International Journal on Artificial Intelligence Tools, Vol. 1 No. 2 (1992) 229–277

NATURAL LANGUAGE PROCESSING WITHIN A SLOT GRAMMAR FRAMEWORK

MICHAEL MCCORD, ARENDSE BERNTH, SHALOM LAPPIN, and WLODEK ZADROZNY

IBM Thomas J. Watson Research Center, P. O. Box 704, Yorktown Heights, NY 10598, USA

Received 3 July 1991

ABSTRACT

This paper contains brief descriptions of the latest form of Slot Grammar and four natural language processing systems developed in this framework. Slot Grammar is a lexicalist, dependency-oriented grammatical system, based on the systematic expression of linguistic rules and data in terms of slots (essentially grammatical relations) and slot frames. The exposition focuses on the kinds of analysis structures produced by the Slot Grammar parser. These structures offer convenient input to post-syntactic processing (in particular to the applications dealt with in the paper); they contain in a *single* structure a useful combination of surface structure and logical form. The four applications discussed are: (1) An anaphora resolution system dealing with both NP anaphora and VP anaphora (and combinations of the two). (2) A meaning postulate based inference system for natural language, in which inference is done directly with Slot Grammar analysis structures. (3) A new transfer system for the machine translation system **LMT**, based on a new representation for Slot Grammar analyses which allows more convenient tree exploration. (4) A parser of "constructions", viewed as an extension of the core grammar allowing one to handle some linguistic phenomena that are often labeled "extragrammatical", and to assign a semantics to them.

1. Introduction

This paper contains brief descriptions of the latest form of Slot Grammar (McCord 1980, 1982, 1989b, 1990, 1991) and four natural language processing systems developed in the Slot Grammar framework, dealing with:

- Anaphora resolution,
- Inference,
- Machine translation,
- Treatment of "extragrammatical" constructions.

Slot Grammar is based on the systematic expression of linguistic rules and data in terms of *slots* and *slot frames*.

Syntactically, slots represent grammatical relations, and they have names like `subj` (subject) and `obj` (direct object). They come in two varieties: *complement slots* and *adjunct slots*. A list of complement slots – a *slot frame* – is associated with each word sense in the lexicon. A set of adjunct slots is associated with each part of speech in the

grammar. Thus the lexicon ultimately determines all the possible slots for a word sense, since a word sense has an associated part of speech.

Given a sentence (or phrase), a word `W` in it (with chosen lexical analysis), and a slot `S` associated with `W`, the grammar specifies when one can say of another word `V` (with chosen lexical analysis)

```
V fills slot S (of W).
```

For example, an English grammar might say that in

The man ate the chocolate.

man fills the `subj` slot of *ate*. Grammatical analysis of a sentence consists mainly in deciding upon lexical analyses for its words and determining the `fills` relationships between them.

Semantically, the complement slots of a word sense are like names for the arguments of the predicate corresponding to the word sense in logical form. In a non-trivial grammar, however, there are many more complement slots than the maximum number of arguments for a word sense predicate, because the syntactic means in natural language for specifying an argument of a word varies quite a bit with the type of word and argument.

The adjunct slots associated with a word sense (via its part of speech) typically correspond semantically to "outer" modifiers of the word sense predication, which either predicate intensionally on it or conjoin with it.

Slot Grammar is useful in two important ways as a methodology for writing natural language grammars:

(A) The analysis structures produced for sentences (or phrases) offer convenient input to post-syntactic processing, as in anaphora resolution and machine translation. These analysis structures contain in a *single* structure a useful combination of surface structure and logical form. The surface structures expressed are dependency trees – each node has an associated *head word* `W` and the daughter nodes are headed by words that fill slots of `W`. In addition, the slot frame of the head word is attached to the node and exhibits both deep and surface grammatical relations, including remote relations. The deep slot information for open-class words can be used for building their contributions to logical form.

(B) The Slot Grammar system facilitates the writing of practical, efficient, broad-coverage grammars, in the following ways:

1. The system is *lexicalist* in that much of the information needed for parsing resides in slot frames specified in the lexicon. Putting information in these small chunks makes it easier to expand and maintain a grammar (plus lexicon).[1]
2. Slot Grammar rules are divided up modularly into different types dealing with different phenomena, relating to slots. No phrase structure rules are used. For instance the basic rules that *fill* slots – *slot filler rules* – say nothing (normally) about left-to-right ordering of the fillers; and there are separate *slot ordering* rules. Separating these two phenomena not only simplifies the grammar of a given language, but it also makes it easier, given one grammar, to start up a grammar for another language, because languages tend to differ more in their ordering rules than in their slot inventories and slot filler rules.[2]
3. The Slot Grammar system includes a large *shell* which contains not only the basic parser (a bottom-up chart parser), but also a language-universal treatment of several grammatical phenomena such as coordination, extraposition, punctuation, and bracketing. Of course some of the information for the treatment of such phenomena is "parameterized" into language-specific grammars. But for the complete Slot Grammar of a given language, approximately 70% of the code resides in the shell.
4. Slot Grammar makes heavy use of a *numerical parse evaluation* system, consisting of heuristics that compare parses for likelihood or preference. These heuristics are used not only for ranking final analyses of a sentence, but can also (optionally) be used *during* parsing to rank partial analyses and prune the parse space. This pruning system makes parsing more efficient, and can also be viewed as defining grammatical constraints, since it reduces (considerably) the set of final analyses. Much of the parse evaluation system is in the shell, although language-specific evaluation rules and data can be given in the language-specific grammar and lexicon.
5. The system is implemented entirely in Prolog and makes crucial use of Prolog unification. Use of normal term unification instead of attribute/value unification probably contributes to efficiency.

Slot Grammars exist, in various stages of completeness, for English (McCord), German (Ulrike Schwall), Spanish (Gerardo Arrarte and Teofilo Redondo), Danish (Bernth), Norwegian (McCord and Bernth), and Hebrew (Shuly Wintner). In the following we refer to the English Slot Grammar as **ESG**.

Section 2 of the paper describes *Slot Grammar Analysis Structures*. We highlight this aspect not only because it gives a good idea of what Slot Grammar is "all about", but

---

1 The first version of Slot Grammar was developed in the period 1976-78 and was described in (McCord 1980). Apparently this was one of the first lexicalist, head-driven systems developed in a computational linguistic framework.

2 Modular treatment of ordering and slot-filling exists in other grammatical systems, such as those of Hudson (1971, 1976) and Pollard and Sag (1987). Modular rule treatment appears to go further in Slot Grammar, because there are other rule types, dealing for example with obligatoriness of slots. See (McCord 1991) for a complete discussion of the Slot Grammar rule formalism, and for more discussion of related systems.

also because the Slot Grammar output is the input to the applications described in subsequent sections.

In Section 3, *Anaphora Resolution*, we present four algorithms for handling different aspects of anaphora. The first algorithm contains a syntactic filter for identifying the pronoun-NP pairs in a sentence for which coreference is excluded on syntactic grounds. The second algorithm identifies the set of possible binding NP antecedents of each lexical anaphor (reflexive or reciprocal pronoun) in a sentence. The third algorithm selects the most likely NP antecedent for a pronoun, either in the same sentence, or in a preceding sentence. Finally, the fourth algorithm generates interpretations for a central class of elliptical VP constructions.

Section 4, *A Meaning Postulate Based Inference System for Natural Language*, describes a system for doing inference directly with the Slot Grammar analysis structures. Axioms, data, and queries can all be given in natural language (English). The system has procedures for extracting axioms from "if . . . then" sentences, and for applying forward and backwards inference on the analysis structures.

Section 5, *Transfer in* **LMT**, describes a new method for handling restructuring in the machine translation system **LMT** (McCord 1986, 1988, 1989a,c,d, Neff and McCord 1990, Rimon, McCord, Schwall and Martínez 1991, Bernth and McCord 1991). The description is essentially self-contained. The new method uses a new representation for Slot Grammar trees described in Section 2, and has the advantage of offering more freedom in referring to parts of the trees being restructured.

Section 6, *A Parser of Constructions*, presents an extension of the core grammar that allows us to handle some linguistic phenomena that are often labeled as "extragrammatical". They include constructions that cannot be handled easily by the Slot Grammar, such as reduplication or sentences like "The more you worry, the faster you eat". The parser presented in that section uses **ESG** to assign structure to fragments of such constructions; those fragments are then combined by the parser into a suitable analysis of the constructions.

2. Slot Grammar Analysis Structures

Although Slot Grammar is dependency-oriented and the basic ingredient of analysis is the slot-filler relation between words (with chosen lexical analyses), the Slot Grammar parser manipulates and produces analysis structures that are somewhat richer. These structures are labeled trees. Their skeletons are surface phrase structure trees, but the nodes are labeled to show complete slot-filler relations (including remote relations), as well as other information important for parsing.

An analysis tree for a sentence (normally) has a 1-1 correspondence between its nodes and the words of the sentence. The word corresponding to a node is called the *head word* of the node. Each node is labeled by a sense, feature structure, and slot frame of its head word (plus some more information that will be described below). The

daughters of a node consist basically of the phrase structure trees corresponding to words that fill slots of the head word.

There are exceptions, however, to this description of daughter nodes, when extraposition is involved. In a sentence like

What is John trying to cook?

what fills the object slot of *cook*. The parser handles this situation by *extraposing* this object slot to the level of *is*, where an *extraposed filler rule* fills it with *what*. Because of this, the phrase structure tree does not show the *what*-node as a daughter of the *cook*-node, but of the *is*-node. However, the complete tree, with node labeling, does exhibit this remote dependency relationship in the slot frame of the *cook*-node.

The *extent* of a node in an analysis tree is defined to be its head word together with the union of the extents of its daughter nodes. The exclusion of extraposed fillers from daughters in the tree results in keeping the extent of a phrase as a contiguous set of words.

In this section we discuss three different representations for Slot Grammar analysis structures:

1. the *nested-term* representation,
2. the *node-list* representation,
3. the *clausal* representation.

The first form is the data structure used by the parser (during parsing, and as the basic parser output). The other two forms can be produced by the system from the nested-term form, and are appropriate in different ways for applications. The clausal form is used in anaphora resolution (Section 3, below) as well as in **LMT** (Section 5). The node-list form is also used in **LMT**.

2.1. The nested-term representation

The *nested-term* representation of an analysis tree is given by a Prolog term

```
phrase(Span,Sense,Features,Frame,Ext,Mods),
```

where the components satisfy the following conditions:

1. The first component `Span` is a triple of positive integers

   ```
   LB . H . RB
   ```

   representing word positions in the sentence. `H` is the position of the head word of the phrase, and is called the *ID* of the phrase. `LB` and `RB` are respectively the *left*

and *right boundaries* of the phrase – the positions of the first and last words of its extent.

2. Sense is a symbol representing a sense of the head word.
3. Features is the *feature structure* of the phrase (node). It is a logic term (using standard positional notation, not attribute/value notation) which depends mainly on the chosen lexical analysis of the head word – exhibiting its part-of-speech as the principal functor and inflectional features as arguments. Some arguments of the feature structure may be used to represent characteristics of the whole phrase, like case for a German noun phrase, which are not determined solely by the head word.
4. Frame is the complement slot frame of the phrase, which will be discussed in more detail below.
5. Ext is the extraposed slot frame of the phrase. Its members are of the same form as those of Frame, and are put there by the parser while handling extraposition, as indicated in the above example.
6. The last component Mods represents the daughters of the phrase, but we normally call these the *modifiers* of the head word (or of the phrase). This component is of the form mods(LMods,RMods) where LMods and RMods are the lists of *left modifiers* and *right modifiers*, respectively. Order in these lists represents left-to-right surface order in the sentence, except that for the convenience of the parser, the members of RMods are listed in the reverse of surface order.

   Each member of a modifier list (left or right) is of the form Slot:Phrase where Slot is basically (details explained below) one of the slots of the head word – from Frame or Ext or an adjunct – and Phrase is a phrase which fills Slot.

It should be noted that in modifier lists a given slot can occur more than once, because adjunct slots can be filled multiply. Also the same complement slot name can appear more than once, arising from different levels due to extraposition. Thus modifier lists are not like attribute-value lists, where slots would correspond to attributes (having unique values).

Now let us look at the form of the slot frame of a phrase. It is a list (where the order has no significance for the grammar) of complement slots of the head word, each of which is of the form:

```
slot(SlotName,Ob,FillerLink,Test).
```

The four arguments of a complement slot are as follows.

1. The first argument, the *slot name*, is a term used in the grammar and lexicon to refer to the slot. Examples for **ESG** are obj(nlfin), which is a direct object that must be filled by (roughly) a noun phrase or a finite clause, and comp(p(tolwith)), which is a complement that must be filled by a PP headed by *to* or *with*. The slots obj(_) and comp(_) are examples of *or-slots*; their argument can be a disjunction of symbols that represent *options* – more specific types of fillers for the slot.

2. The second argument of a slot, the *obligatoriness indicator* `Ob`, equals `ob` or `op` according as the slot is *obligatory* or *optional* (must be filled or not). This is determined chiefly by markings on the slot in the lexicon.

3. The idea of the third argument of a slot, the *filler link*, is to identify how the slot actually gets filled during parsing. Normally, the filler link argument will be an unbound variable in an initial word frame. During parsing, when the slot is filled, the filler link will be bound to a term

   ```
   RealSlot:FillerID.
   ```

   Here `RealSlot` is created by the parser as follows. If the original slot is an or-slot (with non-trivial disjunction), then `RealSlot` is like the original slot but with its argument reduced to the actual option used. This reduced form of the or-slot is also used in building the contribution of the filler to the modifier list of the higher phrase.

   In the case of passive constructions, `RealSlot` will show the deep role of the filler; this situation is described in more detail below.

   The *filler ID* will just be the ID of the filler phrase. Thus in *John gave a book to the man*, the object slot of *gave* would have filler ID `4` since the head, *book*, of the filler has position `4` in the sentence; the complete filler link would be `obj(n):4`.

4. The last argument of a slot, the *slot test*, is in general a λ-expression which is applied as a test on the filler phrase (and must succeed, for the slot-filling to succeed). The most common sort of slot test is a Boolean combination of type tests applied specifically to the word sense of the filler. Such tests can be specified in a compact notation in the lexicon.

The slot frame of a phrase derives from the original lexical analysis of the head word. Lexical analysis determines a base form of the head word (with a choice of sense of the base), plus inflectional or derivational information on the relation of the word to the base. The slot frame of the base, given in the lexicon, may be altered (derived) to form the final slot frame of the head word.

For instance, if the head word is a passive past participle (like *given*), then the original, active slot frame of the base (like *give*) will be transformed to form one or more passive slot frames. Thus for the infinitive form *give* we might get the slot frame:

```
( slot(subj(n), ob, *, t) .
  slot(obj(n), op, *, t) .
  slot(iobj(n|to), op, *, t) . nil ).
```

(Here * is the IBM Prolog symbol for the anonymous variable. The filler link is unbound initially in the slot frame.) For the passive past partciple *given*, the system would produce the two frames:

```
( slot(agent, op, subj(n):*, t) .
  slot(subj(n), op, obj(n):*, op, *, t) .
  slot(iobj(n|to), op, *, t) . nil ).

( slot(agent, op, subj(n):*, t) .
  slot(obj(n), op, *, t) .
  slot(subj(n), op, iobj(n):*, t) . nil ).
```

In the first derived frame, the object is passivized (as in *The book was given to John*); in the second derived frame, the indirect object is passivized (as in *John was given a book*). Note that these frames show both surface and deep grammatical roles. The surface role is shown in the first argument of `slot(...)`, while the deep role is shown in the third (filler link) argument.

As indicated above, parsing is bottom-up. It is also head-driven. The lowest-level step of course involves the formation of initial, one-word `phrase` analyses for the words of the sentence (possibly more than one per word). Such a `phrase` analysis is of the form

```
phrase(H.H.H,Sense,Features,Frame,nil,mods(nil,nil))
```

where `H` is the position of the head word, and the next three arguments are determined by a lexical analysis of the head word. In the `Frame` all the filler IDs will be unbound.

Given these initial one-word analyses, one can think of parsing as proceeding middle-out, attaching slot fillers on the left or right of the head. In the process, filler links are bound and the modifier lists are augmented. The feature structure is changed normally only by unification, but there are facilities in the grammar formalism for really altering it when the `phrase` is updated with a new modifier.

There is a convenient display routine for looking at `phrase` analyses. Fig. 1 shows a parse display for *Alice gave Bob the book*. The slot frame assumed for *gave* is that discussed for *give*.

In the display there is only one line per node, with the information on that line centering on the head word for the node. On each line, the node information consists of: (1) the tree connection lines, (2) the slot filled by the node, (3) the sense predication (to be explained), and (4) the feature structure. The *sense predication* looks like a logic predication (atom) where the predicate is the word sense with the following arguments. The first argument is just the ID of the node (the position of the head word in the input string). The subsequent arguments (if any) are the filler IDs of the slots in the slot frame for the node. The order of these arguments is the same as that given in the slot frame. Thus slot frame order does matter for this display, although it does not matter for the grammar. There is an option for the display where the complete filler links are displayed instead of just the filler IDs, so that one can then see deep slot names in the sense predication.

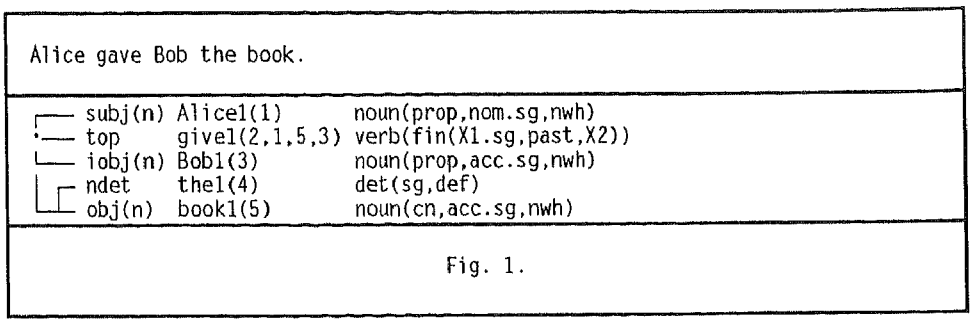


Fig. 1.

Two passivized slot frames for *given* were discussed above. Examples using these are shown in Fig. 2. These displays illustrate two options for abbreviation allowed by the system: (a) the word sense numbers are omitted, and (b) the arguments of feature structures are not shown.

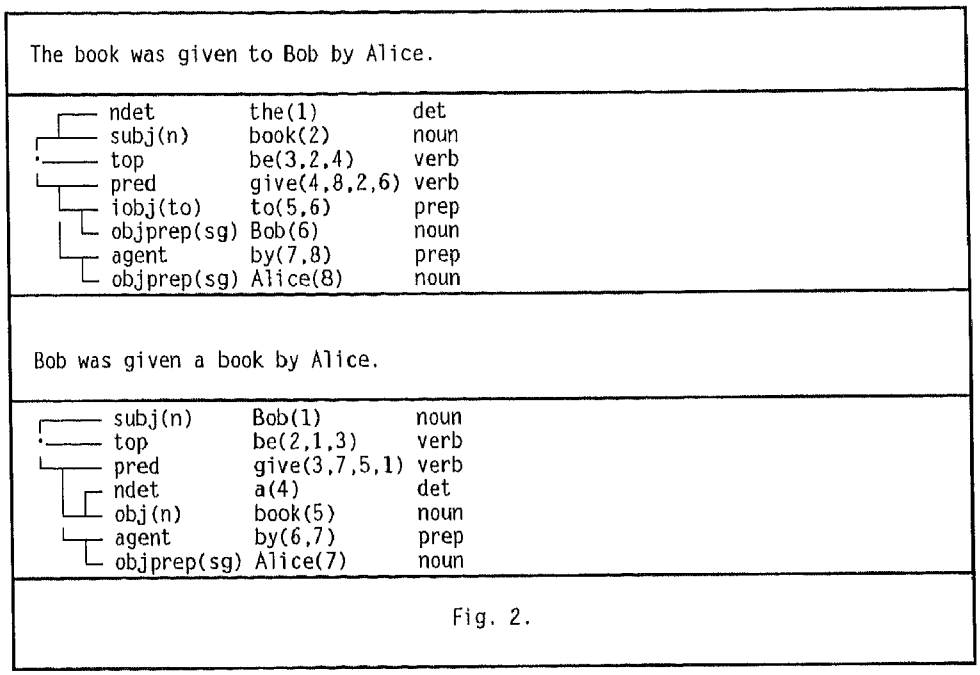


Fig. 2.

The final example given here, in Fig. 3, illustrates the treatment of extraposition in combination with coordination. The symbol u ("unfilled" or "unknown") is used in sense predications when the corresponding slot is unfilled (and cannot be determined by rules of control).

Note that the ID (1) of *who* is shown as both an argument of *find* (corresponding to its `obj(n)` slot) and an argument of *talk* (corresponding to `comp(p(with))`). Also note that

Alice (3) is shown as the subject of *did* and *say*, and *Bob* (5) is shown as the subject of *wanted*, *find*, and *talk*.

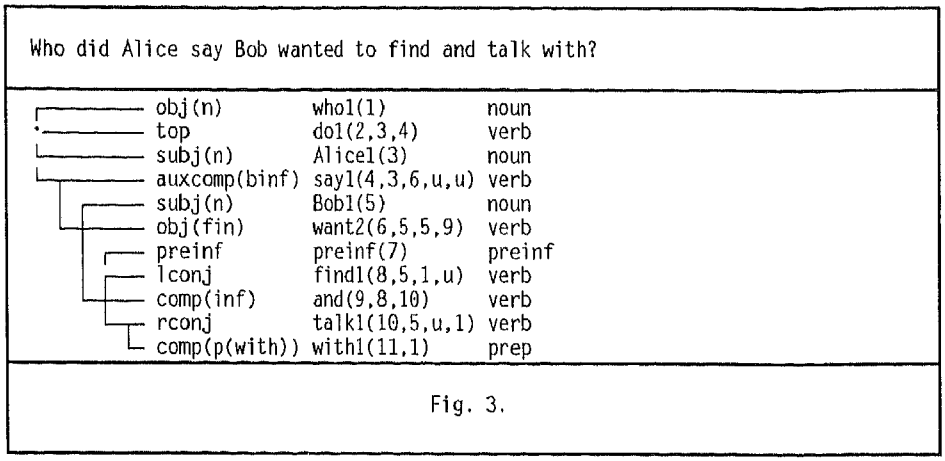


Fig. 3.

2.2. The node-list representation

Both this representation and the clausal representation take advantage of the indexing of nodes by head word numbers, which is available from the nested-term representation.

The *node-list* form of a Slot Grammar analysis is a list of terms

```
node(H,Sense,Features,Frame,Ext,Mods)
```

where the components are as follows:

- `H` is a head word ID (the integer giving the position of the word in the sentence). There is exactly one `node` for each word in the sentence.
- The next four components are identical to the corresponding components of a `phrase` structure.
- The last component, `Mods`, is a list of the form

  $$LM_1. \ \ldots \ .LM_m.\texttt{head}.RM_1. \ \ldots \ . \ RM_n.\texttt{nil}$$

  where the LM_i specify the left modifiers and the RM_i specify the right modifiers (both in left-to-right surface order). Each modifier specification is of the form `Slot:ID` where `Slot` is the slot filled by the modifier and `ID` is its (integer) ID. The member `head` of the list is literally the atom `head` and just marks the position of the head word between the left and right modifiers.

Thus the main difference between the nested-term (`phrase`) representation and the node-list representation is that in the latter, daughter (modifier) nodes are specified by their integer IDs – like pointers – instead of nested terms (other `phrases`).

The two representations are fully equivalent; each can be obtained automatically from the other.

The nested-term representation appears to be more convenient for the kind of structure-building and accessing that goes on during parsing. But the node-list representation can be more useful for post-syntactic modules that involve *transformations* of structures, where one would like to focus on one node (to transform it or its neighbors) but to be able to examine any part of the tree. The use of node-list representations for transformations in **LMT** will be described in Section 5 below.

2.3. The clausal representation

The *clausal* representation of a Slot Grammar analysis consists of a set of Prolog unit clauses, each containing information about a node. It is easiest to describe these clauses by explaining how they are obtained from a nested-term analysis. For each `phrase` node

```
phrase(LB.H.RB,Sense,Features,Frame,Ext,Mods),
```

we assert the following unit clauses:

- `ssense(H,Sense).`
- `sf(H,Features).`
- For each filled slot `slot(Slot,*,RSlot:K,*)` of the slot frame `Frame`, we assert the clauses

  ```
  sarg(H,Slot,K).
  srealarg(H,RSlot,K).
  ```

- For each modifier (left or right) `Slot:P` in `Mods`, where the ID of phrase `P` is `K`, we assert the clause

  ```
  smo(K,Slot,H).
  ```

  This can be read either as "`K` is a modifier of `H` (filling slot `Slot`)", or "The mother of `K` (through `Slot`) is `H`".

  When `H` is the top node of the sentence we also add the clause

  ```
  smo(H,top,0).
  ```

Note that the names for these clausal representation predicates all begin with "s". This is to prevent confusion with other, related Slot Grammar predicates that access parts of `phrase` structures. (For example `sense(P,Sense)` accesses the `Sense` component of a `phrase` `P`.) The "s" can be thought of as standing for "source"; there are similar predicates for examining parts of *target* trees in **LMT** and their names begin with "t".

```
Who was the man seen by?

┌── objprep(n) who1(1)     noun(pron(wh),X3.X4,wh(X5))
•── top         be1(2,4,5)  verb(fin(X1.sg,past,ind:q:wh(X2:1)))
│┌─ ndet        the1(3)     det(sg,def)
└└─ subj(n)     man1(4)     noun(cn,nom.sg,nwh)
 ┌─ pred        see2(5,1,4) verb(pastpart)
 └─ agent       by1(6,1)    prep(by,X7,1)

ssense(1,s(who,1)).   sf(1,noun(pron(wh),'X3'.'X4',wh('X5'))).
smo(1,objprep(n),2).

ssense(2,s(be,1)).    sf(2,verb(fin('X1'.sg,past,ind:q:wh('X2':1)))).
sarg(2,subj(n|v),4).  srealarg(2,subj(n),4)
sarg(2,pred,5).       srealarg(2,pred,5)
smo(2,top,0).

ssense(3,s(the,1)).   sf(3,det(sg,def)).
smo(3,ndet,4).

ssense(4,s(man,1)).   sf(4,noun(cn,nom.sg,nwh)).
smo(4,subj(n),2).

ssense(5,s(see,2)).   sf(5,verb(pastpart)).
sarg(5,agent,1).      srealarg(5,subj(n),1).
sarg(5,subj(n),4).    srealarg(5,obj(n),4).
smo(5,pred,2).

ssense(6,s(by,1)).    sf(6,prep(by,'X7',1)).
sarg(6,objprep(n),1). srealarg(6,objprep(n),1).
smo(6,agent,5).
```

Fig. 4.

An example of the clausal representation for the sentence

Who was the man seen by?

is shown in Fig. 4, following the normal display form for the phrase representation.

Note that in the clausal representation we have not included information about the extraposed slot list Ext (fifth component) of a phrase structure. This is because Ext does not seem particularly useful in applications of the clausal form. The original phrase structure is not quite recoverable without this information, but in case an application requires such recovery, one can include the extra information.

Left-to-right ordering of modifiers can be recognized in the clausal representation simply by normal ordering of the indexing integers.

The clausal representation is quite useful when one wants to explore a complete analysis tree from the point of view of any given node. It can be seen as a graph representation in which one can easily move in either direction along any arc. Using

Prolog clauses for the graph makes it convenient to mix calls to them with other Prolog goals in procedures that test properties of the tree.

It is also convenient to have these "facts" about the current analysis tree "in the background" (in the Prolog workspace) for work involving the tree.

Of course when more than one analysis is involved – either when one looks at alternative analyses of a given sentence or when one looks at more than one sentence at a time – one must either use extra indices in the clauses or delete the current clausal set when one goes on to a new analysis.

The clausal representation is quite useful for *testing* an analysis, but it does not appear so good when *transformations* of the analysis are required (as in LMT transfer transformations). As indicated above, the node-list representation is a compromise that shares with the clausal form some of the convenience of testing (and exploring) the whole tree, but is also good for transformations.

3. Anaphora Resolution

Identifying and interpreting anaphora is an important task in natural language processing. Specifically, in machine translation it is frequently necessary to know the antecedent of a pronoun in order to recover the feature, number, and target gender features which must be assigned to the pronoun's translation in a target language. Pronoun-antecedent relations are also important in the evaluation of natural language text for purposes of information retrieval and query response. For similar reasons, a robust wide-coverage natural language understanding system must also be able to resolve elided VP's (verb phrases) by assigning appropriate interpretations to elliptical VP structures. In most cases, these structures are understood as anaphorically dependent upon full VP's occurring elsewhere in the text.

We have developed two main systems for resolving anaphora within the SG framework. The first deals with pronominal anaphora. It contains three algorithms for identifying different aspects of the anaphoric relations among pronouns and possible NP antecedents. The second component generates interpretations for a central set of elliptical VP structures in English. We will consider each of these systems, and provide examples of their output.

3.1. NP anaphora

The NP anaphora system contains (i) an intra-sentential syntactic filter for ruling out (on syntactic grounds) anaphoric dependence of a pronoun on an NP (Lappin and McCord 1990a), (ii) a lexical anaphor binding algorithm for identifying the possible antecedent binders of a lexical anaphor (reciprocal or reflexive pronoun) within the same sentence (Lappin and McCord 1990b), and (iii) an algorithm for identifying the most

salient NP antecedent of a pronoun from a list of candidate antecedents in the text (Lappin and Leass 1992).

3.1.1. The syntactic filter on pronoun-NP coreference

The filter consists of six conditions for NP-pronoun non-coreference within a sentence. To state these conditions, we use the following terminology. The *agreement features* of an NP are its number, person, and gender features. We will say that a phrase P is in the *argument domain* of a phrase N iff P and N are both arguments of the same head. We will say that P is in the *adjunct domain* of N iff N is an argument of a head H, P is the object of a preposition PREP, and PREP is an adjunct of H. P is in the *NP domain* of N iff N is the determiner of a noun Q and (i) P is an argument of Q, or (ii) P is the object of a preposition PREP and PREP is an adjunct of Q.

A pronoun P is non-coreferential with a (non-reflexive or non-reciprocal) noun phrase N if any of the following conditions hold.

- P and N have incompatible agreement features.
- P is in the argument domain of N.
- P is in the adjunct domain of N.
- P is an argument of a head H, N is not a pronoun, and N is contained in H.
- P is in the NP domain of N.
- P is a determiner of a noun Q, and N is contained in Q.

The filter on pronominal anaphora restricts the search space of pronoun-NP pairs which the NP-antecedent selection algorithm (the third NP anaphora resolution algorithm) considers in the course of evaluating lists of possible antecedents for a given pronoun.

An example of the filter algorithm's output is given in 1.

```
1. This is the girl she wants Mary to talk to.

Non-coreferential pronoun-NP pairs:
she.5 - this.1, she.5 - girl.4, she.5 - Mary.7
```

3.1.2. The lexical anaphor binding algorithm

We take the set of lexical anaphors to include reflexive pronouns and the reciprocal NP "each other". The notion *higher argument slot* used in the formulation of the algorithm is defined by the hierarchy of argument slots given in 2.

```
2. subj > agent > obj > (iobj | pobj)
```

Here `subj` is the surface subject slot, `agent` is the deep subject slot of a verb heading a passive VP, `obj` is the direct object slot, `iobj` is the indirect object slot, and `pobj` is the object of a PP complement of a verb, as in "put NP on *NP*". We assume the definitions of *argument domain*, *adjunct domain*, and *NP domain* given in Section 3.1.1.

A noun phrase N is a possible antecedent binder for a lexical anaphor A iff N and A do not have incompatible agreement features, and one of the following five conditions holds.

- A is in the argument domain of N, and N fills a higher argument slot than A.
- A is in the adjunct domain of N.
- A is in the NP domain of N.
- N is an argument of a verb V, there is an NP Q in the argument domain or the adjunct domain of N such that Q has no noun determiner, and A is (i) an argument of Q, or (ii) A is an argument of a preposition PREP and PREP is an adjunct of Q.
- A is a determiner of a noun Q, and (i) Q is in the argument domain of N and N fills a higher argument slot than Q, or (ii) Q is in the adjunct domain of N.

An example of the anaphor binding algorithm's output is presented in 3.

```
3. They identified each other's portraits of themselves.

Antecedent NP-lexical anaphor pairs:
they.1 - (each . other).3, (each . other).3 - themselves.7
```

3.1.3. A syntactically based algorithm for pronominal anaphora resolution

The pronominal anaphora resolution algorithm incorporates the syntactic filter and the lexical anaphor binding algorithm into a procedure that assigns syntactically based salience values to NP antecedent candidates and selects the most highly valued of these candidates.[3]

The algorithm contains the following main components:

- The intra-sentential syntactic filter for ruling out anaphoric dependence of a pronoun on an NP on syntactic grounds

[3] Shalom Lappin and Herbert Leass (of the IBM Germany Scientific Center, Institute for Knowledge Based Systems, Heidelberg) developed the algorithm for pronominal anaphora resolution in joint research. They tested the algorithm on computer manual text. This work is described in Lappin and Leass (1992).

- A morphological filter for ruling out anaphoric dependence of a pronoun on an NP due to non-agreement of person, number, and gender features
- A syntactic procedure for identifying pleonastic (semantically empty) pronouns
- The anaphor binding algorithm for identifying the possible antecedent binder of a lexical anaphor within the same sentence
- Procedures for assigning values to several syntactic salience parameters (grammatical role, level of embedding, frequency of mention, and sentence recency) for an NP (earlier versions of these procedures are presented in (Leass and Schwall 1991)).
- A procedure for computing a global salience value for the equivalence class of NP's to which an NP is anaphorically linked as the sum of the salience values of the elements of the class.
- A grammatical role hierarchy according to which the evaluation rules assign higher salience weights to (i) subject over non-subject NP's, (ii) direct objects over other complements, (iii) arguments of a verb over adjuncts and objects of PP adjuncts of the verb, and (iv) head nouns over complements of head nouns[4]
- A decision procedure for selecting the preferred element of a list of antecedent candidates for a pronoun.

We can describe the pronominal anaphor resolution algorithm informally as follows. The algorithm assigns initial salience weights to the NP's of the current sentence S_i and degrades the salience values of each equivalence class EC of anaphorically linked NP's in preceding sentences.[5] All EC's in preceding sentences whose revised salience values are below a specified threshold are then deleted. For each non-pleonastic pronoun, if P is a lexical anaphor, then the anaphor binding algorithm is applied to generate a set of possible antecedent binding NP's in S_i. The NP in this set with the highest salience value is selected as P's antecedent. If P is not a lexical anaphor, then for each NP in S_i, the algorithm applies the syntactic and morphological filters to the pair <P,NP>, to generate a list A_i of possible antecedents of P in S_i. The morphological filter is applied to P and each element of the set of EC's from sentences which precede S_i to generate a list of possible antecedents A_{i-1} from previous sentences. Any EC which has been extended to include an NP in S_i that was excluded from A_i by the syntactic or the morphological filter is not included in A_{i-1}. A_i and A_{i-1} are combined to form a single list A of possible antecedents of P. The element of A which has the highest salience

4 This hierarchy is more or less identical to the NP accessibility hierarchy proposed in (Keenan and Comrie 1977). (Lappin 1985) employs it as a salience hierarchy to state a non-coreference constraint for pronouns. Guenthner and Lehman (1983) use a similar salience ranking of grammatical roles to formulate rules of anaphora resolution.

5 This procedure for degrading the salience values of equivalent classes of candidate antecedents in previous sentences in a text is inspired, in part, by (Alshawi 1987).

ranking is selected as P's antecedent. If two or more elements of A are equally ranked, then the NP which is most proximate to P is selected.

It is important to recognize that the algorithm relies exclusively on syntactic and morphological filtering, a syntactic salience measure, recency, and frequency of occurrence in selecting the antecedent of a pronoun. The anaphora resolution procedures do not employ semantic constraints (except for a highly restricted set of feature matching conditions incorporated into the morphological filter) or real world knowledge conditions.

The process of degrading the salience values of EC's in preceding sentences each time the algorithm applies to a new sentence, the assignment of an initial salience value to sentence recency, and the use of proximity to select an antecedent when two candidates are equally ranked for the highest salience value causes the algorithm to strongly prefer intra-sentential to inter-sentential antecedents.

Three examples of the algorithm's output are given in 4, 5-7, and 8, respectively. These examples are taken from IBM computer manual texts.

Pleonastic and non-lexical anaphoric pronouns in the same sentence

```
4. Most of the copyright notices are embedded in the EXEC, but this
keyword makes it possible for a user-supplied function to have its own
copyright notice.

Non-coreferential pronoun-NP pairs:
it.16 - most.1, it.16 - notice.5,
it.16 - keyword.14,
it.16 - function.23,
it.16 - user.20, it.16 - notice.29,
it.16 - copyright.28, its.26 - most.1,
its.26 - notice.5,
its.26 - notice.29,
its.26 - copyright.28

Pleonastic Pronouns:
it.16

Anaphor-Antecedent links:
its.(1.26) to function.(1.23)
```

Inter-sentential and intra-sentential pronominal chaining

```
5. This function is only available in the WSDICT interface.

6. Though it is entered as a command, it is actually a WordSmith
application that creates a box at the cursor location.

Non-coreferential pronoun-NP pairs:
it.2 - command.7,
it.9 - application.14,
it.9 - WordSmith.13,
```

```
it.9 - box.18, it.9 - location.22,
it.9 - cursor.21

Anaphor-Antecedent links:
it.(2.2) to function.(1.2)
it.(2.9) to it.(2.2)

7. It relies on having an entry in the resource table.

Non-coreferential pronoun-NP pairs:
it.1 - entry.6, it.1 - table.10,
it.1 - resource.9

Anaphor-Antecedent links:
it.(3.1) to it.(2.9)
```

Cataphora

```
8. In order to add their own commands to WordSmith, users should be
familiar with the system components of WordSmith.

Non-coreferential pronoun-NP pairs:
their.5 - order.2, their.5 - command.7,
their.5 - WordSmith.9,
their.5 - system.17

Anaphor-Antecedent links:
their.(1.5) to user.(1.11)
```

We applied the algorithm to sentences containing occurrences of third person pronouns (and their antecedents) extracted from six computer manuals containing approximately 11,000 lines of text. It correctly resolved 475 (85%) out of a total of 560 pronoun occurrences. In this set of cases we found 89 (16%) instances of inter-sentential anaphora. The algorithm correctly resolved 72 (81%) of these. The syntactic-morphological filter reduced the set of possible antecedents to a single NP, or identified the pronouns as pleonastic, in 163 (34%) of the 475 cases which the algorithm resolves correctly.

The algorithm's relatively high rate of success indicates the viability of a computational model of anaphora resolution in which the relative salience of an NP in discourse for purposes of anaphora resolution is determined, in large part, by syntactic factors. In this model, semantic and real world knowledge conditions apply to the output of an algorithm which resolves pronominal anaphora on the basis of syntactic measures of salience and recency. These conditions are used to evaluate the anaphoric relations which the algorithm generates.

Ido Dagan and Amnon Ribak of the IBM Scientific Center in Haifa have incorporated statistically based procedures for revising salience rankings of antecedents into our algorithm. These procedures are based upon the frequency counts of head-argument and head-adjunct pairs in the Slot Grammar parses of sentences in corpora. They re-evaluate the salience ranking of a less highly valued possible

antecedent NP when the frequency count for the relation of the NP to the head of which the pronoun is an argument or an adjunct is above a certain threshold. The statistical procedures will effectively perform the role of semantic and pragmatic factors which filter and re-evaluate the salience rankings that the syntactically based algorithm for anaphora resolution generates. We are currently testing the effectiveness of these statistical procedures.

3.2. VP anaphora

Consider the elliptical VP structures in 9 and 10.

9. John arrived today, and Bill did too.

10. Max wrote more notes to Mary than Max may have.

In each case, the elliptical VP in the second clause is anaphorically dependent upon the full VP in the preceding clause. We treat elliptical VP's as structured empty, or partially empty head-argument structures in which the head verb is anaphorically dependent upon the head verb of an antecedent VP. Generating an interpretation of an elliptical VP involves identifying the head of its antecedent, and determining which of its arguments and adjuncts are to be copied along with the antecedent head into the position of the elliptical VP. This analysis also applies to elliptical structures in which selected arguments or adjuncts appear in the elliptical site, as in 11 and 12 (these structures are commonly referred to in the linguistics literature as instances of *subdeletion*).[6]

11. John wrote more letters to Mary than Max did to Lucy.

12. John arrived today, and Bill will tomorrow.

Our algorithm for VP anaphora interpretation identifies the antecedent head of an elliptical VP, filters the arguments and adjuncts of the antecedent to determine which of them will be inherited by the interpreted head of the elliptical VP, and then generates a tree in which the reconstructed VP is substituted for the elliptical VP. The following is a schematic description of the algorithm.

A. Identify an elliptical verb-antecedent verb pair <V,A> as follows.

6 See (Lappin and McCord 1990b) and (Lappin 1991) for detailed discussion of this approach to VP ellipsis. These papers also present extensive motivation for adopting this approach as opposed to others which have been proposed in both the theoretical linguistics and the computational literature.

1. An elliptical verb V is identified by the presence of an auxiliary verb or the infinitival complementizer "to", where the auxiliary verb or the complementizer does not have a realized verb complement.
2. A candidate A for an antecedent of V is a verb which is not elliptical and not an auxiliary verb with a realized complement.
3. Check that A and V stand in at least one of the following relations:
   a. V is contained in the clausal complement of a subordinate conjunction SC, and the SC-phrase is either (i) an adjunct of A, or (ii) an adjunct of a noun N and N heads an NP argument of A, or N heads the NP argument of an adjunct of A.
   b. V is contained in a relative clause which modifies a head noun N, N is contained in A, and if A' is a verb, N is contained in A', and A' is contained in A, then A' is either an infinitival complement of A, or of an infinitival complement of a verbal complement of A.
   c. V is contained in the right conjunct of a sentential conjunction S, and A is contained in the left conjunct of S.

B. Take A as the new interpreted head of the elliptical verb phrase VP' which V heads. (We will refer to this new occurrence of A as A').

C. Consider in sequence each argument slot $Slot_i$ in the argument frame of A.
   1. If $Slot_i$ is filled by a phrase C, then
      a. If there is a phrase C' in VP' which is of the appropriate type for filling $Slot_i$, then fill $Slot_i$ in the argument frame of A' with C'. Else,
      b. Fill $Slot_i$ in A' with C, and list C as a new argument of A'.
   2. If $Slot_i$ is empty in the frame of A, it remains empty in the frame of A'.
   3. Construct a list, Arg-List, of the phrases which fill the argument slots of A'.

D. Consider each Adjunct Adj of A.
   1. If there is no adjunct of the same type as Adj in VP', then list Adj as a new adjunct of A'.
   2. Construct a list, Adj-List, of the phrases which fill adjunct slots of A'.

E. Generate a new tree as follows.
   1. Concatenate Arg-List and Adj-List to create a combined list, Ph-List, of the phrasal arguments and adjuncts of A' which appear to the right of A'.
   2. Reorder the elements of Ph-List to produce a new list, Ord-Ph-List in which the sequence of argument and adjunct types corresponds to the order of postverbal argument and adjunct phrases of A.
   3. Substitute A' for V in the old tree, and assign A' the agreement features required by the head of the elliptical verb phrase VP'.

4. Substitute Ord-Ph-List for the list of postverbal argument and adjunct phrases of V in VP'.

Here are some examples of the algorithm's output for a variety of full and partially elliptical VP structures. In each example, the output shown is the interpreted VP anaphora tree.

13. John read everything which Mary did.

```
subj(n) John1(1)          noun
top     read1(2,1,3,u,u)  verb
obj(n)  everything1(3)    noun
 obj(n) which2(4)         noun
 subj(n) Mary1(5)         noun
 nrel    read1(6,5,4)     verb
```

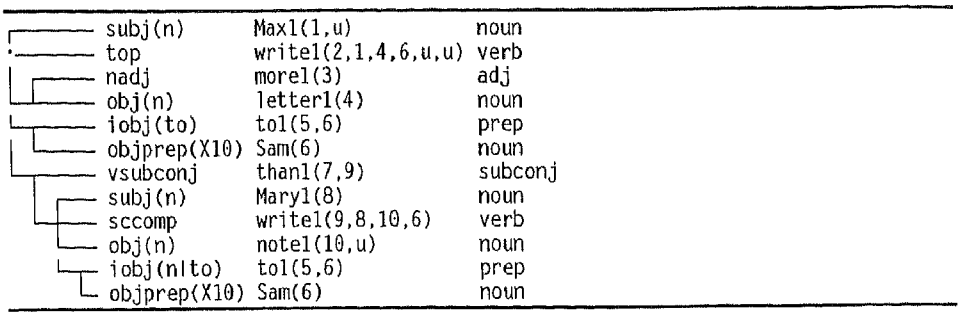
14. John spoke to Mary about Max before Bill asked Lucy to.

```
subj(n)              John1(1)           noun
top                  speak1(2,1,6,4)    verb
comp(p(to))          to1(3,4)           prep
 objprep(sg)         Mary1(4)           noun
obj(p(about))        about1(5,6)        prep
 objprep(sg)         Max1(6,u)          noun
vsubconj             before2(7,9)       subconj
 subj(n)             Bill1(8)           noun
 sccomp              ask1(9,8,11,10)    verb
 iobj(n)             Lucy(10)           noun
 obj(inf)            preinf(11,10,12)   preinf
  auxcomp(inf(bare)) speak1(12,10,6,4)  verb
  comp(p(to|with))   to1(3,4)           prep
   objprep(sg)       Mary1(4)           noun
  obj(p(about))      about1(5,6)        prep
   objprep(sg)       Max1(6,u)          noun
```

15. John wrote letters to Mary before Bill did to Lucy.

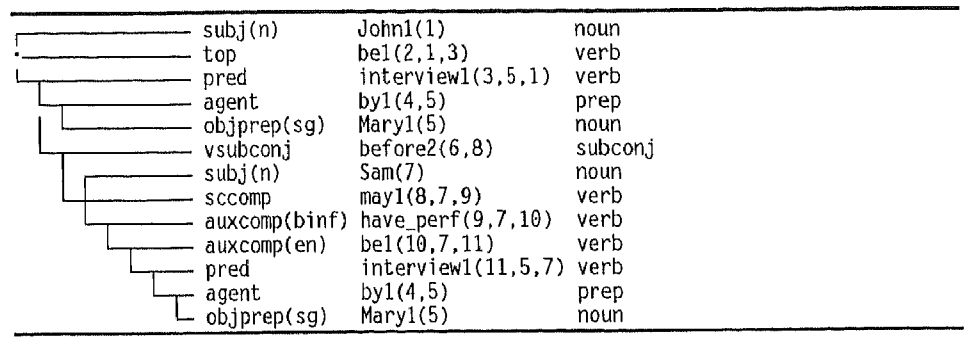
```
subj(n)                 John1(1)              noun
top                     write1(2,1,3,5,u,u)   verb
obj(n)                  letter1(3)            noun
iobj(to)                to1(4,5)              prep
 objprep(sg)            Mary1(5)              noun
vsubconj                before2(6,8)          subconj
 subj(n)                Bill1(7)              noun
 sccomp                 write1(8,7,3,10)      verb
 obj(n|thatc|finq|wh)   letter1(3)            noun
 iobj(n|to)             to1(9,10)             prep
  objprep(X12)          Lucy(10)              noun
```

16. Max writes more letters to Sam than Mary does notes.



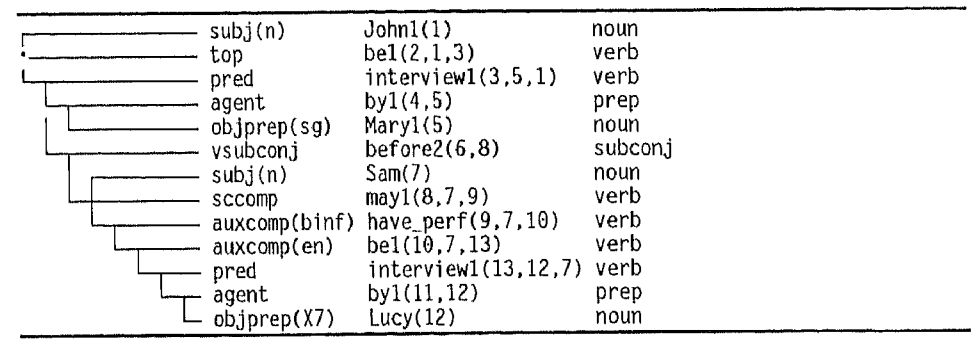
subj(n)	Max1(1,u)	noun
top	write1(2,1,4,6,u,u)	verb
nadj	more1(3)	adj
obj(n)	letter1(4)	noun
iobj(to)	to1(5,6)	prep
objprep(X10)	Sam(6)	noun
vsubconj	than1(7,9)	subconj
subj(n)	Mary1(8)	noun
sccomp	write1(9,8,10,6)	verb
obj(n)	note1(10,u)	noun
iobj(n|to)	to1(5,6)	prep
objprep(X10)	Sam(6)	noun

17. John was interviewed by Mary before Sam may have been.



subj(n)	John1(1)	noun
top	be1(2,1,3)	verb
pred	interview1(3,5,1)	verb
agent	by1(4,5)	prep
objprep(sg)	Mary1(5)	noun
vsubconj	before2(6,8)	subconj
subj(n)	Sam(7)	noun
sccomp	may1(8,7,9)	verb
auxcomp(binf)	have_perf(9,7,10)	verb
auxcomp(en)	be1(10,7,11)	verb
pred	interview1(11,5,7)	verb
agent	by1(4,5)	prep
objprep(sg)	Mary1(5)	noun

18. John was interviewed by Mary before Sam may have been by Lucy.



subj(n)	John1(1)	noun
top	be1(2,1,3)	verb
pred	interview1(3,5,1)	verb
agent	by1(4,5)	prep
objprep(sg)	Mary1(5)	noun
vsubconj	before2(6,8)	subconj
subj(n)	Sam(7)	noun
sccomp	may1(8,7,9)	verb
auxcomp(binf)	have_perf(9,7,10)	verb
auxcomp(en)	be1(10,7,13)	verb
pred	interview1(13,12,7)	verb
agent	by1(11,12)	prep
objprep(X7)	Lucy(12)	noun

19. John arrived yesterday, and Mary did too.

```
┌─ subj(n) John1(1)        noun
├─ lconj   arrive1(2,1,u)  verb
└─ vadv    yesterday1(3)   noun
·─ top     and(5,2,7)      verb
│┌─ subj(n) Mary1(6)       noun
└┼─ rconj   arrive1(7,6,u)  verb
 └─ vadv    yesterday1(3)   noun
  └─ vadv   too2(8)         adv
```

20. John arrived yesterday, and Mary will tomorrow.

```
┌──── subj(n)             John1(1)        noun
├──── lconj               arrive1(2,1,u)  verb
└──── vadv                yesterday1(3)   noun
·──── top                 and(5,2,7)      verb
│┌──── subj(n)            Mary1(6)        noun
└┼──── rconj              will1(7,6,9)    verb
 └──── auxcomp(inf(bare)) arrive1(9,6,u)  verb
  └──── vadv              tomorrow1(8)    noun
```

At this point, we have integrated the syntactic filter and the lexical anaphor binding algorithm into the VP anaphora algorithm. An example of the output of this integrated anaphora resolution system is given in 21.

21. They discussed each other's portraits of themselves before John and Mary may have.

```
┌──────────── subj(n)              they1(1)              noun
·──────────── top                   discuss1(2,1,5)       verb
│┌─────────── nadj                  p(each.other,3)       noun
├┼─────────── nposs                 's                    special
│└─────────── obj(n)                portrait1(5,7)        noun
├┼─────────── nobj(n)               of1(6,7)              prep
│└─────────── objprep(pl)           themselves1(7)        noun
└┼─────────── vsubconj              before2(8,12)         subconj
 ├┌─────────── lconj                 John1(9)              noun
 ├┼─────────── subj(n)              and(10,9,11)          noun
 ├┼─────────── rconj                Mary1(11)             noun
 ├┼─────────── sccomp               may1(12,10,13)        verb
 ├┼─────────── auxcomp(binf)        have_perf(13,10,14)   verb
 ├┼─────────── auxcomp(pastparta)   discuss1(14,10,5)     verb
 │├┌─────────── nadj                p(each.other,3)       noun
 │└┼─────────── nposs               's                    special
 │ ├─────────── obj(n|ing)          portrait1(5,7)        noun
 │ ├─────────── nobj(n)             of1(6,7)              prep
 │ └─────────── objprep(pl)         themselves1(7)        noun
```

```
Non-coreferential pronoun-NP pairs:
  they.1 - portrait.5, they.1 - John.9,
  they.1 - coord(and,s(John,1),s(Mary,1)).10, they.1 - Mary.11

Antecedent NP-anaphor pairs:
  they.1 - (each . other).3, (each . other).3 - themselves.7,
  coord(and,s(John,1),s(Mary,1)).10 - (each . other).3
```

We will soon integrate the full algorithm for pronominal anaphora resolution into the VP anaphora resolution algorithm.

It is important to point out that each of the algorithms discussed in this section makes crucial use of the clausal representation of phrase structure discussed in Section 2.3 above. Specifically, the various tests and constraints which the algorithms apply to syntactic structures are invoked as conditions on the clausal representations of input sentences. Thus, for example, it is possible to use the clausal representation of a sentence to define the notion of an *argument domain* involved in the definition of the syntactic filter on coreference which we presented in Section 3.1.1. Recall from Section 2.3 that `sarg(H,Slot,K)` means node `H` has node `K` as an argument (filling slot `Slot`). With such clausal information available, we can say that `A` is in the *argument domain* of `B`[7] iff there is a head `H` such that

```
sarg(H,*,A) & sarg(H,*,B).
```

In the sentence

```
This is the girl she wants Mary to talk to.
```

for example, `Mary` is in the argument domain of `she` because both are `sarg`'s of `want`.

4. A Meaning Postulate Based Inference System for Natural Language

Disambiguation of word senses is important for natural language processing, not the least for machine translation, since different senses may translate into different target words. Often word senses can only be determined by inference on the context. This section describes a system, **LEXSEM** (see also Bernth and Lappin, 1991), for deriving inferences from sentences on the basis of meaning postulates. The main use of this system has been for lexical disambiguation, even though it is a general-purpose inference system for natural language.

LEXSEM contains a procedure for constructing meaning postulates (axioms) from English sentences in "if . . . then" form, with category variables used to represent both phrasal and lexical constituents. It constructs forward and backward inference chains directly from the syntactic representations of natural language sentences, without mapping these sentences (or their parses) into a logical form representation. It provides natural language sentences and parse representations as output. Therefore, both the construction of the meaning postulates which support the inference process and the inference procedures themselves operate entirely on natural language input, and yield natural language output.

---

7 This formulation is slightly simplified; one must do a bit more to take account of argument relations in the presence of relative clauses. But this can be done solely in terms of the clausal representation of Section 2.3.

Meaning postulates have generally been formulated as constraints on possible models for logical form languages into which the expressions of natural languages are translated.[8] The inference systems which AI researchers have developed have, by and large, conformed to this design of translating natural language sentences into a logical form language in which meaning postulates, as well as knowledge base axioms, the data base, and inference procedures are specified.[9] An obvious difficulty with this approach lies in constructing an adequate algorithm for mapping natural language phrases into appropriate expressions of the formal language. Constructing such a translation algorithm for a system that applies to a wide range of syntactic structures is a non-trivial and cumbersome task. It requires that the logical (denotational) type of every category of natural language phrase be determined. In order for the algorithm to assign a type to each phrase, it must resolve all syntactic ambiguities in the input expression which are relevant to identifying the type, or set of types, to which it corresponds.

Since the syntactic structures provided by the Slot Grammar parser provide the formal representations for both the specifications of meaning postulates and the application of inference procedures to natural language input to **LEXSEM**, these difficulties connected with translation into logical form (as well as generation of natural language output) are completely avoided in our system.

The translational approach also encounters problems with the rich variety of syntactic categories in natural language. Sentential and VP adverbs, relative clauses, PP modifiers of NP's, and adjectives taking tensed or infinitival complements (such as *certain* or *likely*), to cite several examples, are not easily mapped onto the logical types of a formal system in a way which is both uniform and fully expressive of the semantic properties exhibited by the elements of such categories.

In contrast, **LEXSEM** retains the full expressive power of natural language syntax because the inference procedures operate directly on the parse structures of natural language sentences. Furthermore, the system allows us to retain flexible control over the amount of syntactic structure and the semantic features specified in the meaning postulates. Meaning postulates can contain phrasal variables to represent phrasal constituents of any size (up to entire clauses), as well as lexical variables, which stand for lexical items of particular syntactic categories with specific complement structures. The range of a variable can be restricted to a particular syntactic or semantic subset of a lexical or phrasal category.

---

8 See e.g. (Carnap 1947), (Montague 1974) and (Dowty 1989).

9 See for example, (Hobbs et al. 1988) and (Schubert and Hwang 1989) for recent examples of inference systems formulated in enriched first order type languages which use both meaning postulates and knowledge base axioms. (Lehmann 1988) describes LEX, a legal expert system, in which a data base of case descriptions is represented as as sets of Discourse Representation Structures (DRS's) in the sense of (Kamp 1981), and inferences are performed on these structures. Meaning postulates and principles encoding real world (particularly legal) knowledge are formulated as rules defining relations among types of DRS's. Bernth (1989 and 1990) proposes an inference system in which natural language input sentences are translated into Prolog-like logical forms. These are used to construct situation semantics style representations of discourse segments on which inference rules operate.

The problem of constructing an effective translation algorithm from natural language expressions to decompositional models of lexical semantic representation and representation of the sort proposed in (Jackendoff 1976, 1983, 1987, and 1990) is even more acute than in the case of translation into a formal logical system. Unlike the formal logical languages, the syntax of the representation language (the so-called lexical conceptual structure or lcs language) is not explicitly formalized, and so it is not clear how to devise a general category-type correspondence between natural language phrases and lcs structures. Generally, ad hoc and unsystematic linking rules have been used.[10] These rules do not provide a uniform and general procedure for identifying components of lcs's with constituents of natural language phrases. Moreover, unlike formal systems, no model theory has been proposed for the lcs language, and so it is not clear how expressions in the lcs language are themselves interpreted, or in what sense translation into the lcs language induces an indirect interpretation of natural language phrases.

In **LEXSEM**, lexical items are not decomposed by translation into complex expressions of another language, although they may be associated with complex phrases in the same language through meaning postulates. The semantic properties of a lexical item are expressed by the set of postulates which define the inference patterns in which it can appear, and so formalize its semantic relations to other lexical items (and classes of lexical items) in the language.

In Section 4.1, the meaning postulates and category variables are described. Section 4.2 contains a schematic account of the algorithm for constructing forward inference chains. In Section 4.3, **LEXSEM** as an inference-based natural language query system (supported by backwards inference with negation-by-failure) is described. This query system permits us to use **LEXSEM** to resolve lexical ambiguity for purposes of certain kinds of lexical disambiguation and text understanding tasks.

4.1. Constructing meaning postulates in natural language

LEXSEM recognizes sentences of the form "If S_1 . . . then S_2" as meaning postulates.[11] In general, meaning postulates will contain category variables, but this is not a necessary condition for an "if . . . then" sentence to be a postulate. Category

---

10 Such rules frequently take the form of "Agent goes to (d-structure) subject", "Theme goes to (d-structure) object; else to (d-structure) subject", where roles like agent and theme are further defined in terms of an argument position in an lcs type. See (Rappaport and Levin 1988), (Pinker 1989), (Jackendoff 1990), and (Grimshaw 1990) for examples of linking rules.

11 While **LEXSEM** will accept any postulate of this form as an axiom, we are focussing on those postulates which express lexically based semantic implications, and so are meaning postulates in the classical sense of (Carnap 1947), (Montague 1974), and (Dowty 1989). On this view, meaning postulates are constraints on the set of possible models which insure that the non-logical primitives of one's language (ie. the elements of the lexicon) receive the intended interpretations in each admissible model. Clearly, the distinction between meaning postulates in this sense and postulates which express general truths about the world is not absolute. (See the discussion in (Quine 1966) of Carnap's notion of meaning postulates on this point). We see no difficulty in using postulates of the latter sort (ie. those which tend to be of a real

variables may stand for lexical categories such as *noun* or *verb*, or phrasal categories, like NP, VP, or S. Examples of several possible meaning postulates are given in 1.[12]

```
1a. If np1 knows that s1, then s1.

 b. If np1 gives np2 to np3, then np3 receives np2 from np1.

 c. If np1 causes np2 to vpi, then np2 vpi.

    Conditions: ¬negnp(np1) & ¬negnp(np2)

 d. If np1 v2s np2, then np2 is v2ed by np1.
```

Lexical variables have been added to the lexicon of the grammar, so that the parser treats them as lexical items with specific slot frames. As the transitive verb variable v2 in 1d illustrates, lexical variables also receive morphological inflection when required by the agreement rules of the grammar. Phrasal variables are defined separately, and recognized as non-lexical constituents.

Category variables are usually universally quantified (within the relevant category). In a number of cases, however, it is desirable to restrict the permissible range. One such case involves restricting the range of an NP category variable, say, to a certain semantic type like animal. Another case is in dealing with negative natural language quantifiers like *no one* and *nothing*. While a verb like *give* and a meaning postulate like 1b support inferences regardless of whether the NPs are positive or negative, a verb like cause in 1c only gives valid inferences for *positive* NPs; consequently, the permissible range has to be limited to non-negative NPs only.

Constraints on the permissible range are specified as separate conditions on the postulate (at the present state of development, these are expressed directly as Prolog goals). The desired restriction for the NPs in 1c is expressed as ¬negnp(np1) & ¬negnp(np2). Given this condition, **LEXSEM** derives *Mary sings* from *The musicians caused Mary to sing*, but not from *No musician caused Mary to sing*. On the other hand, since no restrictions are given on the range of the variables in 1b, **LEXSEM** correctly derives both *Mary received the book from John* from *John gave the book to Mary* and *Mary received nothing from John* from *John gave nothing to Mary*.

LEXSEM extracts the antecedent phrase and the consequent phrase from the parse tree of a postulate. The postulate is added to the Prolog workspace as a term of the form

```
implies(A,C,Cond).
```

where A is the parse of the antecedent phrase, C is the parse of the consequent phrase, and Cond the constraints supplied by the user.

---

world knowledge variety) in inferences required for lexical disambiguation. We do not claim to have an effective set of criteria for distinguishing between lexical semantic postulates and general empirical assumptions, nor do we think that this distinction need be absolute. We assume that the tasks of lexical disambiguation will determine which postulates must be adopted for inference.

12 vpi is a VP category variable which is initially instantiated by an infinitival VP, and subsequent occurrences of this variable are instantiated by appropriately tensed versions of this VP.

4.2. Forward inference chains

The forward inference algorithm generates forward inference chains by matching the parse structures of input sentences with those of the antecedents of meaning postulates, and instantiating the consequents of these postulates through the substitution of expressions in the input sentence for variables in corresponding syntactic positions in the consequent. We say that a *phrase* is the Prolog term produced by a Slot Grammar parser as the syntactic representation of an expression. A *phrase* P is an *instantiation* of a *phrase* P' iff P and P' are identical in syntactic structure and terminal expressions, except that where P' may contain a category variable in a syntactic (head, argument, or adjunct) position, P may contain a non-variable expression of the same type in the corresponding position.[13] *Instantiation* is, in fact, a relation of unification between phrases.

If the main verb of the input sentence is negated, the unnegated variant of the sentence is matched with the consequent of a meaning postulate, and the negated instantiation of the antecedent is derived. The algorithm applies recursively to inferred sentences to produce forward inference chains. Two examples of chains generated by successive instantiations of the antecedents of postulates are given in 2 and 3, respectively, where the system has been loaded with the meaning postulates listed in 1a-c.[14]

```
2. The opposition knows that the mayor caused the city to give financial
   aid to the poorest part of the population.

   Implied Statements:

   The mayor caused the city to give financial aid to the poorest part
   of the population.

   The city gave financial aid to the poorest part of the population.

   The poorest part of the population received financial aid from the city.

3. The opposition knows that no one will cause the city to give financial
   aid to the poorest part of the population.

   Implied Statements:

   No one will cause the city to give financial aid to the poorest
   part of the population.
```

[13] In fact, we also permit the corresponding head expressions of P and P' to differ in tense and agreement features to obtain a slightly more liberal definition of *instantiation* in **LEXSEM**.

[14] 2, and the examples of inference chains following it, are taken directly from output files generated by **LEXSEM**. The output is abbreviated by omitting the parse trees; only the *strings* generated from the trees are shown.

The affirmation of the antecedent and negation of the consequent components of the forward inference algorithm are integrated. Thus, the meaning postulates and input sentences in 4 and 5, respectively, generate the implied sentences which appear in each example.

```
4a. If np1 v2s np2, then np1 dislikes np2.

    Conditions: member(v2,hate.detest.despise.nil)

 b. If np1 appreciates np2, then np1 does not dislike np2.

 c. Mary appreciates her work.

    Implied Statements:

    Mary does not dislike her work.

    Mary does not despise her work.

    Mary does not detest her work.

    Mary does not hate her work.

5a. If np1 understands np2, then np1 comprehends np2.

 b. If np1 does not understand np2, then np2 is opaque to np1.

 c. If np1 is opaque to np2, then np2 does not yet grasp np1.

 d. Several students in the class do not comprehend the theory of
    quantum mechanics.

    Implied Statements:

    Several students in the class do not understand the theory of
    quantum mechanics.

    The theory of quantum mechanics is opaque to several students in
    the class.

    Several students in the class do not yet grasp the theory of
    quantum mechanics.
```

The algorithm filters out redundant inferences of sentences which have already been inferred from an input sentence in a forward inference chain. This permits **LEXSEM** to operate with circular meaning postulate sets without creating an infinite regress. The chain in 6c is derived from the circular meaning postulate set in 6a-b for *love* and *cherish*. The algorithm halts after inferring the input sentence, and so a regress is avoided.

```
6a. If np1 loves np2, then np1 cherishes np2.

 b. If np1 cherishes np2, then np1 loves np2.

 c. Every man loves his wife.
```

```
Implied Statements:

Every man cherishes his wife.

Every man loves his wife.
```

4.3. LEXSEM as an inference-based query system

In keeping with the general philosophy of using natural language and parses thereof whenever possible, queries are given as natural language questions and parsed by the Slot Grammar parser. In query mode, a backwards inference algorithm with negation-by-failure is applied. Consequently, **LEXSEM** can also be viewed as a natural language data base query system for data bases expressed in natural language, or, in other words, a text understanding system. So far, however, **LEXSEM** has only been applied as a lexical disambiguation system.[15]

The backwards inference is clearly more efficient than the forward inference because it is only necessary to derive *one* solution. For this reason, we formulate lexical disambiguation problems as queries which **LEXSEM** can solve by constructing backwards inference chains.

An ambiguous lexical item may appear in a sentence where it stands in the kind of syntactic and semantic relations to other constituents which will support particular inferences only on one of its readings. The system can disambiguate the term by testing these inferences through querying. When the meaning postulates in 7 are loaded in the work space, **LEXSEM** correctly replies "yes" to the question *Is the pen an enclosure?* when the database consists of the sentence in 8, and "no" to this question when the data base is the sentence in 9. Notice that the meaning postulate in 7b is crucially constrained by the condition `animal(np2)`, which requires that the object which `np1` contains has the feature `animal` associated with the entry for its head noun. In the following examples, we assume that `goat` has the feature `animal`, whereas `ink` does not.

```
7a. If np1 puts np2 in np3, then np3 contains np2.

 b. If np1 contains np2, then np1 is an enclosure.

    Condition: animal(np2)

8.  The farmer put his goat in the pen.

    Is the pen an enclosure?

    The answer is yes.
```

15 We will restrict ourselves to yes/no questions here, although we have, in fact, extended the system to wh-questions.

9. The student put some ink in the pen.

   Is the pen an enclosure?

   The answer is no.

Similarly, when the postulate in 10 is loaded, the sentences in 11 and 12 yield affirmative and negative answers, respectively.

10. If np1 runs np2, then np1 controls np2.

    Conditions: ¬competition(np2).

11. The Chancellor ran the meeting of the Board of Governors.

    Did the Chancellor control the meeting of the Board of Governors?

    The answer is yes.

12. Mary ran the marathon.

    Did Mary control the marathon?

    The answer is no.

Lexical disambiguation through inference-based querying permits a dynamic approach to resolving sense ambiguity. Rather than representing the distinct senses of lexical items as fixed entities in a lexicon, it is possible to determine whether a term has a given sense in the context of a sentence in which it occurs by testing the sentence for an implication that depends on the presence of that sense.

We have implemented a preliminary hookup of **LEXSEM** to the **LMT** English-Danish translation system. The query *Is the pen an enclosure?* is used as a test in the lexical transfer lexicon for determining whether the noun *pen* goes to *fold* (the *enclosure* sense) or to *fyldepen* (the *writing instrument* sense) in Danish. Clearly, the test can be refined by altering the set of meaning postulates in 7 without changing the query which activates the application of the backward inference algorithm to the input source sentence.

In order for the system to be able to chain backwards, it is necessary to pass on the *declarative variant* of a yes/no question to the system. This declarative variant P' of a yes/no question *phrase* P is obtained by removing the fronted auxiliary from P, or, in the case of a fronted main verb *be*, placing *be* in post subject position. The backwards inference algorithm applies to the declarative variant of the query *phrase* by chaining backwards until either an assertion is found that satisfies it, in which case the reply "yes" is generated, *or* the search space is exhausted, in which case the reply "no" is generated.

5. Transfer in LMT

The transfer component of LMT takes a Slot Grammar analysis tree for the source sentence and produces a surface structure tree in the target language for the translation of the sentence. The target tree is labeled by uninflected forms of the target words, plus appropriate target feature structures. A further step, morphological generation, produces the desired inflected forms of the target words.

The source analysis tree is available in both the nested-term (`phrase`) form and the clausal form, described above in Section 2. The target tree is represented essentially in node-list form, because this is convenient for transformations involved in transfer.

Transfer has two main steps: *Compositional Transfer*, and *Restructuring Transfer*. In the first step, a target tree (in node-list form) is produced, having (most of) the appropriate target words and target features, but having a tree structure essentially isomorphic to that of the source tree. The main job of the restructuring step is to put this tree into proper form for the target language, and this is done by applying a battery of transformations.

An example for English-German is shown in Fig. 5. Note that the compositional transfer tree has the same shape as the source tree. For restructuring, a transformation (`verbfinal`) moves the verb *gekauft (bought)* to the end of the dependent clause.

These basic steps for transfer have been described in previous papers (McCord 1986, 1989a,c,d); but there have been recent improvements that have not been published, and we will concentrate on these here (with a largely self-contained description). Of special interest is the new restructuring algorithm and the use of the node-list representation as the data structure that transformations work on. In previous versions of LMT, target trees were represented by nested-term data structures.[16]

5.1. The target node-list representation

The target node-list representations are slightly different from those described in Section 2.2 for source trees, because of the requirements of the transformations. Specifically, a target tree is a list of terms of the form

```
tnode(ID,Head,Features,Slot,Mods)
```

where the components are as follows:

- `ID` is an integer specifying the *ID* of the node. Distinct `tnodes` must have distinct IDs. Usually, the node will correspond to a source node (of which it is the translation) and `ID` will also be the ID of this source node – although transformations may add or delete some nodes.

[16] We would like to thank Danit Segev, IBM Haifa Science Center, for useful suggestions she made relating to the older restructuring system in LMT. Some of the functional requirements she brought up are addressed in the new system described here.

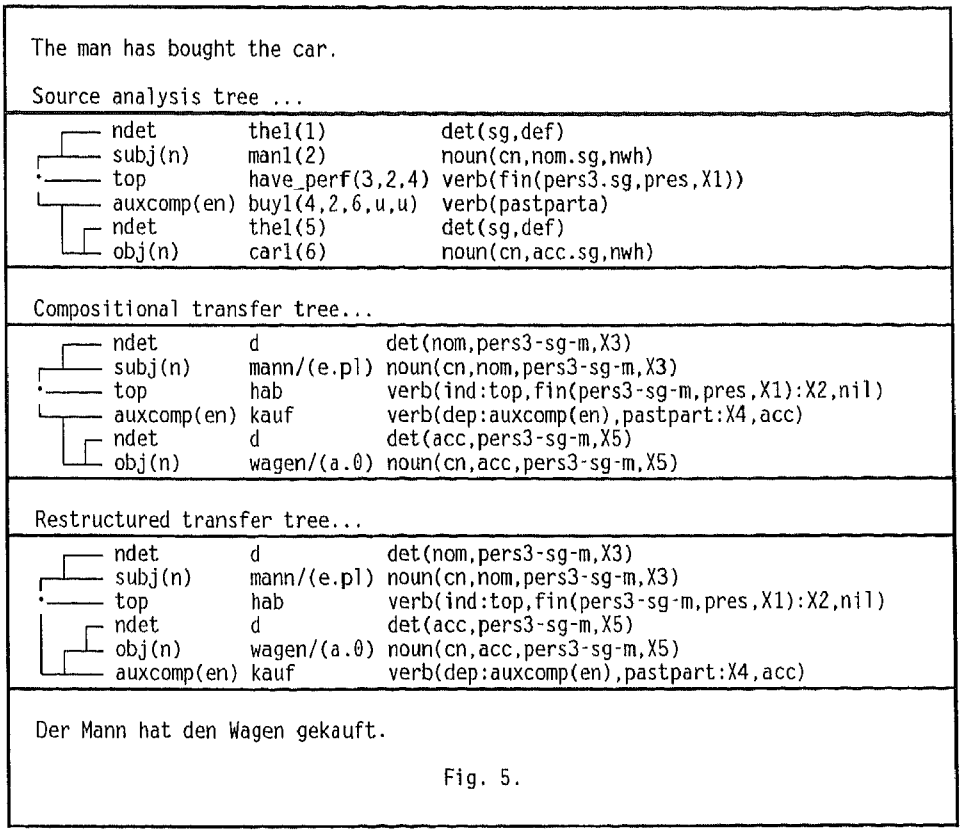


Fig. 5.

- Head is the uninflected form of a target head word of the node (possibly with some markings that will help inflection).
- Slot is the slot filled by the node. This is actually computable from the last (Mods) argument of the mother node, but is stored separately with the node for convenience of the transformations.
- Mods is a list of the form

  $LM_1. \ldots .LM_m.head.RM_1. \ldots . RM_n.nil$

  showing the left modifiers (daughters) LM_i and and the right modifiers RM_j of the node. Each modifier is either (a) of the form Slot:M showing the slot filled by the modifier and its tnode ID M, or (b) some other term (usually an atom), which will be taken literally as a leaf node. Modifiers of type (b) can be added by transformations, and are allowed for convenience.

As an example, the target node-list tree produced by Compositional Transfer for the example sentence in Fig. 5 is as follows:

```
tnode(3, hab, verb(ind:top,fin(pers3-sg-m,pres,*):*,nil),
      top, (subj(n):2).head.(auxcomp(en):4).nil ).
tnode(2, mann/(e.pl), noun(cn,nom,pers3-sg-m,*),
      subj(n), (ndet:1).head.nil ).
tnode(1, d, det(nom,pers3-sg-m,*),
      ndet, head.nil ).
tnode(4, kauf, verb(dep:auxcomp(en),pastpart:*,acc),
      auxcomp(en), head.(obj(n):6).nil ).
tnode(6, wagen/(a.0), noun(cn,acc,pers3-sg-m,*),
      obj(n), (ndet:5).head.nil ).
tnode(5, d, det(acc,pers3-sg-m,*),
      ndet, head.nil ).
nil ).
```

(The display of the second tree in Fig. 5 is created from this structure.)

5.2. Compositional Transfer

The main procedure, tran, in LMT that performs Compositional Transfer works recursively on source phrase structures to produce corresponding target node-list structures. The basic idea of the tran algorithm is simple:

For any source phrase, tran produces a corresponding tnode by doing the following:

1. The target Features of the corresponding tnode are found by a procedure (tranfeas) in the shell that looks at the source features plus information that comes ultimately from the transfer lexicon and is passed around the tree through unification.
2. The target head word Head of the tnode is found by a procedure tranword, discussed below.
3. Using the modifier lists (left and right) of the source phrase, tran calls a procedure tranlist to produce the modifier list of the current tnode, and tranlist calls tran recursively for each source modifier.

We will not give more details of tran here – they are much the same as in previously published accounts (see McCord 1989a,c) – except for commenting briefly on one new ingredient: the use of the clausal representation of the source tree for tests in lexical transfer.

The top-level procedure tranword for lexical transfer deals with many special cases, but the main one involves a call to a procedure twframe whose clauses are compiled from information in the transfer lexicon. A call to twframe is of the form (we give a slightly simplified exposition):

```
twframe(POS,Sense,Args,Head)
```

where POS is the part of speech of the current node, Sense is the source sense, and Head is the output target head word. The argument Args is a list of pairs Case:ID associated with the complement slot frame of Sense. (Actually, the first member of Args is associated with the current node itself, and the remaining ones are associated with its complements.) The Case component for a given complement is an output of twframe and will be the target case on the corresponding target complement. "Case" can include information about prepositional complementation, and even information about the forms of other types of complements than NPs.

The main thing of interest here is that the IDs of complements and of the current node itself are available to twframe. The body of a twframe clause can then make calls to the predicates of the clausal representation of the source tree, and explore any part of the tree, making any kinds of tests desirable on its nodes – including semantic type tests or tests on syntactic or morphological form.

The **LMT** lexical formalism includes compact, convenient ways of specifying such tests on the source tree in lexical transfer elements. The lexical compiler translates these test expressions (in part) into calls within the bodies of twframe clauses referring to the source clausal tree representation predicates.

5.3. Restructuring Transfer

Now let us describe the Restructuring Transfer step, in its new form. Restructuring is handled by a procedure

```
restructure(NL,NL1)
```

which takes the output tree NL of Compositional Transfer (in node-list form) and produces the desired target surface tree NL1 (in node-list form).

What restructure has to work with is a list of transformations, each a procedure

```
Trans(NL,NL1)
```

that takes a node-list tree NL and makes another one NL1.[17] The names (Trans) of these transformations are specified in an ordered list given as the argument of a unit clause:

```
transforms(Tranlist).
```

The algorithm for restructure is very simple:

17 We give a slightly simplified exposition. In the actual system, the transformation predicate gets two more arguments that prevent infinite loops. One can define the *node in focus* of a transformation application. This is roughly the main node that undergoes a change; technically, it is the first node mentioned in a node-accessing goal (described below) appearing in the body of the transformation rule. The extra arguments do bookkeeping that prevents the transformation from applying more than once to a given node in focus.

> *Starting with the input tree to* `restructure`, *run through the list of transformations, and apply each one as many times as it succeeds, to the whole (current) tree. The final tree is the output of* `restructure`.

This algorithm is simpler than previous ones in **LMT**, which (among other things) involved recursive visiting of the nodes of the tree, with a separate run-through of the whole transformation list on each level.

There is a special formalism for writing transformation rules which makes them more compact, and a rule compiler that converts them into Prolog clauses for the transformations as Prolog predicates as indicated above. In this formalism, a transformation is of the form

```
Trans -- Body.
```

where `Trans` is the transformation name (an atom), and the `Body` has the same form as the body of a Prolog clause, except that its goals may be *special goals* which are translated in special ways by the rule compiler (other goals being translated as themselves).

All special goals take a `tnode` ID as their first argument. Special goals are of two types:

1. *Node-accessing* goals. Their additional arguments give conditions on the node specified in the first argument, and they will *find* a node with the given conditions, if their first argument is given unbound, or otherwise simply *test* the node for those conditions. The predicate names for all these goals begin with `t`.
2. *Node-changing* goals. For these, the first argument should be bound (to a node ID), and then the node will be *changed* to have new labels as specified in the additional arguments of the special goal. The predicate names for all these goals begin with `c`.

For an example, let us look at a (fictitious) transformation `addzu` which is supposed to add `zu` as an initial modifier to every verbal node whose feature structure is of the form `verb(*,*,inf)`:

```
addzu --
  tfm(V, verb(*,*,inf), Mods) &
  cm(V, zu.Mods).
```

Here `tfm(ID,Feas,Mods)` is a node-accessing goal which finds (or checks on) a node `ID` with the indicated features and modifier list. The letter `f` in the name `tfm` suggests "features" and the `m` suggests "modifiers". The goal `cm(ID,Mods)` is a node-changing goal which changes node `ID` to have the indicated modifier list. Note that when `tfm` is called, the `ID` is unbound; the call *finds* nodes.

Suppose we want to revise `addzu` so that it adds the `zu` right before the head verb. This can be done as follows:

```
addzu --
  tfm(V, verb(*,*,inf), %LMods.head.RMods) &
  cm(V, %LMods.zu.head.RMods).
```

This illustrates the use of *sublist variables* in arguments to special goals. A sublist variable is of the form `%V`. When used in a node-accessing goal, `%V` will match any sublist (non-deterministically) of a list structure in which it appears, binding `V` to the sublist. When used in a node-changing goal, `%V` (with `V` bound) will insert the sublist into the list structure in which it appears.

Sublist variables were used in earlier versions of **LMT** transformation rules (McCord 1989a,c). The rule compiler basically translates them into calls to `append`.

The rule compiler converts a transformation rule

```
Trans -- Body.
```

into a Prolog clause

```
Trans(INL,ONL) <- Body1.
```

The transformation name is converted into a 2-place predicate[18] that takes an input node-list `INL` and creates an output node-list `ONL`. In running through the `Body` to produce `Body1`, the rule compiler keeps track of the *current node-list*, which is implicit in special goals. Of course the current node-list starts out as `INL` and ends up as `ONL`. Every node-accessing goal refers to the current node-list, and gets this as an extra (final) argument in the compiled form (in `Body1`). Every node-changing goal updates the current node-list, and gets two extra (final) arguments in compiled form to do this. Adding these extra arguments and dealing with sublist variables are the two main jobs of the rule compiler.

There are several node-accessing goals known to the rule compiler, most of them dealing simply with components of a `tnode`:

```
tnode(ID,Head,Feas,Slot,Mods).
```

The most complete node-accessing goal is just

```
tno(ID,Head,Feas,Slot,Mods),
```

which accesses all the components. Of course in compiled form, `tno` takes a node-list as a final argument, and it has the obvious, simple definition:

```
tno(ID,Head,Feas,Slot,Mods,NL) <-
  member(tnode(ID,Head,Feas,Slot,Mods),NL).
```

[18] Recall, from a footnote above, that it is actually a 4-place predicate where the extra arguments do bookkeeping to prevent repeated applications of the transformation to a given node in focus.

All node-accessing could be done with tno, but often one is concerned only with certain components, and there are node-accessing goals for looking at any individual components or combinations of them. Their names follow the simple mnemonic of using letters to indicate which components are picked out:

```
h:  Head
f:  Features
s:  Slot
m:  Modifiers
```

Thus `tfm(ID,Feas,Mods)` would access a node `ID` with the indicated features and modifiers.

The most complete node-changing goal is

```
cno(ID,Head,Feas,Slot,Mods).
```

The definition of its compiled form is

```
cno(ID,Head,Feas,Slot,Mods,NL,NL1) <-
  conc(LNL,tnode(ID,*,*,*,*).RNL,NL) &
  conc(LNL,tnode(ID,Head,Feas,Slot,Mods).RNL,NL1).
```

Here `conc(L,M,N)` means that the concatenation of list `L` with list `M` is list `N`. The first call to it takes apart the list `NL`; the second call puts together the list `NL1`.

As with node-accessing goals, there are node-changing goals to work with any individual components or combinations, and the naming conventions are similar (starting with c instead of t).

Another example of a transformation is one that moves the verb of a dependent clause to the end (a simplified version of one for the English-German **LMT**):

```
verbfinal --
  tfm(V, verb(dep:*,*,*), %LM.head.RM) &
  cm(V, %LM.%RM.head.nil).
```

There are a few other useful special goals that do more than access nodes or change their components. One node-changing goal of special interest is

```
cdelete(H)
```

which "deletes" node `H` in a certain sense. The `tnode` with the given ID (`H`) is deleted from the current node-list, and its head, features, and slot disappear, but its modifiers are not lost. The modifiers are simply moved up to become modifiers (daughters) of the mother node of `H`, inserted in just the position that node `H` had.

An example of the use of `cdelete` is in the English-German transformation `auxdo` that deals with English questions with an auxiliary form of *do*. An example is shown in Fig. 6. The definition of the transformation is:

```
auxdo --
  thm(Do, do1, %M.(auxcomp(*):V).N) &
  th(V, Verb) & ch(Do, Verb) & cdelete(V).
```

```
Do you see the car?

Source analysis tree ...

·——— top           do1(1,2,3)  verb(fin(X1.pl,pres,ind:q:X2))
└——— subj(n)       you1(2)     noun(pron(def),nom.pl,nwh)
└——— auxcomp(binf) see2(3,2,5) verb(inf(bare))
   ┌ ndet          the1(4)     det(sg,def)
   └ obj(n)        car1(5)     noun(cn,acc.sg,nwh)

Compositional transfer tree...

·——— top           do1         verb(ind:top,fin(pers2f-pl-X1,pres,ind:q:X2)..)
└——— subj(n)       Sie         noun(pron(def),nom,pers2f-pl-X1,X4)
└——— auxcomp(binf) seh         verb(dep:auxcomp(binf),inf:X5,auxsubj)
   ┌ ndet          d           det(acc,pers3-sg-m,X6)
   └ obj(n)        wagen/(a.0) noun(cn,acc,pers3-sg-m,X6)

Restructured transfer tree...

·—— top     seh         verb(ind:top,fin(pers2f-pl-X1,pres,ind:q:X2):X3,nil)
└—— subj(n) Sie         noun(pron(def),nom,pers2f-pl-X1,X4)
┌ ndet     d           det(acc,pers3-sg-m,X5)
└ obj(n)   wagen/(a.0) noun(cn,acc,pers3-sg-m,X5)

Sehen Sie den Wagen?
```

Fig. 6.

The auxiliary sense of *do* is `do1` and this translates into itself. So the first goal (`thm`) finds such a node, with ID `Do`, and locates the auxiliary complement `V` (the `seh` node, `V=3`, in Fig. 6). The `th` goal finds the head word `Verb` (`seh`) of `V`, and the `ch` goal *changes* the head of the `Do` node to `Verb`, keeping however the inflectional features of the `Do` node, appropriately. Then the `V` node is `cdeleted` (and its modifiers move up).

Another useful "built-in" node-changing goal is one that *adds* a new node. It is called in the form

```
cadd(H,Head,Features,Slot,Modifiers)
```

where argument `H` is *unbound* and the other arguments specify the components of the new node. The system generates a new unique node integer, binds `H` to it, and adds the new node to the current node-list.

Two advantages of the node-list representation for transfer trees and the new restructuring algorithm, which we do not have space to illustrate here, are the following:

- The representation allows transformations to explore any part of the tree, from the point of view of the node in focus.
- The use of node IDs that originate as IDs of source nodes allows transformations to refer easily to the source clausal representation and thereby make tests on the

source tree – for example tests on semantic types of source words that correspond to target words in focus.

6. A Parser of Constructions

Constructions (Zadrozny and Manaster-Ramer 199?) provide a means for semantic interpretation in a discourse context, and for producing analyses for NL strings where a normal parser fails. Specifically, a parser of constructions can be used for analyzing idiomatic and "non-grammatical" expressions.[19]

6.1. Why constructions?

Limitations of many existing syntactic and semantic analyzers are easily seen when one tries to apply them to constructions such as:

1. *The more carefully you work, the easier it will get.*
2. *Why not fix it yourself?*
3. *He's not half the doctor you are.*
4. *Rain or no rain, I'll go for a walk.*
5. *Much as I like Ronnie, I don't approve of anything he does.*
6. *It's time you brushed your teeth.*
7. *One more and I'll leave.*
8. *Good idea that.*
9. *Him be a doctor?*[20]

The reason for this failure can be attributed to the fact that expressions like these "exhibit properties that are not fully predictable from independently known properties of its lexical make-up and its grammatical structure" – (Fillmore et al. 1988, p.511).

---

[19] In the current paper we view constructions as an augmentation of Slot Grammar analysis, but actually in (Zadrozny and Manaster-Ramer 199?) they are viewed as a general framework for natural language analysis. While the **ESG** parser is a "robust, production quality system", the parser of constructions described in this section is a "small research system." We are using here the classification of E. Hinkelman, according to which, based on the number of examples the system has been tested on, we have:

- 1-10 - demonstration system
- 10-100 - small research system
- 100-1000 - larger research system
- 1000-10000 - robust or production quality system

Thus, while everything described in this section has been implemented, the reader will notice place holders for certain, yet unimplemented functions.

[20] Also written with a comma: *Him, be a doctor?*

Moreover, constructions such as those listed above[21] are productive; that is, many different words may appear in these patterns. Thus, computationally, they cannot be treated as lexical entries, although frozen idioms like *kick the bucket* can. All of them have a certain form, but in many cases they couldn't appear in regular printed dictionaries, because they do not involve specific words: (9) and *Them go to France?* do not share any words.

The next problem has to do with the fact that constructions such as these cannot be directly accommodated by standard grammars without introducing serious overgeneralizations. The structure of (8) is "S --> NP NP"; and in this case, even if we assume it to be a kind of ellipsis, we should notice that such a construction is permitted only with abstract entities; that is, the constraints on its applicability are clearly semantic.[22]

6.2. The function of semantics in a grammar

Although we have begun this section with a list of mostly "strange" constructions (2)-(9), with a closer look, certain apparently straightforward constructions exhibit a much more complex structure than many grammarians admit. For example, notice that while (1) is not as common as S --> NP VP, there is definitely nothing abnormal about it. However, it presents some difficulties to a standard phrase structure based parser: (a) *The ... work* and *the ... get* do not look like of the standard phrases; still, their combination is a sentence. Similarly, there is no natural account of this construction in terms of normal slot filling rules.[23] (b) The structure of *The ... work* is roughly *the ADV(comparative) S*. According to some analyses, *the* has a function of an adverb here, not a determiner. This, however, poses a problem: since *the* functions as an adverb only in a few, very specific, constructions, such as (1), we would not like a parser (especially, if it is used as a grammar checker) to accept sentences like *John likes to the work* or *John came the sooner than others*. We solve it by describing this construction structurally, which allows us to ignore the question whether *the* is a determiner or an adverb here, and use any analysis of it provided by the Slot Grammar.

We use (a) in support of our thesis that the syntactic patterns of language are richer than those permitted by standard grammars[24]; and (b) to claim that in order to describe language we have to specifically describe many constructions, in very much the same way as we describe grammatical functions of words, which cannot be done without referring to semantics, cf. e.g. (McCawley 1988) or (McCawley 1982).

---

21 also called "open idioms" by Fillmore et al.

22 cf. also *Smart guy your friend Bill, His name John Books, This one to Pennsylvania, that one to New York, Dr. Smith to the Operating Room, Me worry?*.

23 But an extension of **ESG** by a metarule could handle this construction.

24 In (Zadrozny and Manaster-Ramer 199?), we give arguments why standard approaches are too weak.

As another, even simpler, and well-known example, consider the role of the determiner *a* in NPs. NP isn't really (Det) (ADJ)* N [25], for, normally, we would like to permit *it's time*, but not *it's book*; or *give me freedom/love/time/attention!*, and not *give me book/bicycle/chair/eagle!*; however in unusual circumstances, these commands would sound just fine, e.g., if a monster were demanding his favorite food. To see why it matters, consider translation into English from languages that normally do not use determiners with nouns, e.g. Russian or Polish.

Information about the roles of the speaker and the hearer is conveyed by discourse constructions. Those most commonly discussed express intentions, presuppositions, conversational implicature, etc., cf. (Levinson 1985); they obviously are important and have to be dealt with, e.g. along the lines of (Cohen and Levesque 1990).

But even simple, ordinary examples are worth looking at:

(10) *Have you washed the dishes?*

(11) *No, but I'll do it right away.*

In the construction used in the answer, which we refer to as *No, but S*, the sentence *S* following *No, but* must be in a relation of coherence with the preceding discourse (Zadrozny and Jensen 1991). In other words, its application is dependent on the semantic and pragmatic context.

Notice also that as one formal language can be described by many formal grammars, so can a natural language be described by many collections of constructions. For instance, the construction *No, but S* could be eliminated if our set of constructions would already contain the discourse constructions *No* (as an answer to the question), *But S*, as in

(12) *It's late.*

(13) *But I'm having such a great time,*

and the construction of "linking", that is, of putting sentences together. In such a case, we could assume that *No, but S* is an orthographic variant of *No. But S.*

6.3. How do we represent constructions?

Before we address the issue of representing constructions and parsing, we should note two points: (a) Words, phrases, sentences, and fragments of discourse can be analyzed as constructions; and we believe that no system concerned with the meanings of real language can avoid the kind of microanalysis that we present in this section. (b) Standard grammars take care of the most common constructions, in the sense that they describe their structural properties; therefore it is possible to look at a grammar of constructions as an extension of a standard one, which, essentially, is the relationship between the small grammar of constructions that we describe here and **ESG**.

[25] a noun preceded by 0 or more adjectives, with or without a determiner at the beginning

Formally, a construction is a triple (*P,V,M*) consisting of a set of semantic and pragmatic *preconditions P*, a *vehicle V* describing the basic pattern of the construction, and a *message*, *M*, containing the meaning of the construction (which may include illocutionary force as well as propositional content). The vehicle *V* is a pattern that is matched against an input string, where the pattern elements can test for syntactic as well as semantic features of substrings of the input. The preconditions *P* and the message *M* can be likened to preconditions and postconditions in the semantics of programming languages.

For example, the matrix of the construction (9) *Him be a doctor?* looks as follows:[26]

$$\left[\begin{array}{l} N\text{: } np(acc).vp(inf,noto) \\ \left[\begin{array}{l} P\text{: [} d \text{ } implies \text{ } prop \text{]} \\ V\text{:} \left[\begin{array}{l} struc((np(X).vp(Y).?.nil)) \\ cons\_name(Y,vp(inf,noto)) \\ cons\_name(X,pronoun) \\ case(X,acc) \end{array}\right] \\ M\text{:} \left[\begin{array}{l} subject(Y) = X \\ [\ illoc\text{:}[excl,\ surpr(speaker, prop)]] \\ [\ prop\text{: } message(Y)] \end{array}\right] \end{array}\right] \end{array}\right]$$

There are some interesting facts about the preconditions of this construction. They can be read in two ways. In the context of generation, the precondition says that the propositional content of this construction must be implied by the (immediately) preceding fragment of discourse *d*. Notice that a constraint on the propositional content of this construction is described in the construction itself. From the point of view of parsing, the formula expresses a constraint; the discourse would be ill-formed if there was no fragment implying the content. But the content is not known before the message is computed. The implementation uses the "wait: ..." predicate which postpones evaluation of this condition until the value of "prop" (in the current implementation, for the above sentence it would be *be(he, doctor)*) is known, that is, after *X* is substituted as the subject in the message of *Y*.

The message freely mixes syntax, semantics and pragmatics; and there is no clear way of separating the three. The structure of the construction is represented by the data structure np(X).vp(Y)."?", and some conditions are placed on the two parts: the NP should be a pronoun in the accusative, and the VP should be infinitival and without *to*.

26 N stands for the name of the construction.

Notice that this construction assumes that its parts have already been identified as other constructions – that is what *cons_name*(×, ×) expresses.

The message in this example contains the illocutionary statement that the speaker is surprised at prop (*be(he,doctor)*). The semantics of surprise, at least its rudimentary theory, could probably be created by following (Sadock 1990), (Cohen and Levesque 1990), and (Vanderveken 1990); and a usable theory of illocutionary acts could probably be implemented using the **LEXSEM** system (Section 4 above, Bernth and Lappin 1991), by means of a set of meaning postulates.

Point (b) becomes apparent when we consider for instance standard N and NP constructions. Thus, the construction describing common nouns in the singular is represented as a Prolog term in the following way:

```
c(noun([common,sing]),
  veh(U : phrase(V1,1,s(Noun,1),noun(cn,[V3|sg],V4),
         [], [],mods([], [])))),
  prec([english_word(Noun)]),
  m([pred(Noun,1), th(Noun,V5)])]).
```

The precondition *prec*(...) says that the language we consider is English; the structure *veh*(...) is the **ESG** structure that is assigned to common nouns in singular (notice the markers *cn* and *sg* there); and the message *m*(...) creates a one argument predicate of the *Noun* and attaches to it information *th*(...) that might be relevant for larger constructions, such as NPs. Thus, *th*(...) will contain information whether the noun is count or mass, human, animal or inanimate, etc.[27]

Standard semantics of NP --> a N is a part of the following construction

```
c(np([[a|noun([common])]]),
veh(struc([det(V1),noun_string(String)]) &
  conds([det(V1) & V1 = [a],
  noun(Noun) & cons_name(Noun,noun([common, sing]))])),
prec([]),
m([ member(Noun, String) & pred(Noun, 1),
    sat(pred(Noun,1),V)]),[]) .
```

There are no preconditions *prec*(...) for this construction, but notice that, because of the reference to the construction *noun*([*common, sing*]), the precondition about dealing with English is implicit. The structural condition describing the construction says that the string "a" is followed by a string that can be identified as *noun*([*common, sing*]). Finally, the meaning of this construction is given as a postulate that an object *V* satisfying the one argument predicate *Noun* exists.

[27] This information is available from an on-line version of (Longman 1978) and as shown in (Braden-Harder 1991) can be used, e.g., for sense disambiguation. The connection from the parser of construction to Longman's has not been implemented yet.

The less common construction such as *The X-er, the Y-er* (sentence (1)) or the expression of surprise, as in (9) have similar representations. The message part of the former says that

```
m([english("proportional dependence between the two"
  : [S1,S2])])) .
```

which is left uninterpreted for the time being.[28]

The representations of constructions refer directly to ESG phrase analysis representations, as in the case of *noun([common, sing])*, or indirectly, e.g. in *np(acc).vp(inf,noto)* by referring recursively to other constructions. This allows the parser described in the next subsection to try to match directly possible structures of constructions with the results of analyses produced by ESG. In other words, ESG assigns structures of the form (cf. Section 2)

```
phrase(Span,Sense,Features,Frame,Ext,Mods),
```

to substrings of an input string, and using this information, the parser of constructions will assign possible constructions to the input string. We give details below.

6.4. Parsing with a set of constructions

The formalism we propose can be viewed as an extension of traditional grammars; e.g., from our perspective, predicate-argument structure, case-markings, etc., constitute part of meanings of sentences. By the same token, parsing with constructions is an extension of syntactic parsing, meaning that (1) the collections of features that are used to drive parsing is richer, because it contains terms with semantic and pragmatic interpretation, and (2) structures assigned to strings are more complex (for the same reason).

The goal of parsing is to assign a structure to a string. In our case, a string may consist of several sentences, and the structure is definitely not restricted to a syntactic parse tree, but consists of a set (or hierarchy) of constructions that can be assigned to it. Because of the ambiguities, we often have mutually exclusive interpretations, and hence more than one such set can be assigned.

We have implemented the parser as an postprocessor to the English Slot Grammar (ESG). The advantages of this approach are at least twofold: we can assign very fast basic syntactic structures to strings, and this allows us to concentrate on how constructions can improve parsing; secondly, we get much better coverage than if we tried to build a parser of constructions from scratch. This parser could also be implemented as a part of ESG, if it were extended to parse augmented phrase structure

[28] To interpret it, we would need, for example, a set of meaning postulates about two orderings. The entities mentioned in the two sentences would be mapped into the orderings, and the meaning postulates would say that when a value on the first ordering changes, it produces a corresponding change of the second measure.

grammar rules. Of course, the extension would have to allow testing of the semantic preconditions that appear in constructions, and the output would contain semantic information.

Given a string (representing a sentence, a fragment of a discourse or a paragraph), the parser assigns a construction to it. From this point of view, the situation is similar to normal parsing, and the possible implementations are also similar: top-down search for applicable constructions, or bottom-up combining of appearing constructions; we use the first approach in our prototype, but we are switching to the bottom up approach for the sake of efficiency.

Note that with regard to parsing, constructions behave like standard grammars; we assign only one construction (which does not preclude having ambiguous sentences/constructions, for ambiguity is a result of different assignments, each of which connects the string with one construction).

As an illustration consider parsing the dialog

Them go to Paris this year?
Why are you so surprised about it?

The important steps in parsing are as follows: The input string consists of both sentences. This string will be divided into parts and the parser will try to assign constructions to various parts. Thus *year* will be recognized as a noun, and *them*, *you* and *it* as pronouns. But what really matters in this case is that both the *np(acc)* and *vp(inf-noto)* constructions are assigned, respectively, to *them* and *go to Paris this year*; the infinitival clause has the meaning given by *Paris* occupying the second argument position of *go* (+ tense + all other goodies). As a result the "surprise" *np(acc).vp(inf, noto).?* construction (cf. sentence (9) at the beginning of this section) is assigned to the sentence, with the message providing the second argument of *go*, and thus making it into a proposition (something that can have a truth value). A similar process is applied to the rest of the input string, and the second sentence is parsed as a why-question.

By the same token, when the parser is applied to the dialog (12) and (13), (13) will be recognized as *No, but S* construction only when (12) is recognized as a question; that is the parser will check the precondition of the construction.

The output of the parser could be used by other modules. E.g., an anaphora resolution procedure could be applied to the semantic representation of the sentences discussed. There the need for semantics is also clear: Since *surprise* requires an event as an object, *Paris* cannot be a referent of *it*, *go(they, Paris)* can.

References

Alshawi, H. (1987) "Memory and Context for Language Interpretation," Cambridge University Press, Cambridge.

Bernth, A. (1989) "Discourse Understanding in Logic," *Proceedings of the North American Conference on Logic Programming*, MIT Press, Cambridge, Mass, pp. 755-771.

Bernth, A. (1990) "Anaphora in Referentially Opaque Contexts" in R. Studer (ed.), *Natural Language and Logic: International Scientific Symposium*, Lecture Notes in Computer Science, Springer Verlag, Berlin.

Bernth A. and S. Lappin (1991) "A Meaning Postulate Based Inference System for Natural Language," Technical Report # RC 16947, IBM T.J. Watson Research Center, Yorktown Heights, NY.

Bernth, A. and McCord, M. C. (1991) "**LMT** for Danish-English Machine Translation," In C.G. Brown and G. Koch (Eds), *Natural Language Understanding and Logic Programming, III*, pp. 179-194, North-Holland.

Braden-Harder, L. C. (1991) "Sense Disambiguation Using On-Line Dictionaries," M. Sc. Thesis, Comp. Sci. Dept, New York University, New York, NY.

Carnap, R. (1947) *Meaning and Necessity*, University of Chicago Press, Chicago.

Cohen, P. R. and Levesque H. J. (1990) "Persistence, Intention, and Committment," in Cohen, P. R., et al. (Eds.), *Intentions in Communication*, pp. 33-70, MIT Press, Cambridge, MA.

Cooper, R. (1983) *Quantification and Syntactic Theory*, Reidel, Dordrecht.

Dowty, D. (1989) "On the Semantic Content of Thematic Roles" in G. Chierchia, B.H. Partee, and R. Turner (eds.), *Properties, Types, and Meaning*, Kluwer, Dordrecht, pp. 69-129.

Fillmore, C. J., Kay, P. and Catherine O'Connor, M. C. (1988) "Regularity and idiomaticity in grammatical constructions," *Language*, 64, pp. 501-538.

Grimshaw, J. (1990) *Argument Structure*, MIT Press, Cambridge, MA.

Guenthner, F. and H. Lehmann (1983) "Rules for Pronominalization" in *Proceedings of the First Annual Meeting of the European Chapter of the ACL*, pp. 144-151.

Hobbs, J., M. Stickel, P. Martin, and D. Edwards (1988) "Interpretation as Abduction," *Proceedings of the 26th Annual Meeting of the Association of Computational Linguistics*, pp. 95-103.

Hudson, R. A. (1971) *English Complex Sentences*, North-Holland, Amsterdam.

Hudson, R. A. (1976) *Arguments for a Non-Transformational Grammar*, University of Chicago Press, Chicago.

Jackendoff, R. (1976) "Toward an Explanatory Semantic Representation," *Linguistic Inquiry* 7, pp. 89-150.

Jackendoff, R. (1983) *Semantics and Cognition*, MIT Press, Cambridge, MA.

Jackendoff, R. (1987) "The Status of Thematic Relations in Linguistic Theory," *Linguistic Inquiry* 18, pp. 369-411.

Jackendoff, R. (1990) *Semantic Structures*, MIT Press, Cambridge, MA.

Kamp, H. (1981) "A Theory of Truth and Semantic Representation" in J. Groenendijk, T. Janssen, and M. Stokhof (eds.), *Formal Methods in the Study of Language*, MC Tract 135 & 136, Amsterdam, pp. 276-322.

Keenan, E. and B. Comrie (1977) "Noun Phrase Accessibility and Universal Grammar," *Linguistic Inquiry* 8, pp. 62-100.

Lappin, S. (1985) "Pronominal Binding and Coreference," *Theoretical Linguistics* 12, pp. 241-263.

Lappin, S. (1991) "Concepts of Logical Form in Linguistics and Philosophy" in A. Kasher (ed.), *The Chomskyan Turn*, pp. 300-333, Blackwell, Oxford.

Lappin, S. and H. Leass (1992) "A Syntactically Based Algorithm for Pronominal Anaphora Resolution," IBM T.J. Watson Research Center, Yorktown Heights, NY and IBM Germany Scientific Center, Institute for Knowledge Based Systems, Heidelberg.

Lappin, S. and M. McCord (1990a) " A Syntactic Filter on Pronominal Anaphora in Slot Grammar" in *Proceedings of the 28th Annual Meeting of the Association for Computational Linguistics*, pp. 135-142.

Lappin, S. and M. McCord (1990b) "Anaphora Resolution in Slot Grammar," *Computational Linguistics* 16, pp. 197-212.

Leass, H. and U. Schwall (1991) *An Anaphora Resolution Procedure for Machine Translation*, IWBS Report 172, IBM Germany Scientific Center, Heidelberg.

Lehmann, H. (1988) "The LEX Project - Concepts and Results" in A. Blaser (ed.), *Natural Language at the Computer*, Springer-Verlag, Lecture Notes in Computer Science, Berlin, pp. 112-146.

Levinson, S. G. (1985) *Pragmatics*, Cambridge University Press, Cambridge, MA.

Longman (1978) *Longman Dictionary of Contemporary English*, Longman Group Ltd., London.

McCawley, J. D. (1988) *The Syntactic Phenomena of English*, University of Chicago Press, Chicago, IL.

McCawley, J. D. (1982) *Thirty Million Theories of Grammar*, University of Chicago Press, Chicago, IL.

McCord, M. C. (1980) "Slot Grammars," *Computational Linguistics*, vol. 6, pp. 31-43.

McCord, M. C. (1982) "Using Slots and Modifiers in Logic Grammars for Natural Language," *Artificial Intelligence*, vol. 18, pp. 327-367.

McCord, M. C. (1986) "Design of a Prolog-based Machine Translation System," *Proceedings of the Third International Logic Programming Conference*, pp. 350-374, Springer-Verlag, Berlin.

McCord, M. C. (1988) "A Multi-Target Machine Translation System," *Proceedings of the International Conference on Fifth Generation Computer Systems 1988*, pp. 1141-1149, Institute for New Generation Computer Technology, Tokyo, Japan.

McCord, M. C. (1989a) "Design of **LMT**: A Prolog-based Machine Translation System," *Computational Linguistics*, vol. 15, pp. 33-52.

McCord, M. C. (1989b) "A New Version of Slot Grammar," Technical Report # RC 14506, IBM T.J. Watson Research Center, Yorktown Heights, NY.

McCord, M. C. (1989c) "A New Version of the Machine Translation System **LMT**," *Literary and Linguistic Computing*, 4, pp. 218-229.

McCord, M. C. (1989d) "**LMT**," *Proceedings of MT Summit II*, pp. 94-99, Deutsche Gesellschaft für Dokumentation, Frankfurt.

McCord, M. C. (1990) "Slot Grammar: A System for Simpler Construction of Practical Natural Language Grammars," In R. Studer (ed.), *Natural Language and Logic: International Scientific Symposium*, Lecture Notes in Computer Science, Springer Verlag, Berlin, pp. 118-145.

McCord, M. C. (1991) "The Slot Grammar System," Technical Report # RC 17313, IBM T.J. Watson Research Center, Yorktown Heights, NY. To appear in J. Wedekind and C. Rohrer (Eds.), *Unification in Grammar*, MIT Press.

Montague, R. (1974) "The Proper Treatment of Quantification in Ordinary English" in R. Thomason (ed.), *Formal Philosophy*, Yale University Press, New Haven, Conn., pp. 247-270.

Neff, M. and McCord, M. C. (1990) "Acquiring Lexical Data From Machine-readable Dictionary Resources for Machine Translation," *Proceedings of the 3rd Int. Conf. on Theoretical and Methodological Issues in Machine Translation of Natural Languages*, pp. 87-92. Linguistics Research Center, Univ. of Texas, Austin.

Pinker, S. (1989) *Learnability and Cognition: The Acquisition of Argument Structure*, MIT Press, Cambridge, MA.

Pollard, C. and Sag, I. A. (1987) *Information-Based Syntax and Semantics*, CSLI Lecture Notes No. 13, Center for the Study of Language and Information, Stanford, CA.

Quine, W.V. (1966) "Carnap and Logical Truth" in W.V. Quine, *The Ways of Paradox*, Random House, New York, pp. 100-125.

Rappaport, M. and B. Levin (1988) "What to Do with Theta-Roles" in W. Wilkins (ed.), *Syntax and Semantics* 21, Academic Press, New York, pp. 7-36.

Rimon, M., McCord, M. C., Schwall, U., and Martínez, P. (1991) "Advances in Machine Translation Research in IBM," *Proceedings of MT Summit III*, pp. 11-18, Washington, D.C.

Sadock, J.M. (1990) "Comments on Vanderveken and on Cohen and Levesque," in Cohen, P. R., et al. (Eds.), *Intentions in Communication*, pp. 257-270 MIT Press, Cambridge, MA.

Schubert, L. and C.H. Hwang, (1989) "An Episodic Knowledge Representation for Narrative Texts," *Proceedings of The First International Conference on Principles of Knowledge Representation and Reasoning*, Toronto, Ontario, pp. 444-458.

Vanderveken, D. (1990) "On the Unification of Speech Acts Theory and Formal Semantics," in Cohen, P. R., et al. (Eds.), *Intentions in Communication*, pp. 195-220 pp. 33-70, MIT Press, Cambridge, MA.

Wilks, Y., Huang, X-M., and Fass, D. (1985) "Syntax, Preference and Right-Attachment," *Proceedings of the 9th International Joint Conference on Artificial Intelligence*, pp. 779-784.

Zadrozny, W. and Jensen, K. (1991) "Semantics of Paragraphs," *Computational Linguistics*, 17, pp. 171-210.

Zadrozny, W. and Manaster-Ramer, A. (199?) "The Significance of Constructions," submitted to *Computational Linguistics*.