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.

```
      s                    succ
     / \                    |
    NP  VP                 succ
                            |
                            0
```

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

```
      s
     / \
    NP  VP
```

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 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".

```
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.

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

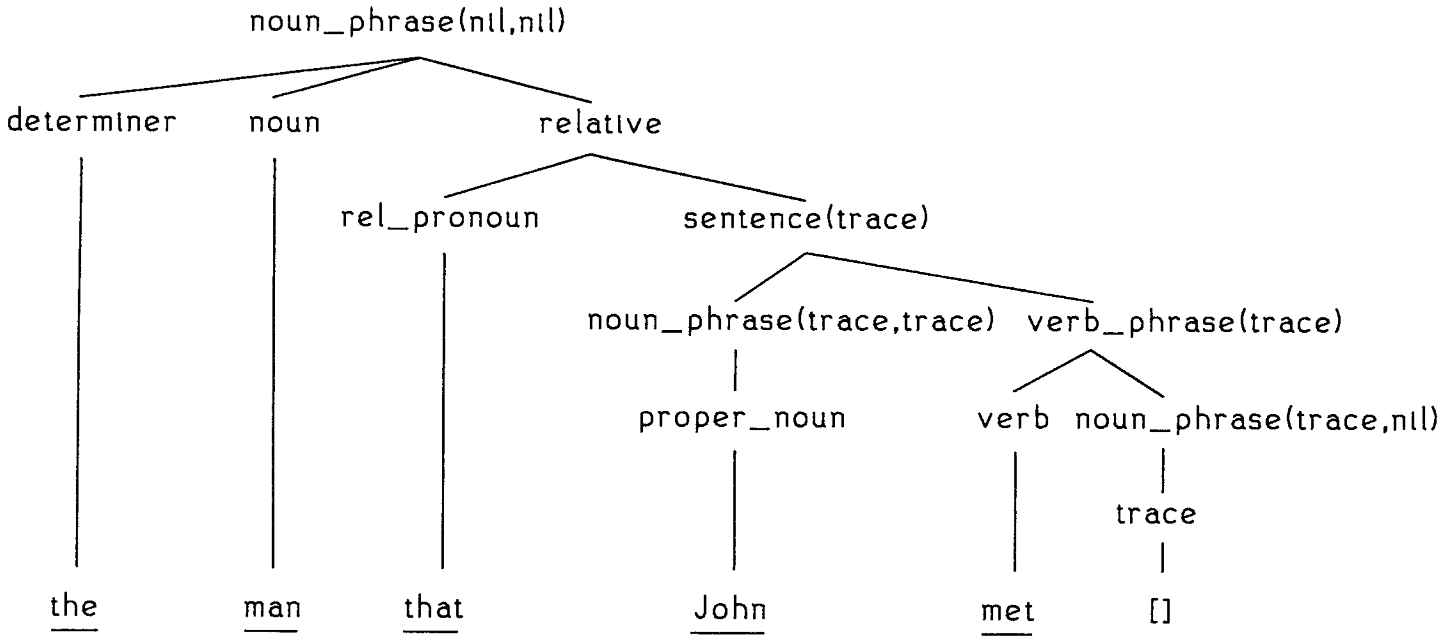


Figure 4.3. DCG parse tree.

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.
```

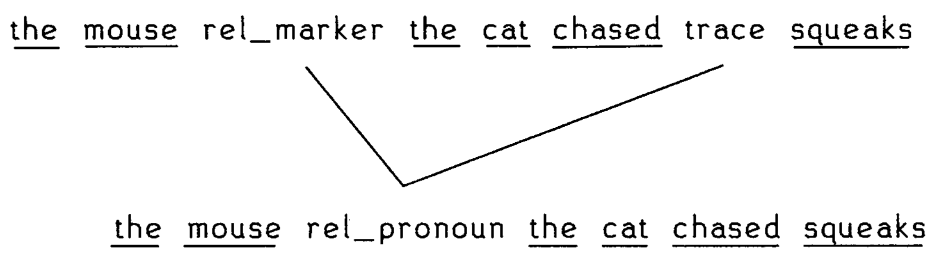

the mouse rel_marker the cat chased trace squeaks
the mouse rel_pronoun the cat chased squeaks

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

$$s_1 \ldots s_2 \text{ etc. } s_{k-1} \ldots s_k \text{ --> } r. \qquad (3)$$

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

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

with arbitrary x_i's, can be rewritten into $r x_1 x_2 \ldots x_{k-1}$.

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

$$u_1 \ldots u_2 \ldots \text{ etc. } \ldots u_n \text{ --> } v.$$

can be applied to bracketed string s if

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

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 $s => t$. 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-

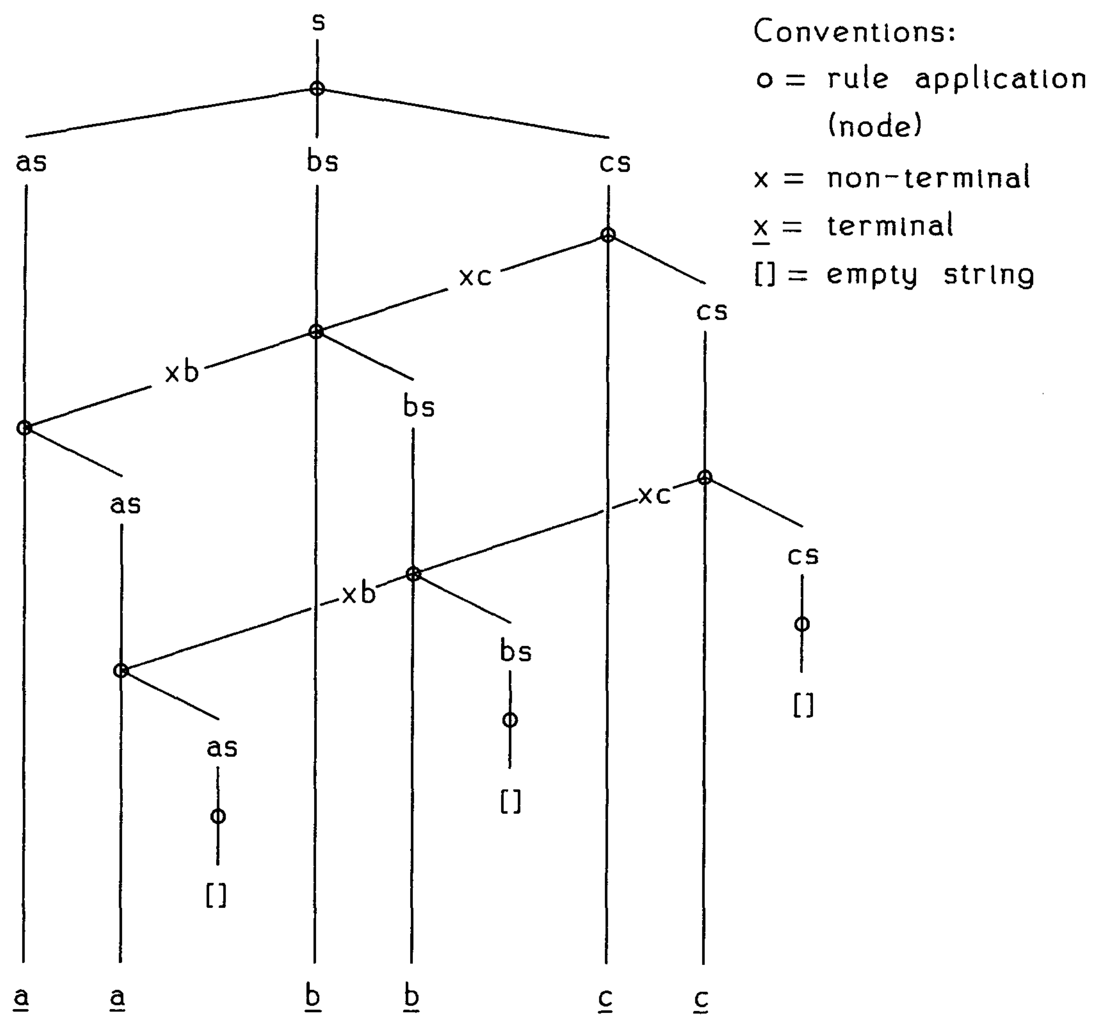


Figure 7.1. Derivation graph for "aabbcc".

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

$a^n b^n c^n$ 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 counter-clockwise following the sequence of the corresponding symbols in the right-hand side of the rule for the node.

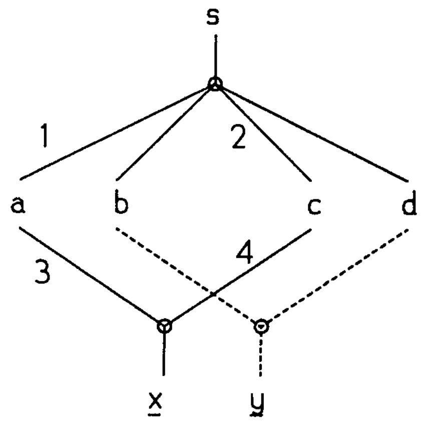
Then the derivation graph obeys the bracketing constraint if and only if it can be drawn, following the conventions, without any edges crossing.[1] The example of Figure 7.2 shows this clearly. In this figure, the closed path formed by edges 1, 2, 3, and 4 has the same effect as a matching pair of brackets 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

```
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.

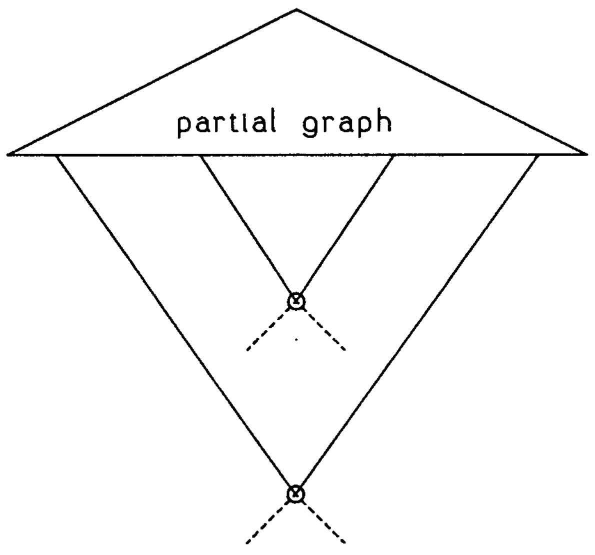
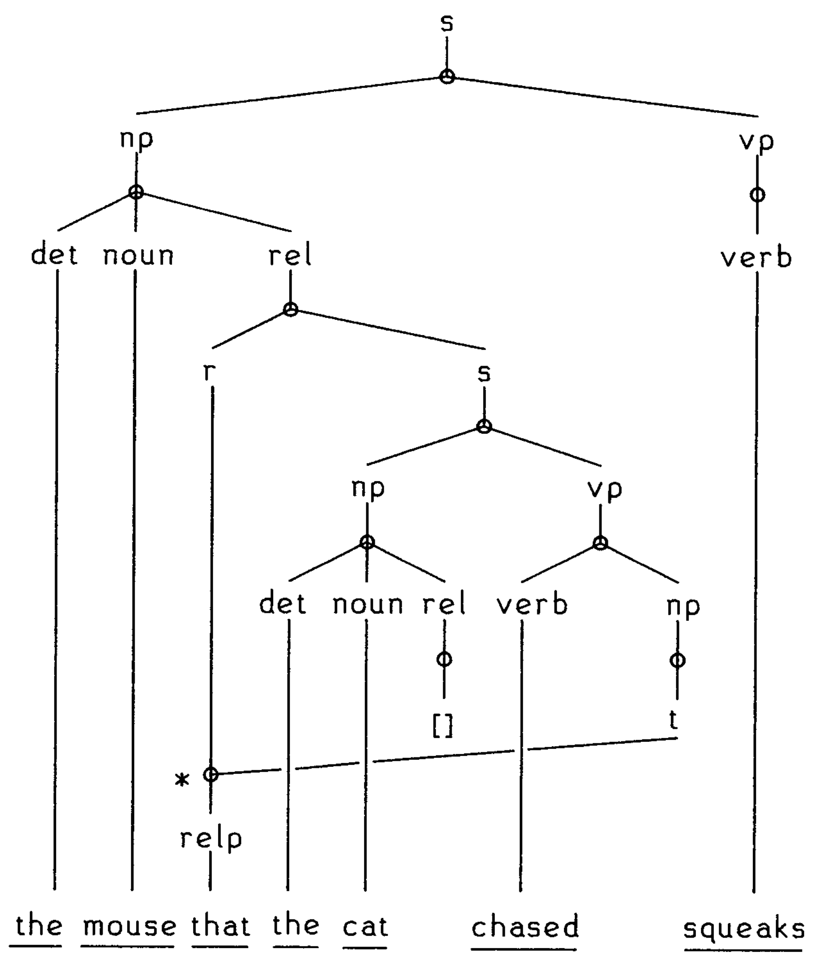


Figure 7.3. Nested rule applications.

```
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-

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



det	=	determiner
np	=	noun_phrase
r	=	rel_marker
rel	=	relative
relp	=	rel_pronoun
s	=	sentence
t	=	trace
vp	=	verb_phrase

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.
```

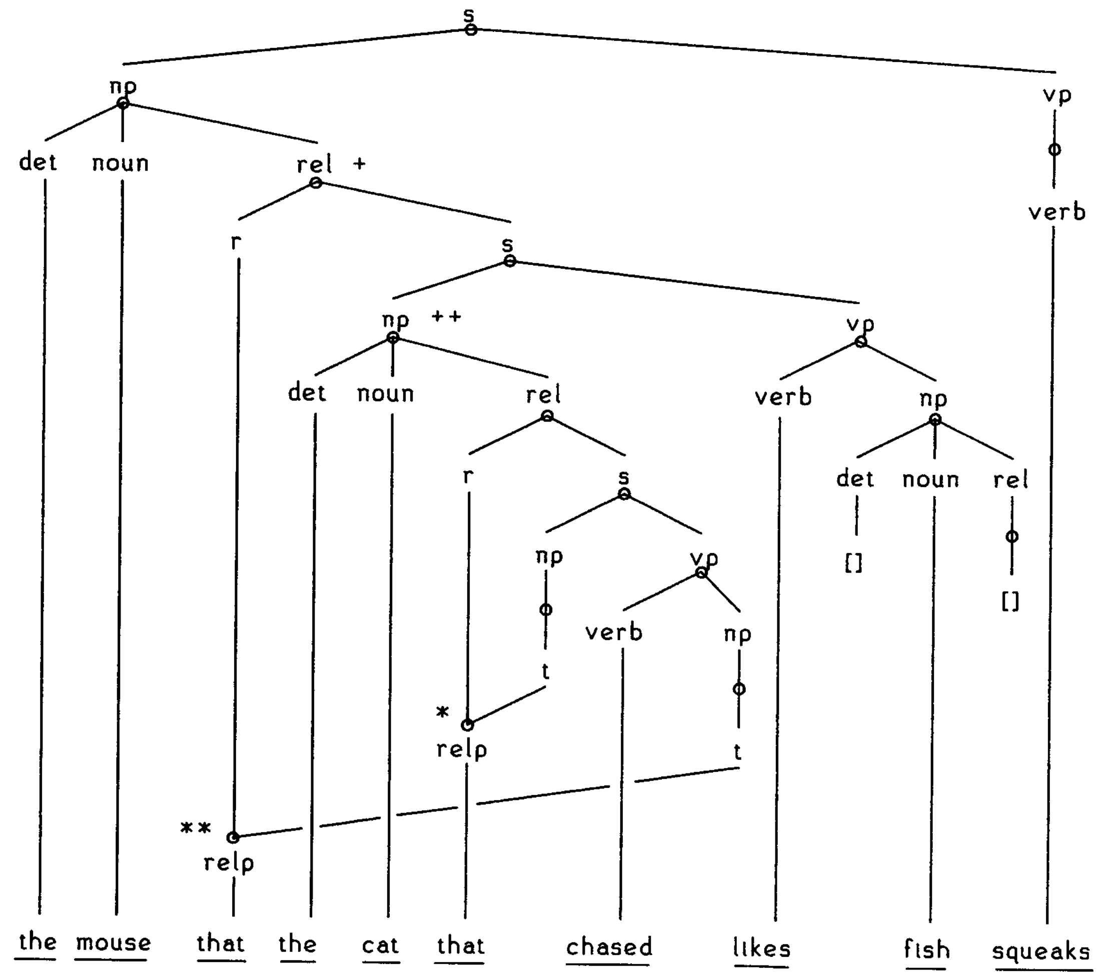


Figure 9.1. Violation of the complex-NP constraint.

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

$$\ldots \text{ open } x \text{ close } \ldots \Rightarrow < x > \ldots$$

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

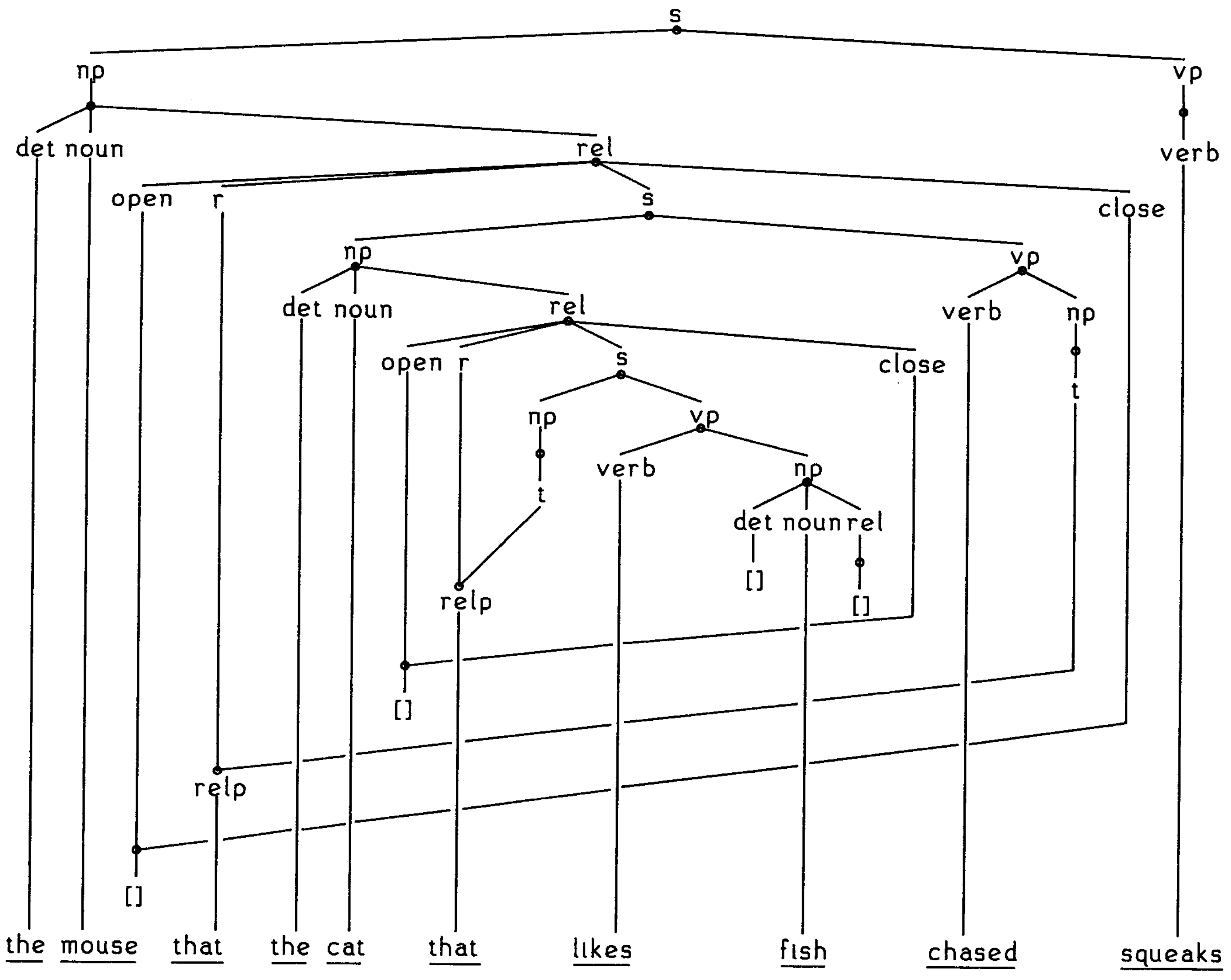


Figure 9.2. Implementation of the complex-NP constraint.

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.