THE UNIVERSITY *of* EDINBURGH

LOGIC FOR NATURAL LANGUAGE ANALYSIS

Fernando C. N. Pereira

Ph. D.
University of Edinburgh
1982

Table of Contents

List of Figures

ABSTRACT

This work investigates the use of formal logic as a practical tool for describing the syntax and semantics of a subset of English, and building a computer program to answer data base queries expressed in that subset.

To achieve an intimate connection between logical descriptions and computer programs, all the descriptions given are in the definite clause subset of the predicate calculus, which is the basis of the programming language Prolog. The logical descriptions run directly as efficient Prolog programs.

Three aspects of the use of logic in natural language analysis are covered: formal representation of syntactic rules by means of a grammar formalism based on logic, extraposition grammars; formal semantics for the chosen English subset, appropriate for data base queries; informal semantic and pragmatic rules to translate analysed sentences into their formal semantics.

On these three aspects, the work improves and extends earlier work by Colmerauer and others, where the use of computational logic in language analysis was first introduced.

Chapter 1

Introduction

The title of this work is a paraphrase of Kowalski's "Logic for Problem Solving" [Kowalski 80]. The choice of title reflects the aim of the work: to apply formal logic to the analysis of natural language. As in "Logic for Problem Solving", logic is seen first as a tool to build formal theories, and computer programs, for some domain, and only secondarily as the object of study. Thus I attempt here to build logical descriptions of certain linguistic phenomena relevant to computational linguistics. These logical descriptions are not just theoretical devices, but also very efficient computer programs for language analysis.

The identification of formal logical theories with computer programs comes from an identification of computation with suitably controlled deduction. However, general deductive procedures for full first-order logic have a computational complexity that precludes their direct use as computational engines. To be able to pursue the identification of computation and deduction, we can narrow the class of logical theories equated with programs. In particular, we can restrict the logical language we use. One such subset of the predicate calculus is the language of definite or Horn clauses, discussed in detail in the next chapter. Informally, definite

clauses are those first-order formulae which can be read as "P if Q and R and ...", where P, Q, R are atomic formulae. As we will see, inference in definite clauses takes a particularly simple form, similar to computation in a programming language. In fact, theoretical considerations also show that the language of definite clauses is privileged with respect to the identification of deduction and computation [van Emden and Kowalski 76]. Definite clauses are used in most of the examples in "Logic for Problem Solving", and will be used in the present work.

The programming language Prolog [Roussel 75] is a realisation of definite clauses as a programming language, which makes it possible to use as computer programs the logical descriptions given in this work. In fact, those descriptions are just abstractions, for explanatory purposes, of parts of the program text of Chat-80, an efficient computer program I developed with David Warren as a prototype natural language query system for databases expressed in logic.

1.1 Overview and Justification

This work is a contribution to the investigation of computational logic as a tool for describing the syntax and semantics of subsets of natural language and using such descriptions to build practical database query systems. On the theoretical side, I propose a grammar formalism based on logic, extraposition grammars, and then examine some grammatical questions, in particular the structure of relative and interrogative clauses and ambiguities in the placement of noun and verb complements, in the light of the new formalism. On a more

practical note, I investigate how the analysis of a sentence produced by an extraposition grammar can be translated into expressions of a computationally motivated logical system, definite closed-world clauses, adequate for database retrieval. Finally, I outline the application of these ideas to a prototype database front-end, Chat-80.

The first major investigation of computational logic in natural language analysis was done by Colmerauer [Colmerauer 78] in Marseille, using ideas from his earlier grammar formalism, Q-systems [Colmerauer 70], and the concepts of logic programming developed by Kowalski [van Emden and Kowalski 76].

The Marseille work [Colmerauer 79a, Dahl 77, Pique 81, Pasero 73, Sabatier 80] covers two distinct areas of application of logic to language analysis: expressing grammar in formal logic, and using logic to represent the meaning of natural language sentences. As I noted before, and elaborate later (see Section 1.1.2), this use of logic is related to but distinct from that in philosophy and linguistics.

In the Marseille work, the identification of computation and deduction leads to two complementary ways of using a logical theory, in grammar and in meaning representation: in grammar, as a formal grammar and as a parser, in meaning representation, as a semantic representation and as a question-answering (or information assimilation) program.

The relation of the present work to that from Marseille, and how it

extends it, will be detailed in Section 1.3 below. Before that, I will give some arguments for the use of logic in grammar and meaning representation.

1.1.1 Logic and Formal Grammar

The intimate connection between formal grammars and definite clause logic was first brought out by Colmerauer, in his article "Grammaires de Metamorphose" [Colmerauer 78]. I have discussed in detail the advantages of grammar formalisms based on definite clauses in an article with Warren, [Pereira and Warren 80], so I will limit myself here to some general remarks.

Until recently, systems of formal grammar could be classified into two categories: those that were precisely defined and whose formal properties are well understood, and those that were used to describe the syntax of natural languages. This contrast was not accidental. The best understood grammars are the context-free grammars, and it was accepted that natural languages exhibit context-sensitive phenomena that of course cannot be described in a context-free grammar.

On the other hand, systems of grammar for natural language tend not to be based in uniform precise formal notions. For instance, transformational grammars are a combination of some formal rules, normally the context-free base rules, and more or less precisely stated informal rules, the transformations (good examples of this kind of grammar can be found in "The Major Syntactic Structures of English" [Stockwell et al. 73]). Also, syntactic processors used in

computational linguistics are complex imperative programs, that can hardly classify as independently useful descriptions of syntactic concepts [Woods et al. 72, Sidner et al. 81].

In contrast, grammar formalisms based on the predicate calculus have no difficulty in representing context-sensitive notions [Colmerauer 78, Pereira and Warren 80], and have rigorous foundations that can be used to investigate their formal properties. Parsing a sentence is now equivalent to proving that a certain formula follows from the axioms comprising the grammar. Furthermore, if we restrict our attention to systems based on the definite clause subset of the predicate calculus, the connection between logic and grammar goes even further, because theorem proving algorithms for definite clauses are no more than generalisations of context-free parsing algorithms [Kowalski 80]. In this way, a grammar formalism capable of describing any recursively enumerable language can still be used for parsing very much in the same way as a context-free grammar. This contrasts with context-sensitive grammar formalisms such as type-0 grammars, transformational grammars and 2-level grammars [van Wijngaarden 75, Marcotty et al. 76], whose relation to context-free parsing algorithms is much less direct.

Finally, grammar formalisms based on definite clauses can be precisely constrained to deal with smaller classes of languages than the full recursively enumerable class. In particular, if we adhere to linguistic theories which postulate that natural languages are really context-free, and that context-sensitive notions are just artifacts of our syntactic formalisms [Gazdar 79a], we can describe precisely

the class of acceptable formalisms based on definite clauses: those where the underlying definite clauses do not contain any function symbols, with the possible exception of the occurrences of the string constructor function symbol in the definition of the predicate that axiomatises the decomposition of a string into words.

1.1.2 Logic and Meaning Representation

To justify the use of computational logic for meaning representation, we have to discuss in the first place the purposes of analysing natural language sentences. Even in theoretical work, analyses of sentences are not done for their own sake, but to elucidate some aspect of language in relation to other phenomena or concepts, such as meaning, entailment, or the intentions and beliefs of speakers. It is clear that the end products of analysis must relate directly to those other notions; otherwise further levels of analysis will be needed to approach the original goal.

Language analysis is used in the present work for a more modest and practical goal than the examples just mentioned, that of mechanically relating queries written in a limited subset of natural language to a database, a set of precise statements about some domain, stored in symbolic form. Here, perhaps even more than in theoretical work, the results of analysis are directly governed by the kinds of operations required by the initial goal.

To answer questions from a database, we need to argue from the statements contained in the database to reach a conclusion which fits the question. These arguments must be credible and reproducible,

and, in the present context, mechanisable. The reproducibility and credibility of a line of argument depend crucially on the kinds of rules used to move from one step of the argument to the next. These rules have to be mechanical, but also visible and simple enough to be examined and criticised. Logical systems satisfy these requirements, which in fact can be seen as their ultimate purpose.

Of course, arguments and dialogues between people make use of much larger, and mostly unexaminable, sets of rules than logical argument. Logical argument is but a stylisation of one aspect of such dialogues. However, I am not concerned with modelling people's performance, but with providing systems which answer questions in a precise, reproducible, predictable manner.

Ultimately, questions are evaluated against a database, and the answers should be the same as those obtained by exhaustive inspection of that database. That is, the relation between a question, the database and the answer should be the same as that between the question "How much is twenty five times thirteen?", the multiplication and addition tables, and the answer "Three hundred and twenty five". In this sense, what I am looking for is an arithmetic of states of affairs, and that is formal logic.

Of course, to use a logic system as "database arithmetic", the system must be simple enough to make the derivation of answers computable, at least in principle, for suitably restricted subject domains[1]. A system based on first order logic which satisfies this requirement is described in Section 2.3.

[1] In logical terms, for suitably restricted theories.

To clarify further the position of the present approach to natural language analysis with respect to the various potentially relevant theories of natural language analysis, I will now discuss the differences between it and two particularly well known and contrasting approaches: the theories, which I call **lexical**, of Schank, Wilks and others [Schank 75, Wilks 72], and the formal semantics, which I call **compositional**, of the Montague school [Montague 70]. The discussion will not be a comprehensive survey of these theories, as it will concentrate on those aspects which differ form the present approach.

1.1.2.1 Lexical Semantics

Lexical semantics develops what one may call the traditional view of semantics. Lexical semantics gives first importance to content words. By describing words as complex statements[2] built of "deeper" notions[3], the lexical semanticist hopes to elucidate how words fit together, for instance how certain arguments are possible for a verb and others are not. A useful representation of the semantics of a sentence for the lexical semanticist is in essence a paraphrase, where the content words of the original sentence have been rewritten into their representations and fitted together (matched) using the argument slots in those representations. Therefore, the representation of a word is a kind of "open" paraphrase. A lexical semantic interpretation program is said to "understand" its input

[2]**Conceptual graphs** [Schank 75], **formulae** [Wilks 72].

[3]**Primitives**, for instance "agent", "physical object", "cause".

when out of the input and of its stock of open paraphrases it can produce a paraphrase of the input, and answer questions about the various cases or roles in the paraphrase. For lack of uniform terminology, I give the name **lexical graph** to a paraphrase of some input in terms of primitives.

The matches between the words in a sentence might not be exact; for example the object of "John drank a glass of water" is not directly of the right kind because "to drink" requires a "liquid" object whereas "a glass of water" is "solid". But the match can still be done, if both "to drink" and "glass (of)" have suitable representations into primitives where the distinction between "container" and "contents" is explicit[4].

The crucial problem with lexical semantics, as far as the present work is concerned, is that it does not supply a precisely defined and general enough mechanism to derive answers to questions from a database of assertions. To answer a question is to try to find in the database a set of statements from which an instance of the question follows, for some precise notion of "follows from", that is, of meaning. The only such notion that seems to be available in lexical semantics is that of a statement X following from a statement Y if the lexical graph of X is some part of an instance of the lexical graph of Y. But this is too weak, because the number of statements in a lexical graph is finite, whereas from a statement such as

[4]See Wilks's notion of **preference** [Wilks 72].

No even number greater than 2 is prime.

it is possible to derive answers to an infinity of distinct questions.

The position of lexical semantics with respect to a notion of meaning is curiously parallel to that of generative grammar in its early stages. Both systems try to reduce equivalence of meaning to identity of structure, where structure is lexical graphs for lexical semantics, deep structure for generative grammar. But either the reduction is found to be too weak in some fundamental way, or extra interpretative machinery has to be brought in to cure the weakness [Jackendoff 72].

In practice, language processing systems based on the ideas of lexical semantics do include some more powerful "ad hoc" inference mechanisms. But because these mechanisms are not precisely described, they fail to elucidate the notion of valid answer which is required here [Newell 82].

However, at least some of the information present in the semantic representations of words in lexical semantics is essential in any practical system that translates sentences into a logical form. Even simple sentences are syntactically ambiguous, and we need then to decide what arguments can reasonably be attached to what words, and of the various readings for a word what are the reasonable ones. Lexical semanticists [Schank 73] use this point to deny the usefulness of an independent notion of syntax. I will return later to this criticism, and show that it is based on too simplistic a view of grammar.

The semantic representations of lexical semantics can be alternatively seen as encodings of logical statements relating the classes of objects that can fit argument places, and the process of matching as trying to find attachments that satisfy all those statements. In the present work, only an extremely limited version of the above deductive process is used. I will explain in Section 4.2.3 how nouns, their arguments and the arguments of verbs are associated to unary predicates (types) that represent the corresponding classes of objects. For practical reasons discussed there, this type system is not used through explicit deductions on the type predicates, but rather through operations on objects that denote types.

1.1.2.2 Compositional Semantics

Broadly, compositional semantics is an approach to the analysis of natural language in which the analysis of a phrase is at the deepest level its translation into formulae of an appropriate logical calculus by rules that describe how the translation of a phrase is built from formulae for its sub-phrases and the context of occurrence of the phrase. The translation of a phrase is its semantic interpretation [Montague 70, Montague 73, Thomason 76, Creswell 73, Shaumyan 77].

Compositional semanticists try to give logical descriptions of how a phrase or word modifies another. Here, a suitable notion of lexical category for a word, often close to the usual syntactic one, is much more important than the actual words involved, except for the closed category words, which play a fundamental role in "glueing" pieces of semantic interpretation together.

The present work, and much of the work on which it is based, has a connection to compositional semantics. The ideas of translation into a logical form, translation rules, the importance of lexical categories and closed category words are the same, and have their common ancestry in the earliest attempts of logicians at formalising the notions of inference and quantification in ordinary language.

However, the present work differs from compositional semantics in its purpose. It may be seen as applied naive compositional semantics. The overriding concern of the compositional semanticist is to find logical analyses of more and more subtle language phenomena, even if the resulting logical systems are not computationally practical[5]. In contrast, the present work is limited by the choice of a system of logic in which automatic deduction can be performed, and performed efficiently. This may well restrict the computational approach to using versions of the predicate calculus, with the consequent problem of having to express in a poorer system notions such as modality or tense that can be expressed with generality only in richer systems [Moore 81].

Also, compositional semantics is not in general concerned with whether translation rules are effectively computable. For example, Montague grammars [Montague 70] describe the alternative readings of a sentence, but also an infinity of unessential, trivially equivalent readings. In contrast, for the present work it is essential that translations can be not only computed but efficiently computed.

[5]For instance, consider the semantics of modal and intensional logics without any restriction on the allowed models [Hughes and Creswell 68, Montague 70].

1.2 Results

The results of the present work can be summarised as follows:

* a formalism for natural language grammar, extraposition grammars;
* a grammar that analyses a subset of English, but delays certain analysis decisions that require semantic information;
* rules to translate analysis trees into formulae of a system of logic for database queries, definite closed-world clauses;
* Chat-80, an efficient Prolog program that uses the above results to implement a prototype database query front-end that can be easily moved to different domains.

1.3 Relations to Other Work

As I have discussed above, there are certain similarities between the present work and work in formal semantics of natural languages. However, I draw most from the work of Colmerauer [Colmerauer 78] and Dahl [Dahl 77], and to a somewhat lesser extent from the LUNAR system [Woods et al. 72]. McCord's logic-based slot grammars are also related, but were developed independently of the ideas discussed here, and therefore their influence on this work is rather limited. However, because McCord's work has partly similar origins, and addresses some of the same problems, I will examine it in some detail. Of course, there is a variety of other connections, which I will note when the relevant questions are discussed.

1.3.1 Colmerauer and Dahl

The present work started as an attempt at clarifying, improving and extending the work of Colmerauer [Colmerauer 78, Colmerauer 79a] and Dahl [Dahl 77]. Colmerauer introduced three important ideas:

* a practical realisation of logic programming based on definite clauses: the programming language Prolog;
* a grammar formalism based on the Prolog subset of logic: metamorphosis grammars;
* a computable system of logic for the semantics of restricted natural language: three-branched quantifiers (3BQs).

Dahl's work makes essential use of all three of the above ideas, to implement a prototype natural language interface to a database.

I try to improve on Colmerauer's and Dahl's work in five main ways, by providing:

* a more concise and theoretically more sound grammar formalism;
* better handling of syntactic ambiguity;
* better rules for modifier attachment;
* more flexible treatment of determiner scope;
* a more adequate treatment of English semantics (joint work with Warren).

Detailed comparisons of the present work with Colmerauer and Dahl's in the areas of grammatical formalism, ambiguity and attachment, and determiner scope are given below in Sections 3.2, 4.2 and 4.3. I will discuss now only the differences in the treatment of natural language

semantics[6].

The semantic interpretation formalism I use, definite closed-world clauses (DCW clauses), differs from the three-branched quantifier (3BQ) logic proposed by Colmerauer and used by Dahl. A 3BQ formula is built from atomic formulae using the usual first-order connectives and three-branched quantifiers. A 3BQ has the form

```
  quant
  / | \
X  P  Q
```

where X is a variable, P and Q are formulae, and quant is any of several specialised quantifiers intended to capture the meaning of natural language determiners. Roughly, formula P will correspond to the translation of a noun phrase, and formula Q to the translation of a verb phrase. For example [Colmerauer 79a], the translation of the sentence "Haddock despises every man who does not sail" is

```
every(X,
      and(isman(X),
          not(sails(X))),
      despises(haddock,X))
```

The semantics of 3BQs, and that of the other connectives is very different from that of superficially similar first-order operators.

Firstly, a closed formula can have three distinct truth values, true, false and undefined. This is intended to capture noun phrase presuppositions, which are assumed to render meaningless rather than vacuously true sentences such as

[6]The differences between French (Colmerauer), Spanish (Dahl) and English (the present work) are trivial in the areas covered by the discussion.

Every city in Antarctica has a cinema.

(assuming of course that there are no cities in Antarctica).

Secondly, variables in 3BQ expressions are interpreted as ranging over sets of individuals rather than over individuals. The purpose of this interpretation, which I will criticise in detail in Section 4.3.2, is to give a uniform translation to both singular and plural noun phrases, and to move the burden of interpreting plurals in a formula from the quantifiers to the atomic predicates in the formula.

Finally, the meaning of 3BQ formulae is given in terms of interpretations (in the model-theoretic sense) ranging over finite domains, the various classes of individuals occurring in the statements in a database. In this way, the meaning of logical operators, and in particular that of negation, can be given easily in terms of the presence and absence of information in the interpretation. Of course, this makes it impossible to deal with infinite domains. Also, it makes the computation of the meaning of a formula very inefficient, because to determine the effect of a connective or a quantifier it is necessary to evaluate the sub-formulae in its scope over the entire domain to which they are applicable.

In contrast to 3BQs, DCW clauses are just definite clauses with the additional operators '\+' for non-provability and 'setof' for the set of objects that satisfy a condition. These two operators rely for their semantics on the closed world assumption of interpretations [Reiter 80, Clark 78], which takes as false anything

that is not explicitly stated, hence the name "definite closed-world clauses". Because they are based on definite clauses, DCW clauses have in principle no difficulty in dealing with infinite domains. The DCW proof procedure, based on the Prolog proof procedure [Roussel 75], does not need to iterate over entire domains. However, the efficiency and even the termination of the evaluation of a DCW clause require a careful choice of the order in which sub-formulae are evaluated. This has been achieved in a DCW clause evaluator due to Warren [Warren 81a], which constitutes the database side (not covered in this work) of the Chat-80 program.

Using DCW clauses instead of 3BQs prevents any treatment of the presuppositions in noun phrases, which is one of the main objectives of the 3BQ formalism. However, I take the view that it is more important to get the basic quantification mechanism right, than to try and deal with presupposition, which anyway poses problems of linguistics and logic that are not solved by three-branched quantifiers.

1.3.2 LUNAR

The system for natural language access to information on Lunar rock samples (informally called LUNAR) developed by W. Woods, B. Nash-Webber and R. Kaplan at BBN [Woods et al. 72], was the first example of a natural language interface to a database with any claims to practicality.

LUNAR still stands out among such systems for the conceptual economy of its design. Three important characteristics of LUNAR's design are:

* the use of a special purpose language for writing grammars, augmented transition networks, coupled with a non-deterministic parser;

* the use of a general subject-independent syntactic grammar;

* the translation of input into expressions of a uniform semantic interpretation formalism, akin to logic.

It is clearly difficult to summarise the linguistic coverage and performance of a natural language interface to a database, because of a lack of accepted measures. The comparisons with LUNAR in this work will therefore be mostly on questions of formal adequacy and conceptual economy of the formalisms used.

As regards efficiency of the overall system, I will just note that, for sentences of similar complexity, syntactic analysis in Chat-80 seems on average 20 times faster than in LUNAR, and semantic interpretation on average 30 times faster[7]. With respect to size, I have not been able to find a published value for the size of the LUNAR system, and I can only say that the text of the published fragments of LUNAR is roughly 4 times larger than that of the corresponding components of Chat-80. To put these figures in perspective I must point out that LUNAR is in several important respects more comprehensive than Chat-80. LUNAR's dictionary is much larger and also more complex. Its parser and semantic rules cope with certain limited forms of anaphora and ellipsis, which Chat-80 does

[7]Calculated over ten of the examples in Appendix G of the LUNAR final report [Woods et al. 72], and assuming that the DEC KL-10 in which Chat-80 has been timed is 6 times faster than the DEC KA-10 in which LUNAR was run, the ratio of the speeds of the same Prolog system on the 2 machines.

not, and the treatment of coordination in Chat-80 is limited to conjunctions of noun complements, whereas that in LUNAR copes with conjunction of other constituents, and across constituent boundaries as well. While we do not know for certain whether the wider coverage of LUNAR in these areas could account for the above differences in performance, it is difficult to accept that the relatively small proportions of LUNAR's parser and semantic rules responsible for the wider coverage justify a factor of 20 or 30 in speed. Rather, some of this extra cost in LUNAR must come from the general disadvantages of LUNAR's machinery for structure building and pattern matching, compared with definite clause programs, which I have argued in joint work elsewhere [Warren et al. 77, Pereira and Warren 80]. A further cause of LUNAR's relative inefficiency could well be the complex mechanisms, such as that for "selective modifier placement", required to allow the parser to build complete analyses in the absence of essential semantic information.

I will now discuss the relation of the present work to LUNAR, in the three aspects mentioned earlier in this section, of syntactic formalism, grammar organisation, and semantic interpretation.

The syntactic component of LUNAR is an augmented transition network (ATN) [Woods 70]. ATNs are non-deterministic pushdown automata with registers, whose state transitions can be supplemented by arbitrarily complex actions involving pieces of code in some external programming language, in LUNAR's case Lisp. Whereas an ATN without any actions is clearly just another notation for a context-free grammar whose rules may have alternation and indefinite repetition (Kleene's '*')

in their right hand sides, ATNs used in practice, such as that in LUNAR, rely to such an extent on actions dependent on the actual parsing strategy, that they cannot be possibly seen as grammars but only as parsing programs written in a special-purpose language.

In contrast, Chat-80 uses an extraposition grammar, which is genuinely a grammar, in the sense that the relation between the grammar and the language it accepts can be characterised without reference to a parsing mechanism. An extraposition grammar is just a convenient notation for a set of definite clauses defining a fragment of natural language.

In an article with Warren [Pereira and Warren 80], I compared definite clause grammars (DCGs), a specialisation of Colmerauer's metamorphosis grammars, with ATNs. As the same argument carries over to XGs, it would be useless to recast it here, and I will just quote a summary of the most important conclusions:

> "Considered as practical tools for implementing language analysers, DCGs are in a real sense more powerful than ATNs, since, in a DCG, the structure returned from the analysis of a phrase may depend on items which have not yet been encountered in the course of parsing a sentence. ... Also on the practical side, the greater clarity and modularity of DCGs is a vital aid in the actual development of systems of the size and complexity necessary for real natural language analysis. Because the DCG consists of small independent rules with a declarative reading, it is much easier to extend the system with new linguistic constructions, or to modify the kind of structures which are built. ... Finally, on the philosophical side, DCGs are significant because they potentially provide a common formalism for theoretical work and for writing efficient natural language systems."

With respect to modularity and clarity, I will note the earlier comparison between the textual sizes of LUNAR and Chat-80. I argued in Section 1.1.1 the adequacy for theoretical work of grammar

formalisms based on definite clauses, and in Chapter 3 I will show how XGs in particular can describe economically important phenomena of interest to linguistics. The usefulness of the formalism in practical language analysis tasks is shown by the efficiency and economy of Chat-80.

As the scale of the present work is smaller than that of LUNAR, the syntactic XG in Chat-80 has a narrower coverage than the LUNAR ATN. However, I suggest that, if one takes the corresponding subset of the LUNAR ATN, the use of XGs makes the Chat-80 grammar much more clear and concise. Also, the attachment mechanism in the Chat-80 grammar does away with the "ad-hoc" techniques of "selective modifier placement" that are needed in LUNAR to control the placement of modifiers in analysis trees.

I try in the present work to bring the actual semantic interpretation process closer to the theory. The theory of semantic interpretation in LUNAR [Woods 77a], which I follow to a considerable extent, was implemented by a large body of rules in an "ad-hoc" tree-manipulation language. By using definite clauses, both for the grammar and for the semantic interpretation rules, I avoid most questions of process control, concentrating on the actual formulation of the rules. From this point of view, the present work may be seen in part as a "rational reconstruction" of the LUNAR semantic interpretation rules. By moving modifier attachment out of the syntax, and by dividing semantic interpretation into inserting

arguments and modifiers on the one hand and scoping on the other[8], through the notion of a predicate-quantifier tree, I bring out those different issues which are entangled in the single system of semantic rules in LUNAR.

Although the target semantic interpretation formalism of LUNAR is similar to a logical language, it is not a full-fledged deductive tool, but only a data-retrieval language, where each "connective" is interpreted as a procedure operating on explicit data tables. For example, LUNAR's translation of the question "Do any breccias contain aluminum?" is the expression

```
(TEST
   (FOR SOME X2 / (SEQ TYPECS) : T;
      (CONTAIN
         X2
         (NPR* X3 / 'AL203))
         'NIL ) ) )
```

which can be paraphrased as "test if for some thing X2 from the class TYPECS (the class of breccias in the database), X2 contains a constant object X3 with name 'AL203' (the name of the element aluminum in the database)". The operator FOR is actually a Lisp procedure that scans the database for a number (SOME meaning at least one in the present example) of objects from a class satisfying a predicate, and returns a suitable truth value. This mechanism is very similar to that used by Dahl to interpret 3BQ queries, and as such suffers from the same problems.

[8] A similar distinction is used in the PSI-KL-ONE system [Sidner et al. 81].

LUNAR, lacking deductive mechanisms, can evaluate the "quantifiers" in the translation of a sentence only extensionally, against actual database entries. In contrast, DCW clauses are a deductive formalism. Because of this, a number of problems that arise in LUNAR in mapping words and constructions into operations on the data tables, are entirely avoided in Chat-80: every content word can be associated to a number of possible interpretations, each of them an atomic relation. The relationship between these atomic relations (predicates) and the actual data base relations is defined as appropriate by an arbitrarily complex DCW clause program. Furthermore, the fact that the DCW clause formalism can be understood independently of execution makes it possible to deal with the problems of query evaluation independently of the problems of semantic interpretation: the result of semantic interpretation doesn't need to be directly executable by the Prolog proof procedure, and in general it is not.

LUNAR is capable of dealing with a number of anaphoric and elliptic constructions. I have made no attempt to treat those phenomena. Although it might be argued that anaphora and ellipsis, because of the syntactic, semantic and pragmatic questions that they raise, and because of their importance for natural language interaction with computers, should be given priority, I think that only a clear, principled account of simpler natural language constructions can provide the groundwork on which to build practical theories of anaphora and ellipsis [Nash-Webber and Reiter 77]. In particular, any satisfactory treatment of anaphora must rely on an adequate formal semantics for plural noun phrases.

1.3.3 Slot Grammars

McCord's approach [McCord 80a] is to generalise the traditional notions of determiner and modifier to represent both syntactic and semantic relationships in a sentence. In fact, the analysis of a sentence is not a phrase tree in the conventional sense, but a tree where each node (or syn) describes the the way in which it modifies its parent node. Each node has a determiner (which for noun phrase nodes corresponds to the lexical determiner), a head word, grammatical features and a list of modifiers. For example, the syntactic structure for "Each man saw John" is:

```
[s,dcl] main see(X,Y,C)
   [np,sg] all:X man(X)
   [advc] conjunct past(C)
   [np,sg] def(sg):Y Y=john
```

where each line gives the features, determiner and head of a syn, and the modifiers of a syn, if any, are listed in a column below and to the right of the syn. In the above analysis, we have a top syn with determiner 'main' indicating the semantic role of a main verb, and a head formed by the predication "see(X,Y,C)" that translates the main verb. The first modifier of this syn is that of the subject, with determiner "all:X" indicating an universal quantification with variable X over the parent syn, and with head "man(X)" giving the predication for the noun of the subject. The second modifier, corresponding to the tense of the main verb, has determiner 'conjunct', specifying that the meaning of this syn is to be conjoined with that of its parent, and head "past(C)" which constrains the "context" argument C of the main verb. The final modifier, corresponding to the direct object of the main verb, has

determiner "def(sg):Y", specifying a definite singular "quantification" with bound variable Y over the parent **syn**, and has the equality "Y=john" as head, forcing the range of the quantification variable Y to the single value 'john'.

To derive the semantic interpretation of a sentence (a logical formula), a tree rewrite process applies each modifier of a **syn**, according to its determiner, to the parent **syn**. Taking the last example, the result of applying rewriting could be[9]:

```
dcl(
  all(X,
    man(X) =>
    exists(Y,
      Y=john &
      see(X,Y,C) &
      past(C) ) ) )
```

We could read this formula as "for all X, if X is a man there is a Y named 'john' and a C such that X saw Y at C, and C is a past event". Notice how the determiners of the noun phrases turn into logical quantifiers, and the 'conjunct' determiner into a conjunction.

The determiners used by McCord are all referencially transparent: the application of a determiner to its arguments is independent of the internal structure of the arguments. This is similar to the semantic composition rules in Montague grammar.

The translation of a sentence into a logical formula is divided into three operations: analysing the sentence into a **syn** tree;

[9]I use here a notation slightly different from McCord's, for ease of explanation.

reshaping the syn tree to take into account the relative scopes of its nodes, mostly derived from the (generalised) determiners; and applying the determiners to get the final formula.

The determination of what phrases modify, or are arguments of, what phrases is done in the parser: head word entries in the dictionary describe what slots (argument places) each translation of the word has.

McCord's theory of determiners of modifiers leads to a more concise description of grammatical relationships and their semantics than approaches based on traditional syntax trees and syntactic categories. Furthermore, adverbials can be incorporated cleanly in the theory [McCord 81]. On the other hand, the exclusive use of transparent operators for determiners causes difficulties in situations were the determiner or head word seem to have an opaque role, such as sentences with main verb "to have". I discuss this question further in Section 4.2.

1.4 Summary

The approach to natural language analysis presented in this work has three main components: a grammar formalism based on definite clause logic, extraposition grammars; a database-oriented semantic representation formalism, definite closed-world clauses, also based on definite clause logic; and a set of informal rules for translating the analysis trees of sentences into their semantic representations. The definition of definite closed-world clauses is a combination of the requirements of semantic representation in a database context,

and of those of practical query processing in a logic database as developed by Warren. The language analysis side and the query processing side have been combined into an efficient natural language database query program, Chat-80.

The use of definite clause logic in language analysis originated with Colmerauer's work, on which I build. The overall organisation of the analysis process is however closer to that of the LUNAR system than to that used by Colmerauer and his co-workers.

Extraposition grammars cure some of the problems earlier logic-based grammar formalisms had when describing left extraposition phenomena. Extraposition grammars also have general theoretical and practical advantages over the ATN formalism used in LUNAR. The greater efficiency of Chat-80 should in part be ascribed to those advantages. Another factor of Chat-80's efficiency is the technique used in the grammar to delay post-modifier attachments and thus reduce the number of parsing alternatives. In contrast to other similar techniques, mine does not depend on the particular parsing algorithm being used.

The semantic interpretation rules are in many ways similar to those of LUNAR, but are made clearer by being expressed in definite clauses. Some of the semantic interpretation concepts are also related to the ones developed by McCord. The main difference is that McCord's rules are more explicit and general because they are expressed in terms of general concepts of modifier and determiner in syntax analysis, instead of the "ad-hoc" parse trees used in Chat-80. On the other hand, McCord's system is compositional, whereas some of

the constructions tackled by Chat-80 seem to require non-compositional rules. This is also one of the main differences between the present work and the compositional semantics of Montague and others, the other being my use of a semantic interpretation formalism that can be used directly for computing answers from a database.

Chapter 2

Background

2.1 Grammars and Logic

2.1.1 Definite Clause Logic

This section summarises the subset of predicate calculus, **definite** clauses, which is used in the whole of the present work. This is the system of logic underlying the programming language Prolog [Roussel 75, Warren et al. 77] and the various grammar formalisms discussed in the present work.

A definite clause has either the form

$P :- Q_1, \ldots, Q_n.$

to be read as " P is true if Q_1 and ... and Q_n are true", or the form

P.

to be read as "P is true". P is the head of the clause; $Q_1, \ldots, Q_n$ are goals, forming the body of the clause. The head and the goals represent instances of predicates, by giving a **predicate symbol**, and possibly some arguments (in parentheses, separated by commas):

father(X,Y) false number(0)

A predicate instance represents a relation between its arguments; e.g. "father(X,Y)" denotes the relation 'father' between X and Y.

Arguments are terms, standing for partially specified objects. Terms may be

* variables, denoting unspecified objects (variable names are capitalised):

    X Case Agreement

* atomic symbols, denoting specific objects:

    plural [] 3

* compound terms, denoting complex objects:

    s(NP,VP) succ(succ(0))

A compound term has a functor and some arguments, which are terms. Compound terms are best seen as trees:

A particular type of term, the list, has a simplified notation. The binary functor '.' makes up non-empty lists, and the atom '[]' denotes the empty list. In the special list notation,

    [a,b] [X|Y]

represent respectively the terms

    .(a,.(b,[]) .(X,Y)

Putting these concepts together, the clause

    grandfather(X,Z) :- father(X,Y), parent(Y,Z).

may be read as "X is grandfather of Z if X is father of Y and Y is a parent of Z"; the clause

```
father(john,mary).
```

may be read as "John is father of Mary" (note the use of lower case for the constants in the clause).

A set of definite clauses forms a **program**. A program defines the relations denoted by the predicates appearing in the heads of clauses. When using a definite clause interpreter, such as Prolog [Roussel 75], a goal statement

```
?- P.
```

specifies that the relation instances that match P are required.

2.1.2 Definite Clause Grammars

I will now describe how definite clauses of a certain form can be seen as grammar rules, leading to the notion of grammar formalism based on definite clauses.

Any context-free rule, such as

```
sentence --> noun_phrase, verb_phrase.
```

(I use ',' for concatenation, and '.' to terminate a rule) may be translated into a definite clause

```
sentence(S0,S) :- noun_phrase(S0,S1), verb_phrase(S1,S).
```

which says: "there is a sentence between points S0 and S in a string if there is a noun phrase between points S0 and S1, and a verb_phrase between points S1 and S". A context-free rule like

```
determiner --> [the].
```

(where the square brackets mark a terminal) can be translated into

```
determiner(S0,S) :- connects(S0,the,S).
```

which may be read as "there is a determiner between points S0 and S in a string if S0 is joined to S by the word 'the'". The predicate 'connects' is used to relate terms denoting points in a string to the words which join those points. Depending on the application, different definitions of 'connects' might be used. In particular, if a point in a string is represented by the list of words after that point, 'connects' has the very simple definition

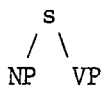
```
connects([Word|S],Word,S).
```

which may be read as "a string point represented by a list of words with first element Word and remainder S is connected by the word Word to the string point represented by list S."

DCGs are the natural extension of context-free grammars obtained through the translation into definite clauses outlined above. A DCG non-terminal may have arguments, of the same form as those of a predicate, and a terminal may be any term. For instance, the rule

```
sentence(s(NP,VP)) --> noun_phrase(NP,N), verb_phrase(VP,N).
```

states: "a sentence with structure



is made of a noun phrase with structure NP and number feature N, followed by a verb phrase with structure VP agreeing with the number N". A DCG rule is just "syntactic sugar" for a definite clause. The clause for the example above is

```
sentence(s(NP,VP),S0,S) :-
   noun_phrase(NP,N,S0,S1),
   verb_phrase(VP,N,S1,S).
```

In general, a DCG non-terminal with n arguments is translated into a predicate of n+2 arguments, the last two of which are the string points, as in the translation of context-free rules into definite clauses.

It is also possible to include in DCG rules tests defined by definite clauses. For example, in the rule

```
s --> np(N), vp(V), {agree(N,V)}.
```

the term in curly brackets is a normal definite clause goal. The rule translates into the definite clause

```
s(S0,S) :- np(N,S0,S1), vp(V,S1,S), agree(N,V).
```

The main idea of DCGs is then that grammar symbols can be general logic terms rather than just atomic symbols. This makes DCGs a general-purpose grammar formalism, capable of describing any type-0 language. As mentioned before, the first grammar formalism with logic terms as grammar symbols was Colmerauer's metamorphosis grammars [Colmerauer 78]. The difference between DCGs and MGs is that the left-hand side of a DCG rule is always a single non-terminal symbol, whereas the left-hand side of an MG rule can be a sequence of non-terminals subject to certain restrictions. Thus, a DCG can be seen as a context-free grammar with logic terms for grammar symbols, whereas a metamorphosis grammar can be seen as somewhat restricted type-0 grammar with logic terms for grammar symbols. We should note, however, the very simple translation of DCGs into definite clauses

presented above doesn't carry over directly to MGs in their full generality.

2.2 Logic Programming

By choosing a suitable inference mechanism to prove goals from definite clause programs, a definite clause program can be seen as a program in the more usual sense. The Prolog language [Roussel 75] is no more than definite clause logic, together with an inference mechanism and some extra-logical facilities to control inference and interface with the environment. The Prolog inference mechanism proves a goal by finding clauses whose head unifies with the goal, and in turn proving the goals in the body of those clauses (top-down execution). It uses backtracking to cycle through alternative clauses whose head unifies with a goal. This inference mechanism can be implemented with particular efficiency, although, in common with the related simple top-down backtrack parsing algorithms, it will not terminate where a more sophisticated mechanism would because it does not check for repeated sub-goals.

The program that results from this work, Chat-80, is written in Prolog, either directly, or indirectly as an extraposition grammar that is then translated into definite clauses. Chat-80 has been run on the DEC-10 Prolog system [Pereira et al. 78], and transcripts with timings are given in Appendix F.

The only extra-logical facility of Prolog that occurs in the components of Chat-80 listed in Appendix C is the **cut** operator '!'. As I noted above, Prolog attempts to prove a goal using a top-down

backtrack procedure. The cut operator is used to control this procedure by removing some of the pending backtracking alternatives created when a proof is being built. When using a clause c to try to prove a goal g, the Prolog proof procedure takes the goals in the body of c in a left-to-right order and tries to prove them. When looking for ways of proving a goal, Prolog tries in turn each of the clauses whose head matches the goal, in the order they appear in the program text. When using a clause to prove goal g, Prolog saves as a pending alternative a record of the clauses that remain to be tried for g. Now, given clauses

```
p :- q, ..., !, ...
p :- ...
        ...
```

if the search for a proof reaches the cut '!', Prolog will discard all the pending alternative ways of trying to prove the already proved goals q, ... before the cut, and also the pending alternative corresponding to the remaining clauses for p. This means that if the proof procedure reaches a dead-end and has to backtrack, those alternative ways of trying to prove the goals q, ... before the cut and of trying to prove p itself will not be explored. Thus, a goal might be a consequence of a program, but Prolog may fail to find it due to a cut. Therefore, the cut operator compromises the completeness of the basic Prolog proof procedure for reductions in the saving of alternatives needed to prove a goal.

In the programs of Appendix C, the cut operator is only used to avoid having to list a large, but finite, number of alternative clauses for a predicate. In each case, the alternatives ruled out by the cut would lead to dead-ends elsewhere in the program.

2.3 A Computational Logic for Natural Language Questions

I discuss now a system of logic, definite closed world (DCW) clauses, that can be used to represent the meaning of natural language questions in a limited database context. The system is based on definite clause logic, with two extensions designed to overcome the lack of negation, full quantification and sets in definite clauses. Because the system is "almost" definite clause logic, it can be used to answer queries efficiently.

To distinguish DCW clauses, which may not be directly executable by Prolog, from definite clauses, a DCW clause, and indeed any other first order-like formula which is not to be seen as a definite clause, will be written

$$H <= G_1 \ \& \ G_2 \ \& \ \ldots$$

instead of

$$H :- G_1, G_2, \ldots$$

The impossibility of using definite clause logic alone to represent the meaning of sentences is easily shown if we try to translate any of the following questions as a conjunction of goals:

Is there any ocean which borders all European countries? (2.1)
Which oceans border at least three European countries? (2.2)

Intuitively, the translation of (2.1) should be:

```
answer(yes) <=
   exists(O, ocean(O) &
      all(C,country(C) & european(C) =>
         borders(O,C) ) )
```

A reasonable translation of (2.2) would be:

```
answer(O) <= ocean(O) &
   card { C : country(C) & european(C) & borders(O,C)} > 2
```

where 'card S' denotes the cardinality of set S.

The general form of the formulae which can be mapped into DCW clauses is

```
literal <= condition                                    (2.3)
```

where condition may contain any of the quantifiers and connectives described below, and all free variables in literal or condition are implicitly universally quantified at the outermost level of the DCW clause, as is usual in clausal logic. Not all such formulae, however, can be mapped meaningfully into DCW clauses. We need certain assumptions, to map the classical quantifiers, negations, and set expressions contained in condition into DCW clauses.

The main assumption required is the closed world assumption [Reiter 80, McDermott and Doyle 80, Bowen and Kowalski 81]. When we assume a closed world, we allow ourselves to infer "not P" from "P is not provable". I leave to the next section the detailed discussion of how such an "inference" rule can be integrated properly with definite clause logic and the Prolog proof procedure.

We should note here, however, that the Prolog proof procedure can be used to establish the non-provability of a conjunction of goals only if all the goals are ground. In particular, if the formula condition of (2.3) above contains a subformula "not P", the formula must obey the producer restriction: any variable free in P must occur in goals, its producers, which are outside P but inside the scope of

any other negation that has P in its scope[10]. That is, any variable X free in a negation should occur in a configuration of the kind

... (... producer(... X ...) ... not (... X ...) ...) ...

Then if all the predicates in the formula satisfy a further assumption, the groundness assumption to be described below, the proof procedure will be able to use the producers of variables in a negated formula to instantiate the variables to ground terms. The groundness assumption, which is acceptable in a database context, states that any provable consequence that consists of a single atomic formula will be ground. In fact, this assumption is also required for dealing with sets, and is precondition of the query planning algorithm used in Chat-80 [Warren 81a, Warren and Pereira 81].

The operators allowed in the right-hand side of a DCW clause are the following:

p & q	Conjunction: it holds if p and q hold;
exists(x,p)	Existential quantifier: it holds if there is an instance y of x such that the goal obtained by substituting y for all occurrences of x in p holds;
\+ p	Non-provability: it holds if p is not provable;
setof(x,p,s)	Set : it holds if s is the (finite) non-empty set of instances of x such that the corresponding instances of p are provable (This is well defined because the groundness assumption and the producer restriction force the instances of x to be ground).

Two further operators, 'all' and 'numberof', can be defined by the following identities:

[10] I am assuming here that all universal quantifications contained in condition have been transformed into explicit existential quantifications and negations.

```
all(x, p => q) <=> \+exists(x, p & \+ q)

numberof(x, p, n) <=>
        exists(s, setof(x, p, s) & card(s, n))
```

where 'card' is a predicate that counts the number of elements in a finite set representation.

The operators '\+' and 'setof' are different from the usual first-order connectives and quantifiers in that their definition is given in terms of provability and not in terms of truth. This has the consequence that a goal may hold in a program but no longer hold if further clauses are added to the program (non-monotonicity [McDermott and Doyle 80]), in contrast with what happens in first-order logic. This raises the question of whether the definitions of '\+' and 'setof' are independent of the actual proof rules used, and therefore logically well defined. I will show this to be the case in the next section.

The role of existential quantifiers in the right-hand side of DCW clauses requires some explanation. If 'exists' occurs outside the scope of any non-monotonic operator, it can be eliminated by using the obvious first-order equivalences

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

that hold with X renamed to avoid clashes with variables in p. If 'exists' occurs within the scope of a 'setof' or of a '\+', the subformula

```
exists(X,p)
```

can be replaced by

p(Y1,...,Yn)

where Y1,...,Yn are the variables free in p and the predicate 'p' is defined by

p(Y1,...,Yn) <= p.

This is a valid transformation for DCW clauses because the meanings of '\+' and 'setof' are given in terms of provability, and the only way of proving the 'exists' goal is to prove an instance of its scope p.

The use of non-provability is a common feature in Prolog programs and question-answering systems in particular [Colmerauer 78, Dahl 77, Pique 81]. The set operator, in the form presented here, is due to Warren [Warren 81b, Warren and Pereira 81].

I can now give a first translation of some English determiners into DCW formulae. The translations are taken from my work with Warren, and are used in Chat-80. Each determiner is translated into a quantification, which introduces some logic variable (X, N, etc.), and which links two predications involving that variable, called the range and scope of the variable, here indicated by R and S.

a, some, the[singular]	exists(X,R & S)
no	\+exists(X,R & S)
every, all	\+exists(X,R & \+S)
the[plural]	exists(S,setof(X,R,S) & 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)

In general, a variable corresponds to an explicit or implicit noun phrase, the range of the variable is the translation of the words making the noun phrase, and the scope of the variable is the translation of the rest of the sentence where the noun phrase occurs.

The following examples show the intended use of the above translations[11]. For simplicity, all but two of the examples are given as translations of declarative sentences. These translations correspond to the right-hand side of the DCW clauses that would translate the corresponding yes-no question. To help relate the examples to the translations, the range (in the technical sense given above) of the quantified variable in each example is underlined.

Some birds migrate.
exists(X,bird(X) & migrates(X)).

The population of Britain exceeds 50 million.
exists(X,population(britain,X) & X > 50000000).

There are no rivers in Antarctica.
\+exists(X,river(X) & in(X,antarctica)).

Man inhabits every continent.
\+exists(X,continent(X) & \+inhabits(man,X)).

Jupiter is the largest of the planets.
exists(S,setof(X,planet(X),S) & largest(S,jupiter)).

The Rhine flows through three countries.
numberof(X,country(X) & flows_through(rhine,X),3).

Which birds migrate?
answer(X) <= bird(X) & migrates(X).

How many countries export oil?
answer(N) <= numberof(X,country(X) & exports(X,oil),N).

[11]For the purposes of this example, I take the simplistic view that mass nouns such as "oil" can be translated as logical constants.

The determiners "a", "some", "every", "all" and "no" are given the usual "naive" interpretation as first-order quantifiers, except that negation is replaced by non-provability. In Chapter 4 I will discuss in more detail these interpretations, and in particular those given here to the determiner "the" and to questions.

2.3.1 The Semantics of DCW Clauses

The difficulties in the formal definition of the semantics of DCW clauses arise from non-monotonic goals, which do not belong to classical first-order logic. Both '\+' and 'setof' are defined in terms of a notion of provability for definite clauses. A suitable notion of provability is that provided by special linear definite resolution (SLD-resolution), a family of proof procedures of which the Prolog proof procedure is an instance. SLD-resolution is complete; that is there is an SLD-resolution proof for any true goal[12] [van Emden and Kowalski 76, Apt and van Emden 80].

SLD-resolution may be seen as taking a definite clause program and a goal and producing a possibly non-terminating enumeration of true instances of the goal. If the procedure terminates, the resulting instances (or solutions) have as instances all provable consequences of the program that are instances of the goal. That is, a solution is either a ground term or a term with variables such that all its instances are instances of the goal. For example, a SLD proof procedure applied to the goal "p(X,Y)" and the program

[12]Note however that proof procedures of the SLD-resolution family, for example the Prolog proof procedure, may fail to find a proof, and loop instead.

```
p(a,b).
p(X,Y) :- q(X,Y).

q(Z,Z).
```

would return the two solutions (modulo renaming of variables):

```
p(a,b)
p(Z,Z)
```

The instances of the goal implied by the program are "p(a,b)" and any literal of the form "p(t,t)" for some term t. A literal of any other form is not an instance of the goal provable from the program.

Clark [Clark 78] proves the following results:

(1) If an SLD proof procedure terminates without finding a proof for a given ground goal (a goal without variables) from a program P, the negation of the goal is a consequence of a particular first-order formula called the iff-version of P. The iff-version of a definite clause program is a first-order formula that basically corresponds to making all the implications in the program into equivalences, and adding axioms stating that all syntactically distinct objects of the Herbrand universe of the program are different.

(2) If an SLD proof procedure terminates after enumerating all the alternative proofs of a goal

    p(X)

    with a single variable X and the corresponding instantiations of X are

    $x_1, \ldots, x_n$

    then

    $$all(X, p(X) <=> X=x_1 \text{ v } \ldots \text{ v } X=x_n) \qquad (2.4)$$

    is a consequence of the iff-version of the program[13].

---

[13]This is follows from Theorem 3 in Clark's "Negation as Failure" [Clark 78].

These results support the following kind of justification for non-monotonic goals: there is a mapping from non-monotonic goals to first-order formulae such that if the non-monotonic goal holds in a program, the corresponding first-order formula (the meaning of the non-monotonic goal) is a consequence of the iff-version of the program. We have here a kind of soundness result for DCW clauses. We should note however that the converse result, that the completeness of SLD-resolution carries over to DCW clauses, only holds under certain complex assumptions, which I will not discuss here [Siegel and Bossu 81].

The justification for the operator '\+' follows directly from (1) above. I will now derive a similar justification for 'setof'.

Note first that any goal

```
setof(x,p,s)                                    (2.5)
```

has precisely the same solutions as

```
setof(X,q(X,Y),s)                               (2.6)
```

where, without loss of generality, I assume that there is a single variable Y occurring in p and not in x, X is a new variable, and 'q' is defined by

```
q(x,Y        .) <= p.
```

Furthermore, (2.6) has by definition the same solutions as the conjunction

```
setof(<X,Z>,q(X,Z),S) & project(S,Y,s)
```

where 'project' is defined by

```
project(P,Y,S) <=
   skim(P,Y,S0) &
   merge(S0,S).

skim([],Y,[]).
skim([<X,Y>|P],Y,[X|S]) <=
   skim(P,Y,S).
skim([<Y0,X>|P],Y,S) <=
   Y0 ≠ Y &                                          (2.7)
   skim(P,Y,S).
```

A goal "project(P,Y,S)" is true if P is a list of pairs, Y is the second coordinate of some element of P, and S is a list without repetitions of the first coordinates of all the elements of P whose second coordinate is Y. The first clause of the definition takes a list of pairs P, produces the list S0 of first coordinates of elements of P that share a specific second coordinate Y ("skim(P,Y,S0)"), and removes duplications from S0 to produce S ("merge(S0,S)"). The first clause for 'skim' returns an empty list of first coordinates for the empty list of pairs. The second clause for 'skim' places the first coordinate of a pair in the result list given that its second coordinate coincides with the shared value Y. The third clause skips a pair from the set of pairs, because its second coordinate is different from the shared value Y. The predicate 'merge', not defined here, just removes repetitions from a list. Goal (2.7) above, and a similar one that would be needed in the definition of 'merge', cause no problems in the present context, because of the implicit axioms that guarantee that distinct elements of the Herbrand universe are not equal.

Now, applying again the rewrite of (2.5) into (2.6), we need to look only at 'setof' goals of the form

```
setof(X,r(X),s)                                      (2.8)
```

To finish these preliminaries to the justification of 'setof', I need to define a membership predicate for finite sets represented as lists

```
member(X,[X|L]).
member(X,[Y|L]) <= member(X,L).
```

From result (2) above it follows that if $x_1, \ldots, x_n$ are ground terms, the following is a consequence of the iff-version of 'member':

$$\text{all}(X,\ \text{member}(X,[x_1,\ldots,x_n]) \iff X=x_1 \vee \ldots \vee X=x_n) \qquad (2.9)$$

It is now clear from the operational definition of 'setof' (2.4) and (2.9), that if (2.8) holds, the following is a logical consequence of the iff-version of the program and the 'member' definition:

```
all(X, r(X) <=> member(X,s) )
```

Given that the operational definition of 'setof' requires the sets returned to be non-empty, and their representations sorted in some canonical order, the desired meaning of a goal "setof(x,p,s)" will then be

```
all(x, p <=> member(x,s)) &
non_empty(s) &
sorted(s)
```

with suitable definitions of 'non_empty' and 'sorted'.

The preceding argument serves to show that the meaning of a 'setof' goal can be characterised in logical terms, independent of the actual proof procedure used to prove the goal, thus justifying the use of the term "logic" as applied to the language of DCW clauses used here to express the meaning of natural language questions.

2.4 Summary

In this chapter I surveyed the techniques that underpin my use of logic for the analysis and semantic interpretation of natural language.

I described the definite clause subset of first-order logic. This subset has unique properties that make it possible to use automatic deduction as an efficient computational mechanism. Definite clause logic also suggests a grammar formalism, definite clause grammars, a natural extension of context-free grammars in which phrase structure rules can be combined with complex tests and structure-building operations.

Definite clause logic is clearly inadequate to express the meaning of natural language sentences, because of its lack of negation and general quantification. However, by extending definite clauses with the non-monotonic operators '\+' (non-provability) and 'setof' (set of provable instances), it is possible to express the meaning of natural language questions about a world satisfying certain assumptions required to give this extended logical system, definite closed-world clauses, a well-defined semantics. Because they are defined in terms of definite clause provability, the non-monotonic operators inherit the computational advantages of definite clauses, and definite closed-world clause proofs are very directly interpretable as the computation of answers to queries to a database.

Chapter 3

Extraposition Grammars

This chapter is an expanded version of my articles "Extraposition Grammars" and "Ambiguity in Logic Grammars" [Pereira 81a, Pereira 81b].

The following conventions are used in this chapter: in the main text, but not in the actual rules, the names of non-terminals, predicates and other constants are underlined, e.g. np; meta-variables which range over grammar symbols are in boldface, e.g. **nt**; and in the figures, only terminal symbols are underlined.

3.1 Left Extraposition

Roughly speaking, left extraposition occurs in a natural language sentence when a subconstituent of some constituent is missing, and some other constituent, to the left of the incomplete one, represents the missing constituent in some way. It is useful to think that an empty constituent, the trace, occupies the "hole" left by the missing constituent, and that the constituent to the left which represents the missing part is a marker, indicating that a constituent to its right contains a trace [Chomsky 75]. One can then say that the constituent in whose place the trace stands has been extraposed to the left, and, in its new position, is represented by the marker.

For instance, relative clauses are formed by a marker, which in the simpler cases is just a relative pronoun, followed by a sentence where some noun phrase has been substituted by a trace. This is represented in the following annotated surface structure:

The man that$_i$[$_s$John met t_i] is a grammarian.

In this example, t stands for the trace; "that" is the surface form of the marker, and the connection between the the two is indicated by the common index i.

The concept of left extraposition plays an essential role, directly or indirectly, in many formal descriptions of relative and interrogative clauses. Related to this concept, there are several "global constraints", the "island constraints", that have been introduced to restrict the situations in which left extraposition can be applied. For instance, the Ross complex-NP constraint [Ross 74], implies that any relative pronoun occurring outside a given noun phrase cannot be bound to a trace occurring inside a relative clause which is a subconstituent of the noun phrase. That is, configurations of the form

$$X_1 \ldots [_{np} \ldots [_{rel}\ X_2\ [_s \ldots t_2 \ldots t_1 \ldots]] \ldots]$$

are not possible.

Note that I use the concept of left extraposition here in a loose sense, without relating it to transformations as in transformational grammar. In XGs, and also in other formalisms for describing languages (for instance the context-free rule schemas of Gazdar [Gazdar 79b]), the notion of transformation is not used, but a

conceptual operation of some kind is required to relate a relative pronoun to a "hole" in the structural representation of the constituent following the pronoun.

3.2 Limitations of Other Formalisms

To describe a fragment of language where left extraposition occurs, one might start with a context-free grammar which gives a rough approximation of the fragment. The grammar may then be refined by adding arguments to non-terminals, to carry extraposed constituents across phrases. This method is analogous to the introduction of "derived" rules by Gazdar [Gazdar 79b]. Take for example the CFG

```
sentence --> noun_phrase, verb_phrase.

noun_phrase --> proper_noun.
noun_phrase --> determiner, noun, relative.
noun_phrase --> determiner, noun, prep_phrase.
noun_phrase --> trace.                                    (1)

trace --> [].

verb_phrase --> verb, noun_phrase.
verb_phrase --> verb.

relative --> [].
relative --> rel_pronoun, sentence.

prep_phrase --> preposition, noun_phrase.
```

In this grammar it is possible to use rule (1) to expand a noun_phrase into a trace, even outside a relative clause. To prevent this, I will add arguments to all non-terminals from which a noun phrase might be extraposed. The modified grammar, now a DCG, has the following rules:

```
full_sentence --> sentence(nil).

sentence(Hole0) -->
      noun_phrase(Hole0,Hole1), verb_phrase(Hole1).
```

```
noun_phrase(Hole,Hole) --> proper_noun.
noun_phrase(Hole,Hole) -->
      determiner, noun, relative.
noun_phrase(Hole0,Hole) -->
      determiner, noun, prep_phrase(Hole0,Hole).
noun_phrase(trace,nil) --> trace.                              (2)

trace --> [].

verb_phrase(Hole) -->
      verb, noun_phrase(Hole,nil).
verb_phrase(nil) --> verb.

relative --> [].
relative -->
      rel_pronoun, sentence(trace).

prep_phrase(Hole0,Hole) -->
      preposition, noun_phrase(Hole0,Hole).
```

A variable "Hole..." will have the value trace if an extraposed noun phrase occurs somewhere to the right, nil otherwise. The parse tree of Figure 3-1 shows the variable values when this grammar is used to analyse the noun phrase "the man that John met".

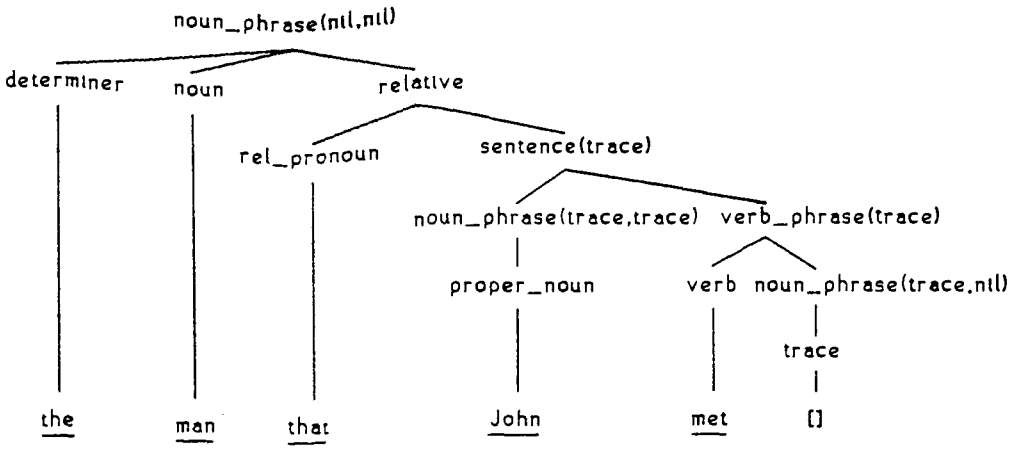


Figure 3-1: DCG parse tree

Intuitively, we either can see noun phrases moving to the left, leaving traces behind, or traces appearing from markers and moving to the right. In a phrase "noun_phrase(Hole1,Hole2)", Hole1 will have the value trace when a trace occurs somewhere to the right of the left end of the phrase. In that case, Hole2 will be nil if the noun phrase contains the trace, trace if the trace appears to the right of the right end of this noun phrase. Thus, rule (2) above specifies that a noun phrase expands into a trace if a trace appears from the left, and as this trace is now placed, it will not be found further to the right.

The non-terminal relative has no arguments, because the complex-NP

constraint prevents noun phrases from moving out of a relative clause. However, that constraint does not apply to prepositional phrases, so prep_phrase has arguments. The non-terminal sentence (and consequently verb_phrase) has a single argument because in a relative clause the trace must occur in the sentence immediately to the right of the relative pronoun.

In a more extensive grammar, many non-terminals would need extraposition arguments, and the increased complication would make the grammar larger and less readable.

Colmerauer's MG formalism allows an alternative way to express left extraposition. It involves the use of a kind of rule whose left-hand side is a non-terminal followed by a string of "dummy" terminal symbols which do not occur in the input vocabulary. An example of such a rule is:

```
rel_marker, [t] --> rel_pronoun.
```

Its meaning is that rel_pronoun can be analysed as a rel_marker provided that the terminal 't' is added to the front of the input remaining after the rule application. The following grammar, adapted from Dahls's work [Dahl 77], shows the use of dummy left-hand side terminals:

```
np -> det, n, relative.

relative --> [].
relative --> r1, s.                                        (3)

r1 --> r2.                                                 (4)
r1, [s_head, moved_arg] --> r2, arg, s_head.               (5)

r2, [arg] --> rel_pronoun.                                 (6)
```

```
compls --> [].
compls --> compls, arg.
compls --> moved_arg, compls.                    (7)

s --> s_head, compls.

s_head --> [s_head].                             (8)
s_head --> arg, v.

arg --> [arg].                                   (9)
arg --> np.

moved_arg --> [moved_arg].                       (10)
```

These rules describe left extraposition in relative clauses in the following way. A relative clause is made of a marker r1 followed by a sentence s (rule (3)). The marker r1 can be expanded either by rule (4) for an extraposed subject or by rule (5) for an extraposed verb complement. Rule (6) defines the basic relative marker r2, analysing a relative pronoun rel pronoun as a marker r2 and a dummy terminal arg to be placed in front of the remaining input. Using rules (4) and (6) the extraposed noun phrase arg is moved back into subject position (figure 3-2). Using rules (5) and (6), the extraposed noun phrase arg is moved into complement position, jumping over the subject and verb s head of the relative clause and being finally picked by rule (7) (figure 3-3). Rules (8), (9) and (10) reinterpret dummy terminals as the corresponding non-terminals.

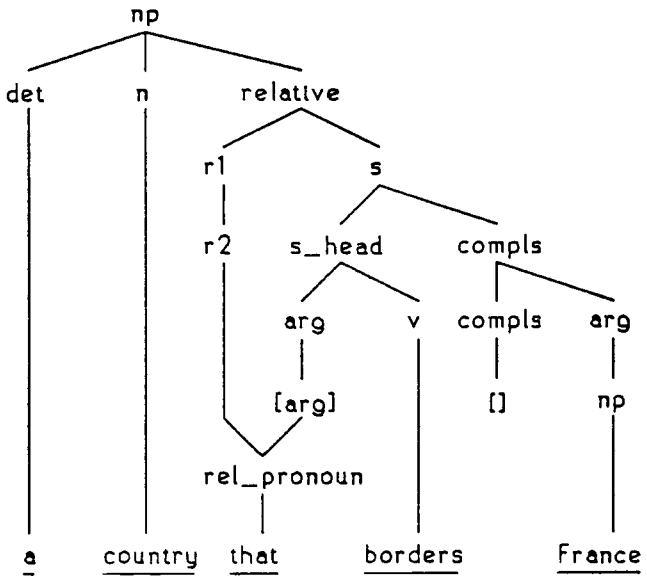


Figure 3-2: Subject movement

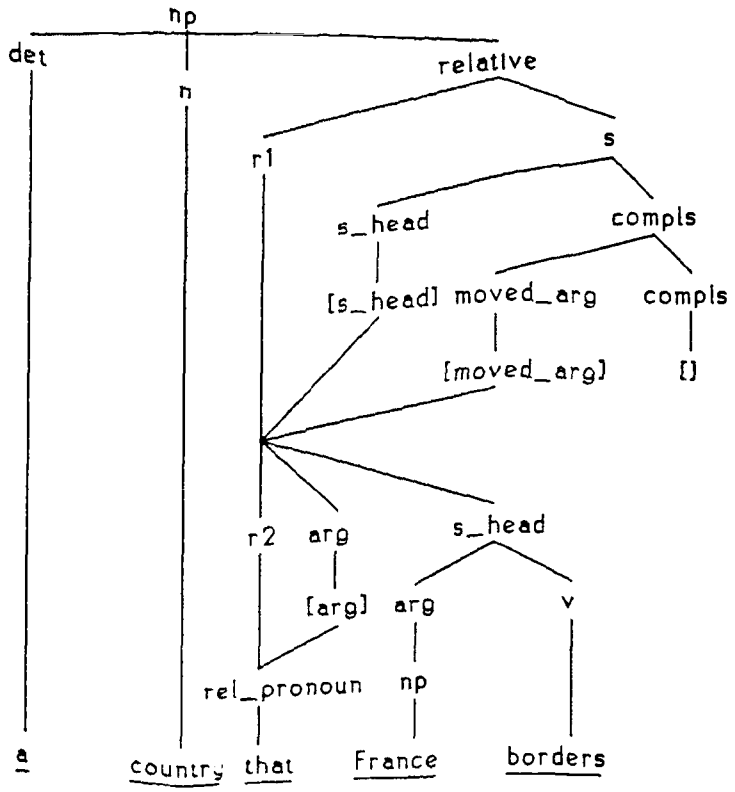


Figure 3-3: Object movement

The constituent s head and rule (5) are needed only because the movement of a constituent over others must be explicitly represented, given that dummy terminals can only be placed back in the input string immediately to the right of the constituent in whose analysis they occur. If this leads to the introduction of the otherwise unmotivated s head constituent in the present example, it becomes much more awkward when trying to describe extraposition from within arbitrarily nested constituents, such as from prepositional phrases or complement clauses. For example, in the sentence

$[_{np}$the bottle $[_{rel}$that$_i[_s$John lost $[_{np}$the top $[_{pp}$of $t_i]\ ]\ ]\ ]\ ]$

the extraposed noun phrase would have to move not only over the subject and verb "John lost" but also over part of the direct object "the top of". This means that the noun phrase rules would have to be prepared to cope with dummy moved arg terminals.

The use of dummy left-hand side terminals suffers also from a theoretical problem. In general, the language defined by such a grammar will contain extra sentences involving the dummy terminals. For parsing, however, no problem arises because the input sentences are not supposed to contain dummy terminals. These inadequacies of MGs were the main motivation for the development of XGs.

3.3 Informal Description of XGs

To describe left extraposition, we need to relate non-contiguous parts of a sentence. But neither DCGs nor MGs have means of representing such a relationship by specific grammar rules. Rather, the relationship can only be described implicitly, by adding extra information to many unrelated rules in the grammar. That is, one cannot look at a grammar and find a set of rules specific to the constructions which involve left extraposition. It is to to allow such rules to be written that I introduce extraposition grammars.

In this informal introduction to the XG formalism, I will avoid the extra complications of non-terminal arguments. So, in the discussion that follows, we may look at XGs as an extension of CFGs.

Sometimes it is easier to look at grammar rules in the left-to-

right, or synthesis, direction. I will say then that a rule is being used to **expand** or rewrite a string. In other cases, it is easier to look at a rule in the right-to-left, or analysis, direction. I will say then that the rule is being used to **analyse** a string.

Let us first look at the following XG fragment:

```
sentence --> noun_phrase, verb_phrase.

noun_phrase --> determiner, noun, relative.
noun_phrase --> trace.

relative --> [].
relative --> rel_marker, sentence.

rel_marker ... trace --> rel_pronoun.
```

All rules but the last are context-free. The last rule expresses the extraposition in simple relative clauses. It states that a relative pronoun is to be analysed as a marker, followed by some unknown constituents (denoted by '...'), followed by a trace. This is shown in Figure 3-4.

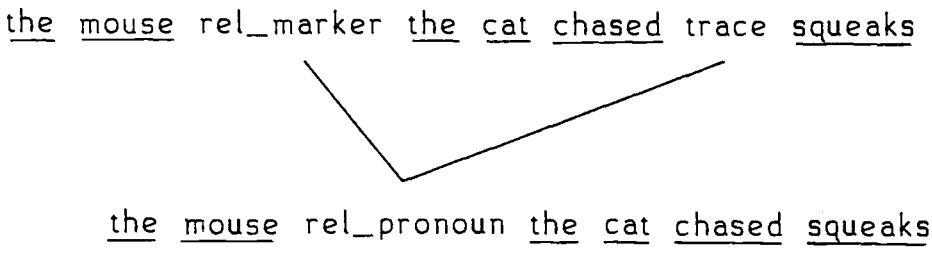


Figure 3-4: Applying an XG rule

As in the DCG example of the previous section, the extraposed noun

phrase is expanded into a trace. However, instead of the trace being rewritten into the empty string, the trace is used as part of the analysis of rel pronoun.

The difference between XG rules and DCG rules is that the left-hand side of an XG rule may contain several symbols. Where a DCG rule is seen as expressing the expansion of a single non-terminal into a string, an XG rule is seen as expanding together several non-contiguous symbols into a string. More precisely, an XG rule has the general form

$$s_1 \ldots s_2 \text{ etc. } s_{k-1} \ldots s_k \text{ --> } r. \qquad (11)$$

Here each segment s_i (separated from other segments by '...') is a sequence of terminals and non-terminals (written in DCG notation, with ',' for concatenation). The first symbol in s_1, the leading symbol, is restricted to be a non-terminal. The right-hand side r of the rule has the same form as the right-hand side of a DCG rule.

Leaving aside the constraints discussed in the next section, the meaning of a rule like (11) is that any sequence of symbols of the form

$$s_1 x_1 s_2 x_2 \text{ etc. } s_{k-1} x_{k-1} s_k$$

with arbitrary x_i's, can be rewritten into $r\ x_1 x_2 \ldots x_{k-1}$. In other words, the right sisters of the leading symbol and of its ancestors must expand in such a way that between them they will cover the rest of the left-hand side of the rule.

Thinking procedurally, one can say that a non-terminal may be expanded by matching it to the leading symbol on the left-hand side

of a rule, and the rest of the left-hand side is "put aside" to wait for the derivation of symbols which match each of its symbols in sequence. This sequence of symbols can be interrupted by arbitrary strings, paired to the occurrences of '...' on the left-hand side of the rule.

The above definition of the effect of an XG rule allows the symbols that match the non-leading symbols of the left-hand side of the rule to come from positions arbitrarily up and to the right in the derivation. This contrasts with what happens in context-sensitive grammars [Joshi 77], where the symbols that match the context of a rule may come from positions arbitrarily up in the derivation, but must be adjacent to the symbol being expanded. The effect of this is that XG rules without arguments, even when used for analysis only, can recognise non-context free languages (see the example in Section 3.5).

3.4 XG Derivations

When several XG rules are involved, the derivation of a surface string becomes more complicated than in the single rule example of the previous section because rule applications interact in the way now to be described.

To represent the intermed e stages in an XG derivation, I will use **bracketed strings**, made up of

* terminal symbols
* non-terminal symbols
* **the open bracket <**

* the close bracket >

A bracketed string is balanced if the brackets in it balance in the usual way.

Now, an XG rule

$u_1 \ldots u_2 \ldots$ etc. $\ldots u_n$ --> v.

can be applied to bracketed string s if

$s = x_0 u_1 x_1 u_2$ etc. $x_{n-1} u_n x_n$

and each of the gaps $x_1, \ldots, x_{n-1}$ is balanced. The substring of s between x_0 and x_n is the span of the rule application. The application rewrites s into new string t, replacing u_1 by v followed by n-1 open brackets, and replacing each of $u_2, \ldots, u_n$ by a close bracket; in short s is replaced by $x_0 v << \ldots < x_1 > x_2 > \ldots x_{n-1} > x_n$.

The relation between the original string s and the derived string t is abbreviated as s => t. In the new string t, the substring between x_0 and x_n is the result of the application. In particular, the application of a rule with a single segment in its left-hand side is no different from what it would be in a type-0 grammar.

Taking again the rule

rel_marker ... trace --> rel_pronoun.

its application to

rel_marker John likes trace

produces

rel_pronoun < John likes >

After this rule application, it is not possible to apply any rule with a segment matching inside a bracketed portion and another segment matching outside it. The use of the above rule has divided the string into two isolated portions, each of which must be independently expanded.

Given an XG with initial symbol s, a sentence t is in the language defined by the XG if there is a sequence of rule applications which transforms s into a string from which t can be obtained by deleting all brackets.

I shall refer to the restrictions on XG rule application which I have just described as the bracketing constraint. The effect of the bracketing constraint is independent of the order of application of rules, because if two rules are used in a derivation, the brackets introduced by each of them must be compatible in the way described above. As brackets are added and never deleted, it is clear that the order of application is irrelevant. For similar reasons, for any two applications in a derivation where the rules involved have more than one segment in their left-hand sides, one and only one of the two following situations arises:

* the span of neither application intersects the result of the other;
* the result of one of the applications is contained entirely in a gap of the other application -- the applications are nested.

If one follows to the letter the definitions in this section, then checking, in a parsing procedure, whether an XG rule may be applied, would require a scan of the whole intermediate string. However, we

will see in Section 3.8 that this check may be done "on the fly" as brackets are introduced, with a cost independent of the length of the current intermediate string in the derivation.

3.5 Derivation Graphs

In the same way as parse trees are used to visualise context-free derivations, I use derivation graphs to represent XG derivations.

In a derivation graph, as in a parse tree, each node corresponds to a rule application or to a terminal symbol in the derived sentence, and the edges leaving a node correspond to the symbols in the right-hand side of that node's rule. In a derivation graph, however, a node can have more than one incoming edge - in fact, one such edge for each of the symbols on the left-hand side of the rule corresponding to that node. Of these edges, only the the one corresponding to the leading symbol is used to define the left-to-right order of the symbols in the sentence whose derivation is represented by the graph. If one deletes from a derivation graph all except the first of the incoming edges to every node, the result is a tree analogous to a parse tree.

For example, Figure 3-5 shows the derivation graph for the string "aabbcc" according to the XG:

```
s --> as, bs, cs.

as --> [].
as ... xb --> [a], as.

bs --> [].
bs ... xc --> xb, [b], bs.

cs --> [].
cs --> xc, [c], cs.
```

This XG defines the language formed by the set of all strings

$a^n b^n c^n$ for $n \geq 0$.

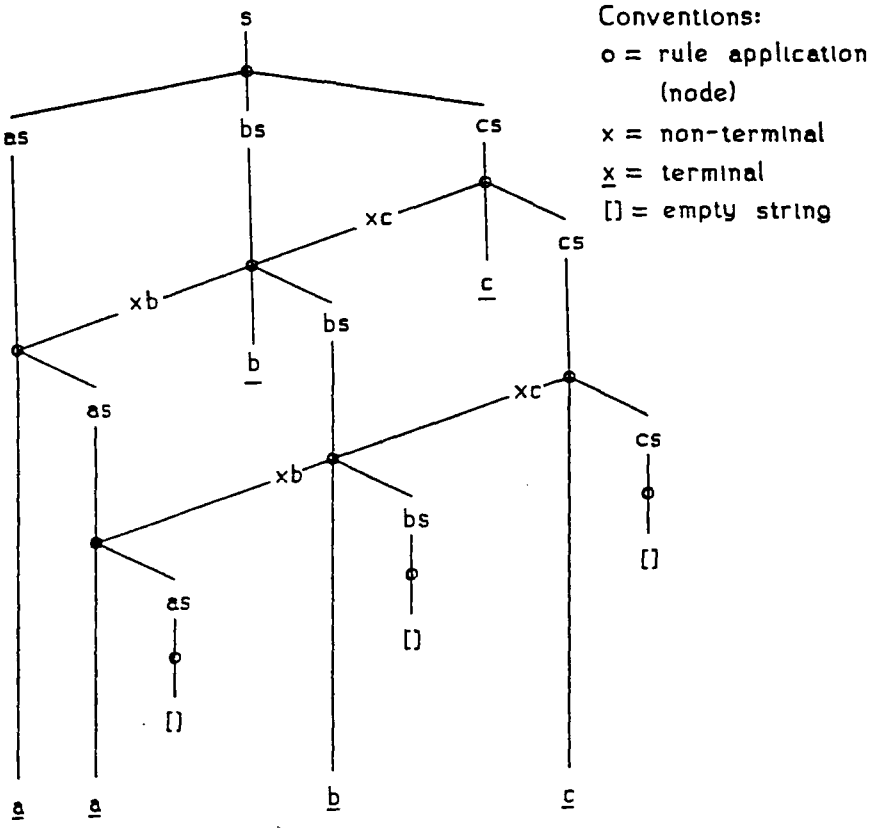


Figure 3-5: Derivation graph for "aabbcc"

The language described is not context-free, showing that XGs, even without arguments, are strictly more powerful than context-free grammars.

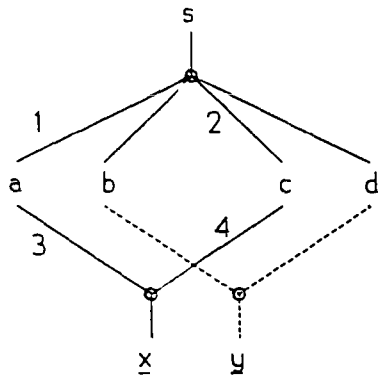
The topology of derivation graphs reflects clearly the bracketing constraint. Assume the following two conventions for the drawing of a derivation graph, which are followed in all the graphs shown here:

* the edges entering a node are ordered clockwise following the sequence of the corresponding symbols in the left-hand side of the rule for that node;

* the edges issuing from a node are ordered counterclockwise following the sequence of the corresponding symbols in the right-hand side of the rule for the node.

Then the derivation graph obeys the bracketing constraint if and only if it can be drawn, following the conventions, without any edges crossing[14]. The example of Figure 3-6 shows this clearly.

```
s --> a, b, c, d.
a ... c --> [x].
b ... d --> [y].
```




```
s => a b c d => x < b > d => ? (blocks)
s => a b c d => a y < c > => ?
```

Figure 3-6: Relating derivations to derivation graphs

In this figure, the closed path formed by edges 1, 2, 3 and 4 has the same effect as a matching pair of brackets in a bracketed string.

---

[14] In some of the examples of this work, edges cross to make the graphs more readable, but such crossings could be trivially avoided.

It is also worth noting that nested rule applications appear in a derivation graph as a configuration like the one depicted in Figure 3-7.

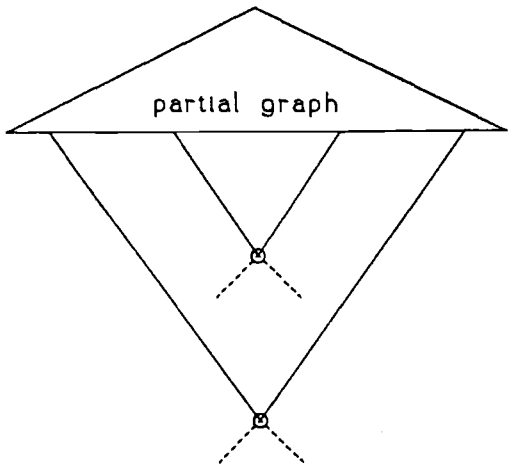


Figure 3-7: Nested rule applications

3.6 XGs and Left Extraposition

We saw in Section 3.2 a DCG for (some) relative clauses. The XG below describes essentially the same language fragment, showing how easy it is to describe left extraposition in an XG.

```
sentence --> noun_phrase, verb_phrase.

noun_phrase --> proper_noun.
noun_phrase --> determiner, noun, relative.
noun_phrase --> determiner, noun, prep_phrase.
noun_phrase --> trace.

verb_phrase --> verb, noun_phrase.
verb_phrase --> verb.
```

```
relative --> [].
relative --> rel_marker, sentence.                    (12)

rel_marker ... trace --> rel_pronoun.

prep_phrase --> preposition, noun_phrase.
```

In this grammar, the sentence

The mouse that the cat chased squeaks.

has the derivation graph shown in Figure 3-8.

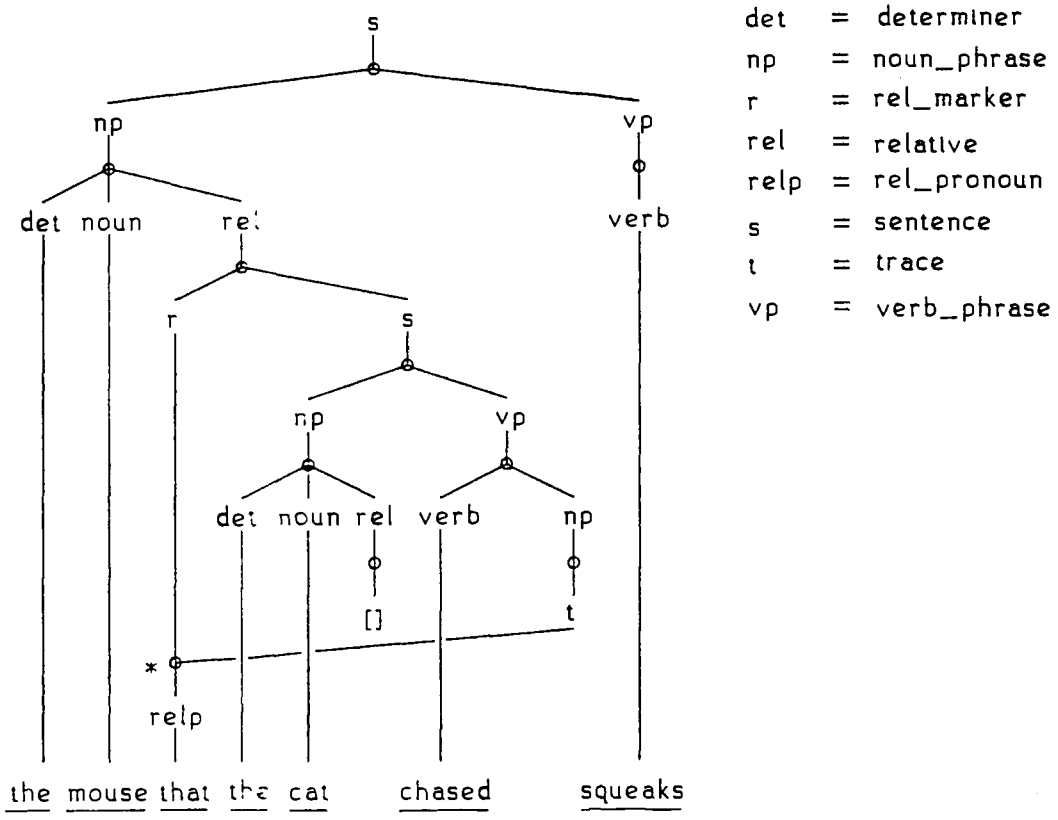


Figure 3-8: Example of derivation graph for the XG above

The left extraposition implicit in the structure of the sentence is

represented in the derivation graph by the application of the rule for rel marker, at the node marked (*) in the figure. One can say that the left extraposition has been "reversed" in the derivation by the use of this rule, which may be looked at as repositioning trace to the right, thus "reversing" the extraposition of the original sentence.

In the rest of this work, I will often refer to a constituent being repositioned into a bracketed string, or into a fragment of derivation graph, to mean that a rule having that constituent as a non-leading symbol in the left-hand side has been applied, and the symbol matches some symbol in the string (or corresponds to some edge in the fragment). For example, in Figure 3-8 the trace t is repositioned into the subgraph with root s.

3.7 Using the Bracketing Constraint

In the example of Figure 3-8, there is only one application of a non-DCG rule, at the place marked (*). However, we have seen that when a derivation contains several applications of such rules, the applications must obey the bracketing constraint. The use of the constraint in a grammar is better explained with an example. From the sentences

The mouse squeaks.
The cat likes fish.
The cat chased the mouse.

the grammar of Section 3.6 can derive the following string, which violates the complex-NP constraint:

* The mouse that the cat that chased likes fish squeaks.

The derivation graph for that non-English string is shown in Figure 3-9.

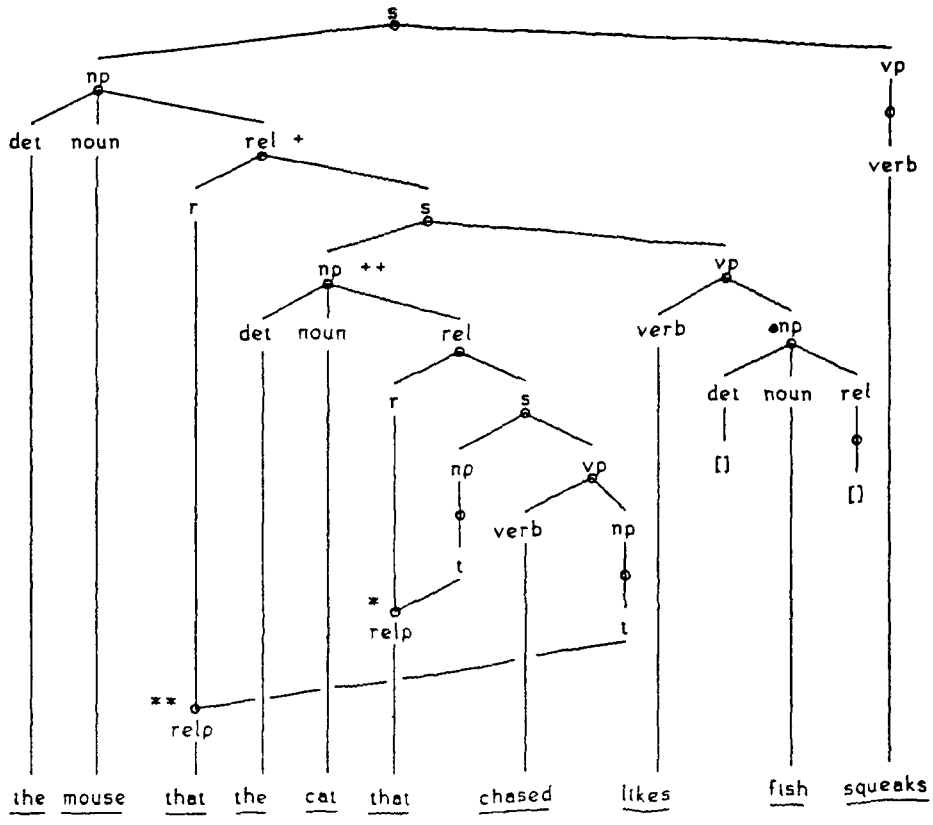


Figure 3-9: Violation of the complex-NP constraint

In the graph, (*) and (**) mark two nested applications of the rule for rel_marker. The string is non-English because the higher relative (marked (+) in the graph) binds a trace occurring inside a sentence which is part of the subordinated noun phrase (++).

Now, using the bracketing constraint one can neatly express the complex-NP constraint. It is only necessary to change the second rule for relative given in Section 3.6

```
relative -> rel_marker, sentence.
```

to

```
relative --> open, rel_marker, sentence, close.          (13)
```

and add the rule

```
open ... close --> [].                                   (14)
```

With this modified grammar, it is no longer possible to violate the complex-NP constraint, because no constituent can be repositioned from outside into the gap created by the application of rule (14) to the result of applying the rule for relatives (13).

The non-terminals open and close bracket a subderivation

```
... open x close ... => ... < x > ...
```

preventing any constituent from being repositioned from outside that subderivation into it. The bracketed subderivation is just what Ross [Ross 74] calls an "island". Figure 3-10

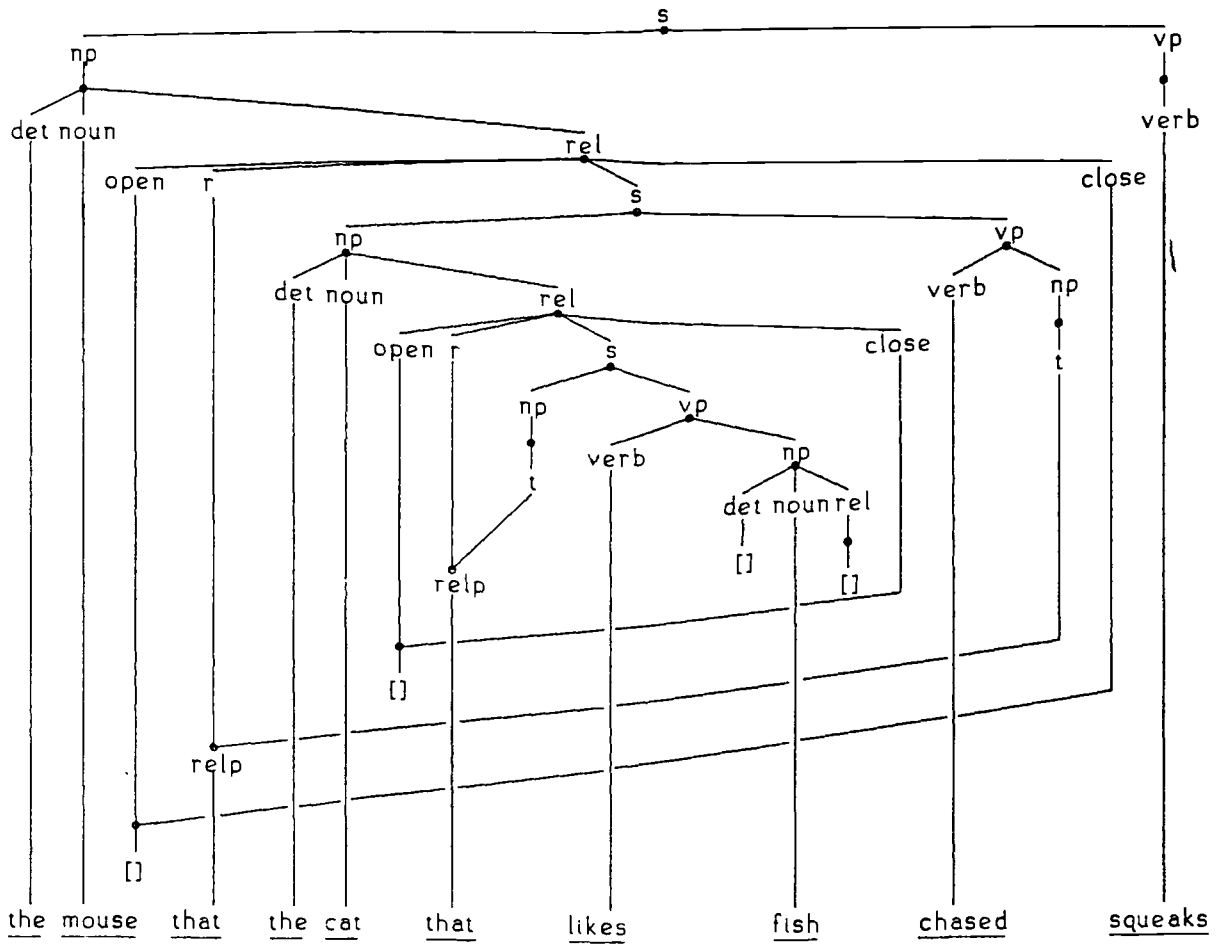


Figure 3-10: Implementation of the complex-NP constraint

shows the use of rule (14) in the derivation of the sentence

The mouse that the cat that likes fish chased squeaks.

This is based on the same three simple sentences as the ungrammatical

string of Figure 3-9. Any attempt at deriving the ungrammatical string with the revised grammar will fail because the extraposition of the outer relative clause would have to cross the edges for the open-close pair of the inner relative clause.

3.8 XGs as Logic Programs

In the previous sections, I avoided the complication of non-terminal arguments. Although it would be possible to describe fully the operation of XGs in terms of derivations on bracketed strings, it is much simpler to complete the explanation of XGs using the translation of XG rules into definite clauses. In fact, a rigorous definition of XGs independently of definite clauses would require a formal apparatus very similar to the one needed to formalise definite clause programs in the first place, and so it would fall outside the scope of the present work. A full discussion of those issues can be found in two articles by Colmerauer [Colmerauer 78, Colmerauer 79b].

Like a DCG, a general XG is no more than a convenient notation for a set of definite clauses. An XG non-terminal of arity n corresponds to an n+4 place predicate (with the same name). Of the extra four arguments, two are used to represent string positions as in DCGs, and the other two are used to represent positions in an extraposition list, which carries symbols to be repositioned.

Each element of the extraposition list represents a symbol being repositioned as a 4-tuple

x(context, type, symbol, xlist)

where context is either gap, if the symbol was preceded by '...' in

the rule where it originated, or nogap, if the symbol was preceded by ',' ; type may be terminal or nonterminal, depending on whether symbol was a terminal (marked by '[' ']') or a non-terminal in the rule left-hand side where it comes from; symbol is the symbol proper; xlist is the remainder of the extraposition list (an empty list being represented by '[]').

An XG rule is translated into a clause for the predicate corresponding to the leading symbol of the rule. In the case where the XG rule has just a single symbol on the left-hand side, the translation is very similar to that of DCG rules. For example, the rule

```
sentence --> noun_phrase, verb_phrase.
```

translates into

```
sentence(S0,S,X0,X) :-
      noun_phrase(S0,S1,X0,X1), verb_phrase(S1,S,X1,X).
```

A terminal t in the right-hand side of a rule translates into a call to the predicate terminal, defined below, whose role is analogous to that of connects in DCGs. For example, the rule

```
rel_pronoun --> [that].
```

translates into

```
rel_pronoun(S0,S,X0,X) :- terminal(that,S0,S,X0,X).
```

The translation of a rule with more than one symbol in the left-hand side is a bit more complicated. Informally, each symbol after the first is made into a 4-tuple as described above, and fronted to the extraposition list. Thus, for example, the rule

```
rel_marker ... trace --> rel_pronoun.
```

translates into

```
rel_marker(S0,S,X0,x(gap,nonterminal,trace,X) ) :-
      rel_pronoun(S0,S,X0,X).
```

Furthermore, for each distinct non-leading non-terminal **nt** (with arity n) in the left-hand side of a rule of the XG, the translation includes the clause

```
nt(V1,...,Vn,S,S,X0,X) :- virtual(nt(V1,...,Vn),X0,X).
```

where "virtual(C,X0,X)", defined later, can be read as "C is the constituent between X0 and X in the extraposition list", and the variables Vi transfer the arguments of the symbol in the extraposition list to the predicate which translates that symbol.

For example, the rule

```
marker(Var), [the] ... [of,whom], trace(Var) --> [whose].
```

which can be used in a more complex grammar of relative clauses to transform "whose X" into "the X of whom", corresponds to the clauses:

```
marker(Var,S0,S,X0,x(nogap,terminal,the,
                     x(gap,terminal,of,
                     x(nogap,terminal,whom,
                     x(nogap,nonterminal,trace(Var),
                     X )))) ) :-
      terminal(whose,S0,S,X0,X).

trace(Var,S,S,X0,X) :- virtual(trace(Var),X0,X).
```

Finally, the two auxiliary predicates <u>virtual</u> and <u>terminal</u> are defined as follows:

```
virtual(NT, x(C,nonterminal,NT,X), X).

terminal(T, S0, S, X, X) :- gap(X), connects(S0, T, S).
terminal(T, S, S, x(C,terminal,T,X), X).
```

```
gap(x(gap,T,S,X)).
gap([]).
```

where the definition of connects is the same as for DCGs.

I will now explain these definitions. The clause for virtual extracts a non-terminal NT from the extraposition list, returning the rest of the list X. The first clause for terminal consumes a terminal T from the input string S0 returning the rest of the string S, provided that the current extraposition list X allows a gap to appear in the derivation. Predicate connects actually takes the next terminal off the input, and predicate gap tests for a gap in the extraposition list. The alternative clause for terminal also consumes a terminal T, this time from the extraposition list instead of from the input string. The first symbol in the current extraposition list must of course have the type terminal for this to happen, and the rest X of the extraposition list is returned. The input string S is returned unchanged. The first clause for gap states that a gap is allowed if the first symbol in the current extraposition list has gap as its context feature. The second clause states that a gap is always allowed if the current extraposition list is empty (that is no extrapositions are involved and therefore the use of terminals is just like in DCG rules). Figure 3-11 shows a fragment of the analysis in Figure 3-10, but now in terms of the translation of XG rules into definite clauses. Points on the sentence are labelled as follows:

```
 the mouse that the cat that likes fish chased  squeaks
1   2     3    4   5   6    7     8     9      10      11
```

The nodes of the analysis fragment, for the relative clause "that

likes fish", are represented by the corresponding goals, indented in proportion to their distance from the root of the graph. Because this is a sub-phrase of a larger relative clause, there are two symbols in the extraposition list which are left there at the end of the analysis, the trace and close bracket of the outer relative. To simplify the figure, the values of the extraposition arguments are explicitly represented only for those goals that add or delete something to the extraposition list; for the other goals, the two identical values are represented by the variable X. Also, the sub-goals of terminal are not shown.

The definite clause program corresponding to the grammar for this example is listed in Appendix B.

The example shows how the bracketing constraint works. Symbols are placed in the extraposition list by rules with more than one symbol in the left-hand side, and removed by calls to virtual, on a first-in-last-out basis, that is, the extraposition list is a stack. But this property of the extraposition list is exactly what is needed to balance "on the fly" the auxiliary brackets in the intermediate steps of a derivation.

Being no more than a logic program, an XG can be used for analysis and for synthesis in the same way as a DCG. For instance, to determine whether a string s with initial point initial and final point final is in the language defined by the XG of Section 3.6, one tries to prove the goal statement

```
?- sentence(initial,final,[],[]).
```

```
* relative(6,9,X,X)
*  open(6,6,x(gap,nonterminal,trace,x(gap,nonterminal,close,[])),
       x(gap,nonterminal,close,x(gap,nonterminal,trace,
       x(gap,nonterminal,close,[]))))
*  rel_marker(6,7,x(gap,nonterminal,close,x(gap,nonterminal,trace,
       x(gap,nonterminal,close,[]))),
       x(gap,nonterminal,trace,x(gap,nonterminal,close,
       x(gap,nonterminal,trace,x(gap,nonterminal,close,[])))))
*   rel_pronoun(6,7,X,X)
*    terminal(that,6,7,X,X)
*  sentence(7,9,x(gap,nonterminal,trace,x(gap,nonterminal,close,
       x(gap,nonterminal,trace,x(gap,nonterminal,close,[])))),
       x(gap,nonterminal,close,x(gap,nonterminal,trace,
       x(gap,nonterminal,close,[]))))
*   noun_phrase(7,7,x(gap,nonterminal,trace,x(gap,nonterminal,close,
        x(gap,nonterminal,trace,x(gap,nonterminal,close,[])))),
        x(gap,nonterminal,close,x(gap,nonterminal,trace,
        x(gap,nonterminal,close,[]))))
*    trace(7,7,x(gap,nonterminal,trace,x(gap,nonterminal,close,
         x(gap,nonterminal,trace,x(gap,nonterminal,close,[])))),
         x(gap,nonterminal,close,x(gap,nonterminal,trace,
         x(gap,nonterminal,close,[]))))
*     virtual(trace,x(gap,nonterminal,trace,x(gap,nonterminal,close,
         x(gap,nonterminal,trace,x(gap,nonterminal,close,[])))),
         x(gap,nonterminal,close,x(gap,nonterminal,trace,
         x(gap,nonterminal,close,[]))))
*   verb_phrase(7,9,X,X)
*    verb(7,8,X,X)
*     terminal(likes,7,8,X,X)
*    noun_phrase(8,9,X,X)
*     determiner(8,8,X,X)
*     noun(8,9,X,X)
*      terminal(fish,8,9,X,X)
*     relative(9,9,X,X)
*  close(9,9,x(gap,nonterminal,close,x(gap,nonterminal,trace,
       x(gap,nonterminal,close,[]))),
       x(gap,nonterminal,trace,x(gap,nonterminal,close,[])))
```

Figure 3-11: Derivation of "that likes fish"

As for DCGs, the string s can be represented in several ways. If it is represented as a list, the above goal would be written

```
?- sentence(s,[],[],[]).
```

The last two arguments of the goal are '[]' to mean that the overall extraposition list goes from '[]' to '[]', that is, it is empty. Thus, no constituent can be repositioned into or out of the top level sentence.

3.9 Using XGs

I will now discuss in detail an XG which employs all the techniques described so far. The XG is based on that used in Chat-80, with the omission of most of the non-terminal arguments as well as many of the less interesting rules. Arguments are used in the Chat-80 grammar for agreement checks, for syntactic features, for producing a parse tree and to restrict the attachment possibilities of post-modifiers. The first two uses have been sufficiently discussed elsewhere [Pereira and Warren 80, Colmerauer 78, Dahl 77]. The use of arguments to control attachment will be discussed in the next section. The use of arguments for syntactic features will be seen in the rules that follow.

The use of the rules is shown in example parse graphs, in which each node is labelled with the name of the corresponding rule[15].

Two particular points of the XG notation should be kept in mind: the symbol '_' denotes an anonymous unspecified value; and a term "{goal}" in the body of a rule denotes a condition that must be satisfied for the rule to apply, where goal is a call to some definite clause predicate.

The initial non-terminal of the grammar is sentence . A sentence may be a declarative sentence, a yes-no question or a WH-question:

[15] Parts of the graphs and non-terminals not relevant for the rules being exemplified are omitted.

```
sentence --> declarative.                                   [sent-1]
sentence --> wh_question.                                   [sent-2]
sentence --> yn_question.                                   [sent-3]

declarative --> s.                                          [decl]
```

The WH-questions considered here involve either the extraposition of a noun phrase np or of a prepositional phrase pp:

```
wh_question --> q_marker(Case), question(Case).             [wh-quest]

q_marker(Case) ... np(Case) --> whq(Case).                  [q-mark-1]
q_marker(compl) ... pp --> prep, whq(compl).                [q-mark-2]

whq(Case) --> int_det(Case), np(_).                         [whq-1]
whq(Case) --> int_pron(Case).                               [whq-2]
```

The argument of q marker is the "case" feature of the interrogated noun phrase. This feature is determined by the position of the trace of the extraposed WH-phrase, taking the value subj for a trace in subject position and compl for a trace that is a verb or noun phrase complement. The distinction determines whether there will be an auxiliary verb inversion in the sentence. Of course, a WH-question introduced by a fronted prepositional phrase must be a question over some complement. But in general the case argument of q marker is obtained from the corresponding argument of the extraposed noun phrase, which is itself set by the rules that have the non-terminal np in their right-hand sides (see the rules for subj, pp and verb args below).

A WH-word phrase whq can be a noun phrase with an interrogative determiner int det or an interrogative pronoun int pron (figures 3-12 and 3-13).

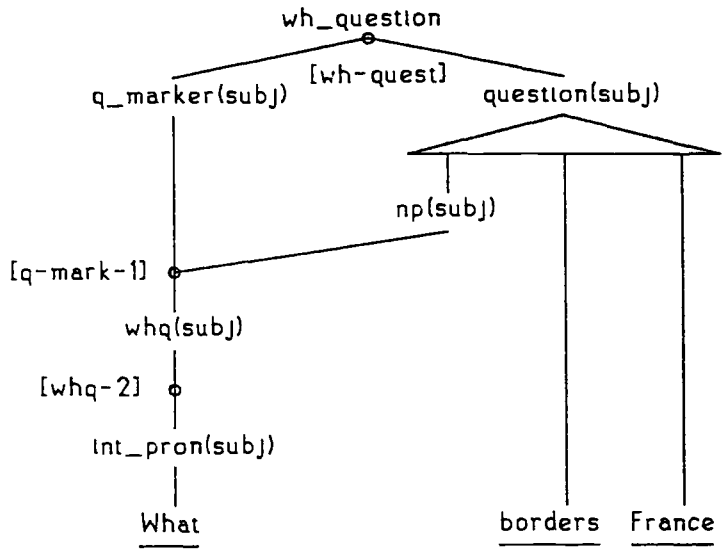


Figure 3-12: Subject question

Rule [q-mark-2] deals with questions with a fronted prepositional phrase such as "In what country does the Danube rise?" (figure 3-13). The alternative formulation of that question, "What country does the Danube rise in?", is also covered, by the first q marker rule, which will reposition the trace of the questioned noun phrase "What country" into the complement place of the preposition "in".

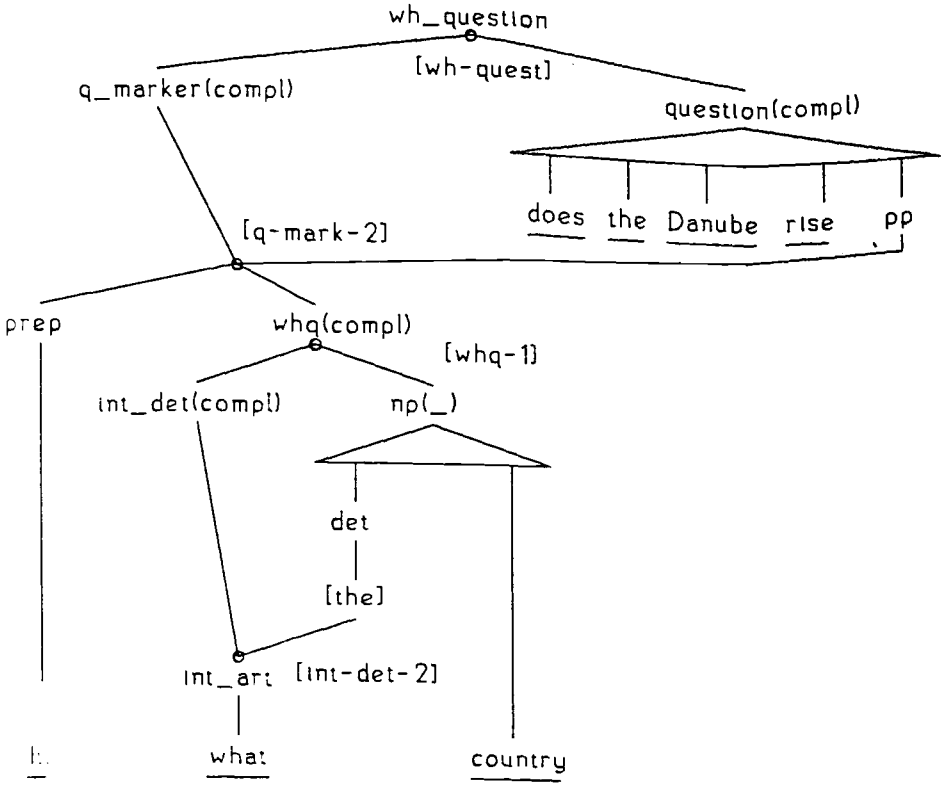


Figure 3-13: Prepositional phrase question

An interrogative determiner may be an interrogative article int_art, as in "what country...", or the word "whose", as in "whose book is this?":

```
int_det(compl) --> whose.                                  [int-det-1]
int_det(_), [the] --> int_art.                             [int-det-2]

whose, simple_np(proper), gen_marker --> [whose].          [whose]
```

Rule [int-det-2] introduces the article "the" in place of the interrogative article, to be used in the questioned noun phrase. The whose rule analyses "whose" as a dummy proper noun phrase

"simple_np(proper)" followed by a genitive marker gen_marker. This transformation builds a possessive "prefix" which modifies the fragment of noun phrase after "whose", changing "whose book..." into a partially analysed form of "X's book..." (figure 3-14).

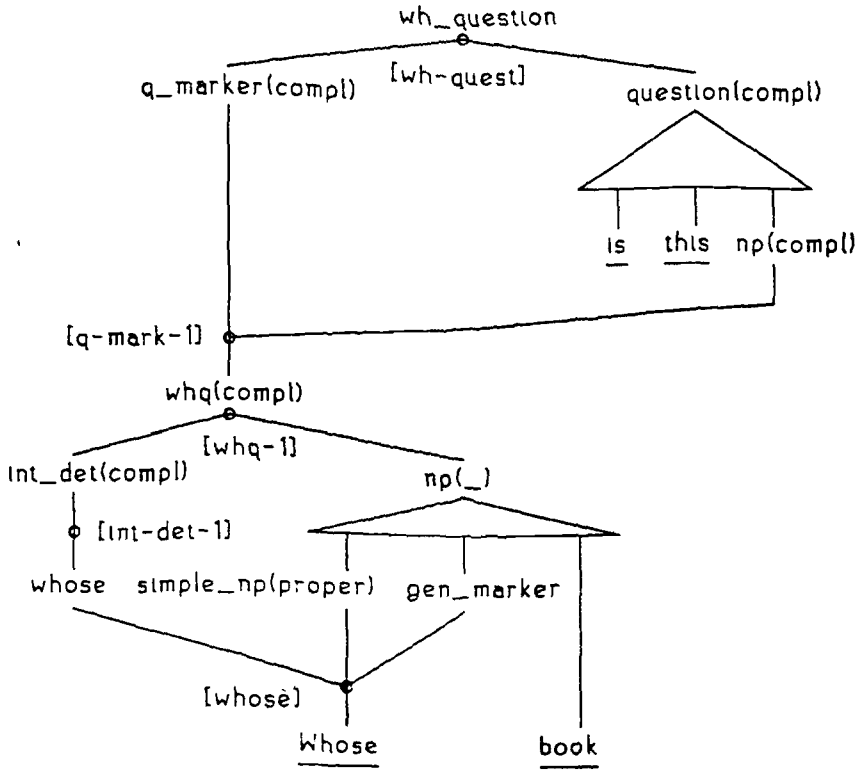


Figure 3-14: "Whose"

Notice that the rules for WH-questions given here, and those for relative clauses given below, describe only the *syntax* of those constructions. In the actual Chat-80 grammar (Appendix C), further arguments are used to represent the structure of phrases in a tree form convenient for semantic interpretation, and in particular to

introduce variables that link a marker to the corresponding trace. For instance, the rules above would be augmented as follows:

```
wh_question(whq(X,P)) -->
   q_marker(Case,X), question(Case,P).                    [wh-quest']

q_marker(Case,X) ... np(Case,NP) -->
   whq(Case,X,NP).                                        [q-mark-1']
q_marker(compl,X) ... pp(pp(P,NP)) -->
   prep(P), whq(compl,X,NP).                              [q-mark-2']

whq(Case,X,NP) --> int_det(Case,X), np(_,NP).             [whq-1']
whq(Case,X,wh(X)) --> int_pron(Case).                     [whq-2']

int_det(compl,X) --> whose(X).                            [int_det-1']
int_det(_,X), det(Det) --> int_art(X,Det).                [int-det-2']

whose(X), simple_np(proper,wh(X)), gen_marker
   --> [whose].                                           [whose']
```

In these rules, variable X stands for the meaning (as yet unknown) of the questioned noun phrase, that is, the actual answer to the question. The second argument of np and simple_np is the structure that will represent the noun phrase in the input to the semantic interpreter. In particular, the structure "wh(X)" represents a trace, and denotes to the semantic interpreter an object whose meaning is its argument X. Whereas an interrogative pronoun stands for a completely unspecified object, and therefore its trace has just the representation "wh(X)" (rule [whq-2']), an interrogative determiner is associated to a partially specified entity given by the noun phrase it belongs to. Because of the way the meaning of a noun phrase is built around its determiner (see Chapter 4), interrogative articles (rule [int-det-2']) encode the meaning variable X into the determiner representation Det. Note that the terminal "the" in the left-hand side of the rule has now been replaced by the general determiner non-terminal det, which will carry a structure Det built from the representation of the determiner "the" and the meaning

variable X. A related situation occurs for "whose" interrogatives (rule [whose']). As remarked earlier, "whose N" is analysed as "X's N", and therefore the noun phrase simple_np, modified by the implicit genitive, is given the structure "wh(X)".

For the very sketchy treatment of verbs in the present grammar, two verb features are enough. The aux feature determines whether the verb can be used as an auxiliary, and the args feature what arguments the verb takes. This feature pair is represented in the rules by the term "aux+args", where aux will be aux for an auxiliary verb and main for other verbs, and args takes one of the values intrans, trans, be or have, to specify that the verb is intransitive, transitive or one of the special verbs "to be" and "to have".

As I explained above, auxiliary inversion in an WH-question is determined by the case of the interrogated noun phrase. A yes-no question is always inverted. An auxiliary inversion fronted_verb involves the repositioning of the fronted auxiliary, which is an inflected verb form verb_form of a verb 'Root' with the aux feature (specified by a verb_type condition). The negation neg which might follow the fronted verb must, of course, be moved with it[16]. The inversion rules are:

```
yn_question --> question(compl).                    [yn-quest]

question(subj) --> s.                               [quest-1]
question(compl) --> fronted_verb, s.                [quest-2]
```

[16]The arguments of neg will be explained later, with the verb rules.

```
fronted_verb ... verb_form(Root), neg(Type,Neg) -->          [front]
   verb_form(Root),
   {verb_type(Root,aux+_)},
   neg(Type,Neg).
```

Figure 3-12 gives the analysis of a non-inverted question, and figure 3-15 the analysis of an inverted question.

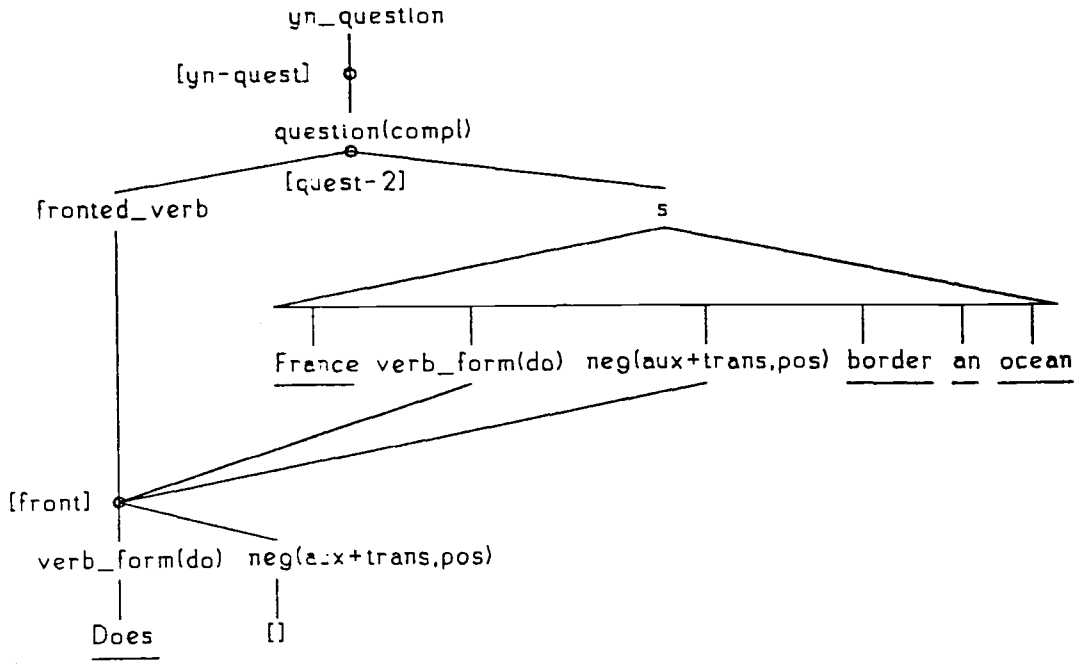


Figure 3-15: Yes-no question

The basic sentence s is defined by the rules:

```
s -->                                                        [s]
   subj(Type),
   vp(Type).

subj(_+be) --> [there].                                      [subj-1]
subj(_) --> np(subj).                                        [subj-2]
```

where 'Type' is the feature pair of the main verb. Only if the **args** feature has the value _be_ is the existential "there" allowed as subject (figure 3-16).

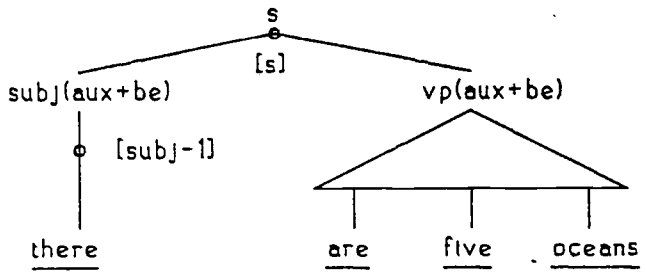


Figure 3-16: Existential statement

We will look now at some noun phrase rules, simplified from those in Chat-80 (for instance, rules for pronouns and partitives are omitted):

```
np(_) --> np_head(Type), np_compls(Type).                    [np]

np_head(Type) -->                                            [np-head]
   simple_np(Type0), possessive(Type0,Type).

simple_np(proper) --> name.                                  [simple-1]
simple_np(common) --> determiner, adjs, noun.                [simple-2]

possessive(_,Type) -->                                       [poss-1]
   gen_case,
   simple_np(Type0),
   possessive(Type0,Type).
possessive(Type,Type) --> [].                                [poss-2]

gen_case, [the] --> gen_marker.                              [gen]

np_compls(proper)  --> [].                                   [np-comp-1]
np_compls(common) --> np_mods, relative.                     [np-comp-2]
```

As noted before, an np has a case argument which is used to relate the position where a trace np occurs to the relative or interrogative construction bound to that trace. We have seen above how the argument is used to distinguish between WH-questions with and without inversion. A non-extraposed noun phrase, however, is not affected by where it occurs, and therefore the case argument in the main np rule is the don't-care '_'.

An np has an np head which may be followed by some complements np compls if the head noun is a common noun (the argument of np head is common). An np head may contain a possessive construction, in which case the simple np which starts it will be the "possessor", marked by a gen marker[17]. This will be the "'s" postfix, and will be analysed as a case marker gen case followed by an implicit article "the". Thus, "X's car" will be analysed as "the car of X" (figure 3-17).

---

[17]The use here of right-recursive rules to analyse possessives rather than the simpler and more natural left-recursive analysis is only due to my use of a top-down backtrack parser, which will loop with left-recursive rules.

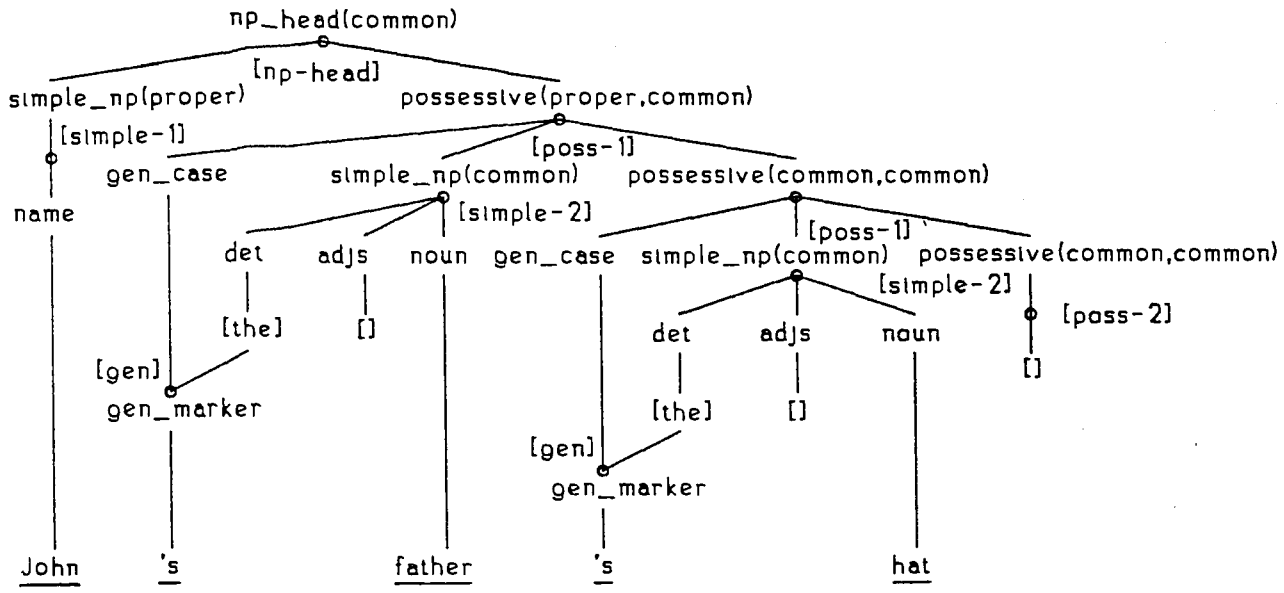


Figure 3-17: Possessive

A common noun phrase may also be modified by post modifiers np_mods, such as prepositional phrases (figure 3-18) and reduced relative clauses (figures 3-21 and 3-22). For reasons that are discussed in Section 4.2.4, I include in the reduced_relative category such post modifiers as participial phrases and comparatives (more on this below). The np_mods rules are fairly obvious:

```
np_mods --> np_mod, np_mods.                    [np-mods-1]
np_mods --> [].                                 [np-mods-2]

np_mod --> pp.                                  [np-mod-1]
np_mod --> reduced_relative.                    [np-mod-2]

pp --> prep, np(compl).                         [pp]
```

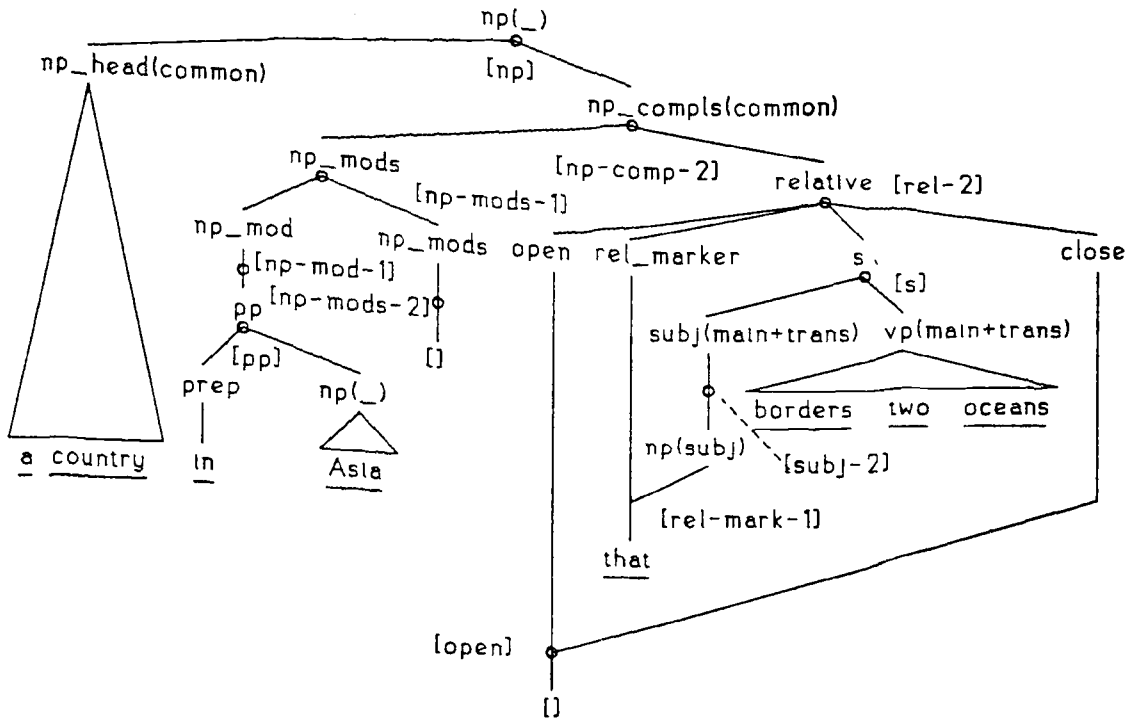


Figure 3-18: Noun phrase complements

I have now set the context to discuss the other important application of XG rules, the rules for relative and reduced relative clauses. I will start with relative clauses:

```
relative --> [].                                          [rel-1]
relative --> open, rel_marker, s, close.                  [rel-2]

rel_marker ... np(_) --> [that].                          [rel-mark-1]
rel_marker ... np(Case) --> wh(Case).                     [rel-mark-2]
rel_marker ... pp --> prep, wh(compl).                    [rel-mark-3]

wh(Case) --> rel_pron(Case).                              [wh-1]
wh(_) --> simple_np(common), prep, wh(compl).             [wh-2]
wh(_) --> whose, np(_).                                   [wh-3]
```

Most of these rules are similar to those for WH-question, so I will not comment on them further (See figure 3-18 for a simple example of relative clause). However, the second wh rule has no counterpart in WH-questions, because it describes the "pied piping" mechanism of English relative clauses [Stockwell et al. 73], which occurs in noun phrases such as

the concepts in terms of which the theory was formulated

an analysis of which is shown in figure 3-19.

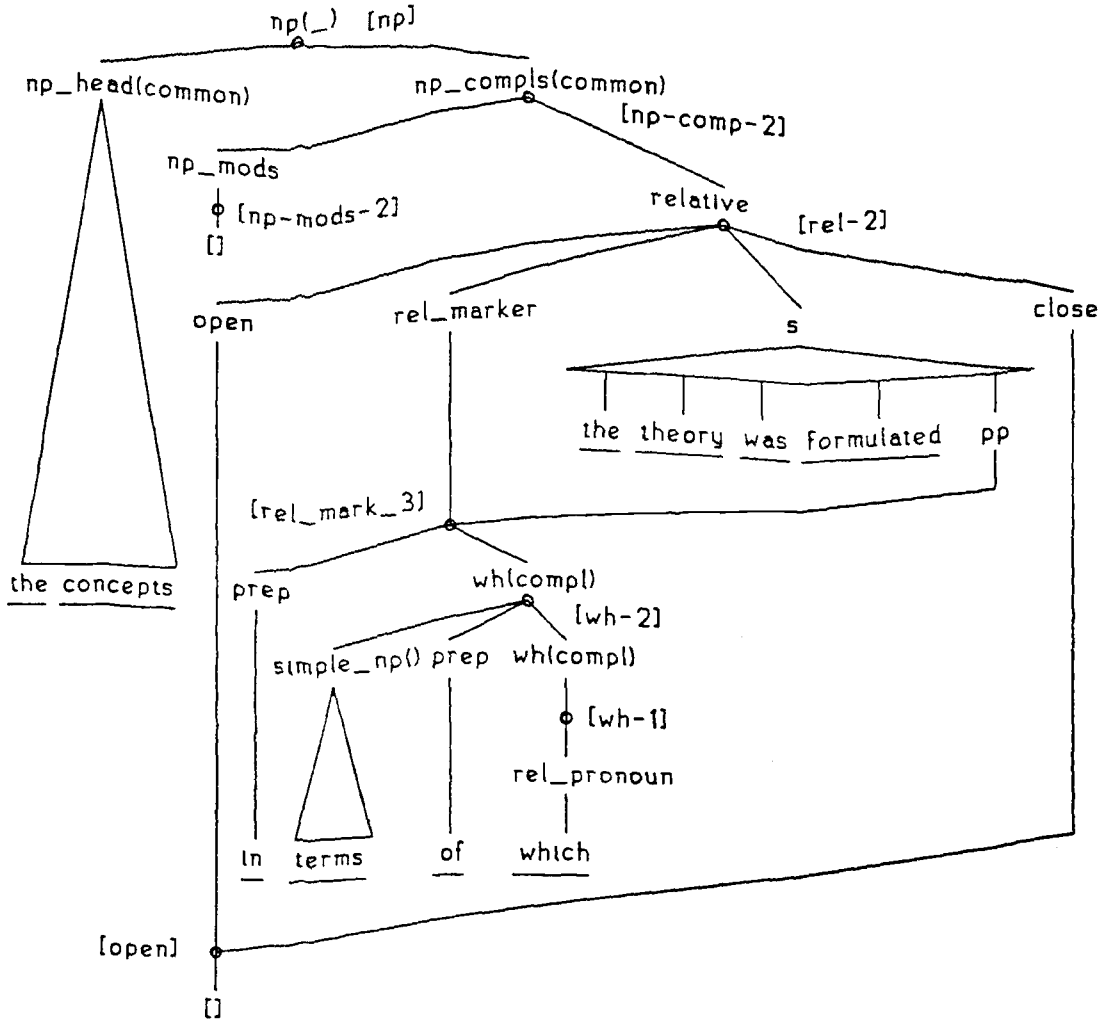


Figure 3-19: "Pied piping"

The third rule for wh describes how "whose" can appear inside a complex rel marker, in a way which is not possible in WH-questions (figure 3-20).

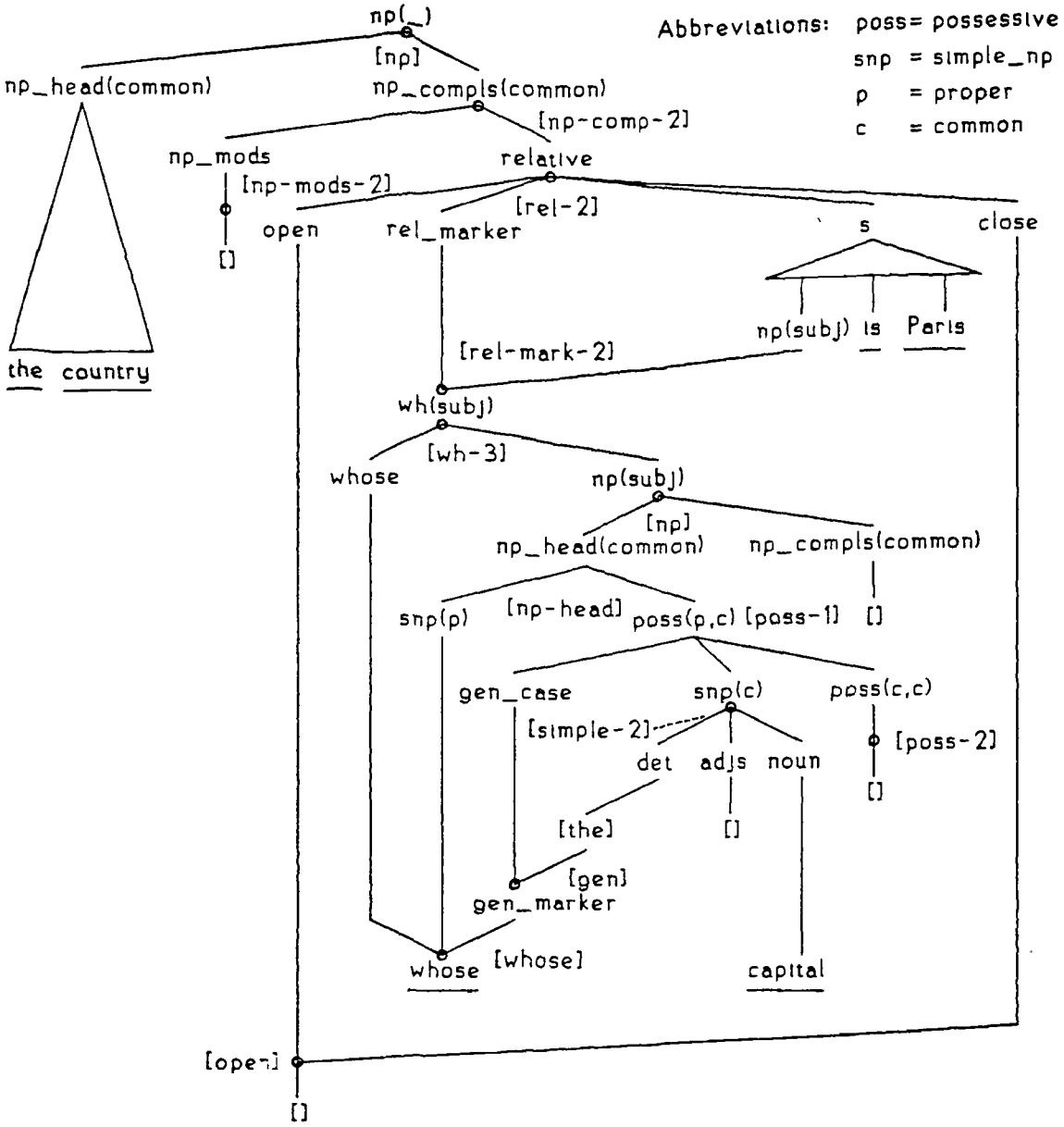


Figure 3-20: Relative "whose"

The rule for relative, as well as the rule below for reduced relative, uses the extraposition brackets open and close that

were discussed in Section (14). Let us recall the definition of the brackets here:

```
open ... close --> [].                                    [open]
```

I have mentioned before that all restrictive noun post modifiers are analysed here as reduced relative clauses. In this way, constituents, such as adverbs[18] and negation, which can occur in noun post-modifiers but which intuitively are sentential constituents, are seen as parts of some sentence, instead of parts of some other, tailor made, constituent. This conceptual economy is further supported by the rules of semantic interpretation discussed in Section 4.2.4.

However, the treatment of noun post-modifiers as reduced relative clauses is not as simple as the comments above suggest, because the corresponding rules have to be restricted in some way to apply only when what follows a noun is unequivocally a post modifier. This is achieved below by giving the rules right-hand sides to match an appropriate initial segment of the post modifier:

```
reduced_relative --> open, implicit_rel, s, close.          [red]

implicit_rel,                                               [impl-1]
   np(subj), verb_form(be), neg(Type,Neg), adj_phrase -->
   neg(Type,Neg),
   adj_phrase.
implicit_rel, np(subj), verb(Type) -->                      [impl-2]
   participle(Type).
implicit_rel, np(subj) ... np(compl) --> np(subj).          [impl-3]
```

The first implicit_rel rule analyses an optional negation neg

---

[18] In fact, the present discussion does not cover adverbs, but it is potentially easier to handle them with this formulation.

followed by an adjective phrase adj_phrase as a sentence with main verb "to be". The second implicit_rel rule analyses a participial phrase as a sentence whose main verb takes its features from the participle (figure 3-21). Note that in the full version of these two rules, there would be extra arguments and conditions to give tense and aspect features to the verb non-terminals that are introduced in the left-hand side of the rules. Finally, the third rule describes a conventional reduced relative clause, which is introduced by the subject noun phrase of the embedded sentence (figure 3-22).

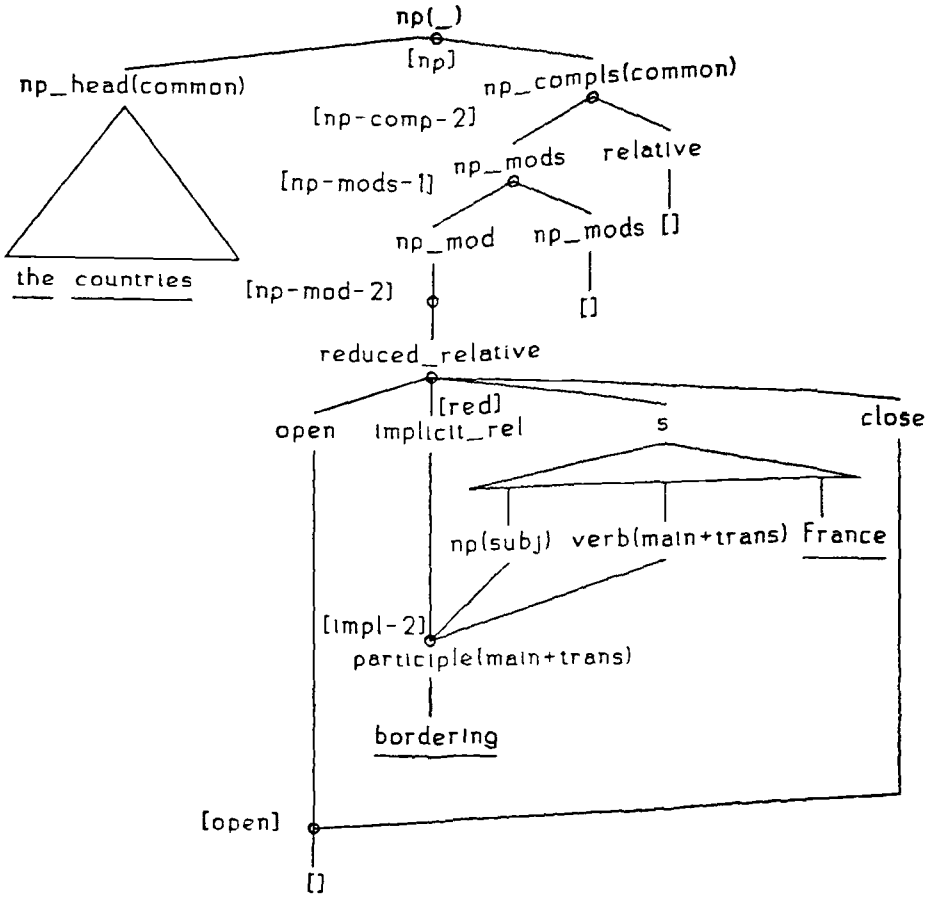


Figure 3-21: Participial complement

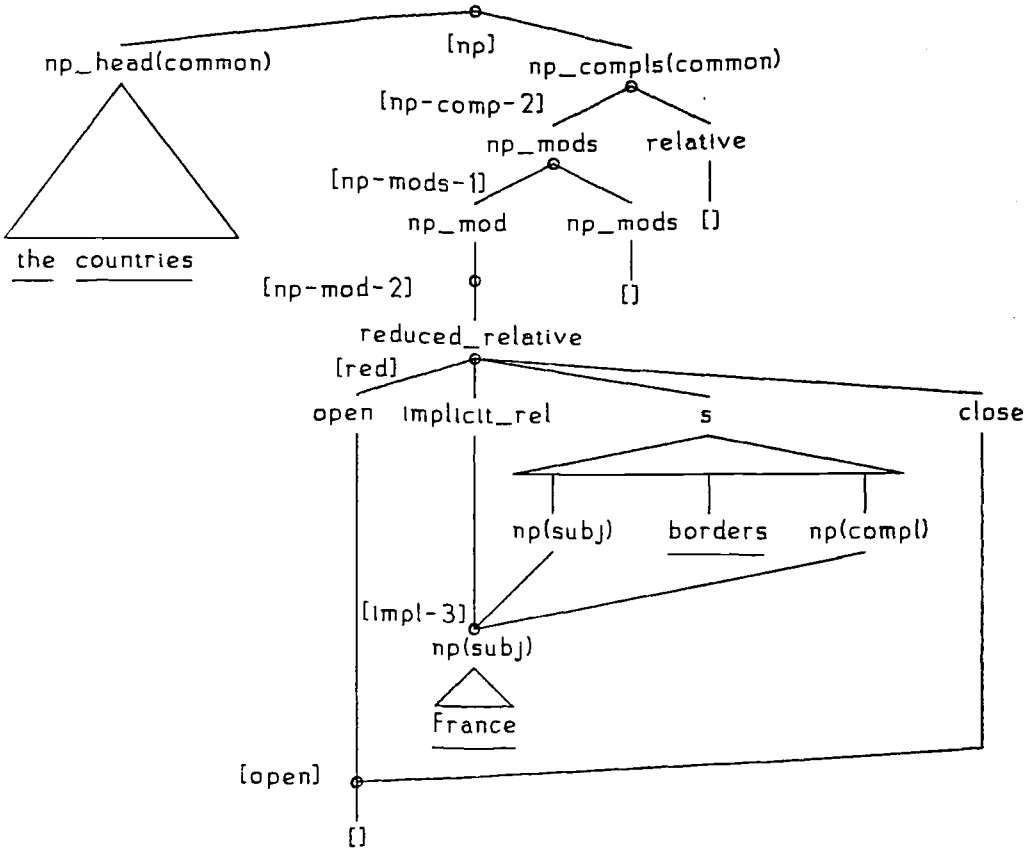


Figure 3-22: reduced relative

I will not go here into the details of adjective phrases and participial phrases, which are complicated and do not add much to the discussion. For these details, I refer the reader to the listing of the Chat-80 grammar in Appendix C. The same applies to the verb phrase rules, of which the following is a rather simplified version:

```
vp(Type) -->                                              [vp]
   verb(Type),
   verb_args(Type),
   vp_mods.
```

```
verb(Type) -->                                          [verb]
   verb_form(Root),
   {verb_type(Root,Type0)},
   neg(Type0,Neg),
   rest_verb(Type0,Type).

neg(_,pos) --> [].                                      [neg-1]
neg(aux+_,neg) --> [not].                               [neg-2]

rest_verb(Type,Type) --> [].                            [rest-verb-1]
rest_verb(aux+_,Type) -->                               [rest-verb-2]
   verb_form(Root),
   {verb_type(Root,Type0)},
   rest_verb(Type0,Type).

verb_args(_+trans) --> np(compl).                       [verb-args-1]
verb_args(_+be) --> adj_phrase.                         [verb-args-2]
verb_args(_+be) --> np(compl).                          [verb-args-3]
verb_args(_+Type) --> {no_args(Type)}.                  [verb-args-4]

no_args(intrans).                                       [no-args-1]
no_args(trans).                                         [no-args-2]

vp_mods --> vp_mod, vp_mods.                            [vp-mods-1]
vp_mods --> [].                                         [vp-mods-2]

vp_mod --> pp.                                          [vp-mod-1]
vp_mod --> adverb.                                      [vp-mod-2]
```

A verb phrase *vp* is a verb *verb* followed by arguments *verb_args* and by optional modifiers *vp_mods* (rule [vp]). The rule for *verb* picks a verb form, the optional negation *neg* following it, and the rest of the verb *rest_verb*, which may be empty or a string of auxiliaries followed by the main verb. The **aux** feature in the verb feature pair is used to determine if a verb form can precede another as an auxiliary. It is also used in the first argument of *neg* to restrict negation to following an auxiliary verb. In the Chat-80 grammar, these rules are complicated further to constrain the tenses of the verb forms in the verb, and deal with passive verbs.

The **args** feature of the verb feature pair is used in these rules to distinguish between verbs which take different kinds of arguments.

The no_args condition defines what verb_types have optional arguments. One should note that the rules leave all logical verb arguments marked by a preposition, even a passive subject, to be picked by the vp_mods rule. This is so because the grammar rules at this level have not enough information to make a good decision about what prepositional phrases are verb arguments. That decision must be taken at a different level, discussed in Section 4.2. Note also that the Chat-80 grammar makes no attempt at covering the sentential arguments required by verbs such as "to know" and "to tell". To show how some of the verb rules are used, I give in figures 3-23 and 3-24 full versions of the abbreviated parse graphs of figures 3-15 and 3-13.

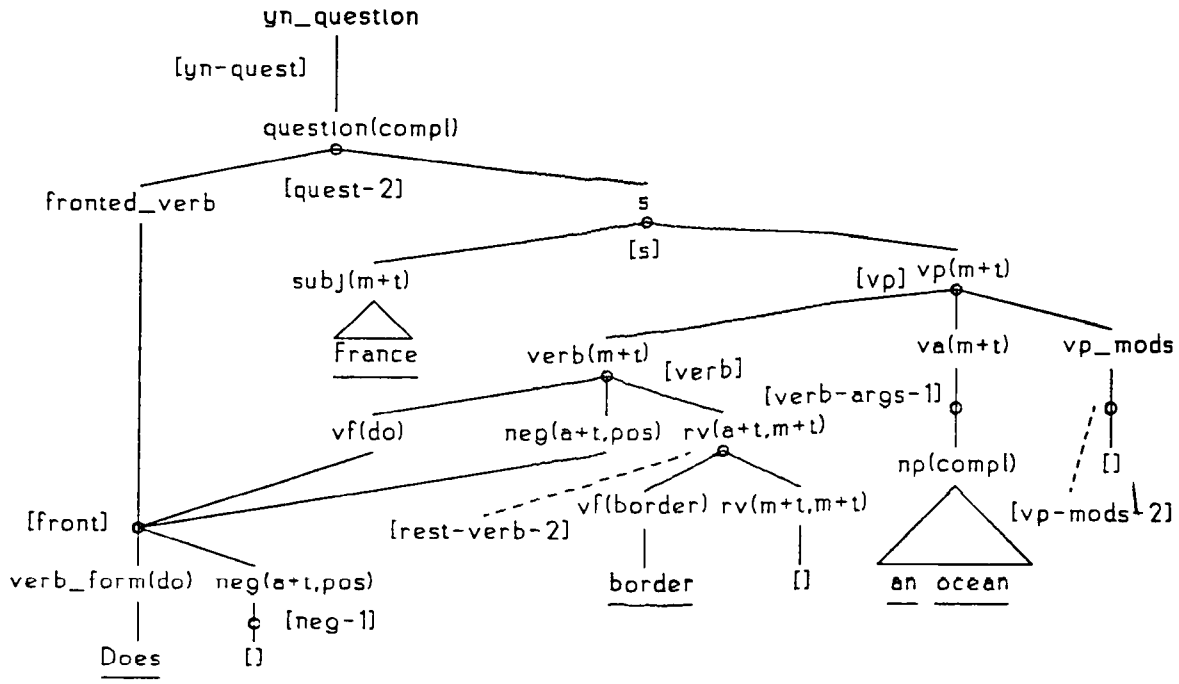


Figure 3-23: Details of a yes-no question

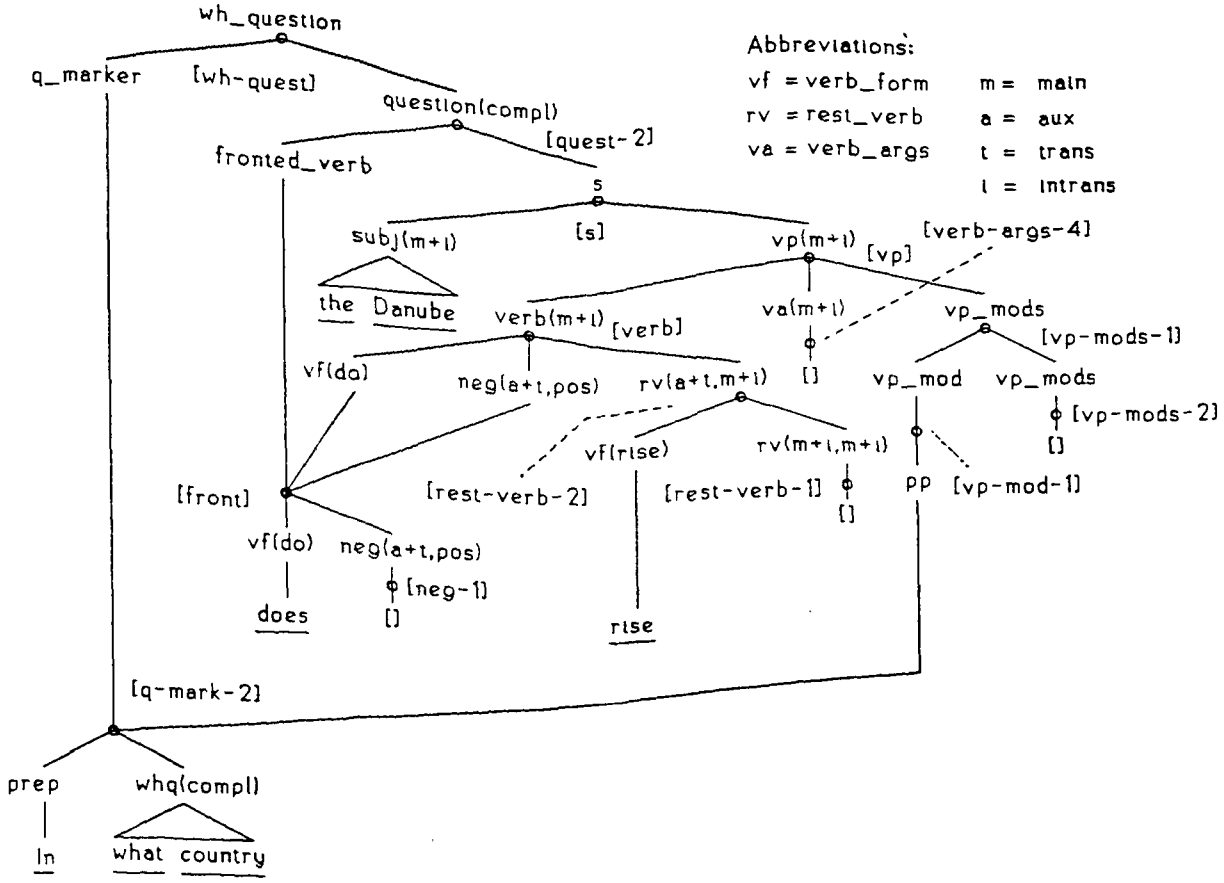


Figure 3-24: Details of a prepositional phrase question

3.10 Attachment Ambiguities

In this section, most examples are based on the grammar of the last section, which for that reason will be referred to as "the grammar".

If we look at the rules for modifier attachment in the grammar, (rules [np-mods-1], [np-mods-2], [vp-mods-1] [vp-mods-2]), it is clear that in a nested construction it is possible to attach a post-modifier at many different places in a parse tree for the

construction. This is the well known problem of attachment ambiguity that seems to need more than syntactic constraints for its solution. In Section 4.2, I will discuss means of finding reasonable attachments on semantic grounds. I will argue that there is no point in having the syntactic level describe all the possible attachments, because syntax has no way of employing that proliferation of analyses for a useful analytic purpose. From this point of view, the fact that the grammar can generate a large set of attachment permutations is only a formal accident, which adds nothing to its descriptive power. On the contrary, by swamping genuine, semantically significant ambiguities, the attachment ambiguities hinder the use of the grammar both as a descriptive tool and as a practical input analyser for programs such as Chat-80. This problem has plagued writers of formal grammars [Woods 73, Woods 77b], and has been used to refute the usefulness of such grammars [Mellish 81].

I take a different view of attachment ambiguities. If they do not add anything to the descriptive power of a grammar, the grammar should not have them in the first place. Because the elimination of ambiguities does not change the language weakly defined by a grammar, those who see a grammar purely as a recognition device will have no objection to that elimination. On the other hand, if the purpose of the grammar is to describe the phrase structure of the input, the elimination of attachment ambiguities has the only effect of producing a single analysis from which it is a simple matter to derive the other attachments. That is, the single analysis produced is just a representation of an ordered set of post-modifiers uncommitted with respect to their attachment, a **normal form** analysis.

Starting from a grammar with attachment ambiguities, we need first to define what is a normal form, and then add extra arguments and conditions to prevent the generation of non-normal analyses. Although this may seem obvious, one should bear in mind that we are dealing with grammars, which are descriptive devices, and not with parsers, which are algorithms. In a parser, we may postulate that analyses produced in some particular situation, for instance the first analysis produced, are the normal form analyses. With grammars, however, we have no such liberties: we have to state explicitly what normal form means.

In the Chat-80 grammar, I use right-most normal form (RMNF for short), simply defined as that analysis, of a set of analyses differing only on modifier attachments, where each modifier is attached to the smallest constituent it may modify in the original, ambiguous, grammar. In other words, the analysis tree is as "deep" as possible by the original rules. RMNF is closely related to the Right Attachment principle of Kimball [Kimball 73, Wanner 80, Fodor and Frazier 80].

Although RMNF is easy to visualise and very convenient for the semantic and pragmatic levels, it is formulated as a global condition involving the comparison of all possible analyses. Clearly, it is not possible to introduce such a condition as additional arguments and tests in an XG. In fact, given an arbitrary XG, it is not possible to produce a set of definite clauses which directly represents such a global constraint. Thus, the above formulation must be translated into local conditions describable by extra arguments and tests. We will now see how to add those extra arguments and tests.

I start by restating the attachment ambiguity problem in terms of partially constructed analyses. For ease of exposition, I assume that analyses are built from left to right[19]. A partial derivation graph is obtained from a derivation graph by taking some path from the root and, for each node in the path, omitting all outgoing edges to the right of the path, and also possibly some other nodes and edges to the right of the path. The selected path is the cliff of the partial graph. The terminal string spanned by the sub-graph to the left of the cliff is the attached part of the given input string, separated from the the rest of the input string by a notional cliff base. Figure 3-25 shows a partial derivation graph for the grammar taking s as start symbol, with cliff from s to the word "man" and attached string "John saw a man".

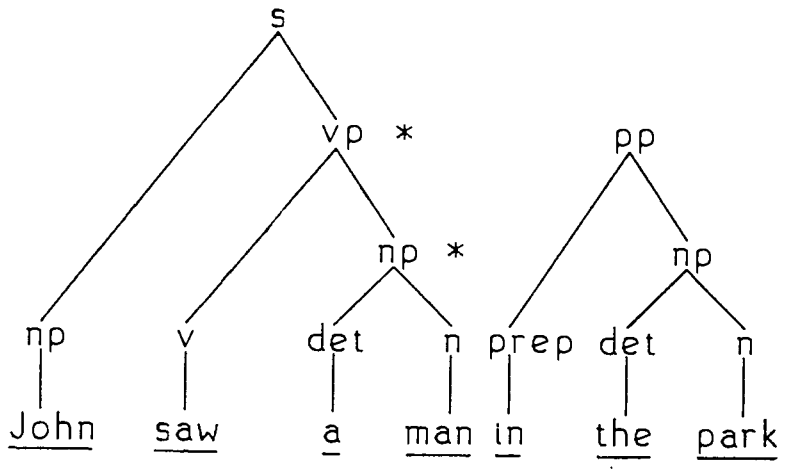


Figure 3-25: Partial derivation graph

[19]The methods discussed, however, do not depend in any way on this assumption.

Given a partial graph and some constituent immediately to the right of the cliff base, an attachment ambiguity is present if the graph can be extended in more than one way by connecting a node on the cliff by some new edges and nodes to the constituent. In other words, there are several alternative niches in the cliff for the constituent. RMNF is achieved by selecting the niche furthest away from the root to extend the graph.

In general, not all niches in a cliff are suitable for a given modifier. Each niche has an attachment set of phrase patterns that may be attached to it. An element of an attachment set will be in most cases a phrase category (a non-terminal), but not always, as I will show later. I will assume that there is a definite set of phrase patterns relevant for attachment, and represent each by a token or kind.

Now, only RMNF analyses will be generated if a grammar will not attach a phrase to a niche unless there is no lower niche which can receive the phrase. In a first approximation, that condition will be satisfied if:

* a phrase is attached to a node if its kind is in the attachment set of the node;
* the kinds in the attachment sets of lower niches are excluded from the attachment sets of higher niches.

To see how these rules might be used in a grammar, consider again some of the rules for noun phrase post-modifiers:

```
np_mods --> np_mod, np_mods.                    [np-mods-1]
np_mods --> [].                                 [np-mods-2]
```

```
np_mod --> pp.                                    [np-mod-1]
np_mod --> reduced_relative.                      [np-mod-2]
```

Two types of arguments are added to non-terminals: set arguments, which describe the attachment sets of niches, and mask arguments, which describe the kinds of phrase that may be attached to a non-terminal, and which therefore should not be attached to non-terminals higher up in the analysis. The current attachment set of a niche is passed to potential modifiers, where it determines whether the modifier can be attached to the niche.

I will now augment the above rules with arguments and tests as described. To make the modified rules easier to read, the set operations are represented functionally. The operator '+' denotes set union, the operator '-' set difference, and the predicate *in* checks whether its first argument is in the set denoted by the second argument.

```
np_mods(Set0,_,Mask) -->                          [rmnf-mods-1]
   np_mod(Set0,Mask0),
   np_mods(Set0-Mask0,Mask0+Set0,Mask).
np_mods(_,Mask,Mask) --> [].                      [rmnf-mods-2]

np_mod(Set,Mask) -->                               [rmnf-mod-1]
   {in(pp,Set)},
   pp(Mask).
np_mod(Set,Mask) -->                               [rmnf-mod-2]
   {in(rel,Set)},
   reduced_relative(Mask).
```

The value of the first argument of an *np mods* non-terminal is the attachment set for the niches within the *np mods*, that is, it the set of the kinds of modifier that can be attached inside this particular *np mods* as part of an RMNF analysis; the second argument of an *np mods* represents the mask from the right-most *np mod* in the

subgraph to the left of the np mods; and the third argument represents the mask for the whole np mods node, that is the set of kinds of modifiers for which there are open niches inside the subgraph dominated by the np mods node.

The two np mod rules [rmnf-mod-1] and [rmnf-mod-2] are easiest to explain. The first states that an np mod may be a pp, provided that the kind pp is in the attachment set for the node; the mask for the node is that of the pp. The second is the analogous statement for a reduced relative, with the kind pp changed into the kind rel of relative clauses.

The main np mods rule [rmnf-mods-1] takes the attachment set for the whole node Set0 and passes it as the set of the node's leftmost descendant np mod, which returns a mask Mask0. The np mods in the body of [rmnf-mods-1] represents the (possibly open) niche at this level in the analysis: its attachment set is the difference Set0-Mask0 between the set for the whole node and the set Mask0 of those kinds that can be attached within the lower np mod node. This niche may be left empty, and then the second rule for np mods [rmnf-mods-2] is used to expand the np mods. In this case, the higher np mods is complete, and its mask is the set Mask0+(Set0-Mask0) = Mask0+Set0 of those kinds that can be attached either in the lower np mod node or in the niche for the node, the right-hand side np mods in rule [rmnf-mods-1].

We can see how the augmented np mods is used in the following simplified version of the noun phrase rules:

```
np({}) --> name.
np(Mask) --> np_head, np_mods({pp,rel},{pp,rel},Mask).
```

Sets are represented here in the usual curly bracket notation, which together with the operators for set union and difference will be interpreted by the predicate in. We see that a proper noun cannot have any modifiers (of the kinds being considered here), and so returns the empty mask. On the other hand, a common noun phrase returns as its mask the union Set0+Mask0 of the mask Mask0 of its last modifier with the attachment set Set0 of its right-most non-empty descendant meaning that any phrase that may be attached as a modifier of that noun phrase or of one of its modifiers cannot be attached higher up. The use of the augmented rules is shown in figure 3-26.

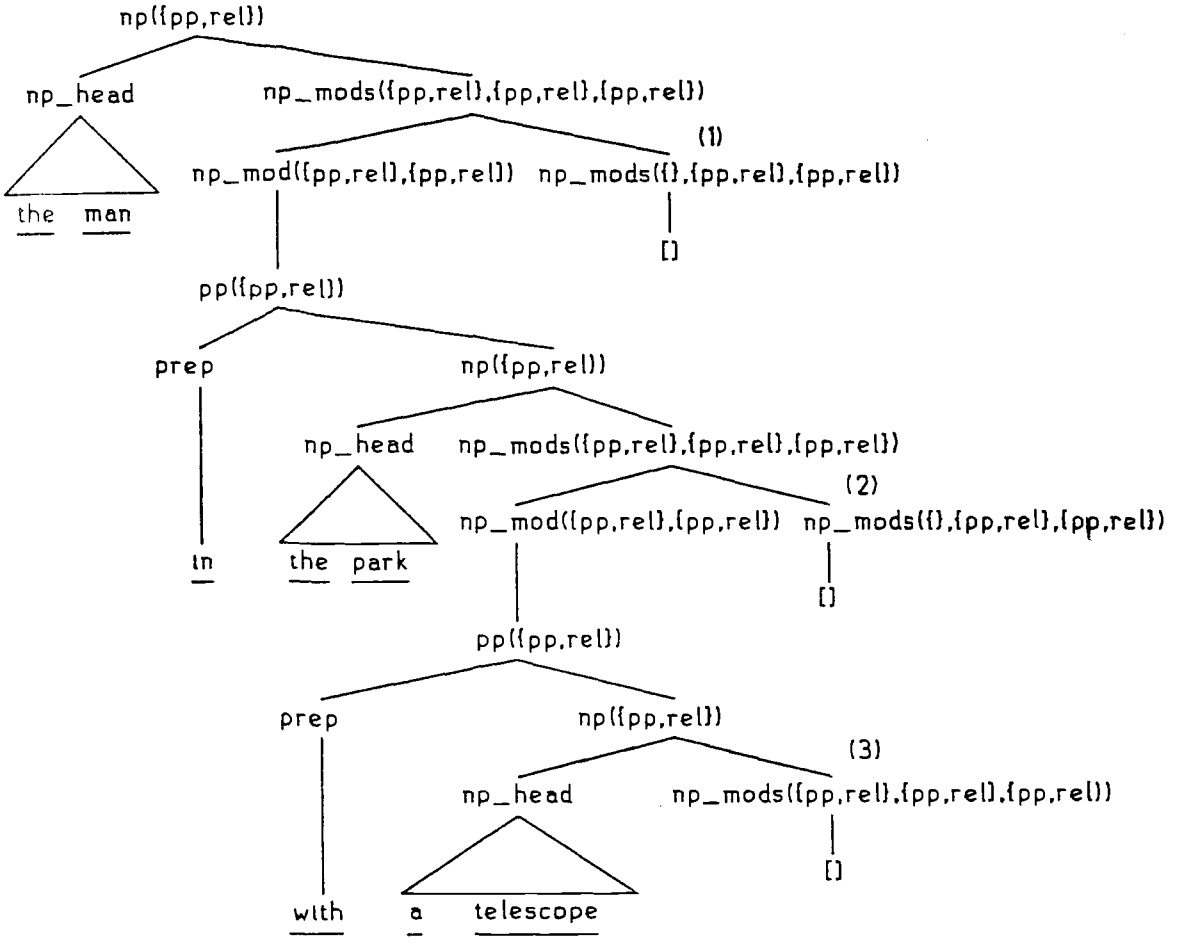


Figure 3-26: RMNF attachment

Notice how the attachment sets for the higher niches (1) and (2) are empty, and therefore the only niche left open is the right-most one (3).

As I have said before, the phrase kinds used for attachment control may have to reflect more than just the syntactic category of the phrase. For example, the attachment properties of traces are different from those of non-extraposed phrases. On the one hand, a

trace cannot have any modifiers. On the other, constraints such as the complex-NP constraint restrict the niches which may receive a trace. To cope with these problems in the example grammar, we can introduce a new modifier kind, *trace*, which will be used to control the attachment of traces. However, the grammar does not distinguish in any way between a a noun phrase trace, say, and a "real" noun phrase. In fact, it might be argued that this is one of the main reasons for its conciseness.

In the current Chat-80 grammar, the problem of distinguishing alternative kinds of the same non-terminal is solved by adding a sub-categorisation argument to non-terminals, and representing kinds, sets and masks in such a way that the *in* test can be reduced to the unification (in the theorem proving sense [Robinson 65]) of its two arguments. With such a set representation, set union and difference must be defined explicitly by predicates, and set constants are also better introduced indirectly by predicates. The sub-categorisation argument of a non-terminal is then unified against the representation of the set whose single element is the non-terminal's sub-categorisation. In the case of traces, the sub-categorisation argument is filled with the appropriate kind representation in the actual XG rule that repositions the trace, so that the kind is checked against the current attachment set of a node when the trace is tried as a modifier of that node. I will now apply this technique to some of the example rules. There will be several auxiliary predicates:

* to represent kinds, *trace*, *obj* ("real" noun or prepositional phrase) and *rel*;

* for the set operations, union and diff;

* to represent specific sets, empty (the empty set), np_all (the initial attachment set of a common noun phrase) and a_trace (the singleton set with trace);

The new rules, which would replace the rules with the same labels in the preceding section, are as follows:

```
np(_,Set,Mask) -->                                              [np]
   {obj(Set)}, np_head(Type),
   {np_all(All)}, np_compls(Type,All,Mask).

np_compls(proper,_,Empty) --> {empty(Empty)}.                   [np-comp-1]
np_compls(common,Set0,Mask) -->
   {np_all(All)}, np_mods(Set0,Set,All,Mask0),                  [np-comp-2]
   relative(Set,Mask0,Mask).

np_mods(Set0,Set,_,Mask) -->                                    [np-mods-1]
   np_mod(Set0,Mask0),
   {minus(Set0,Mask0,Set1), plus(Mask0,Set0,Mask1)},
   np_mods(Set1,Set,Mask1,Mask).
np_mods(Set,Set,Mask,Mask) --> [].                              [np-mods-2]

relative(_,Mask,Mask) --> [].                                       [rel-1]
relative(Set,_,Mask)  -->                                           [rel-2]
   {rel(Set)},
   open, rel_marker, s(Mask0), close,
   {a_trace(Trace), minus(Mask0,Trace,Mask)}.

rel_marker ... np(Case,Item,Empty) -->                          [rel-mark-2]
   wh(Case),
   {trace(Item), empty(Empty)}.
rel_marker ... pp(Item,Empty) -->                               [rel-mark-3]
   prep, wh(compl),
   {trace(Item), empty(Empty)}.

np_mod(Set,Mask) --> pp(Set,Mask).                                  [np-mod-1]
np_mod(Set,Mask) -->                                                [np-mod-2]
   {rel(Set)},
   reduced_relative(Mask).

pp(Set,Mask) --> prep, np(compl,Set,Mask).
```

The sub-categorisation argument is the second argument of np. Because the relative clause rule [rel-2] needs an attachment set argument, a suitable value must be returned by np_mods. This new argument of np_mods is the second one, between the arguments for the attachment set and for the mask of nodes to the left, which were described earlier when discussing rule [rmnf-mods-1].

In rule [np-rel-2], the kind *trace* is excluded from the mask returned by a relative clause. This is so because the *open-close* bracket prevents any trace from being repositioned into the relative clause. A trace-accepting niche in the relative clause cannot receive traces from outside the relative clause and therefore such traces must be the responsibility of niches above the relative clause and should not be masked by it.

The graph of Figure 3-27 shows the use of the augmented rules. The analysis is the same as the one in Figure 3-18 except for the attachment arguments. For readability, sets and masks are represented in the figure in the usual set notation.

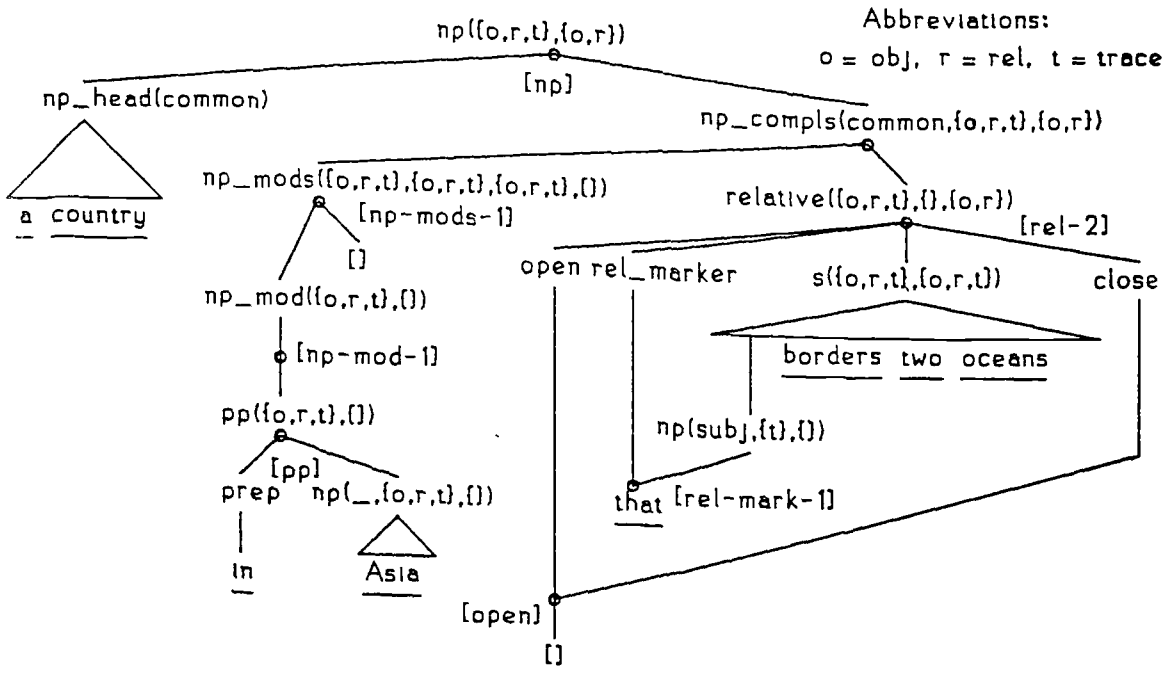


Figure 3-27: RMNF attachment

The same attachment constraints could of course be represented in

the more concise way presented earlier, but then traces and the expansion of various phrase types into traces would have to be made explicit.

From the point of view of parsing, the technique discussed here reduces substantially the non-determinacy of a parser derived from a grammar which produces only normal form analyses. A normal form resembles a "closure strategy" for parsing, as discussed by Church [Church 80]. RMNF is related in this way to what he describes as "late closure". Notice that the attachment sets and masks are simply subsets of a pre-defined finite set, and thus may be seen as packs of features. No structure is being built or destroyed.

Finally, nothing forbids further sub-categorisations of phrase kinds, based on semantic or pragmatic considerations. Although I have not chosen this path, for reasons detailed in the next chapter, the sets and masks technique is attractive for those more complex attachment strategies because it gives a purely declarative description of global constraints whose implementation is more usually seen in terms of process control. That is, constraints on the shape of analysis graphs have been decoupled from the algorithms for generating analyses.

3.11 Summary

Extraposition grammars are a grammar formalism based on definite clauses that can represent concisely the phenomenon of left extraposition. In particular, grammatical notions such as the complex-NP constraint and "islands" can be expressed in the

formalism. An extraposition grammar is no more than a convenient notation for a definite clause program, and as such it can be run as a parser by a definite clause interpreter such as Prolog.

To show how extraposition grammars can be used in practice, I have given a simplified version of the grammar used in the program Chat-80. The Chat-80 grammar uses arguments to non-terminals and tests to limit the problem of syntactic ambiguity in the attachment of post-modifiers to phrases. The arguments and tests only allow analyses in right-most normal form, in which all post-modifiers are attached to the right-most node they can modify. Incorporating these constraints in the grammar, instead of in the parsing algorithm, means that the same reduction in the number of analyses will occur whichever parsing algorithm is used.

Chapter 4

Interpreting Sentences

4.1 Motivation

Montague, in his article "English as a Formal Language" [Montague 70], rejected "the contention that an important theoretical difference exists between formal and natural languages". From this point of view, the purpose of a formal syntax for a natural language is to provide a basis for connecting the meanings of the components of a sentence, much in the same way as the syntax of a logical system provides the skeleton for the composition of meanings of formulas. Syntax would thus lose an independent motivation, as each syntactic rule would be just an image of a rule of semantic composition.

If syntax is to be solely governed by how the meanings of words fit together, then by a suitable choice of meanings it might even be possible to reduce grammar to a few trivial rules such as "the meaning of the concatenation of two sequences of words X and Y is the meaning of X applied to the meaning of Y." This is roughly the sole "grammar" rule for combinatory logic [Curry and Feys 68]. Several theories of grammar, such as "categorial grammar" [Lewis 72, Creswell 73] and "applicational grammar" [Shaumyan 77], try to carry out this programme for natural languages. However, it soon becomes clear that, even when the semantics of a natural language phrase are intuitively

obvious, a certain amount of syntactic manipulation is needed to get from the phrase to the semantics. This happens, for example, with relative clauses, where an abstracted variable has to be introduced by syntactic means (as described in the last chapter) into the translation of the clause. Certainly, the syntax rules of Montague [Montague 73] and his followers have become less and less trivial. Many syntax rules are associated to the "identity" semantic operation, implying that such rules are required purely for syntactic reasons. The practical problem is clearly expressed by Woods [Woods 77a]:

> This type of specification is clean and straightforward and works well for artificial languages that can be defined by context-free or almost context-free grammars. For interpreting natural language sentences, whose structure is less isomorphic to the kind of logical meaning that one would like to derive, it is less convenient, although not impossible.

It seems fair to say that, in the present state of knowledge, the more comprehensive a grammar becomes, the more rules with a trivial semantic part it contains. For instance, Montague's grammar fragment in "The Proper Treatment of Quantification in Ordinary English" [Montague 73] has 7 syntactic rules paired with the same translation rule of functional application

$$F(a,b) = a'(\hat{}b')$$

but Bennett's extended fragment [Bennett 76] has 18 syntactic rules associated to functional application. In Thomason's and Rodman's fragments [Thomason 76, Rodman 76], the syntactic operations associated to a given semantic rule become far more complicated than in Montague's fragment.

If a grammar is to be used as the input component of some application, the need to cope with "pragmatic" matters complicates further the relationship between syntax description and semantic composition rules. Not all possible attachments of modifiers, and not all possible nestings of quantifications, are equally plausible. In particular, clues for reasonable choices in this area may come from the order of phrases in the input, which of course would be forgotten in a Montague-type grammar. This is one of the major defects of the grammars by Colmerauer and Dahl [Dahl 77, Colmerauer 79a], where the scope of operators (using Seuren's terminology [Seuren 69]) is defined by the phrase structure of the input alone, as semantic composition rules are inextricably bound to syntactic rules.

If the syntactic part of Montague-type grammars is so complex that syntactic rules can no longer be understood by simply looking at the corresponding semantic rule, we are justified in suspecting that, after all, a distinct syntactic level, with its own concepts, is needed to describe the language. Furthermore, if a distinct pragmatic level is also involved, we will need some intermediate representation to be the object of this pragmatic level.

Against the foregoing argument, it might be argued that the need to separate the syntactic component from the semantic and pragmatic components arises in great part from my choice of semantic representation language. Indeed, being a variation of first-order logic, DCW clauses lack the higher order constructions that are used in Montague grammar to compose together the meanings ot the parts of a sentence. For example, the simple syntactic rule

```
s --> np, vp.
```

could be paired with the Montague-style semantic rule

```
s' = np'(vp')
```

where x' denotes the translation of x. In contrast, an analogous rule to produce directly a first-order formula would be [Pereira and Warren 80]

```
s(S) --> np(X,VP,S), vp(X,VP).
```

where S is the translation of the sentence, VP the translation of the verb phrase and X the variable bound by the quantification introduced by the noun phrase. In a larger grammar, the rule arguments required to build a translation in this way become very complex [Dahl 77], and thus less practical than separate syntactic and semantic components.

The argument as put here is so far one of formal style. It could however become more than that if we could find situations where a translation into first-order logic became radically more complicated, or even impossible, as compared with a translation into a more powerful logical language. In examining this question, three points should be taken into account.

First, the separate practical requirement of this work, of a semantic formalism with reasonable computational properties, has ruled out trying to deal with semantic phenomena that could only be accounted for in formalisms for which suitable proof procedures are not available, such as modal or intensional logic.

The second point, which follows from the first, is that in the

absence of intensional or higher-order quantification, the use of higher-order variables and lambda abstraction in a formalism is purely a syntactic convenience, as any closed sentence with these constructions can be simplified down to a first-order sentence [Montague 73, Pereira 78]. As the example above shows, any such "syntactically convenient" use of higher order constructions can be easily replaced by the use of extra arguments in grammar rules, with a modest loss in perspicuity.

Finally, the difficulties in translating certain words, together with the pragmatic questions of attachment and quantifier scope, seem to require non-compositional analysis mechanisms, which could hardly fit into the compositional framework underpining the pairing of syntactical rules and composition of higher order predicates. An example of the difficult words I have in mind is the verb "to have". As discussed in detail in Section 4.2.6 below, interpretations for the verb "to have" depend crucially on syntactic and semantic properties of the verb's arguments, a situation that seems to be difficult to describe compositionally. That is, a translation of the verb "to have" cannot operate transparently over its arguments. Another example is the role of prepositions, which in certain contexts behave as argument markers whereas in others they seem to behave as verbs (Section 4.2.2). In not seeming to cope with these difficulties, higher order semantic representation formalisms lose one of the advantages over first-order logic that they might have in the present context.

To find a logical form for a sentence, we need to know how the

meanings of words are bound together into the nucleus [Seuren 69] of the sentence. In most cases, for the semantic theory used in this work, the nucleus is determined by what the arguments are of the predicates which translate "content" words. In grammatical terms, finding the arguments of predicates requires us to find what modifies what, that is, to find where to attach verb complements, prepositional phrases, relative clauses and adjective phrases. This is the subject of the next section.

To complete the logical form for a sentence, we also need to know the relative scopes of operators, such as articles, negation and conjunctions. Certain operators will be seen as governing other operators, whenever they appear together in a specific relationship in a parse tree. This will be the subject of Section 4.3.

4.2 What Modifies What?

It is clear that constituent structure cannot be determined on syntactic grounds alone. This has been amply demonstrated [Woods 73] by examples such as

I saw a man in the park with a telescope (4.1)

I have re-stated in the last section that a distinct syntactic level is required to describe even rather trivial natural language constructions, if we are to derive a logical form for those constructions. However, it is not possible for that syntactic level alone to decide, say, that "with a telescope" in (4.1) is a sentence modifier rather than a modifier of the noun phrase with head "man" (or even of that with head "park"). We are left in a situation where the syntactic level must produce some analysis of the input, but it

has not enough information to do so. The following are possible ways of attacking this problem:

* 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.

The first of these methods has been used in large scale grammars [Woods et al. 72], but it is rather inefficient to implement in a program [Woods 73], because many parse trees are generated which are then rejected by other levels of analysis.

The second method is conceptually the most satisfying because it has to be based on a deeper integration of the syntax, semantics and pragmatics of modification. The grammars of Colmerauer and Dahl use this approach, although their pragmatic notions are too simple even for the limited subsets of language they deal with. McCord [McCord 80a] has devised a much more comprehensive theory of modification, where every node of a syntax tree specifies how it modifies its parent.

I have adopted the third method, for two main reasons. First, because syntactic analysis is decoupled from modifier attachment, syntax rules (XG rules) and attachment rules (definite clauses) can be run by Prolog without the attachment rules causing backtracking in the syntax rules. Secondly, by having a separate attachment level, it is easier to cope with constructions which require <u>syntactic</u> manipulations to derive an interpretation, such as sentences with

main verb "to have". This question will be discussed in sections 4.2.2 and 4.2.6.

4.2.1 Representing Attachments

The modifiers that concern most of the attachment rules fall into two classes: fillers, whose role is to fill an argument place of the predicate corresponding to a noun, verb or preposition; and restrictions, which specify properties of the modified entity. These notions will be discussed in more detail in the sections that follow. For the moment, given that the target formalism is first-order logic, it is enough to note that the objects denoted by fillers and constrained by restrictions are represented by quantified variables. Therefore, modifier attachment requires the identification of argument places of predicates with quantified variables. Thus, the representation of attachments cannot be just a reshaping of the initial parse tree, but has to contain information about these identifications of variables. The structures I use are trees with three kinds of nodes:

* quantification nodes (Quants for short), that describe a phrase whose translation will introduce a new variable in the target logical form;
* predication nodes (Preds for short), that describe a phrase whose translation will relate together variables introduced by Quants;
* conjunction nodes (Conjs) that describe a set of Preds for phrases joined together in a conjoined phrase.

The most complex nodes are Quants, which in general are produced when translating a noun phrase. The fields of a Quant are:

* the determiner, in most cases corresponding to an English

determiner, which will translate into the quantifier binding the variable introduced by this node;

* the head, which is either the predication translating the head noun of a noun phrase, or a term denoting a higher-order function;

* the predication, a tree for restrictions modifying this node whose operators have necessarily narrower scope than the determiner on this node;

* the modifiers, trees for all the other modifiers attached to this node;

* the bound variable introduced by this node;

* the range variable, the variable restricted by the head, restriction and modifiers, which defines the individual entities defined by this Quant - this might be distinct from the bound variable, for those determiners, such as plural determiners, which make the bound variable range over sets.

The distinction between the predication and the modifier list of a Quant has to do with scoping heuristics for restrictive modifiers, and will be discussed further in Section 4.3.

Recalling the interpretations for determiners given in Section 2.3, we can see a determiner as a higher-order predicate taking as arguments a range predication and a scope predication. If we ignore the effects of scope rules, the range corresponds to the noun phrase that contains the determiner, and the scope to the rest of the sentence. The interpretation of a Quant is then:

```
determiner'(lambda(range variable).
                head' & predication' & modifiers',
            lambda(bound variable).scope' )
```

where x' is the translation of x. Taking for example the sentence

John visited every country in Asia that borders an ocean

and leaving out irrelevant detail, the Quant for the direct object of the main verb will have the following fields:

```
determiner = every
head =       country(R)
predication ="R borders an ocean"
modifiers =  in(R,asia)
```

with R the node's range variable. The interpretation of the Quant will then be:

```
every(lambda(R).(country(R) & "R borders an ocean" & in(R,asia)),
      lambda(S).( "John visited S"))
```

where of course the quoted fragments would have been translated in a similar way.

This view of determiners as higher-order predicates is of course in the present context only for convenience, as the determiner will be ultimately translated into a DCW quantifier as described in Section 2.3.

Below, Quants will be represented by logic terms:

```
quant(det,range,head,pred,mods,bound)
```

where det is the determiner, range the range variable, head the head, pred the predication, mods the modifiers and bound the bound variable. For example, the Quant for the example above would be:

```
quant(every, R, country(R), "R borders an ocean", [in(R,asia)],S)
```

The other tree node types, Preds and Conjs will be discussed in Section 4.2.4.

4.2.2 Arguments

The notion of argument that I use is close to the notion of predicate arguments in logic. The subject and objects of a verb are arguments of the verb; the noun phrase Y in "the X of Y" is an argument of the head noun of X. In general, any prepositional phrase modifying a noun will be taken as an argument of the noun if the prepositional phrase cannot be paraphrased by a relative clause [Stockwell et al. 73]. For example, contrast the pair of noun phrases

the cabin in the forest
the cabin which is in the forest

with the pair

the destruction of the city
* the destruction which is of the city

In other words, the preposition of an argument prepositional phrase is just a place ("case") marker without independent meaning, whereas the preposition of a non-argument prepositional phrase is akin to a verb whose subject is given by the noun which the prepositional phrase modifies.

This distinction between place marker prepositions and content prepositions might be used to separate verb arguments from verb and sentence modifiers. In this case, one tries to see whether or not the preposition has a meaning independent from the verb being modified, although this is not as clear as in the case of noun modification because it depends on how general the meanings assigned to verbs and prepositions are.

Many works on formal semantics [Montague 73, Creswell 73, Shaumyan 77, Lewis 72] use logical forms based on higher-order logical systems, which make it easy to assign higher-order expressions as the "meanings" for modifiers. This avoids the problem of having to distinguish between place markers and content prepositions, and some related problems such as the distinction between intersective and non-intersective adjectives. All modifiers are uniformly translated as higher-order predicates applying to the modified item. However, the question is only removed to a next level, where the effect of the higher-order predicates on their arguments has to be defined. Once again, Woods [Woods 77a] explains the practical problem:

> ... the diversity of possible modifiers makes it unlikely that all adjectives and prepositional phrases could be interpretable as role fillers in any general or economical fashion. Thus the distinction between predicators and role fillers seems to be necessary.

The difficulties described by Woods are particularly evident when, as in the present work, the purpose of analysing a sentence is to evaluate its meaning against a database, a set of first-order statements of the kind "X did Y at Z". The criterion for an adequate analysis will then be that the "meaning" produced can be related to the database by our inference procedure. Of course, one of the main motivations of the theoretical work on formal semantics cited above is to find formal analyses of intensional constructions, for which a reduction to first-order predicate logic is not directly possible.

As the target formalism used in this work is basically first-order, it is clear that a modification must be translated in one of the two ways mentioned before:

* as a filler, introducing a new variable to occupy an argument place of the meaning of the modified word;
* as a restriction, constraining some already existing variables.

In some special cases, like those of the verbs "to be" and "to have", the apparent arguments cannot be treated as fillers or restrictions, because the verbs themselves cannot be given independent first-order meanings. Instead, the grammar specifies how those words manipulate their arguments (not their meanings!) as described below in Section 4.2.6. It might have been possible to translate such words by suitable higher order predicates, but that would require giving many other words complicated higher-order meanings as well. As I have already noted, the problem would not be simplified but only postponed. The same comments apply to non-intersective adjectives such as "average", discussed in Section 4.2.5.

4.2.3 Slots, Cases and Types

The criterion for word arguments I presented in the last section appeals of course to the reader's grammatical intuition. Studies on the semantics of verbs such as those of Shaumyan [Shaumyan 77] and Fillmore [Fillmore 68] have proposed a very small set of argument types and grammatical cases for all verbs, based on a very small set of "prototypical" verbs, such as "to go". Unfortunately, even in the case of the logic-based work of Shaumyan, it is not clear how those very general classes of verbs and arguments are to be instantiated to detailed rigorous semantics for particular arguments and verbs.

In the absence of suitable generalisations, each word must be

treated on its own. In this work, sentences are interpreted in some restricted domain, and in such a domain each noun or verb has a few alternative translations as a predicate. Each argument place has to be filled by a quantified variable to make a closed formula. In the intermediate representation, this filler will be the bound variable of some Quant.

I take the limited view that each argument place has associated to it a case marker, which defines the syntactically acceptable argument fillers. A case marker will be a preposition, or a verb argument role such as subject or direct object. Also associated to each argument place I have a type, which denotes the most general class of subject domain entities that may fill that place. Through these types, all variables in a formula become typed. The tuple

```
slot(case,type,argument)
```

is called a slot, following McCord's "slot grammars" [McCord 80b]. This notion of slot also owes much to Dahl's grammar [Dahl 77].

Each meaning of a word will then be described by a dictionary entry, or template, containing the translation of the word and a list of slots describing the fillers for the argument places of the translation. For example,

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

is a simplified version of a template for the verb "to flow" in a domain of geographical facts. It states that the verb "to flow" may be translated by the predication "flows_through(R,C)" where R corresponds to a subject slot of type "river" and C to a complement

slot marked by the preposition "through" and with type "country". Examples of slots as used in a practical application can be found in Appendix D.

Although I am not using general notions of case and argument, some generalisations are still possible. For instance, the genitive case (or the preposition "of") can be used in English to mark the argument of a "property" of entities, such as "size", "weight", etc. The grammar may then contain a general description of that kind of slot

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

stating that the noun Word names a property Pred with value Val of type Type that applies to objects Arg of type ArgType, with a syntactic realisation where the value of the property corresponds to the noun and the object with the property to the noun phrase filling the genitive slot of the property noun. A typical property template would be

```
property(area,measure,A,region,R,area(R,A),[]).
```

describing the "area" property as a the value A of type "measure" obtained by predicate "area" from objects R of type "region".

The example also shows that one of the arguments of the translation of a noun is never associated to a slot, namely the range argument that takes the quantified variable associated to the noun phrase of which the noun is the head. For example, the shape of the translation of the noun phrase "the area of France" would be

```
the(lambda(A).area(france,A), ...)
```

where A is the variable associated to the noun phrase. In the intermediate representation, the range argument is bound to the range variable of the Quant containing the translation of the noun. As any other argument, the range argument needs to have a type, which is matched against the type of whatever slot the noun phrase with this noun as head is going to fill.

If the entity classes that define types are structured in a "subclass of" hierarchy, types can be represented in a particularly elegant way, due to Dahl [Dahl 77]. A particular type will be represented by the path to the corresponding class from a root of the hierarchy. Then, two types will be compatible if one is an initial segment of the other. If paths are written as lists of class names terminated by a variable representing a further, as yet unknown, specialisation of the type, then two types are compatible if their representations are unifiable.

A language interpretation system using slots will have a set of general rules that describe what phrases can be used to fill each slot. Apart from the information contained in the slot itself, there are of course syntactic constraints on what phrases may be used to fill a particular slot. As explained earlier, I have chosen to have the syntactic constraints in a separate syntactic level (an XG), and have the slot filling rules operate on the result of the syntactic level. To decouple completely the syntactic level from the attachment rules, the XG uses the node closure constraints discussed in the last chapter. Of a set of syntactically acceptable analyses differing only on modifier attachments, the constraints only allow a "normal

form" analysis, where all modifiers are attached to the rightmost (deepest) phrase they can modify. The slot filling rules are able to recover from the normal form analysis other analyses where arguments are attached higher up, if a given attachment is in conflict with the case and type of a slot.

Usually, the meaning of a word will be defined in more generality than will be needed for any given occurrence of the word. This means that some of its slots will be left unfilled. I assume that all unfilled slots are optional, and fill the corresponding argument places with existentially quantified variables (we will see in Section 4.3 how such variables interact with questions and negation). This method simplifies the treatment of passives. A slot filling rule allows the subject slot of a transitive verb to be filled from a "by" prepositional phrase if the sentence is in the passive form. If the "by" phrase doesn't exist, the corresponding argument is given the right quantification by the rule for unfilled slots.

This approach to unfilled slots is too simplistic in two ways. First, a concept of obligatory slot might be useful to catch truly incomplete inputs. Adding such constraints doesn't seem to raise difficulties. Secondly, it is not possible to distinguish between the past participle of a verb and an homonymous adjective, unless the past participle is given a separate syntactic dictionary entry as an adjective, introducing a lexical ambiguity. For example, the rule for unfilled slots would make the sentence

The door is closed. (4.2)

equivalent to

The door was closed by something.

which is only one of the readings of (4.2). In the other reading, "closed" is an adjective describing the state of the door, and of course it doesn't presuppose an action of "closing".

4.2.4 Restrictions

I classify as restrictions all modifiers that constrain further a variable introduced by the phrase they modify. Thus, prepositional phrases which are not slot fillers, relative clauses, adjective phrases and participial phrases are all restrictions. The final placement of restrictions is determined by the syntactic rules that describe the relevant types of phrases, but also by type matching. A restriction may be seen as a predication lambda(X).P which is to be applied to a variable Y introduced by the phrase it modifies. The abstracted variable X corresponds to a dummy Quant which fills a slot in the complex formula P. Therefore, the type of that slot must be compatible with the type of the slot corresponding to variable Y. For example, the phrase "a country France borders" has the (simplified) Quant tree:

```
quant(a,X,country(X),border,true,Z)
                      /    \
              "France"      quant(id,X,true,true,true,X)
```

The dummy, or identity, Quant above has the abstracted variable X as bound variable, and the special determiner 'id' which signals that such a Quant doesn't introduce a quantification of the bound variable, but only "carries" the variable into slots.

As we saw in the last chapter, relative clauses, participial

phrases and adjective phrases are all treated syntactically as sentences. In the intermediate formalism, they will be represented by Preds. In general, a Pred translates a sentence or sentence-like phrase, and has the form

pred(subject,operator,predicate,arguments)

where **predicate** represents the explicit or implicit verb, **operator** represents the possible negation associated with that verb, **subject** is a Quant representing the explicit or implicit surface subject[20] of the verb, and arguments is a list of trees representing the arguments of the verb of the Pred. In the case of participial and adjective phrases, the predicate is always equality, translating an implicit "to be" that has either been introduced in the syntactic analysis (participial phrases) or by the attachment rules (adjective phrases); the subject is an identity Quant repeating the bound variable of the modified Quant. In the case of relative clauses, the trace is similarly translated as an identity Quant (see example above).

Restrictions may be grouped in conjoined phrases. This is the only type of conjunction I have looked at, in view of the limitations of the XG formalism discussed earlier in Section 6.1. The intermediate representation of conjoined restrictions is done by Conj nodes, of the form

conj(conjunction,left predicate,left modifiers,
right predicate,right modifiers)

The **conjunction** field describes the lexical conjunction for this

---

[20]The surface subject role must be available to determine operator scope.

node, and the left (right) predicate and modifiers fields have the same purpose as the corresponding fields in a Quant, for the translation of the phrase to the left (right) of the conjunction.

Clearly, conjoined restrictions are treated as a single restriction, except that the dummy Quant for the abstracted variable is replicated in both sides of a conjunction. That is, a restriction

p and q

is translated as

lambda(X).(lambda(Y).p')(X) and' (lambda(Z).q')(X)

4.2.5 Adjectives and Higher-Order Operations

In this work, I make no attempt to examine in general the semantics of adjectives. However, it is possible to draw from existing theory [Bartsch and Vennemann 72] to provide a simple but useful treatment of adjectives and related constructions in a grammar for a specific domain. Some of the ideas to deal with aggregations (see below) are also very similar to those used in LUNAR [Woods et al. 72, Woods 77a].

Intersective adjectives, whose meaning just conjoins with that of the phrase they modify, are of course the easiest to treat, and are translated into extra predications restricting the range variable of the Quant for the noun phrase where they appear.

Comparatives and superlatives are seen as referring implicitly to some property of the modified noun. The values of such a property must be members of some ordered domain, and the comparison (or the taking of the superlative) will be referred to the property values of

the objects being compared. For example, if the adjective "larger", when applied to countries, is understood as referring to areas, then the sentence fragment

Every country larger than France ...

will be roughly translated as

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

Of course, the interpretation of superlatives and comparatives depends on what they apply to, so a dictionary will contain several translations for the same adjective. The semantic types of the objects being related by the adjective will then be used to select among those alternative translations.

As can be seen from the example above, a comparative introduces a restriction in the same way as other restrictive post-modifiers, and in fact, as we have seen before, there are arguments for treating comparatives in the same way as participial clauses, as reduced relatives with the implicit main verb "to be". Comparatives are thus a special case of intersective adjectives.

Superlatives, on the other hand, are clearly non-intersective. In fact, superlatives do not apply to a noun, but to the whole fragment of noun phrase they precede. That is,

the largest country in Asia

refers to the largest of the Asian countries, and not to the largest of all countries, which happens to be in Asia. This property makes

superlatives similar to other non-intersective adjectives, such as 'average'. I call such adjectives aggregations.

In general, an aggregation can be seen as a second-order predicate which applies to a predication p with two free variables R (the range) and V (the values) to produce some object O:

adj(lambda(R,V).p,O)

Thus, the noun phrase fragment "largest country in Asia" roughly translates into

largest(lambda(C,A).country(C) & area(C,A) & in(C,asia), L)

where 'largest' selects objects in the range for which the value is not smaller than for any other object in the range. In a similar way, 'average' selects the average of the values over the range objects.

One major problem with aggregations is to determine the predication to which they apply when they occur in a noun phrase with complex post-modifiers. The scope of operators influences this decision, so this topic will be further discussed in Section 4.3.3 below.

Nouns such as 'average' and 'sum' in the context

<noun> of <noun phrase>

behave in much the same way as aggregations, so the discussion above applies to them as well.

4.2.6 Words Which Look at Their Arguments

I have pointed out that it is difficult (or laborious) to give an independent meaning to certain words, whose role is to take their arguments and connect them together in some way. The definitions of these words in the grammar are better seen as describing operations on the syntax trees for the arguments. In this sense, these words have to "look" at their arguments, and so there must be special rules for them, instead of the general attachment rules.

The verb "to have", used as a main verb, falls into this category. The sentence templates

<entity> has <attribute> of <value> (4.3)
<entity> has <value> as <attribute> (4.4)
<entity> has <attribute> (4.5)

state that some (perhaps unspecified) attribute or role <attribute> of entity <entity> is fulfilled by the value or entity <value>. Ie., the sentence can be roughly paraphrased as

<attribute> of <entity> is <value>

This paraphrase is only approximate, because in general the determiners which make the paraphrase reasonable English are different from those used in the original sentence. For example, compare the intuitively equivalent sentences

London has a population of less than 10 million
The population of London is less than 10 million

The analysis outlined here is similar to that used in the USL system [Lehmann 78, Zoeppritz 81].

Of course, the attachment rules that describe this role of the verb

"to have" need not be concerned with the details of paraphrasing. The rules have only to relate a filler <entity> for the subject role of the verb to the arguments <attribute> and <value> of the verb. For this to be possible, the template used for translating the head of <attribute> must have an unfilled genitive slot which can be filled by <entity>, and the type of <attribute> must be compatible with that of <value>.

Let det x and pred x be the translations of the determiner and of the conditions (head, predication and arguments) of noun phrase **x**, respectively. Ignoring the problems of determiner scope, the translation of (4.3) or (4.4) will be

```
det <entity> (
   lambda(E1).pred <entity>,
   lambda(E2).(
      det <attribute> (
         lambda(A1).pred <attribute> (E1),
          lambda(A2).(
             det <value> (
                lambda(V1).pred <value>,
                lambda(V2). A2 = V2 ) ) ) ) )
```

which may be read informally as "the entities E are such that their attributes A are the same as the values V." For example, the translation of "Some European country has a population of less than 10 million" will have the translation[21]

```
some(
   lambda(C1). european(C1) & country(C1).
   lambda(C2). a(
      lambda(P1). population(P1,C1),
      lambda(P2). less_than(
         lambda(V1). V1 = "10 million",
         lambda(V2). P2 = V2 ) ) )
```

[21]For simplicity I treat here "less than" as a determiner.

Taking both "some" and "a" as first-order existential quantifiers, the translation simplifies further to

```
exists(C, european(C) & country(C),
   exists(P, population(P,C) & P < "10 million" ) )
```

In the case of a sentence of type (4.5), the default rule for unfilled slots will just introduce an existentially quantified variable instead of the noun phrase <value>. That is, the sentence

John has a car

will be translated roughly as

The car of John is something.

Some uses of the preposition "with" can be translated in a similar way, by seeing "with" as equivalent to "having". More precisely, the uses of "with" to which this observation applies are those where the object of the preposition is a necessary property of the subject, such as "population" or "weight". Of course, the translation is not valid when the object of the preposition is a contingent (as opposed to a necessary) feature of the subject.

Another case of a word which is best treated as "looking" at its arguments is the noun "number". This is obvious if we consider the two noun phrases

the number of the component
the number of components

Depending on whether the argument noun phrase has a determiner or is generic, the noun "number" will be translated as a property of the argument or as a higher order predicate applying to its argument.

4.3 What Governs What?

Take an assertion containing more than one operator, such as:

An ambassador visited every European country.

Should the sentence be read distributively, with possibly different ambassadors visiting the different countries, or collectively, with a single ambassador visiting all the European countries? It is hardly possible to say that there are compelling reasons to chose one reading over the other. If the article "all" had been used instead of "every", the collective reading would gain strength. If "each" had been used, the distributive reading might be preferred.

But the ambiguity is not fatal. Even without resolving it, some information can be obtained from the sentence[22] . The Quant trees of the previous section may be seen as an abstraction of this information.

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

But the operators in a DCW clause (or, more precisely, the translations of operator words) sit in precise positions with respect to each other: for any two operators which operate upon a common predication in the sentence, one will be in the scope of the other.

[22]Also, on hearing such sentences we might not even be aware of the problem, until some question brings it to the foreground.

In other words, its meaning will be governed by the meaning of the other. Thus, the distributive reading for the example corresponds to the operator for "a" being in the scope of the operator for "every", whereas the collective reading puts "every" in the scope of "a". These two scope orderings are the only possible ones for the sentence because both operators operate on the verb "visited".

As suggested above, the operators used are not in general sufficient to derive a unique choice of operator scopes. The lack of accepted scope rules is so drastic that it has led Vanlehn [Vanlehn 78] to propose that operator scope is an epiphenomenon, which has no reality in the hearer's understanding of a sentence, but is only superimposed by an observer upon a behaviour based in altogether different, if unspecified, mechanisms.

Vanlehn's view, however, conflicts with the basis of this work, that a sentence of logic is a useful, if limited, abstraction of the meaning of a sentence. After all, Vanlehn doesn't dismiss as epiphenomenic the distinction between collective and distributive readings of a sentence. That distinction can only be reflected in first-order logic by the distinction between sentences with the same predications but different operator scopes. But it is clear that the simple scope mechanism in standard first-order logic forces scope decisions even where there are no grounds for those decisions[23]

[23] I am constrained here to deal with systems of logic in which mechanical inference is practical. The theoretically motivated system of Branched Quantification [Hintikka 74], allows several operators (quantifiers) to operate on a predication without any of them being in the scope of another.

Although I do not have a comprehensive set of scope rules, which in any case may well be an illusory goal given that the target language forces scope decisions for sentences in which there may not be a "scope intention", it is possible to formulate some coarse rules which produce reasonable readings in practice. This has been the approach taken in the LUNAR system [Woods et al. 72, Woods 77a], and my rules may be seen as a reconstruction and extension of Woods's. A similar set of rules, more limited in some ways but based on very elegant linguistic abstractions, was developed independently by McCord [McCord 80a].

4.3.1 Determiner Precedence

The determiner scope rules assume as default scoping that given by the order of quantifications in a Quant tree, and just list the exceptions to this default. The exception rules 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". In short, determiner B governs determiner A. The same mechanism is used to specify the behaviour of determiners with respect to other operators, such as negation or the implicit "interrogative operator" in questions (more on this below).

Because the determiner rules use no pragmatic information from the particular words being used, they cannot make fine distinctions. In Chat-80, determiners are divided into two classes, "strong" and "weak" determiners, with any strong determiner governing any weak one. The strong determiners are "each" and "any". All other determiners are weak. The exception rules give the determiner of "country" in

the population of each country
the population of any country

wider scope than the determiner of "population", which makes the value of "population" depend on the "country". But the default rule gives the determiner of "country" in

the population of every country

narrower scope than the determiner of "population". This is a rather dubious reading, given that "population" is a function of "country", a piece of information which I am not using. However, the other scoping can always be achieved by using "each" instead of "every", and we are thus left with a means of expressing a narrow scope universal quantification, which otherwise would not be possible.

The determiner field in a Quant may also be filled by symbols which do not correspond to English determiners, but to certain special roles of the corresponding Quant in the Quant tree. These include Quants for unfilled slots, labelled by 'void', and identity Quants (Section 4.2.4), labelled by 'id'. Of course, these "determiners" do not govern any other determiners.

The above description of the exception rules uses the notion of a Quant node being "above" another Quant node. Now, a Quant node in general dominates two subtrees, the node's predication and the node's modifiers. As mentioned in Section 4.2.1, this division of lower nodes into predication and modifiers is intended to capture different scope properties of different post_modifiers. The predication comprises post-modifiers, such as relative clauses, from which no determiner is allowed to escape to have a larger scope than the

modified node. In contrast, the modifiers of the Quant are phrases such as arguments and other prepositional phrases, whose determiners may be moved by the exception rules to have a larger scope than the modified node[24]. Therefore, the notion of a node being "above" another node is relevant only for a Quant and the Quants in its subtree of modifiers, as a Quant is necessarily "above" its predication.

The two classes of subnodes of a Quant, predication and modifiers, are only the extremes of a range of scope relationships between a noun phrase and its post-modifiers. The scope judgments of people vary in sharpness as one moves in this range, from the full relative clause at the predication end of the range to a noun argument at the modifier end of the range:

A friend that visited each member of the club was also invited.
A friend visiting each member of the club was also invited.
A friend of each member of the club was also invited.

Vanlehn's experiments [Vanlehn 78] show that, even at the extremes of that range, it is possible to contrive situations and sentences in which at least some people disagree with the scopes produced by the predication/modifier distinction I use here. Nevertheless, this distinction together with the other scope rules appears not to produce strongly counter-intuitive scopings, at least in the restricted domain of question-answering in Chat-80.

[24] Cooper [Cooper 79] proposes a similar distinction, and sees it as the effect of the complex-NP constraint on the possible movements of quantifications.

I have discussed the scope relationship between a Quant node and lower nodes in the Quant tree. But one has also to consider the scope relationships between nodes that are immediate descendants of a single Quant or Pred, as it is clear from examples such as:

[At least one country] borders [each European country].
[At least one country] borders [all European countries].

In general, the exception rules for siblings may have to be different from the rules discussed previously (see McCord's rules [McCord 80a]). However, for the limited kinds of operators I am dealing with, the same rules can be used for both situations. This can be justified by noting that the sibling rules are mostly used to decide the relative scopes of the subject and a verb argument in a sentence, which is a similar relationship to that between a noun phrase and the argument of a prepositional phrase that modifies it.

One should note finally that the exception rules for precedence are applied to the results of applying the rules to lower nodes of the Quant tree. Thus, a Quant with a "strong" determiner will move up and to the left until it finds another "strong" determiner. When a Quant cannot move further up or to the left, its final position has been found, and it is then translated into the corresponding logical form, by applying it to its range (made of the Quant's head, predication and the modifiers weaker than the Quant) and to its scope (the translation of the nodes over which the determiner was moved by the default rules).

4.3.2 Plural Determiners and Sets

As described before (Section 2.3), a useful first analysis of definite plural noun phrases is to treat them uniformly as introducing sets. This analysis is pushed to its limits by Colmerauer [Colmerauer 79a] and Dahl [Dahl 77], who translate *every* noun phrase as a set expression. Thus, the predicates which translate words would take sets as their arguments, and operate on those arguments according to their peculiarities. In this way, the distinction between collective and distributive predications would be removed from the translation rules to a level where the predicates that translate words would be analysed once and for all in terms of the available database predicates. For example, an adjective such as "parallel", which applies to a set as a whole (a collective predication), would be analysed by the same rules as "red", which presumably applies to all entities described in a plural noun phrase separately (a distributive predication).

This approach, however, has severe shortcomings. The least of these is the lack of economy in description caused by treating separately each word in large uniform classes, such as the class of intersective adjectives. A more important practical difficulty arises when translating nested definite plural noun phrases. Intuitively, in a noun phrase such as

The children of the employees (4.6)

we are considering each employee in turn, and finding who his or her children are. A reasonable answer to the question

Who are the children of the employees? (4.7)

would be a table of employees with their children. Clearly, such a result cannot be obtained by taking a set of children and a set of employees and applying the predicate 'child_of' to elements of these sets to produce a new set of children.

Actually, the problem in the work of Colmerauer and Dahl is even more deep seated. Even if we were satisfied with producing a set of children as the result of that question, that set could not be derived by taking 'child_of' to be either distributive or collective. Seen as applying to sets, the predicates which translate the first head noun in

x of y (4.8)

constructions must be characterised in a different way. A possible analysis is that such a predicate p applies to sets x (children in (4.6)) and y (employees in (4.6)) 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 approach I have taken to produce reasonable answers to questions like (4.7) is to translate phrases of the kind (4.8) by indexed sets. For example, the translation of the noun phrase (4.6) is the set of pairs:

{ (E,SC) : employee(E) & SC = { C : child_of(C,E) } }

That is, we have sets of children SC indexed by employees E. A deeper nested noun phrase, such as:

the populations of the main cities of the European countries

would be translated by

```
{ (E,C,SP) : european_country(E)
          & main_city_of(C,E)
          & SP = { P : population_of(P,C) } }
```

Note that in this example there is a single value of population for each city, and therefore all the sets of values of population, indexed by country and city, are singletons.

Although this translation seems adequate for definite noun phrases which are used to answer questions, if the noun phrase is an argument for some other predicate, that predicate will have to be capable of operating on indexed sets.

Non-nested definite plural noun phrases are translated by sets as in the Colmerauer approach. Thus, we have still the problem of defining the effect of predications on sets, and also on indexed sets. However, this further complication seems to be justified by the improvement in the answers to questions which motivated the introduction of indexed sets.

The interpretation rules construct an indexed set when they find a definite plural noun phrase whose head noun has a suitably marked slot filled by another definite plural noun phrase. Thus the dictionary entries for nouns will determine the argument slots which may lead to the construction of an indexed set. In practice, the relevant slots seem to be those marked with the genitive case.

One should note that the projective reading of a predicate over

sets is implicit in the indexed set concept. The union of all the elements of an indexed set (which are sets, of course) is just the projection of the predicate on its first argument, as discussed above.

The translation of agreggations is another area where indexed sets are useful. A noun phrase such as

The average of the areas of the European countries

will be interpreted by averaging area values over the set of European countries, which is exactly the set of indices from which the indexed set for "the areas of the European countries" would be built.

4.3.3 Higher-Order Predicates

The predication to which an aggregation or other higher order operator applies is influenced by determiner scope. For example, in the noun phrases

the average area of the countries in each continent
the number of countries in any continent

the preferred readings seem to be distributive on "continents". Such a reading can be simply achieved by applying the same scope rules as for phrases without aggregations or higher order operators. Taking the first example noun phrase above, the determiner "each" is stronger than the determiner "the" of the aggregation, and therefore the translation of the noun phrase will be

```
each(
   lambda(C).continent(C),
   lambda(C).
      the(lambda(V).
         average(lambda(A,C1). country(C1) &
                               area(A,C1) &
                               in(C1,C),       V),
                               ... ) )
```

4.3.4 Negation and "any"

The scope relationships between negation and determiners are dealt with by essentially the same mechanisms which are used for the relative scope of determiners. In fact, the only exception rule is that "any" governs negation. The reasons for this are discussed below.

One should remember that negation corresponds in the Quant tree to a field of the Pred node for the negated verb. This verb might, of course, not be explicit in the original sentence, but be an implicit "to be" from a noun post-modifier, as in

every country [that is] not smaller than France

The operator for a non-negated Pred is 'id', which is governed by all "real" determiners.

The determiner "any" may be seen as an existential quantifier with narrow scope with respect to negation, or as a universal quantifier with wide scope with respect to negation. The translation given in Section 2.3 assumes the latter interpretation. Both interpretations give equivalent readings in declaratives or WH-questions with an explicitly negated main verb. However, both interpretations have

problems in sentences without explicit negation[25]. What do the two above interpretations of "any" imply for sentences without negation? In a sentence such as

John likes any girl.

the narrow existential translation would produce

exists(G, girl(G) & likes(john,G))

which is not acceptable. In contrast, the wide universal translation would give an acceptable interpretation:

every(G, girl(G) ⇒ likes(john,G))

Furthermore, the same wide scope universal translation of "any" would work for yes-no questions if we could assume a topmost 'yes-no' operator with the same properties as negation. This is just the case with the translation I am using for questions. The question "Does John like any girl?" would be translated as

every(G, girl(G) => (likes(john,G) => answer(yes)))

which is logically equivalent to the intuitive reading

answer(yes) <= exists(G, girl(G) & likes(john,G))

The universal interpretation of "any" has an interesting similarity with the role of free variables in clausal logic. The parallel is apparent if we consider the clauses

[25]One problem area which I will not discuss is that of verbs (such as "to deny"), adverbs (such as "hardly") and phrases (such as certain comparatives) that may be seen as behaving with respect to "any" as if they contain an implicit negation [Seuren 69]. Another problem that I will not consider is that of the "indefiniteness" of "any" compared with a universal quantifier such as "every".

mortal(X) <= human(X)

<= mortal(X)

and their respective renderings in English

Any human is mortal.
Is there any mortal?

This view of the role of "any" still leaves out the cases of WH-questions without a main verb negation and yes-no questions with negated verb. However, sentences with "any" in any of those contexts seem awkward at best, and may well be considered ungrammatical, as for example

* Which countries border *any* ocean?

4.3.5 Questions and "Each"

In the previous sections, it became apparent that the treatment of questions requires the introduction of top level operators that indicate that the underlying sentence is questioned. These operators are used as implicit determiners at the top of the Quant tree, and we have seen that it is possible for a determiner to move above them if the corresponding exception rule is available. This is particularly useful for the treatment of the determiner "each" in a WH-question. It is reasonable to assume that an "each" noun phrase which is not dominated by a negation specifies an indexing of the answers to the question by the objects described by the noun phrase. Therefore, "each" is assumed to govern the WH-question operator.

As we have seen in Section 2.3, a WH-question is translated as a clause for an "answer" predicate, defining the answers in terms of the conditions expressed in the sentence. For example, the question

Which countries border France?

is translated as

```
answer(C) <= country(C) & borders(C,france)
```

In fact, the WH-question operator corresponds roughly to the occurrence of '<='; therefore the universal quantifier that translates an "each" determiner will have the widest scope and may be omitted as usual. On the other hand, the pragmatics of an WH-question require an answer that includes the values of the "each" noun phrases. Combining the scope argument with this pragmatic requirement, the translation of a sentence such as

Which countries border each European country?

will be

```
answer(E,C) <=
   country(E) & european(E)
 & country(C) & borders(C,E)
```

4.4 Summary

In this chapter I have discussed the main aspects of a mechanism to translate syntactic analysis trees into DCW clauses.

An explicit mechanism for semantic interpretation is needed because the construction of the meaning of a sentence from its parts cannot in general be mapped directly into composition of first-order formulae. However, this theoretical disadvantage of a first-order language with respect to higher-order languages such as intensional logic was overshadowed by the need to deal with constructions whose meaning does not appear to be expressible in compositional terms, bringing back the need for explicit translation machinery beyond the composition of target language formulae.

The mechanism I propose consists of two sets of semantic/pragmatic rules, attachment rules and scope rules, linked by an intermediate representation, Quant trees. Attachment rules deal with the attachement of arguments and modifiers to words, scope rules with deciding the relative scope of operators such as determiners and negation and building a final logical form. Both sets of rules are strongly motivated by pragmatic considerations related to their use in translating database queries, as required in the Chat-80 program.

The principles used in the rules are partly borrowed from earlier work [Woods 77a, Dahl 77, Bartsch and Vennemann 72, Vanlehn 78, Zoeppritz 81] and partly extensions or improvements on existing work (in particular the treatment of plural "the" and of "any").

In Appendices D and E, I give the Prolog programs that implement the slot filler and the scope rules in Chat-80. Both the slot filler program and the scope determination program are tree rewrite programs, where to each node type in the input tree is associated a set of clauses that defines how that node type is transformed in fragments of the output tree. In the slot filler, the input tree is the parse tree produced by the syntactic grammar, and the major predicates correspond to the various non-terminals in the tree; the output is a Quant tree. In the scope determination program, the input is a Quant tree, and the output tree is a DCW clause. Slot filling is a non-determinate procedure, because of the potential alternatives in choosing templates and in attaching post-modifiers. In contrast, scope determination is a determinate procedure, always resolving potential scope ambiguities by using the default left-to-right scope ordering rule.

Chapter 5

An Application to Database Access

The ideas presented in the two preceding chapters have been used in Chat-80, a Prolog program which answers questions about a database. The purpose of writing Chat-80 was to show that logic-based formalisms for query representation and for actually expressing the grammar and semantic interpretation rules make it possible to implement the major components of a natural language interface to databases in a modular, modifiable and very efficient manner. Chat-80 is not a finished system, and for it to be used in practice it would need substantial changes and additions, such as some form of user-oriented error detection and recovery, and extensions to the semantic rules to handle pronominal references. For the latter, it would be possible to incorporate in the semantic rules some of the methods proposed by Pasero [Pasero 73] and by B. Lynn-Webber [Nash-Webber and Reiter 77].

Chat-80 translates questions into DCW clauses, and proceeds to execute the clauses against the database. Because in general DCW clauses are not directly executable by Prolog, and even when they are the Prolog proof procedure is too inefficient for this kind of database access, DCW clauses are first transformed into clauses which are suitable for direct execution by Prolog. This transformation process, due to Warren, is described elsewhere [Warren 81a].

5.1 Organisation

The discussion in the two preceding chapters implies a division of the mapping from English sentences into logical formulae into a composition of three mappings: parsing, from sentences to parse trees, slot filling, from parse trees to Quant trees, and scoping, from Quant trees to DCW clauses.

The simplest computational realisation of this conceptual layout is of course as one procedure for each mapping, with the procedures invoked following the composition order of the mappings. Such an organisation for a language analysis process has been criticised [Mellish 81] as forcing the procedures that are called first to make decisions on insufficient information, where the missing information may well become available from the application of the results so far of procedures further along the sequence. This criticism depends of course on accepting that those decisions for which there is not enough information in the earlier stages must be taken there. However, as I have pointed out in Sections 3.10 and 4.2, it is possible, at least in the case of modifier attachment, to postpone such decisions to later stages but still have something left to do in the earlier stages.

We should also remember that this sequential realisation of the conceptual layout is not the only possible. The rules that describe the three mappings do not impose any particular execution method. Although I have not explored this aspect, it would be in theory possible to interleave the application of syntactic, slot filling and scope rules, to stop useless rule applications at an earlier stage.

As the rules stand, however, the scope rules specify a total function, and therefore would give no information to earlier passes. Also, as will be seen below in the discussion of the parser, the remaining local ambiguities in the grammar do not seem to cause an intolerable combinatorial explosion.

The Chat-80 parser is just the combination of an XG with the proof procedure for definite clauses embodied in Prolog. The XG was outlined in Sections 3.9 and 3.10, and is listed in full in Appendix C. Below, I make a few observations about the efficiency of the parser.

Slot filling and scoping are both described by definite clause programs. In a small number of the clauses, I use the non-logical Prolog operator '!' ("cut") [Pereira et al. 78]. Its role in those clauses is simply to avoid having to list explicitly a finite number of alternative cases, and the logic of the programs is not affected by this. The rules are used directly as a Prolog program, and the Prolog proof procedure supplies the search over templates required in filling slots. Details of the slot filling and scoping predicates, including the structure of Quant trees, can be found in appendices D and E.

5.2 Efficiency

Both the syntactic analysis and the semantic interpretation times for the sentences in Appendix F are well under 100 msec. Over those examples, the average syntactic analysis time per input word is 3.3 msec. (standard deviation = .8 msec.), and the average interpretation

time is 3.7 msec. (standard deviation = 1.2 msec.). In terms of the speed of Prolog on the DEC KL-10, these figures correspond to an average of 120 procedure calls per input word in syntactic analysis, and 130 in semantic interpretation.

Because of the amount of work done in the area of syntactic analysis, it is most interesting to discuss the performance of the parser in detail. This is the subject of the next section.

5.2.1 The Parser

Executed by Prolog, the Chat-80 XG behaves as a top-down backtrack parser. Using this parsing strategy means that left-recursive rules cannot be used, which makes the expression of some constructions, such as possessives, more complicated than it would be using unrestricted rules.

Although in the worst case top-down backtrack parsing has exponential time complexity (given a sufficiently perverse grammar) [Aho and Ullman 72], this kind of theoretical bound seems of no relevance for the parsing of natural language sentences. On one hand, the size of the grammar is in practice much more important to performance than the length of the input. The reason for this is that actual sentences are fairly short, whereas adding a new construction to the grammar increases the number of alternative rules for some non-terminals and the complexity of the parse trees for others. On the other hand, the efficiency with which backtracking is implemented in Prolog more than offsets the cost of exploring blindly some parsing alternatives, a cost which anyway is bounded by the

above mentioned practical limits on the length of the input. This wouldn't be so only if English were a deterministic language in a very strong sense (e.g. LALR(1) [Aho and Johnson 74]), which is clearly not the case. As it is, a more determinate parsing strategy could only be of practical benefit if the cost of achieving this determinacy were less than the cost of exploring very efficiently some wrong paths, which has yet to be demonstrated for natural language (as opposed to artificial languages, whose design is already biased towards some form of determinacy).

One should note, however, that other parsing mechanisms could be used with the same grammar. In fact, it is possible to extend many well known context-free parsing methods to deal with grammars based on definite clauses. This applies in particular to tabular (well-formed substring table) methods, such as the Earley parsing algorithm [Earley 70], chart parsing [Kay 80], and the Pratt algorithm [Uehara and Toyoda 81]. The advantages that these might have in dealing with certain problems caused by ambiguity or by incomplete input will be discussed later (Section 6.2). On the other hand, as we will see below, other overheads inherent in tabular methods eliminated them from consideration in deciding for a parsing method for Chat-80.

I have done some rough estimates of the work done by parsers with the Chat-80 grammar, both with Prolog's top-down backtrack method and a top-down method with a well-formed substring table. The data were the sentences in Appendix F. Because of the attachment mechanism in the grammar, all those sentences have a single analysis.

First, I will show the averages and standard deviations of the

ratios of completed non-terminals and attempted non-terminals to the number of non-terminals in the analyses:

Completed		Attempted	
average	std. deviation	average	std. deviation
1.77	.98	4.5	.97

That is, more than 50% of the non-terminals completed before the first (and only) analysis are useful for that analysis, and more than 20% of the non-terminals attempted before the first analysis are useful for the analysis.

A tabular parsing method might avoid duplication of effort, and reduce the ratio between completed and useful non-terminals; look-ahead might prevent futile non-terminal expansions, reducing the ratio between attempted and useful non-terminals. However, in avoiding duplication of effort a tabular parsing method must nonetheless make sure that all possibilities are eventually examined, and the book-keeping information required for this in the tabular method is in general much more complicated than in top-down backtrack parsing. Basically, a table of already completed non-terminals is only useful if it is exhaustive, because otherwise we do not know whether for completeness we need to expand again some non-terminal in the table. General tabular parsing methods, such as the Earley algorithm [Earley 70] and chart parsing [Kay 80], ensure this by storing not only completed non-terminals (passive nodes) but also partial analyses (active nodes). The efficiency question is whether the cost of storing those partial analyses is justified for the elimination of duplicated work. The results that I show now suggest that, at least for the Chat-80 grammar, that theoretical advantage of tabular parsers is not realised.

I have used for the comparison a chart parsing algorithm which is as near as possible to top-down backtrack parsing, while still being able to avoid repeating analyses[26]. The following table gives the average and standard deviation of two interesting ratios over the sentences[27] of Appendix F. The first ratio is that of completed non-terminals in the Chat-80 parser to completed non-terminals) in the tabular parser; the second is the ratio of attempted analyses in the Chat-80 parser to stored partial analyses in the tabular parser.

Passive nodes		Active nodes	
average	std. deviation	average	std. deviation
1.2	.09	.75	.06

That is, the top-down backtrack parser in average only wastes 16% of its effort, but needs to examine only 75% of the analysis paths that the tabular parser examines[28].

The tabular parser, in contrast to the top-down backtrack parser, has to store all the partial analyses, and check that they are not being repeated, for the whole duration of the analysis. Furthermore, for the examples examined, the number of partial analyses stored is 6 times the number of useful non-terminals. The time and space costs of this exceed by far the wasted effort of the backtrack parser.

---

[26]Based on a notion of scheduling due to Henry Thompson [personal communication].

[27]With two omissions due to space problems in the tabular parser.

[28]In fact, the above comparison is somewhat biased towards the chart parser, because active nodes starting with a terminal were only stored if the terminal coincided with the next input word, whereas the Chat-80 parser has no look-ahead at all. The look-ahead in the chart parser was needed to make the examples run within the memory constraints of my naive implementation.

5.3 Sample Interactions

I give in this section examples of the operation of the language analysis part of Chat-80. In Appendix F, Chat-80 interactions and timings are listed in full, including Warren's query optimisation, which is not part of the present work. The examples that follow give the input sentence, a simplified parse tree, and its translation into a DCW clause. The timings for these and other examples on a DEC KL-10 computer are given in Appendix F.

Does Afghanistan border China?

```
q
   s
      np
         name [afghanistan]
      verb [border]
      arg
         np
            name [china]

answer <=
   borders(afghanistan,china)
```

Figure 5-1: Yes-no question

What is the capital of Upper-Volta?

```
whq
   X
   s
      np
         wh(X)
      verb [be]
      arg
         np
            np_head
               det [the(sin)]
               noun [capital]
            pp
               prep [of]
               np
                  name [upper_volta]

answer(X) <=
   capital(upper_volta,X)
```

Figure 5-2: WH-question

```
Where is the largest country?

whq
   X
   s
      np
         np_head
            det [the(sin)]
            sup
               most
               adj [large]
            noun [country]
      verb [be]
      arg
         pp
            prep [in]
            np
               np_head
                  int_det(X)
                  noun [place]

answer(P) <=
   exists(C,
     exists(S,
       setof(A:C,
         country(C)
       & area(C,A), S)
     & aggregate(max,S,C) )
   & place(P)
   & in(C,P) )
```

Figure 5-3: WH-question with implicit prepositional phrase

```
Which country's capital is London?

whq
   X
   s
      np
         np_head
            det [the(sin)]
            noun [capital]
         pp
            poss
            np
               np_head
                  int_det(X)
                  noun [country]
      verb [be]
      arg
         np
            name [london]

answer(C) <=
   country(C)
 & capital(C,london)
```

Figure 5-4: Possessive and nested trace

What are the capitals of the countries bordering the Baltic?

```
whq
   X
   s
      np
         wh(X)
      verb [be]
      arg
         dir
         np
            np_head
               det [the(plu)]
               noun [capital]
            pp
               prep [of]
               np
                  np_head
                     det [the(plu)]
                     noun [country]
                  reduced_rel
                     Y
                     s
                        np
                           wh(Y)
                        verb [border]
                        arg
                           np
                              name [baltic]

answer(S) <=
   setof([Co]:CaS,
     country(Co)
   & borders(Co,baltic)
   & setof(Ca,
       capital(Co,Ca), CaS), S)
```

Figure 5-5: Nested plural "the" and reduced relative

What is the average area of the countries in each continent?

```
whq
   X
   s
      np
         wh(X)
      verb [be]
      arg
         np
            np_head
               det [the(sin)]
               adj [average]
               noun [area]
            pp
               prep [of]
               np
                  np_head
                     det [the(plu)]
                     noun [country]
                  pp
                     prep [in]
                     np
                        np_head
                           det [each]
                           noun [continent]

answer([Cont,Aver]) <=
   continent(Cont)
 & exists(S,
     setof(A:[C]
       area(C,A)
     & country(C)
     & in(C,Cont), S)
   & aggregate(average,S,Aver) )
```

Figure 5-6: Aggregation operator and "strong" determiner

What are the countries through which no river flows?

```
whq
   X
   s
      np
         wh(X)
      verb [be]
      arg
         np
            np_head
               det [the(plu)]
               noun country
            rel
               Y
               s
                  np
                     np_head
                        det [no]
                        noun [river]
                  verb [flow]
                  pp
                     prep [through]
                     np
                        wh(Y)

answer(S) <=
   setof(C ,
     country(C)
   & \+
       exists(R,
         river(R)
       & flows(R,C) ), S)
```

Figure 5-7: "Pied piping" in relative clause

Is there some ocean that does not border any country?

```
q
   s
      [there]
      verb [be]
      arg
         np
            np_head
               det [some]
               noun [ocean]
            rel
               X
               s
                  np
                     wh(X)
                  verb [border]
                  arg
                     np
                        np_head
                           det [any]
                           noun [country]
```

```
answer <=
   exists(0,
     ocean(0)
   & \+
       exists(C,
         country(C)
       & borders(0,C) ) )
```

Figure 5-8: "Any"

Chapter 6

Discussion and Further Work

In this chapter I look at some limitations of the work presented in the previous chapters, and suggest ways in which these limitations might be overcome. The discussion is brief and selective, as the limitations of a given technique were discussed in most cases when the technique was introduced.

6.1 Limits of XGs

In Chapter 3, I showed how certain English constructions, which have been usually described in terms of transformations or, more recently, metarules, can be easily described in a "one level" formalism, XGs. These constructions include WH-questions, relative clauses and auxiliary inversion. However, other fundamental constructions cannot be described so easily in a "one level" formalism. I will look now at one of those, conjunction.

Even assuming that only conjunction of full phrases is allowed, there is both a major theoretical difficulty and a practical problem in trying to describe conjunction with XGs. The practical problem is easily stated: for each phrase type (non-terminal), a complicated set of rules is needed to describe the conjunction of phrases of that type. This can be seen in the rules for 'relative' in the Chat-80 grammar in Appendix C.

The theoretical difficulty has to do with the interaction of left extraposition and conjunction. For example, in the phrase

the letter that$_i$[$_s$Mary [$_{vp}$[$_{vp}$wrote t$_i$]and [$_{vp}$sent t$_i$]]

there are two traces for the relative pronoun, one in each of the conjoined verb phrases that make the verb phrase from which a trace must be extraposed. That layout is obligatory. That is, the following is ungrammatical:

* the letter that$_i$[$_s$Mary [$_{vp}$[$_{vp}$wrote t$_i$]and [$_{vp}$sent a book]]

This phenomenon, called across the board deletion in transformational grammar, cannot be described in an XG where extraposition is described just by extraposition rules. Two reposition the same constituent into two different non-terminals, it would be necessary that the extraposition arguments of the two non-terminals be the same. But this is not possible because the contents of the extraposition list change in parallel with the left-to-right order of constituents, and therefore it is not possible for two constituents to have the same extraposition arguments.

To put the problem in more technical terms, consider a non-terminal 'nt'. Ignoring for the moment the details of how conjunctive phrases with more than two conjuncts are expressed in English, a non-terminal 'conj_nt' for conjoined 'nt's could be simply expressed in a context-free grammar by

```
conj_nt --> nt, rest_conj_nt.

rest_conj_nt --> [].
rest_conj_nt --> conj, conj_nt.
```

But, as I have noted above, this formulation would not in general work for an XG.

However, looking again at the translation of XGs into definite clauses, it is quite simple to express precisely what is required:

```
conj_nt(S0,S,X0,X) :- nt(S0,S1,X0,X), rest_conj_nt(S1,S,X0,X).

rest_conj_nt(S,S,X0,X).
rest_conj_nt(S0,S,X0,X) :- conj(S0,S1), conj_nt(S1,S,X0,X).
```

In other words, the extraposition list arguments X0 and X are the same for all the conjoined 'nt's, and therefore if something is extraposed from one of them, it will be extraposed from all the others.

Following these observations, we could now try to add to the XG formalism some notational device that expands exactly into clauses as those above. However, non-terminal arguments will in general cause a new problem. In the clauses, the string arguments are used "serially", expressing the concatenation of phrases, whereas the extraposition arguments are used "in parallel" to express deletion. The same distinction will apply to non-terminal arguments: some, such as some features, will be shared by conjuncts, but others, such as those that represent the analyses of phrases, need to be treated in the same way as the string arguments. For example, in an analysis of the conjoined verbs in

John $[_{v_{trans}}[_{v_{trans}}$ stole]and $[_{v_{trans}}$ ate]] some tarts (6.1)

the two conjuncts will share the feature value "trans", specifying that the conjuncts must be transitive, but the representation of the conjunction must contain two separate parts for the two conjuncts.

One possible way of expressing the two kinds of argument combination in conjunctions is to introduce conjunction schemata in

the XG formalism. In a conjunction schema we abstract out, in a way similar to lambda abstraction, those arguments of the conjoined non-terminal that must have different values on different conjuncts. The abstracted variables are associated to argument places of a predicate that puts together their values from the different conjuncts. To describe the conjoined verbs above, we would have a non-terminal 'v' with the structure argument V and the verb argument feature A. The conjunction rule would be

```
conj_v(V,A) --> conj([V], v(V:1,A), and(V:1,V:2,V))          (6.2)
```

where 'and' is defined by

```
and(V1,V2,and(V1,V2)) --> [and].                             (6.3)
```

In the schema, V is the abstracted variable, V:1 represents its value for the "left" branch of the conjunction tree, and V:2 its value for the "right" branch. Using (6.2) to analyse (6.1), the variables would take values as follows:

```
A   = trans
V:1 = "stole"
V:2 = "ate"
V   = and("stole","ate")
```

where "x" stands for the structure of x.

The meaning of (6.2) would be given in definite clauses in a way similar to 'conj_nt' above:

```
conj_v(V,A,S0,S,X0,X) :-
   v(V0,A,S0,S1,X0,X),
   rest_conj_v(V0,A,V,S1,S,X0,X).

rest_conj_v(V,A,V,S,S,X0,X).
rest_conj_v(V0,A,V,S0,S,X0,X) :-
   and(V0,V1,V,S0,S1,X0,X0),                                 (6.4)
   conj_v(V1,A,S1,S,X0,X).
```

Goal (6.4) has the correct argument layout to fit the translation into a definite clause of the XG rule (6.3), and the use of the same extraposition list for its two extraposition arguments forbids extraposition from inside the conjunction "and", which is as it should be. Note that, although in the example above extraposition would not be likely to be involved, it would be in a similar example for a sentence non-terminal 's':

```
conj_s(S) --> conj([S], s(S:1), and(S:1,S:2,S)).
```

Notice that conjunction schemata are not "higher-order" in the logical sense, any more than lambda abstraction over predicate arguments is "higher-order" with respect to definite clause logic [Warren 81b].

Introducing conjunction schemata has the disadvantage of making the formalism more specialised but still requiring an explicit statement of those non-terminals that may be conjoined. An alternative but related approach is to use metarules [Gazdar 79b] to describe implicitly the conjunction rules in terms of an initial set of rules without conjunction. This leads to the whole question of the relations between logic-based grammar formalisms and generalised phrase structure grammars, both from the formal point of view and from the point of view of parsing.

6.2 Parsing Method

In Section 5.2.1, I argued informally that the theoretical superiority of tabular parsing methods and the related shift-reduce look-ahead parsers[29] over top-down backtrack parsers did not seem to be attainable for the Chat-80 grammar. There are other reasons, however, for using tabular parsers. I will just mention two that deserve further research.

First, in general tabular and LR-type context-free parsers are better at error detection and recovery than top-down parsers [Aho and Ullman 72, Anderson and Backhouse 81]. Practical natural language front-ends need proper error detection, and certainly the current Chat-80 parser has no such facilities.

Second, it is possible that tabular algorithms, which store partial analyses, may give a method of dealing with incomplete input, such as elliptic phrases, but using a grammar that only describes complete phrases.

Both of these questions presuppose that concepts from context-free parsing theory can be extended to DCGs, as suggested by work on the relation between parsing algorithms and definite clause theorem provers [Warren 75, Uehara and Toyoda 81].

[29] LR parsers in the determinate case.

6.3 Modularity and Extensibility

One of the obstacles that the present organisation of Chat-80 poses to extending it or moving it to another domain is the organisation of the dictionary. The separation of information about a word into syntactic clauses and semantic/pragmatic clauses (templates) is convenient from the programming point of view, but not from the point of view of the dictionary builder.

The predicates and arguments of dictionary clauses were also chosen to simplify the grammar and interpretation rules. In particular, to specify or change the semantic type denotations associated to slots in a template requires constant reference to an implicit type tree, whose relation with the common nouns that correspond to its nodes is also implicit.

The template features that govern the interpretation of nested plural noun phrases as indexed sets are clearly related to a notion of a word being "functionally dependent" on some of its arguments, or the arguments representing "necessary" as opposed to "contingent" attributes of the word [Fillmore 68]. Again, those notions are not used explicitly in the templates.

One way to have a dictionary organised in terms of higher-level linguistic and semantic concepts but still have the relative simplicity of the actual interpretation rules would be to design a "dictionary compiler" that would translate descriptions of a type hierarchy and of syntactic and semantic properties of words (such as surface cases and semantic roles of arguments) into the lower-level

features used in the existing rules. Of course, a deeper higher-level description in the dictionary might lead to a different set of low-level features.

The above discussion of semantic roles and low-level features leads to a more general question, that of the organisation of interpretation rules. Interpretation rules may be directly attached to syntactic rules [Montague 73, Dahl 77, Robinson 82], or to parse tree nodes, as in the present work and in LUNAR [Woods et al. 72, Woods 77a]. In both methods, the interpretation rules contain implicitly a notion of the semantic roles that relate phrases in a sentence. As McCord has shown [McCord 80a], it is possible to bring out those semantic roles and make them central to the organisation of the interpretation rules. In a system that uses an explicit surface parse tree (not the case in McCord's), the interpretation rules may then become a small set of general rules that traverse the recursive structure of the parse tree collecting related sub-trees, together with a table that describes the semantic roles associated to each phrase class (non-terminal). This would decouple the tree-traversing aspect of interpretation rules from the role determination aspect. Roles that the rules in Chat-80 do not cover at the moment, such as those in adverbial modification, might then be easier to treat [McCord 81].

6.4 Scoping

My treatment of quantification in natural language, in common with much of the work in this area, uses the classical notions of quantifier and bound variable to express the meaning of determiners in natural language. However, there are problems with this use of classical quantification. The scopes of quantifiers impose a tree structure on the formulae that represent the meaning of sentences. This tree structure determines the dependencies between the quantifiers: if some open sub-formula of a semantic interpretation contains variables bound by two different quantifiers, one of the quantifiers will be in the scope of the other. Hintikka [Hintikka 74] has given arguments against these constraints on dependencies between quantifiers. I will not report Hintikka's views here but will only note that he gives numerous examples where the dependencies imposed on the translation of a sentence by classical quantification do not seem to correspond to dependencies in the original sentence. He then proposes new notions of quantifier and of the underlying semantics, which he shows to be outside classical predicate calculus.

Less radically, we may try to cure some of the problems caused by unwanted quantifier dependencies within the framework of classical logic. Vanlehn [Vanlehn 78] observes that some unwanted dependencies can be avoided if existential quantifiers are not used in a translation, their position being taken by Skolem functions of universally quantified variables. He proposes this technique to avoid having to give relative scopes to quantifiers when scope judgements for the corresponding determiners are uncertain. However, the differences between using Skolem functions and using existential

quantifiers only come into being for sentences whose classical translation contains several existential quantifiers, all in the scope of several universal quantifiers.

I will now mention two other areas where the representation of quantifications has important consequences.

The first area is that of the pragmatic importance of properties of predicates in determining the scope of quantifications. As I pointed out in Section 4.3, quantifier precedence rules are only an expedient to get a practical approximation of human judgements that all evidence suggests are based on a deeper pragmatic classification of dependencies between the arguments of words. Moving from a quantifier-based approach to a argument-dependency approach might get us nearer those deeper criteria.

The other area is the use of binding (in the sense of the binding between a relative pronoun and a trace) to help derive logical forms for some restricted cases of anaphoric reference. Although this approach is superficially attractive, and has been proposed in the context of Montague grammar [Montague 73, Creswell 73], it fails if it is employed in the straightforward way [Pereira 78, Cooper 79]. It is easy to find examples of the problem. The sentence

Every man who owns a car washes it.

cannot be given a reasonable translation that logically binds the translation of "a car" to "it"

```
all(M, man(M) &                                          (6.5)
       exists(C, car(C) & owns(M,C) ) =>
       washes(M,?) )
```

because the occurrence of the pronoun is outside the scope of the quantification for its antecedent. However, there is a not so direct but straightforward translation of the sentence:

```
all(M,
      all(C, man(M) & car(C) & owns(M,C) => washes(M,C) ) )
```

This translation could be obtained from (6.5) by moving the inner quantifier out using the obvious logical equivalence and only then looking at the binding of references. Bindings between arguments of predicates thus cut across scopes and they should be seen not as represented by scopes, but as determining scopes. That is, the notion of binding precedes the notion of quantification, and quantification is just a convenient mechanism to render this notion of binding that is nearer natural language. In an implementation of this approach in the Chat-80 framework, pronoun bindings would be represented in the Quant tree, and would then influence the way in which the Quant tree is rewritten into scopes and quantifications.

6.5 Summary

I have discussed four problem areas for further work.

The first and best defined is the treatment of conjunction in XGs. XGs as presently formulated cannot deal cleanly with the phenomenon of across the board deletion in conjunctions, and I suggest an extension to the formalism that will solve this problem. The exact form of the extension and its interaction with the rest of the formalism need however further investigation.

The second area is that of parsing algorithms for XGs. Although I

have shown in the previous chapter that the simple top-down backtrack algorithm provided by Prolog is much better that might be expected, questions other than those of efficiency, such as error recovery and the treatment of incomplete input, might lead to the choice of other parsing algorithms. The general connection between parsing algorithms and theorem proving strategies also suggests that only a very small class of parsing algorithms for logic-based grammars has yet been explored.

The third area is that of the relation between dictionary organisation and the formulation of semantic interpretation rules. What needs to be done here is to formulate a system of semantic roles to be used in the dictionary and the interpretation rules, in a way similar to McCord's, but still allowing for non-compositional operators in building semantic interpretations.

Finally, extending the present work to deal with some limited form of anaphoric reference will need a clarification of the relationship between quantification and referents. The present mechanisms use classical quantifiers as the exclusive representation of bindings, but the treatment of anaphora may well require a more flexible notion of binding, possibly by means of Skolem functions.

Chapter 7

Conclusion

In this work, I set out to develop the idea of using a system of logic with good computational properties, namely definite clauses, both as a descriptive language for the analysis of natural language sentences and as the language in which to express the results of the analysis.

The two fundamental motivations have been to improve on the earlier work of Colmerauer and others, and show how the use of definite clause logic reduces radically the gap between theoretical notions and the actual programs that realise them, allowing power to coexist with clarity. The program that has been written, Chat-80, is a joint work where the syntactic, semantic and pragmatic notions presented above are combined with a meaning representation language, definite closed-world clauses, to which powerful query optimisation methods can be applied. Chat-80 compares very favourably in speed with similar programs, even though the grammar and interpretation rules have been written with clarity, rather than speed, in mind. The clauses that define application-dependent words are clearly separated from the clauses that deal with general-purpose words and constructions.

I first identified certain problems in earlier work: in the grammar formalism and grammar organisation, in the concepts and techniques used to translate from sentences to logical form, and in the choice of translation for various words and constructions.

On the grammar side, I have introduced a grammar formalism, extraposition grammars, which simplifies the description of the syntactic notions of left extraposition and its associated constraints, but which still maps very closely into definite clauses. Then I showed how to use the XG formalism to write a description of the surface syntax of a fragment of English. The major basic construction left out of that grammar is general conjunction, whose description would require changes to the formalism.

Using a syntactic grammar raises the problem of what to do with apparent ambiguities that can be resolved through semantic or pragmatic information. A major instance of this difficulty, the attachment of post-modifiers, can be cured in an XG by augmenting the XG with tests that rule out all but one of the set of analyses differing only on post-modifier attachment. This technique is seen to be related to theories of attachment preference and deterministic parsing.

On the question of parsing with XGs, the experience with Chat-80 shows that top-down backtrack parsing is much more practical than has been previously suggested. However, the problems of error detection and incomplete input would possibly benefit from different parsing algorithms. The practical use of those algorithms may well require some precompilation of the grammars, based on suitable

generalisations to DCGs and XGs of grammar analysis algorithms for context-free grammars.

Whereas grammar rules have been fairly directly expressed as XG rules, the interpretation rules that relate the surface syntax of a sentence to its proposed logical form could not be so easily expressed as definite clauses. The interpretation rules are therefore described informally, and a possible implementation of those rules as a definite clause program is given separately.

Interpretation rules fall into two groups, the slot filling rules that decide how the fragments of logical form translating individual words get their arguments and qualifications, and the scoping rules that determine the scopes of the various logical operators in the translation of a sentence. To connect those two groups of rules, I introduce the intermediate representation by quantifier trees.

The practical results achieved with Chat-80 point to some unorthodox conclusions: that efficiency and clarity need not be opposed when writing grammars which can be used in parsers; that simple top-down backtrack parsing is hard to beat in efficiency, if the grammar is written without irrelevant ambiguities; that a separate syntactic component does not necessarily lead to combinatorial explosion.

The present work has obvious technical limitations, and others that are of a deeper nature. On the technical side, I consider that the concepts used could support mild extensions to the grammar and interpretation rules, such as those needed to cope with anaphoric

expressions referring to individuals, or to cover a wider range of conjunctive phrases. Of the deeper problems, two stand out as particularly interesting. First, is that of what kinds of semantic abstractions would be needed to bring the translation rules nearer their implementation as definite clauses, as the syntactic grammar is today. Second, what properties of the predicates translating words could be used to improve the scope rules, and tie in with the slot filling process and semantic types in an explicit logical expression of properties of words.

Acknowledgements

This work is part of a research project with David Warren. With constant discussion and patient criticism, he greatly helped me in choosing sensible questions and in shaping the answers. He also helped to keep me from straying into theoretical minefields, and to concentrate in writing my share of the results of our joint work. Without his work in making Prolog a practical programming language, it is unlikely that I would have ever moved in this direction.

If this work is can be read by anyone other than myself, I owe it in no small measure to the thorough criticisms of Dr. Henry Thompson. Shape and style also benefited from the comments of my supervisor, Dr. J. Howe.

While a post-graduate student in the Department of Artificial Intelligence, I was funded by a British Council fellowship, and the computing resources for the development of the Chat-80 program where made available by the Science and Engineering Research Council. I owe this funding to the support of my supervisors, first Professor B. Meltzer and then Dr. J. Howe.

I finalised the work while employed on SERC research grants at CAAD Studies in the Department of Architecture. I owe to CAAD's Director Aart Bijl and my colleagues, the support, not only material, that

made it possible to complete what was just a heap of programs and half-written ideas when I started work at CAAD. In particular, Jim Nash's STAG graphics editor made drawing possible to a totally incapable hand, and everyone tolerated the strange working hours caused by my writing.

Many other people contributed, directly or indirectly, to the development of these ideas. I thank in particular Alain Colmerauer and Robert Kowalski, who started it all and gave plenty of their time in discussions, Luis Pereira, who first told me about logic programming and many other things, and Veronica Dahl, whose work stimulated and developed my interest in natural language analysis.

References

[Aho and Johnson 74]
Aho, A. V. and Johnson, S. C.
LR Parsing.
Computing Surveys 6(2):99-124, 1974.

[Aho and Ullman 72]
Aho, A. V. and Ullman, J. D.
The Theory of Parsing, Translation and Compiling.
Prentice-Hall, 1972.

[Anderson and Backhouse 81]
Anderson, S. O., Backhouse, R. C.
Locally Least-Cost Error Recovery in Earley's Algorithm.
ACM Trans. Prog. Lang. Sys. 3(3):318-347, July, 1981.

[Apt and van Emden 80]
Apt.
Contributions to the Theory of Logic Programming.
Research Report CS-80-12, Dept. of Computer Science, University of Waterloo, Ontario, Canada, 1980.

[Bartsch and Vennemann 72]
Bartsch, R. and Vennemann, T.
Semantic Structures - A Study in the Relation Between Semantics and Syntax.
Athenaum Verlag, Frankfurt, 1972.

[Bennett 76]
Bennett, M.
A Variation and Extension of a Montague Fragment of English.
In B. Partee, editor, Montague Grammar . Academic Press, 1976.

[Bowen and Kowalski 81]
Bowen, K. A. and Kowalski, R. A.
Almagamating Language and Metalanguage in Logic Programming.
Technical Report, School of Computer and Information Science, Syracuse University, 1981.

[Chomsky 75]
Chomsky, N.
Reflections on Language.
Pantheon, 1975.

[Church 80]
Church, K. A.
On Memory Limitations in Natural Language Processing.
Master's thesis, M.I.T., 1980.
Published as Report MIT/LCS/TR-245.

[Clark 78]
Clark, K. L.
Negation as Failure.
In H. Gallaire and J. Minker, editor, Logic and Data Bases . Plenum Press, New York, 1978.

[Colmerauer 70]
Colmerauer, A.
Les Systèmes-Q ou un Formalisme pour Analyser et Synthétiser des Phrases sur Ordinateur.
Internal Publication 43, Département d'Informatique, Université de Montreal, Canada, 1970.

[Colmerauer 78]
Colmerauer, A.
Metamorphosis Grammars.
In L .Bolc, editor, Natural Language Communication with Computers . Springer-Verlag, 1978.
First appeared as an internal report, 'Les Grammaires de Metamorphose', in November 1975.

[Colmerauer 79a]
Colmerauer, A.
Les Bases Theoriques de Prolog.
Technical Report, Groupe d'Intelligence Artificielle, U. E. R. de Luminy, Universite d'Aix-Marseille II, 1979.

[Colmerauer 79b]
Colmerauer, A.
Un Sous-Ensemble Interessant du Français.
RAIRO 13(4):309-336, 1979.
Presented under the title "An Interesting Natural Language Subset" at the Workshop on Logic and Databases, Toulouse, 1977.

[Cooper 79]
Cooper, R.
Variable Binding and Relative Clauses.
In F. Guenthner and S. J. Schmidt, editor, Formal Semantics and Pragmatics for Natural Languages D. Reidel, 1979.

[Creswell 73]
Creswell, M.
Logics and Languages.
Metween, 1973.

[Curry and Feys 68]
Curry, H. B. and Feys, R.
Combinatory Logic.
North-Holland, Amsterdam, 1968.

[Dahl 77]
Dahl, V.
Un Systeme Deductif d'Interrogation de Banques de Donnees en Espagnol.
Thèse de 3ème Cycle, Université d'Aix-Marseille II, Faculté des Sciences de Luminy, , 1977.

[Earley 70]
Earley, J.
An Efficient Context-Free Parsing Algorithm.
Communications of the ACM 13(2):94-102, 1970.

[Fillmore 68]
Fillmore, C. J.
The Case for Case.
In E. Bach, editor, Universal Grammar . Holt, Rinehart and Winston, 1968.

[Fodor and Frazier 80]
Fodor, J. D. and Frazier, L.
Is the Human Sentence Parsing Mechanism an ATN?
Cognition 8:417-459, 1980.

[Gazdar 79a]
Gazdar, G.
Constituent Structures.
Cognitive Studies Programme, School of Social Sciences, University of Sussex, January, 1979.

[Gazdar 79b]
Gazdar, G.
English as a Context-Free Language.
Cognitive Studies Programme, School of Social Sciences, University of Sussex, April, 1979.

[Hintikka 74]
Hintikka, J.
Quantifiers vs. Quantification Theory.
Linguistic Inquiry 5(2):153-177, 1974.

[Hughes and Creswell 68]
Hughes, G. E. and Creswell, M. J.
An Introduction to Modal Logic.
Methuen, London, 1968.

[Jackendoff 72]
Jackendoff, R. S.
Semantic Interpretation in Generative Grammar.
MIT Press, 1972.

[Joshi 77]
Joshi, A. K. and Levy, L. S.
Constraints on Structural Descriptions: Local Transformations.
SIAM Journal of Computing 6(2):272-284, June, 1977.

[Kay 80]
Kay, M.
Algorithm Schemata and Data Structures in Syntactic Processing.
Technical Report, XEROX Palo Alto Research Center, 1980.
A version will appear in the proceedings of the Nobel Symposium on Text Processing, Gothenburg, 1980.

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

[Kowalski 80]
Kowalski, R. A.
Logic for Problem Solving.
North Holland, 1980.

[Lehmann 78]
Lehmann, H.
Interpretation of Natural Language in an Information System.
IBM J. Res. Develop. 22(5):560-572, September, 1978.

[Lewis 72]
Lewis, D.
General Semantics.
In D. Davidson and G. Harman, editor, Semantics of Natural Language . D. Reidel, 1972.

[Marcotty et al. 76]
Marcotty, M., Ledgard, H. F. and Bochman, G. V.
A Sampler of Formal Definitions.
Computing Surveys 8(2):191-276, 1976.

[McCord 80]
McCord, M. C.
Slot Grammars.
AJCL 6(1):255-286, 1980.

[McCord 81]
McCord M. C.
Focalizers, the Scoping Problem, and Semantic Interpretation Rules in Logic Grammars.
In Procs. of the Workshop on Logic Programming for Intelligent Systems. Logicon Inc., 1981.

[McCord 82]
McCord, M. C.
Using Slots and Modifiers in Logic Grammars for Natural Language.
Artificial Intelligence 18(3):327-367, 1982.

[McDermott and Doyle 80]
McDermott, D. and Doyle, J.
Non-monotonic Logic I.
Artificial Intelligence 13:41-72, 1980.

[Mellish 81]
Mellish, C.S.
Coping with Uncertainty: Noun Phrase Interpretation and Early Semantic Analysis.
PhD thesis, Dept. of Artificial Intelligence, University of Edinburgh, 1981.

[Montague 70]
Montague, R.
English as a Formal Language.
In R. Thomason, editor, Formal Philosphy, Selected Papers of Richard Montague . Yale University Press, 1970.

[Montague 73]
Montague, R.
The Proper Treatment of Quantification in Ordinary English.
In R. Thomason, editor, Formal Philosphy, Selected Papers of Richard Montague . Yale University Press, 1973.

[Moore 81]
Moore, R. C.
Problems in Logical Form.
In Proceedings of the 19th Annual Meeting of the Association for Computational Linguistics. ACL, Stanford, California, 1981.

[Nash-Webber and Reiter 77]
Nash-Webber, B. and Reiter, R.
Anaphora and Logical Form: on Formal Meaning Representations for Natural Language.
In Proceedings of the 5th IJCAI, pages 121-131. MIT, 1977.

[Newell 82]
Newell, A.
The Knowledge Level.
Artificial Intelligence 18(1):87-127, 1982.

[Pasero 73]
Pasero, R.
Representation du Français en Logique du Premier Ordre, en Vue de Dialoguer avec un Ordinateur.
Thèse de 3ème Cycle, Université d'Aix-Marseille II, Faculté des Sciences de Luminy, , 1973.

[Pereira 78]
Pereira, F. C. N.
A Pedestrian Look at Montague.
Working Paper 47, Dept. of Artificial Intelligence, University of Edinburgh, 1978.

[Pereira 81a]
Pereira, F. C. N.
Ambiguity in Logic Grammars.
In Procs. of the Workshop on Logic Programming for Intelligent Systems. Logicon Inc., 1981.

[Pereira 81b]
Pereira, F.
Extraposition Grammars.
AJCL 7(4):243-256, 1981.

[Pereira and Warren 80]
Pereira, F. and Warren, D. H. D.
Definite Clause Grammars for Language Analysis - A Survey of the Formalism and a Comparison with Augmented Transition Networks.
Artificial Intelligence 13:231-278, 1980.

[Pereira et al. 78]
Pereira, L. M., Pereira, F. and Warren, D. H. D.
User's Guide to DECsystem-10 Prolog.
Technical Report, Dept. of Artificial Intelligence, University of Edinburgh, 1978.

[Pique 81]
Pique, J. F.
Sur un Modèle Logique du Language Naturel et son Utilisation pour L'Interrogation des Banques de Donées.
Thèse de 3ème Cycle, Université d'Aix-Marseille II, Faculté des Sciences de Luminy, , 1981.

[Reiter 80]
Reiter, R.
On Reasoning by Default.
Artificial Intelligence 13:81-132, 1980.

[Robinson 65]
Robinson, J. A.
A Machine-Oriented Logic Based on the Resolution Principle.
Journal of the ACM 12:23-44, 1965.

[Robinson 82]
Robinson, J. J.
Diagram: a Grammar for Dialogues.
Communications of the ACM 25(1):27-47, 1982.

[Rodman 76]
Rodman, R.
Scope Phenomena, "Movement Transformations", and Relative Clauses.
In B. Partee, editor, Montague Grammar . Academic Press, 1976.

[Ross 74]
Ross, J. R.
Excerpts from 'Constraints on Variables in Syntax'.
In G. Harman, editor, On Noam Chomsky: Critical Essays Anchor Books, 1974.

[Roussel 75]
Roussel, P.
Prolog : Manuel de Reference et Utilisation.
Technical Report, Groupe d'Intelligence Artificielle, U. E. R. de Luminy, Universite d'Aix-Marseille II, 1975.

[Sabatier 80]
Sabatier, P.
Dialogues en Français avec un Ordinateur.
Thèse de 3ème Cycle, Université d'Aix-Marseille II, Faculté des Sciences de Luminy, , 1980.

[Schank 73]
Schank, R.
The Conceptual Analysis of Natural Language.
In R. Rustin, editor, Natural Language Processing Algorithmics Press, New York, 1973.

[Schank 75]
Schank, R.
Conceptual Information Processing.
North-Holland, 1975.

[Seuren 69]
Seuren, P. A. M.
Operators and Nucleus.
Cambridge University Press, 1969.

[Shaumyan 77]
Shaumyan, S. K.
Applicational Grammar.
Edinburgh University Press, 1977.

[Sidner et al. 81]
Sidner, C. L, Bates. M., Bobrow, R. J., Brachman, R. J, Cohen, P. R., Israel, D. J., Webber, B. L. and Woods, W. A.
Research in Knowledge Representation for Natural Language Understanding.
Report 4785, Bolt Beranek and Newman Inc., 1981.

[Siegel and Bossu 81]
Siegel, P. and Bossu, G.
La Saturation au Secours de la Non-monotonie.
Thèse de 3ème Cycle, Université d'Aix-Marseille II, Faculté des Sciences de Luminy, , 1981.

[Stockwell et al. 73]
Stockwell, R. P., Schachter, P. and Partee, B. H.
The Major Syntactic Structures of English.
Holt, Rinehart & Winston, 1973.

[Thomason 76]
Thomason, R.
Some Extensions of Montague Grammar.
In B. Partee, editor, Montague Grammar . Academic Press, 1976.

[Uehara and Toyoda 81]
Uehara, K. and Toyoda, J.
A Pattern Matching Directed Parser: PAMPS.
1981.
Presented at the ACL/LSA Meeting, December 1981.

[van Emden and Kowalski 76]
van Emden, M. H. and Kowalski, R. A.
The Semantics of Predicate Logic as a Programmir Language.
Journal of the ACM 23(4):733-742, 1976.

[van Wijngaarden 75]
van Wijngaarden, A. et al.
Revised Report on the Algorithmic Language Algol-68.
Acta Informatica 5:1-236, 1975.

[Vanlehn 78]
Vanlehn, K. A.
Determining the Scope of English Quantifiers.
Master's thesis, M.I.T., Jun, 1978.
Published as Report AI-TR-483.

[Wanner 80]
Wanner, E.
The ATN and the Sausage Machine: Which One Is Baloney?
Cognition 8:209-225, 1980.

[Warren 75]
Warren, D. H. D.
Earley Deduction.
1975.
Unpublished note.

[Warren 81a]
Warren, D. H. D.
Efficient Processing of Interactive Relational Database Queries Expressed in Logic.
In 7th International Conference on Very Large Data Bases. IEEE, 1981.

[Warren 81b]
Warren, D. H. D.
Higher-Order Extensions to Prolog - Are They Needed?.
Research Paper 154, Dept. of Artificial Intelligence, University of Edinburgh, 1981.

[Warren and Pereira 81]
Warren, D. H. D. and Pereira F. C. N.
An Efficient Easily Adaptable System for Interpreting Natural Language Queries.
Research Paper 155, Dept. of Artificial Intelligence, University of Edinburgh, 1981.

[Warren et al. 77]
Warren, D. H. D., Pereira, L. M. and Pereira, F.
Prolog - The Language and its Implementation Compared with Lisp.
In SIGPLAN/SIGART Newsletter. ACM Symposium on A.I. and Programming Languages, August, 1977.

[Wilks 72]
Wilks, Y.
Grammar, Meaning and the Machine Analysis of Language.
Routledge & Kegan Paul, London, 1972.

[Woods 70]
Woods, W. A.
Transition Network Grammars for Natural Language Analysis.
Communications of the ACM 13:591-606, October, 1970.

[Woods 73]
Woods, W. A.
An Experimental Parsing System for Transition Network Grammars.
In R. Rustin, editor, Natural Language Processing . Algorithmics Press, New York, 1973.

[Woods 77a]
Woods, W. A.
Lunar Rocks in Natural English: Explorations in Natural Language Question Answering.
In A. Zampoli, editor, Linguistic Structures Processing . North-Holland, 1977.

[Woods 77b]
Woods, W. A.
Semantics and Quantification in Natural Language Question Answering.
Report 3687, Bolt Beranek and Newman Inc., November, 1977.

[Woods et al. 72]
Woods, W. A., Kaplan, R. M. and Nash-Webber, B.
The Lunar Sciences Natural Language Information System: Final Report.
Report 3438, Bolt Beranek and Newman Inc., June, 1972.

[Zoeppritz 81]
Zoeppritz, M.
The Meaning of OF and HAVE in the USL System.
AJCL 7(2):109-119, 1981.

Appendix A

Translating XGs

The following Prolog program (for the DEC-10 Prolog system) defines a predicate 'grammar(File)' which translates and stores the XG rules contained in File.

```
:- op(1001,xfy,(...)).          % Definition of the grammar rule
:- op(1200,xfx,(-->)).          % operators

grammar(File) :-                % Processes the XG in File
   seeing(Old),
   see(File),
   consume,
   seen,
   see(Old).

consume :-                      % Loop until end_of_file
   repeat,
      read(X),
    ( X=end_of_file, !;
      process(X),
         fail ).

process((L-->R)) :- !,          % Process a grammar rule
   expandlhs(L,S0,S,H0,H,P),
   expandrhs(R,S0,S,H0,H,Q),
   assertz((P :- Q)), !.
process(( :- G)) :- !,          % Execute a command
   G.
process((P :- Q)) :-            % Store a normal clause
   assertz((P :- Q)).
process(P) :-                   % Store a unit clause
   assertz(P).

% Translate an XG rule

expandlhs(T,S0,S,H0,H1,Q) :-    % Translate the left-hand side
   flatten(T,[P|L],[]),
   front(L,H1,H),
   tag(P,S0,S,H0,H,Q).
```

```
flatten((X...Y),L0,L) :- !,
   flatten(X,L0,[gap|L1]),
   flatten(Y,L1,L).
flatten((X,Y),L0,L) :- !,
   flatten(X,L0,[nogap|L1]),
   flatten(Y,L1,L).
flatten(X,[X|L],L).

front([],H,H).
front([K,X|L],H0,H) :-
   case(X,K,H1,H),
   front(L,H0,H1).

case([T|Ts],K,H0,x(K,terminal,T,H)) :- !,
   unwind(Ts,H0,H).
case(Nt,K,H,x(K,nonterminal,Nt,H)) :-
   virtual_rule(Nt).

virtual_rule(Nt) :-              % Create the clause
   functor(Nt,F,N),              %  Nt(S,S,X0,X) :- virtual(Nt,X0,X)
   functor(Y,F,N),               % for extraposed symbol Nt
   tag(Y,S,S,Hx,Hy,P),
 ( clause(P,virtual(_,_,_),_), !;
   asserta((P :- virtual(Y,Hx,Hy))) ).

expandrhs((X1,X2),S0,S,H0,H,Y) :- !, % Translate the right-hand side
   expandrhs(X1,S0,S1,H0,H1,Y1),
   expandrhs(X2,S1,S,H1,H,Y2),
   and(Y1,Y2,Y).
expandrhs((X1;X2),S0,S,H0,H,(Y1;Y2)) :- !,
   expandor(X1,S0,S,H0,H,Y1),
   expandor(X2,S0,S,H0,H,Y2).
expandrhs({X},S,S,H,H,X) :- !.
expandrhs(L,S0,S,H0,H,G) :- islist(L), !,
   expandlist(L,S0,S,H0,H,G).
expandrhs(X,S0,S,H0,H,Y) :-
   tag(X,S0,S,H0,H,Y).

expandor(X,S0,S,H0,H,Y) :-
   expandrhs(X,S0a,S,H0a,H,Ya),
 ( S\==S0a, !, S0=S0a, Yb=Ya; and(S0=S0a,Ya,Yb) ),
 ( H\==H0a, !, H0=H0a, Y=Yb; and(H0=H0a,Yb,Y) ).

expandlist([],S,S,H,H,true).
expandlist([X],S0,S,H0,H,terminal(X,S0,S,H0,H) ) :- !.
expandlist([X|L],S0,S,H0,H,(terminal(X,S0,S1,H0,H1),Y)) :-
   expandlist(L,S1,S,H1,H,Y).

tag(P,A1,A2,A3,A4,Q) :-
   P=..[F|Args0],
   conc(Args0,[A1,A2,A3,A4],Args),
   Q=..[F|Args].
```

```
and(true,P,P) :- !.
and(P,true,P) :- !.
and(P,Q,(P,Q)).

islist([_|_]).
islist([]).

unwind([],H,H) :- !.
unwind([T|Ts],H0,x(nogap,terminal,T,H)) :-
   unwind(Ts,H0,H).

conc([],L,L) :- !.
conc([X|L1],L2,[X|L3]) :-
   conc(L1,L2,L3).
```

Appendix B

Definite Clauses for the Grammar Used in Figure 3-10

```
sentence(S0,S,X0,X) :-
   noun_phrase(S0,S1,X0,X1),
   verb_phrase(S1,S,X1,X).

noun_phrase(S0,S,X0,X) :- proper_noun(S0,S,X0,X).
noun_phrase(S0,S,X0,X) :-
   determiner(S0,S1,X0,X1),
   noun(S1,S2,X1,X2),
   relative(S2,S,X2,X).
noun_phrase(S0,S,X0,X) :-
   determiner(S0,S1,X0,X1),
   noun(S1,S2,X1,X2),
   prep_phrase(S2,S,X2,X).
noun_phrase(S0,S,X0,X) :- trace(S0,S,X0,X).

verb_phrase(S0,S,X0,X) :-
   verb(S0,S1,X0,X1),
   noun_phrase(S1,S,X1,X).
verb_phrase(S0,S,X0,X) :- verb(S0,S,X0,X).

relative(S0,S0,X,X).
relative(S0,S,X0,X) :-
   open(S0,S1,X0,X1),
   rel_marker(S1,S2,X1,X2),
   sentence(S2,S3,X2,X3),
   close(S3,S,X3,X).

trace(S0,S0,X0,X) :- virtual(trace,X0,X).

rel_marker(S0,S,X0,x(gap,nonterminal,trace,X)) :-
   rel_pronoun(S0,S,X0,X).

prep_phrase(S0,S,X0,X) :-
   preposition(S0,S1,X0,X1),
   noun_phrase(S1,S,X1,X).

open(S0,S0,X,x(gap,nonterminal,close,X)).

close(S0,S0,X0,X) :- virtual(close,X0,X).
```

Appendix C

The Chat-80 Grammar

The symbol "sentence(tree)" is the start symbol of the grammar, where tree is a tree representation of the analysis obtained.

```
/* Sentences */

sentence(S) --> declarative(S), terminator(.) .
sentence(S) --> wh_question(S), terminator(?) .
sentence(S) --> yn_question(S), terminator(?) .
sentence(S) --> imperative(S), terminator(!) .

/* Declarative sentence */

declarative(decl(S)) --> s(S,_).

/* Wh-questions */

wh_question(whq(X,S)) -->
   variable_q(X,_,QCase,NPCase),
   question(QCase,NPCase,S).

variable_q(X,Agmt,QCase,NPCase) ...
      np(NP,Agmt,NPCase,_,_,Set,Mask) -->
   whq(X,Agmt,NP,QCase),
   {trace(Set,Mask)}.
variable_q(X,Agmt,compl,CCase) ...
 pp(pp(Prep,NP),compl,Set,Mask) -->
   prep(Prep),
   whq(X,Agmt,NP,_),
   {trace(Set,Mask), compl_case(CCase)}.
variable_q(X,Agmt,compl,VCase) ...
      adv_phrase(pp(Prep,
                    np(Agmt,
                     np_head(
                      int_det(X),[],Noun),[])),
                  Set,Mask) -->
   context_pron(Prep,Noun),
   {trace(Set,Mask), verb_case(VCase)}.
variable_q(X,_,compl,VCase) ...
      pred(adj,value(Adj,wh(X)),Mask) -->
   [how],
   adj(quant,Adj),
```

```
   {empty(Mask), verb_case(VCase)}.

whq(X,Agmt,NP,undef) -->
   int_det(X,Agmt),
   {s_all(SAll)},
   np(NP,Agmt,_,_,subj,SAll,_).
whq(X,3+No,np(3+No,wh(X),[]),Case) --> int_pron(Case).

int_det(X,3+Agmt) --> whose(X,Agmt).
int_det(X,3+Agmt) --> int_art(X,Agmt).

whose(X,Agmt), np_head0(wh(X),Agmt,proper), gen_marker --> [whose].

question(QCase,NPCase,S) -->
   {subj_question(QCase), role(subj,_,NPCase)},
   s(S,_).
question(QCase,NPCase,S) -->
   fronted_verb(QCase,NPCase),
   s(S,_).

int_art(X,Agmt), det(DX,Agmt,def) --> int_art(X,Agmt,DX).

subj_question(subj).
subj_question(undef).

/* Yes-no questions */

yn_question(q(S)) -->
   fronted_verb(nil,_),
   s(S,_).

fronted_verb(QCase,NPCase) ...
      verb_form(Root,Tense,Agmt,Role), neg(_,Neg) -->
   verb_form(Root,Tense,Agmt,_),
   {verb_type(Root,aux+_),
    role(QCase,Role,NPCase)},
   neg(_,Neg).

/* Imperative sentences */

imperative(imp(S)) -->
   imperative_verb,
   s(S,_).

imperative_verb,
   [you],
   verb_form(Root,imp+fin,2+sin,main) -->
   verb_form(Root,inf,_,_).

/* Basic sentence (actually, declarative sentence) */

s(s(Subj,Verb,Args,Mods),Mask) -->
   subj(Subj,Agmt,Type),
   verb(Verb,Agmt,Type,Voice),
   {empty(Nil), s_all(SAll)},
```

```
   verb_args(Type,Voice,Args,Nil,Mask0),
   {minus(SAll,Mask0,Set), plus(SAll,Mask0,Mask1)},
   verb_mods(Mods,Set,Mask1,Mask).

subj(there,Agmt,_+be) --> [there].
subj(Subj,Agmt,_) -->
   {s_all(SAll), subj_case(Case)},
   np(Subj,Agmt,Case,_,subj,SAll,_).

/* Noun Phrase */

np(np(Agmt,Pronoun,[]),Agmt,NPCase,def,_,Set,Nil) -->
   {is_pp(Set)},
   pers_pron(Pronoun,Agmt,Case),
   {empty(Nil), role(Case,decl,NPCase)}.
np(np(Agmt,Kernel,Mods),Agmt,Case,Def,Role,Set,Mask) -->
   {is_pp(Set)},
   np_head(Kernel,Agmt,Def+Type,PostMods,Mods),
   {np_all(NPAll)},
   np_compls(Type,Agmt,Role,PostMods,NPAll,Mask).
np(part(Det,NP),3+Number,_,indef,Role,Set,Mask) -->
   {is_pp(Set)},
   determiner(Det,Number,indef),
   [of],
   {s_all(SAll), prep_case(Case)},
   np(NP,3+plu,Case,def,Role,SAll,Mask).

np_head(Kernel,Agmt,Type,PostMods,Mods) -->
   np_head0(Kernel0,Agmt0,Type0),
   possessive(Kernel0,Agmt0,Type0,Mods0,Mods0,
              Kernel,Agmt,Type,PostMods,Mods).

np_compls(proper,_,_,[],_,Nil) --> {empty(Nil)}.
np_compls(common,Agmt,Case,Mods,Set0,Mask) -->
   {np_all(NPAll)},
   np_mods(Agmt,Case,Rel,Mods,Set0,Set,NPAll,Mask0),
   relative(Agmt,Rel,Set,Mask0,Mask).

/* Nuclear noun phrase */

np_head0(name(Name),3+sin,def+proper) --> name(Name).
np_head0(np_head(Det,Adjs,Head),3+Number,Def+common) -->
   determiner(Det,Number,Def),
   adjs(Adjs),
   noun(Head,Number).
np_head0(Pronoun,Agmt,def+proper), gen_marker -->
   poss_pron(Pronoun,Agmt).
np_head0(np_head(Det,[],Noun),3+sin,indef+common) -->
   quantifier_pron(Det,Noun).

/* Possessive construction */

possessive(Kernel0,Agmt0,_,[],Mods0,
           Kernel,Agmt,Type,PostMods,Mods) -->
   gen_case,
```

```
   np_head0(Kernel1,Agmt1,Type1),
   possessive(Kernel1,Agmt1,Type1,PostMods1,
              [pp(poss,np(Agmt0,Kernel0,Mods0))|PostMods1],
              Kernel,Agmt,Type,PostMods,Mods).
possessive(Kernel,Agmt,Type,PostMods,Mods,
           Kernel,Agmt,Type,PostMods,Mods) --> [].

gen_case, [the] --> gen_marker.

gen_marker --> [''''], an_s.

an_s --> [s].
an_s --> [].

/* Determiners */

determiner(Det,Number,Def) --> det(Det,Number,Def).
determiner(Det,Number,Def) --> quant_phrase(Det,Number,Def).

quant_phrase(quant(Op,Quant),Number,Def) -->
   quant(Op,Def),
   number(Quant,Number).

quant(Op,indef) -->
   neg_adv(Op0,Op),
   comp_adv(Op0),
   [than].
quant(Op,indef) -->
   [at],
   sup_adv(Adv),
   {sup_op(Adv,Op)}.
quant(the,def) --> [the].
quant(same,indef) --> [].

neg_adv(Adv,not+Adv) --> [not].
neg_adv(Adv,Adv) --> [].

sup_op(least,not+less).
sup_op(most,not+more).

/* Noun phrase modifiers */

np_mods(Agmt,Case,                ),[Mod|Mods],Set0,Set,_,Mask) -->
   np_mod(Agmt,Case,     Set0,Mask0),
   {trace(Trace), plu     ace,Mask0,Mask1), minus(Set0,Mask1,Set1),
    plus(Mask0,Set0,Ma     },
   np_mods(Agmt,Case,Moc     Mods,Set1,Set,Mask2,Mask).
np_mods( ,_,Mods,Mods,Set     ,Mask,Mask) --> [].

np_mod( ,Case,PP,Set,Mask)       pp(PP,Case,Set,Mask).
np_mod(Agmt,Case,WH,Set,Mask     >
   reduced_relative(Agmt,WH,S     ask).

/* Verb modifiers */
```

```
verb_mods([Mod|Mods],Set0,_,Mask) -->
   verb_mod(Mod,Set0,Mask0),
   {trace(Trace), plus(Trace,Mask0,Mask1), minus(Set0,Mask1,Set1),
    plus(Mask0,Set0,Mask2)},
   verb_mods(Mods,Set1,Mask2,Mask).
verb_mods([],_,Mask,Mask) --> [].

verb_mod(Adv,Set,Mask) --> adv_phrase(Adv,Set,Mask).
verb_mod(Adv,Set,Nil) -->
   {is_adv(Set)},
   adverb(Adv),
   {empty(Nil)}.
verb_mod(PP,Set,Mask) --> pp(PP,compl,Set,Mask).

adv_phrase(pp(Prep,NP),Set,Mask) -->
   loc_pred(Prep),
   pp(pp(prep(of),NP),compl,Set,Mask).

/* Adjectival Constructions */

adjs([Adj|Adjs]) -->
   pre_adj(Adj),
   adjs(Adjs).
adjs([]) --> [].

pre_adj(Adj) --> adj(_,Adj).
pre_adj(Adj) -->  sup_phrase(Adj).

sup_phrase(sup(most,Adj)) --> sup_adj(Adj).
sup_phrase(sup(Op,Adj)) -->
   sup_adv(Adv),
   adj(quant,Adj).

comp_phrase(comp(Comp,Adj,Arg),Mask) -->
   comp(Comp,Adj),
   {np_no_trace(NPNT), prep_case(Case)},
   np(Arg,_,Case,_,compl,NPNT,Mask).

comp(Comp,Adj) -->
   comp_adv(Comp),
   adj(quant,Adj),
   [than].
comp(more,Adj) -->
   rel_adj(Adj),
   [than].
comp(same,Adj) -->
   [as],
   adj(quant,Adj),
   [as].

/* Prepositional Phrase */

pp(pp(Prep,Arg),Case,Set,Mask) -->
   prep(Prep),
   {prep_case(NPCase)},
```

```
   np(Arg,_,NPCase,_,Case,Set,Mask).

/* Relative clause */

relative(Agmt,[Rel],Set,_,Mask) -->
   {is_pred(Set)},
   rel_conj(Agmt,Conj,Rel,Mask).
relative(_,[],_,Mask,Mask) --> [].

rel_conj(Agmt,Conj,Rel,Mask) -->
   rel(Agmt,Rel0,Mask0),
   rel_rest(Agmt,Conj,Rel0,Rel,Mask0,Mask).

rel_rest(Agmt,Conj0,Rel0,Rel,_,Mask) -->
   conj(Conj0,Conj,Rel0,Rel1,Rel),
   rel_conj(Agmt,Conj,Rel1,Mask).
rel_rest(_,_,Rel,Rel,Mask,Mask) -->

rel(Agmt,rel(X,S),Mask) -->
   open,
   variable(Agmt,X),
   s(S,Mask0),
   {trace(Trace), minus(Mask0,Trace,Mask)},
   close.

variable(Agmt,X) ... np(np(Agmt,wh(X),[]),Agmt,_,_,_,Set,Mask) -->
   [that],
   {trace(Set,Mask)}.
variable(Agmt0,X) ... np(NP,Agmt,NPCase,_,_,Set,Mask) -->
   wh(X,Agmt0,NP,Agmt,NPCase),
   {trace(Set,Mask)}.
variable(Agmt0,X) ... pp(pp(Prep,NP),compl,Set,Mask) -->
   prep(Prep),
   wh(X,Agmt0,NP,Agmt,Case),
   {trace(Set,Mask), compl_case(Case)}.

wh(X,Agmt,np(Agmt,wh(X),[]),Agmt,NPCase) -->
   rel_pron(Case),
   {role(Case,decl,NPCase)}.
wh(X,Agmt0,np(Agmt,Kernel,[pp(Prep,NP)]),Agmt,_) -->
   np_head0(Kernel,Agmt,_+common),
   prep(Prep),
   wh(X,Agmt0,NP,_,_).
wh(X,Agmt0,NP,Agmt,Case) -->
   whose(X,Agmt0),
   {s_all(SAll)},
   np(NP,Agmt,Case,def,subj,SAll,_).

/* Reduced relative clause */

reduced_relative(Agmt,Rel,Set,Mask) -->
   {is_pred(Set)},
   reduced_rel_conj(Agmt,Conj,Rel,Mask).

reduced_rel_conj(Agmt,Conj,Rel,Mask) -->
```

```
   reduced_rel(Agmt,Rel0,Mask0),
   reduced_rel_rest(Agmt,Conj,Rel0,Rel,Mask0,Mask).

reduced_rel_rest(Agmt,Conj0,Rel0,Rel,_,Mask) -->
   conj(Conj0,Conj,Rel0,Rel1,Rel),
   reduced_rel_conj(Agmt,Conj,Rel1,Mask).
reduced_rel_rest(_,_,Rel,Rel,Mask,Mask) --> [].

reduced_rel(Agmt,reduced_rel(X,S),Mask) -->
   open,
   reduced_wh(Agmt,X),
   s(S,Mask0),
   {trace(Trace), minus(Mask0,Trace,Mask)},
   close.

reduced_wh(Agmt,X),
   np(np(Agmt,wh(X),[]),Agmt,NPCase,_,_,Set0,Mask0),
   verb_form(be,pres+fin,Agmt,main),
   neg(_,Neg),
   pred(Neg,Pred,Mask) -->
   neg(_,Neg),
   pred(Neg,Pred,Mask),
   {trace(Set0,Mask0), subj_case(NPCase)}.
reduced_wh(Agmt,X),
   np(np(Agmt,wh(X),[]),Agmt,NPCase,_,_,Set,Mask),
   verb(Verb,_,Type,Voice) -->
   participle(Verb,Type,Voice),
   {trace(Set,Mask), subj_case(NPCase)}.
reduced_wh(AgmtX,X),
   np(Subj,Agmt,SCase,Def,_,Set0,Mask0) ...
   np(np(AgmtX,wh(X),[]),AgmtX,VCase,_,_,Set,Mask) -->
   {s_all(SAll), subj_case(SCase), verb_case(VCase)},
   np(Subj,Agmt,_,Def,subj,SAll,_),
   {trace(Set0,Mask0), trace(Set,Mask)}.

/* Verb phrase (less the complements) */

verb(verb(Root,Voice,Time+fin,Aspect,Neg),Agmt,Type,Voice) -->
   verb_form(Root0,Time+fin,Agmt,Role),
   {verb_type(Root0,Type0)},
   neg(Type0,Neg),
   rest_verb(Role,Root0,Root,Voice,Aspect),
   {verb_type(Root,Type)}.

neg(aux+_,neg) --> [not].
neg(_,pos) --> [].

rest_verb(aux,have,Root,Voice,[perf|Aspect]) -->
   verb_form(Root0,past+part,_,_),
   have(Root0,Root,Voice,Aspect).
rest_verb(aux,be,Root,Voice,Aspect) -->
   verb_form(Root0,Tense0,_,_),
   be(Tense0,Root0,Root,Voice,Aspect).
rest_verb(aux,do,Root,active,[]) -->
   verb_form(Root,inf,_,_).
```

```
rest_verb(main,Root,Root,active,[]) --> [].

have(be,Root,Voice,Aspect) -->
   verb_form(Root0,Tense0,_,_),
   be(Tense0,Root0,Root,Voice,Aspect).
have(Root,Root,active,[]) --> [].

be(past+part,Root,Root,passive,[]) --> [].
be(pres+part,Root0,Root,Voice,[prog]) -->
   passive(Root0,Root,Voice).

passive(be,Root,passive) -->
   verb_form(Root,past+part,_,_),
   {verb_type(Root,Type), passive(Type)}.
passive(Root,Root,active) --> [].

participle(verb(Root,Voice,inf,Aspect,Neg),Type,Voice) -->
   neg(_,Neg),
   verb_form(Root,Tense,_,_),
   {participle(Tense,Voice,Aspect), verb_type(Root,Type)}.

passive(_+trans).
passive(_+ditrans).

participle(pres+part,active,[prog]).
participle(past+part,passive,[]).

/* Extraposition brackets */

open ... close --> [].

/* Verb Arguments */

verb_args(_+Type,Voice,AdvArgs,Mask0,Mask) -->
   advs(AdvArgs,Args,_),
   verb_args(Type,Voice,Args,Mask0,Mask).

verb_args(trans,active,[arg(dir,Dir)],_,Mask) -->
   verb_arg(np,Dir,Mask).
verb_args(ditrans,_,[arg(Case,NP)|Obj],_,Mask) -->
   verb_arg(np,NP,Mask0),
   object(Case,Obj,Mask0,Mask).
verb_args(be,_,[void],Mask,Mask) --> [there].
verb_args(be,_,[arg(pred,P)],_,Mask) -->
   pred_conj(_,P,Mask).
verb_args(be,_,[arg(dir,P)],_,Mask) --> verb_arg(np,P,Mask).
verb_args(have,active,[arg(dir,P)],_,Mask) --> verb_arg(np,P,Mask).
verb_args(Type,_,[],Mask,Mask) --> {no_args(Type)}.

object(Case,AdvArg,Mask0,Mask) -->
   {adv(Adv), minus(Adv,Mask0,Mask1)},
   advs(AdvArg,Obj,Mask1),
   obj(Case,Obj,Mask0,Mask).

obj(ind,[arg(dir,NP)],_,Mask) --> verb_arg(np,NP,Mask).
```

```
obj(dir,[],Mask,Mask) --> [].

pred_conj(Conj,Arg,Mask) -->
   pred(_,Arg0,Mask0),
   pred_rest(Conj,Arg0,Arg,Mask0,Mask).

pred_rest(Conj0,Arg0,Arg,_,Mask) -->
   conj(Conj0,Conj,Arg0,Arg1,Arg),
   pred_conj(Conj,Arg1,Mask).
pred_rest(_,Arg,Arg,Mask,Mask) --> [].

verb_arg(np,NP,Mask) -->
   {s_all(SAll), verb_case(VCase)},
   np(NP,_,VCase,_,compl,SAll,Mask).

pred(_,Adj,Mask) --> adj_phrase(Adj,Mask).
pred(neg,PP,Mask) -->
   {s_all(SAll)},
   pp(PP,compl,SAll,Mask).
pred(_,Adv,Mask) -->
   {s_all(SAll)},
   adv_phrase(Adv,SAll,Mask).

advs([Adv|R0],R,Set) -->
   {is_adv(Set)},
   adverb(Adv),
   advs(R0,R,Set).
advs(R,R,_) --> [].

adj_phrase(P,Nil) --> adj(_,P), { empty(Nil) }.
adj_phrase(P,Mask) -->  comp_phrase(P,Mask).

no_args(trans).
no_args(ditrans).
no_args(intrans).

/* Conjunctions */

conj(conj(Conj,Ctx0),
     conj(Conj,Ctx),Left,Right,
     conj(Conj,Left,Right)) -->
   conj(Conj,Ctx0,Ctx).
```

Parts of Speech

The following rules define the parts of speech used in the grammar in terms of dictionary predicates:

```
/* Parts of speech */

noun(Noun,Agmt) -->
   [W],
   {noun_form(W,Noun,Agmt)}.
```

```
det(det(Det),Number,Def) -->
   [W],
   {det(W,Number,Det,Def)}.
det(generic,_,generic) --> [].

adj(Type,adj(Adj)) -->
   [Adj],
   {adj(Adj,Type)}.

prep(prep(Prep)) -->
   [Prep],
   {prep(Prep)}.

rel_adj(adj(Adj)) -->
   [RAdj],
   {rel_adj(RAdj,Adj)}.

sup_adj(adj(Adj)) -->
   [SAdj],
   {sup_adj(SAdj,Adj)}.

comp_adv(less) --> [less].
comp_adv(more) --> [more].

sup_adv(least) --> [least].
sup_adv(most) --> [most].

rel_pron(Case) -->
   [W],
   {rel_pron(W,Case)}.

verb_form(Verb,Tense,Agmt,Role) -->
   [W],
   {verb_form(W,Verb,Tense,Agmt)}.

name(Name) -->
   opt_the,
   [Name],
   {name(Name)}.

int_art(X,plu,quant(same,wh(X))) --> [how,many].
int_art(X,Agmt,DX) -->
   [Art],
   {int_art(Art,X,Agmt,DX)}.

int_pron(Case) -->
   [Pron],
   {int_pron(Pron,Case)}.

adverb(adv(Adv)) -->
   [Adv],
   {adverb(Adv)}.

poss_pron(pronoun(Gender),Person+Number) -->
```

```
   [W],
   {poss_pron(W,Gender,Person,Number)}.

pers_pron(pronoun(Gender),Person+Number,Case) -->
   [W],
   {pers_pron(W,Gender,Person,Number,Case)}.

quantifier_pron(Det,Noun) -->
   [W],
   {quantifier_pron(W,Det,Noun)}.

context_pron(prep(in),place) --> [where].
context_pron(prep(at),time) --> [when].

number(nb(I),Number) -->
   [W],
   {number(W,I,Number)}.

terminator(Type) -->
   [Term],
   {terminator(Term,Type)}.

opt_the --> [].
opt_the --> [the].

conj(_,list,list) --> [',].
conj(Conj,list,end) -->
   [Conj],
   {conj(Conj)}.

loc_pred(P) -->
   [W],
   {loc_pred(W,P)}.
```

Set Definitions and Operators for Attachment Control

```
% Normal form masks

is_pp(#(1,_,_,_)).

is_pred(#(_,1,_,_)).

is_trace(#(_,_,1,_)).

is_adv(#(_,_,_,1)).

trace(#(_,_,1,_),#(0,0,0,0)).

trace(#(0,0,1,0)).

adv(#(0,0,0,1)).

empty(#(0,0,0,0)).
```

```
np_all(#(1,1,1,0)).

s_all(#(1,0,1,1)).

np_no_trace(#(1,1,0,0)).

% Mask operations

plus(#(B1,B2,B3,B4),#(C1,C2,C3,C4),#(D1,D2,D3,D4)) :-
   or(B1,C1,D1),
   or(B2,C2,D2),
   or(B3,C3,D3),
   or(B4,C4,D4).

minus(#(B1,B2,B3,B4),#(C1,C2,C3,C4),#(D1,D2,D3,D4)) :-
   anot(B1,C1,D1),
   anot(B2,C2,D2),
   anot(B3,C3,D3),
   anot(B4,C4,D4).

or(1,_,1).
or(0,1,1).
or(0,0,0).

anot(X,0,X).
anot(X,1,0).

% Noun phrase position features

role(subj,_,#(1,0,0)).
role(compl,_,#(0,_,_)).
role(undef,main,#(_,0,_)).
role(undef,aux,#(0,_,_)).
role(undef,decl,_).
role(nil,_,_).

subj_case(#(1,0,0)).
verb_case(#(0,1,0)).
prep_case(#(0,0,1)).
compl_case(#(0,_,_)).
```

Sample Lexicon

The clauses below specify some of the words used in the Chat-80 geographical database. Most clauses for the less interesting predicates have been left out (in the places marked with the comment "% more ...").

```
% ================================================================
% General Dictionary
```

```
%

conj(and).
conj(or).

int_pron(what,undef).
int_pron(which,undef).
int_pron(who,subj).
int_pron(whom,compl).

int_art(what,X,_,int_det(X)).
int_art(which,X,_,int_det(X)).

det(the,No,the(No),def).
det(a,sin,a,indef).
det(an,sin,a,indef).
det(every,sin,every,indef).
det(some,_,some,indef).
det(any,_,any,indef).
det(all,plu,all,indef).
det(each,sin,each,indef).
det(no,_,no,indef).

number(W,I,Nb) :-
   tr_number(W,I),
   ag_number(I,Nb).

tr_number(nb(I),I).
tr_number(one,1).
%   more ...

ag_number(1,sin).
ag_number(N,plu) :- N>1.

quantifier_pron(everybody,every,person).
   % more ..

prep(of).
prep(by).
prep(with).
prep(in).
prep(into).
prep(through).
   % more ...

noun_form(Plu,Sin,plu) :- noun_plu(Plu,Sin).
noun_form(Sin,Sin,sin) :- noun_sin(Sin).

verb_form(V,V,inf,_) :- verb_root(V).
verb_form(V,V,pres+fin,Agmt) :-
   regular_pres(V),
   root_form(Agmt),
   verb_root(V).
verb_form(Past,Root,past+_,_) :-
   regular_past(Past,Root).
```

```
root_form(1+sin).
root_form(2+_).
root_form(1+plu).
root_form(3+plu).

verb_root(be).
verb_root(have).
verb_root(do).

verb_form(is,be,pres+fin,3+sin).
verb_form(are,be,pres+fin,3+plu).
   % more ...

verb_type(be,aux+be).

regular_pres(have).

regular_past(had,have).

verb_form(has,have,pres+fin,3+sin).
verb_form(having,have,pres+part,_).

verb_type(have,aux+have).

regular_pres(do).

verb_form(does,do,pres+fin,3+sin).
verb_form(did,do,past+fin,_).
verb_form(doing,do,pres+part,_).
verb_form(done,do,past+part,_).

verb_type(do,aux+ditrans).

rel_pron(who,subj).
rel_pron(whom,compl).
rel_pron(which,undef).

poss_pron(my,_,1,sin).
   % more ...

pers_pron(you,_,2,_,_).
   % more ...

terminator(.,_).
terminator(?,?).
terminator(!,!).

name(Name) :-
   name_template(Name,_), !.

% ==========================================================================
% Specialised Dictionary

adj(average,restr).
```

```
adj(african,restr).
adj(large,quant).

rel_adj(larger,large).

sup_adj(largest,large).

noun_form(proportion,proportion,_).

noun_sin(average).
noun_sin(ksqmile).
noun_sin(region).
noun_sin(sea).

noun_plu(averages,average).
noun_plu(ksqmiles,ksqmile).
noun_plu(regions,region).
noun_plu(seas,sea).

verb_root(border).
verb_root(exceed).
verb_root(flow).

regular_pres(border).

regular_past(bordered,border).

verb_form(borders,border,pres+fin,3+sin).
verb_form(bordering,border,pres+part,_).

regular_pres(exceed).

regular_past(exceeded,exceed).

verb_form(exceeds,exceed,pres+fin,3+sin).
verb_form(exceeding,exceed,pres+part,_).

verb_type(border,main+trans).
verb_type(exceed,main+trans).

regular_pres(flow).

regular_past(flowed,flow).

verb_form(flows,flow,pres+fin,3+sin).
verb_form(flowing,flow,pres+part,_).

verb_type(flow,main+intrans).
```

Appendix D

The Slot Filler

As explained in Section 4.2, the role of the slot filler is to translate a parse tree into a structure, the Quant tree, that represents the relationships between predicates and the objects that fill their argument places.

Let us recall the three main kinds of Quant tree node:

Quant	represents an argument filler, normally a noun phrase from the input;
Pred	represents a verb-like predication;
Conj	represents a conjunction of noun phrase restrictions.

The following BNF-like description gives in detail the structure of Quant trees:

```
quant-tree
    : quant(det,var,head,quant-tree,quant-list,var)
    | pred(quant-tree,op,pred,quant-list)
    | conj(conj,quant-tree,quant-list,quant-tree,quant-list)
    | quant-tree & quant-tree
    | simple-predication

var : variable - type

head : simple-predication
     | apply(variable,pred)
     | aggr(aggregation,variable,var-list,head,quant-tree)

quant-list : []
           | [quant-tree,quant-list]
```

```
var-list : []
         | [variable,var-list]

simple-predication : ` b(pred)
```

where the meanings of functors and terminal categories are the following:

P & Q	logical conjunction of the complex predications P and Q;
` P	marks a simple predication P;
apply(P,F)	the higher order operator F applied to the predication P;
aggr(F,V,L,H,P)	the aggregation operator F applied to the predicate abstraction with predication defined by H and P and abstracted variables L, gives the aggregate value V;
variable	a variable;
pred	a predication "predicate(arg_1,...,arg_n)";
type	a semantic type denotation as described in Section 4.2.3;
det	a term representing a lexical determiner or a special quantification operator (see below);
op	the name of the "modality" of a verb: currently only 'id' for empty (identity) modality or 'not' for negation;
aggregation	the name of an aggregation operator.

I discussed the format of slots in Section 4.2.3. The actual slot records used in the program are somewhat more complicated, and have the following form:

```
slot(case,type,variable,filler-id,index)
```

The case field represents the case marker for the slot:

prep(prep)	The preposition **prep**;
poss	possessive;
subj	subject;
dir	direct object;
ind	indirect object;
pred	a predicative complement to a verb, eg. an adjective phrase as direct object of the verb "to be".

The **type** field has already been described. The **variable** is the predicate argument corresponding to this slot. The **filler-id** is the unique identifier for the subtree that fills this slot. A given predicate argument may have alternative slots that share a single **filler-id**. As different modifiers have different identifiers, at most one of the slots that share a given **filler-id** may be used to attach modifiers. Finally, the **index** field has either the value 'index', meaning that the filler of this slot is allowed to appear as index of the indexed set that translates a plural noun phrase to whose head the slot belongs (see Section 4.3.2), or the value 'free', meaning that the filler of the slot cannot be used as the index of an indexed set.

The overall structure of the slot filler is very simple. For each type of parse tree node node, eg. 'np', there is an interpretation predicate 'i_node' that translates that nodes of that type into one or a set of Quant tree fragments. Because of the attachment rules in the grammar, a syntax tree node may contain sub-nodes that actually belong higher up in the tree. Therefore, interpretation predicates may need to pass to higher levels an uninterpreted **modifier list**. As

I noted above, parse tree nodes must be given unique identifiers so that each node that fills a slot fills the slot's slot-id with a unique value. Another argument is needed to generate these unique identifiers. The prototypical form of an interpretation predicate is thus:

```
i_node(syntax,
       Quant tree fragments,
       uninterpreted syntax,
       identifier)
```

Particular interpretation predicates will be variations on this pattern. For instance, predicates for interpreting sub-nodes of a noun phrase will in general contribute to build both the predication and the modifiers of the Quant for the noun phrase. The arguments Quant tree fragments will then have the form:

```
range variable,
slot list pair,
predication pair,
modifier list pair
```

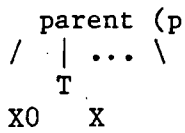
where the predication and modifier list contributed by this node and the slots used are for programming convenience represented by pairs of arguments (the "difference pair technique"):

value before this node, value after it

Some interpretation predicates have arguments for two further purposes: to change the determiners of lower Quants in nested plural plural noun phrases; and to move into its right place the "extraposed" possessive Quant created by the verb "to have" or the preposition "with" (see Section 4.2.6).

Outline of the Slot Filling Predicates

The outline that follows refers to the clauses listed in Appendix C. In describing the arguments of the predicates, I will use without further comment the concepts introduced in Section 4.2. Many of the arguments are common to most of the predicates, and their naming will reflect the following picture



For a node T, and an attribute X being used or produced in the slot filling process, X0 names the attribute value for the subtree rooted at the parent p of T containing T and all the descendants of p to the right of T; X names the value of the same attribute for the descendants of p for the right of T. That is, X and X0 form a difference pair of attribute X. These are the names used for the common arguments:

T	the syntax tree to be interpreted;
Q	the resulting Quant tree;
U	the uninterpreted modifier list for tree T;
I	the unique identifier for T;
S0	the slot list used to attach T and the nodes to its right;
S	slot list for the nodes to the right of T;
A0	the (partial) argument list of the enclosing Quant or Pred including the contributions of T and the nodes to its right;
A	as above, but excluding T;

PO — the (partial) predication of the enclosing noun phrase Quant containing the contributions of T and the nodes to its right;

P — as above, but excluding T;

IX — specifies whether any subtree attached at this level or below will be inside an higher plural noun phrase;

XA0, XA — as noted above, the verb "to have" is interpreted by creating a possessive modifier to modify the direct object of the verb; XA0 carries the possible modifier to the predicate, and XA returns either '[]' or the modifier if it was on XA0 and was not attached in the node.

i_sentence(T,Q) translates sentence T into Q (only questions are treated in the current code).

i_s(T,Q,U,I) translates the basic declarative sentence.

i_verb(T,P,Tense,Voice,Neg,Slots,XMod,Meta)
T is a verb in tense Tense, voice Voice, negated or not according to Neg, with slots Slots, and needing the reshaping of the final Quant tree described by extraposed possessive modifier XMod and feature Meta.

i_subj(V,T,S0,S,Q,U,I)
translates the subject of a sentence whose main verb is in voice V, and has slot list S0.

fill_verb(T,XA0,XA,S0,S,A0,A,U,I)
translates the verb arguments or modifiers T; 'i_verb_args' and 'i_verb_mods' are special cases of this predicate.

verb_slot(T,XA0,XA,S0,S,A0,A,U,I)
attaches T to a verb slot.

i_np(T,X,Q,U,I,IX,XA0,XA)
translates a noun phrase into a Quant with bound variable X.

i_pred(T,X,A0,A,U,I)
translates a verb argument capable of filling a 'pred' slot (eg. an adjective phrase), restricting variable X.

i_np_head(T,Y,Q,Det,Det0,X,P,A,S0,I)
translates the head of a noun phrase T with lexical determiner Det0 into a Quant with determiner Det[30], range variable X and bound variable Y.

held_arg(XA0,XA,S0,S,I0,I)
determines whether the an extraposed possessive can be attached to the current noun phrase, and modifies the unique identifier assignment I0 into I if that is the case, so that the extraposed modifier has also a unique identifier.

i_np_rest(T,Det,Det0,X,P0,A0,S0,U,I,IX)
translates the post-modifiers of a noun phrase T with lexical determiner Det0, Quant determiner Det and range variable X; 'i_np_mods' does most of the work of this predicate, except as for 'index_args' below.

index_args(Det0,IX0,I,Det,IX)
translates a lexical determiner Det0 into a Quant determiner Det and an indexing feature for lower subtrees, according to the role of Det's noun phrase in an indexed set or higher order construction defined by Det and the slot index feature IX0.

i_np_mod(T,X,S0,S,P0,P,A0,A,U,I,IX)
translates a noun phrase post-modifier for a noun phrase with range variable X.

i_rel(T,X,P0,P,A0,A,U,I)
translates a relative clause as a modifier of a noun phrase with range variable X.

i_bind(Prep,S0,S,X,Y,I,Fn,P,PSlots,XA0)
Fn and P describe the way (argument or adjunction) in which a prepositional phrase, whose preposition is Prep and whose noun phrase has bound variable Y, modifies a higher noun phrase with range variable X; adjoining prepositions behave like verbs, and return therefore a slot list PSlots.

i_np_modify(Fn,P,Q0,Q,IX0,IX)
a prepositional phrase is modifying a noun phrase as described by arguments Fn and P of 'i_bind', making a Quant argument Q out of the prepositional phrase's noun phrase Quant Q0, and changing the indexed set feature IX0 of the higher noun phrase into a new value IX for any lower noun phrases.

---

[30]Not always the same as Det0 because of words that modify their arguments.

in_slot(S0,Case,X,I,S,IX)
finds a slot with case Case, (typed) variable X, identifier I, and indexing feature IX.

Slot Filler Code

```
:- op(450,xfy,:).
:- op(400,xfy,&).
:- op(300,fx,`).
:- op(200,xfx,--).

i_sentence(q(S),question([],P)) :-
   i_s(S,P,[],0).
i_sentence(whq(X,S),question([X],P)) :-
   i_s(S,P,[],0).
i_sentence(imp(s(_,Verb,VArgs,VMods)),imp(V,Args)) :-
   i_verb(Verb,V,_,active,pos,Slots0,[],transparent),
   i_verb_args(VArgs,[],[],Slots0,Slots,Args,Args0,Up,-0),
   conc(Up,VMods,Mods),
   i_verb_mods(Mods,_,[],Slots,Args0,Up,+0).

i_np(there,Y,quant(void,X,`true,`true,[],Y),[],_,_,XA,XA).
i_np(NP,Y,Q,Up,Id0,Index,XA0,XA) :-
   i_np_head(NP,Y,Q,Det,Det0,X,Pred,QMods,Slots0,Id0),
   held_arg(XA0,XA,Slots0,Slots,Id0,Id),
   i_np_rest(NP,Det,Det0,X,Pred,QMods,Slots,Up,Id,Index).

i_np_head(np(_,Kernel,_),Y,
      quant(Det,T,Head,Pred0,QMods,Y),
      Det,Det0,X,Pred,QMods,Slots,Id) :-
   i_np_head0(Kernel,X,T,Det0,Head,Pred0,Pred,Slots),
   Type-_=Y, Type-_=T.

i_np_rest(np(_,_,Mods),Det,Det0,X,Pred,QMods,Slots,Up,Id,Index) :-
   index_args(Det0,Index,Id,Det,IndexA),
   i_np_mods(Mods,X,Slots,Pred,QMods,Up,Id,IndexA).

held_arg(held_arg(Case,-Id,X),[],S0,S,Id,+Id) :-
   in_slot(S0,Case,X,Id,S,_).
held_arg(XA,XA,S,S,Id,Id).

i_np_head0(np_head(Det,Adjs,Noun),X,T,Det,Head0,Pred0,Pred,Slots) :-
   i_adjs(Adjs,X,T,X,Head0,Head,Pred0,Pred),
   i_noun(Noun,X,Head,Slots).
i_np_head0(np_head(int_det(V),Adjs,Noun),
      Type-X,Type-X,Det,`true,Pred,Pred,
      [slot(prep(of),Type,X,_,comparator)]) :-
   comparator(Noun,Type,V,Adjs,Det).
i_np_head0(np_head(quant(Op0,N),Adjs,Noun),
      Type-X,Type-X,void,`P,Pred,Pred,[]) :-
   measure(Noun,Type,Adjs,Units),
   conversion(N,Op0,Type,V,Op),
   measure_op(Op,X,V--Units,P).
i_np_head0(name(Name),
```

```
      Type-Name,Type-Name,id,`true,Pred,Pred,[]) :-
   name_template(Name,Type).
i_np_head0(wh(X),X,X,id,`true,Pred,Pred,[]).

i_np_mods([],_,[],`true,[],[],_,_).
i_np_mods([Mod|Mods],X,Slots0,Pred0,QMods0,Up,Id,Index) :-
   i_np_mod(Mod,X,Slots0,Slots,
            Pred0,Pred,QMods0,QMods,Up0,-Id,Index),
   conc(Up0,Mods,Mods0),
   i_np_mods(Mods0,X,Slots,Pred,QMods,Up,+Id,Index).
i_np_mods(Mods,_,[Slot|Slots],`true,QMods,Mods,Id,_) :-
   i_voids([Slot|Slots],QMods,Id).

i_voids([],[],_).
i_voids([Slot|Slots],[quant(void,X,`true,`true,[],_)|QMods],Id) :-
   slot_tag(Slot,X,-Id), !,
   i_voids(Slots,QMods,+Id).
i_voids([_|Slots],QMods,Id) :-
   i_voids(Slots,QMods,Id).

i_rel(rel(X,S),X,P&Pred,Pred,QMods,QMods,Up,Id) :-
   i_s(S,P,Up,Id).
i_rel(reduced_rel(X,S),X,Pred,Pred,[A|QMods],QMods,Up,Id) :-
   i_s(S,A,Up,Id).
i_rel(conj(Conj,Left,Right),X,
      conj(Conj,LPred,LQMods,RPred,RQMods)&Pred,Pred,
      QMods,QMods,Up,Id) :-
   i_rel(Left,X,LPred,`true,LQMods,[],[],-Id),
   i_rel(Right,X,RPred,`true,RQMods,[],Up,+Id).

i_np_mod(pp(Prep,NP),
      X,Slots0,Slots,Pred,Pred,[QMod|QMods],QMods,Up,Id0,Index0) :-
   i_np_head(NP,Y,Q,LDet,LDet0,LX,LPred,LQMods,LSlots0,Id0),
   i_bind(Prep,Slots0,Slots1,X,Y,Id0,Function,P,PSlots,XArg),
   conc(PSlots,Slots1,Slots),
   i_np_modify(Function,P,Q,QMod,Index0,Index),
   held_arg(XArg,[],LSlots0,LSlots,Id0,Id),
   i_np_rest(NP,LDet,LDet0,LX,LPred,LQMods,LSlots,Up,Id,Index).
i_np_mod(Mod,X,Slots,Slots,Pred0,Pred,QMods0,QMods,Up,Id,_) :-
   i_rel(Mod,X,Pred0,Pred,QMods0,QMods,Up,Id).

i_noun(Noun,Type-X,P,Slots) :-
   noun_template(Noun,Type,X,P,Slots).

i_bind(Prep,Slots0,Slots,_,X,Id,arg,P,[],[]) :-
   in_slot(Slots0,Case,X,Id,Slots,P),
   deepen_case(Prep,Case).
i_bind(prep(Prep),Slots,SLots,X,Y,_,adjoin,`P,PSlots,XArg) :-
   i_adjoin(Prep,X,Y,PSlots,XArg,P).

i_np_modify(adjoin,P,N,N&P,_,unit).
i_np_modify(arg,F,N,N,Index0,Index) :-
   index_slot(F,Index0,Index).

in_slot([Slot|Slots],Case,X,Id,Slots,F) :-
```

```
   slot_match(Slot,Case,X,Id,F).
in_slot([Slot|Slots0],Case,X,Id,[Slot|Slots],F) :-
   in_slot(Slots0,Case,X,Id,Slots,F).

slot_match(slot(Case,Type,X,Id,F),Case,Type-X,Id,F).

i_adjs([],X,T,T,Head,Head,Pred,Pred).
i_adjs([Adj|Adjs],X,T,T0,Head0,Head,Pred0,Pred) :-
   i_adj(Adj,X,T,T1,Head0,Head1,Pred0,Pred1),
   i_adjs(Adjs,X,T1,T0,Head1,Head,Pred1,Pred).

i_adj(adj(Adj),Type-X,T,T,Head,Head,`P&Pred,Pred) :-
   restriction(Adj,Type,X,P).
i_adj(adj(Adj),TypeX-X,TypeV-V,_,
   aggr(F,V,[X],Head,Pred),Head,`true,Pred) :-
   aggr_adj(Adj,TypeV,TypeX,F).
i_adj(sup(Op0,adj(Adj)),Type-X,Type-V,_,
      aggr(F,V,[Y,X],Head,`P&Pred),Head,`true,Pred) :-
   sign(Adj,Sign),
   inverse(Op0,Sign,Op),
   i_sup_op(Op,F),
   attribute(Adj,Type,X,_,Y,P).
i_adj(adj(Adj),TypeX-X,T,T,_,
      Head,Head,quant(void,TypeX-Y,`P,`Q&Pred,[],_),Pred) :-
   attribute(Adj,TypeX,X,_,Y,P),
   standard(Adj,TypeX,Y,Q).

i_s(s(Subj,Verb,VArgs,VMods),Pred,Up,Id) :-
   i_verb(Verb,P,Tense,Voice,Neg,Slots0,XA0,Meta),
   i_subj(Voice,Subj,Slots0,Slots1,QSubj,SUp,-(-Id)),
   conc(SUp,VArgs,TArgs),
   i_verb_args(TArgs,XA0,XA,Slots1,Slots,Args0,Args,Up0,+(-Id)),
   conc(Up0,VMods,Mods),
   i_verb_mods(Mods,Tense,XA,Slots,Args,Up,+Id),
   reshape_pred(Meta,QSubj,Neg,P,Args0,Pred).

i_verb(verb(Root,Voice,Tense,Aspect,Neg),
      P,Tense,Voice,Det,Slots,XArg,Meta) :-
   verb_template(Root,P,Slots,XArg,Meta),
   i_neg(Neg,Det).

reshape_pred(transparent,S,N,P,A,pred(S,N,P,A)).
reshape_pred(have,Subj,Neg,Verb0,
      [quant(Det,X,Head0,Pred,QArgs,Y)|MRest],
      pred(Subj,Neg,Verb,[quant(Det,X,Head,Pred,QArgs,Y)|MRest])) :-
   have_pred(Head0,Verb0,Head,Verb).

have_pred(`Head,Verb,`true,(Head,Verb)).
have_pred(Head,Verb,Head,Verb) :-
   meta_head(Head).

meta_head(apply(_,_)).
meta_head(aggr(_,_,_,_,_)).

i_neg(pos,id).
```

```
i_neg(neg,not).

i_subj(Voice,Subj,Slots0,Slots,Quant,Up,Id) :-
   subj_case(Voice,Case),
   verb_slot(arg(Case,Subj),[],[],Slots0,Slots,[Quant],[],Up,Id).

i_verb_args(VArgs,XA0,XA,Slots0,Slots,Args0,Args,Up,Id) :-
   fill_verb(VArgs,XA0,XA,Slots0,Slots,Args0,Args,Up,Id).

subj_case(active,subj).
subj_case(passive,s_subj).

fill_verb([],XA,XA,Slots,Slots,Args,Args,[],_).
fill_verb([Node|Nodes0],XA0,XA,Slots0,Slots,Args0,Args,Up,Id) :-
   verb_slot(Node,XA0,XA1,Slots0,Slots1,Args0,Args1,Up0,-Id),
   conc(Up0,Nodes0,Nodes),
   fill_verb(Nodes,XA1,XA,Slots1,Slots,Args1,Args,Up,+Id).

verb_slot(pp(Prep,NP),
      XArg0,XArg,Slots0,Slots,[Q|Args],Args,Up,Id) :-
   i_np(NP,X,Q,Up,Id,unit,XArg0,XArg),
   in_slot(Slots0,Case,X,Id,Slots,_),
   deepen_case(Prep,Case).
verb_slot(void,XA,XA,Slots,Slots,Args,Args,[],_) :-
   in_slot(Slots,pred,_,_,_,_).
verb_slot(pp(prep(Prep),NP),
      TXArg,TXArg,Slots0,Slots,[Q& `P|Args],Args,Up,Id0) :-
   in_slot(Slots0,pred,X,Id0,Slots1,_),
   i_adjoin(Prep,X,Y,PSlots,XArg,P),
   i_np_head(NP,Y,Q,LDet,LDet0,LX,LPred,LQMods,LSlots0,Id0),
   held_arg(XArg,[],LSlots0,LSlots,Id0,Id),
   i_np_rest(NP,LDet,LDet0,LX,LPred,LQMods,LSlots,Up,Id,free),
   conc(PSlots,Slots1,Slots).
verb_slot(arg(SCase,NP),
      XArg0,XArg,Slots0,Slots,[Q|Args],Args,Up,Id) :-
   i_np(NP,X,Q,Up,Id,unit,XArg0,XArg),
   in_slot(Slots0,Case,X,Id,Slots,_),
   deepen_case(SCase,Case).
verb_slot(adverb(Adv),XA,XA,Slots0,Slots,[`P|Args],Args,[],Id) :-
   adv_template(Adv,Case,X,P),
   in_slot(Slots0,Case,X,Id,Slots,_).
verb_slot(arg(pred,AP),XA,XA,Slots0,Slots,Args0,Args,Up,Id) :-
   in_slot(Slots0,pred,X,Id,Slots,_),
   i_pred(AP,X,Args0,Args,Up,Id).

i_pred(conj(Conj,Left,Right),X,
      [conj(Conj,`true,LQMods,`true,RQMods)|QMods],
      QMods,Up,Id) :-
   i_pred(Left,X,LQMods,[],[],-Id),
   i_pred(Right,X,RQMods,[],Up,+Id).
i_pred(AP,T,[`Head&Pred|As],As,[],_) :-
   i_adj(AP,T,_,_,Head,true,Pred,`true).
i_pred(value(adj(Adj),wh(TypeY-Y)),Type-X,[`H|As],As,[],_) :-
   attribute(Adj,Type,X,TypeY,Y,H).
i_pred(comp(Op0,adj(Adj),NP),X,[P1 & P2 & `P3,Q|As],As,Up,Id) :-
```

```
    _np(NP,Y,Q,Up,Id,unit,[],[]),
   sign(Adj,Sign),
   i_measure(X,Adj,Type,U,P1),
   i_measure(Y,Adj,Type,V,P2),
   inverse(Op0,Sign,Op),
   measure_op(Op,U,V,P3).
i_pred(pp(prep(Prep),NP),X,[`H,Q|As],As,Up,Id) :-
   i_np(NP,Y,Q,Up,Id,unit,[],[]),
   adjunction(Prep,X,Y,H).

i_adjoin(with,TS-S,TV-Y,[slot(prep(of),TV,Z,_,free)],
        held_arg(poss,-Id,TS-S),
        Y=Z).
i_adjoin(Prep,X,Y,[],[],P) :-
   adjunction(Prep,X,Y,P).

i_measure(Type-X,Adj,Type,X,`true) :-
   units(Adj,Type).
i_measure(TypeX-X,Adj,TypeY,Y,quant(void,TypeY-Y,`P,`true,[],_)) :-
   attribute(Adj,TypeX,X,TypeY,Y,P).

i_verb_mods(Mods,_,XA,Slots0,Args0,Up,Id) :-
   fill_verb(Mods,XA,[],Slots0,Slots,Args0,Args,Up,-Id),
   i_voids(Slots,Args,+Id).

slot_tag(slot(_,Type,X,Id,_),Type-X,Id).

i_sup_op(least,min).
i_sup_op(most,max).

conversion(wh(Type-X),same,Type,X,id).
conversion(nb(N),Op,_,N,Op).

measure_op(id,X,X,true).
measure_op(same,X,Y,X=Y).
measure_op(less,X,Y,exceeds(Y,X)).
measure_op(not+less,X,Y,\+exceeds(Y,X)).
measure_op(more,X,Y,exceeds(X,Y)).
measure_op(not+more,X,Y,\+exceeds(X,Y)).

inverse(most,-,least).
inverse(least,-,most).
inverse(same,-,same).
inverse(less,-,more).
inverse(more,-,less).
inverse(X,+,X).

noun_template(Noun,TypeV,V,`P,
      [slot(poss,Type0,0,Os,index)|Slots]) :-
   property(Noun,TypeV,V,Type0,0,P,Slots,Os,_).
noun_template(Noun,TypeV,V,aggr(F,V,[],`true,`true),
   [slot(prep(of),TypeS,_,_,free)]) :-
   aggr_noun(Noun,TypeV,TypeS,F).
noun_template(Noun,Type,X,`P,Slots) :-
   thing(Noun,Type,X,P,Slots,_).
```

```
noun_template(Noun,TypeV,V,apply(F,P),
      [slot(prep(of),TypeX,X,_,apply)]) :-
   meta_noun(Noun,TypeV,V,TypeX,X,P,F).

verb_template(have,Y=Z,
                [slot(subj,TypeS,S,-Id,free),
                 slot(dir,TypeV,Y,_,free),
                 slot(prep(of),TypeV,Z,_,free)],
                held_arg(poss,-(-(+Id)),TypeS-S), have).
verb_template(have,Y=Z,
        [slot(subj,TypeS,S,-(-(Id)),free),
         slot(dir,TypeV,Y,_,free),
         slot(prep(as),TypeV,Z,_,free)],
        held_arg(poss,-(-(-(+Id))),TypeS-S), have).
verb_template(Verb,Pred,
      [slot(subj,TypeS,S,_,free)|Slots],[],transparent) :-
   verb_type(Verb,+Kind),
   verb_kind(Kind,Verb,TypeS,S,Pred,Slots).

verb_kind(be,_,TypeS,S,S=A,[slot(dir,TypeS,A,Slot,free),
                            slot(pred,TypeS,A,Slot,free)]).
verb_kind(intrans,Verb,TypeS,S,Pred,Slots) :-
   intrans(Verb,TypeS,S,Pred,Slots,_).
verb_kind(trans,Verb,TypeS,S,Pred,
      [slot(dir,TypeD,D,SlotD,free)|Slots]) :-
   trans(Verb,TypeS,S,TypeD,D,Pred,Slots,SlotD,_).
verb_kind(ditrans,Verb,TypeS,S,Pred,
      [slot(dir,TypeD,D,SlotD,free),
       slot(ind,TypeI,I,SlotI,free)|Slots]) :-
   ditrans(Verb,TypeS,S,TypeD,D,TypeI,I,Pred,Slots,SlotD,SlotI,_).

deepen_case(prep(at),time).
deepen_case(s_subj,dir).
deepen_case(s_subj,ind).
deepen_case(prep(by),subj).
deepen_case(prep(to),ind).
deepen_case(prep(of),poss).
deepen_case(X,X).

% ================================================================
% Determiner Indexing Table

index_slot(index,I,I).
index_slot(free,_,unit).
index_slot(apply,_,apply).
index_slot(comparator,_,comparator).

index_args(det(the(plu)),unit,I,set(I),index(I)) :- !.
index_args(int_det(X),index(I),_,int_det(I,X),unit) :- !.
index_args(generic,apply,_,lambda,unit) :-!.
index_args(D,comparator,_,id,unit) :-
 ( indexable(D); D=generic), !.
index_args(D,unit,_,D,unit) :- !.
index_args(det(D),I,_,I,I) :-
   indexable(D),
```

```
   index(I), !.
index_args(D,I,_,D,I).

indexable(the(plu)).
indexable(all).

index(index(I)).

% ================================================================
% Utilities

conc([],L,L).
conc([X|L1],L2,[X|L3]) :-
   conc(L1,L2,L3).
```

The Template Dictionary

The actual predicates that represent the template dictionary take a word, denoted below by W, and define the various parts of the slot list and the translation of the word. Because of the mechanism described above for dealing with alternative slots for the same argument, some of the predicates need an argument, denoted below by C, to hold a table of shared slot identifiers. The type of variable x below is written below as Typex. Here are the predicates:

property(W,TypeX,X,TypeY,Y,P,S,IY,C)
: A noun that translates as a property P assigning to object X a value Y, with additional slots S, and identifier IY for the Y tree.

thing(W,Type,X,P,S,C)
: A noun representing a class of objects defined by predicate P over variable X, S is the slot list for the other arguments of P.

measure(W,Type,Adjs,M)
: A noun that together with a sequence of adjectives Adjs names a unit of measure of type Type and unit name m.

aggr_noun(W,TypeV,TypeS,F)
: an aggregation noun that translates into operator F producing a value of type TypeV when applied to a set of objects of type TypeS.

meta_noun(W,TypeV,V,TypeX,X,P,F)

A noun that translates into a higher level operator F which when applied to the predicate abstraction lambda(X).P produces a value V.

restriction(W,Type,X,P)
a restrictive adjective that translates into predication P applied to variable X.

attribute(W,TypeX,X,TypeV,V,P)
an adjective of measure, which is translated in terms of a predication P that assigns to an object X a value V.

units(Adj,Type) returns the type of value Type associated to the adjective of measure Adj.

aggr_adj(W,TypeV,TypeS,F)
The same as 'aggr_noun' but for W used as an adjective.

intrans(W,TypeX,X,P,S,C)
an intransitive verb with subject X and all other arguments as in 'thing' above.

trans(W,TypeX,X,TypeY,Y,P,S,IY,C)
a transitive verb with subject X and object Y, and all other arguments as in 'property' above.

ditrans(W,TypeX,X,TypeY,Y,TypeZ,Z,P,S,IY,IZ,X)
a ditransitive verb W with indirect object Z and subtree identifier IZ, and all the other arguments as for 'trans'.

adjunction(W,TypeX,X,TypeY,Y,P)
a preposition that can be used to introduce an adjunction, translated into predication P restricting variable X and taking variable Y as the "object" of the preposition.

The following templates are some of those used in the Chat-80 geographical database.

Sample Templates

```
/* Nouns */

property(area,measure&area,X,feature&place&_,Y,area(Y,X),[],_,_).
property(capital,feature&city,X,feature&place&country,Y,
        capital(Y,X),[],_,_).
```

```
thing(region,feature&place&_,X,region(X),[],_).
thing(sea,feature&place&seamass,X,sea(X),[],_).

aggr_noun(average,_,_,average).

meta_noun(number,_,V,feature&_,X,P,numberof(X,P,V)).

/* Proper nouns */

name_template(X,feature&place&_) :- region(X).
name_template(X,feature&place&seamass) :- seamass(X).

/* Verbs */

trans(border,
      feature&place&_,X,feature&place&_,Y,borders(X,Y),[],_,_
trans(exceed,measure&Type,X,measure&Type,Y,exceeds(X,Y),[],_,_).

intrans(flow,feature&river,X,flows(X,Y),
   [slot(prep(through),feature&place&_,Y,_,free)],_).
intrans(flow,feature&river,X,flows(X,Y,Z),
   [slot(prep(into),feature&place&_,Z,_,free),
    slot(prep(from),feature&place&_,Y,_,free)],_).

/* Adjectives */

restriction(african,feature&_,X,african(X)).

attribute(large,feature&place&_,X,measure&area,Y,area(X,Y)).

aggr_adj(average,_,_,average).

/* Prepositions */

adjunction(in,feature&_-X,feature&place&_-Y,in(X,Y)).

/* Measure */

measure(ksqmile,measure&area,[],ksqmiles).

units(large,measure&_).

sign(large,+).

/* Proportions and the like */

comparator(proportion,_,V,[],proportion(V)).
```

Appendix E

Scope Determination

The derivation of a final logical form from a Quant tree is organised in four interleaved operations:

* Quants "quant(Det,X,Head,Pred,Args,Y)" are translated by the predicate 'pre_apply' into range terms "quant(Det,X,P,Y)" where P is the logical form of the range of the Quant but the scope hasn't yet been found;

* range terms are sorted according to the precedences of their determiners (predicate 'split_quants');

* the residues, ie. the portions of the final form that have no quantifiers or whose shape has been determined, are separated from un-applied range terms (predicate 'quantify');

* the determiners in range terms are applied to their range and scope to produce quantified logical formulae (predicates 'chain_apply' and 'det_apply').

This picture is complicated by the special treatment of plural noun phrases, aggregations and higher order operators, which is achieved by 'pre_apply' and the predicates used in its definition.

The relative scopes of determiners are defined by the predicate 'governs', which is a straightforward implementation of the notions introduced in Section 4.3.1.

Outline of Scoping Predicates

The following are the main scoping predicates:

clausify(T,P) clause P is a translation of the question with Quant tree T.

quantify(T,Q,R,P)

Q is the list of range terms from Quants in T and in subtrees to its right whose scope is not smaller than the residue P that translates the rest of T, where R is a list of range terms from Quants to the right of T.

split_quants(D,Q,A0,A,B0,B)

For a determiner or operator D and a list of range terms Q, the pair <A0,A> is a difference pair representation of a list of the elements of Q whose determiner governs D, and <B0,B> is a similar representation of a list of those elements of Q that do not govern D.

pre_apply(H,D,X,P1,P2,Y,B,Q)

Q is the range term for a Quant with head H, determiner D, range variable X and scope variable Y, where P1 is the residue of the arguments of the Quant, P2 is the translation of the predication of the Quant, and B is a list of those range terms from the arguments of the Quant that do not govern it.

quantify_args(A,Q,P)

the list of arguments A rewrites into a list of range terms Q and a residue P.

det_apply(Q,P0,P)

the application of range term Q to the scope P0 is formula P.

chain_apply(Q,P0,P)

P is the open formula obtained by applying each range term in Q to a scope formed by the application of all the elements of Q with less precedence to the open formula P0.

Scoping code

```
clausify(question(V0,P),(answer(V):-B)) :-
   quantify(P,Quants,[],R0),
   split_quants(question(V0),Quants,HQuants,[],BQuants,[]),
   chain_apply(BQuants,R0,R1),
   head_vars(HQuants,B,R1,V,V0).

quantify(quant(Det,X,Head,Pred,Args,Y),Above,Right,true) :-
   close_tree(Pred,P2),
   quantify_args(Args,AQuants,P1),
   split_quants(Det,AQuants,Above,[Q|Right],Below,[]),
   pre_apply(Head,Det,X,P1,P2,Y,Below,Q).
quantify(conj(Conj,LPred,LArgs,RPred,RArgs),Up,Up,P) :-
   close_tree(LPred,LP0),
```

```
   quantify_args(LArgs,LQs,LP1),
   chain_apply(LQs,(LP0,LP1),LP),
   close_tree(RPred,RP0),
   quantify_args(RArgs,RQs,RP1),
   chain_apply(RQs,(RP0,RP1),RP),
   conj_apply(Conj,LP,RP,P).
quantify(pred(Subj,Op,Head,Args),Above,Right,P) :-
   quantify(Subj,SQuants,[],P0),
   quantify_args(Args,AQuants,P1),
   split_quants(Op,AQuants,Up,Right,Here,[]),
   conc(SQuants,Up,Above),
   chain_apply(Here,(P0,Head,P1),P2),
   op_apply(Op,P2,P).
quantify(`P,Q,Q,P).
quantify(P&Q,Above,Right,(S,T)) :-
   quantify(Q,Right0,Right,T),
   quantify(P,Above,Right0,S).

head_vars([],P,P,L,L0) :-
   strip_types(L0,L).
head_vars([Quant|Quants],(P,R0),R,[X|V],V0) :-
   extract_var(Quant,P,X),
   head_vars(Quants,R0,R,V,V0).

strip_types([],[]).
strip_types([_-X|L0],[X|L]) :-
   strip_types(L0,L).

extract_var(quant(_,_-X,P,_-X),P,X).

chain_apply(Q0,P0,P) :-
   sort_quants(Q0,Q,[]),
   chain_apply0(Q,P0,P).

chain_apply0([],P,P).
chain_apply0([Q|Quants],P0,P) :-
   chain_apply0(Quants,P0,P1),
   det_apply(Q,P1,P).

quantify_args([],[],true).
quantify_args([Arg|Args],Quants,(P,Q)) :-
   quantify_args(Args,Quants0,Q),
   quantify(Arg,Quants,Quants0,P).

pre_apply(`Head,set(I),X,P1,P2,Y,Quants,Quant) :-
   indices(Quants,I,Indices,RestQ),
   chain_apply(RestQ,(Head,P1),P),
   setify(Indices,X,(P,P2),Y,Quant).
pre_apply(`Head,Det,X,P1,P2,Y,Quants,quant(Det,X,(P,P2),Y)) :-
 ( unit_det(Det);
   index_det(Det,_)),
   chain_apply(Quants,(Head,P1),P).
pre_apply(apply(F,P0),Det,X,P1,P2,Y,
      Quants0,quant(Det,X,(P3,P2),Y)) :-
   but_last(Quants0,quant(lambda,Z,P0,Z),Quants),
```

```
   chain_apply(Quants,(F,P1),P3).
pre_apply(aggr(F,Value,L,Head,Pred),Det,X,P1,P2,Y,Quants,
      quant(Det,X,
            (S^(setof(Range:Domain,P,S),
                aggregate(F,S,Value)),P2),Y)) :-
   close_tree(Pred,R),
   complete_aggr(L,Head,(R,P1),Quants,P,Range,Domain).

but_last([X|L0],Y,L) :-
   but_last0(L0,X,Y,L).

but_last0([],X,X,[]).
but_last0([X|L0],Y,Z,[Y|L]) :-
   but_last0(L0,X,Z,L).

close_tree(T,P) :-
   quantify(T,Q,[],P0),
   chain_apply(Q,P0,P).

meta_apply(`G,R,Q,G,R,Q).
meta_apply(apply(F,(R,P)),R,Q0,F,true,Q) :-
   but_last(Q0,quant(lambda,Z,P,Z),Q).

indices([],_,[],[]).
indices([Q|Quants],I,[Q|Indices],Rest) :-
   open_quant(Q,Det,_,_,_),
   index_det(Det,I),
   indices(Quants,I,Indices,Rest).
indices([Q|Quants],I,Indices,[Q|Rest]) :-
   open_quant(Q,Det,_,_,_),
   unit_det(Det),
   indices(Quants,I,Indices,Rest).

setify([],Type-X,P,Y,quant(set,Type-([]:X),true:P,Y)).
setify([Index|Indices],X,P,Y,Quant) :-
   pipe(Index,Indices,X,P,Y,Quant).

pipe(quant(int_det(_,Z),Z,P1,Z),
     Indices,X,P0,Y,quant(det(a),X,P,Y)) :-
   chain_apply(Indices,(P0,P1),P).
pipe(quant(index(_),_-Z,P0,_-Z),Indices,Type-X,P,Y,
     quant(set,Type-([Z|IndexV]:X),(P0,P1):P,Y)) :-
   index_vars(Indices,IndexV,P1).

index_vars([],[],true).
index_vars([quant(index(_),_-X,P0,_-X)|Indices],
     [X|IndexV],(P0,P)) :-
   index_vars(Indices,IndexV,P).

complete_aggr([Att,Obj],`G,R,Quants,(P,R),Att,Obj) :-
   chain_apply(Quants,G,P).
complete_aggr([Att],Head,R0,Quants0,(P1,P2,R),Att,Obj) :-
   meta_apply(Head,R0,Quants0,G,R,Quants),
   set_vars(Quants,Obj,Rest,P2),
   chain_apply(Rest,G,P1).
```

```
complete_aggr([],`G,R,[quant(set,_-(Obj:Att),S:T,_)],
      (G,R,S,T),Att,Obj).

set_vars([quant(set,_-(I:X),P:Q,_-X)],[X|I],[],(P,Q)).
set_vars([],[],[],true).
set_vars([Q|Qs],[I|Is],R,(P,Ps)) :-
   open_quant(Q,Det,X,P,Y),
   set_var(Det,X,Y,I), !,
   set_vars(Qs,Is,R,Ps).
set_vars([Q|Qs],I,[Q|R],P) :-
   set_vars(Qs,I,R,P).

set_var(Det,_-X,_-X,X) :-
   setifiable(Det).

sort_quants([],L,L).
sort_quants([Q|Qs],S,S0) :-
   open_quant(Q,Det,_,_,_),
   split_quants(Det,Qs,A,[],B,[]),
   sort_quants(A,S,[Q|S1]),
   sort_quants(B,S1,S0).

split_quants(_,[],A,A,B,B).
split_quants(Det0,[Quant|Quants],Above,Above0,Below,Below0) :-
   compare_dets(Det0,Quant,Above,Above1,Below,Below1),
   split_quants(Det0,Quants,Above1,Above0,Below1,Below0).

compare_dets(Det0,Q,[quant(Det,X,P,Y)|Above],Above,Below,Below) :-
   open_quant(Q,Det1,X,P,Y),
   governs(Det1,Det0), !,
   bubble(Det0,Det1,Det).
compare_dets(Det0,Q0,Above,Above,[Q|Below],Below) :-
   lower(Det0,Q0,Q).

open_quant(quant(Det,X,P,Y),Det,X,P,Y).

% ============================================================
% Determiner Properties

index_det(index(I),I).
index_det(int_det(I,_),I).

unit_det(set).
unit_det(lambda).
unit_det(quant(_,_)).
unit_det(det(_)).
unit_det(question(_)).
unit_det(id).
unit_det(void).
unit_det(not).
unit_det(generic).
unit_det(int_det(_)).
unit_det(proportion(_)).

det_apply(quant(Det,Type-X,P,_-Y),Q0,Q) :-
```

```
   apply(Det,Type,X,P,Y,Q0,Q).

apply(generic,_,X,P,X,Q,X^(P,Q)).
apply(proportion(Type-V),_,X,P,Y,Q,
      S^(setof(X,P,S),
         N^(numberof(Y,(one_of(S,Y),Q),N),
            M^(card(S,M),ratio(N,M,V))))).
apply(id,_,X,P,X,Q,(P,Q)).
apply(void,_,X,P,X,Q,X^(P,Q)).
apply(set,_,Index:X,P0,S,Q,S^(P,Q)) :-
   apply_set(Index,X,P0,S,P).
apply(int_det(Type-X),Type,X,P,X,Q,(P,Q)).
apply(index(_),_,X,P,X,Q,X^(P,Q)).
apply(quant(Op,N),Type,X,P,X,Q,R) :-
   value(N,Type,Y),
   quant_op(Op,Z,Y,numberof(X,(P,Q),Z),R).
apply(det(Det),_,X,P,Y,Q,R) :-
   apply0(Det,X,P,Y,Q,R).

apply0(Some,X,P,X,Q,X^(P,Q)) :-
   some(Some).
apply0(All,X,P,X,Q,\+X^(P,\+Q)) :-
   all(All).
apply0(no,X,P,X,Q,\+X^(P,Q)).
apply0(notall,X,P,X,Q,X^(P,\+Q)).

quant_op(same,X,X,P,P).
quant_op(Op,X,Y,P,X^(P,F)) :-
   quant_op(Op,X,Y,F).

quant_op(not+more,X,Y,X=<Y).
quant_op(not+less,X,Y,X>=Y).
quant_op(less,X,Y,X<Y).
quant_op(more,X,Y,X>Y).

value(wh(Type-X),Type,X).
value(nb(X),_,X).

all(all).
all(every).
all(each).
all(any).

some(a).
some(the(sin)).
some(some).

apply_set([],X,true:P,S,setof(X,P,S)).
apply_set([I|Is],X,Range:P,S,
      setof([I|Is]:V,(Range,setof(X,P,V)),S)).


governs(Det,set(J)) :-
   index_det(Det,I),
   I \== J.
```

```
governs(Det0,Det) :-
    index_det(Det0,_),
  ( index_det(Det,_);
    Det=det(_);
    Det=quant(_,_)).
governs(_,void).
governs(_,lambda).
governs(_,id).
governs(det(each),question([_|_])).
governs(det(each),det(each)).
governs(det(any),not).
governs(quant(same,wh(_)),Det) :-
    weak(Det).

governs(det(Strong),Det) :-
    strong0(Strong),
    weak(Det).

strong(det(Det)) :-
    strong0(Det).

strong0(each).
strong0(any).

weak(det(Det)) :-
    weak0(Det).
weak(quant(_,_)).
weak(index(_)).
weak(int_det(_,_)).
weak(set(_)).
weak(int_det(_)).
weak(generic).
weak(proportion(_)).

weak0(no).
weak0(a).
weak0(all).
weak0(some).
weak0(every).
weak0(the(sin)).
weak0(notall).

lower(question(_),Q,quant(det(a),X,P,Y)) :-
    open_quant(Q,det(any),X,P,Y), !.
lower(_,Q,Q).

setifiable(generic).
setifiable(det(a)).
setifiable(det(all)).

% ================================================================
% Operators (currently, identity, negation and 'and')

op_apply(id,P,P).
op_apply(not,P,\+P).
```

```
bubble(not,det(any),det(every)) :- !.
bubble(_,D,D).


conj_apply(and,P,Q,(P,Q)).
```

Appendix F

Chat-80 Examples

The following is a slightly edited transcript of a Chat-80 session. The program and database were compiled with the Edinburgh DEC-10 Prolog compiler [Pereira et al. 78], and run on a DEC KL-10 processor.

```
Does Afghanistan border China?

Parse: 14msec.
q
   s
      np
         3+sin
         name(afghanistan)
         []
      verb(border,active,pres+fin,[],pos)
      arg
         dir
         np
            3+sin
            name(china)
            []
      []

Semantics: 26msec.
answer([]) :-
   borders(afghanistan,china)

Planning: 0msec.
answer([]) :-
   {borders(afghanistan,china)}
Yes.

Reply: 0msec.

What is the capital of Upper_volta?

Parse: 31msec.
whq
   _1
```

```
s
   np
      3+sin
      wh(_1)
      []
   verb(be,active,pres+fin,[],...
   arg
      dir
      np
         3+sin
         nucleus
            det(the(sin))
            []
            capital
         pp
            prep(of)
            np
               3+sin
               name(upper_volta)
               []
   []

Semantics: 20msec.
answer([_1]) :-
   capital(upper_volta,_1)

Planning: 0msec.
answer([_1]) :-
   capital(upper_volta,_1)
ouagadougou.

Reply: 12msec.

Where is the largest country?

Parse: 28msec.
whq
   _1
   s
      np
         3+sin
         nucleus
            det(the(sin))
            sup
               most
               adj
                  large
            country
         []
      verb(be,active,pres+fin,[],pos)
      arg
         pred
         pp
            prep(in)
            np
```

```
                _2
                nucleus
                   int_det(_1)
                   []
                   place
                []
      []

Semantics: 36msec.
answer([_1]) :-
   exists _2
     exists _3
       _3 = setof _4:_5
         country(_5)
        & area(_5,_4)
   & aggregate(max,_3,_2)
   & place(_1)
   & in(_2,_1)

Planning: 37msec.
answer([_1]) :-
   exists _2 _3
     _3 = setof _4:_5
       country(_5)
      & area(_5,_4)
   & aggregate(max,_3,_2)
   & in(_2,_1)
   & {place(_1)}
asia and northern_asia.

Reply: 859msec.

Which country's capital is London?

Parse: 20msec.
whq
   _1
   s
      np
         3+sin
         nucleus
            det(the(sin))
            []
            capital
         pp
            poss
            np
               3+sin
               nucleus
                  int_det(_1)
                  []
                  country
               []
      verb(be,active,pres+fin,[],pos)
      arg
```

```
         dir
         np
            3+sin
            name(london)
            []
      []

Semantics: 25msec.
answer([_1]) :-
   country(_1)
 & capital(_1,london)

Planning: 13msec.
answer([_1]) :-
   capital(_1,london)
 & {country(_1)}
united_kingdom.

Reply: 11msec.

Which is the largest African country?

Parse: 16msec.
whq
   _1
   s
      np
         3+sin
         wh(_1)
         []
      verb(be,active,pres+fin,[],pos)
      arg
         dir
         np
            3+sin
            nucleus
               det(the(sin))
               sup
                  most
                  adj
                     large
               adj
                  african
               country
            []
      []

Semantics: 32msec.
answer([_1]) :-
   exists _2
     _2 = setof _3:_4
       country(_4)
     & area(_4,_3)
     & african(_4)
 & aggregate(max,_2,_1)
```

```
Planning: 24msec.
answer([_1]) :-
   exists _2
     _2 = setof _3:_4
       african(_4)
     & {country(_4)}
     & area(_4,_3)
   & aggregate(max,_2,_1)
sudan.

Reply: 217msec.

How large is the smallest American country?

Parse: 32msec.
whq
   _1
   s
      np
         3+sin
         nucleus
            det(the(sin))
            sup
               most
               adj
                  small
            adj
               american
            country
         []
      verb(be,active,pres+fin,[],pos)
      arg
         pred
         value
            adj
               large
            wh(_1)
      []

Semantics: 27msec.
answer([_1]) :-
   exists _2
     exists _3
       _3 = setof _4:_5
         country(_5)
       & area(_5,_4)
       & american(_5)
   & aggregate(min,_3,_2)
   & area(_2,_1)

Planning: 29msec.
answer([_1]) :-
   exists _2 _3
     _3 = setof _4:_5
```

```
      american(_5)
    & {country(_5)}
    & area(_5,_4)
  & aggregate(min,_3,_2)
  & area(_2,_1)
0 ksqmiles.
```

Reply: 135msec.

What is the ocean that borders African countries and that borders Asian countries?

Parse: 47msec.

```
whq
   _1
   s
      np
         3+sin
         wh(_1)
         []
      verb(be,active,pres+fin,[],pos)
      arg
         dir
         np
            3+sin
            nucleus
               det(the(sin))
               []
               ocean
            conj
               and
               rel
                  _2
                  s
                     np
                        3+sin
                        wh(_2)
                        []
                     verb(border,active,pres+fin,[],pos)
                     arg
                        dir
                        np
                           3+plu
                           nucleus
                              generic
                              adj
                                 africa
                              country
                           []
                     []
               rel
                  _3
                  s
                     np
                        3+sin
```

```
                        wh(_3)
                        []
                     verb(border,active,pres+fin,[],pos)
                     arg
                        dir
                        np
                           3+plu
                           nucleus
                              generic
                              adj
                                 asian
                              country
                           []
                     []
      []

Semantics: 49msec.
answer([_1]) :-
   ocean(_1)
 & exists _2
     country(_2)
   & african(_2)
   & borders(_1,_2)
 & exists _3
     country(_3)
   & asian(_3)
   & borders(_1,_3)

Planning: 40msec.
answer([_1]) :-
   exists _2 _3
     ocean(_1)
   & { borders(_1,_2)
     & {african(_2)}
     & {country(_2)} }
   & { borders(_1,_3)
     & {asian(_3)}
     & {country(_3)} }
indian_ocean.

Reply: 95msec.

What are the capitals of the countries bordering the Baltic?

Parse: 40msec.
whq
   _1
   s
      np
         3+plu
         wh(_1)
         []
      verb(be,active,pres+fin,[],pos)
      arg
         dir
```

```
np
   3+plu
   nucleus
      det(the(plu))
      []
      capital
   pp
      prep(of)
      np
         3+plu
         nucleus
            det(the(plu))
            []
            country
         reduced_rel
            _2
            s
               np
                  3+plu
                  wh(_2)
                  []
               verb(border,active,inf,[prog],pos)
               arg
                  dir
                  np
                     3+sin
                     name(baltic)
                     []
               []
[]

Semantics: 38msec.
answer([_1]) :-
   _1 = setof [_2]:_3
      country(_2)
   & borders(_2,baltic)
   & _3 = setof _4
      capital(_2,_4)

Planning: 16msec.
answer([_1]) :-
   _1 = setof [_2]:_3
      borders(_2,baltic)
   & {country(_2)}
   & _3 = setof _4
      capital(_2,_4)

[[denmark]:[copenhagen],
 [east_germany]:[east_berlin],
 [finland]:[helsinki],
 [poland]:[warsaw],
 [soviet_union]:[moscow],
 [sweden]:[stockholm],
 [west_germany]:[bonn]].
```

```
Reply: 83msec.

Which countries are bordered by two seas?

Parse: 31msec.
whq
   _1
   s
      np
         3+plu
         nucleus
            int_det(_1)
            []
            country
         []
      verb(border,passive,pres+fin,[],pos)
      []
      pp
         prep(by)
         np
            3+plu
            nucleus
               quant(same,nb(2))
               []
               sea
            []

Semantics: 10msec.
answer([_1]) :-
   country(_1)
 & 2 = numberof _2
     sea(_2)
   & borders(_2,_1)

Planning: 15msec.
answer([_1]) :-
   2 = numberof _2
     sea(_2)
   & borders(_2,_1)
 & {country(_1)}

egypt, iran, israel, saudi_arabia and turkey.

Reply: 212msec.

How many countries does the Danube flow through?

Parse: 31msec.
whq
   _1
   s
      np
         3+sin
         name(danube)
         []
```

```
      verb(flow,active,pres+fin,[],pos)
      []
      pp
         prep(through)
         np
            3+plu
            nucleus
               quant(same,wh(_1))
               []
               country
            []

Semantics: 12msec.
answer([_1]) :-
   _1 = numberof _2
     country(_2)
   & flows(danube,_2)

Planning: 13msec.
answer([_1]) :-
   _1 = numberof _2
     flows(danube,_2)
   & {country(_2)}
6.

Reply: 12msec.
```

What is the total area of countries south of the Equator and not in Australasia?

```
Parse: 47msec.
whq
   _1
   s
      np
         3+sin
         wh(_1)
         []
      verb(be,active,pres+fin,[],pos)
      arg
         dir
         np
            3+sin
            nucleus
               det(the(sin))
               adj
                  total
               area
            pp
               prep(of)
               np
                  3+plu
                  nucleus
                     generic
                     []
```

```
                  country
               conj
                  and
                  reduced_rel
                     _2
                     s
                        np
                           3+plu
                           wh(_2)
                           []
                        verb(be,active,pres+fin,[],pos)
                        arg
                           pred
                           pp
                              prep(southof)
                              np
                                 3+sin
                                 name(equator)
                                 []
                        []
                  reduced_rel
                     _3
                     s
                        np
                           3+plu
                           wh(_3)
                           []
                        verb(be,active,pres+fin,[],neg)
                        arg
                           pred
                           pp
                              prep(in)
                              np
                                 3+sin
                                 name(australasia)
                                 []
                        []
      []

Semantics: 78msec.
answer([_1]) :-
   exists _2
     _2 = setof _3:[_4]
       area(_4,_3)
     & country(_4)
     & southof(_4,equator)
     & \+in(_4,australasia)
 & aggregate(total,_2,_1)

Planning: 40msec.
answer([_1]) :-
   exists _2
     _2 = setof _3:[_4]
       southof(_4,equator)
     & area(_4,_3)
```

```
     & {country(_4)}
     & {\+in(_4,australasia)}
   & aggregate(total,_2,_1)
10228 ksqmiles.

Reply: 177msec.

What is the average area of the countries in each continent?

Parse: 23msec.
whq
   _1
   s
      np
         3+sin
         wh(_1)
         []
      verb(be,active,pres+fin,[],pos)
      arg
         dir
         np
            3+sin
            nucleus
               det(the(sin))
               adj
                  average
               area
            pp
               prep(of)
               np
                  3+plu
                  nucleus
                     det(the(plu))
                     []
                     country
                  pp
                     prep(in)
                     np
                        3+sin
                        nucleus
                           det(each)
                           []
                           continent
                        []
      []

Semantics: 44msec.
answer([_1,_2]) :-
   continent(_1)
 & exists _3
     _3 = setof _4:[_5]
       area(_5,_4)
     & country(_5)
     & in(_5,_1)
 & aggregate(average,_3,_2)
```

```
Planning: 32msec.
answer([_1,_2]) :-
   exists _3
     continent(_1)
   & _3 = setof _4:[_5]
       in(_5,_1)
     & area(_5,_4)
     & {country(_5)}
   & aggregate(average,_3,_2)
[africa,233--ksqmiles],
[america,496--ksqmiles],
[asia,485--ksqmiles],
[australasia,543--ksqmiles] and [europe,58--ksqmiles].

Reply: 754msec.

Is there more than one country in each continent?

Parse: 29msec.
q
   s
      there
      verb(be,active,pres+fin,[],pos)
      arg
         dir
         np
            3+sin
            nucleus
               quant(more,nb(1))
               []
               country
            pp
               prep(in)
               np
                  3+sin
                  nucleus
                     det(each)
                     []
                     continent
                  []
      []

Semantics: 28msec.
answer([]) :-
   \+
     exists _1
       continent(_1)
     & \+
         exists _2
           _2 = numberof _3
             country(_3)
           & in(_3,_1)
         & _2>1
```

```
Planning: 28msec.
answer([]) :-
   { \+
       exists _1
         continent(_1)
       & { \+
             exists _2
               _2 = numberof _3
                 in(_3,_1)
               & {country(_3)}
             & {_2>1} } }
No.

Reply: 164msec.

Is there some ocean that does not border any country?

Parse: 29msec.
q
   s
      there
      verb(be,active,pres+fin,[],pos)
      arg
         dir
         np
            3+sin
            nucleus
               det(some)
               []
               ocean
            rel
               _1
               s
                  np
                     3+sin
                     wh(_1)
                     []
                  verb(border,active,pres+fin,[],neg)
                  arg
                     dir
                     np
                        3+sin
                        nucleus
                           det(any)
                           []
                           country
                        []
                  []
      []

Semantics: 43msec.
answer([]) :-
   exists _1
     ocean(_1)
   & \+
```

```
       exists _2
         country(_2)
       & borders(_1,_2)

Planning: 14msec.
answer([]) :-
   exists _1
     { ocean(_1)
     & { \+
           exists _2
             borders(_1,_2)
           & {country(_2)} } }
Yes.

Reply: 0msec.

What are the countries from which a river flows into the Black_sea?

Parse: 30msec.
whq
   _1
   s
      np
         3+plu
         wh(_1)
         []
      verb(be,active,pres+fin,[],pos)
      arg
         dir
         np
            3+plu
            nucleus
               det(the(plu))
               []
               country
            rel
               _2
               s
                  np
                     3+sin
                     nucleus
                        det(a)
                        []
                        river
                     []
                  verb(flow,active,pres+fin,[],pos)
                  []
                  pp
                     prep(from)
                     np
                        3+plu
                        wh(_2)
                        []
                  pp
                     prep(into)
```

```
                    np
                       3+sin
                       name(black_sea)
                       []
      []

Semantics: 50msec.
answer([_1]) :-
   _1 = setof _2
     country(_2)
   & exists _3
       river(_3)
     & flows(_3,_2,black_sea)

Planning: 14msec.
answer([_1]) :-
   _1 = setof _2
     exists _3
       flows(_3,_2,black_sea)
     & {country(_2)}
     & {river(_3)}
[romania,soviet_union].

Reply: 27msec.

What are the continents no country in which contains more
than two cities whose population exceeds 1 million ?

Parse: 86msec.
whq
   _1
   s
      np
         3+plu
         wh(_1)
         []
      verb(be,active,pres+fin,[],pos)
      arg
         dir
         np
            3+plu
            nucleus
               det(the(plu))
               []
               continent
            rel
               _2
               s
                  np
                     3+sin
                     nucleus
                        det(no)
                        []
                        country
                     pp
```

```
                    prep(in)
                    np
                       3+plu
                       wh(_2)
                       []
                verb(contain,active,pres+fin,[],pos)
                arg
                   dir
                   np
                      3+plu
                      nucleus
                         quant(more,nb(2))
                         []
                         city
                      rel
                         _3
                         s
                            np
                               3+sin
                               nucleus
                                  det(the(sin))
                                  []
                                  population
                               pp
                                  poss
                                  np
                                     3+plu
                                     wh(_3)
                                     []
                            verb(exceed,active,pres+fin,[],pos)
                            arg
                               dir
                               np
                                  3+sin
                                  nucleus
                                     quant(same,nb(1))
                                     []
                                     million
                                  []
                            []
                []
      []

Semantics: 80msec.
answer([_1]) :-
   _1 = setof _2
     continent(_2)
   & \+
       exists _3
         country(_3)
       & in(_3,_2)
       & exists _4
           _4 = numberof _5
             city(_5)
           & exists _6
```

```
              population(_5,_6)
            & exceeds(_6,1--million)
          & in(_5,_3)
        & _4>2

Planning: 52msec.
answer([_1]) :-
   _1 = setof _2
     continent(_2)
   & { \+
         exists _3 _4
           in(_3,_2)
         & {country(_3)}
         & { _4 = numberof _5
               exists _6
                 in(_5,_3)
               & {city(_5)}
               & { population(_5,_6)
                 & {exceeds(_6,1--million)} }
           & {_4>2} } }
[africa,antarctica,australasia].

Reply: 733msec.

Which country bordering the Mediterranean borders a country
that is bordered by a country whose population exceeds
the population of India?

Parse: 83msec.
whq
   _1
   s
      np
         3+sin
         nucleus
            int_det(_1)
            []
            country
         reduced_rel
            _2
            s
               np
                  3+sin
                  wh(_2)
                  []
               verb(border,active,inf,[prog],pos)
               arg
                  dir
                  np
                     3+sin
                     name(mediterranean)
                     []
               []
      verb(border,active,pres+fin,[],pos)
      arg
```

```
dir
np
   3+sin
   nucleus
      det(a)
      []
      country
   rel
      _3
      s
         np
            3+sin
            wh(_3)
            []
         verb(border,passive,pres+fin,[],pos
         []
         pp
            prep(by)
            np
               3+sin
               nucleus
                  det(a)
                  []
                  country
               rel
                  _4
                  s
                     np
                        3+sin
                        nucleus
                           det(the(sin))
                           []
                           population
                        pp
                           poss
                           np
                              3+sin
                              wh(_4)
                              []
                     verb(exceed,active,pres+fin,[],pos)
                     arg
                        dir
                        np
                           3+sin
                           nucleus
                              det(the(sin))
                              []
                              population
                           pp
                              prep(of)
                              np
                                 3+sin
                                 name(india)
                                 []
                     []
```

```
      []

Semantics: 86msec.
answer([_1]) :-
   country(_1)
 & borders(_1,mediterranean)
 & exists _2
     country(_2)
   & exists _3
       country(_3)
     & exists _4
         population(_3,_4)
       & exists _5
           population(india,_5)
         & exceeds(_4,_5)
     & borders(_3,_2)
   & borders(_1,_2)

Planning: 58msec.
answer([_1]) :-
   exists _2 _3 _4 _5
     population(india,_5)
   & borders(_1,mediterranean)
   & {country(_1)}
   & { borders(_1,_2)
     & {country(_2)}
     & { borders(_3,_2)
       & {country(_3)}
       & { population(_3,_4)
         & {exceeds(_4,_5)} } } }
turkey.

Reply: 195msec.
```

Extraposition Grammars

Fernando Pereira

Department of Architecture
University of Edinburgh
Edinburgh EH1 1JZ SCOTLAND

Extraposition grammars are an extension of definite clause grammars, and are similarly defined in terms of logic clauses. The extended formalism makes it easy to describe left extraposition of constituents, an important feature of natural language syntax.

1. Introduction

This paper presents a grammar formalism for natural language analysis, called *extraposition grammars* (XGs), based on the subset of predicate calculus known as definite, or Horn, clauses. It is argued that certain important linguistic phenomena, collectively known in transformational grammar as *left extraposition,* can be described better in XGs than in earlier grammar formalisms based on definite clauses.

The XG formalism is an extension of the *definite clause grammar* (DCG) [6] formalism, which is itself a restriction of Colmerauer's formalism of *metamorphosis grammars* (MGs) [2]. Thus XGs and MGs may be seen as two alternative extensions of the same basic formalism, DCGs.

The argument for XGs will start with a comparison with DCGs. I should point out, however, that the motivation for the development of XGs came from studying large MGs for natural language [4,7].

The relationship between MGs and DCGs is analogous to that between type-0 grammars and context-free grammars. So, some of the linguistic phenomena which are seen as rewriting one sequence of constituents into another might be described better in a MG than in a DCG. However, it will be shown that rewritings such as the one involved in left extraposition cannot easily be described in either of the two formalisms.

Left extraposition has been used by grammarians to describe the form of interrogative sentences and relative clauses, at least in languages such as English, French, Spanish and Portuguese. The importance of these constructions, even in simplified subsets of natural language, such as those used in database interfaces, suggests that a grammar formalism should be able to express them in a clear and concise manner. This is the purpose of XGs.

2. Grammars in Logic

This section summarises the concepts of *definite clause grammars* (DCGs), and of the underlying system of logic, *definite clauses,* needed for the rest of the paper. A fuller discussion can be found elsewhere [6].

A *definite clause* has either the form

$$P :- Q_1, ..., Q_n .$$

to be read as "P is true if Q_1, ..., Q_n are true", or the form

$$P .$$

to be read as "P is true". P is the *head* of the clause, Q_1, ..., Q_n are *goals*, forming the *body* of the clause. The symbols P, Q_1, ..., Q_n stand for *literals.* A literal has a *predicate symbol,* and possibly some *arguments* (in parentheses, separated by commas), e.g.

```
father(X,Y)   false   number(0)
```

A literal is to be interpreted as denoting a relation between its arguments; e.g. "father(X,Y)" denotes the relation 'father' between X and Y.

Arguments are *terms,* standing for partially specified objects. Terms may be

- *variables,* denoting unspecified objects (variable names are capitalised):

  ```
  X   Case   Agreement
  ```

- *atomic symbols,* denoting specific objects:

  ```
  plural   [ ]   3
  ```

- *compound terms,* denoting complex objects:

  ```
  s(NP,VP)   succ(succ(0))
  ```

A compound term has a *functor* and some arguments, which are terms. Compound terms are best seen as

0362-613X/81/040243-14$01.00

trees, e.g.

A particular type of term, the *list,* has a simplified notation. The binary functor ' . ' makes up non-empty lists, and the atom '[ ]' denotes the empty list. In the special list notation,

```
[a,b]   [X|Y]
```

represent respectively the terms

```
.(a,.(b,[ ])    .(X,Y)
```

Putting these concepts together, the clause

```
grandfather(X,Z) :- father(X,Y), parent(Y,Z).
```

may be read as "X is grandfather of Z if X is father of Y and Y is a parent of Z"; the clause

```
father(john,mary).
```

may be read as "John is father of Mary" (note the use of lower case for the constants in the clause).

A set of definite clauses forms a *program.* A program defines the relations denoted by the predicates appearing on the head of clauses. When using a definite clause interpreter, such as PROLOG [9], a *goal statement*

```
?- P.
```

specifies that the relation instances that match P are required.

Now, any context-free rule, such as

```
sentence --> noun_phrase, verb_phrase.
```

(I use ',' for concatenation, and '.' to terminate a rule) may be translated into a definite clause

```
sentence(S0,S) :- noun_phrase(S0,S1),
                  verb_phrase(S1,S).
```

which says: "there is a sentence between points S0 and S in a string if there is a noun phrase between points S0 and S1, and a verb phrase between points S1 and S". A context-free rule like

```
determiner --> [the].
```

(where the square brackets mark a terminal) can be translated into

```
determiner(S0,S) :- connects(S0,the,S).
```

which may be read as "there is a determiner between points S0 and S in a string if S0 is joined to S by the word 'the'". The predicate 'connects' is used to relate terms denoting points in a string to the words which join those points. Depending on the application, different definitions of 'connects' might be used. In particular, if a point in a string is represented by the list of words after that point, 'connects' has the very simple definition

```
connects([Word|S],Word,S).
```

which may be read as "a string point represented by a list of words with first element Word and rest S is connected by the word Word to the string point represented by list S."

DCGs are the natural extension of context-free grammars (CFGs) obtained through the translation into definite clauses outlined above. A DCG non-terminal may have arguments, of the same form as those of a predicate, and a terminal may be any term. For instance, the rule

```
sentence(s(NP,VP)) --> noun_phrase(NP,N),
                       verb_phrase(VP,N).
```

states: "A sentence with structure

is made of a noun phrase with structure NP and number N (which can be either 'singular' or 'plural'), followed by a verb phrase with structure VP agreeing with the number N". A DCG rule is just "syntactic sugar" for a definite clause. The clause for the example above is

```
sentence(s(NP,VP),S0,S) :-
        noun_phrase(NP,N,S0,S1),
        verb_phrase(VP,N,S1,S).
```

In general, a DCG non-terminal with n arguments is translated into a predicate of $n+2$ arguments, the last two of which are the string points, as in the translation of context-free rules into definite clauses.

The main idea of DCGs is then that grammar symbols can be *general logic terms* rather than just atomic symbols. This makes DCGs a general-purpose grammar formalism, capable of describing any type-0 language. The first grammar formalism with logic terms as grammar symbols was Colmerauer's metamorphosis grammars [2]. Where a DCG is a CFG with logic terms for grammar symbols, a MG is a somewhat restricted type-0 grammar with logic terms for grammar symbols. However, the very simple translation of DCGs into definite clauses presented above does not carry over directly to MGs.

3. Left Extraposition

Roughly speaking, *left extraposition* occurs in a natural language sentence when a subconstituent of some constituent is missing, and some other constituent, to the left of the incomplete one, represents the missing constituent in some way. It is useful to think that an empty constituent, the *trace,* occupies the "hole" left by the missing constituent, and that the constituent to the left, which represents the missing part, is a *marker,* indicating that a constituent to its right contains a trace [1]. One can then say that the constituent in whose place the trace stands has been extraposed to the left, and, in its new position, is rep-

```
sentence --> noun_phrase, verb_phrase.

noun_phrase --> proper_noun.
noun_phrase --> determiner, noun, relative.
noun_phrase --> determiner, noun, prep_phrase.
noun_phrase --> trace.                                  (1)

trace --> [ ].

verb_phrase --> verb, noun_phrase.
verb_phrase --> verb.

relative --> [ ].
relative --> rel_pronoun, sentence.

prep_phrase --> preposition, noun_phrase.
```

Figure 4.1. CFG for relative clauses.

resented by the marker. For instance, relative clauses are formed by a marker, which in the simpler cases is just a relative pronoun, followed by a sentence where some noun phrase has been replaced by a trace. This is represented in the following annotated surface structure:

The man that$_i$ [$_S$John met t_i] is a grammarian.

In this example, t stands for the trace, 'that' is the surface form of the marker, and the connection between the two is indicated by the common index i.

The concept of left extraposition plays an essential role, directly or indirectly, in many formal descriptions of relative and interrogative clauses. Related to this concept, there are several "global constraints", the "island constraints", that have been introduced to restrict the situations in which left extraposition can be applied. For instance, the Ross complex-NP constraint [8], implies that any relative pronoun occurring outside a given noun phrase cannot be bound to a trace occurring inside a relative clause which is a subconstituent of the noun phrase. This means that it is not possible to have a configuration like

$$X_1 \ldots [_{np} \ldots [_{rel}\ X_2\ [_S \ldots t_2 \ldots\ t_1 \ldots]] \ldots]$$

Note that here I use the concept of left extraposition in a loose sense, without relating it to transformations as in transformational grammar. In XGs, and also in other formalisms for describing languages (for instance the context-free rule schemas of Gazdar [5]), the notion of transformation is not used, but a conceptual operation of some kind is required for instance to relate a relative pronoun to a "hole" in the structural representation of the constituent following the pronoun.

4. Limitations of Other Formalisms

To describe a fragment of language where left extraposition occurs, one might start with a CFG which gives a rough approximation of the fragment. The grammar may then be refined by adding arguments to

```
full_sentence --> sentence(nil).

sentence(Hole0) -->
   noun_phrase(Hole0,Hole1), verb_phrase(Hole1).

noun_phrase(Hole,Hole) --> proper_noun.
noun_phrase(Hole,Hole) -->
   determiner, noun, relative.
noun_phrase(Hole0,Hole) -->
   determiner, noun, prep_phrase(Hole0,Hole).
noun_phrase(trace,nil) --> trace.                       (2)

trace --> [ ].

verb_phrase(Hole) -->
   verb, noun_phrase(Hole,nil).
verb_phrase(nil) --> verb.

relative --> [ ].
relative -->
   rel_pronoun, sentence(trace).

prep_phrase(Hole0,Hole) -->
   preposition, noun_phrase(Hole0,Hole).
```

Figure 4.2. DCG for relative clauses.

non-terminals, to carry extraposed constituents across phrases. This method is analogous to the introduction of "derived" rules by Gazdar [5]. Take for example the CFG in Figure 4.1. In this grammar it is possible to use rule (1) to expand a noun phrase into a trace, even outside a relative clause. To prevent this, I will add arguments to all non-terminals from which a noun phrase might be extraposed. The modified grammar, now a DCG, is given in Figure 4.2. A variable 'Hole...' will have the value 'trace' if an extraposed noun phrase occurs somewhere to the right, 'nil' otherwise. The parse tree of Figure 4.3 shows the variable values when the grammar of Figure 4.2 is used to analyse the noun phrase "the man that John met".

Intuitively, we either can see noun phrases moving to the left, leaving traces behind, or traces appearing from markers and moving to the right. In a phrase "noun_phrase(Hole1,Hole2)", Hole1 will have the value 'trace' when a trace occurs somewhere to the right of the left end of the phrase. In that case, Hole2 will be 'nil' if the noun phrase contains the trace, 'trace' if the trace appears to the right of the right end of this noun phrase. Thus, rule (2) in Figure 4.2 specifies that a noun phrase expands into a trace if a trace appears from the left, and as this trace is now placed, it will not be found further to the right.

The non-terminal 'relative' has no arguments, because the complex-NP constraint prevents noun phrases from moving out of a relative clause. However, that constraint does not apply to prepositional phrases, so 'prep_phrase' has arguments. The non-terminal 'sentence' (and consequently 'verb_phrase') has a single argument, because in a relative clause the trace

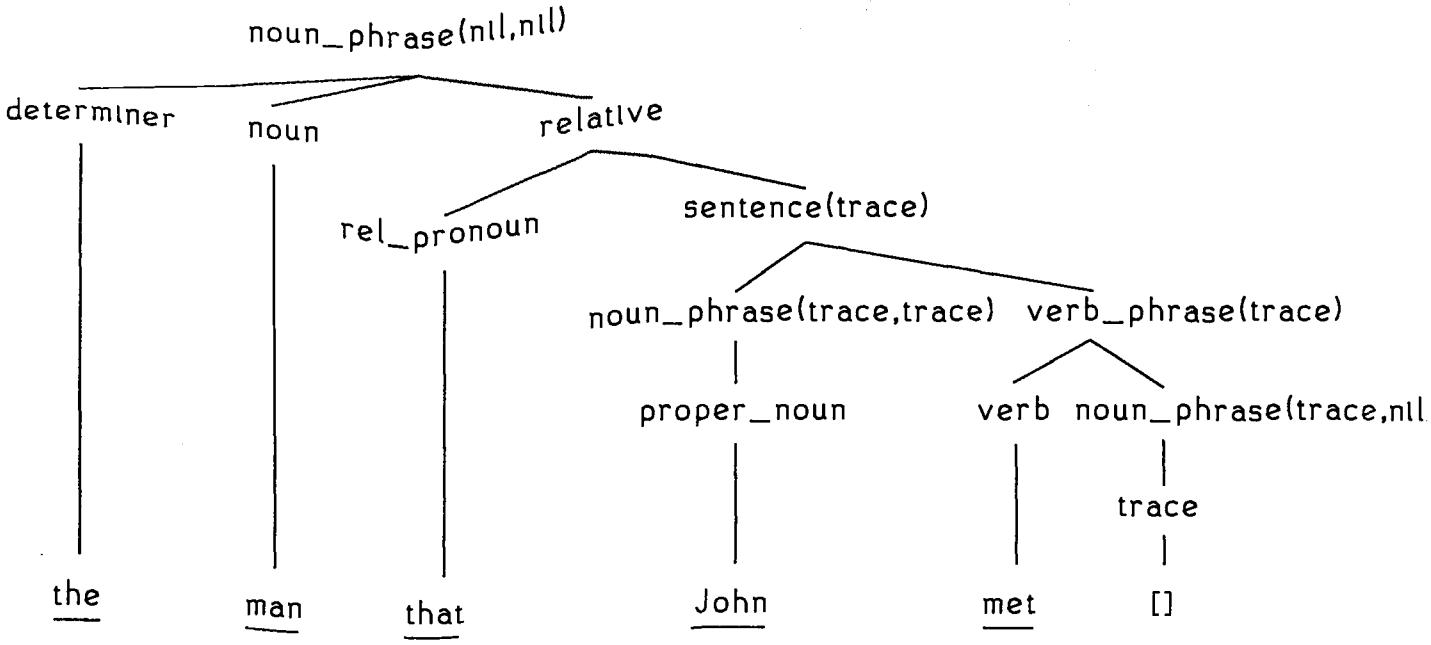


Figure 4.3. DCG parse tree.

must occur in the sentence immediately to the right of the relative pronoun.

It is obvious that in a more extensive grammar, many non-terminals would need extraposition arguments, and the increased complication would make the grammar larger and less readable.

Colmerauer's MG formalism allows an alternative way to express left extraposition. It involves the use of rules whose left-hand side is a non-terminal followed by a string of "dummy" terminal symbols which do not occur in the input vocabulary. An example of such a rule is:

```
rel_marker, [t] --> rel_pronoun.
```

Its meaning is that 'rel_pronoun' can be analysed as a 'rel_marker' provided that the terminal 't' is added to the front of the input remaining after the rule application. Subsequent rule applications will have to cope explicitly with such dummy terminals. This method has been used in several published grammars [2, 4, 7], but in a large grammar it has the same (if not worse) problems of size and clarity as the previous method. It also suffers from a theoretical problem: in general, the language defined by such a grammar will contain extra sentences involving the dummy terminals. For parsing, however, no problem arises, because the input sentences are not supposed to contain dummy terminals. These inadequacies of MGs were the main motivation for the development of XGs.

5. Informal Description of XGs

To describe left extraposition, we need to relate non-contiguous parts of a sentence. But neither DCGs nor MGs have means of representing such a relationship by specific grammar rules. Rather, the relationship can only be described implicitly, by adding extra information to many unrelated rules in the grammar. That is, one cannot look at a grammar and find a set of rules specific to the constructions which involve left extraposition.

With extraposition grammars, I attempt to provide a formalism in which such rules can be written.

In this informal introduction to the XG formalism, I will avoid the extra complications of non-terminal arguments. So, in the discussion that follows, we may look at XGs as an extension of CFGs.

Sometimes it is easier to look at grammar rules in the left-to-right, or synthesis, direction. I will say then that a rule is being used to *expand* or *rewrite* a string. In other cases, it is easier to look at a rule in the right-to-left, or analysis, direction. I will say then that the rule is being used to *analyse* a string.

Let us first look at the following XG fragment:

```
sentence --> noun_phrase, verb_phrase.

noun_phrase --> determiner, noun, relative.
noun_phrase --> trace.

relative --> [ ].
relative --> rel_marker, sentence.

rel_marker ... trace --> rel_pronoun.
```

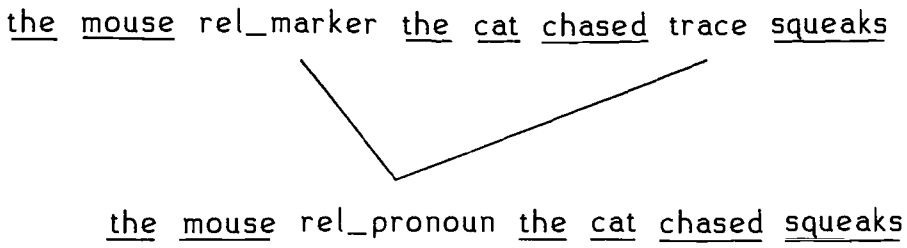


Figure 5.1. Applying an XG rule.

All rules but the last are context-free. The last rule expresses the extraposition in simple relative clauses. It states that a relative pronoun is to be analysed as a marker, followed by some unknown constituents (denoted by '...'), followed by a trace. This is shown in Figure 5.1. As in the DCG example of the previous section, the extraposed noun phrase is expanded into a trace. However, instead of the trace being rewritten into the empty string, the trace is used as part of the analysis of 'rel_marker'.

The difference between XG rules and DCG rules is then that the left-hand side of an XG rule may contain several symbols. Where a DCG rule is seen as expressing the expansion of a single non-terminal into a string, an XG rule is seen as *expanding together* several non-contiguous symbols into a string. More precisely, an XG rule has the general form

$$s_1 \ldots s_2 \quad \text{etc.} \quad s_{k-1} \ldots s_k \quad \text{-->} \quad r. \qquad (3)$$

Here each *segment* s_i (separated from other segments by '...') is a sequence of terminals and non-terminals (written in DCG notation, with ',' for concatenation). The first symbol in s_1, the *leading symbol*, is restricted to be a non-terminal. The right-hand side r is as in a DCG rule.

Leaving aside the constraints discussed in the next section, the meaning of a rule like (3) is that any sequence of symbols of the form

$$s_1 x_1 s_2 x_2 \quad \text{etc.} \quad s_{k-1} x_{k-1} s_k$$

with arbitrary x_i's, can be rewritten into $r x_1 x_2 \ldots x_{k-1}$.

Thinking procedurally, one can say that a non-terminal may be expanded by matching it to the leading symbol on the left-hand side of a rule, and the rest of the left-hand side is "put aside" to wait for the derivation of symbols which match each of its symbols in sequence. This sequence of symbols can be interrupted by arbitrary strings, paired to the occurrences of '...' on the left-hand side of the rule.

6. XG Derivations

When several XG rules are involved, the derivation of a surface string becomes more complicated than in the single rule example of the previous section, because rule applications interact in the way now to be described.

To represent the intermediate stages in an XG derivation, I will use *bracketed strings*, made up of

- terminal symbols
- non-terminal symbols
- the *open bracket* <
- the *close bracket* >

A bracketed string is *balanced* if the brackets in it balance in the usual way.

Now, an XG rule

$$u_1 \ldots u_2 \ldots \quad \text{etc.} \quad \ldots u_n \quad \text{-->} \quad v.$$

can be applied to bracketed string s if

$$s \;=\; x_0 u_1 x_1 u_2 \quad \text{etc.} \quad x_{n-1} u_n x_n$$

and each of the *gaps* x_1, ..., x_{n-1} is balanced. The substring of s between x_0 and x_n is the *span* of the rule application. The application rewrites s into new string t, replacing u_1 by v followed by n-1 open brackets, and replacing each of u_2, ..., u_n by a close bracket; in short, s is replaced by

$$x_0 v << \ \ldots \ <x_1>x_2> \ \ldots \ x_{n-1}>x_n$$

The relation between the original string s and the derived string t is abbreviated as $s \Rightarrow t$. In the new string t, the substring between x_0 and x_n is the *result* of the application. In particular, the application of a rule with a single segment in its left-hand side is no different from what it would be in a type-0 grammar.

Taking again the rule

```
rel_marker ... trace --> rel_pronoun.
```

its application to

```
rel_marker John likes trace
```

produces

```
rel_pronoun < John likes >
```

After this rule application, it is not possible to apply any rule with a segment matching inside a bracketed portion and another segment matching outside it. The use of the above rule has divided the string into two isolated portions, each of which must be independently expanded.

Given an XG with initial symbol s, a sentence t is in the language defined by the XG if there is a se-

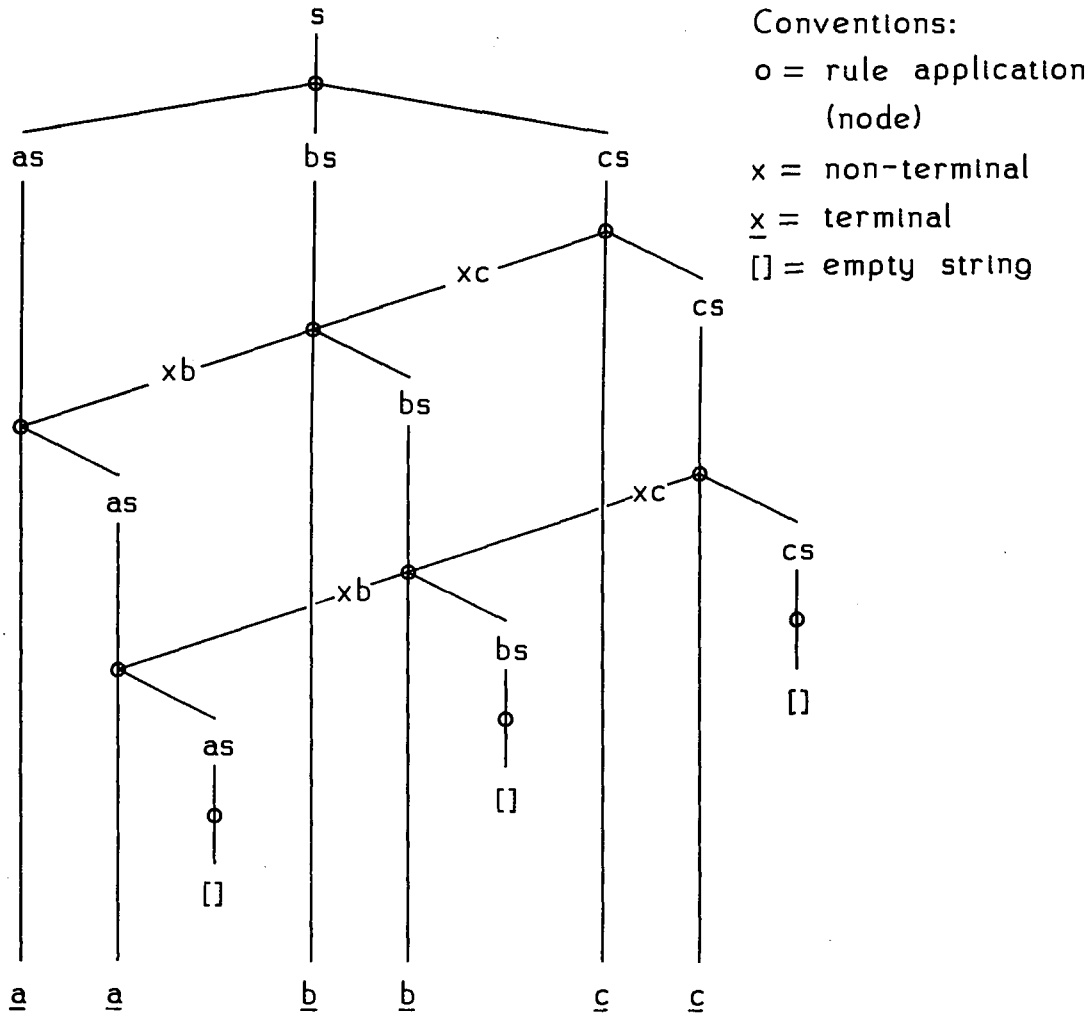


Figure 7.1. Derivation graph for "aabbcc".

quence of rule applications that transforms *s* into a string from which *t* can be obtained by deleting all brackets.

I shall refer to the restrictions on XG rule application which I have just described as the *bracketing constraint*. The effect of the bracketing constraint is independent of the order of application of rules, because if two rules are used in a derivation, the brackets introduced by each of them must be compatible in the way described above. As brackets are added and never deleted, it is clear that the order of application is irrelevant. For similar reasons, any two applications in a derivation where the rules involved have more than one segment in their left-hand sides, one and only one of the two following situations arises:

- the span of neither application intersects the result of the other;
- the result of one of the applications is contained entirely in a gap of the other application – the applications are *nested*.

If one follows to the letter the definitions in this section, then checking, in a parsing procedure, whether an XG rule may be applied, would require a scan of the whole intermediate string. However, we will see in Section 10 that this check may be done "on the fly" as brackets are introduced, with a cost independent of the length of the current intermediate string in the derivation.

7. Derivation Graphs

In the same way as parse trees are used to visualise context-free derivations, I use *derivation graphs* to represent XG derivations.

In a derivation graph, as in a parse tree, each node corresponds to a rule application or to a terminal symbol in the derived sentence, and the edges leaving a node correspond to the symbols in the right-hand side of that node's rule. In a derivation graph, however, a node can have more than one incoming edge – in fact, one such edge for each of the symbols on the left-

hand side of the rule corresponding to that node. Of these edges, only the one corresponding to the leading symbol is used to define the left-to-right order of the symbols in the sentence whose derivation is represented by the graph. If one deletes from a derivation graph all except the first of the incoming edges to every node, the result is a tree analogous to a parse tree.

For example, Figure 7.1 shows the derivation graph for the string "aabbcc" according to the XG:

```
s --> as, bs, cs.

as --> [ ].
as ... xb --> [a], as.

bs --> [ ].
bs ... xc --> xb, [b], bs.

cs --> [ ].
cs --> xc, [c], cs.
```

This XG defines the language formed by the set of all strings

$$a^n b^n c^n \quad \text{for } n \geq 0.$$

The example shows, incidentally, that XGs, even without arguments, are strictly more powerful than CFGs, since the language described is not context-free.

The topology of derivation graphs reflects clearly the bracketing constraint. Assume the following two conventions for the drawing of a derivation graph, which are followed in all the graphs shown here:

- the edges entering a node are ordered clockwise following the sequence of the corresponding symbols in the left-hand side of the rule for that node;
- the edges issuing from a node are ordered counterclockwise following the sequence of the corresponding symbols in the right-hand side of the rule for the node.

Then the derivation graph obeys the bracketing constraint if and only if it can be drawn, following the conventions, without any edges crossing.[1] The example of Figure 7.2 shows this clearly. In this figure, the closed path formed by edges 1, 2, 3, and 4 has the same effect as a matching pair of brackets in a bracketed string.

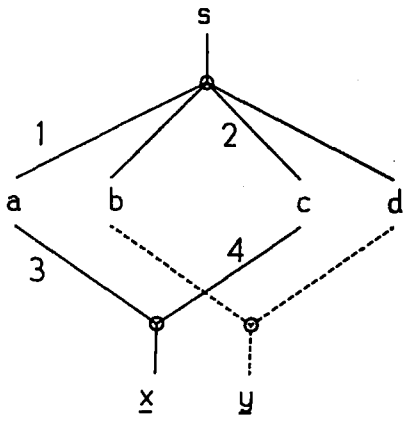
It is also worth noting that nested rule applications appear in a derivation graph as a configuration like the one depicted in Figure 7.3.

8. XGs and Left Extraposition

We saw in Figure 4.2 a DCG for (some) relative clauses. The XG of Figure 8.1 describes essentially the same language fragment, showing how easy it is to describe left extraposition in an XG. In that grammar, the sentence

[1] In some of the examples of this article, edges cross to make the graphs more readable, but such crossings could be trivially avoided.

```
s --> a, b, c, d.
a ... c --> [x].
b ... d --> [y].
```




```
s => a b c d => x < b > d => ? (blocks)
s => a b c d => a y < c > => ?
```

Figure 7.2. Relating derivations to derivation graphs.

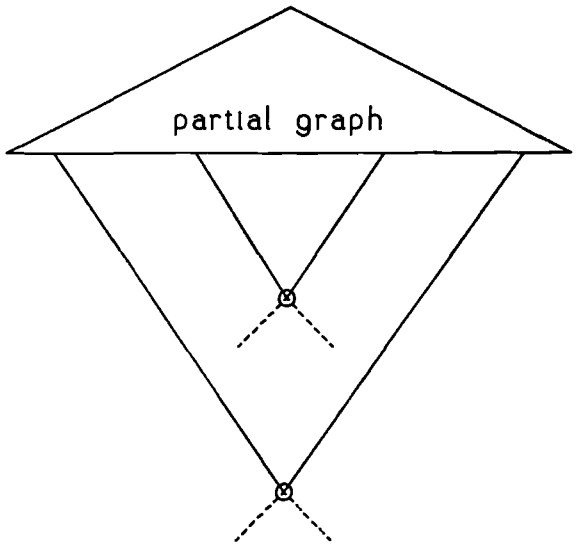


Figure 7.3. Nested rule applications.

```
The mouse that the cat chased squeaks.
```

has the derivation graph shown in Figure 8.2. The left extraposition implicit in the structure of the sentence is represented in the derivation graph by the applica-

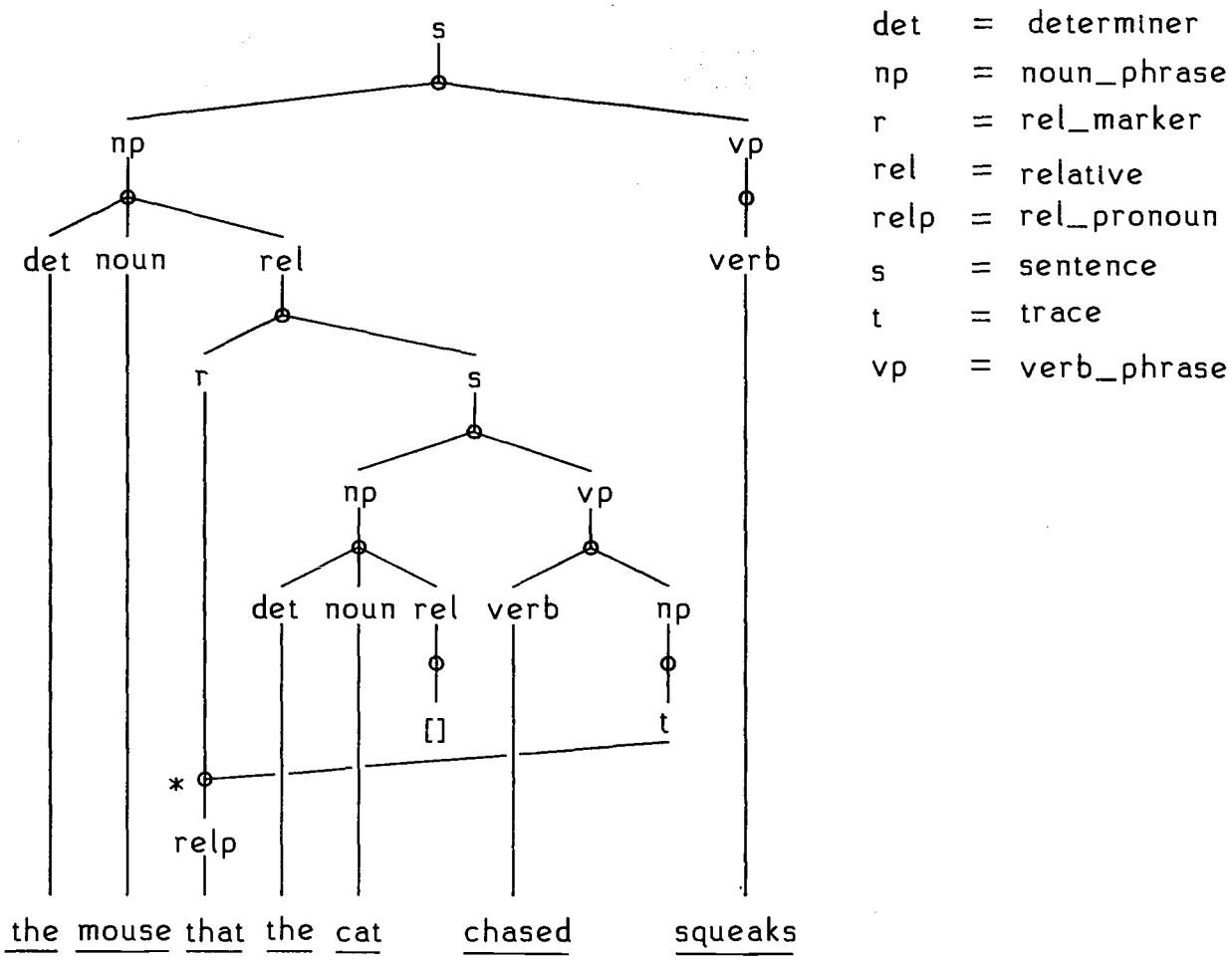


Figure 8.2. Example of derivation graph for the XG in Figure 8.1.

```
sentence --> noun_phrase, verb_phrase.

noun_phrase --> proper_noun.
noun_phrase --> determiner, noun, relative.
noun_phrase --> determiner, noun, prep_phrase.
noun_phrase --> trace.

verb_phrase --> verb, noun_phrase.
verb_phrase --> verb.

relative --> [ ].
relative --> rel_marker, sentence.                    (4)

rel_marker ... trace --> rel_pronoun.

prep_phrase --> preposition, noun_phrase.
```

Figure 8.1. XG for relative clauses.

tion of the rule for 'rel_marker', at the node marked (*) in the figure. One can say that the left extraposition has been "reversed" in the derivation by the use of this rule, which may be looked at as *repositioning* 'trace' to the right, thus "reversing" the extraposition of the original sentence.

In the rest of this paper, I often refer to a constituent being *repositioned into* a bracketed string (or into a fragment of derivation graph), to mean that a rule having that constituent as a non-leading symbol in the left-hand side has been applied, and the symbol matches some symbol in the string (or corresponds to some edge in the fragment). For example, in Figure 8.2 the trace 't' is repositioned into the subgraph with root 's'.

9. Using the Bracketing Constraint

In the example of Figure 8.2, there is only one application of a non-DCG rule, at the place marked (*). However, we have seen that when a derivation contains several applications of such rules, the applications must obey the bracketing constraint. The use of the constraint in a grammar is better explained with an example. From the sentences

```
The mouse squeaks.
The cat likes fish.
The cat chased the mouse.
```

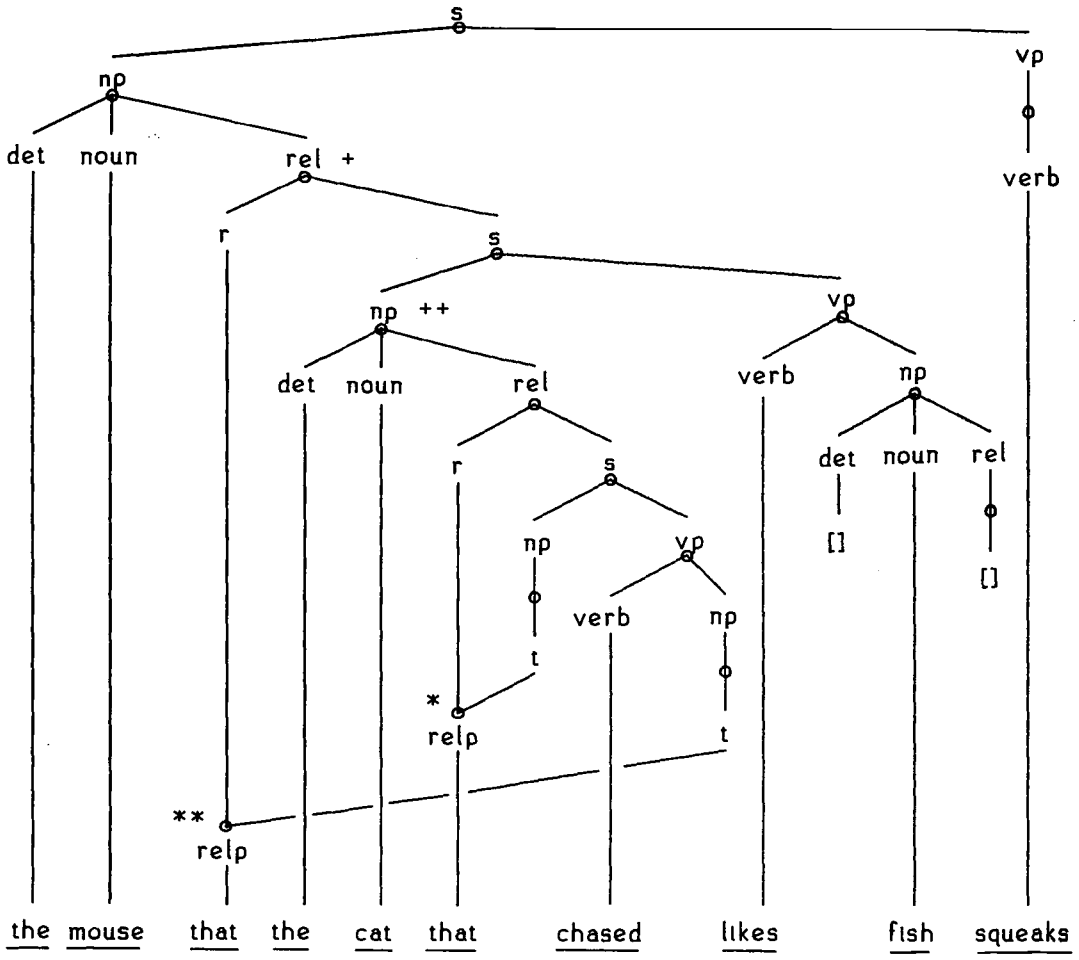


Figure 9.1. Violation of the complex-NP constraint.

the grammar of Figure 8.1 can derive the following string, which violates the complex-NP constraint:

```
* The mouse that the cat that chased likes fish squeaks.
```

The derivation of this ungrammatical string can be better understood if we compare it with a sentence outside the fragment:

```
The mouse, that the cat which chased it likes fish,
        squeaks.
```

where the pronoun 'it' takes the place of the incorrect trace.

The derivation graph for that un-English string is shown in Figure 9.1. In the graph, (*) and (**) mark two nested applications of the rule for 'rel_marker'. The string is un-English because the higher 'relative' (marked (+) in the graph) binds a trace occurring inside a sentence which is part of the subordinated 'noun_phrase' (++).

Now, using the bracketing constraint one can neatly express the complex-NP constraint. It is only necessary to change the second rule for 'relative' in Figure 8.1 to

```
relative --> open, rel_marker, sentence, close.    (5)
```

and add the rule

```
open ... close --> [ ].                            (6)
```

With this modified grammar, it is no longer possible to violate the complex-NP constraint, because no constituent can be repositioned from outside into the gap created by the application of rule (6) to the result of applying the rule for relatives (5).

The non-terminals 'open' and 'close' bracket a subderivation

```
... open X close ... => < X > ...
```

preventing any constituent from being repositioned from outside that subderivation into it. Figure 9.2 shows the use of rule (6) in the derivation of the sentence

```
The mouse that the cat that likes fish chased squeaks.
```

This is based on the same three simple sentences as the ungrammatical string of Figure 9.1, which the

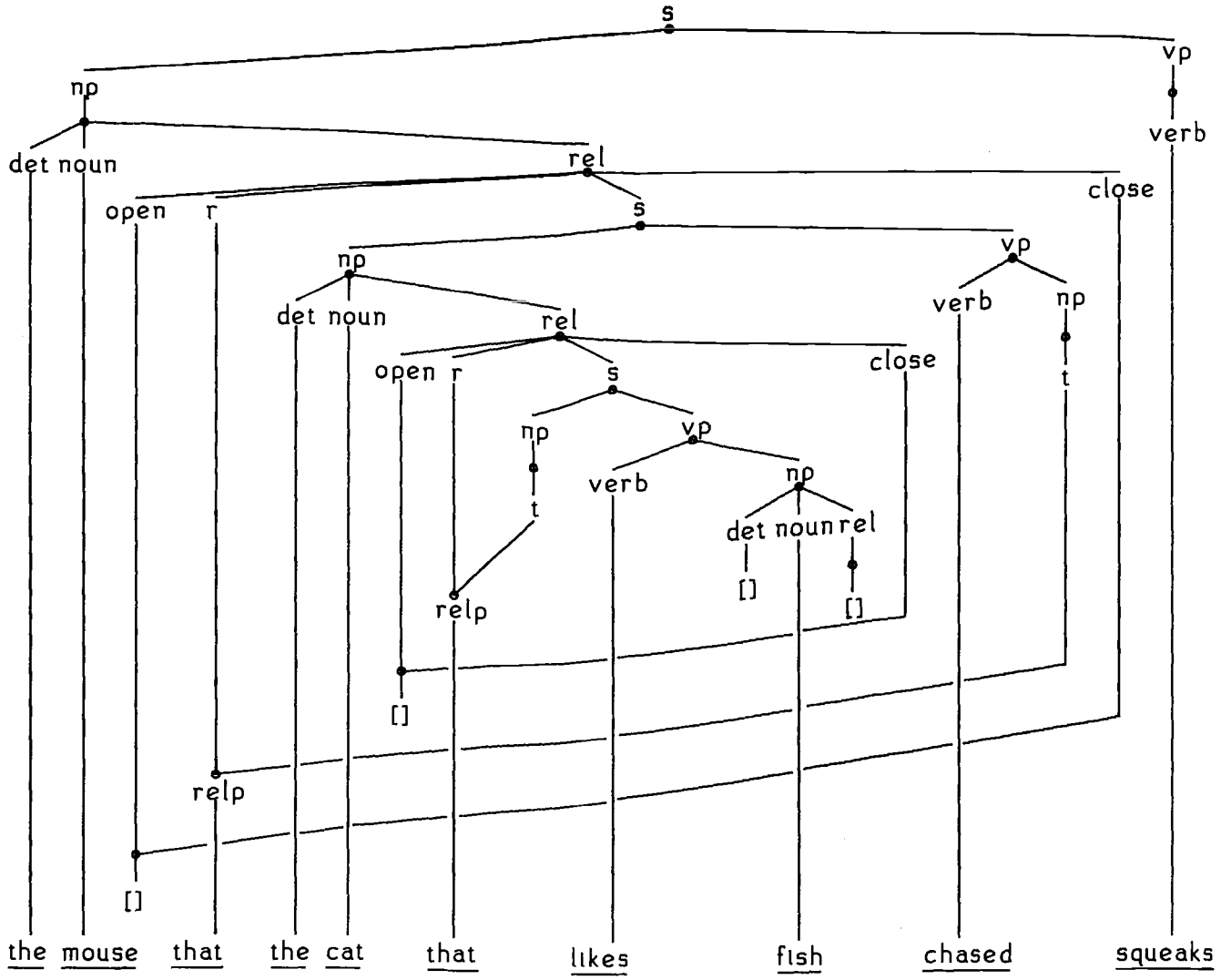


Figure 9.2. Implementation of the complex-NP constraint.

reader can now try to derive in the modified grammar, to see how the bracketing constraint prevents the derivation.

10. XGs as Logic Programs

In the previous sections, I avoided the complication of non-terminal arguments. Although it would be possible to describe fully the operation of XGs in terms of derivations on bracketed strings, it is much simpler to complete the explanation of XGs using the translation of XG rules into definite clauses. In fact, a rigorous definition of XGs independently of definite clauses would require a formal apparatus very similar to the one needed to formalise definite clause programs in the first place, and so it would fall outside the scope of the present paper. The interested reader will find a full discussion of those issues in two articles by Colmerauer [2,3].

Like a DCG, a general XG is no more than a convenient notation for a set of definite clauses. An XG non-terminal of arity n corresponds to an $n+4$ place predicate (with the same name). Of the extra four arguments, two are used to represent string positions as in DCGs, and the other two are used to represent positions in an *extraposition list,* which carries symbols to be repositioned.

Each element of the extraposition list represents a symbol being repositioned as a 4-tuple

x(*context, type, symbol, xlist*)

where *context* is either 'gap', if the symbol was preceded by '...' in the rule where it originated, or 'nogap', if the symbol was preceded by ','; *type* may be 'terminal' or 'nonterminal', with the obvious meaning; *symbol* is the symbol proper; *xlist* is the remainder of the extraposition list (an empty list being represented by '[ ]').

An XG rule is translated into a clause for the predicate corresponding to the leading symbol of the rule. In the case where the XG rule has just a single symbol on the left-hand side, the translation is very similar to that of DCG rules. For example, the rule

```
sentence --> noun_phrase, verb_phrase.
```

translates into

```
sentence(S0,S,X0,X) :-
        noun_phrase(S0,S1,X0,X1),
        verb_phrase(S1,S,X1,X).
```

A terminal *t* in the right-hand side of a rule translates into a call to the predicate 'terminal', defined below, whose role is analogous to that of 'connects' in DCGs. For example, the rule

```
rel_pronoun -->  [that] .
```

translates into

```
rel_pronoun(S0,S,X0,X) :-
                terminal(that,S0,S,X0,X).
```

The translation of a rule with more than one symbol in the left-hand side is a bit more complicated. Informally, each symbol after the first is made into a 4-tuple as described above, and fronted to the extraposition list. Thus, for example, the rule

```
rel_marker ... trace --> rel_pronoun.
```

translates into

```
rel_marker(S0,S,X0,x(gap,nonterminal,trace,X)) :-
            rel_pronoun(S0,S,X0,X).
```

Furthermore, for each distinct non-leading non-terminal *nt* (with arity *n*) in the left-hand side of a rule of the XG, the translation includes the clause

```
nt(V1,...,Vn,S,S,X0,X) :-
            virtual(nt(V1,...,Vn),X0,X).
```

where 'virtual(C,X0,X)', defined later, can be read as "C is the constituent between X0 and X in the extraposition list", and the variables V*i* transfer the arguments of the symbol in the extraposition list to the predicate which translates that symbol.

For example, the rule

```
marker(Var), [the] ... [of.whom], trace(Var) -->
        [whose].
```

which can be used in a more complex grammar of relative clauses to transform "whose *X*" into "the *X* of whom", corresponds to the clauses:

```
marker(Var,S0,S,X0,
   x(nogap,terminal,the,
   x(gap,terminal,of,
   x(nogap,terminal,whom,
   x(nogap,nonterminal,trace(Var),
   X )))) ) :-
      terminal(whose,S0,S,X0,X).

trace(Var,S,S,X0,X) :- virtual(trace(Var),X0,X).
```

Finally, the two auxiliary predicates 'virtual' and 'terminal' are defined as follows:-

```
virtual(NT, x(C,nonterminal,NT,X), X).

terminal(T, S0, S, X, X) :-
   gap(X), connects(S0, T, S).
terminal(T, S, S, x(C,terminal,T,X), X).

gap(x(gap,T,S,X)).
gap([ ]).
```

where 'connects' is as for DCGs.

These definitions need some comment. The first clause for 'terminal' says that, provided the current extraposition list allows a gap to appear in the derivation, terminal symbol T may be taken from the position S0 in the source string, where T connects S0 to some new position S. The second clause for 'terminal' says that if the next symbol in the current extraposition list is a terminal T, then this symbol can be taken as if it occurred at S in the source string. The clause for 'virtual' allows a non-terminal to be "read off from" the extraposition list.

```
* relative(6,9,X,X)
*  open(6,6,x(gap,nt,trace,x(gap,nt,close,[])),
       x(gap,nt,close,x(gap,nt,trace,
       x(gap,nt,close,[]))))
*  rel_marker(6,7,x(gap,nt,close,x(gap,nt,trace,
       x(gap,nt,close,[]))),
       x(gap,nt,trace,x(gap,nt,close,
       x(gap,nt,trace,x(gap,nt,close,[])))))
*   rel_pronoun(6,7,X,X)
     [that]
*  sentence(7,9,x(gap,nt,trace,x(gap,nt,close,
       x(gap,nt,trace,x(gap,nt,close,[]))))),
       x(gap,nt,close,x(gap,nt,trace,
       x(gap,nt,close,[]))))
*   noun_phrase(7,7,x(gap,nt,trace,x(gap,nt,close,
        x(gap,nt,trace,x(gap,nt,close,[])))),
        x(gap,nt,close,x(gap,nt,trace,
        x(gap,nt,close,[]))))
*    trace(7,7,x(gap,nt,trace,x(gap,nt,close,
         x(gap,nt,trace,x(gap,nt,close,[])))),
         x(gap,nt,close,x(gap,nt,trace,
         x(gap,nt,close,[]))))
*   verb_phrase(7,9,X,X)
*    verb(7,8,X,X)
      [likes]
*    noun_phrase(8,9,X,X)
*     determiner(8,8,X,X)
*     noun(8,9,X,X)
       [fish]
*     relative(9,9,X,X)
*  close(9,9,x(gap,nt,close,x(gap,nt,trace,
       x(gap,nt,close,[]))),
       x(gap,nt,trace,x(gap,nt,close,[])))
```

Figure 10.1. Derivation of "that likes fish".

Figure 10.1 shows a fragment of the analysis in Figure 9.2, but now in terms of the translation of XG rules into definite clauses. Points on the sentence are labelled as follows:

```
the mouse that the cat that likes fish chased  squeaks
1   2     3    4   5   6    7     8     9      10      11
```

The nodes of the analysis fragment, for the relative clause "that likes fish", are represented by the corresponding goals, indented in proportion to their distance from the root of the graph. The following conventions are used to simplify the figure:

- The leaves (terminals) of the graph are listed directly;
- the values of the extraposition arguments are explictly represented only for those goals that add or delete something to the extraposition list; for the other goals, the two identical values are represented by the variable 'X';
- the goals for 'terminal' and 'virtual' are left out as they can be easily reconstructed from the other goals and the definitions above;
- 'nonterminal' is abbreviated as 'nt'.

The definite clause program corresponding to the grammar for this example is listed in Appendix II.

The example shows clearly how the bracketing constraint works. Symbols are placed in the extraposition list by rules with more than one symbol in the left-hand side, and removed by calls to 'virtual', on a first-in-last-out basis; that is, the extraposition list is a *stack*. But this property of the extraposition list is exactly what is needed to balance "on the fly" the auxiliary brackets in the intermediate steps of a derivation.

Being no more than a logic program, an XG can be used for analysis and for synthesis in the same way as a DCG. For instance, to determine whether a string *s* with initial point *initial* and final point *final* is in the language defined by the XG of Figure 8.1, one tries to prove the goal statement

```
?- sentence(initial,final,[ ],[ ]).
```

As for DCGs, the string *s* can be represented in several ways. If it is represented as a list, the above goal would be written

```
?- sentence(s,[ ],[ ],[ ]).
```

The last two arguments of the goal are '[ ]' to mean that the overall extraposition list goes from '[ ]' to '[ ]'; i.e., it is the empty list. Thus, no constituent can be repositioned into or out of the top level 'sentence'.

11. Conclusions and Further Work

In this paper I have proposed an extension of DCGs. The motivation for this extension was to provide a simple formal device to describe the structure of such important natural language constructions as relative clauses and interrogative sentences. In transformational grammar, these constructions have usually been analysed in terms of left extraposition, together with global constraints, such as the complex-NP constraint, which restrict the range of the extraposition. Global constraints are not explicit in the grammar rules, but are given externally to be enforced across rule applications. These external global constraints cause theoretical difficulties, because the formal properties of the resulting systems are far from evident, and practical difficulties, because they lead to obscure grammars and prevent the use of any reasonable parsing algorithm.

DCGs, although they provide the basic machinery for a clear description of languages and their structures, lack a mechanism to describe simply left extraposition and the associated restrictions. MGs can express the rewrite of several symbols in a single rule, but the symbols must be contiguous, as in a type-0 grammar rule. This is still not enough to describe left extraposition without complicating the rest of the grammar. XGs are an answer to those limitations.

An XG has the same fundamental property as a DCG, that it is no more than a convenient notation for the clauses of an ordinary logic program. XGs and their translation into definite clauses have been designed to meet three requirements: (i) to be a principled extension of DCGs, which can be interpreted as a grammar formalism independently of its translation into definite clauses; (ii) to provide for simple description of left extraposition and related restrictions; (iii) to be comparable in efficiency with DCGs when executed by PROLOG. It turns out that these requirements are not contradictory, and that the resulting design is extremely simple. The restrictions on extraposition are naturally expressed in terms of scope, and scope is expressed in the formalism by "bracketing out" subderivations corresponding to balanced strings. The notion of bracketed string derivation is introduced in order to describe extraposition and bracketing independently of the translation of XGs into logic programs.

Some questions about XGs have not been tackled in this paper. First, from a theoretical point of view it would be necessary to complete the independent characterisation of XGs in terms of bracketed strings, and show rigorously that the translation of XGs into logic programs correctly renders this independent characterisation of the semantics of XGs. As pointed out before, this formalisation does not offer any substantial problems.

Next, it is not clear whether XGs are as general as they could be. For instance, it might be possible to extend them to handle *right* extraposition of constituents, which, although less common than left extraposition, can be used to describe quite frequent English constructions, such as the gap between head noun and relative clause in:

```
What files are there that were created today?
```

It may however be possible to describe such situations in terms of left extraposition of some other constituent (e.g. the verb phrase "are there" in the example above).

Finally, I have been looking at what transformations should be applied to an XG developed as a clear description of a language, so that the resulting grammar could be used more efficiently in parsing. In particular, I have been trying to generalise results on deterministic parsing of context-free languages into appropriate principles of transformation.

Acknowledgements

David Warren and Michael McCord read drafts of this paper, and their comments led to many improvements, both in content and in form. The comments of the referees were also very useful. A British Council Fellowship partly supported my work in this subject. The computing facilities I used to experiment with XGs and to prepare this paper were made available by British Science Research Council grants.

Appendix I. Translating XGs

The following PROLOG program (for the DEC-10 PROLOG system) defines a predicate 'grammar(File)' which translates and stores the XG rules contained in File. The symbol '__' as a predicate or functor argument denotes an "anonymous" variable, i.e. each such occurrence stands for a separate variable with a single occurrence.

```
% Definition of the grammar rule operators

:- op(1001,xfy,(...)).
:- op(1200,xfx,(-->)).

% Process the XG in File

grammar(File) :-
   seeing(Old),
   see(File),
   consume,
   seen,
   see(Old).

% Loop until end_of_file

consume :-
   repeat,
      read(X),
    ( X=end_of_file, !;
      process(X),
         fail ).

% Process a grammar rule

process((L-->R)) :- !,
   expandlhs(L,S0,S,H0,H,P),
   expandrhs(R,S0,S,H0,H,Q),
   assertz((P :- Q)), !.

% Execute a command

process(( :- G)) :- !,
   G.

% Store a normal clause

process((P :- Q)) :-
   assertz((P :- Q)).

% Store a unit clause

process(P) :-
   assertz(P).

% Translate an XG rule
% Translate the left-hand side

expandlhs(T,S0,S,H0,H1,Q) :-
   flatten(T,[P|L],[]),
   front(L,H1,H),
   tag(P,S0,S,H0,H,Q).

flatten((X...Y),L0,L) :- !,
   flatten(X,L0,[gap|L1]),
   flatten(Y,L1,L).
flatten((X,Y),L0,L) :- !,
   flatten(X,L0,[nogap|L1]),
   flatten(Y,L1,L).
flatten(X,[X|L],L).

front([],H,H).
front([K,X|L],H0,H)
   case(X,K,H1,H),
   front(L,H0,H1).

case([T|Ts],K,H0,x(K,terminal,T,H)) :- !,
   unwind(Ts,H0,H).
case(Nt,K,H,x(K,nonterminal,Nt,H)) :-
   virtual_rule(Nt).

% Create the clause
%    Nt(S,S,X0,X) :- virtual(Nt,X0,X)
% for extraposed symbol Nt

virtual_rule(Nt) :-
   functor(Nt,F,N),
   functor(Y,F,N),
   tag(Y,S,S,Hx,Hy,P),
 ( clause(P,virtual(_,_,_),_), !;
   asserta((P :- virtual(Y,Hx,Hy))) ).

% Translate the right-hand side

expandrhs((X1,X2),S0,S,H0,H,Y) :- !,
   expandrhs(X1,S0,S1,H0,H1,Y1),
   expandrhs(X2,S1,S,H1,H,Y2),
   and(Y1,Y2,Y).
expandrhs((X1;X2),S0,S,H0,H,(Y1;Y2)) :- !,
   expandor(X1,S0,S,H0,H,Y1),
   expandor(X2,S0,S,H0,H,Y2).
expandrhs({X},S,S,H,H,X) :- .
expandrhs(L,S0,S,H0,H,G) :- islist(L), !,
   expandlist(L,S0,S,H0,H,G).
expandrhs(X,S0,S,H0,H,Y) :-
   tag(X,S0,S,H0,H,Y).

expandor(X,S0,S,H0,H,Y) :-
   expandrhs(X,S0a,S,H0a,H,Ya),
 ( S\==S0a, !, S0=S0a, Yb=Ya; and(S0=S0a,Ya,Yb) ),
 ( H\==H0a, !, H0=H0a, Y=Yb; and(H0=H0a,Yb,Y) ).

expandlist([],S,S,H,H,true).
expandlist([X],S0,S,H0,H,terminal(X,S0,S,H0,H) ) :- !.
expandlist([X|L],S0,S,H0,H,
```

```
      (terminal(X,S0,S1,H0,H1),Y)) :-
   expandlist(L,S1,S,H1,H,Y).

tag(P,A1,A2,A3,A4,Q) :-
   P=..[F|Args0],
   conc(Args0,[A1,A2,A3,A4],Args),
   Q=..[F|Args].

and(true,P,P) :- !.
and(P,true,P) :- !.
and(P,Q,(P,Q)).

islist([_|_]).
islist([]).

unwind([],H,H) :- !.
unwind([T|Ts],H0,x(nogap,terminal,T,H)) :-
   unwind(Ts,H0,H).

conc([],L,L) :- !.
conc([X|L1],L2,[X|L3]) :-
   conc(L1,L2,L3).
```

Appendix II. Definite clauses for the grammar used in Figure 9.2

```
sentence(S0,S,X0,X) :-
   noun_phrase(S0,S1,X0,X1),
   verb_phrase(S1,S,X1,X).

noun_phrase(S0,S,X0,X) :-
   proper_noun(S0,S,X0,X).
noun_phrase(S0,S,X0,X) :-
   determiner(S0,S1,X0,X1),
   noun(S1,S2,X1,X2),
   relative(S2,S,X2,X).
noun_phrase(S0,S,X0,X) :-
   determiner(S0,S1,X0,X1),
   noun(S1,S2,X1,X2),
   prep_phrase(S2,S,X2,X).
noun_phrase(S0,S,X0,X) :-
   trace(S0,S,X0,X).

verb_phrase(S0,S,X0,X) :-
   verb(S0,S1,X0,X1),
   noun_phrase(S1,S,X1,X).
verb_phrase(S0,S,X0,X) :-
   verb(S0,S,X0,X).

relative(S0,S0,X,X).
relative(S0,S,X0,X) :-
   open(S0,S1,X0,X1),
   rel_marker(S1,S2,X1,X2),
   sentence(S2,S3,X2,X3),
   close(S3,S,X3,X).

trace(S0,S0,X0,X) :-
   virtual(trace,X0,X).

rel_marker(S0,S,X0,x(gap,nonterminal,trace,X)) :-
   rel_pronoun(S0,S,X0,X).

prep_phrase(S0,S,X0,X) :-
   preposition(S0,S1,X0,X1),
   noun_phrase(S1,S,X1,X).

open(S0,S0,X,x(gap,nonterminal,close,X)).

close(S0,S0,X0,X) :-
   virtual(close,X0,X).
```

References

1. Chomsky, N. *Reflections on Language.* Pantheon, 1975.
2. Colmerauer, A. "Metamorphosis Grammars." In *Natural Language Communication with Computers,* L .Bolc (ed.). Springer-Verlag, 1978. First appeared as an internal report, 'Les Grammaires de Metamorphose', in November 1975
3. Colmerauer, A. "Les Bases Théoriques de PROLOG." Groupe d'Intelligence Artificielle, U. E. R. de Luminy, Université d'Aix-Marseille II, 1979.
4. Dahl, V. "Un Système Déductif d'Interrogation de Banques de Données en Espagnol." Groupe d'Intelligence Artificielle, U. E. R. de Luminy, Université d'Aix-Marseille II, 1977.
5. Gazdar, G. "English as a Context-Free Language." School of Social Sciences, University of Sussex, April, 1979.
6. Pereira, F. and Warren, D. H. D. "Definite Clause Grammars for Language Analysis – A Survey of the Formalism and a Comparison with Augmented Transition Networks." *Artificial Intelligence* 13 (1980) 231-278.
7. Pique, J. F. "Interrogation en Francais d'une Base de Données Relationnelle." Groupe d'Intelligence Artificielle, U. E. R. de Luminy, Université d'Aix-Marseille II, 1978.
8. Ross, J. R. Excerpts from 'Constraints on Variables in Syntax'. In G. Harman (ed.): *On Noam Chomsky: Critical Essays,* Anchor Books, 1974.
9. Roussel, P. "PROLOG : Manuel de Réference et Utilisation." Groupe d'Intelligence Artificielle, U.E.R. de Luminy, Université d'Aix-Marseille II, 1975.

Fernando C.N. Pereira is a research associate in the Department of Architecture at Edinburgh University, and also a graduate student in the Department of Artificial Intelligence. He received the M.Sc. degree in mathematics from Lisbon University in 1975.