Treating Coordination in Logic Grammars

Veronica Dahl[1]

Computing Sciences Department
Simon Fraser University
Burnaby, B.C. V5A 1S6

Michael C. McCord[2]

Computer Science Department
University of Kentucky
Lexington, KY 40506

Logic grammars are grammars expressible in predicate logic. Implemented in the programming language Prolog, logic grammar systems have proved to be a good basis for natural language processing. One of the most difficult constructions for natural language grammars to treat is coordination (construction with conjunctions like 'and'). This paper describes a logic grammar formalism, *modifier structure grammars* (MSGs), together with an interpreter written in Prolog, which can handle coordination (and other natural language constructions) in a reasonable and general way. The system produces both syntactic analyses and logical forms, and problems of scoping for coordination and quantifiers are dealt with. The MSG formalism seems of interest in its own right (perhaps even outside natural language processing) because the notions of syntactic structure and semantic interpretation are more constrained than in many previous systems (made more implicit in the formalism itself), so that less burden is put on the grammar writer.

1. Introduction

Since the development of the Prolog programming language (Colmerauer 1973; Roussel 1975), logic programming (Kowalski 1974, 1979; Van Emden 1975) has been applied in many different fields. In natural language processing, useful grammar formalisms have been developed and incorporated in Prolog: metamorphosis grammars, due to Colmerauer (1978), and extraposition grammars, defined by F. Pereira (1981); definite clause grammars (Pereira and Warren 1980) are a special case of metamorphosis grammars. The first sizable application of logic grammars was a Spanish/French-consultable data base system by Dahl (1977, 1981), which was later adapted to Portuguese by L. Pereira and H. Coelho and to English by F. Pereira and D. Warren. Coelho (1979) developed a consulting system in Portuguese for library service, and F. Pereira and D. Warren (1980) developed a sizable English data base query system with facilities for query optimization. McCord (1982, 1981) presented ideas for syntactic analysis and semantic interpretation in logic grammars, with application to English grammar; some of these ideas are used in our work described here.

Coordination (grammatical construction with the conjunctions 'and', 'or', 'but') has long been one of the most difficult natural language phenomena to handle, because it can involve such a wide range of grammatical constituents (or non-constituent fragments), and ellipsis (or reduction) can occur in the items conjoined. In most grammatical frameworks, the grammar writer desiring to handle coordination can get by reasonably well by writing enough specific rules involving particular grammatical categories; but it appears that a proper and general treatment must recognize coordina-

[1] Visiting in the Computer Science Department, University of Kentucky, during part of this research. Work partially supported by Canadian NSERC Grant A2436 and Simon Fraser P.R. Grant 42406.

[2] Current address: IBM Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, NY 10598.

0362-613X/83/020069-23$03.00

tion as a "metagrammatical" construction, in the sense that metarules, general system operations, or "second-pass" operations such as transformations, are needed for its formulation.

Perhaps the most general and powerful metagrammatical device for handling coordination in computational linguistics has been the SYSCONJ facility for augmented transition networks (ATNs) (Woods 1973; Bates 1978). The ATN interpreter with this facility built into it can take an ATN that does not itself mention conjunctions at all, and will parse reduced coordinate constructions, which are of the form

A X and Y B,

for example,

John drove his car through and
A X

completely demolished a plate glass window.
Y B

where the unreduced deep structure corresponds to

A X B and A Y B.

The result of the parse is this unreduced structure. SYSCONJ accomplishes this by treating the conjunction as an interruption which causes the parser to back up in its history of the parse. Before backing up, the current configuration (immediately before the interruption) is suspended for later merging. The backing up is done to a point when the string X was being parsed (this defines X), and with this configuration the string Y is parsed. The parsing of Y stops when a configuration is reached that can be merged with the suspended configuration, whereupon B is parsed. The choices made in this process can be deterministic or non-deterministic, and can be guided by syntactic or semantic heuristics.

There are some problems with SYSCONJ, however. It suffers from inefficiency, due to the combinatorial explosion from all the choices it makes. Because of this inefficiency, it in fact has not been used to a great extent in ATN parsing. Another problem is that it does not handle embedded complex structures. Furthermore, it is not clear to us that SYSCONJ offers a good basis for handling the scoping problems that arise for semantic interpretation when conjunctions interact with quantifiers (and other modifiers) in the sentence. This latter problem is discussed in detail below.

In this paper we present a system for handling coordination in logic grammars. The system consists of three things:

(1) a new formalism for logic grammars, which we call *modifier structure grammars* (MSGs),
(2) an interpreter (or parser) for MSGs that takes all the responsibility for the syntactic aspects of coordination (as with SYSCONJ), and
(3) a semantic interpretation component that produces logical forms from the output of the parser and deals with scoping problems for coordination.

The whole system is implemented in Prolog-10 (Pereira, Pereira, and Warren 1978).

Coordination has of course received some treatment in standard logic grammars by the writing of specific grammar rules. The most extensive treatment of this sort that we know of is in Pereira et al. (1982), which also deals with ellipsis. However, we are aware of no general, metagrammatical treatment of coordination in logic grammars previous to ours.

Modifier structure grammars, described in detail in Section 2, are true logic grammars, in that they can be translated (compiled) directly into Horn clause systems, the program format for Prolog. In fact, the treatment of extraposition in MSGs is based on F. Pereira's (1981) extraposition grammars (XGs), and MSGs can be compiled into XGs (which in turn can be compiled into Horn clause systems). A new element in MSGs is that the formation of analysis structures of sentences has been made largely implicit in the grammar formalism. For previous logic grammar formalisms, the formation of analyses is entirely the responsibility of the grammar writer. Compiling MSGs into XGs consists in making this formation of analyses explicit.

Although MSGs can be *compiled* into XGs, it seems difficult to do this in a way that treats coordination automatically (it appears to require more metalogical facilities than are currently available in Prolog systems). Therefore, we are using an *interpreter* for MSGs (written in Prolog).

For MSGs, the analysis structure associated (by the system) with a sentence is called the *modifier structure* (MS) of the sentence. This structure can be considered an annotated phrase structure tree, and in fact the name "modifier structure grammar" is intended to be parallel to "phrase structure grammar". If extraposition and coordination are neglected, there is a context-free phrase structure grammar underlying an MSG; and the MS trees are indeed derivation trees for this underlying grammar, but with extra information attached to the nodes.

In an MS tree, each node contains not only syntactic information but also a term called a *semantic item* (supplied in the grammar), which determines the node's contribution to the logical form of the sentence. This contribution is for the node *alone,* and does not refer to the daughters of the node, as in the approach of Gazdar (1981). Through their semantic items, the daughters of a node act as *modifiers* of the node, in a fairly traditional sense made precise below – hence the term "modifier structure".

The notion of modifier structure used here and the semantic interpretation component, which depends on it, are much the same as in previous work by McCord

(1982, 1981), especially the latter paper. But new elements are the notion of MSG (making modifier structure implicit in the grammar), the MSG interpreter, with its treatment of coordination, and the specific rules for semantic interpretation of coordination.

The MSG interpreter is described in Section 3. As indicated above, the interpreter completely handles the syntax of coordination. The MSG grammar itself should not mention conjunctions at all. The interpreter has a general facility for treating certain words as *demons* (cf. Winograd 1972), and conjunctions are handled in this way. When a conjunction demon appears in a sentence

A X conj Y B,

a process is set off which in outline is like SYSCONJ, in that backing up is done in the parse history in order to parse Y parallel to X, and B is parsed by merger with the state interrupted by the conjunction. However, our system has the following interesting features:

(1) The MSG interpreter manipulates stacks in such a way that embedded coordination (and coordination of more than two elements) and interactions with extraposition are handled. (Examples are given in the Appendix.)

(2) The interpreter produces a modifier structure for the sentence

A X conj Y B

which remains close to the surface form, as opposed to the unreduced structure

A X B conj A Y B

(but it *does* show all the pertinent semantic relations through unification of variables). Not expanding to the unreduced form is important for keeping the modifier relationships straight, in particular, getting the right quantifier scoping. Our system analyzes the sentence

Each man drove a car through and
completely demolished a glass window,

producing the logical form

```
each(X,man(X),exists(Y,car(Y),
   exists(Z,glass(Z)&window(Z),
   drove_through(X,Y,Z)
      &completely(demolished(X,Z)) )))
```

This logical form would be difficult to recover from the unreduced structure, because the quantified noun phrases are repeated in the unreduced structure, and the logical form that corresponds most naturally to the unreduced structure is not logically equivalent to the above logical form.

(3) In general, the use of modifier structures and the associated semantic interpretation component permits a good treatment of scoping problems involving coordination. Examples are given below.

(4) The system seems reasonably efficient. For example, the analysis of the example sentence under (2) above (including syntactic analysis and semantic interpretation) was done in 177 milliseconds. The reader can examine analysis times for other examples in the Appendix. One reason for the efficiency is just that the system is formulated as a logic programming system, and especially that it uses Prolog-10, with its compiler. Another reason presumably lies in the details of the MSG interpreter. For example, the interpreter does not save the complete history of the parse, so that the backing up necessary for coordination does not examine as much.

(5) The code for the system seems short, and most of it is listed in this paper.

The semantic interpretation component is described in Section 4, but not in complete detail since it is taken in the main from McCord (1982, 1981). Emphasis is on the new aspects involving semantic interpretation of coordinate modifiers.

Semantic interpretation of a modifier structure tree is done in two stages. The first stage, called *reshaping,* deals heuristically with the well-known scoping problem, which arises because of the discrepancies that can exist between (surface) syntactic relations and intended semantic relations. Reshaping is a transformation of the syntactic MS tree into another MS tree with the (hopefully) correct modifier relations. The second stage takes the reshaped tree and translates it into logical form. The modifiers actually do their work of modification in this second stage, through their semantic items.

As an example of the effects of reshaping on coordinate structures involving quantifiers, the sentence

Each man and each woman ate an apple

is given the logical form

```
each(X,man(X),exists(Y,apple(Y),ate(X,Y)))
   & each(X,woman(X),exists(Y,apple(Y),ate(X,Y))),
```

whereas the sentence

A man and a woman sat at each table

is given the form

```
each(Y,table(Y), exists(X,man(X),sat_at(X,Y))
                    & exists(X,woman(X),sat_at(X,Y))).
```

Section 5 of the paper presents a short discussion of possible improvements for the system, and Section 6 consists of concluding remarks. The Appendix to the paper contains a listing of most of the system, a sample MSG, and sample parses. The reader may wish to examine the sample parses at this point.

2. Modifier Structure Grammars

The most fundamental type of logic grammar is Colmerauer's (1978) metamorphosis grammar (MG). Grammars of this type can be viewed as generalized type-0 phrase structure grammars in which the grammar symbols (terminals and non-terminals) are terms from predicate logic. In derivations, the rewriting of symbol strings involves *unification* (Robinson 1965), instead of simple replacement.

F. Pereira's (1981) extraposition grammars (XGs) are essentially generalizations of MGs designed to handle (left) extraposition. In the left-hand side of an XG rule, grammar symbols can be connected by the infix operator '...', indicating a *gap*. When such a rule is used in rewriting, the gaps appearing in the left-hand side may match arbitrary strings of grammar symbols, and then the left-hand side is replaced by the right-hand side followed by the symbol strings matched by the gaps (in the same order). For example, the XG rule

```
a,b...c...d --> e,f
```

is really a rule schema

```
a,b,X,c,Y,d --> e,f,X,Y
```

where X and Y stand for arbitrary grammar symbol strings. There is a constraint on the use of gaps in rewriting called the *bracketing constraint,* for which we refer to F. Pereira (1981). However, our MSG interpreter includes XG interpretation, so the use of gaps is in fact completely specified below.

In XG rules, symbols on the left-hand side following gaps represent left-extraposed elements. For example, the extraposition of noun phrases to the front of relative clauses (with replacement by relative pronouns) can be handled by the XG rules:

```
relative_clause --> rel_marker, sentence.
rel_marker...trace --> rel_pronoun.
noun_phrase --> trace.
```

where 'trace' marks the position out of which the noun phrase is being moved, and is used by the second rule above in conjunction with a relative marker to produce (or analyze) a relative pronoun.

Pereira's implementation of XGs is a Prolog program that compiles XGs to Horn clause systems, which in turn can be run by Prolog for parsing sentences. In the compiled systems, extraposition is handled by the manipulation of a stack called the *extraposition list,* which is similar to the HOLD list for ATN's (Woods 1973). Elements (like 'trace' above) on the left-hand sides of XG rules following the initial symbol are in effect put on the extraposition list during parsing, and can be taken off when they are required later by the right-hand side of another rule. Our MSG interpreter uses a reformulation of this same method.

Since the grammar symbols in XGs (and MGs) can be arbitrary terms from predicate logic, they can contain arguments. These arguments can be used to hold useful information such as selectional restrictions and analysis structures. For example, in the rule

```
sentence(s(Subj,Pred)) -->
   noun_phrase(Subj),verb_phrase(Pred)
```

each non-terminal is augmented with an argument representing a syntactic structure. (Here, following Prolog-10 syntax, the capitalized items are variables.) Manipulating such arguments is the only way of getting analysis structures in XGs. As indicated in the Introduction, a new ingredient in MSGs over XGs is to automate this process, or to make it implicit in the grammar.

MSG rules are of two forms. The basic form is

```
A:Sem --> B.
```

where A-->B is an XG rule and Sem is a term called a *semantic item,* which plays a role in the semantic interpretation of a phrase analyzed by application of the rule. The semantic item is (as in McCord 1981) of the form

```
Operator–LogicalForm
```

where, roughly, LogicalForm is the part of the logical form of the sentence contributed by the rule, and Operator determines the way this partial structure combines with others. Details on semantic items are postponed to Section 4 (on semantic interpretation). Actually, the current section and Section 3 deal mainly with syntactic constructions which are independent of the form of semantic items.

The second type of MSG rule looks exactly like an XG rule (no Sem is exhibited), but the system takes care of inserting a special "trivial" Sem, ℓ-true. (Here the 'ℓ' is the operator for left-conjoining, described in Section 4.) Most MSG rules for higher (non-lexical) types of phrases are of this type, but not all of them are.

A simple example of an MSG is shown in Figure 1. Following the notational conventions of XGs (as well as the simpler definite clause grammars built into Prolog-10), we indicate terminal symbols by enclosing them in brackets [ ]. Rules can contain tests on their right-hand sides, enclosed in braces {}, which are Prolog goals. In this example, the tests are calls to the lexicon, shown after the grammar rules, which consists of assertions (non-conditional Horn clauses).

This grammar, together with the semantic interpretation component, will handle sentences like the following, producing the indicated logical forms:

```
sent --> nounph(X),verbph(X).

nounph(X) --> det(X),noun(X).
nounph(X) --> proper_noun(X).

verbph(X) --> verb(X,Y),nounph(Y).

det(X):Sem --> [D],{d(D,X,Sem)}.

noun(X):ℓ-Pred --> [N],{n(N,X,Pred)}.

proper_noun(N) --> [N],{npr(N)}.

verb(X,Y):ℓ-Pred --> [V],{v(V,X,Y,Pred)}.

/* Lexical entries */

n(man,X,man(X)).   n(woman,X,woman(X)).

npr(john).   npr(mary).

v(saw,X,Y,saw(X,Y)).   v(heard,X,Y,heard(X,Y)).

d(each,X,P/Q-each(X,Q,P)).
d(a,X,P/Q-exists(X,Q,P)).
```

Figure 1. A simple MSG with lexicon.

```
John saw Mary.
   saw(john,mary).

John heard each woman.
   each(Y,woman(Y),heard(john,Y)).

Each man saw a woman.
   each(X,man(X),exists(Y,woman(Y),saw(X,Y))).
```

A larger example MSG is listed in the Appendix. This grammar includes rules dealing with extraposition, and the lexicon contains rules used by the MSG interpreter in handling coordination.

Now let us look at the formation of syntactic structures by the MSG system. As stated in the Introduction, syntactic structures are trees called *modifier structure* (MS) trees.

Suppose that a phrase is analyzed by application of an MSG rule

```
A:Sem --> B.
```

and further rule applications in an MSG. (The Sem may be explicit or supplied by the system for the second type of rule.) Then the *modifier structure* of the phrase is a term of the form

```
syn(NT,Sem,Mods)
```

where NT is the leading symbol (a non-terminal) in A and where Mods is the list of modifier structures of the subphrases analyzed with the right-hand side B of the rule. The 'syn' structure is considered a tree node, labeled with the two items NT and Sem, and having daughter list Mods.

As an example, the MS tree for the sentence "Each man saw a woman" produced by the grammar in Figure 1 is shown in Figure 2. This tree is printed by displaying the first two fields of a 'syn' on one line and then recursively displaying the daughters, indented a fixed amount.

```
sent ℓ-true
   nounph(X) ℓ-true
      det(X) P/Q-each(X,Q,P)
      noun(X) ℓ-man(X)
   verbph(X) ℓ-true
      verb(X,Y) ℓ-saw(X,Y)
      nounph(Y) ℓ-true
         det(Y) R/S-exists(Y,S,R)
         noun(Y) ℓ-woman(Y)
```

Figure 2. MS tree for "Each man saw a woman.".

Let us now indicate briefly how MSGs can be compiled into XGs so that these MS trees are produced as analyses. This method of compiling does not handle coordination metagrammatically (as does the interpreter), but it does seem to be of general interest for MSGs.

In the compiled XG version of an MSG, each non-terminal is given two additional arguments, added, say, at the end. Each argument holds a list of modifiers. If the original non-terminal is nt(X1,...,Xn), the new non-terminal will look like

```
nt(X1,...,Xn,Mods1,Mods2).
```

When this non-terminal is expanded by a non-trivial rule, then Mods1 will differ from Mods2 by having one additional modifier on the front, namely the modifier contributed by the rule application. A rule is *trivial* if its right-hand side is empty. When a trivial rule is used to expand 'nt' above, Mods1 will equal Mods2.

As an example of rule translation, the first rule in Figure 1 is translated to

```
sent([syn(sent,ℓ-true,Mods1) | Mods],Mods) -->
      nounph(X,Mods1,Mods2),verbph(X,Mods2,[ ]).
```

(Here [X | Y] denotes the list with first member X and remainder Y.)

Any non-terminal on the left-hand side of an MSG besides the leading non-terminal is given a pair of identical Mods arguments (because it contributes no modifier by itself). For example, the MSG rule

```
rel_mk(X)...trace(X) --> rel_pron.
```

would translate to

```
rel_mk(X,[syn(rel_mk(X),ℓ-true,Mods1) | Mods],Mods)
   ...trace(X,Mods2,Mods2) --> rel_pron(Mods1,[ ]).
```

For parsing a sentence with respect to the grammar in Figure 1, one would use

sent([MST],[ ])

as start symbol (with MST unknown) and the parse would bind MST to the modifier structure tree of the sentence.

Pairs of list arguments manipulated in the way just outlined are called "difference lists", and the technique is common in logic programming. One part of compiling MGs to Horn clauses is the addition to each non-terminal of an argument pair for the terminal strings being analyzed. Pereira's compilation of XGs to Horn clauses involves one more argument pair for extraposition lists. So the compilation of MSGs to Horn clauses involves three argument pairs totally. In the MSG interpreter, described in the next section, only a single argument (not a pair) is needed for each of these three lists.

3. The MSG Interpreter and the Syntax of Coordination

Our MSG processor actually has a bit of compiler in it, because there is a preprocessor that translates MSG rules into a form more convenient for the interpreter to use.

An MSG rule

A:Sem --> B

is preprocessed into a term

rule(NT,Ext,Sem,B1)

where NT is the leading non-terminal in A, Ext is the conversion of the remainder of A into an *extraposition list,* and B1 is the conversion of B to list form.

The notion and representation of *extraposition lists* used here are just the same as F. Pereira's (1981). A node in such a list is of the form

x(Context,Type,Symbol,Ext)

where Context is either 'gap' or 'nogap', Type is either 'terminal' or 'nonterminal', Symbol is a grammar symbol, and Ext is the remainder of the list. We denote the empty extraposition list by 'nil' (Pereira used [ ]).

The "left-hand side remainder" in a grammar rule (the part after the leading symbol) is converted to an extraposition list in a straightforward way, with one node for each symbol in the remainder. The Context says whether the symbol has a gap preceding it, and the remaining fields of an 'x' node have the obvious meaning. For the rule

a,[b]...c --> d

the extraposition list would be

x(nogap,terminal,b,x(gap,nonterminal,c,nil)).

The right-hand side of an MSG rule is preprocessed to a (simple) list form in the obvious way. Thus, a right-hand side (d,e,f) is converted to the list [d,e,f], and a right-hand side with a single element d is converted to the list [d].

As a complete example, the MSG rule

a...b:ℓ-p --> [c],{d},e

is converted to

rule(a,x(gap,nonterminal,b,nil),ℓ-p,[[c],{d},e]).

If the MSG rule exhibits no semantic item, then the preprocessor supplies the trivial item ℓ-true.

The 'rule' forms of the rules in an MSG are stored as assertions in the Prolog data base, to be used by the interpreter. One can understand

rule(NT,Ext,Sem,B1)

as the assertion: "There is a rule for the non-terminal NT with extraposition list Ext, etc."

The rule preprocessor is listed at the beginning of the Appendix.

Now let us look at the interpreter itself, which is listed after the preprocessor in the Appendix.

The top-level procedure is

parse(String,NonTerminal,Syn)

which takes a String of terminals and attempts to parse it as a phrase of type NonTerminal, with the syntactic structure Syn. We should comment that 'parse' defines a top-down parser.

This procedure calls the main working procedure

prs(BodyList,String,Mods,Par,Mer,Ext)

which parses String against a list BodyList of goals of the type that can appear in the right-hand side (the body) of a rule. The list of resulting syntactic structures is Mods (one modifier for each non-trivial expansion of a non-terminal in BodyList). The remaining three arguments of 'prs' are for stacks called the *parent* stack, the *merge* stack, and the extraposition list. These are initialized to 'nil' in the call of 'parse' to 'prs'.

The parent stack serves two purposes. One is to control the recursion in the normal working of the parser. (The parser is much like an interpreter for a programming language – in fact, for a specialized version of Prolog itself.) The other purpose is to provide information for the coordination demon, when it backs up in (part of) the parse history.

A non-empty parent stack is a term of the form

parent(BodyList,Mods,Par)

where BodyList is a body list, Mods is a modifier list, and Par is again a parent stack. A new level gets pushed onto the parent stack by the sixth rule for 'prs' and the ancillary procedure 'prspush'. This happens

when 'prs' is looking at a body list of the form [NT | BL], where the initial element NT is a non-terminal that can be expanded by a 'rule' entry. If that rule is trivial (if its own body is empty), then no actual push is done, and 'prs' continues with the remaining current body list BL. Otherwise, 'prspush' goes to a lower level, to parse against the body of the expanding rule. The items [NT | BL] and Mods from the higher level are saved on the parent stack (Mods is a variable for the *remaining* modifiers to be found on the higher level).

Note that the body list [NT | BL] saved in the first field of the 'parent' term is more than is needed for managing the recursive return to the higher level. Only the remainder, BL, is needed for this, because NT has already been used in the parse. In fact, the rule that pops to the higher level (the eighth rule for 'prs') does ignore NT in doing the pop. The extra information, NT, is saved for the second purpose of the parent stack, the backing up by the coordination demon.

Before going into the details of coordination, though, let us continue with the description of the "normal" working of the parser.

In normal parsing, there is exactly one place where a new 'syn' node is added to the MS trees being built. This is in the second rule for 'prspush', which handles non-trivial rule expansions. The addition of this node is in accordance with the definition of modifier structure given in the preceding section.

The pushing rule of 'prs' (the sixth rule) also manipulates the extraposition stack. The extraposition component of the expanding rule is concatenated onto the front of the main extraposition list (being carried in the last argument of 'prs'). This is analogous to a HOLD operation in ATNs. Of course, if no extraposition is shown in the rule, the extraposition list will remain the same.

The third and fourth rules for 'prs' handle terminals in the body list. The first of these tries to remove the terminal from the string argument, and the second tries to remove it from the extraposition list (as in a VIR arc for ATNs).

The seventh 'prs' rule tries to remove a non-terminal from the extraposition list (again, like a VIR arc).

The last 'prs' rule is the termination condition for the parse. It just requires that all arguments be null.

Now we can discuss coordination demons. All the rest of the interpreter rules deal with these.

The first 'prs' rule is the one that notices demon words D. It calls a procedure 'demon', passing D as the first argument and all the rest of the information it has in other arguments. 'demon' takes control of the rest of the parse. In the listed interpreter there is only one 'demon' rule, one that tests whether D is a conjunction. It does this with the goal

```
conjunction(D,Cat,Sem),
```

which gives the syntactic category Cat for the conjunction D, and the semantic item Sem for a new modifier node to be constructed for the right conjunct. The lexicon contains 'conjunction' entries as assertions.

For understanding what the conjunction demon does, it is best to look at an example, as it would be parsed for the grammar in the Appendix. We will use the specific notation (for variables, etc.) given in the demon rule, and the reader should refer to that rule in the Appendix. It should be borne in mind that Prolog is non-deterministic; we will only state what happens on the *successful* path through the choices made.

The example is

John saw and Mary heard the train.

When the demon for 'and' is called, the current body list is

```
BL=[comps([obj-Y])].
```

The non-terminal comps(Comps) looks for a list Comps of complements, and in this case there is to be one complement, an object noun phrase. The MS tree constructed so far is

```
sent ℓ-true
   nounph(X,def) @P-def(X,X=john,P)
   verbph(X) ℓ-true
      verb(X,[obj-Y]) ℓ-saw(X,Y)
       | Mods
   | Mods2
```

Here the entry | Mods in the last daughter position for the verb phrase indicates further modifiers on that level to be put in the unbound variable Mods. (This is explicitly the same variable 'Mods' used in the demon rule.) Similarly, | Mods2 represents the remaining modifiers for 'sent' node. The variable Mods2 does not appear in the 'demon' rule, but will be referred to below.

The parent stack Par available to the demon has two levels, and the two body lists are

```
[verbph(X)],
[sent].
```

(Recall that we are describing the state of affairs in the successful path through the search space.) The recursive procedure 'backup' is called, which can look any number of levels through the parent stack. It goes to the second level, where the body list is [sent]. (Choosing the first level with [verbph(X)] would be appropriate for the sentence "John saw and barely heard the train".) In passing up a level, 'backup' requires that the body list skipped over must be 'satisfied', which means that any pending goals in the body list (members of its tail) are satisfiable by trivial

rules. When 'backup' does pass up a level, the modifier list for that level is closed off. Thus Mods in the tree displayed above will be bound to the empty list. (There are no more modifiers for that 'verbph' node.)

As a single remaining daughter for the level backed up to, a new 'syn' node for the right conjunct is attached by the demon. This means binding the variable Mods2 in the above tree to the list consisting of this node. Now our tree looks like

```
sent ℓ-true
   nounph(X,def)  @P-def(X,X=john,P)
   verbph(X) ℓ-true
      verb(X,[obj-Y]) ℓ-saw(X,Y)
   conj(and) Q*R-(Q&R)
      | Mods0
```

The variable Mods0 is to contain the list of modifiers for the conjunction node. This list will turn out to have a single element, a new 'sent' node for the remainder of the sentence, "Mary heard the train".

Backing up to the [sent] level makes the nonterminal NT=sent available to the demon, and the parent stack Par1 at the [sent] level. The demon then continues the parse by calling 'prs' with body list [NT]=[sent], but with information pushed onto the merge stack. The main item stored on the merge stack is the body list BL=[comps([obj-Y])], which was pending at the time of interruption by the conjunction. The items Par1, Ext, and of course the old merge stack Mer are also pushed on.

So now we continue parsing "Mary heard the train", but with another kind of demon lurking, the interrupted goal BL. The second rule for 'prs' notices this demon. When we are parsing and come to a goal that can be *unified* with BL, then we can try merging. This happens when we are looking for the complements of "heard". This unification includes the unification of the object variable Y of "saw" with the object variable of "heard", so that "the train" will logically be the object of "saw" as well as "heard".

The procedure 'cutoff' called by the second 'prs' rule requires that no new unsatisfied goals have developed in parsing the right conjunct (aside from the goal BL to be merged) and also closes off modifier lists in the local parent stack Par for the right conjunct.

Then the merged parse is continued by a call to 'prs', with BL as goal and with the parent stack, merge stack, and extraposition list popped from the merge stack. When this is completed, our MS tree is as shown in Figure 3.

The meanings of the semantic items used in this MS tree, and their use in producing the logical form, will be explained in the next section; but it is worth stating now what the resulting logical form is:

```
def(Y,train(Y),saw(john,Y)&heard(mary,Y)).
```

The reader may examine the analyses produced for other examples listed in the Appendix.

```
sent ℓ-true
   nounph(X,def) @P-def(X,X=john,P)
   verbph(X) ℓ-true
      verb(X,[obj-Y]) ℓ-saw(X,Y)
   conj(and) Q*R-(Q&R)
      sent ℓ-true
         nounph(Z,def) @S-def(Z,Z=mary,S)
         verbph(Z) ℓ-true
            comps([obj-Y]) ℓ-true
               comp(obj-Y) ℓ-true
                  nounph(Y,def) ℓ-true
                     det(Y,def) T/U-def(Y,U,T)
                     noun(Y,[ ]) ℓ-train(Y)
```

Figure 3. MS tree for "John saw and Mary heard the train."

4. Semantic Interpretation and Coordination

The overall idea of the semantic interpretation component was given in the Introduction. The rule system is listed completely in the Appendix. This system is taken essentially from McCord (1981), with some rules deleted (rules dealing with focus), and some rules added for coordination.

For a discussion of MS tree reshaping as a means of handling scoping of modifiers, we refer to McCord (1982, 1981). Also, the reader may examine the examples of reshaped trees given in the Appendix.

We will, however, review the second stage of semantic interpretation, because the new rules for coordination are added here and because it is more central for understanding modifier structure. In this stage, the reshaped MS tree is translated to logical form, and the top-level procedure for this is 'translate'. This procedure actually works only with the semantic-item components of MS tree nodes. (Reshaping uses the first, syntactic components.)

One semantic item can combine with (or modify) a second semantic item to produce a third semantic item. 'translate' uses these combining operations in a straightforward recursive fashion to produce the logical form of an MS tree. The ancillary procedure ('transmod') that actually does the recursion produces complete semantic items as translations, not just logical forms. For the top-level result, the operator component is thrown away. 'transmod' works simply as follows: The daughters (modifiers) of a tree node N are translated recursively (to semantic items) and these items cumulatively modify the semantic item of N, the leftmost acting as the outermost modifier, etc.

So the heart of the translation process is in the rules that say how semantic items can combine with

other semantic items. These are rules for the procedure

trans(Sem0,Sem1,Sem2)

which says that Sem0 combines with (modifies) Sem1 to produce Sem2. In the *typical* case, this combination depends only on the Operator component of Sem0; but there are exceptional cases where it depends as well on the operator in Sem1. Furthermore, 'trans' is free to create a new operator for the result, Sem2, which can affect later operations. This happens with coordinate modifiers. We often speak of Sem0 "operating on" Sem1, but "combining with" is the more accurate term generally.

The only operators appearing in the small sample grammar in the Appendix are of the form ℓ, @P, P/Q, and P*Q. Here P and Q are variables standing for logical forms. The listing for 'trans' in the Appendix includes only rules for these operators and their auxiliaries, although larger grammars involve other operators. We will elucidate the effects of these four operators with examples. The last one, P*Q, is used for coordination.

The operator 'ℓ' is for left-conjoining. When ℓ-man(X) operates on ℓ-see(X,Y), the result is ℓ-man(X)&see(X,Y).

The operator @P is used for substitutions in its associated logical form. When @P-not(P) operates on ℓ-laugh(X), the result is ℓ-not(laugh(X)).

The operator P/Q is used for forms that require two substitutions. When

P/Q-each(X,Q,P)

operates on ℓ-man(X), the result is

@P-each(X,man(X),P),

which in turn can operate by substituting for P.

Notice that @p and P/Q are similar to lambda(P) and lambda(Q)lambda(P) respectively. But they also interact with other operators in the system in specific ways.

To show these first three operators working together, let us look at the MS tree that would be produced for the sentence "Each man laughed". (Reshaping leaves this tree unaltered.) We throw away the syntactic fields in the tree nodes (working only with the semantic items), and show the successive stages in producing the logical form in Figure 4. In following the steps in Figure 4, the reader should refer to the 'trans' rules in the Appendix, which are numbered for reference here. In each step of the translation, a node combines with its parent, and the 'trans' rule used to do this is indicated.

The operator P*Q appears in coordinate modifiers. The first four 'trans' rules deal with it, and they create auxiliary operators. The following example will make clear how these are manipulated. The sentence is

(1) ℓ-true
  ℓ-true
    P/Q-each(X,Q,P)
    ℓ-man(X)
  ℓ-true
    ℓ-laughed(X) (Rule 7 applies)

(2) ℓ-true
  ℓ-true
    P/Q-each(X,Q,P)
    ℓ-man(X)
  ℓ-laughed(X) (Rule 7)

(3) ℓ-laughed(X)
  ℓ-true
    P/Q-each(X,Q,P)
    ℓ-man(X) (Rule 7)

(4) ℓ-laughed(X)
  ℓ-man(X)
    P/Q-each(X,Q,P) Rule 5)

(5) ℓ-laughed(X)
  @P-each(X,man(X),P) (Rule 6)

(6) ℓ-each(X,man(X),laughed(X)).

Figure 4. The working of 'translate'.

"Each man ate an apple and a pear."

This example is shown in the Appendix, with the initial syntactic analysis and the reshaped tree. In the reshaped tree, the 'sent' node has three daughters, the first being for the simple noun phrase "each man", the second for the conjoined noun phrase "an apple and a pear", and the third for the verb phrase with the object removed.

If we perform all the modifications that are possible in this tree without involving the coordination operator, and if we remove the syntactic fields, then the tree looks like the following:

ℓ-ate(X,Y)
  @P-each(X,man(X),P)
  ℓ-true
    Q/R-exists(Y,R,Q)
    ℓ-apple(Y)
    S*T-(S&T)
      @U-exists(Y,pear(Y),U)

Now the first 'trans' rule can apply to the lowest pair of nodes, and the tree becomes:

ℓ-ate(X,Y)
  @P-each(X,man(X),P)
  ℓ-true
    Q/R-exists(Y,R,Q)
    ℓ-apple(Y)
    cbase1(@U-exists(Y,pear(Y),U),S,T)-(S&T)

We have *saved* the modifier for "a pear" in the first argument of the 'cbase1' operator. Next, this item operates on the ℓ-true node, by application of the second 'trans' rule, and we get the tree

```
ℓ-ate(X,Y)
   @P-each(X,man(X),P)
   cbase2(ℓ,@U-exists(Y,pear(Y),U),S,T,S&T)-true
      Q/R-exists(Y,R,Q)
      ℓ-apple(Y)
```

Now, the third 'trans' rule is applied twice, to the two daughters of the 'cbase2' node, and we get

```
ℓ-ate(X,Y)
   @P-each(X,man(X),P)
   cbase2(@Q,@U-exists(Y,pear(Y),U),S,T,S&T)
                              -exists(Y,apple(Y),Q)
```

Then, as the last step with coordination operators, the fourth 'trans' rule is applied to let the 'cbase2' node operate on the top node of the tree. This involves two recursive calls to 'trans', in which the two conjunct noun phrases operate on the material in the scope of the coordinate node. (In this case, the material in the scope is ate(X,Y).) This material gets *duplicated,* because of the double application to it. The resulting tree now is

```
ℓ-exists(Y,apple(Y),ate(X,Y))&exists(Y,pear(Y),
                                          ate(X,Y))
   @P-each(X,man(X),P)
```

Finally, the @P node modifies the top node, and after discarding the operator (an 'ℓ') in the resulting item, we get the logical form

```
each(X,man(X),exists(Y,apple(Y),ate(X,Y))
                        &exists(Y),pear(Y),ate(X,Y)) )
```

Near the end of the Introduction, examples were given of two syntactically similar sentences with coordination, for which the produced logical forms are quite different. For the sentence

"Each man and each woman ate an apple",

the reshaping stage produces a tree that in outline looks like the following:

```
sent
   nounph "each man"
      conj(and)
         nounph "each woman"
   nounph "an apple"
   verbph "ate"
```

Then, the material for "ate an apple" will be in the scope of the conjoined noun phrase and this material gets duplicated, with the resulting logical form being

```
each(X,man(X),exists(Y,apple(Y),ate(X,Y)))
    &each(X,woman(X),exists(Y,apple(Y),ate(X,Y))).
```

On the other hand, for the sentence

"A man and a woman sat at each table",

reshaping moves the universally quantified noun phrase to the left of the existentially quantified conjoined noun phrase, and the tree is as follows:

```
sent
   nounph "each table"
   nounph "a man"
      conj(and)
         nounph "a woman"
   verbph "sat at"
```

Then the only material in the scope of the conjoined noun phrase is for "sat at", and only this gets duplicated. (In fact, the scoping is like that for our earlier example, "Each man ate an apple and a pear".) The complete logical form is

```
each(Y,table(Y), exists(X,man(X),sat_at(X,Y))
                & exists(X,woman(X),sat_at(X,Y)) ).
```

Notice that the logical forms for conjoined phrases in the above analyses share variables. For instance, the same variable X is used in both man(X) and woman(X) in the last analysis. This sharing of variables arises naturally because of the unification of body lists that is performed during parsing by the 'merge' demon. It keeps things straight very nicely, because the shared variables may appear in another predication, like sat_at(X,Y) above, which occurs only once, outside the conjoined phrase, but is related logically to both conjuncts.

This sharing of variables presents no problems as long as the variables are quantified over (as they are by the existential in the preceding example). But it makes proper nouns less convenient to treat. If coordination were not being considered, it would be convenient to parse proper nouns by the sort of rule listed in Figure 1 in Section 2, where the proper noun gets immediately unified with the variable X appearing in nounph(X). But if such a rule is used with the MSG parser, then a sentence as simple as "John and Mary laughed" will not parse, because the parser attempts to unify the logical subject variable with both 'john' and 'mary'.

Therefore, as the semantic item for a proper noun N, we use a quantified form, specifically

```
@P-def(X,X=N,P),
```

and this is carried through in most of the processing. However, the procedure 'translate', after it has carried out all the modification, calls a procedure 'simplify' which simplifies the logical form. This gets rid of some unnecessary 'true's and it carries out the substitutions implicit in the proper noun forms, by doing some copying of structures and renaming of variables.

For example, the logical form for "John and Mary laughed" prior to simplification is essentially

```
def(X,X=john,laughed(X))
   &def(X,X=mary,laughed(X)).
```

But after simplification, it is

```
laughed(john)&laughed(mary).
```

In the sample analyses in the Appendix, we give in some cases only the logical form and in other cases the intermediate structures also (the syntactic analysis tree and the reshaped tree). Analysis times are in milliseconds. These do not include times for I/O and conversion of character strings to word lists. Variables are printed by Prolog-10 in the form _n, where n is an integer.

5. Possible Extensions

The main advantages of the formalism presented here are:

- automating the treatment of coordination,
- freeing the user of concern with structure-building, and
- providing a modular treatment of semantics, based upon information given locally in each rule.

While making a reasonable compromise between power and elegance on the one hand, and efficiency on the other, our present implementation could be improved in several ways. For instance, because the parsing history is kept in a stack that is regularly popped – the Parent stack – some parsing states are no longer available for backing up to, so the possibility exists for some acceptable sentences not to be recognized.

We have experimented with modifications of the MSG interpreter in which more of the parse history is saved, and have also considered compiling MSGs into Prolog and using a general 'state' predicate which returns the proof history, but we have not as yet obtained satisfactory results along these lines.

Another possible improvement is to use some semantic guidance for the (at present blind) backing up through parsing states. The parser already carries along semantic information (in semantic items) to be used later on. Some of this information could perhaps also be used during parsing, in order to improve the backup. Research along these lines may well provide some more insight into the dilemma of whether syntax and semantics should be kept separate or intermingled.

It would also be interesting to include collective and respective readings of coordinated noun phrases, perhaps along the lines proposed in Dahl (1981).

We do not presume that our general treatment of coordination will work for all possible MSG grammars. Care is necessary in writing an MSG, as with any other formalism. What we do provide are enough elements to arrive at a grammar definition that can treat most structure-building and coordination problems in a modular and largely automated manner.

We have also investigated an alternative approach to coordination, which is not metagrammatical but is nevertheless more flexible than previous approaches, and involves still another grammar formalism we believe worth studying in itself. We have named it the *gapping grammar* (GG) formalism, as its main feature is that it allows a grammar rule to rearrange gaps in a fairly arbitrary fashion. This will be the subject of a forthcoming article.

6. Concluding Remarks

We have described a new logic grammar system for handling coordination metagrammatically, which also automatically builds up the modifier structure of a sentence during parsing. This structure, as we have seen, can be considered an annotated phrase structure tree, but the underlying grammar – unlike other recent approaches to NL processing – is not necessarily context-free. The rules accepted are generalized type-0 rules that may include gaps (in view, for instance, of left extraposition), and semantic interpretation, as we have seen, is guided through the semantic items, local to each rule, which help resolve scoping problems. The system's semantic interpretation component can in particular deal with scoping problems involving coordination.

While the treatment of coordination is the main motivation for developing still another logic grammar formalism, we believe our notion of modifier structure grammar to be particularly attractive for allowing the user to write grammars in a more straightforward manner and more clearly. Also, because the semantic information about the structure being built up is described modularly in the grammar rules, it becomes easier to adapt the parser to alternative domains of application: modifying the logical representation obtained need only involve the semantic items in each rule. A related but less flexible idea was independently developed for Restriction Grammars by Hirshman and Puder (1982). RGs are also logic grammars in the sense that they are based on Prolog, but they deal only with context-free definitions augmented by restrictions (which are procedures attached to the rules). In RGs, a tree record of the context-free rules applied is automatically generated during the parse. More evolved representations for the sentence, however, are again the user's responsibility and require processing this automatically generated parse tree.

Another important point, in our view, is the fact that our system does not preclude context-sensitive rules, transformations, or gaps. This is contrary to what seems to be the general tendency today, both in theoretical linguistics (for example, Gazdar 1981) and in computational linguistics (for example, Hirshman and Puder 1982, Joshi and Levy 1982, Robinson

1982, Schubert and Pelletier 1982), towards using context-free grammars (which, however, are often augmented in some way – through restrictions, local constraints, rule schemata, metarules, etc. – compensating for the lack of expressiveness in simple context-free grammars). This approach was largely motivated by the need to provide alternatives to transformational grammar, which on the one hand was felt by AI researchers to deal insufficiently with semantics and with sentence analysis, and on the other hand, as observed by Gazdar (1981), could not offer linguistically adequate explanations for important constructs, such as coordination and unbounded dependencies. Further arguments supporting this approach include claims of more efficient parsability, simplicity, and modularity.

From the particular point of view of logic grammars, more evolved grammar formalisms make a great deal of sense for various reasons. In the first place, they provide various advantages that have been illustrated in Dahl (1981), namely modularity and conciseness, clarity and efficiency. A detailed discussion of these advantages with respect to augmented transition networks can be found in Pereira and Warren (1980).

Furthermore, they include lower-level grammars as a special case. In particular, context-free rules augmented with procedures may be written, since even the simplest logic grammar defined to date (DCGs) allows Prolog calls to be interspersed with the rules. The greater expressive power allowed by more evolved formalisms, then, can only represent a gain, since it does not preclude more elementary approaches. Logic grammars, in short, seem to be developing – like other computer formalisms – into higher-level tools that allow the user to avoid mechanizable effort in order to concentrate on as yet unmechanizable, creative tasks. MSGs are intended as a contribution in this direction.

7. References

Bates, M. 1978 The Theory and Practice of Augmented Transition Network Grammars. In Bolc, L., Ed., *Natural Language Communication with Computers.* Springer-Verlag, New York: 191-259.

Coelho, H.M.F. 1979 A Program Conversing in Portuguese Providing a Library Service. Ph.D. thesis, University of Edinburgh.

Colmerauer, A. 1973 *Un systeme de communication homme-machine in francais.* Groupe d' Intelligence Artificielle, Université d'Aix-Marseille.

Colmerauer, A. 1978 Metamorphosis Grammars. In Bolc, L., Ed., *Natural Language Communication with Computers.* Springer-Verlag, New York: 133-189.

Dahl, V. 1977 *Un Systeme Deductif d'Interrogation de Banques de Donnees en Espagnol.* Thèse de Doctorat de Spécialité, Université d'Aix-Marsielle.

Dahl, V. 1981 Translating Spanish into Logic through Logic. *American Journal of Computational Linguistics* 13: 149-164.

Gazdar, G. 1981 Unbounded Dependencies and Coordinate Structure. *Linguistic Inquiry* 12(2): 155-184.

Hirshman, L. and Puder, K. 1982 Restriction Grammar in Prolog. *Proc. First International Logic Programming Conference.* Marseille, France: 85-90.

Joshi, A. and Levy, L.S. 1982 Phrase Structure Trees Bear More Fruit than You Would Have Thought. *American Journal of Computational Linguistics* 8: 1-11.

Kowalski, R.A. 1974 Predicate Logic as a Programming Language. *Proc. IFIP 74.* North-Holland, Amsterdam, The Netherlands: 569-574.

Kowalski, R.A. 1979 *Logic for Problem Solving.* North-Holland, New York, New York.

McCord, M.C. 1982 Using Slots and Modifiers in Logic Grammars for Natural Language. *Artificial Intelligence* 18: 327-367.

McCord, M.C. 1981 Focalizers, the Scoping Problem, and Semantic Interpretation Rules in Logic Grammars. Technical Report, University of Kentucky. To appear in Warren, D. and van Caneghem, M., Eds., *Logic Programming and its Applications.*

Pereira, F. 1981 Extraposition Grammars. *American Journal of Computational Linguistics* 7: 243-256.

Pereira, F. and Warren, D. 1980 Definite Clause Grammars for Language Analysis – a Survey of the Formalism and a Comparison with Transition Networks. *Artificial Intelligence* 13: 231-278.

Pereira, F. and Warren, D. 1982 An Efficient Easily Adaptable System for Interpreting Natural Language Queries. *American Journal of Computational Linguistics* 8: 110-122.

Pereira, L.M. et al. 1982 ORBI – An Expert System for Environmental Resource Evaluation through Natural Language. Universidade Nova de Lisboa.

Pereira, L.; Pereira, F.; and Warren, D. 1978 *User's Guide to DEC System-10 Prolog.* Department of Artificial Intelligence, University of Edinburgh.

Robinson, J. 1982 Diagram: a Grammar for Dialogues. *Comm. ACM* 25: 27-47.

Robinson, J.A. 1965 A Machine-Oriented Logic Based on the Resolution Principle. *J. ACM* 12: 23-41.

Roussel, P.L. 1975 Prolog Manuel de Reference et d'Utilisation. Université d'Aix-Marseille.

Schubert, L. and Pelletier, F. 1982 From English to Logic: Context-Free Computation of 'Conventional' Logical Translation. *American Journal of Computational Linguistics* 8(1): 27-44.

Van Emden, M.H. 1975 Programming with Resolution Logic. *Machine Intelligence, 8.* John Wiley, New York, New York.

Winograd, T. 1972. *Understanding Natural Language.* Academic Press, New York, New York.

Woods, W.A. 1973 An Experimental Parsing System for Transition Network Grammars. In Rustin, R., Ed., *Natural Language Processing.* Algorithmics Press, New York, New York: 145-149.

APPENDIX

```
/* Grammar rule preprocessor. */

:- public readRules/0,parse/3,go/0.

:- op(1000,xfy,(...)).
:- op(1100,xfy,:).

readRules :-
    tell(metgrr),
    repeat,
    read(Rule),
    process(Rule).

process(endRules) :-!,told.
process(Rule) :-!,
    parts(Rule,Head,Sem,Body),
    makex(Head,NT,Ext),
    makelist(Body,Body1),
    write(rule(NT,Ext,Sem,Body1)),write('.'),
    nl,nl,
    fail.
process(Clause) :-
    assertz(Clause),
    fail.

parts((Head:Sem --> Body),Head,Sem,Body) :-!.
parts((Head --> Body),Head,ℓ-true,Body).

makelist((X,L),[X | L1]) :-!,makelist(L,L1).
makelist([ ],[ ]) :-!.
makelist(X,[X]).

makex((NT,L),NT,Ext) :- !,makex1(nogap,L,Ext).
makex((NT...L),NT,Ext) :- !,makex1(gap,L,Ext).
makex(NT,NT,nil).

makex1(CT,(S,L),x(CT,Type,S1,X)) :-!,
    type(S,S1,Type),
    makex1(nogap,L,X).
makex1(CT,(S...L),x(CT,Type,S1,X)) :-!,
    type(S,S1,Type),
    makex1(gap,L,X).
makex1(CT,S,x(CT,Type,S1,nil)) :-
    type(S,S1,Type).

type([S],S,terminal) :-!.
type(S,S,nonterminal).

/* Parser */

parse(String,NonTerminal,Syn) :-
    prs([NonTerminal],String,[Syn],nil,nil,nil),!.

prs(BL,[D | X],Mods,Par,Mer,Ext) :-
    demon(D,BL,X,Mods,Par,Mer,Ext).
prs(BL,X,Mods,Par,merge(BL,Par1,Mer,Ext),nil) :-
    cutoff(Par),
    prs(BL,X,Mods,Par1,Mer,Ext).
prs([[W] | BL],[W | X],Mods,Par,Mer,Ext) :-
    gap(Ext),prs(BL,X,Mods,Par,Mer,Ext).
prs([[W] | BL],X,Mods,Par,Mer,x(_,terminal,W,Ext)):-
    prs(BL,X,Mods,Par,Mer,Ext).
```

```
prs([{B} | BL],X,Mods,Par,Mer,Ext) :-!,
    B,prs(BL,X,Mods,Par,Mer,Ext).
prs([NT | BL],X,Mods,Par,Mer,Ext) :-
    rule(NT,Ext0,Sem,Body),
    stack(Ext0,Ext,Ext1),
    prspush(Body,NT,BL,Sem,X,Mods,Par,Mer,Ext1).
prs([NT | BL],X,Mods,Par,Mer,x(_,nonterminal,NT,Ext)):-
    prs(BL,X,Mods,Par,Mer,Ext).
prs([ ],X,[ ],parent([_ | BL],Mods,Par),Mer,Ext) :-
    prs(BL,X,Mods,Par,Mer,Ext).
prs([ ],[ ],[ ],nil,nil,nil).

prspush([ ],_,BL,_,X,Mods,Par,Mer,Ext) :-!,
    prs(BL,X,Mods,Par,Mer,Ext).
prspush(Body,NT,BL,Sem,X,[syn(NT,Sem,Mods1) | Mods],Par,Mer,Ext) :-
    prs(Body,X,Mods1,parent([NT | BL],Mods,Par),Mer,Ext).

gap(x(gap,_,_,_)).
gap(nil).

stack(nil,X,X).
stack(x(C,T,S,X1),X2,x(C,T,S,X3)) :- stack(X1,X2,X3).

cutoff(parent([_ | BL],[ ],Par)) :- satisfied(BL), cutoff(Par).
cutoff(nil).

demon(D,BL,X,Mods,Par,Mer,Ext) :-
    conjunction(D,Cat,Sem),
    backup(Par,Mods,[syn(Cat,Sem,Mods0)],[NT | _],Par1),
    prs([NT],X,Mods0,nil,merge(BL,Par1,Mer,Ext),nil).

backup(parent(BL,Mods,Par),Mods0,Mods0,BL,parent(BL,Mods,Par)).
backup(parent([_ | BL],Mods,Par),[ ],Mods0,BL1,Par1) :-
    satisfied(BL),
    backup(Par,Mods,Mods0,BL1,Par1).

satisfied([ ]) :-!.
satisfied([NT | BL]) :- rule(NT,_,_,[ ]),!,satisfied(BL).
```

/* Semantic Interpretation Rules */

```
:- op(400,xfy,&).
:- op(300,fx,@).

/* Reshaping Rules */

reshape(Tree,Sisters,Tree1) :-
    daughters(Tree,Daus),
    reshapelist(Daus,Daus1),
    reorder(Daus1,Daus2),
    raise(Daus2,Tree,Sisters,Daus3),
    newdaus(Tree,Daus3,Tree1).

reshapelist([Tree | Trees],Trees2) :-!,
    reshapelist(Trees,Trees1),
    reshape(Tree,Sisters,Tree1),
    concat(Sisters,[Tree1 | Trees1],Trees2).
reshapelist([ ],[ ]).

reorder([A | L],M) :-
    reorder(L,L1),
    insert(A,L1,M).
reorder([ ],[ ]).
```

```
insert(A,[B | L],M) :-
   prec(A,PA),prec(B,PB),
   (PB>PA,!,M=[B | L1],insert(A,L,L1) |
     M=[A,B | L]).
insert(A,[ ],[A]).

raise([Dau | Daus],Tree,[Dau | Sisters],Daus1) :-
   above(Dau,Tree),!,
   raise(Daus,Tree,Sisters,Daus1).
raise([Dau | Daus],Tree,Sisters,[Dau | Daus1]) :-
   raise(Daus,Tree,Sisters,Daus1).
raise([ ],Tree,[ ],[ ]).

daughters(syn(_,_,Daus),Daus).

newdaus(syn(NT,Sem,_),Daus,syn(NT,Sem,Daus)).

cat(syn(NT,_,_),NT).

prec(Syn,P) :- stype(Syn,S),prec1(S,P),!.
prec(_,0).

stype(syn(nounph(_,Stype),_,_),Stype).

prec1(def,6).
prec1(all,6).
prec1(indef,4).

above(Syn1,Syn2) :-
   cat(Syn1,nounph(_,_)),cat(Syn2,Cat),
     ¬+ (Cat=relative(_) | Cat=conj(_)).

/* Translation Rules */

translate(Syn,LogForm1) :-
   transmod(Syn,ℓ-true,ℓ-LogForm),
   simplify(LogForm,[ ],LogForm1).

transmods([Mod | Mods],Sem1,Sem3) :-
   transmods(Mods,Sem1,Sem2),
   transmod(Mod,Sem2,Sem3).
transmods([ ],Sem,Sem).

transmod(syn(_,Sem,Mods),Sem1,Sem2) :-
   transmods(Mods,Sem,Sem0),
   trans(Sem0,Sem1,Sem2).

/*  Rules for 'trans' are numbered for     convenient reference in the text.  */

/*1*/  trans(Sem,C*D-P,cbase1(Sem,C,D)-P) :-!.
/*2*/  trans(cbase1(Sem,C,D)-P,Op-Q,
                    cbase2(Op,Sem,C,D,R)-true) :-!,
              conj(P,Q,R).
/*3*/  trans(Op-P,cbase2(Op1,Sem,C,D,B)-P1,
                    cbase2(Op2,Sem,C,D,B)-P2) :-!,
              trans(Op-P,Op1-P1,Op2-P2).
/*4*/  trans(cbase2(Op,Sem1,C,D,B)-P,Sem2,Op1-B) :-!,
              trans(Op-P,Sem2,Op1-C),
              trans(Sem1,Sem2,Op1-D).
/*5*/  trans(P/Q-R,Op-Q,@P-R).
/*6*/  trans(@P-Q,Op-P,Op-Q).
/*7*/  trans(ℓ-P,Op-Q,Op-R) :- conj(P,Q,R).
/*8*/  trans(r-P,Op-Q,Op-R) :- conj(Q,P,R).
/*9*/  trans(subst(X)-X,Sem,Sem).
/*10*/ trans(id-P,Sem,Sem).
```

```
conj(true,Q,Q) :-!.
conj(P,true,P) :-!.
conj(P,Q,P&Q).

/* Simplification Rules */

simplify(X,D,Y) :- var(X),!,find(X,D,Y).
simplify(E,_,E) :- atomic(E),!.
simplify(def(X,X=Y,E),D,E1) :-!, simplify(E,[X=Y | D],E1).
simplify(true&E,D,E1) :-!, simplify(E,D,E1).
simplify(E&true,D,E1) :-!, simplify(E,D,E1).
simplify(E,D,E1) :-
   E=..[P | Args],
   simplist(Args,D,Args1),
   E1=..[P | Args1].

find(X,[X1=Y | _],Y) :- X==X1,!.
find(X,[_ | D],Y) :- find(X,D,Y),!.
find(X,[ ],X).

simplist([E | L],D,[E1 | L1]) :-
   simplify(E,D,E1),
   simplist(L,D,L1).
simplist([ ],_,[ ]).
```

/* Syntax */

```
:- readRules.

sent --> nounph(X,_),verbph(X).

nounph(X,def):@P-def(X,X=N,P)  -->  [N],{prop(N)}.
nounph(X,Stype) -->
   det(X,Stype),adj(X),noun(X,Comps),comps(Comps),relative(X).
nounph(X,_) --> trace(X).

det(X,Stype):Sem --> [D],{deter(D,X,Stype,Sem)}.

adj(_) --> [ ].
adj(X):Sem --> [Adj],{adjec(Adj,X,Sem)}.

noun(X,Comps):ℓ-Pred --> [N],{n(N,X,Comps,Pred)}.

relative(X) --> [ ].
relative(X) --> open,rel_mk(X),sent,close.

open...close --> [ ].

rel_mk(X)...trace(X) --> [N],{rel_pro(N)}.
rel_mk(X)...[P],trace(X) --> [P],{prep(P)},[N],{rel_pro(N)}.

verbph(X) --> advl,verb(X,Comps),comps(Comps).

advl --> [ ].
advl:Sem --> [Adv],{adverb(Adv,Sem)}.

verb(X,Comps):ℓ-Pred --> [V],{v(V,Pred,X,Comps)}.

comps([ ]) --> [ ].
comps([X | L]) --> comp(X),comps(L).

comp(obj-X) --> nounph(X,_).
comp(pobj(Prep)-X) --> [Prep],nounph(X,_).

endRules.
```

```
/* Lexicon */

conjunction(and,conj(and),P*Q-(P&Q)).
conjunction(or,conj(or),P*Q-(P;Q)).
conjunction(but,conj(but),P*Q-but(P,Q)).

prop(john).
prop(bill).
prop(mary).

deter(a,X,indef,Q/P-exists(X,P,Q)).
deter(an,X,indef,Q/P-exists(X,P,Q)).
deter(the,X,def,Q/P-def(X,P,Q)).
deter(each,X,all,Q/P-each(X,P,Q)).
deter(every,X,all,Q/P-every(X,P,Q)).

adjec(red,X,ℓ-red(X)).
adjec(blue,X,ℓ-blue(X)).
adjec(glass,X,ℓ-glass(X)).

n(man,X,[ ],man(X)).
n(woman,X,[ ],woman(X)).
n(car,X,[ ],car(X)).
n(train,X,[ ],train(X)).
n(book,X,[ ],book(X)).
n(pencil,X,[ ],pencil(X)).
n(table,X,[ ],table(X)).
n(window,X,[ ],window(X)).
n(father,X,[pobj(of)-Y],father(X,Y)).
n(friend,X,[pobj(of)-Y],friend(X,Y)).
n(apple,X,[ ],apple(X)).
n(pear,X,[ ],pear(X)).

v(saw,saw(X,Y),X,[obj-Y]).
v(heard,heard(X,Y),X,[obj-Y]).
v(demolished,demolished(X,Y),X,[obj-Y]).
v(laughed,laughed(X),X,[ ]).
v(drove,drove_through(X,Y,Z),X,[obj-Y,pobj(through)-Z]).
v(gave,gave(X,Y,Z),X,[obj-Y,pobj(to)-Z]).
v(ate,ate(X,Y),X,[obj-Y]).
v(sat,sat_at(X,Y),X,[pobj(at)-Y]).

adverb(completely,@P-completely(P)).
adverb(finally,@P-finally(P)).

rel_pro(who).
rel_pro(whom).
rel_pro(that).
rel_pro(which).

prep(to).
prep(from).
prep(with).
prep(of).
prep(through).
```

EXAMPLES

```
|: Prolog-10  version 3

Input sentence:
|: each man ate an apple and a pear.

Syntactic analysis, time = 143 msec.

sent ℓ-true
   nounph(_1,all) ℓ-true
      det(_1,all) _2/_3-each(_1,_3,_2)
      noun(_1,[ ]) ℓ-man(_1)
   verbph(_1) ℓ-true
      verb(_1,[obj-_4]) ℓ-ate(_1,_4)
      comps([obj-_4]) ℓ-true
         comp(obj-_4) ℓ-true
            nounph(_4,indef) ℓ-true
               det(_4,indef) _5/_6-exists(_4,_6,_5)
               noun(_4,[ ]) ℓ-apple(_4)
               conj(and) _7*_8-_7&_8
                  nounph(_4,indef) ℓ-true
                     det(_4,indef) _9/_10-exists(_4,_10,_9)
                     noun(_4,[ ]) ℓ-pear(_4)

Reshaping tree, time = 16 msec.

sent ℓ-true
   nounph(_1,all) ℓ-true
      det(_1,all) _2/_3-each(_1,_3,_2)
      noun(_1,[ ]) ℓ-man(_1)
   nounph(_4,indef) ℓ-true
      det(_4,indef) _5/_6-exists(_4,_6,_5)
      noun(_4,[ ]) ℓ-apple(_4)
      conj(and) _7*_8-_7&_8
         nounph(_4,indef) ℓ-true
            det(_4,indef) _9/_10-exists(_4,_10,_9)
            noun(_4,[ ]) ℓ-pear(_4)
   verbph(_1) ℓ-true
      verb(_1,[obj-_4]) ℓ-ate(_1,_4)
      comps([obj-_4]) ℓ-true
         comp(obj-_4) ℓ-true

Semantic analysis, time = 22 msec.

each(_1,man(_1),exists(_2,apple(_2),ate(_1,_2))
       &exists(_2,pear(_2),ate(_1,_2)))

Input sentence:
|: john ate an apple and a pear.

Syntactic analysis, time = 144 msec.
Reshaping tree, time = 35 msec.
Semantic analysis, time = 11 msec.

exists(_1,apple(_1),ate(john,_1))&exists(_1,pear(_1),ate(john,_1))

Input sentence:
|: a man and a woman saw each train.

Syntactic analysis, time = 94 msec.

sent ℓ-true
   nounph(_1,indef) ℓ-true
```

```
      det(_1,indef) _2/_3-exists(_1,_3,_2)
      noun(_1,[ ]) ℓ-man(_1)
      conj(and) _4*_5-_4&_5
         nounph(_1,indef) ℓ-true
            det(_1,indef) _6/_7-exists(_1,_7,_6)
            noun(_1,[ ]) ℓ-woman(_1)
   verbph(_1) ℓ-true
      verb(_1,[obj-_8]) ℓ-saw(_1,_8)
      comps([obj-_8]) ℓ-true
         comp(obj-_8) ℓ-true
            nounph(_8,all) ℓ-true
               det(_8,all) _9/_10-each(_8,_10,_9)
               noun(_8,[ ]) ℓ-train(_8)
```

Reshaping tree, time = 24 msec.

```
sent ℓ-true
   nounph(_1,all) ℓ-true
      det(_1,all) _2/_3-each(_1,_3,_2)
      noun(_1,[ ]) ℓ-train(_1)
   nounph(_4,indef) ℓ-true
      det(_4,indef) _5/_6-exists(_4,_6,_5)
      noun(_4,[ ]) ℓ-man(_4)
      conj(and) _7*_8-_7&_8
         nounph(_4,indef) ℓ-true
            det(_4,indef) _9/_10-exists(_4,_10,_9)
            noun(_4,[ ]) ℓ-woman(_4)
   verbph(_4) ℓ-true
      verb(_4,[obj-_1]) ℓ-saw(_4,_1)
      comps([obj-_1]) ℓ-true
         comp(obj-_1) ℓ-true
```

Semantic analysis, time = 16 msec.

```
each(_1,train(_1),exists(_2,man(_2),saw(_2,_1))
       &exists(_2,woman(_2),saw(_2,_1)))
```

Input sentence:
|: each man and each woman ate an apple.

Syntactic analysis, time = 78 msec.

```
sent ℓ-true
   nounph(_1,all) ℓ-true
      det(_1,all) _2/_3-each(_1,_3,_2)
      noun(_1,[ ]) ℓ-man(_1)
      conj(and) _4*_5-_4&_5
         nounph(_1,all) ℓ-true
            det(_1,all) _6/_7-each(_1,_7,_6)
            noun(_1,[ ]) ℓ-woman(_1)
   verbph(_1) ℓ-true
      verb(_1,[obj-_8]) ℓ-ate(_1,_8)
      comps([obj-_8]) ℓ-true
         comp(obj-_8) ℓ-true
            nounph(_8,indef) ℓ-true
               det(_8,indef) _9/_10-exists(_8,_10,_9)
               noun(_8,[ ]) ℓ-apple(_8)
```

Reshaping tree, time = 14 msec.

```
sent ℓ-true
   nounph(_1,all) ℓ-true
      det(_1,all) _2/_3-each(_1,_3,_2)
      noun(_1,[ ]) ℓ-man(_1)
      conj(and) _4*_5-_4&_5
         nounph(_1,all) ℓ-true
            det(_1,all) _6/_7-each(_1,_7,_6)
            noun(_1,[ ]) ℓ-woman(_1)
   nounph(_8,indef) ℓ-true
      det(_8,indef) _9/_10-exists(_8,_10,_9)
      noun(_8,[ ]) ℓ-apple(_8)
   verbph(_1) ℓ-true
      verb(_1,[obj-_8]) ℓ-ate(_1,_8)
      comps([obj-_8]) ℓ-true
         comp(obj-_8) ℓ-true
```

Semantic analysis, time = 20 msec.

```
each(_1,man(_1),exists(_2,apple(_2),ate(_1,_2)))
       &each(_1,woman(_1),exists(_2,apple(_2),ate(_1,_2)))
```

Input sentence:
| : john saw and the woman heard a man that laughed.

Syntactic analysis, time = 182 msec.

```
sent ℓ-true
   nounph(_1,def) @_2-def(_1,_1=john,_2)
   verbph(_1) ℓ-true
      verb(_1,[obj-_3]) ℓ-saw(_1,_3)
   conj(and) _4*_5-_4&_5
      sent ℓ-true
         nounph(_6,def) ℓ-true
            det(_6,def) _7/_8-def(_6,_8,_7)
            noun(_6,[ ]) ℓ-woman(_6)
         verbph(_6) ℓ-true
            verb(_6,[obj-_3]) ℓ-heard(_6,_3)
            comps([obj-_3]) ℓ-true
               comp(obj-_3) ℓ-true
                  nounph(_3,indef) ℓ-true
                     det(_3,indef) _9/_10-exists(_3,_10,_9)
                     noun(_3,[ ]) ℓ-man(_3)
                     relative(_3) ℓ-true
                        rel_mk(_3) ℓ-true
                        sent ℓ-true
                           nounph(_3,_11) ℓ-true
                           verbph(_3) ℓ-true
                              verb(_3,[ ]) ℓ-laughed(_3)
```

Reshaping tree, time = 24 msec.

```
sent ℓ-true
   nounph(_1,def) @_2-def(_1,_1=john,_2)
   verbph(_1) ℓ-true
      verb(_1,[obj-_3]) ℓ-saw(_1,_3)
   conj(and) _4*_5-_4&_5
      nounph(_6,def) ℓ-true
         det(_6,def) _7/_8-def(_6,_8,_7)
         noun(_6,[ ]) ℓ-woman(_6)
      nounph(_3,indef) ℓ-true
```

```
      det(_3,indef) _9/_10-exists(_3,_10,_9)
      noun(_3,[ ]) ℓ-man(_3)
      relative(_3) ℓ-true
         nounph(_3,wh) ℓ-true
         rel_mk(_3) ℓ-true
         sent ℓ-true
            verbph(_3) ℓ-true
               verb(_3,[ ]) ℓ-laughed(_3)
   sent ℓ-true
      verbph(_6) ℓ-true
         verb(_6,[obj-_3]) ℓ-heard(_6,_3)
         comps([obj-_3]) ℓ-true
            comp(obj-_3) ℓ-true
```

Semantic analysis, time = 18 msec.

```
def(_1,woman(_1),exists(_2,man(_2)&laughed(_2),
    saw(john,_2)&heard(_1,_2)))
```

Input sentence:
|: john drove the car through and completely demolished a window.

Syntactic analysis, time = 80 msec.

```
sent ℓ-true
   nounph(_1,def) @_2-def(_1,_1=john,_2)
   verbph(_1) ℓ-true
      verb(_1,[obj-_3,pobj(through)-_4]) ℓ-drove_through(_1,_3,_4)
      comps([obj-_3,pobj(through)-_4]) ℓ-true
         comp(obj-_3) ℓ-true
            nounph(_3,def) ℓ-true
               det(_3,def) _5/_6-def(_3,_6,_5)
               noun(_3,[ ]) ℓ-car(_3)
         comps([pobj(through)-_4]) ℓ-true
            comp(pobj(through)-_4) ℓ-true
      conj(and) _7*_8-_7&_8
         verbph(_1) ℓ-true
            advl @_9-completely(_9)
            verb(_1,[obj-_4]) ℓ-demolished(_1,_4)
            comps([obj-_4]) ℓ-true
               comp(obj-_4) ℓ-true
                  nounph(_4,indef) ℓ-true
                     det(_4,indef) _10/_11-exists(_4,_11,_10)
                     noun(_4,[ ]) ℓ-window(_4)
```

Reshaping tree, time = 22 msec.

```
sent ℓ-true
   nounph(_1,def) @_2-def(_1,_1=john,_2)
   nounph(_3,def) ℓ-true
      det(_3,def) _4/_5-def(_3,_5,_4)
      noun(_3,[ ]) ℓ-car(_3)
   verbph(_1) ℓ-true
      verb(_1,[obj-_3,pobj(through)-_6]) ℓ-drove_through(_1,_3,_6)
      comps([obj-_3,pobj(through)-_6]) ℓ-true
         comp(obj-_3) ℓ-true
         comps([pobj(through)-_6]) ℓ-true
            comp(pobj(through)-_6) ℓ-true
      conj(and) _7*_8-_7&_8
         nounph(_6,indef) ℓ-true
```

```
            det(_6,indef) _9/_10-exists(_6,_10,_9)
            noun(_6,[ ]) ℓ-window(_6)
         verbph(_1) ℓ-true
            advl @_11-completely(_11)
            verb(_1,[obj-_6]) ℓ-demolished(_1,_6)
            comps([obj-_6]) ℓ-true
               comp(obj-_6) ℓ-true
```

Semantic analysis, time = 9 msec.

```
def(_1,car(_1),exists(_2,window(_2),
      drove_through(john,_1,_2)&completely(demolished(john,_2))))
```

Input sentence:
|: the woman who gave a book to john and
drove a car through a window laughed.

Syntactic analysis, time = 250 msec.
Reshaping tree, time = 108 msec.
Semantic analysis, time = 28 msec.

```
def(_1,woman(_1)&exists(_2,book(_2),
      exists(_3,car(_3),exists(_4,window(_4),gave(_1,_2,john)&
                drove_through(_1,_3,_4)))),laughed(_1))
```

Input sentence:
|: john saw the man that mary saw and bill heard.

Syntactic analysis, time = 87 msec.
Reshaping tree, time = 27 msec.
Semantic analysis, time = 25 msec.

```
def(_1,man(_1)&saw(mary,_1)&heard(bill,_1),saw(john,_1))
```

Input sentence:
|: john saw the man that heard the woman that laughed and saw bill.

Syntactic analysis, time = 174 msec.
Reshaping tree, time = 101 msec.
Semantic analysis, time = 23 msec.

```
def(_1,man(_1)&def(_2,woman(_2)&laughed(_2)&
      saw(_2,bill),heard(_1,_2)),saw(john,_1))
```

Input sentence:
|: the man that mary saw and john heard and bill gave a book to laughed.

Syntactic analysis, time = 1199 msec.
Reshaping tree, time = 106 msec.
Semantic analysis, time = 40 msec.

```
def(_1,man(_1)&saw(mary,_1)&exists(_2,book(_2),
      heard(john,_1)&gave(bill,_2,_1)),laughed(_1))
```

Input sentence:
|: the man that mary saw and heard gave an apple to each woman.

Syntactic analysis, time = 144 msec.

```
sent ℓ-true
   nounph(_1,def) ℓ-true
      det(_1,def) _2/_3-def(_1,_3,_2)
      noun(_1,[ ]) ℓ-man(_1)
      relative(_1) ℓ-true
         rel_mk(_1) ℓ-true
```

```
        sent ℓ-true
           nounph(_4,def) @_5-def(_4,_4=mary,_5)
           verbph(_4) ℓ-true
              verb(_4,[obj-_1]) ℓ-saw(_4,_1)
                 conj(and) _6*_7-_6&_7
                    verb(_4,[obj-_1]) ℓ-heard(_4,_1)
              comps([obj-_1]) ℓ-true
                 comp(obj-_1) ℓ-true
                    nounph(_1,_8) ℓ-true
  verbph(_1) ℓ-true
     verb(_1,[obj-_9,pobj(to)-_10]) ℓ-gave(_1,_9,_10)
     comps([obj-_9,pobj(to)-_10]) ℓ-true
        comp(obj-_9) ℓ-true
           nounph(_9,indef) ℓ-true
              det(_9,indef) _11/_12-exists(_9,_12,_11)
              noun(_9,[ ]) ℓ-apple(_9)
        comps([pobj(to)-_10]) ℓ-true
           comp(pobj(to)-_10) ℓ-true
              nounph(_10,all) ℓ-true
                 det(_10,all) _13/_14-each(_10,_14,_13)
                 noun(_10,[ ]) ℓ-woman(_10)
```

Reshaping tree, time = 26 msec.

```
sent ℓ-true
  nounph(_1,def) ℓ-true
     det(_1,def) _2/_3-def(_1,_3,_2)
     noun(_1,[ ]) ℓ-man(_1)
     relative(_1) ℓ-true
        nounph(_1,wh) ℓ-true
        nounph(_4,def) @_5-def(_4,_4=mary,_5)
        rel_mk(_1) ℓ-true
        sent ℓ-true
           verbph(_4) ℓ-true
              verb(_4,[obj-_1]) ℓ-saw(_4,_1)
                 conj(and) _6*_7-_6&_7
                    verb(_4,[obj-_1]) ℓ-heard(_4,_1)
              comps([obj-_1]) ℓ-true
                 comp(obj-_1) ℓ-true
  nounph(_8,all) ℓ-true
     det(_8,all) _9/_10-each(_8,_10,_9)
     noun(_8,[ ]) ℓ-woman(_8)
  nounph(_11,indef) ℓ-true
     det(_11,indef) _12/_13-exists(_11,_13,_12)
     noun(_11,[ ]) ℓ-apple(_11)
  verbph(_1) ℓ-true
     verb(_1,[obj-_11,pobj(to)-_8]) ℓ-gave(_1,_11,_8)
     comps([obj-_11,pobj(to)-_8]) ℓ-true
        comp(obj-_11) ℓ-true
        comps([pobj(to)-_8]) ℓ-true
           comp(pobj(to)-_8) ℓ-true
```

Semantic analysis, time = 22 msec.

```
def(_1,man(_1)&heard(mary,_1)&saw(mary,_1),
      each(_2,woman(_2),exists(_3,apple(_3),gave(_1,_3,_2))))
```