equivalent Prolog goals and pragmatic representations. Indexed sets are employed in Chat-80 to produce reasonable answers to questions containing plural genitive noun-phrases such as :

'Who are the children of the employees ?'

It might be acceptable in response to this query to produce a simple list of all employees' children, but a more useful response would a table of employees and their respective children. Even if a list of children were acceptable, it would not be a simple matter to produce such a list [Pereira, 1982, p147]. Given a set of children and a set of employees, it might take a very long time to apply a predicate 'child_of(Employee, Child)' to these lists. To produce reasonable answers, indexed sets are introduced :

> " a predicate *p* applies to sets *x* (children in 4.6 [the sample query above]) and *y* (employees in 4.6 [the sample query]) when the following relation holds :
>
> x = { X : for some Y in y, p(X, Y) }
>
> In other words, if we see *p* as a set of pairs, *x* is the set of the first coordinates (the projection on the first coordinate) of all pairs whose second coordinate is in *y*. I call such a predicate projective. "

The noun phrase in the sample query is thus translated as :

```
{ (E, SC) : employee(E) & SC = { C : child_of(C, E) } }
```

The occurrence of genitive noun phrases in the 3BQ type representation is signalled by the occurrence of an 'of_the' quantifier in conjunction with a plural quantifier. A representation such as :

```
of_the(E, employee(E), the_pl(C, child_of(E, C) )
```

is transformed into :

```
setof(E : C,  employee(E) & child_of(E, C) )
```

Other transformations convert 3BQs into their usual Prolog equivalents :

```
each(V, P & P1)  ⟶   \+(P & \+( P1) )
any(V, P)        ⟶   existential quantifier (also eliminated)
no(V, P)         ⟶   \+(P)
```

After having performed these transformations, the query representation is passed to the query planning module.

8.2 Query Planning

The Chat-80 system directly produces a first-order logic expression which may be executed as Prolog program against a Prolog database. In many cases however, the execution of this 'program' will be most inefficient, as the query lacks the organization (ordering of procedure calls) and control information (cuts) usually provided by the programmer. Warren [1981] has described an algorithm for planning queries and providing some of this missing information. It should be noted that the algorithm is based on a number of heuristics and it does not always produce a query which is as efficient as the query that a programmer could produce by hand crafting. Query planning is here discussed under two headings but it should be noted that these are not independent stages of processing. Planning can be described as a recursive process where on each recursion either :

- the remaining goals in the query, given the current set of variable states (instantiated or uninstantiated), can be subdivided into independent goal sequences where no uninstantiated variables are shared by the goal sequences.

or

- the next 'best' (least costly to execute) goal is found, from those remaining to be executed, and execution of this goal is simulated by recording any uninstantiated variables in the goal as instantiated (goal ordering).

8.2.1 Ordering Sub-Goals

Consider the Prolog query shown below (taken from [Warren, 1981]) first in Chat-80 format *(a)* and then in the format produced by the system described here *(b)* :

'Which countries bordering the Mediterranean border Asian countries.'

(a)
```
answer(C) <=
        country(C) & borders(C, mediterranean) &
        asian(C1) & country(C1) & borders(C, C1) .
```

(b)
```
wh(C, country(C) & borders(C, mediterranean) &
        asian(C1) & country(C1) & borders(C, C1) ) .
```

These queries are obviously equivalent and differ only slightly in format. The order of predicates in the query is simply that which might be expected to be produced from the English query. Warren [1981, p275] states that :

> " Executed as it stands, this query would take a time of the order of the number of Mediterranean countries multiplied by the total number of countries. "

This time for the Chat-80 database is roughly of the order 20 * 150 = 3000, and in the worst case (ordering of predicates) could be as high as 150 * 150 = 22500. After planning, Chat-80 produces the new query *(a')* :

(a')
```
answer(C) <=
        borders(C, mediterranean) & country(C)  &
        borders(C, C1) & asian(C1) & country(C1) .
```

The time now being of the order of Mediterranean country borders (roughly 90). In this example case, the reduction in execution time is significant but not critical. In other cases (particularly when larger number of goals occur with a larger number of variables), the saving may reduce execution time from an unacceptable time, from perhaps many minutes to a few seconds.

So as to order goals in a query, a cost estimate of the time to execute each goal in the query must be made. Having done this, the lowest costing goal can be selected; its variable arguments noted as instantiated, and the process repeated on the remaining goals of the query. Note that only the costs of goals whose argument instantiations have changed need be recomputed. The cost of executing a goal is related to the number of alternative solutions in the database to the goal. This in turn depends on the particular goal and its argument's state of instantiation. In Chat-80, statistical meta-data about predicates in the database is used to find the cost of executing a goal. For example meta-data about goals in the sample query above might be :

Predicate/Arity	Predicate Size	Argument Domain Sizes	
country/1	150	300	
asian/1	100	300	
borders/2	900	180	180

The size of a predicate is equal to the number of tuples making up the relation in the database which correspond to the predicate. The size of a virtual relation in the database is simply equal to the number of different tuples that belong to the relation. The exact meaning of an "argument domain size" is unclear but is described as the size of the domain over which the argument ranges. Warren also notes that "it is somewhat problematic to decide what constitutes the domain of a predicate argument position". Relations which have an 'infinite' size (<, >, =) can be dealt with by assigning them a very high cost unless both arguments are instantiated.

The cost function used in Chat-80 calculates the cost of a predicate as the predicate size divided by the product of the sizes of each argument domain of each instantiated argument. The cost of executing the *borders/2* predicate, with arguments *C* and *C1*, for each state of instantiation will be :

- if both *C* and *C1* are uninstantiated : 900
- if one of *C* or *C1* are instantiated : 900 / 180 = 5
- if both *C* and *C1* are instantiated : 900 / 180 * 180 = 1 / 36

The Chat-80 program actually works on logarithms of these values so that computations can be made by addition, rather than more costly multiplication.

Planning in the system described here is very closely based on planning in Chat-80, which has been described above, but has been extended and clarified somewhat. Firstly, the meta-data about the database is compiled automatically : in the case of Chat-80 this has clearly been collected (estimated) by hand.

To automatically collect meta-data, a specification about relations in the database must be supplied. The specification contains :

- the predicate with which relations can be accessed. This implies that all relations in the database must be accessible by a single predicate :

```
access <predicate>
```

- the Prolog module containing the predicate with which the database can be accessed :

```
module <module_name>
```

- a list of relations, their arities and argument descriptions :

```
relations [   <relation_description>1 ,
              <relation_description>2 ,
                    .....
              <relation_description>n  ]  .
```

where each *<relation_description>* takes the form :

```
<predicate_name> type [ <argument_descriptions> ]
```

and for each argument of each predicate there is an *<argument_description>* which is either *symbol*, if the argument can take a limited number of symbolic values or *inf*, if the value can have an infinite number of values.

The specification allows the meta-data collecting program to access database relations and find how many solutions there are for each relation. Then a rough estimate of the time cost of each relation can be made based on the relation's size. The size of a predicate is estimated simply as the number of solutions to the predicate if both arguments are uninstantiated. For a predicate with description :

```
...., p/2   type   [symbol, symbol ], ....
```

this can be found by finding the length of a bag of solutions generated by the Prolog *bagof/3* predicate :

```
predicate_size(Ps) :-
          bagof(A-B, p(A, B), Bag),
          length(Ps, Bag) .
```

Argument domain sizes are similarly found by estimating the number of solutions a predicate will produce if its arguments are instantiated. In this case, the size is estimated as the length of a set of solutions generated by the Prolog *setof/3* procedure when an existential quantifier (Arg^ in Prolog) is placed on other arguments :

- argument 1 instantiated :

```
domain_size(Ds) :-
        setof(A, B ^ p(A, B), Set) , length(Ds, Set) .
```

- argument 2 instantiated :

```
domain_size(Ds) :-
        setof(B,  A ^ p(A, B), Set) , length(Ds, Set) .
```

This method of generating cost estimates is not intended to be accurate but merely to produce numbers which reflect the relative costs of accessing predicates. Having found the cost of the predicate and size of each domain, a cost table for each predicate (database relation) at each stage of instantiation can now be compiled, as described earlier (costs are multiplied by ten so that integers may be used). In Chat-80, logarithms are used and the cost for any particular instantiation state calculated at run time, but there seems no reason why costs should not be calculated for each state of instantiation off-line.

For predicates with one or more arguments described as 'infinite', the calculation of cost is not possible for all states of instantiation. For a predicate with description :

```
...., p/2  type  [inf, inf ], ....
```

the cost is recorded as infinite (actually a very large number) if either or both arguments are uninstantiated, and as one if both arguments are instantiated. For a predicate with description :

```
...., p/2  type  [symbol, inf ], ....
```

the cost is recorded as infinite if the second argument is uninstantiated, and is calculated as the length of the bag of solutions, if the first argument is uninstantiated and the second instantiated, and as the length of the bag divided by the length of the set if both arguments are instantiated.

The use of sets and bags to estimate costs can be explained in the following way. A bag of solutions for any predicate will contain all the solutions that the predicate can generate (including duplicate solutions) and is thus applicable when a goal solution instantiates an uninstantiated variable. A set of solutions for any argument, with others

existentially quantified, will contain all the values the argument may take, without duplicates. This set is then used as the 'domain' of the argument.

As well as simple predicate calls to relations, a query may contain set expressions and negation. Warren uses the following heuristics to allow Chat-80 to cope with these situations :

- the cost of a set expression is taken to be the cost of its least costly internal subgoal.
- the cost of a negated goal(s) is 'infinity' if the goal(s) contain uninstantiated free variables, and otherwise is 1.

The first of these heuristics has had to be extended in order to cope with an unusual but important case of ordering set goals. Consider the case of ordering goals in the goal sequence :

```
ocean(O) &                                          (goal a)
setof(C, country(C), C_set) &                       (goal b)
numberof(Co, pick(Co, C_set) & borders(Co, O), N) . (goal c)
```

This sequence of goals might be generated by a query such as :

'What percentage of countries border each ocean ?'

The *numberof/3* predicate is costed in the same way as a set predicate as the implementation underlying *numberof/3* is :

```
numberof(Variable, Predicates, Number) :-
        setof(Variable, Predicates, Set) ,
        length(Number, Set) .
```

After executing *(goal a) ocean/1* (which has lowest cost), the *setof/3* and *numberof/3* goals must be costed and then ordered. It so happens that in the Chat-80 database, the cost of the *borders/2* predicate with one argument instantiated (as an ocean) is much less than the cost of the *country/1* predicate with an uninstantiated argument (fewer 'things' border a specific ocean than there are countries). This means that, using Warren's heuristic for sets, *(goal c)* will be executed before *(goal b)*. However, the predicate *pick/2*, it should be noted (*oneof/2* in Chat-80) is defined :

```
pick(Element, [Element | _]) .
pick(Element, [_ | Rest]) :-
          pick(Element, Rest) .
```

The (meta-data) specification (for which costs are actually pre-defined) for the *pick/2* predicate would thus be :

```
...., pick/2   type   [symbol, inf ], ....
```

In Prolog the predicate would normally be described by 'pick(-Element, +List)' where *pick/2* returns an element, non-deterministically, from a given list. This means that in the goal sequence above, *(goal b)* should be executed before *(goal c)* so that the list from which *pick/2* selects the next element is instantiated. The set heuristic can be extended to cope with this situation :

- the cost of a set expression is taken to be the cost of its least costly internal subgoal, unless the set expression contains a subgoal with an infinite cost, in which case the cost of the set expression is also infinite.

This new heuristic allows the case outlined above to be dealt with, as only after the *setof/3* subgoal has been executed will the list that *pick/2* operates on be instantiated and the cost of *pick/2* fall below infinity. Thus the new heuristic can be seen to cope with set expressions which are dependent upon one another. However, the heuristic is not able to cope with cases where set goals interact with other first order goals. An example of this will now be considered.

A more general rule can be stated which deals with the *setof/3* predicate. This predicate, in certain circumstances can be viewed as a "generator" of values for use in other goals (*cf* prover). Consider the sequence of goals produced from a query such as *'which continent contains the largest country ?'* :

```
continent(Co) &                                  (goal a)
contains(Co, Lc) &                               (goal b)
setof(C, country(C), C_set) &                    (goal c)
aggreg(largest(Lc), country, C_set, Lc)          (goal d)
```

If in the database (which is of course the case), there are fewer continents than countries, then the goals will be executed in the order shown above. This means taking each continent, finding a country in that continent and then determining if this is the largest

known country. Obviously query planning should reorder the goals so that the largest country is found first and then the continent which contains the largest country.

The required ordering can be achieved if set type goals are viewed as value generators for other goals. The *setof/3 (goal c)*, for example, contains sub-goals which do not share any variables with other goals of the query. Set type goals only function as generators when the sub-goals they contain do not share variables with other goals. The *numberof/3* goal in the previous example is thus not a generator until the set *C_set* has been instantiated, as this variable is shared with the *setof/3* goal. The heuristic for set type predicates can thus be revised (although it currently seems unnecessary given the examples above, the notion of an infinite cost due to a sub-goal with infinite cost being retained) :

- the cost of a set type goal is the cost of the least costly sub-goal within the set type goal unless, one of the sub-goals has an infinite cost, in which case the cost of the set goal is infinite, or the sub-goals do not share any variable (except the set or number 'output' variable) with other goals in the query, in which case the cost of the set goal is one (or very low).

This heuristic can be used to deal with both of the goal ordering examples illustrated above. Although this is not an aspect of goal ordering discussed by Warren [1981], it is presumed that a similar rule has been employed in the Chat-80 system The *pick/2* predicate can, in a way, also be viewed as a generator, where the cost of this goal is very low (one) if the list argument is instantiated and infinite if it is not.

8.2.2 Isolating Independent Goal Sequences

In addition to ordering subgoals, a Prolog programmer may improve the efficiency of a program by placing cuts ('!') at appropriate points in a goal sequence between goals. A cut commits the top-down backtracking search of Prolog to the current path taken in a search tree so that, if a subsequent goal fails, other alternative solutions of goals not expanded are not tried. Obviously this may greatly improve the efficiency of program execution.

In query planning, Warren observes [1981, p276] that "when two parts of a conjunction no longer share variables they should be solved independently". This

assumes that goal solutions are ground (contain no variables themselves), which is a valid assumption in the database querying context. Once one of the independent conjunctions, (goal *(b)* in the previous goal sequence) has been solved, a commitment to the solution should be made. Redoing the conjunction *(b)*, which may have many identical solutions, when a subsequent independent conjunction *(c)* fails cannot produce a new variable assignment in conjunction *(c)* which might be solvable. Consider the Chat-80 query *(q)* below :

```
(q)  answer(C) <=
         borders(C, mediterranean) &
         country(C) & borders(C, C1) &
         asian(C1) & country(C1) .
```

When this query is executed, Warren observes that :

> " some of the final solutions are separately generated more than once .... for example, the solution 'answer(turkey)' is produced four times, the reason being that Turkey borders four different Asian countries. "

Obviously such independent goal sequences should only be allowed to produce a single solution for any set of variable instantiations. This is exactly the control imposed if a Prolog cut is placed after the goal sequence. In Chat-80, independent sequences of goals are enclosed in braces so that the query *(q)*, after planning has the form :

```
answer(C) <=
     borders(C, mediterranean) &
     { country(C) } &
     { borders(C, C1) &
       { asian(C1) } &
       { country(C1) } } .
```

The braces are viewed as additional control information and executed in Prolog by an interpreter :

```
execute( { Goals } ) :- execute(Goals), ! .
```

The Chat-80 query planner introduces this control information whilst ordering goals. At each ordering stage, the remaining goals of a query are checked to determine if they may be partitioned into independent subsets, not containing common variables. Each independent part of the query is then planned separately and the separate parts ordered

according to their costs, where the cost of a part is heuristically taken as the cost of its first (least costly) goal. Division of the query also obviously simplifies the planning process itself.

Warren does not describe how independent sets of goals should be found, and it is not made clear whether his system breaks queries such as :

<goal_sequence>$_1$ & <goal_sequence>$_2$ & <goal_sequence>$_3$

where each <goal_sequence> is an independent sequence of goals, into :

{ <goal_sequence>$_1$ & <goal_sequence>$_2$ } & { <goal_sequence>$_3$ }

and then recursively breaks the first set into two parts, or whether partitioning into three sequences takes place in one step. Certainly the first recursive solution is non-trivial to program for general cases in order to produce the most effective partitioning.

In the system described here, a "dependency analysis" and "topological sort" technique [Peyton-Jones, 1987, p118] has been employed to partition a sequence of goals. In the partitioning of a query, the system :

- produces a list of elements, where each element is a list of the uninstantiated variable arguments of each successive subgoal in the goal sequence. For example, the remainder of the query *(q)* after execution of the *borders/2* predicate (which unifies *C* with *turkey*) would look like :

```
(goal 1)   country(turkey) &
(goal 2)   borders(turkey, C1) &
(goal 3)   asian(C1) &
(goal 4)   country(C1) .
```

  and the list produced at this stage would be :

  [[ ], [C1], [C1], [C1]]

- each successive element in this list is assigned a number, and then a new list is created where each element of the list contains the numbers of other elements in the list, which have at least one variable in common with each successive element of the original list. This produces a new list :

[[1], [2, 3, 4], [2, 3, 4], [2, 3, 4]]

- each element of the list is then taken in turn and each other element having a number in common with the element is added to the element. This process is performed recursively so that if there are three sublists in the list :

(a) [1, 2] *(b)* [1, 2, 3] *(c)* [3, 4, 5]

then each element of *(b)* is added to the first element *(a)* as this has number 1 (or 2) in common with element *(a)* to produce a new sublist *a1* and then element *(c)* is added to this, as *a1* and *c* have number 3 in common, to produce a single list. In the previous example case this will thus finally produce a list with two sublists :

[[1], [2, 3, 4]]

This list indicates precisely how the sequence of goals can be partitioned, in this case into two independent parts, where one part contains only goal *1* and the other goals *2*, *3* and *4* :

```
{ country(turkey) } &
{ borders(turkey, C1) & asian(C1) & country(C1) } .
```

Dependency analysis is then carried out at each subsequent stage of planning to further partition the query. Note that dependency analysis may produce up to *N* partitions of *N* goals and that partitioning is guaranteed to be maximized at each stage.

8.3 Query Execution

The query representation is executed against the Chat-80 Prolog geographical database by a simple query interpreter. The goals which appear in a query may be divided roughly into three groups :

(1) domain free predicates (quantifiers, set operators, negation).

(2) pseudo-domain predicates (aggregates, proportions).

(3) domain (database) predicates.

Domain free predicates *(1)* are all matched by the Prolog rules of the interpreter and executed according to predefined semantics which are part of the interface system. These predicates include the quantifiers (Chat-80) set operations (including indexed sets) and negation. A rule to execute the top-level 'y_n' quantifier and produce a response, for example, is

```
execute( y_n(Preds) ) :-
     execute(Preds), !, write(yes), nl
     |
     write(no) .
```

This rule, for evaluating polar questions, recursively calls for the execution of the predicates *Preds* and responds *yes* if a solution can be found to these otherwise, a negative response is made. The execution of set expressions is based not on the Prolog *setof/3* predicate but on the *findall/3* predicate. These predicates are very similar except that the *setof/3* predicate fails if no solutions can be found, whereas *findall/3* will return an empty set. As presupposition is not dealt with by the Chat-80 query representation, failure does not seem appropriate when no solution can be found. Instead an empty set is returned (if *findall/3* is used). This means that in response to a query such as *'what percentage of countries border each ocean'* (query 22 in the test corpus, Appendix D) the response will be the same as that from Chat-80 :

```
2: artic_ocean, 35:atlantic, 14:indian_ocean, 20:pacific
```

but an additional solution is produced :

```
0:southern_ocean
```

as no countries are known to border the southern ocean in the database, but this does not cause failure of query execution.

The pseudo-domain predicates (aggregations) are dealt with in much the same way as in Chat-80. These are used to represent words such as *largest* which, although domain free themselves, must be interpreted in a domain orientated manner. So *largest* in the domain of Chat-80's geographical database domain, when applied to countries, is taken as referring to the countries' areas. The typing of the entity type that an aggregation operates on is used to select between the possible interpretations it may have. These alternative definitions are actually separated out from the definition of an aggregation so that, for example, the aggregation for *largest* is defined :

```
aggreg(largest(Entity), Type, Set, Largest) :-
        def(largest(Entity), Type, Set, Attribute, Preds),
        aggregate(max(Attribute, Entity),
                    execute(Preds), max(_, Largest) ) .
```

where *aggreg/4* matches the query representation of an aggregate operator (for *largest*) with arguments (in the order they appear); the predicate name of the particular aggregation with a single argument which will be used during execution to hold each value of the entity over which the aggregation operates, the domain type of the entities, the set (list) of entities and a variable which will be used to pass out the result of the aggregation.

The predicate *def/5* is called by the *aggreg/4* predicate above in order to retrieve the domain interpretation of the particular aggregation operator. So if the aggregation *largest* is applied to entities of type country then *largest* is taken as referring to the areas for which the *def/5* entry will be:

```
def(largest(Entity), country, Countries, Area,
                    (pick(Entity, Countries) &
                                area(Entity, Area)) ).
```

This particular interpretation of *largest* (the first argument) is defined to act on entities of type country (the second argument) where the attribute of the countries (the third argument) which is to be used for the aggregation calculation is returned with a conjunction of predicates which specify how the value of this attribute is to be determined for each country in the list of countries. Given this description, the Quintus Prolog library procedure *aggregate/3* is then used to execute the predicates provided by *def/5* and find the corresponding entity for which the attribute (area) has a maximum value. Other domain type predicates are solved simply by matching against tuple values in the Prolog database taken from Chat-80.

Chapter 9

Future Development and Conclusion

Many aspects of the system have not yet been fully developed. In particular the implementation itself is quite 'sparse' in that some sections of code only have rules for dealing with the cases arising in the query corpus. However these queries do appear to have been chosen so as to cover quite a large number of the constructs which would occur in typical queries to a Prolog database and, in addition, to provide a few examples of quite complex and perhaps less common constructs. Encounters with other types of construct perhaps might produce functional analyses which would require extensions to the functional semantic translation component.

Disjunction is not currently allowed in either grammar rules (although it is used in the original definition of LFG [Kaplan & Bresnan, 1982]) or within a lexical definition (however in this system a word may have several disjunct definitions). This does not however appear to be a great weakness of the implementation. Although there has been some work [Lindert & Calder, 1987] on specifying disjunctions in lexicons, it seems to be a matter of personal preference as to whether these are desirable. Allowing even a limited disjunction in F-structures, so that the different entries for *be* were represented as a single F-structure, would greatly improve parsing efficiency. Disjunction in F-structures may be desirable for reasons other than compactness or descriptional elegance [Karttunen, 1984, p30] and is noted as a powerful component of the FUG formalism [Shieber, 1985a]. The omission of a disjunctive notational device from LFG may thus be seen as a serious weakness of the formalism.

An obvious area for extension is that of the quantifier scoping rules. These currently only make the same distinction between quantifiers as does Chat-80, in that a quantifier is either 'strong' or 'weak'. The quantifier scoping rules are currently primarily based on the function in which a quantifier appears. Little variation from these default rules has been required to deal with the query corpus and those examples presented by Colmerauer. Only in one case (query 22 in the test corpus, Appendix D) was it found necessary to invoke the exception rules which promote a strong quantifier over weak quantifiers. It may well be that the default scoping rules which relate to functions should be specialized to also take account of the inherent properties of specific

quantifiers, rather than these taking lower priority than the rules which move strong quantifiers. It has however been shown that the functional basis is at least as adequate as syntactic means for determining quantifier scoping.

The semantic theory proposed here should support the semantic interpretation of anaphoric references that the LFG component itself is at present not able to deal with. Anaphora is still a matter for research in natural language processing itself but the cases of this which might commonly occur in queries to a simple database (particularly the Chat-80 database of geographical data) could perhaps be dealt with more simply. Further work in the context of the domain of the Chat-80 database would contribute little to progress in this area. Anaphoric references could be dealt with if the antecedents of these could be reliably located. Given that this can be done in the case of a unification grammar such as LFG [Hayes, 1981; Frey, Reyle & Rohrer, 1983; Johnson & Klein, 1986; Wada & Asher, 1986], then the occurrence of an anaphoric reference can be co-indexed with its antecedent and then translated using indexes and shared variable features in much the same way as functions subject to constituent control.

Ellipses could perhaps be introduced in a limited way. These were dealt with extensively using semantic grammars in the LIFER system [Hendrix, 1977; Hendrix, Sacerdoti & Slocum, 1978] but are more difficult to deal with in transportable systems. A basic approach to handling ellipses in LFG was outlined in Section 4.9.

Another area in which the current system could be extended is in the use of lexical rules, with associated morphological changes, which can be used to generate large lexicons with a greater coverage from a few initial basic entries [Flickinger, Pollard & Wasow, 1985]. Lexical rules can be used to transform for example verb root entries of regular verbs into a number of verb entries for each tense, person and number. They may also be used to specify default values. Such rules and defaults can be added to the system to manipulate the input to the lexicon pre-processor and can thus operate at the level of the LFG notation.

Essentially future extensions to the implementation can be divided into three parts: improvement and extension of the implementation itself; alterations and additions to LFG itself in order to improve efficiency, and further extension and verification of the semantic interpretation principles.

The LFG formalism has been implemented in a way that allows both grammar and lexicon to be written in a notation very similar to LFG itself and at the same descriptive level as LFG. Many of the formalism's notational conveniences and mechanisms have been implemented (Kleene-star, long-distance dependencies, gaps, literals, constraints and sets). To do this, it has been necessary to move away from a direct use of both Prolog's TD control and unification and employ a specialized version of the active Chart parser which operates in a methodical bottom-up fashion, here called word-incorporation (WI).

The suitability of Prolog's control for parsing has been made evident by its effective use in Chat-80 but its direct use in a parser for LFG results in a great loss of felicity. WI has proved quite effective and efficient as a BU parsing algorithm developing alternative parses in parallel and allowing parsing to be practical with quite a large and 'arbitrary' LFG grammar (ie not specifically engineered for efficiency). WI also supports left-recursion in grammar rules. This has allowed the creation of a high-level, felicitous grammar system.

The performance of the parser, whilst currently much lower than that of the Chat-80 system, could be improved greatly by reducing memory usage during parsing. Several techniques for achieving this have been outlined (Section 7.7). It should also be noted that in the face of a larger grammar (with less determinism) and greater lexical ambiguity, a TD parsing action can be expected to deteriorate by a much greater degree than a BU parsing action.

Several alterations and additions (here and by others) to the LFG formalism have been discussed. These have mainly been concerned with the addition of constraints to LFG in order to improve the grammar's parsability. In particular it has been found desirable to support some degree of completeness checking during parsing. Whilst coherence can be naturally incorporated into parsing (when constructing C-structure and F-structure incrementally), completeness cannot be ensured until after parsing is completed. This is viewed as a distinct computational weakness of LFG itself. Additional language dependent constraints, such as subject-verb-object ordering might be incorporated in the formalism to aid completeness checking but this is complicated by the movement of functions (when produced by functional assignment equations in C-structure).

Although LFG is a clear, high-level formalism the slot-filling type analysis required for the production of Prolog queries sometimes conflicts with, or greatly complicates, expression in LFG. In particular the completeness and coherency conditions often require rather strange analyses be employed in order to simultaneously satisfy these constraints and fill slots correctly. The analyses used here are an attempt to reflect not only these requirements but those of the linguists who have worked to develop LFG itself. Also an account of *be* as main verb (unifying the LFG and TG accounts) has been proposed. This satisfies both linguistic and logical requirements of analysis.

A new method of semantically interpreting the F-structure output of LFG has been outlined which uses only the F-structure component of LFG, as intended in the original definition of LFG by Kaplan and Bresnan [1982]. The F-structure is however augmented with semantic features but these fit naturally into the basic concepts and notation of the formalism. The problem of quantification has been approached on an essentially functional basis and, whilst this development is still embryonic, it has been developed sufficiently to deal with the Chat-80 query corpus and the examples described by Colmerauer [1982]. The success of this approach, it is hoped, will further serve to strengthen evidence for the basis of functional linguistic analysis.

References

Barr, A. and Feigenbaum, E.A. *eds.* (1981). *The handbook of artificial intelligence: vol 1.* London: Pitman.

Bates, M. (1978). The theory and practice of Augmented Transition Network Grammars. *In :*
Bolc, L. *ed.* (1978). *Natural language communication with computers. Berlin: Springer.* pp 191-259.

Berghel, H. and Traudt, E. (1986). Spelling verification in Prolog. *ACM SIGPLAN Notices 21*(1). pp 19-27.

Berwick, R.C. (1981). Computational complexity and lexical functional grammar. *In :*
Proceedings of the 19th Annual Meeting of the Association for Computational Linguistics, Stanford, 29 June-1 July 1981. [s.l.]: ACL. pp 7-12.

Block, H.U. and Haugeneder, H. (1986). The treatment of movement-rules in a LFG-parser. *In :*
Proceedings of the 11th International Conference on Computational Linguistics and the 24th Annual Meeting of the Association for Computational Linguistics, Bonn, 25-29 August 1986. [s.l.]: ACL. pp 482-486.

Block, H.U. and Hunze, R. (1986). Incremental construction of C- and F-structure in a LFG-parser. *In :*
Proceedings of the 11th International Conference on Computational Linguistics and the 24th Annual Meeting of the Association for Computational Linguistics, Bonn, 25-29 August 1986. [s.l.]: ACL. pp 490-493.

Bourne, S.R. (1982). *The UNIX system.* Wokingham: Addison-Wesley.

Brand, D. (1986). On typing in Prolog. *ACM SIGPLAN Notices 21*(1). pp 28-30.

Bratko, I. (1986). *Prolog programming for artificial intelligence.* Wokingham: Addision-Wesley.

Bresnan, J. *et al.* (1982). Cross-serial dependencies in Dutch. *Linguistic Inquiry 13*(4). pp 613-635.

Burton-Roberts, N. (1986). *Analysing sentences: an introduction to English syntax.* New York: Longman.

Chomsky, N. (1959). On certain formal properties of grammars. *Information and control 2*(1). pp 137-167.

Clocksin, W.F. and Mellish, C.S. (1984). *Programming in Prolog*. 2nd edition. Berlin: Springer.

Colmerauer, A. (1982). An interesting subset of natural language. *In :* Clark, K.L. and Tärnlund, S.A. *eds*. (1982). *Logic programming*. New York: Academic Press. pp 45-66.

Dahl, V. (1979). Quantification in a three-valued logic for natural language question-answering systems. *In :*
Proceedings of the Sixth International Joint Conference on Artificial Intelligence, Tokyo, 20-23 August 1979. Los Altos: Morgan Kaufmann. pp 182-187.

Dahl, V. (1981). Translating Spanish into logic through logic. *American Journal of Computational Linguistics 7*(3). pp 149-164.

Dahl, V. and McCord, M.C. (1983). Treating coordination in Logic Grammars. *American Journal of Computational Linguistics 9*(2). pp 69-91.

De, S., Pan, S.S. and Whinston, A.B. (1985). Natural language query processing in a temporal database. *Data and Knowledge Engineering 1*(1). pp 3-15.

Earley, J. (1970). An efficient context-free parsing algorithm. *Communications of the Association for Computing Machinery 13*(2). pp 94-102.

Eisele, A. (1984). *A Lexical-Functional Grammar system in Prolog*. Stuttgart: Department of Linguistics, University of Stuttgart. (Working paper).

Eisele, A. and Dörre, J. (1984). A Lexical-Functional Grammar system in Prolog. *In :*
Proceedings of the 11th International Conference on Computational Linguistics and the 24th Annual Meeting of the Association for Computational Linguistics, Bonn, 25-29 August 1986. [s.l.]: ACL. pp 551-553.

Falk, Y.N. (1983). Subjects and long-distance dependencies. *Linguistic Analysis 12*(3). pp 245-270.

Falk, Y.N. (1984). The English auxiliary system: a Lexical-Functional analysis. *Language 60*(3). pp 483-509.

Flickinger, D., Pollard, C. and Wasow, T. (1985). Structure sharing in lexical representation. *In :*
Proceedings of the 23rd Annual Meeting of the Association for Computational Linguistics, Chicago, 8-12 July 1985. [s.l.]: ACL. pp 262-267.

Frey, W. (1985). Noun phrases in Lexical-Functional Grammar. *In :* Dahl, V. and Saint-Dizier, P. *eds.* (1985). *Natural language understanding and logic programming: proceedings of the First International Workshop on Natural Language Understanding and Logic Programming, Rennes, 18-20 September 1984.* Amsterdam: Elsevier Science. pp 121-137.

Frey, W. and Reyle, U. (1983). A Prolog implementation of Lexical-Functional Grammar as a base for a natural language processing system. *In :*
Proceedings of the First Conference of the European Chapter of the Association for Computational Linguistics, Pisa, 1-2 September 1983. [s.l.]: ACL. pp 52-57.

Frey, W., Reyle, U. and Rohrer, C. (1983). Automatic construction of a knowledge base by analysing texts in natural language. *In :*
Proceedings of the Eighth International Joint Conference on Artificial Intelligence, Karlsruhe, 8-12 August 1983. Los Altos: Morgan Kaufmann. pp 727-729.

Gazdar, G. *et al.* (1985). *Generalised Phrase Structure Grammar.* Oxford: Blackwell.

Grishman, R., Hirschman, L. and Friedman, C. (1983). Isolating domain dependencies in natural language interfaces. *In :*
Proceedings of the Conference on Applied Natural Language Processing, Santa Monica, 1-3 February 1983. [s.l.]: ACL. pp 46-53.

Grosz, B.J. (1982). Transportable natural language interfaces: problems and techniques. *In :*
Proceedings of the 20th Annual Meeting of the Association for Computational Linguistics, Toronto, 16-18 June 1982. [s.l.]: ACL. pp 46-50.

Grosz, B.J. (1983). TEAM: a transportable natural language interface system. *In :*
Proceedings of the Conference on Applied Natural Language Processing, Santa Monica, 1-3 February 1983. [s.l.]: ACL. pp 39-45.

Hafner, C.D. (1985). Semantics of temporal queries and temporal data. *In :*
Proceedings of the 23rd Annual Meeting of the Association for Computational Linguistics, Chicago, 8-12 July 1985. [s.l.]: ACL. pp 1-8.

Halvorsen, P.K. (1983). Semantics for Lexical-Functional Grammar. *Linguistic Inquiry 14*(4). pp 567-615.

Hayes, P.J. (1981). Anaphora for limited domain systems. *In :*
Proceedings of the Seventh International Joint Conference on Artificial Intelligence, Vancouver, 24-28 August 1981. Los Altos: Morgan Kaufmann. pp 416-422.

Hendrix, G.G. (1977). Human engineering for applied natural language processing. *In :*
Proceedings of the Fifth International Joint Conference Artificial Intelligence, Cambridge, Mass., 22-25 August 1977. Los Altos: Morgan Kaufmann. pp 183-191.

Hendrix, G.G. and Lewis, W. (1981). Transportable natural-language interfaces to databases. *In :*
Proceedings of the 19th Annual Meeting of the Association for Computational Linguistics, Stanford, 29 June-1 July 1981. [s.l.]: ACL. pp 159-165.

Hendrix, G.G., Sacerdoti, D.S. and Slocum, J. (1978). Developing a natural language interface to complex data. *ACM Transactions on Database Systems 3*(2). pp 105-147.

Hewitt, C. (1977). Viewing control structures as patterns of passing messages. *Artificial Intelligence 8*(3). pp 323-364.

Horsfall, H.J. (1986). An interactive English-Japanese machine translation system. *Presented at :*
International Conference on Machine and Machine-Aided Translation, Birmingham, 7-9 April 1986.

Ioup, G. (1975). Some universals for quantifier scope. *In :*
Kimball, J.P. *ed.* (1975). *Syntax and Semantics: vol 4.* London: Academic Press. pp 37-58.

Jacobsen, B. (1978). *Transformational-Generative Grammar: an introductory survey of its genesis and development.* 2nd edition. Oxford: North Holland.

Jacobsen, B. (1986). *Modern Transformational Grammar, with particular reference to the theory of Government and Binding.* Oxford: North Holland.

Johnson, M. and Klein, E. (1986). Discourse, anaphora, and parsing. *In :*
Proceedings of the 11th International Conference on Computational Linguistics and the 24th Annual Meeting of the Association for Computational Linguistics, Bonn, 25-29 August 1986. [s.l.]: ACL. pp 669-675.

Kamp, H. (1981). A theory of truth and semantic representation. *In :*
Groenendijk, J.A.G., Janssen, T.M.V. and Stokhof, M.B.J. *eds.* (1981). *Formal methods in the study of language.* Amsterdam: Mathematical Centre. (Mathematical centre tracts; 135). pp 277-322.

Kaplan, R.M. and Bresnan, J. (1982). Lexical-Functional Grammar: a formal system for grammatical representation. *In :*
Bresnan, J. *ed.* (1982). *The mental representation of grammatical relations.* Cambridge, Mass.: MIT Press. pp 173-281.

Karttunen, L. (1984). Features and values. *In :*
Proceedings of the 10th International Conference on Computational Linguistics and the 22nd Annual Meeting of the Association for Computational Linguistics, Stanford, 2-6 July 1984. [s.l.]: ACL. pp 28-33.

Kay, M. (1979). Functional grammar. *In :*
Proceedings of the Fifth Annual Meeting of the Berkeley Linguistics Society, Berkeley, 17-19 February 1979. [s.l.: s.n.]. pp 142-158.
Kay, M. (1984). Functional unification grammar: a formalism for machine translation. *In :*
Proceedings of the 10th International Conference on Computational Linguistics and the 22nd Annual Meeting of the Association for Computational Linguistics, Stanford, 2-6 July 1984. [s.l.]: ACL. pp 75-78.

Kay, M. (1985a). Parsing in functional unification grammar. *In :*
Dowty, D.R., Karttunen, L. and Zwicky, A.M. *eds.* (1985). *Natural language parsing: psychological, computational and theoretical perspectives.* Cambridge: Cambridge University Press. pp 251-278.

Kay, M. (1985b). Unification in grammar. *In :*
Dahl, V. and Saint-Dizier, P. *eds.* (1985). *Natural language understanding and logic programming: proceedings of the First International Workshop on Natural Language Understanding and Logic Programming, Rennes, 18-20 September 1984.* Amsterdam: Elsevier Science. pp 233-241.

King, M. (1983). Transformational parsing. *In :*
King, M. *ed.* (1983). *Parsing natural language: proceedings of the Second Lugano Tutorial, Lugano, 6-11 July.1981.* London: Academic Press, pp 19-34.

Kudo, I. and Nomura, H. (1986). Lexical-Functional transfer: a transfer framework in a machine translation system based on LFG. *In :*
Proceedings of the 11th International Conference on Computational Linguistics and the 24th Annual Meeting of the Association for Computational Linguistics, Bonn, 25-29 August 1986. [s.l.]: ACL. pp 112-114.

Lindert, R. te. and Calder, J. (1987). Towards a high-level language for lexical description. *In :*
Lindert, R. te. and Calder, J. (1987). *Natural language processing, unification and grammar formalisms: proceedings of an Alvey/SERC sponsored workshop, Stirling, 10-12 June 1987.* [s.l.]: Alvey. pp 66-67.

Matsumoto, Y. and Tanaka, H. (1983). BUP: a bottom-up parser embedded in Prolog. *New Generation Computing 1*(2). pp 145-158.

Matsumoto, Y., Kiyono, M. and Tanaka, H. (1985). Facilities of the BUP parsing system. *In :*
Dahl, V. and Saint-Dizier, P. *eds.* (1985). *Natural language understanding and logic programming: proceedings of the First International Workshop on Natural Language Understanding and Logic Programming, Rennes, 18-20 September 1984.* Amsterdam: Elsevier Science. pp 97-106.

McCord, M.C. (1980). Slot Grammars. *American Journal of Computational Linguistics 6*(1). pp 31-43.

McCord, M.C. (1982). Using slots and modifiers in Logic Grammars for natural language. *Artificial Intelligence 18*(3). pp 327-367.

McCord, M.C. (1985a). Focalizers, the scoping problem and semantic interpretation rules in Logic Grammars. *In :*
Van Caneghem, M. and Warren, D.H.D. *eds.* (1985). *Logic programming and its applications*. New Jersey: Ablex. pp 223-243.

McCord, M.C. (1985a). Modular Logic Grammars. *In :*
Proceedings of the 23rd Annual Meeting of the Association for Computational Linguistics, Chicago, 8-12 July 1985. [s.l.]: ACL. pp 104-117.

McCord, M.C. (1987). *Natural language processing in Prolog. In :*
Walker, A. *ed.* (1987). *Knowledge systems and Prolog: a logical approach to expert systems and natural language processing*. Wokingham: Addison-Wesley. pp 291-402.

Momma, S. (1987). Generation from F-structures. *In :*
Lindert, R. te. and Calder, J. (1987). *Natural language processing, unification and grammar formalisms: proceedings of an Alvey/SERC sponsored workshop, Stirling, 10-12 June 1987*. [s.l.]: Alvey. pp 36-40.

Netter, K. (1986). Getting things out of order: an LFG proposal for the treatment of German word order. *In :*
Proceedings of the 11th International Conference on Computational Linguistics and the 24th Annual Meeting of the Association for Computational Linguistics, Bonn, 25-29 August 1986. [s.l.]: ACL. pp 494-496.

Netter, K. and Wedekind, J. (1986). An LFG-based approach to machine translation. *In :*
Proceedings of IAI-MT86: International Conference on the State of the Art in Machine Translation in America, Asia and Europe, Dudweiler, 20-22 August 1986. pp 197-209.

Pereira, F.C.N. (1981). Extraposition Grammars. *American Journal of Computational Linguistics 7*(4). pp 253-256.

Pereira, F.C.N. (1982). *Logic for natural language analysis*. Unpublished PhD thesis, Department of Artificial Intelligence, University of Edinburgh.

Pereira, F.C.N. (1985). A structure-sharing representation for unification-based grammar formalisms. *In :*
Proceedings of the 23rd Annual Meeting of the Association for Computational Linguistics, Chicago, 8-12 July 1985. [s.l.]: ACL. pp 137-144.

Pereira, L.M., Pereira, F.C.N. and Warren, D.H.D. (1978). *User's guide to DEC System-10 Prolog*. Department of Artificial Intelligence, University of Edinburgh. (DAI research paper; 154).

Pereira, F.C.N. and Warren, D.H.D. (1980). Definite clause grammars for language analysis: a survey of the formalism and a comparison with augmented transition networks. *Artificial Intelligence 13*(3). pp 231-278.

Pereira, F.C.N. and Warren, D.H.D. (1983). Parsing as deduction. *In : Proceedings of the 21st Annual Meeting of the Association for Computational Linguistics, Cambridge, Mass., 15-17 June 1983*. [s.l.]: ACL. pp 137-144.

Peyton-Jones, S.L. (1987). *The implementation of functional programming languages*. London: Prentice-Hall.

Phillips, J.D. (1986). A simple, efficient parser for phrase-structure grammars. *AISB Quarterly* (59). pp 14-17.

Pinker, S. (1982). A theory of the acquisition of Lexical Interpretive Grammars. *In :*
Bresnan, J. *ed.* (1982). *The mental representation of grammatical relations*. Cambridge, Mass.: MIT Press. pp 655-726.

Quintus Computer Systems Inc. (1987). *Quintus Prolog user's guide and reference manual (V2.0)*. Mountain View: Quintus Computer Systems.

Radford, A. (1981). *Transformational syntax: a student's guide to Chomsky's Extended Standard Theory*. Cambridge: Cambridge University Press.

Reyle, U. (1985). Grammatical functions, discourse referents and quantification. *In :*
Proceedings of the Ninth International Joint Conference on Artificial Intelligence, Los Angeles, 19-23 August 1985. Los Altos: Morgan Kaufmann. pp 829-831.

Reyle, U. and Frey, W. (1983). A Prolog implementation of Lexical-Functional Grammar. *In :*
Proceedings of the Eighth International Joint Conference on Artificial Intelligence, Karlsruhe, 8-12 August 1983. Los Altos: Morgan Kaufmann. pp 693-695.

Sedogbo, C. and Bull, G. (1985). A meta-grammar for handling coordination in logic grammars. *In :*
Dahl, V. and Saint-Dizier, P. *eds.* (1985). *Natural language understanding and logic programming: proceedings of the First International Workshop on Natural Language Understanding and Logic Programming, Rennes, 18-20 September 1984*. Amsterdam: Elsevier Science. pp 153-163.

Sells, P. (1985). *Lectures on contemporary syntactic theories: an introduction to Government Binding theory, Generalised Phrase Structure Grammar and Lexical-Functional Grammar*. Stanford: CSLI. (Center for the Study of Language and Information, lecture notes; 3).
- Chapter 4, Lexical-Functional Grammar (pp 135-191).

Sharman, R.A. (1987). A Prolog-based notation for developing unification grammars. *In :*
Lindert, R. te. and Calder, J. (1987). *Natural language processing, unification and grammar formalisms: proceedings of an Alvey/SERC sponsored workshop, Stirling, 10-12 June 1987*. [s.l.]: Alvey. pp 19-23.

Shieber, S.M. (1984). The design of a computer language for linguistic information. *In :*
Proceedings of the 10th International Conference on Computational Linguistics and the 22nd Annual Meeting of the Association for Computational Linguistics, Stanford, 2-6 July 1984. [s.l.]: ACL.
pp 362-366.

Shieber, S.M. (1985a). *An introduction to unification-based approaches to grammar*. Stanford: CSLI. (Center for the Study of Language and Information, lecture notes; 4).
Note: Originally supplement to: *An introduction to unification-based approaches to grammar*, presented as a tutorial session at: *23rd Annual Meeting of the Association for Computational Linguistics, Chicago, 8-12 July 1985*.

Shieber, S.M. (1985b). Using restriction to extend parsing algorithms for complex-feature-based formalisms. *In :*
Proceedings of the 23rd Annual Meeting of the Association for Computational Linguistics, Chicago, 8-12 July 1985. [s.l.]: ACL. pp 145-152.

Shieber, S.M. *et al.* (1983). The formalism and implementation of PATR-II. *In :*
Grosz, B. and Stickel, M. *eds.* (1983). *Research on interactive acquisition and use of knowledge: final report*. Menlo Park: SRI International. pp 63-79. (AD-A137 436).

Uehara, K. *et al.* (1984a). A bottom-up parser based on predicate logic: a survey of the formalism and its implementation technique. *In :*
Proceedings of the International IEEE Conference on Logic Programming, Atlantic City, 6-9 February 1984. Los Angeles: IEEE Computer Society.
pp 220-227.

Uehara, K. *et al.* (1984b). Steps toward an actor-orientated parser. *In :*
Fifth generation computer systems: proceedings of the International Conference on Fifth Generation Computer Systems 1984, Tokyo, 6-9 November 1984. Amsterdam: North Holland. pp 660-668.

Uehara, K. *et al.* (1985). An integrated parser for text understanding: viewing parsing as passing messages among actors. *In :*
Dahl, V. and Saint-Dizier, P. *eds.* (1985). *Natural language understanding and logic programming: proceedings of the First International Workshop on Natural Language Understanding and Logic Programming, Rennes, 18-20 September 1984.* Amsterdam: Elsevier Science. pp 79-95.

Uszkoreit, H. (1986). Categorial Unification Grammars. *In :*
Proceedings of the 11th International Conference on Computational Linguistics and the 24th Annual Meeting of the Association for Computational Linguistics, Bonn, 25-29 August 1986. [s.l.]: ACL. pp 187-194.

Varile, G.B. (1983). Charts: a data structure for parsing. *In :*
King, M. *ed.* (1983). *Parsing natural language: proceedings of the Second Lugano Tutorial, Lugano, 6-11 July 1981.* London: Academic Press, pp 73-87.

Wada, H. and Asher, N. (1986). BUILDRS: an implementation of DR theory and LFG. *In :*
Proceedings of the 11th International Conference on Computational Linguistics and the 24th Annual Meeting of the Association for Computational Linguistics, Bonn, 25-29 August 1986. [s.l.]: ACL. pp 540-545.

Waltz, D.L. and Goodman, B.A. (1977). Writing a natural language data base system. *In :*
Proceedings of the Fifth International Joint Conference Artificial Intelligence, Cambridge, Mass., 22-25 August 1977. Los Altos: Morgan Kaufmann. pp 144-150.

Waltz, D.L. *et al.* (1976). *The PLANES system: natural language access to a large database.* Urbana: University of Illinois. (Report; T-34. AD-A028 316).

Warren, D.H.D. (1981). Efficient processing of interactive database queries expressed in logic. *In :*
Pirotte, A and Vassiliou, Y. eds. (1985). *Proceedings of the 11th International Conference on Very Large Data Bases, Stockholm, 21-23 August 1985.* [s.l.: s.n.]. pp 272-281.

Warren, D.H.D. (1982). Higher-order extensions to Prolog: are they needed? *In :*
Hayes, J.E., Michie, D. and Pao, Y.H. *eds.* (1982). *Machine intelligence: 10.* Chichester: Ellis Horwood. pp 441-454.

Warren, D.H.D and Pereira, F.C.N. (1982). An efficient easily adaptable system for interpreting NL queries. *American Journal of Computational Linguistics 8*(3-4). pp 110-122.

Wedekind, J. (1986). A concept of derivation for LFG. *In :*
Proceedings of the 11th International Conference on Computational Linguistics and the 24th Annual Meeting of the Association for Computational Linguistics, Bonn, 25-29 August 1986. [s.l.]: ACL. pp 487-489.

Winograd, T. (1983). *Language as a cognitive process: volume 1, syntax.* Reading, Mass.: Addison-Wesley.

Woods, W.A. (1970). Transition Network Grammars for natural language analysis. *Communications of the Association for Computing Machinery 13*(10). pp 591-606.

Yasukawa, H. (1984). LFG system in Prolog. *In : Proceedings of the 10th International Conference on Computational Linguistics and the 22nd Annual Meeting of the Association for Computational Linguistics, Stanford, 2-6 July 1984*. [s.l.]: ACL. pp 358-361.

Zeevat, H., Klein, E. and Calder, J. (1987). Unification Categorial Grammar. *In :*
Haddock, N., Klein, E. and Morrill, G. *eds*. (1987). *Categorial grammar, unification grammar and parsing*. Edinburgh: Cognitive Science Centre, University of Edinburgh. (Working papers in cognitive science; 1).

Appendix A

LFG Notation Summary

A.1 Grammar Notation

A grammar rule has the general form :

$$\left(\begin{array}{c}\text{LHS}\\ \text{category}\end{array}\right) \longrightarrow \left(\begin{array}{c}\text{RHS}\\ \text{description}\end{array}\right)_1 \left(\begin{array}{c}\text{RHS}\\ \text{description}\end{array}\right)_2 \cdots \left(\begin{array}{c}\text{RHS}\\ \text{description}\end{array}\right)_n$$

where LHS category is a simple single non-terminal grammatical category which re-writes to an ordered sequence of *n* RHS descriptions. Each RHS description consists of either :

(1) a grammatical category (terminal or non-terminal) annotated with a non-empty set of equations (placed below the category).

(2) a literal (non-lexical terminal) annotated with a (possibly empty) set of equations (placed below the literal).

In addition to an equation set a RHS description may be subject to additional annotation :

$\boxed{\text{np}}$	Bounding node (here a np category), not applicable to literals.
$\left(\begin{array}{c}\text{np}\\ \downarrow{=}\uparrow\end{array}\right)$	Optionality of description (one or zero occurrences).
np*	Kleene-star operator (zero or more repetitions of description).
$\left\{\begin{array}{l}\text{description(s)}_1\\ \text{description(s)}_2\\ \cdots\\ \text{description(s)}_n\end{array}\right\}$	Disjunction (braces) can be used to denote alternative RHS descriptions or sequences of these (placed one on top of another).

Equations are of several types (outlined below) and contain the following elementary components (metavariables and operators) :

$\downarrow$	Immediate domination metavariable (refers to F-structure below RHS description).
$\uparrow$	Immediate domination metavariable (refers to F-structure above RHS description).

$\Downarrow^{cat1}_{cat2}$	Bounded domination metavariable (controller) belonging to domain root (RHS description) with category cat1, marked with C-structure category cat2.
$\Uparrow_{cat2}$	Bounded domination metavariable (controllee), marked with C-structure category cat2 (matches controller above).
=	Equality ('realised by unification').
$=_c$	Constraint.
¬	Negation (of equation).
∈	Set inclusion.

An equation set consists of an unordered set of equations placed one-per-line below the grammatical category or literal in a RHS description. Grammar equations may in general refer to either the F-structure above (↑) or that below (↓). As examples of each grammar equation type :

↑ = ↓	trivial (head) : passes F-structure from below description to that above.
(↑ subj) = ↓	defining (or assignment) : the example here unifies the F-structure from below with the subj function (path value) of that above.
(↑ tense)	existential constraint : ensures that the F-structure above has some value for the attribute tense.
¬ (↑ tense)	negative existential constraint : ensures that the F-structure above has no value for tense.
(↑ num) $=_c$ pl	value constraint : ensures that the F-structure above has the value pl for num (number).
¬ (↑ num) $=_c$ pl	negative value constraint : ensures that the F-structure above does not have the value pl for attribute num.
(↑(↓ pcase)) = ↓	cyclic equation : takes the value of pcase from the F-structure below and assigns the F-structure from below as the value of the pcase value in the F-structure above.
⇑ = ⇓	linking equation (by-pass bounding node) : allows a single controller from above a bounding node to penetrate the domain below a bounding node (taking a TD view of parsing).
↓ ∈ (↑ adjuncts)	set inclusion : assigns the F-structure from below as a member of the adjunct set of the F-structure above.

A.2 Lexicon Notation

A lexical entry has the general form :

(word) : (cat) (equations)

for example :

a	:	det	(↑ det) = a (↑ num) = sg

where 'word' is the lexical item, 'cat' its grammatical category and 'equations' an unordered set of equations. A lexical equation can obviously only refer to the F-structure above (↑). Only certain equation types are appropriate to lexical entries (linking, cyclic and set inclusion are those not appropriate) and in addition lexical entries may have certain other equation types (outlined below).

In addition to allowing the subset of grammar equations (outlined above) lexical entries allow semantic forms (pred attribute specifications) which have the general form :

(↑ pred) = 'pred_name sub_cat_list'

where 'pred_name' is the semantic form's predication and 'sub_cat_list' is a list of references (using immediate domination metavariables and path names) to subcategorized functions in the F-structure above (ommitted if there are none). For example :

man	:	n	(↑ pred) = 'man'
persuaded	:	v	(↑ pred) = 'persuade⟨(↑ subj) (↑ obj) (↑ vcomp)⟩'